SUPERFLUIDITY

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

Research and Innovation Action GA 671566

DELIVERABLE I6.2B:

INITIAL DESIGN FOR SLA BASED DEPLOYMENT

Deliverable Type:	Report
Dissemination Level:	Public
Contractual Date of Delivery to the EU:	01/07/2017
Actual Date of Delivery to the EU:	
Workpackage Contributing to the Deliverable:	WP6
Editor(s):	Erez Biton (Nokia IL)
Author(s):	Itai Segall (Nokia IL), Erez Biton (Nokia IL), Yaniv Saar (Nokia IL), George Tsolis (Citrix), Carlos Parada (Altice Labs), Isabel Borges (Altice Labs), Francisco Fontes (Altice Labs), Omer Gurewitz (BGU), Mark Shifrin (BGU), Pedro Andres Aranda Gutierrez (Telefónica I+D), Haim Daniel (Red Hat)
Internal Reviewer(s)	Elisa Rojas (Telcaria), Omer Gurewitz (BGU)
Abstract:	This internal deliverable carries report on the design of an SLA based descriptors and APIs that are required for the access-agnostic deployment of the network service as envisioned in SUPERFLUIDITY. The deliverable describes the initial algorithms for the

	deployment and SLA provision.
Keyword List:	Orchestration, Management

INDEX

List of Figures

List of Tables

Glossary

SUPERFLUIDITY DICTIONARY	
TERM	DEFINITION
UE	User Equipment
OSS	Operation Support System
VIM	Virtual Infrastructure Management
VM	Virtual Machine
MANO	Management And Orchestration
NFV	Network Function Virtualization
KPI	Key Platform Indicator

Table 1: SUPERFLUIDITY Dictionary.

Introduction

1.1 Deliverable Rationale

This internal deliverable will report on the design of the SLA based descriptors and APIs required for the access-agnostic deployment of the network service, as envisioned in the objectives of SUPERFLUIDITY. The deliverable should further describe the initial algorithms for the deployment and SLA provision.

1.2 Quality Review

Review Team member responsible of the deliverable: _____

VERSION CONTROL TABLE			
VERSION N.	PURPOSE/CHANGES	AUTHOR	DATE
1	I6.2 draft		

1.3 Executive summary

1.3.1 Deliverable description

In order to realize the vision developed in the SUPERFLUIDITY project, there is a need to develop an effective management and orchestration system. Applications and service chains deployed in this system will declare their Quality of Service (QoS) requirements, and the system will be able to gather these requirements, compare them against the existing resources, and their current and planned utilization, and take placement decisions to facilitate these requirements. These decisions consist of initial placement and allocation, as well as ongoing migration as required.

The orchestrator system need to take into account the heterogeneity of the execution environments. For this reason, we have extended the descriptor models contained in the ETSI NFV documents released so far. In particular, we have focused on the issue of supporting the “nested” orchestration of containers (using kubernetes) and Unikernels (considering ClickOS).

1.3.2 Summary of results

This document describes the steps taken towards SLA-aware management and orchestration.

We considered two main deployment scenarios, namely, (i) at the core network, and (ii) at the edge.

For the core network, we first study two possible service chain placement strategies – one that gathers all components of each chain into the same host, and one that distributes them between different hosts. This is performed both in the presence of hardware acceleration (DPDK), and without it. Following the comprehensive evaluation of the two placement strategies of service chains, we also model the cost of network switching. Given an arbitrary number of service chains, our model accurately predicts the CPU cost of the network switching they require.

Then, we study the scaling decisions, both for virtualized workload deployed in VMs and for containerized workload. For virtualized workload, we formalise the scaling decision and load balancing problems. In particular, we formalise the decision whether to increase the resources allocated to a certain VNF (*scale out* operation) or to release some of these resources (*scale in* operation) based on the current and expected demand from the network service and the required QoS as a Markov Decision Process (MDP). Our formulation also incorporates the load balancing challenge steering the traffic flows to the different VMs and balancing the load between them.

For the containerized workload, we utilize machine learning techniques that minimize the resource utilization while keeping with the application required quality. Here, we demonstrate our approach with a containerized video streaming server.

Then, we studied the resource allocation and placement for MEC applications. In this context, we discuss the management and orchestrator, as well as the resource allocation problem. We also examine service migration for MEC applications.

Finally, we turn to the deployment model and consider the deployment descriptors. Here, we first present NEMO – a language to be used as SLA descriptor, for an application owner to declare its QoS requirements. NEMO is a human-readable language used for network modelling. It is placed by the authors in the scope of Intent-Based Networking (IBN), because it describes *what* the network should do, instead of *how* to do it. NEMO provides basic network descriptors (Node, Link, Flow,


Policy) to define the infrastructure and controller communication commands to interact with the controller.

Then, we propose extensions to the ETSI NFV ISG specification to support nested VDUs using heterogeneous technologies. Specifically, 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.

2 Core Network

2.1 Resource Allocation and Placement

NFV enables flexible and scalable implementation of network services on cloud infrastructure. One of the key factors in the success of NFV is the ability to dynamically allocate physical resources according to the actual demand. Yet, orchestrating service chains that require high traffic throughput is a very complex task and most existing solutions:

1. concentrate on hand crafted tuning of the servers' configuration to get the desire performance (using hardware accelerators such as DPDK or SRIOV); and/or
2. rely on theoretical placement functions that assume the operational cost is part of the input.

Since many network functions deal with data plane, the cost associated with virtual switching is a crucial factor of the overall operational cost. The amount of resources required to support virtual switching depends on the way service chains are deployed (implying intra- or inter-servers) and the amount of network traffic that goes through the service chains.

As a result, service chain placement in such environment is mostly static and operators lose one of the main attractive features of NFV -- the ability to dynamically allocate resources according to the current need. Such a dynamic mechanism would allow a much more efficient utilization of resources, since the same physical resource can be used by different NFs when needed. Thus, achieving both high performance and agility, by being able to dynamically change the resource allocation of service chains as needed, is a great challenge.

Identifying deployment mechanisms that minimize the provisioning cost of service chains has recently received significant attention from both academia and industry. However, existing studies neglected the operational cost of real NFV deployments. Therefor typical models (for example in

NFV orchestrators) might either lead to infeasible solutions (e.g., in terms of CPU requirements) or suffer high penalties on the expected performance.

In an attempt to address this gap, we have recently focused [1] on evaluating and modeling the virtual switching cost in NFV-based infrastructure. Virtual switching is an essential building block that enables flexible communication between VNFs but it also comes with an extra cost in terms of computing resources that are allocated specifically to software switching in order to steer the traffic through running services (in addition to computing resources required by VNFs). This cost depends primarily on the way the VNFs are internally chained, packet processing requirements, and accelerating technologies (e.g., packet acceleration such as Intel DPDK).

2.1.1 Motivating Example

Figure 1 illustrates a possible deployment of four service chains on three identical physical servers (A, B and C). As one can see, service chain chains φ^1 is composed of three VNFs – φ^1=< $\varphi^1{}_1$, $\varphi^1{}_2$, $\varphi^1{}_3$>, φ^2 is composed of four VNFs, φ^3 is composed of five, and φ^4 is composed of two VNFs.

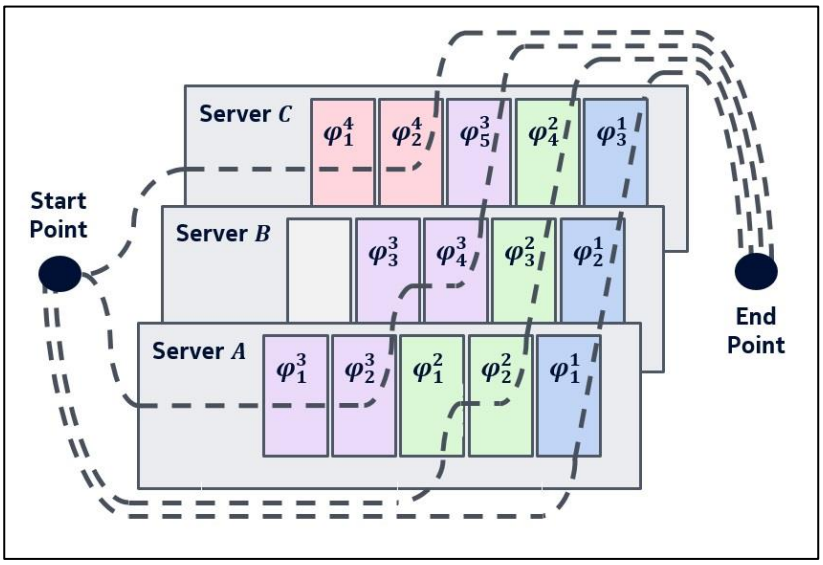

Figure 1: Example of possible deployment of four service chains on top of three physical NFV servers.

For simplicity, we assume that all traffic steering is done by servers' internal virtual switching, all VNFs require the same amount of processing power to perform their task, and that each of the service chains is associated with a known amount of traffic it has to process. In the depicted deployment, servers A and C have the same number of deployed VNFs and thus they have the same computation resource requirement for processing. However, determining the amount of processing resources (CPU) needed for switching inside the server is far from being straightforward.

For example, consider the deployment presented in Figure 2, where we have the example same resources, and the exact same requests. It is not straightforward to deduce which type of deployment consumes more resources to operate.

