D6.2 Configuration of the Pilots and pre-validation

Project Acronym	SONATA
Project Title	Service Programing and Orchestration for Virtualized Software Networks
Project Number	671517 (co-funded by the European Commission through Horizon 2020)
Instrument	Collaborative Innovation Action
Start Date	01/07/2015
Duration	30 months
Thematic Priority	ICT-14-2014 Advanced 5G Network Infrastructure for the Future Internet

Deliverable	D6.2 Configuration of the Pilots and pre-validation
Workpackage	WP6 Infrastructure setup, validation and pilots
Due Date	30/04/2017
Submission Date	02/06/2017
Version	0.1
Status	To be approved by EC
Editor	George Xilouris (NCSRD)
Contributors	Michael Bredel (NEC), Alberto Rocha (ALB), Wouter Tavernier (IMEC), Manuel Peuster (UPB), Shuaib Siddiqui (i2CAT), Dani Guija (i2CAT), Einar Meyerson (i2CAT), Juan Antonio Nuez (i2CAT), Javier Fernandez (i2CAT), Felipe Vicens (ATOS), Panos Trakadas (SYN), Dario Valocchi (UCL), Stavros Kolometsos (NCSRD), Philip Eardley (BT), Eugene Otoakhia (BT)
Reviewer(s)	Michael Bredel (NEC), Santiago Rodríguez (Optare), Sharon Mendel (Nokia), Eugene Otoakhia (BT)

Keywords:

Qualification Infrastructure, Pilots, NFV, vCDN, PSA, SP2SP

Deliverable Type		
R	Document	**X**
DEM	Demonstrator, pilot, prototype	
DEC	Websites, patent filings, videos, etc.	
OTHER		
Dissemination Level		
PU	Public	**X**
CO	Confidential, only for members of the consortium (including the Commission Services)	

Executive Summary:

This deliverable presents the current status of pilots configuration and pre-validation activities. The deliverable comprises of two parts. The first part aims to renew information delivered in Deliverable 6.1, as a infrastructure deployment moved on and new configurations were put in place. The second part provides technical details on the pilot deployment and configuration as well as the preliminary implementation of VNFs. The summary this deliverable updates the information with respect to the deployed infrastructure supporting SONATA integration, qualification and demonstration environments. The demonstration infrastructure is now expanded via connection of remote testbeds to the original core infrastructure of SONATA in Athens. The remote testbeds (i.e. Aveiro, Paderborn, Tel-Aviv, and London) are interconnected via VPN connections and provide a more large scale view of the SONATA deployment. Under this prism, multiple configuration may be applied in order to conduct experimentations and realise the pilots. Detailed technical descriptions are provided for each testbed and the hosting infrastructure. Moreover, specific focus and role is initially assigned for each participating testbed. The final selections of pilots, according to the selection methodology followed in Deliverable 6.1 concluded to the: (i) Virtual Content Delivery Network pilot; (ii) Personal Security Application pilot and (iii) the Hierarchical Service Providers pilot. The detailed technical information about the above pilots is provided by this document taking into account the relative progress in its one. The documents also provides information on the VNFs and the Network Services that will be used to implement the pilots. Finally the document concludes with the evaluation strategy of SONATA which is based on the approach proposed in ETSI NFV TST documents. The mapping of the the pilots mapping to the elicited requirements of Deliverable D2.3 is updated as well as the metrics that should be used for the validation of non-functional requirements.

Contents

List of Figures

List of Tables

1 Introduction

In the current progress phase, SONATA project is moving from the development phase towards the demonstration phase. This deliverable is continuation of the work done in Deliverable 6.1. As such it covers the aspects of the SONATA environment evolution in order to cope with the requirements of the selected Pilots. In D6.1 the focus point of the deliverable was mainly to provide a definition and an overview of the SONATA infrastructures used for the first year of the project, deployment scenarios and the original hosting infrastructure used to form all the anticipated environments. These infrastructures are: (1.) integration infrastructure (II); (2.) qualification infrastructure (QI) and (3.) demonstration infrastructure (DI). The extensions described in this deliverable are regards the changes and configurations made in all the environments plus the introduction of more testbed islands interconnected with the main hosting infrastructure. The above details are shown in Section 2 of this deliverable.

As a consequence of the adoption of Continuous Integration / Continuous Deployment paradigm, the need for certain infrastructure deployment guidelines was considered. Section 3 of this deliverable covers this aspect providing hints and lessons learnt from the original deployment of the SONATA environments. Although this information is actually a work in progress, the deliverable presents an initial approach.