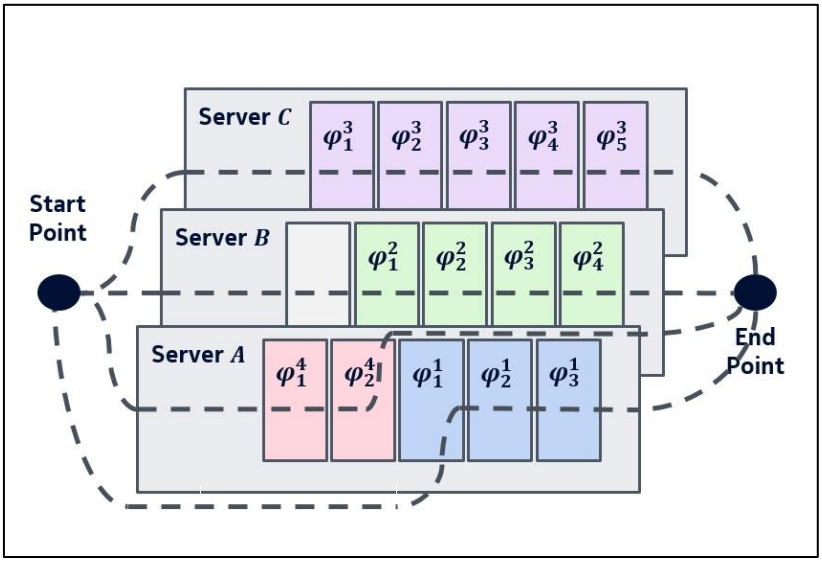

Figure 2: Yet another example of possible deployment of four service chains on top of three physical NFV servers.

In fact, in some cases the deployment can be infeasible due to lack of sufficient computing resources for the switching task. As was shown in [1], the amount of computing resources needed for the internal switching depends on the structure of the chaining in the server, and the amount of traffic associated with each chain.

2.1.2 Our Contribution

We continue this line of work and develop a general service chain deployment mechanism that considers both the actual performance of the service chains as well as the needed extra internal switching resource. This is done by decomposing service chains into sub-chains and deploying each such sub-chain on a (possibly different) physical server, in a way that minimizes the total switching overhead cost. Of course, there are exponentially many ways to map the sub-chains to serves. We introduce a novel algorithm based on an extension of the well-known reduction from weight matching to min-cost flow problem, and show that it gives an almost optimal solution, with much more efficient run time comparing to the exhaustive search.

We evaluate the performance of this algorithm against the fully distribute or fully gather solutions which are very similar to the placement of the standard mechanism commonly utilized on cloud schedulers (e.g., *nova-scheduler* module in OpenStack with load balancing or energy conserving weights) and show that our algorithm significantly outperforms OpenStack (can be up to a factor of 4 in some cases) with respect to operational costs.

Our main contributions in this work are:

- **NFV deployment model.** We extended the switching cost computation of [1] from considering only the gather and distribute placements to a general deployment scheme that can compute the switching related CPU cost for arbitrary deployments.
- **Optimal deployment mechanism**. We develop an online algorithm that minimizes the switching cost for online service chain requests.
- **Performance evaluations**. We provide performance evaluations which indicates that using this algorithm can significantly increase utilization by allowing more network functions to run on the same NFV infrastructure in a more efficient way.

2.2Optimal scaling and load balancing based on MDP models

2.3 Introduction

We address the problem in which the enterprise has to find dynamic policy of VM deployment, displacement and scheduling of incoming VNF tasks, as a function of the set of costs and the number of VNF instances currently being served in the already active VMs.

An enterprise, which decides to virtualise any of their network services needs first to deploy ("build") the corresponding service, in order to be able to run VNF instances.

The deployment process involves having leased a VM and loading on it the corresponding software.

Once deployed, the enterprise has to pay per time of usage for the leased VMs, regardless of the load.

The process of deployment can take time, and an additional deployment cost. Hence, having idle resources is undesired. On the other hand, the delay involved with the deployment or lack of space for the new tasks can cause some VNF instances to be rejected from running, thus inflicting a profit loss to the enterprise. Note that in some cases, the displacement ("destroy") operation, i.e., the process of releasing VMs can also incur a cost.

An additional basic demand is to facilitate scaling. For simplicity, consider an enterprise which needs to run networking tasks of only one type. Hence, we assume all VMs are similar and able to run similar VNF tasks (e.g., flows a firewall handles). The total number of VMs allocated is dynamically increased (via a scale out operation) or decreased (scale in operation) according to the demand from this network service. In scale out/in operations, we increase/decrease (i.e., deploy/displace) the number of VMs that are allocated, respectively. Note that we do not consider the scale up and down operation, which are less common in NFV use cases. (In scale up/down operation we can, for example, add/remove CPU cores to a given VM).


Clearly, increasing the number of VMs would disperse the total load and improve the performance of each VM, yet would imply having leased more costly resources. Accordingly, our scaling decision should minimize the number of leased VMs, while keeping with the application required SLA.

Hence, the decision whether to scale out or in remains an important problem that needs to be addressed, especially in dynamic scenarios.

In conjunction with the scaling decisions, we are also facing a load balancing challenge, steering the traffic flows to the different VMs and balancing the load between them. In this study, we tackle both the scaling decision and the load balancing strategy as a single problem.

Going back to the simple single-VNF enterprise scenario, the load balancing will merely amount to having equally loaded VMs. Hence the trade-off in this case means the average load on a VM against the number of deployed VMs. Yet, having in mind deployment time and cost, this trade-off still represents a significant challenge as the optimal policy derivation is not straightforward.

To this end, we model the problem by queuing system with a dynamic (but limited) number of queues; each queue stands for a VM running VNF instances. We assume that VNF tasks arrive with constant average rate. Upon each arrival event, the Decision Maker (DM) decides to which VM the arriving task should be scheduled, or whether it should be rejected. The VM deployment decisions are made at arrival times, as long as the limit of active VMs is not reached.

At departure from a queue (i.e., running of VNF task ends) which has been left empty, DM decides whether to keep that queue alive or to destroy it.

In order to reflect the load at busy VMs we introduce a delay cost which (not necessarily linearly) increases with the load.

The maximal number of running task on a single VM is limited by borderline number which indicates a performance fall beyond the SLA demands and, hence, should never be exceeded.

Deployment and displacement operations of VM consume additional fixed costs. Once a VM was deployed, the enterprise is charged with fixed per-unit of time reservation and maintenance cost. We term this keep-alive or the holding cost.

The DM aims to find a policy which will maximize the total income in the long run.

While the set of costs described above implies no trivial policy could exist, some of the parameters have contradicting impacts which may dictate certain properties of the policy.

For example, in the case where the delay cost function sharply increases with a load, the DM will attempt to deploy as many VMs as possible.

On the other hand, in the case where keep-alive cost is comparatively high, the DM may prioritize minimization of the active VMs number.

A distinctive impact has a VM deployment time, which we separately explore.

In this work, we treat the dynamic VM deployment and VNF tasks scheduling problem by introducing a control model based on MDP formulation. The solution of the MDP provides the optimal policy.

Once applied, the policy potentially reveals the average number of active VMs and average number of tasks in the task queue of each VM.

This allows the enterprise to assess the demands and to plan ahead.

Moreover, the solution to the MDP is expressed by value function which indicates what are the costs and revenues the enterprise will receive in the long run, for a given scenario along with its set of parameters.

Complete details of this work are available at: [2] . In the following we present some of the numerical results of our work.

2.3.1 Numerical results

In the following, we present simulation results which both show additional threshold properties of optimal policies, and provide a comprehensive study of the value functions and the corresponding policies. We explored a system with five identical queues.

Figure 3 demonstrates the impact of the keep-alive queue cost. The simulation was run with negligible delay cost, rejecting fine equal to 10, $\lambda = 4$ (VNF tasks arrival rate) and $\mu_i = 1, \forall i$ (servers total processing rate). Buffer limit was 4 tasks. One sees that there is a small region of keep-alive queue cost, where the number of active queues declines. The trade-off in this simulation is the accumulatively paid fine against the accumulatively paid keep-alive queue cost. As long as the queues are identical, there is no preference which queue should be kept idle. That is why the number of active queues declines to zero within a very small interval of keep-alive queue cost. Hence, once it is more affordable to pay the fines by rejecting all incoming tasks rather than maintaining the VMs (i.e., the queues) the queues stay idle at all times.