In Deliverable 6.1 and also in the deliverable updating the WP2 architecture and requirements, Pilot selection was covered. In summary the D6.1 imposed a methodology for the Pilot selection and D2.3 updated the information on the anticipated Pilots. In this context D6.2 delivers detailed information on the pilots architecture, deployment and building blocks and summarises the efforts on each component implementation. At the same time, the SONATA added value and mapping to the original objectives of the projects is covered.

The document is structured as follows:

Chapter 1 is the introduction of the deliverable discussing the contribution and scope of the document.

Chapter 2 presents the infrastructures employed by the SONATA framework.

Chapter 3 introduces the recommended deployment flavours.

Chapter 4 presents the Pilots technical details and the relevant VNFs.

Chapter 5 shows the pre-evaluation mapping and requirements coverage for all the Pilots.

Chapter 6 concludes the document.

1.1 Document Interdependencies

This deliverable integrates the work carried out so far within the other technical WPs, and as such, contains either implicit or explicit references to deliverables summarised in the following table.

Table 1.1: Document interdependencies

Deliverable Name	Description	Reference
D2.1 - Requirements Elicitation	This deliverable is relevant due to the elaboration on the initial use cases envisaged by SONATA in order to elicit requirements. The use cases are used in this deliverable in order to lead to pilot specification and specific demonstrators. Furthermore, clarifications on the specification of the SONATA infrastructure flavours are fuelled by this deliverable.	[1]
D2.3 - Updated Requirements and Architecture Design	This deliverable is the updated requirements and use cases discussed in D2.1 and D2.2. The updated information in D2.3 already describes those use cases that are considered for Pilots.	[4]
D3.1 - SDK prototype	This deliverable has relevant information that affects the specification of the Integration and Qualification infrastructure flavours.	[5]
D4.1 - Orchestrator prototype	This deliverable has relevant information regarding the definition of the Infrastructure Adaptor, component that interacts directly with the NFV infrastructure manager (VIM) or WAN infrastructure manager (WIM)	[6]
D5.1 - Continuous integration and testing approach	This deliverable provides relevant information regarding the definition and specification of the Continuous Integration / Continuous Deployment (CI/CD) methodology followed by SONATA. The established methodology and identified integrated components are exploited by this Deliverable in order to precisely define the Integration/Qualification infrastructures. Moreover this deliverable also elaborated on the initial results from the infrastructure versions of WP6.	[3]
D5.2 - Integrated Lab-based SONATA Platform	This deliverable is synchronous to D6.1, it serves as a first integration of the SONATA platform and elaborates on the integration tests for the SONATA components. The results of this effort will be transferred to the integration and qualification infrastructure for further testing and validation. From the D6.1 scope this deliverable provides a more technical insight on the components integration and interfacing with the underlying infrastructures.	[9]
D5.3 - Integrated and qualified public release of SONATA platform	his deliverable updates on the work done in D5.1 and D5.2 by providing information on the integration and qualification environments.	[7]
D6.1 - Definition of the Pilots, infrastructure setup and maintenance report	This deliverable is being updated by this document, providing final information on the Infrastructures used for SONATA deployment. Moreover building on top of the Pilot selection methodology it provides more technical information on them.	[8]

2 Infrastructure Flavours

This section elaborates on the infrastructure flavours used to support the SONATA environments. The complete SONATA deployment utilizes a hosting physical infrastructure supporting Integration and Qualification environments.

2.1 Hosting Infrastructure

In order to be able to automatically deploy SONATA artifacts, the existence of an underlying available data centre infrastructure is assumed. Additionally it requires communication with VIM and WIM components in order to be able to orchestrate the available resources in the underlying physical infrastructures. This section describes the hosting infrastructure as it was laid out for the second year of the project. The hosting infrastructure is located in Athens (mainly) and Aveiro testbeds. It comprises of compute and network resources along with their management entities.

The following figure (Figure 2.1), illustrates the main segments of the infrastructure currently deployed in Athens. Similar structure, apart from the CI/CD tools is used in Aveiro testbed. More details on the full spectrum of testbeds used in SONATA are provided in following sections.