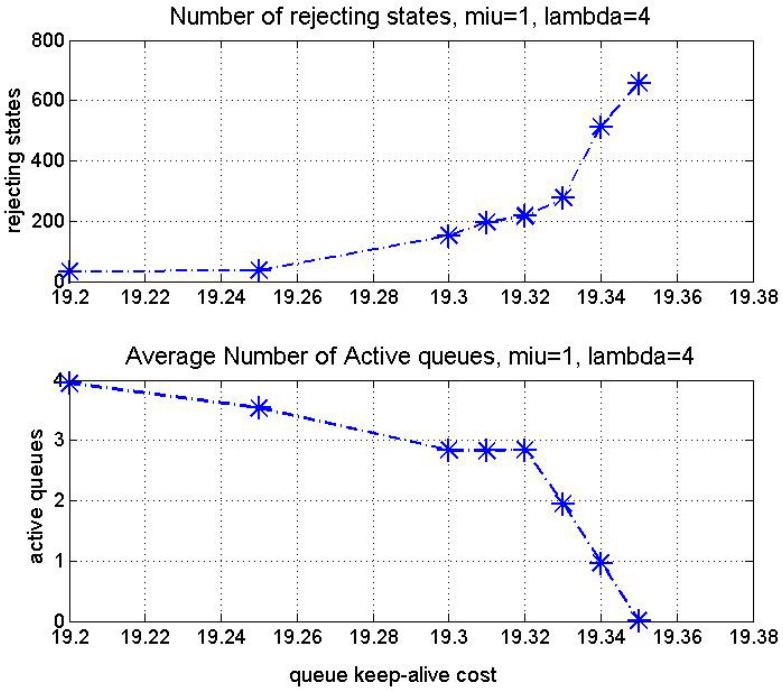

Figure 3: Impact of the allocated VNF cost

Figure 4 shows simulation of the delay cost h, while the keep-alive cost was fixed equal to 1, the buffer size of each queue was 6. The arrival intensity was 4.75 and $\mu_i = 1, \forall i$. Rejecting fine was equal to 10. The delay constants were $\eta_1 = 1, \eta_2 = 1.8, \eta_3 = 2.5, \eta_4 = 3.5, \eta_5 = 4.5, \eta_6 = 5.5$. By the cost model, one expects that the tasks will be balanced in the queues. Observe that in this simulation, the keep-alive cost was low enough to allow that. Indeed, all queues were active while the total average number of tasks declined with the delay cost. Observe that in this simulation, the interval of the varying cost was significantly larger. This stems primarily from the fact that η_1 is small.

Note that the lower graph in both simulations shows the number of rejecting states, out of the states-space.

Additional thresholds can be seen in "build" and "destroy" actions. For example, if The building cost β and/or the destroy cost ψ are high if compared to keep-alive cost, the optimal policy acts to leave all queues active, even if empty.

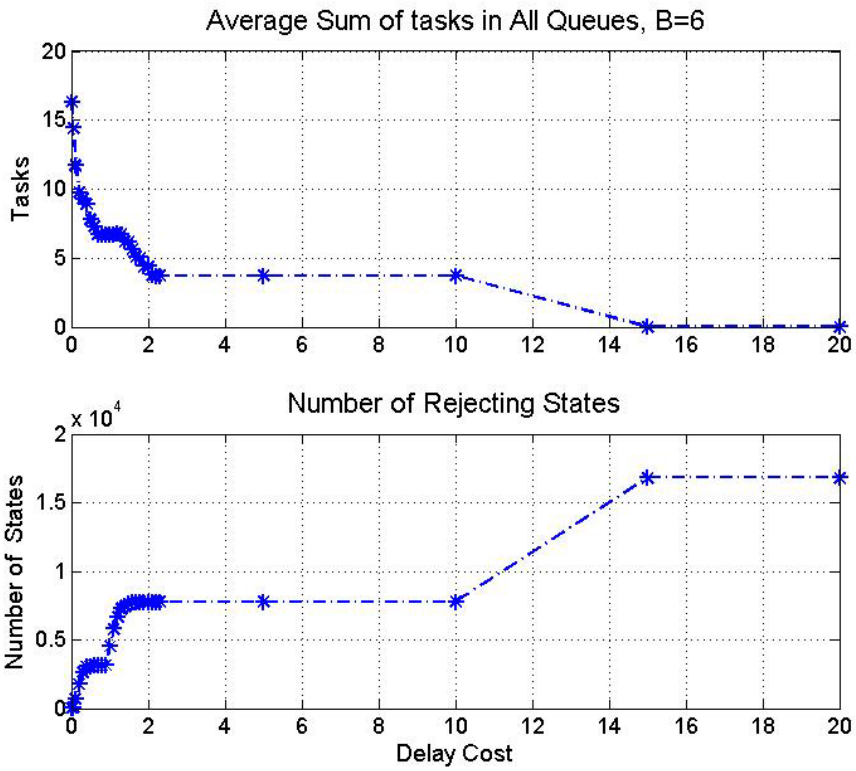

Figure 4: Impact of delay cost

To conclude, the set of system parameters will determine if the optimal policy will prioritize balanced and mostly active queues or a small number of active and comparatively busy queues. The values of β and ψ will determine if keeping alive the empty queues is economically reasonable.

2.4Machine learning techniques for the LCM for containerized workload

2.4.1 Introduction

To allow multi-tenancy of containerized applications, typically, each tenant/application is allocated VMs as the underlay infrastructure. Then, the containerized application scales out and in within that underlay. Accordingly, we are now facing two scaling processes, that is, (i) of the underlay VMs, and (ii) of the containerized applications.

For the first process, namely the VM Scaling, we have our MDP model (see section 2.4) that can be extended to tackle containerized workload. Such extension is pending investigation in the third year.

In this work, we focus on the second process of the scaling of the containers. Here we devise through machine learning techniques, a mechanism that automatically scales the containerized application based on infrastructure metrics. We further demonstrate that our mechanism

dramatically outperforms Kubernetes. Specifically, we demonstrated that a containerized video streaming application suffers from high delays (due to lack of resources) while being managed by Kubernetes. On the other hand, our mechanism manages to scale the application in a timely manner while providing the appropriate resources to obtain the required application QoS (i.e., packet latency).

2.4.1 Kubernetes scaling mechanism

Typically, the LCM operation of containerized applications is handled by the Containers Orchestration Engine (COE). Since Kubernetes is the most dominant COE of today, in this work we adopted it as our COE of choice. Indeed, Kubernetes manages the scaling operation. However, in this work, we demonstrated that Kubernetes suffers from severe drawbacks. Specifically, Kubernetes v1.57 triggers scaling only based on CPU utilization or based on application metrics (via APIs). Indeed, this was handled in Kubernetes v1.6, where other infrastructure metrics may trigger scaling. Yet a function that considers all infrastructure metrics is not defined and it is not clear how to combine those metrics to trigger scaling operation.

Accordingly, in the project's second year we tackle this issue by applying machine learning techniques to control the scaling decisions.

2.4.2 Machine learning based scaling

Our goal is to devise a machine learning mechanism that receives as an input samples of the resource utilizations by each container and predicts application performance.

As a learning sequence, we assume that our system receives indications on the application performance. Based on this learning metric, we aim at devising a mechanism that learns the optimal scaling policy based on infrastructure metrics that minimize the consumed resources (minimize the number of deployed containers) while maintaining the application's required performance.

2.4.3 Offline implementation – video streaming application

Aiming at devising the optimal online machine learning scheme, we first analyse and solve this problem in an offline scenario.

To that end, we consider a containerized video streaming server. Specifically, we containerized this server especially for this work (available at Docker Hub under ruvenmil/mycont2). Next, we ran this server under different deployment configurations, i.e., with multiple number of clients as well as with multiple numbers of deployed pods (Kubernete's minimal deployment unit). For each configuration, we recorded the packet delay. Note that for video streaming the packet delay constitutes a representative indication for the video quality. Our setup topology is given in Figure 5.

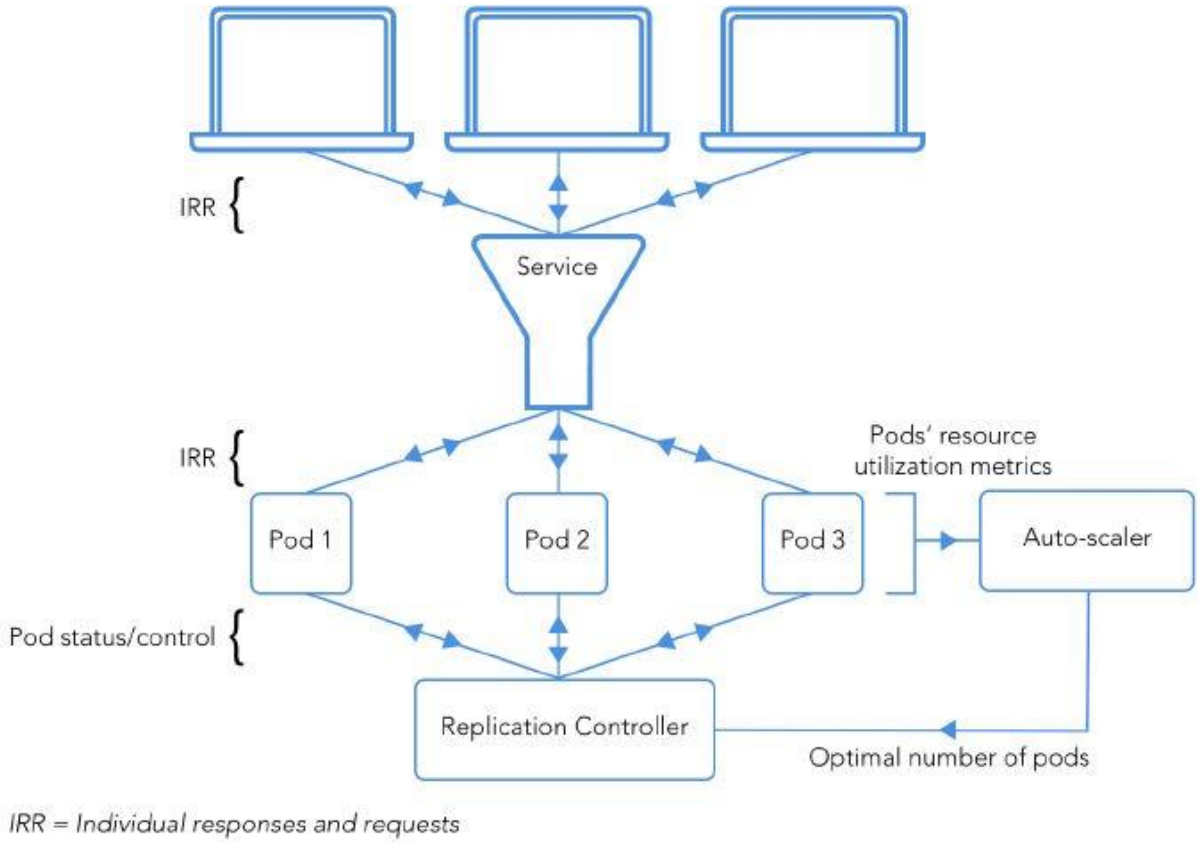

Figure 5: setup topology

In addition to the packet delays, each record also includes the infrastructure utilization, namely, CPU, memory and bandwidth consumption.

With all data at hand, we employed several prediction mechanisms, including: random forest and gradient boosting, and obtained a predictor for the video quality based on the infrastructure resource consumption. Next, we triggered the scaling operation based on this predictive function. Utilizing that function obtained a much better video quality and initiated scaling just on time.

2.4.4 Implementation results

In the following we compare our auto-scale algorithm with the default scheme of kubernetes.

Packet error measurement:

We can see in Figure 6 that the suggested algorithm has better performance and less error burst (depicted in dark blue).

Kubernetes: suggesred algoeythem

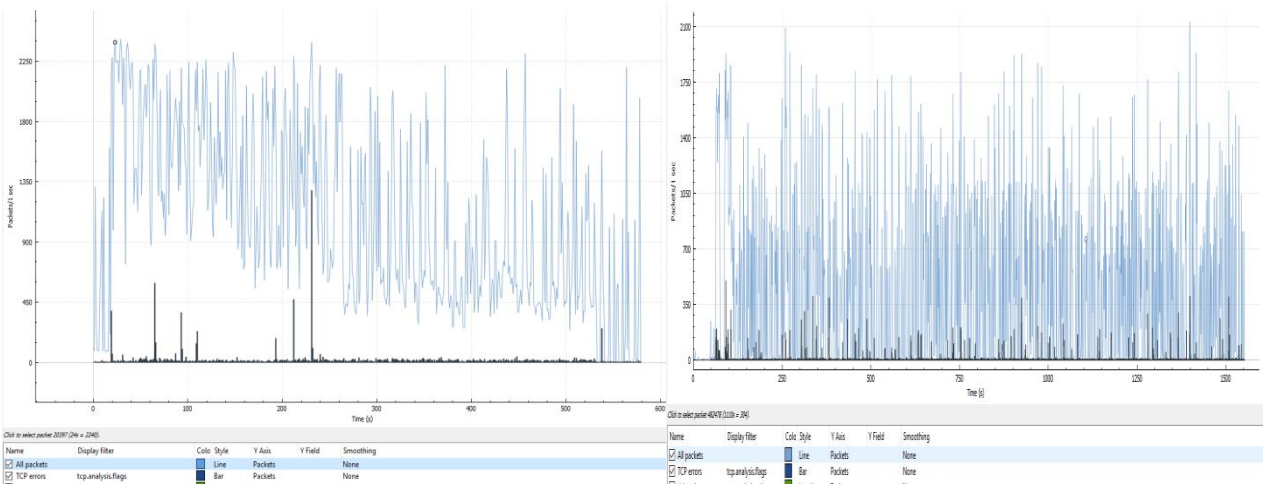

Figure 6: packet errors

Throughput measurement:

Figure 7 depicts the throughput measurement results. One can see that the achieved throughput is higher with our suggested algorithm compared to with the of the shelf Kubernetes mechanism.

Kubernetes: suggested algorythem

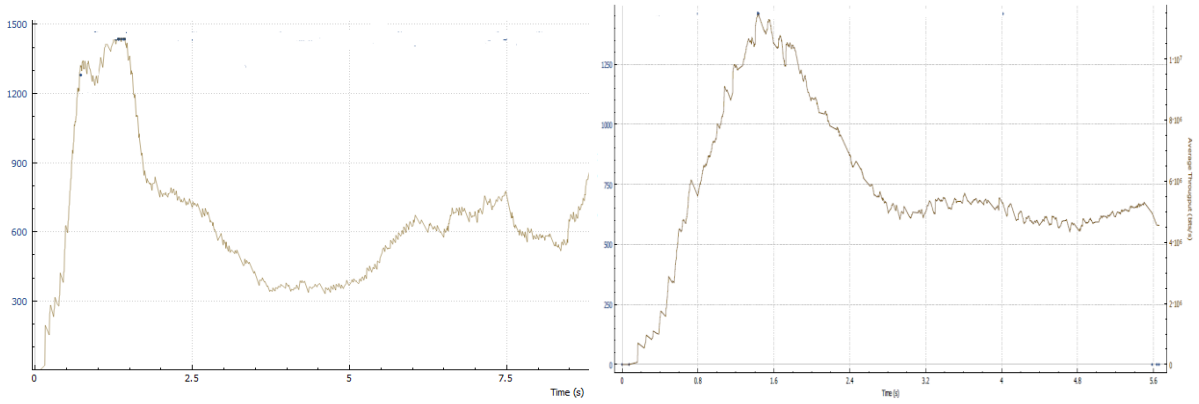

Figure 7: throughput measurements

2.5LBaaS

2.5.1 Load Balancing principles

Efficient load balancing is an area of paramount importance for identifying possible flaws and limitations of the SUPERFLUIDITY architectural components, most of which are currently under heavy development, undergoing continuous modifications to improve performance.

The actual definition of load balancing, as the distribution of network or application traffic across a cluster of servers, consequently leading to improved responsiveness and increased service availability, was given in Deliverable I5.3. In addition, Deliverable D2.1 identified the load balancing-dependent use cases, where specific performance must be established to fulfil the pre-defined scalability and high availability requirements, while the specific requirements of Load Balancer as a functional block were further analyzed in Deliverable D2.2. Last but not least, an additional analysis of certain load balancing approaches can also be found in Internal deliverable I6.1b. With load balancing references being omnipresent in the majority of technical deliverables, one may identify the significance of this functionality for the SUPERFLUIDITY platform as a whole.

2.5.2 HA Proxy and LBaaS in OpenStack

The project's virtual infrastructure manager (VIM) of choice, OpenStack, offers certain open-source options for implementing the infrastructure load balancing capability, originally through HAProxy, a single-threaded, event-driven, non-blocking engine combining a very fast I/O layer with a priority-based scheduler, currently integrated from an upstream open-source project. OpenStack fully supports HAProxy deployment in its controller nodes where each instance of the software configures its front end to accept connections only to the virtual IP (VIP) address, while the backend is a list of all available IP addresses that may need load balancing of their ingress traffic. However, the most effective load balancing service of OpenStack, currently integrated inside Neutron is no other than Load Balancing as a Service (LBaaS), which allows both proprietary and open-source load balancing technologies to handle the excessive traffic load. As stated in [2] , LBaaS builds on top of HAProxy by leveraging agents that control HAProxy configuration and manage the HAProxy daemon, is therefore somehow considered as an enabling module that introduces additional functionality to the core load balancing mechanism of OpenStack. OpenStack is also introducing carrier-grade load balancing backend mechanism through Octavia [4] .

2.5.3 Citrix Netscaler ADC and MAS

The increased demands of contemporary networking environments in terms of traffic, necessitate the evaluation of the proposed SUPERFLUIDITY architecture paired with commercial, carrier-grade

load balancing options. Citrix NetScaler ADC [5] is an application delivery controller that provides flexible delivery services for traditional, containerized and microservice applications from any cloud or private datacenter, and was evaluated as part of SUPERFLUIDITY. As described in Internal deliverable I6.1b, NetScaler ADC can be integrated into the NFV architecture, working in parallel with all existing MANO entities, such as VIM and the SDN Controller. This integration is also presented in Figure 8.

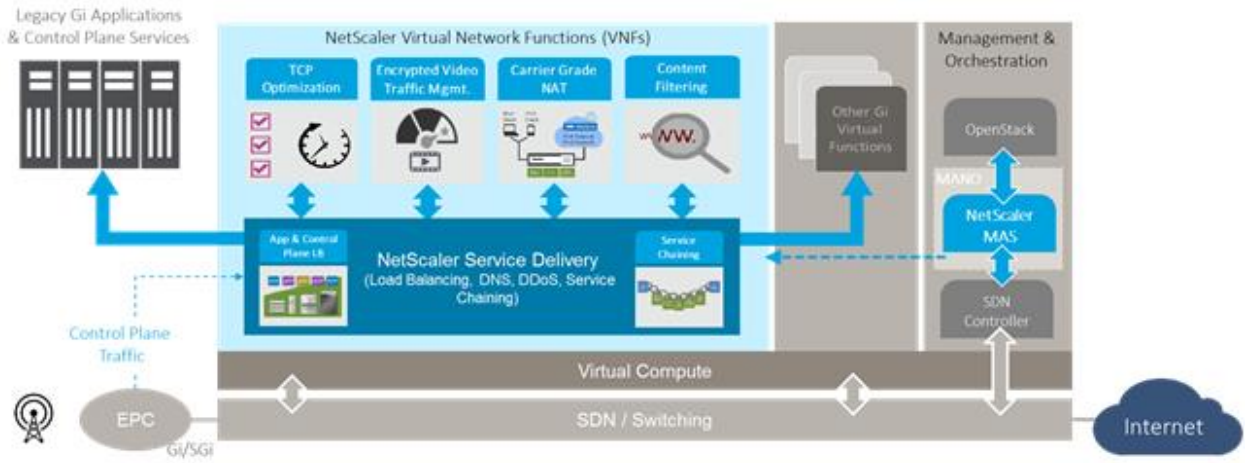

Figure 8: Netscaler at NFV Architecture [6]

NetScaler ADC is operated through a dedicated Element Manager, namely the NetScaler Management and Analytics System (MAS) [7] NetScaler MAS integrates using standard APIs with OpenStack, translating necessary messages to the RESTful APIs supported by NetScaler ADC. It facilitates administrators to monitor, automate and manage network services for scale-out application architectures with ease, and provides application-level integration with external orchestration systems [7] .