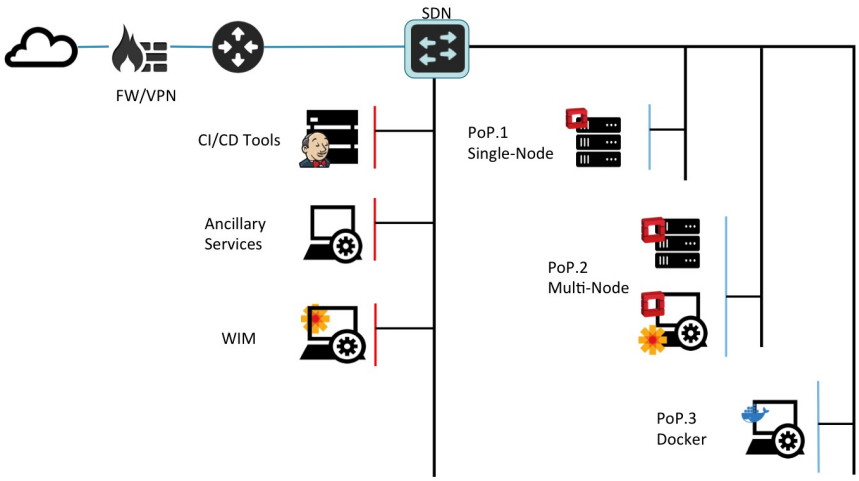

Figure 2.1: Hosting infrastructure

The enumeration of the components comprising the hosting infrastructure is as follows:

- SDN Switch - provides the networking infrastructure to be used for the connection of all the segments, and allows for the management and control of traffic forwarding across the infrastructure. The networking equipment used is, of course, accompanied with a number of non-SDN switches. In this deployment Pica8 SDN switch is employed.
- 3 NFVI-PoPs - the PoPs where VNFs are instantiated are represented by three PoPs, (i) a Single node - all-in-one Openstack deployment named PoP.1; (ii) a Multi-Node Openstack deployment named PoP.2 and (iii) a pure docker based PoP named PoP.3 (experimental).

- CI/CD tools - are deployed on separate infrastructure based on ESXi (VMware), that hosts components and services used by the integration environment.
- Ancillary services segment - provides a collection of tools and performance and validation software.
- VPN endpoints connecting testbeds in other premises, are also routed towards the central SDN switch.
- WAN Infrastructure Manager (WIM) - manages and controls the traffic across the infrastructure, focusing on interconnecting Users, Servers and PoPs.
- ODL Controller - control of the SDN networking equipment.

In the following Table 2.1, detailed information on the hardware can be found:

Table 2.1: Hosting infrastructure specifications

Function	IP	CPUs	HDD(GB)	Memory(GB)
Single-node PoP 1	10.100.32.200	Intel(R) Xeon(R) CPU E5620 @2.40GHz	1000	28
Multi-node PoP 2 - Controller	10.100.16.0/20	Intel(R) Xeon(R) CPU E5620 @2.40GHz	1800	16
Multi-node PoP 2 - Node1	10.100.16.0/20	Intel(R) Xeon(R) CPU E5620 @2.40GHz	2x146G + 1T	28
Multi-node PoP 2 - Node2	10.100.16.0/20	Intel(R) Xeon(R) CPU E5620 @2.40GHz	2x146G + 1T	28
Docker-based PoP 3	10.30.0.248	Intel(R) Xeon(R) CPU X3450 @2.67GHz	500	16
WIM	10.30.0.13	Intel(R) Core(TM)2 Quad CPU Q9300 @ 2.50GHz	500	8

2.2 Integration Infrastructure

The integration infrastructure (II) has changed since D6.1 [8]. Due to an issue with the hardware in the testbed, the integration VMs and Repository were transferred to a VMware ESXi 6.5 cloud environment (see CI/CD tools in Figure 2.1). This environment provides with direct access to the storage system and with virtualisation factor of 1, which implies that the amount of CPUs and Memory that can be virtualised equals to the actual cores provided by the system. Hyper-threading support for the CPU is enabled to double the CPU core enumeration using the actual CPU capabilities for multiple memory pipelines towards the physical CPU. Part of the performance boost is also due to the use of newer hard disks and 10G network card for the attached storage using NFS protocol (mainly used for back up and caching).

The most significant update is the migration of whole environment to this new server with more compute resources and stability. The VMs were resized and adapted to new requirements of SONATA.

2.2.1 Comparison of physical hardware between old and new integration environment

Table 2.2 presents the difference between the previous and current version of Integration Environment physical server specifications.

Table 2.2: Previous versus current integration environment physical server description

Environment	CPUs	RAM(GB)	HDD(GB)	Openstack Version
Previous	16xIntel Xeon(R) E5620@2.5GHz	64GB	1000	Openstack Liberty
Current	32xIntel Xeon(R) E5677@3.47GHz	96GB	3000	VMware ESXi 6.5

Table 2.4 provides a summary of new VM characteristics, and the changes that were performed in comparison with Table 2.3.