Integrating MAS with OpenStack

The overall evaluation process conducted as part of SUPERFLUIDITY, involved a direct comparison of the capabilities of open-source load balancing options already integrated in vanilla versions of OpenStack (HAProxy) against NetScaler ADC. This process also required deploying and validating NetScaler ADC and MAS in a dedicated NFV Lab, thus verifying that certain enhancements introduced in these products as well as the NetScaler LBaaS driver [8] [9] are working properly.

As stated in previous paragraphs, for being able to use NetScaler ADC instead of the integrated load-balancing solutions of OpenStack, certain modifications are needed in Neutron LBaaS module. In particular, Citrix engineers developed an LBaaS plugin which can be deployed in Neutron and

implements all the LBaaS driver CRUD APIs for operating on OpenStack VIPs, Pools, Pool Members and Health Monitor entities. The integration consists of a driver class configured in the Neutron config file (neutron.conf), and the accompanying unit tests, while its abstract functionality is shown in Figure 9:

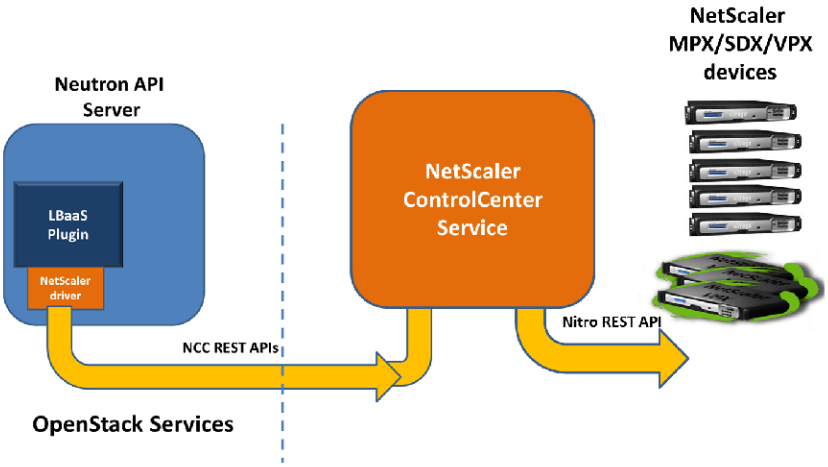

Figure 9: NetScaler LBaaS Integration [8]

The NetScaler LBaaS integration consists of a driver class that implements the Neutron LBaaS driver which calls the NetScaler Control Center (NCC) service using NCC REST APIs. NCC is a separate service that runs outside of OpenStack infrastructure, and is deployed as a "virtual appliance" on supported hypervisor platforms (KVM/ESX/XenServer) for ease of setup. Since Release 11.1, NCC is integrated in NetScaler MAS, which is now in charge with all NetScaler resource and tenancy management tasks, as well as NetScaler device configuration, allowing the driver component in OpenStack to remain lightweight, simpler to maintain and easier to evolve going forward as the Neutron service evolves. During SUPERFLUIDITY validation process, a full-scale NetScaler MAS node was deployed and integrated with the existing OpenStack Platform, as follows.

OpenStack Neutron LBaaS plugin includes a NetScaler driver that enables OpenStack to communicate with the NetScaler MAS. OpenStack uses this driver to forward any load balancing configuration done through LBaaS APIs, to the NetScaler MAS, which creates the load balancer configuration on the desired NetScaler instances. OpenStack also uses the driver to call NetScaler MAS at regular intervals to retrieve the status of different entities (such as VIPs and Pools) of all load balancing configurations from the NetScaler ADCs. NetScaler driver software for OpenStack platform is bundled along with the NetScaler MAS. To download and install the drivers, it is necessary to first install NetScaler MAS and launch the application.

2.5.4 NetScaler MAS Installation

For consistency reasons NetScaler MAS was installed on a Linux KVM server, after verifying that all hardware virtualization extensions and ***virsh***, a command line tool for managing virtual machines, were available. As described in [10] after obtaining the necessary image files the installation process includes, navigating to the folder where the compressed file is saved, using the tar command to untar the NetScaler MAS image, verify that a domain disk image (.qcow2 file format), a domain XML file and the necessary checksum file were present, editing the XML file for specifying the necessary networking attributes, using ***virsh*** to define the VM attributes through the recently added XML file and initiating NetScaler MAS by entering the following command:

```
virsh start [<DomainName> | <DomainUUID>]
```

Certain configuration steps for the NetScaler MAS were also needed, after the previous process is concluded and the service became available. In particular it was necessary to:

- Login to the NetScaler MAS node
- Type `shell > networkconfig` to configure the management IP address
- Complete the initial network configuration by adding information regarding Netmask and GW IP
- Execute the deployment script by typing the following command in the shell prompt # `deployment_type.py`
- Restart the server and login to the GUI

2.5.5 NetScaler Driver Software Installation

It is possible to install NetScaler driver on OpenStack via NetScaler MAS GUI. After logging in, click Downloads and download the latest NetScaler bundle .tar file to a temporary directory of the OpenStack Controller. The bundle includes LBaaS V2 drivers for Openstack Liberty/Mitaka/Newton releases along with the Heat plug-ing. Extract the files from the NetScaler driver tar, navigate in to the OpenStack <Release Name> folder and execute the following command to install the driver and specify the NetScaler MAS IP address:

```
./install.sh --ip=<NetScaler_MAS_IP> --password=<password> --protocol=<protocol> --neutron-lbaas-path <neutron-lbaas-directory-path>
```

2.5.6 Registering OpenStack with NetScaler MAS

OpenStack information needs to be registered on the NetScaler MAS. The process includes specifying the OpenStack controller IP address and cloud administrative user credentials, and also the OpenStack NetScaler driver user credentials. It is also possible to later specify the same login

credentials in the NetScaler_driver section of the Neutron configuration file (neutron.conf) so that NetScaler driver in OpenStack can connect to NetScaler MAS during LB configurations.

After OpenStack and NetScaler MAS are registered with each other, both can talk to each other. Also, OpenStack users can use their existing credentials in OpenStack to log on to the NetScaler MAS user interface to check how their LB configurations are performing in NetScaler [11] .

2.5.7 Adding OpenStack Tenants in NetScaler MAS

Provided that OpenStack and NetScaler MAS are now interconnected through the previous registration process, it is possible to create a Tenant in OpenStack using the NetScaler MAS. Simply

- Navigate to **Orchestration** > **Cloud Orchestration** > **OpenStack** > **OpenStack Tenants**, and then click **Add**.
- In **Add OpenStack Tenants** page, click **+Add**, and then select the OpenStack tenant.
- Click **OK**.

2.5.8 Provisioning NetScaler VPX instance in OpenStack

Download the required NetScaler instance image from the Citrix download page, and upload it on Glance, the OpenStack Imaging service. Having an image available on Glance allows you to configure a NetScaler instance on-demand when assigning the instance to the tenant.

To auto-provision the NetScaler VPX devices on OpenStack

1. In NetScaler MAS, navigate to **Orchestration** > **Cloud Orchestration** > **OpenStack**.
2. Click **Deployment Settings**.
3. Set the necessary parameters:
    - Management Network
    - Profile Name
    - Licences
    - NetScaler VPX image in Glance
    - Proxy settings

The overall process of NetScaler MAS integration with OpenStack Workflow is shown in Figure 10.

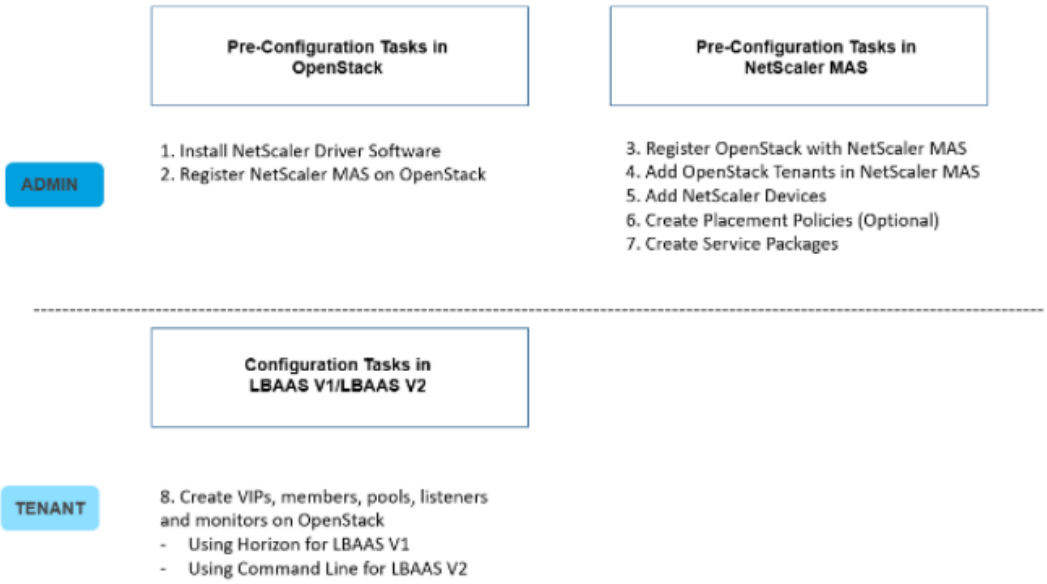

Figure 10: NetScaler MAS - OpenStack Integration Workflow [11]

The above outcomes will be integrated with the SUPERFLUIDITY platform as part of WP7 activities.