Table 2.3: Integration Environment Old Setup

VM	Function	SW tools	vCPU	HDD(GB)	Memory(GB)
#1	CICD engine	Jenkins	4	120	4
#2	int-srv-1	SP, API, BSS, GUI	4	80	8
#3	int-srv-2	SP, API	4	80	8
#4	int-srv-3	SP	4	80	8
#5	Repos	Docker Registry	4	80	8
#6	Monitory	MONIT	4	80	8
#7	Logging	LOGs	8	160	16

Table 2.4: Integration Environment New Setup

VM	Function	SW tools	vCPU	HDD(GB)	Memory(GB)
#1	CICD engine	Jenkins	8	120	8
#2	int-srv-1	SP, API, BSS, GUI	2	80	4
#3	int-srv-2	SP, API	2	80	4
#4	int-srv-3	SP	2	80	4
#5	Repos	Docker Registry	2	80	2
#6	Monitory	MONIT	4	80	8
#7	Logging	LOGs	2	80	4

Figure 2.2 shows the components of the integration environment VMWare ESXi server, which are briefly described in the following paragraphs.

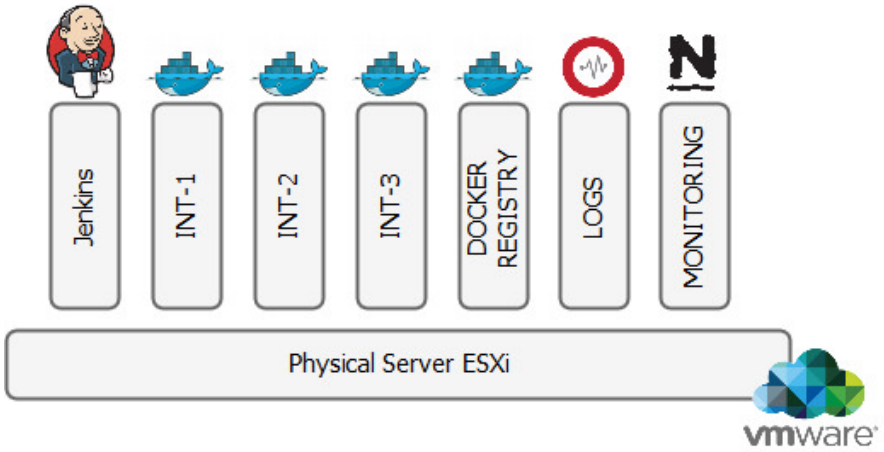

Figure 2.2: Components of the integration environment

The integration environment deployed in the II is composed by 7 VMs. Four of theses VMs take care of the second phase (integration phase) of CI/CD pipeline. The Jenkins server is the core

of the integration environment and therefore requires most of the compute resources. This server has a direct connection to the integration servers (int-srv) 1, 2 and 3. The integration server 1 is used to run a set of integration tests required by the development phase. The integration server 2, functions solely as the monitoring test server. This is because a clean environment is required to run monitoring tests. The integration server 3 is prepared to be the step prior to qualification, this server runs all the components of the service platform for developers to immediately test code that has been pushed to Github.

To provide a complete set of integration tools, the integration infrastructure has 3 additional VMs. A logging server that has installed the open source tool Graylog (graylog.com). The developers can access the the compilation of all logs that comes from integration servers and have an easy way to identify the bugs in real time. The docker registry server supports the storage of the docker containers created in Jenkins to be distributed between the environments. The last VM that is part of integration environment is a monitoring server that has the opensource tools Cacti (cacti.net)and Nagios (nagios.org) to monitor the status of the integration environment and send alerts to the sysadmins in case of system failures.

2.3 Qualification Infrastructure

SONATA CI/CD process requires an infrastructure to execute the qualification test for the software coming from earlier phases in the pipeline as development and integration (i.e. Qualification Infrastructure - QI). This qualification phase require an infrastructure to perform a set of tests namely performance, security, conformance and functional tests to guarantee the quality of software that SONATA is releasing. To this end, the QI is composed by a set of compute and networking resources allocated in the testbed of multiple parters. These resources are a slice of the infrastructure described below in demonstration infrastructure section. The following paragraphs describe the architecture of QI and the quota allocated for this environment.

2.3.1 Qualification Tests

The conformance tests are executed on the following specific environments suitable for each test.

- The first environment is used to perform end-to-end tests related to deployment of network services with multiple flavours.
- The second environment is used to perform stress and security tests related to the service platform.
- The third environment is used to perform installation of the SONATA SP and its verification in multiple operational environments.