3 Edge Network

3.1Mobile Edge Computing

3.1.1 Resources Allocation

In a Mobile Edge Computing (MEC) System, applications will run at ME (Mobile Edge) Hosts in a virtualized environment, as VDUs (*Virtualization Deployment Units*), e.g. VM or Container. For that purpose, and in line with ETSI NFV, management and orchestration components have been added to the MEC system, composed by “Mobile Edge Host Level” and “Mobile Edge System Level”. That is, while the latter has a global view of the entire MEC System, including all ME Hosts, the former acts at the ME Hosts level, with the functions: Element Manager for the ME Host, Rules and Requirements management for MEC applications, and MEC Applications LCM operations (similar to VNFM for NFV). At each ME Host, the ME Platform component may or may not be deployed over the virtualization infrastructure, possibly using specific HW and SW. Figure 11 depicts the full ETSI MEC architecture.

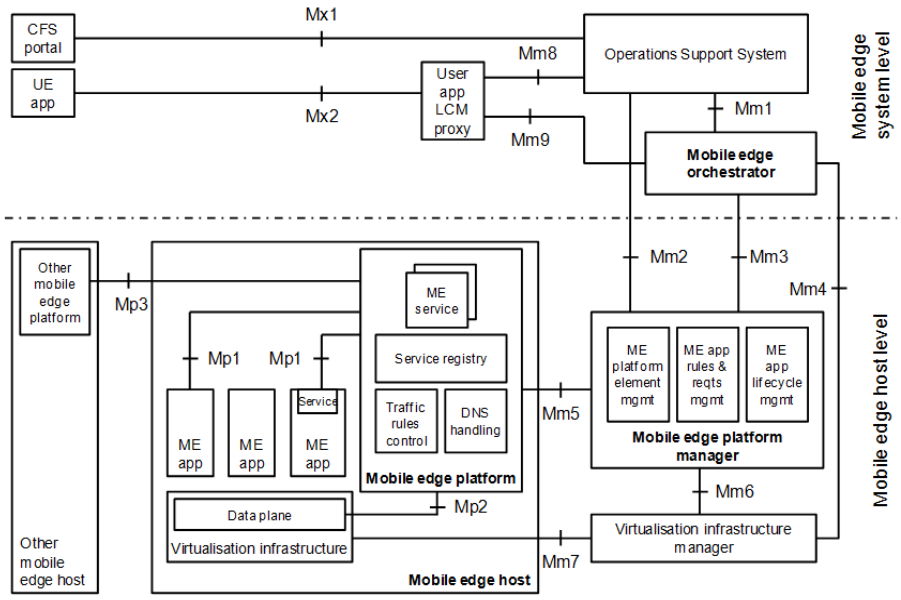

Figure 11 - ETSI MEC Architecture [ETSI-MEC-Arch.]

In this context, resources allocation to MEC Applications will occur on the virtualization infrastructure, existing at each ME Host. This will be controlled by the ME Orchestrator, as part of ME Applications Life-Cycle Management (LCM) operations. SUPERFLUIDITY Deliverable I6.1 "Initial Design for Control Framework" documents MEC management and orchestration requirements as well as respective flows.

The following entities, hosted at the MEO, are required for ME Applications resources allocation:

- MEC App Catalogue
  - Stores the catalogue of deployable MEC Applications, with associated images and MEC Descriptor (where applicable rules and requirements are specified)
- MEC App Descriptor
  - Even if not yet defined by the ETSI MEC ISG, MEC Applications will have a descriptor associated to them. This will be in line with the VNF Descriptor, including specific information like, delay budget, required MEC Services to run and required virtualization resources.
- MEC Infrastructure
  - Keeps track of available, reserved and in use resources at all ME Hosts, for usage by MEC Applications
- MEC Hosts Inventory
  - Lists and describes the ME Hosts under MEO control. This shall include, per ME Host, a mapping with served eNB and available services (LOC, RNIS, DNS, etc.)

MEC Apps instantiation may occur:

1. Triggered by MEO


- Selection of ME Hosts to deploy MEC Apps shall be automatic and based on existing information at the MEO and associated MEC Applications' Descriptors. Thus, from no need for allocate resources (only on-boarding), to allocate resources at all ME Hosts, all scenarios are possible.

2. Triggered by management (OSS)
    - Selection of ME Hosts for Applications instantiation will be the decided by a third-party entity, eventually including the analysis of the information contained in MEC Application Descriptors.
3. Triggered by UE
    - Via the MEC "User App LCM Proxy" component, entities at the UE may request the instantiation of specific MEC Apps. Application requirements (e.g. latency, compute resources, storage resources, location, network capability, security condition etc.) will be analysed in order to select a host fulfilling all the requirements.

In any of the previous situations, upon identification of the ME Hosts for Application deployment, resources at the corresponding virtualized infrastructures, must be reserved, involving the defined ME Platform Manager and the VIM, in a similar way to what happens with VNFs.

Even if current ETSI ISG MEC work does not address MEC Applications scale in/out or up/down, there is no reason to exclude that as part of LCM operations, in the context of resources allocation. In the same sense, ETSI ISG MEC considers that individual MEC Applications will be made of single VDUs. Similarly to VNF, there is no strong reason for not considering that MEC Applications may be the result of the composition of several VMs.

Resources allocation for MEC Applications will be managed by the interactions between MEO, MEPM and VIM. Flows identified in SUPERFLUIDITY's Deliverable I6.1 shall be followed.

3.1.2 Placement

Placement of MEC Applications, meaning defining at which ME Hosts they will be deployed and run, must be determined by their requirements, especially by the maximum allowed delay to the UEs to be served. Considering that MEC addresses the need to have services provided as close as possible to users, it is expected that MEC Applications will be deployed at the closest ME Host to UEs to serve. However, the existing delay budget and resources availability may decide differently. Additionally, availability of certain ME Services, like RNIS (radio interface status) or LOC (location), will also constrain the MEC Applications placement.

In addition, depending on Applications' provided services and resources availability, Applications may not be immediately moved to all the ME Hosts, as part of the on-boarding procedure. Business and licensing conditions may also determine when and where to deploy and run MEC Applications.

ME Hosts will serve a number of eNBs. This number may vary from one ME Host per eNB, possibly running at the eNB, to several eNB being served by a single ME Host. Considering C-RAN (Cloud-

RAN) deployments, it makes all sense to consider the deployment of ME Hosts, sharing the same cloud infrastructure with RAN centralized components.

ME Orchestration (MEO) must have a topological view of the entire MEC System, in order to determine where to deploy ME Applications to serve specific geographical areas. Thus, a mapping between eNB and ME Hosts is required. This information was referred before as stored in the MEC Hosts Inventory repository.

In addition, ME Application descriptors must include parameters to help MEO to decide where to deploy each ME Application. This must be complemented by Operators' rules and established business relationships, and translated into MEO understandable policies.

MEC Applications' placement will be decided based on Applications and Operators' requirements and constraints, and not UEs, with the exception of instantiations requested by entities running at UEs (e.g. Applications counterparts running at UEs).

Thus, in general, the following placement factors may apply, determining how many MEC Application instances are required and where to deploy those instances:

- Delay

  MEC Application Descriptors shall indicate desirable and maximum admissible delay towards UEs to be served.

- Geographical scope

  Some ME Applications may provide localized services, like Augmented Reality for a specific building or street. The geographical scope will, most likely, be provided in such a way (e.g. reference to a monument or specifying a geographical area) that will need to be mapped to eNBs, and from that to ME Hosts, by the MEO.

- Required resources

  Besides specifying computational and networking resources to run, MEC Applications may require specific hardware to run, like video processing.

- Available resources

  Considering that all on-boarded ME Applications cannot be deployed and run at all ME Hosts, their instantiation may depend on the availability and establishment of priorities to access ME Hosts' resources.

- Required MEC Services

  MEC Applications may need to access local MEC Services to run, for instance, with RNIS or LOC.

- Licensing and agreements with service provider

  Business agreements with ME Application providers may also determine where and how many instances to deploy. The limitations may be in any of the parties (Operator only allowed or allowing to simultaneously run a certain number of Application instances).

- UEs' location

This is a specific scenario that will apply to Applications' instantiation requests generated by the UEs (via the "User App LCM Proxy", e.g. for Application computation off-loading). Besides any of the abovementioned factors, this specific one will be determinant in the placement of the requested MEC Application instance.

MEO shall handle the identification and evaluation of the parameters that apply. This will happen after MEC Applications on-boarding and whenever the MEC System topology change, for instance, by the addition or removal of MEC Hosts or the reorganization of the mapping between eNB and ME Hosts. It will also happen whenever a UE or the OSS requests the instantiation of some MEC Application. No algorithm or parameters evaluation process is proposed for the moment.

3.1.3 Service Migration & Mobility

3.1.3.1 Mobility in MEC scope

In MEC context, service migration need, or relocation, may happen as a result of UEs' mobility, and applies to MEC Applications. While moving, most likely UEs will attach to different eNB. Depending on the network topology, the same or different MEC Hosts may serve those eNB. While moving between eNB served by the same MEC Host, no relocation shall happen, as UE's mobility will not be noticed by the MEC System. For the other situations, the frequency and the type of relocation to be executed depend on the Application type and mode of operation:

- Generic Applications (not tied to any UE in particular)
  - Stateless applications: no need for any relocation action
    - Application instance is already working at the destination ME Hosts: Service is provided at the edge
    - Application instance is not working at the destination ME Hosts: Service is not provided at the edge
  - Stateful applications: state may be is need
    - Scenario detailed below
- UE specific Applications
  - Service needs to follow the UE, independently from being stateless or stateful, in order to keep proximity, according to the specified requirements

(Generic) MEC Applications will be deployed at certain ME Hosts, as described above. They will have traffic rules associated to them, instructing the ME Host Data Plane on how to identify and handle traffic of interest, to be delivered to each MEC Application. These rules will be defined, in general, based in destination FQDN or IP/Ports. It means that, in general, Applications will be deployed and make their services available in anticipation and independently of the UE which will request them. Thus, MEC Applications' provided services are accessible at the specific ME Hosts on which the MEC System decided to deploy those MEC Applications. Therefore, UE's mobility shall not trigger, in


general, MEC Application relocations, except for the ones instantiated under UE's request. However, application state created and associated to the UEs may need to be relocated.

MEC Applications mobility is mentioned in current ETSI MEC ISG documents and its discussion triggered the creation of an Work Item (WI), to be developed till end of MEC Phase 1 (end of 2016). For instance, [ETSI GS MEC 002, Mobile-Edge Computing (MEC); Technical Requirements] states that:

"Other mobile edge applications, notably in the category 'consumer-oriented services', are specifically related to the user activity. Either the whole application is specific to the user, or at least it needs to maintain some application-specific user-related information that needs to be provided to the instance of that application running on another mobile edge host.

As a consequence of UE mobility, the mobile edge system needs to support the following:

- *continuity of the service,*
- *mobility of applications (VM), and*
- *mobility of application-specific user-related information.*

"

The same document identifies three mobility requirements:

1. *"[Mobility-01] The mobile edge system shall be able to maintain connectivity between a UE and an application instance when the UE performs a handover to another cell associated with the same mobile edge host."*
2. *"[Mobility-02] The mobile edge system shall be able to maintain connectivity between a UE and an application instance when the UE performs a handover to another cell not associated with the same mobile edge host."*
3. *"[Mobility-03] The mobile edge platform may use available radio network information to optimize the mobility procedures required to support service continuity."*

As a summary, the MEC System shall guarantee service continuity, by application or "application-specific user-related information" mobility, or maintaining connectivity to the ME Host where the required MEC Application is running. This last option will have as a limitation the maximum delay allowed by the Application.

Whenever relocation is needed, the following types may apply.

3.1.3.2 MEC relocation types

Considering the stated requirements, the following application relocations types can be envisioned:

- Application state relocation

  Application state associated to served UEs is transferred between different MEC Applications instances, located at the old and new MEC Hosts, involved in the UE's mobility.

Even if it is possible to have a MEC System assistance (aspect not yet defined at ETSI ISG MEC), MEC Applications will most likely manage themselves the state transfer via communication between the involved instances.

- Application instance live relocation

  Applications running at a ME Host, as VDU, and providing services to specific UEs, are live migrated from old to the new ME Hosts (between different NFVI), serving the eNBs where the UEs are now connected. Mechanisms for a complete VDU relocation, shall resort to virtualized infrastructure capabilities.

  MEC Management and Orchestration needs to participate in the process, coordinating it.

- Application instance emulated relocation

  Application instances relocation can be emulated by the creation of a new instance and deletion of old instance, at originating and target ME Hosts

  If there is no need for Application migration or state transfer but the provided services are needed at the new MEC Host, relocation can be emulated by starting a new Application instance at the new ME Host, while stopping the previous instance at the originating ME Host. For the UE, this operation shall be transparent and the service works in a continuous way.

  MEC Management and Orchestration needs to deal with this process.

- No relocation, keeping service anchor

  In order to keep service continuity and due to required Application state maintenance specificities, the solution may consist in keeping the provided service anchored at the original MEC Application instance, running at the first ME Host the service started being provided.
  This implies associated traffic rerouting between ME Hosts.

- No relocation at all

  At target ME Host, either the MEC Application is not running (e.g. because of the defined Application geographical scope) or an already existing instance is able to provide the same service, without need for any communication with previous instance (e.g. stateless MEC Application). There is no state or application relocation.

3.1.3.3 UE's mobility detection

In all relocation scenarios, and without considering any interaction with EPC control plane (e.g. S1-MME), UE's mobility is only detected and the new serving MEC Hosts is only known once the UE attaches to the new eNB and its traffic is identified. Even if the UE is aware of eNB change, it is not aware of the possible MEC Host change. This way, any needed relocations can only be triggered after UE's arrival at the new MEC Host.

UE's mobility detection can be done at two levels:

1. By the ME Hosts, by observing traffic send to/from a new IP/TEID (Data Plane level)


As a result, the detecting ME Host may proactively query neighbouring ME Hosts about source IP (which ME Host was previously handling that IP).

The previous ME Host handling the UE, may query/notify all running Applications or only the ones that stated the existence of state associated to that IP, about relocation needs.

2. By the MEC Applications themselves, when applicable traffic rules deliver traffic to them

   As a result, and if required (stateful Applications), will require the hosting ME Host to provide its mobility services.

In addition, UE's mobility may be detected by the UE itself. If eNB change is notified to local Application counterpart, this one may notify the MEC System, via the "LCM Proxy", about the possible need for relocation actions.

3.1.3.4 Relocation need detection

In parallel to that, the need for relocation (instance or state) must be identified. This can be known, and also considering previous scenarios:

1. Previously by the ME Host as part of the MEC Application description and communicated to the ME Host at instantiation time
2. At Application execution time (state is created and exists, associated to certain UEs) and communicated to the ME Hosts via an appropriate API:
   a. Inform the MEC System, what IP addresses need to be monitored regarding mobility aspects (proactive)
   b. Whenever a new IP is detected by an Application instance, it may ask the hosting ME Host to trigger the required relocation actions (reactive)
3. Unknown by the ME Host

3.1.3.5 Proposed processes

Both aspects, UE's mobility detection and need for relocation, must be considered together. One approach consists in delegating to MEC Applications the identification of UE's mobility and the identification of relocation needs. Based on that, the following proposals are made.

A. Proposed process for Generic MEC Applications:

1. UE handovers to a new eNB, served by a new MEC Host
2. Configured traffic rules will extract and deliver UE's traffic to applicable MEC Applications
3. Upon detecting traffic from a new IP, stateful MEC Applications will query the hosting ME Host if Application's state exists in another ME Host, expressing relocation actions
4. ME Host queries neighbouring ME Hosts about state related to that IP and MEC Application (IP, AppID)
5. If existing, previous ME Host, notifies, based on AppID, local Application instance about relocation needs


6. As a consequence, that MEC Application instance requests the ME Host to:
    a. Establish a tunnel to the new MEC Host for traffic redirection or
    b. Provide new MEC Application IP address, for managing state relocation
7. Depending on the previous:
    a. Traffic rules at the new ME Hosts are changed accordingly
    b. Application instances exchange state and service for that UE continues at the new Application instance

A similar behaviour may be achieved but with Applications communicating with the corresponding Manager, running at the respective ME Platform Manager. The Application Manager may work as a central point for all instances, coordinating with the MEC System entities, relocation needs and actions. No details for this option are provided.

B. Proposed process for UE's specific MEC Applications:

1. UE handovers to a new eNB, served by a new MEC Host
2. ME Host Data Plane detects a new IP/TEID
3. ME Host queries neighbouring ME Hosts about specific MEC Applications running for that UE related to that IP
4. If existing, previous ME Host, notifies, the local Application instance about UE's mobility
5. As a consequence that MEC Application instance requests the ME Host to:
    a. Establish a tunnel to the new MEC Host for traffic redirection or
    b. Proceed with Application instance relocation to the new ME Host

3.1.3.6 Mobility API

Besides the proposed relocation mechanisms, other solutions are possible. The existence of an API for Applications to communicate with ME Hosts is needed and additional options may be provided, giving place to the definition of other solutions. It is therefore proposed the following API features:

1. Apps to notify ME Host about UEs (IP addresses) to be monitored
2. Apps to request ME Host about UE's originating ME Hosts
3. Apps to request ME Host about originating App IP address
4. Apps to request ME Host about another hosting ME Host IP address
5. Apps to request ME Host to store information at local persistent storage
6. Apps to request ME Host to move/copy stored information to another ME Host
7. Apps to request ME Hosts LCM operations (Stop, Create)
8. ME Hosts to notify Apps about UE's mobility (IP address)
9. ME Host to notify an App about the arrival of stored information (AppID)
10. ME Host to query other ME Hosts about UE handling


4 SLA-Based Descriptors

4.1NEMO

NEMO [12] is a human-readable command language used for Network Modelling. It is placed by the authors in the scope of Intent-Based Networking (IBN), since it is more descriptive and prescriptive. The NEMO project has launched a series of efforts to get the language standardised. As such, the IBNEMO project within the OpenDaylight (ODL) community is classified in the Intent-Based northbound interfaces (NBIs) group.

NEMO provides basic network commands (Node, Link, Flow, Policy) to describe the infrastructure and controller communication commands to interact with the controller.

4.1.1 Enhancements proposed in SUPERFLUIDITY

We proposed to extend the Node definition command to import TOSCA or OSM based descriptors as Node definitions. Since Node models can make use of previously defined node models, the resulting language would be recursive and therefore support our notion of (recursive) reusable function blocks.

This concept has been proposed as an Internet draft [13] at the NFV research group (NFV-RG) of the IRTF. In addition to the import process proper, two additional features are defined in NeMo: 1.- the ConnectionPoint to map the VNF's interfaces that are significant in the function description and the connections between them, and 2.- the CONNECTION as a way to express the relations between the enhanced VNFCs (here NodeModels) in a service graph.

Importing OSM VNF Descriptors is proposed in the draft as a two-step process, where the descriptor is first imported and then used to provide the pointers to the connection points. The proposed syntax to import VNFDs is:

```
CREATE NodeModel sample_vnf
    VNFD  https://github.com/nfvlabs/openmano.git
/openmano/vnfs/examples/dataplaneVNF1.yaml;
    ConnectionPoint data_inside at VNFD: ge0;
    ConnectionPoint data_outside at VNFD: ge;
```

The proposed way to define service graphs in NeMo is:

```
CREATE NodeModel                complex_node
       Node     input_vnf       Type  sample_vnf;
       Node     output_vnf      Type    shaper_vnf;
         ConnectionPoint input;
         ConnectionPoint output;
        Connection input_connection Type p2p EndNodes input,
        input_vnf.data_inside;        Connection output_connection   Type p2p
EndNodes output,  output_vnf.wa ;       Connection internal Type p2p EndNodes
input_vnf.data_outside, output_vnf.lan;
```


4.1.2 Implementation

In order to integrate NFV expressions into NEMO a set of new features has been implemented.

Firstly, there is the need to reference the VNFD file as Universal Resource Identifier (URI). For this purpose, a new parameter has been defined in the NodeModel so that the lexical analyser will recognize a new token corresponding to the VNFD and the parser will add it as part of the NodeModel's parameters.

Secondly, NEMO needs to be aware of the virtual network interfaces defined in the VFND. In order for this to happen, a new network object has been defined, named ConnectionPoint. This new object has different functions: it can record the name of one network interface or it can be just the point of connection for a new NodeModel. Moreover, a new interesting feature has been integrated: we can think about VNFDs with N interfaces that could be set for the ConnectionPoint and we can choose which of these interfaces could be attached to the ConnectionPoint when it is instantiated. This is already possible by using the NodeModel's properties.

Thirdly, NEMO model provided a connection model to express the link between two network nodes. However, this feature has been extended and it now provides the link between either network nodes or connectionPoints. Because of this, it is possible to create the link between two simpler VNFDs.

The implementation of these basic features has enabled the creation of recursive VNFs. This means that it is possible to reuse NodeModels recursively to create complex NodeModel. Each NodeModels will check whether a node type matches the basic definitions (host, l2-group, l3-group, ext-group, chain-group, firewall and loadbalancer) or needs to be instantiated because it has a template definition (so achieving the recursiveness).

Moreover, it is possible to delete every Object created by NEMO. So that if a mistake occurs while writing the intent, it is possible to delete it and rewrite it again.

Finally, the processing of the NodeModels is needed in order to generate a complex YAML file (based on the baseline provided by OSM VNFDs) as outcome.

4.2 Support for heterogeneous and nested execution environments

This section proposes extensions to the ETSI NFV ISG specification [14] [13] [15] to support nested VDUs using heterogeneous technologies.

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.


	Qualifier	Cardinality	Content	Description
vduNestedDesc	O	0..1	Identifier (Reference to a VduNestedDesc)	This is a reference to the actual 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.
vduNestedDescType	O	0..1	Enum	Identifies the type of descriptor file referred in the vduNestedDesc field (Click configuration, kubernetes template, etc.) and consequently the type of the Execution Environment running in the VDU. This field must be present if vduNestedDesc is present.
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. Setting this field to the special value “None” specifies that this VDU is set to be deployed on bare metal.
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 and specified. The absense of this field is equivalent to setting its value to “false”.

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 configurations

VDUs. In particular, we have introduced four new attributes to the VDU information element: namely *vduNestedDescType*, *vduNestedDesc, vduParent* and *vduParentMandatory*.

- The *vduNestedDescType* attribute defines the type of RFB Execution Environment that is running in the VDU (in our case *kubernetes* or *click*).
- The *vduNestedDesc* attribute is an identifier. It provides a reference to the actual descriptor that is deployed in the REE running in the VDU (in our case a Click configuration file or a Kubernetes template). The proposed *vduNestedDesc* attribute uses the same approach of the *swImage* attribute, which provides a reference to the actual software image that is deployed in a "regular" VDU.
- 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 *vduParentMandatory* attribute applies also 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).

7.1.6.4.2 [VduCpd] Attributes

Clause 7.1.6.4.2 is moadified 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”.

4.2.1 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, 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.

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:

- vduNestedDescType: kubernetes
- *vduNestedDesc*: k8stemplate
- vduParent: vmid
- vduParentMandatory: true

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:

- vduNestedDescType: kubernetes
- *vduNestedDesc*: k8stemplate
- vduParent: none
- vduParentMandatory: true

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:

- vduNestedDescType: kubernetes
- *vduNestedDesc*: k8stemplate
- vduParent: vmid
- vduParentMandatory: false

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.

4.2.2 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 diffferent deploymentFlavors for the different deployment scenarios?

5 Conclusion

The work on Task 6.2, as summarized in this internal deliverable, focused on the resource allocation and life cycle management for both the core and edge network. Specifically, for the core network we modelled the problem of virtual network function (VNF) allocation by queuing system with flexible number of queues, formulated it as a Markov decision process (MDP) and numerically solved it to find the optimal allocation policy. Then, we studied near optimal deployment strategies for service chains that provide the needed performance yet minimizing the operation resource overhead. Next, looking at the life cycle management of containerized VNFs, we investigated the scaling behaviour of the underlaying VM-based infrastructure. Here, we derived scaling policies by means of machine learning based on applications KPIs. Finally, to facilitate scaling and migration we investigated load balancing as a service. We deployed NetScaler ADC (VPX edition) as infrastructure load balancer of the NFV Reference Lab and integration with OpenStack using the corresponding Neutron LBaaS plugin, configured via NetScaler MAS. In addition, we implemented an LBaaS network function with NetScaler ADC, as described in Deliverable I6.2b. For the MEC, we studied the challenges in placement of ME application and derived the factors to determine the required number of MEC instances and their deployment location. Then, we studied the service migration and mobility.

6 References

[1] M. C. Luizelli, D. Raz, Y. Saar and J. Yallouz, “The Actual Cost of Software Switching for NFV Chaining”, IFIP/IEEE International Symposium on Integrated Network Management (IM 2017), 2017.

[2] M. Shifrin, E. Biton, and O. Gurewitz. Optimal control of VNF deployment and scheduling. In Science of Electrical Engineering (ICSEE), IEEE International Conference on the, 2016.

[3] OpenStack Project “Neutron/LBaaS” [Online]. Available: https://wiki.openstack.org/wiki/Neutron/LBaaS [Accessed 28/06/17]

[4] OpenStack Octavia [Online]. Available: https://wiki.openstack.org/wiki/Octavia [Accessed 28/06/2017]

[5] Citrix NetScaler ADC [Online]. Available: https://www.citrix.com/products/netscaler-adc/

[6] SUPERFLUIDITY Deliverable I6.1b

[7] Citrix NetScaler MAS [Online]. Available: https://www.citrix.com/products/netscaler-management-and-analytics-system/

[8] OpenStack Neutron/LBaaS Project [Online]. Available: https://wiki.openstack.org/wiki/Neutron/LBaaS/NetScaler

[9] OpenStack LBaaS Netscaler Driver Repository [Online]. Available: https://github.com/openstack/neutron-lbaas/tree/master/neutron_lbaas/drivers/netscaler

[10] Citrix Docs, “Installing NetScaler MAS on KVM” [Online]. Available: http://docs.citrix.com/en-us/netscaler-mas/12/deploy-netscaler-mas/install-mas-on-kvm.html

[11] Citrix Docs, “Integrating NetScaler MAS and OpenStack - Preconfiguration” [Online]. Available: http://docs.citrix.com/en-us/netscaler-mas/12/integrating-netscaler-mas-with-openstack-platform/preconfiguration-tasks-mas-openstack.html

[12] “NeMo: An Application’s Interface to Intent Based Networks” http://nemo-project.net/

[13] “High-level VNF Descriptors using NEMO” https://datatracker.ietf.org/doc/draft-aranda-nfvrg-recursive-vnf/

[14] ETSI NFV ISG, “Network Functions Virtualisation (NFV); VNF Packaging Specification”, ETSI GS NFV-IFA 011 V2.1.1 (2016-10) pdf link

[15] ETSI NFV ISG, “Network Functions Virtualisation (NFV); Network Service Templates Specification”, ETSI GS NFV-IFA 014 V2.1.1 (2016-10) pdf link