2.3.1.1 End-to-End tests

The QI will support end-to-end qualification tests with the resources described in this section (see Figure 2.3). The qualification end-to-end tests uses 2 testbeds, Athens and Aveiro. In this environment requires four components:

- First is the Jenkins+Ansible VM, which is the main component of CI/CD process. Notice that Jenkins+Ansible server is installed in an ESXi environment. This is the same server mentioned in II. This VM contains all the tests jobs implemented and it uses ansible to communicate to Openstack to deploy, reset and configure the SONATA service platform. Jenkins will perform the tests and show the results.

- The second one is the PoPs, in this case it is Openstack installation in Athens testbed (2PoPs) and Aveiro testbed (1PoP). These Openstack will support the VNFs deployment of the NS created in the tests.
- The third component is the SONATA Service Platform installed in Aveiro testbed. It comprises of two VMs running the service platform components to be used to instantiate the Network Services for the tests.
- The last component is the Physical End-User machine that is a Desktop server controlled by Jenkins to inject traffic into the network service as part of the test to verify the deployment of the network services.

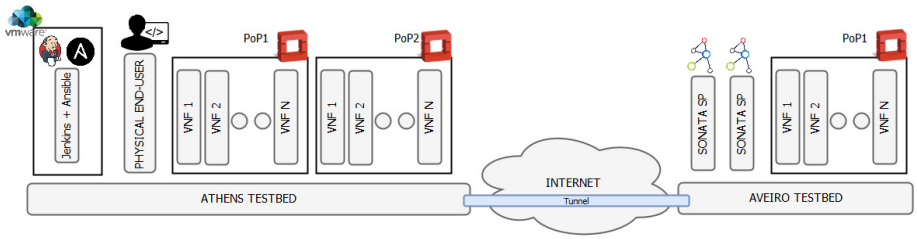

Figure 2.3: End to end tests resources

Table 2.5 details the used resources in each testbed.

Table 2.5: End to end test resources

Resource Name	CPU	Memory	Disk
PoP1 Athens	4/8()	28G	2x146G
PoP2 Athens	8/16(HT)	56G	2x146G
PoP3 Athens	8/16(HT)	16G	2x146G
PoP1 Aveiro	16	48G	500G
SONATA SP	4	8G	40G
Physical Desktop	2	4G	20G

Below there is a list of self-explanatory associated qualification tests:

1. POST-INSTALL test script
2. Deploy 1 VNFs 1 PoPs
3. Deploy 2 VNFs 1 PoPs
4. Deploy 2 VNFs 2 PoPs
5. Deploy 2 VNFs 2 PoPs (SSM react when CPU utilisation is high)

2.3.1.2 Stress and Security tests

QI will support stress and security verification tests. Just like end-to-end tests, these tests will use Athens and Aveiro testbeds. This environment requires 2 components:

- The first one is the Jenkins+Ansible VM located in Athens testbed. This VM have Jenkins jobs to perform the security and performance tests.

- The other component is a fresh installation of the SONATA Service Platform located in Aveiro testbed. This Service Platform installation have the same IP address and DNS to link the Jenkins tests to this Service Platform.

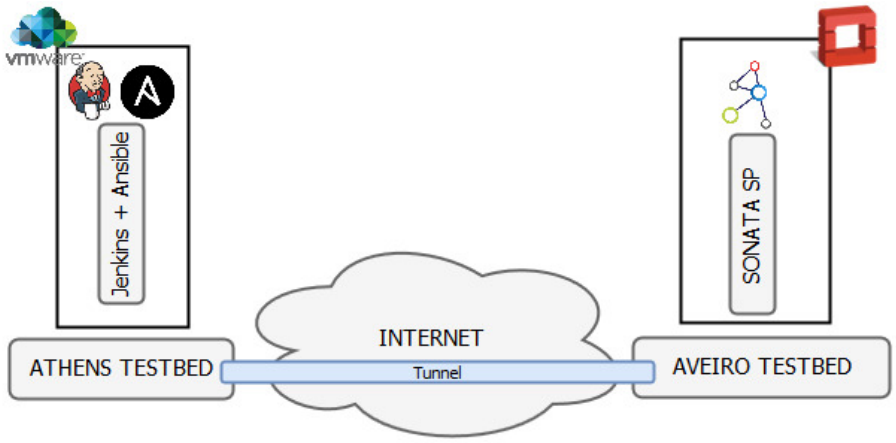

Figure 2.4: Stress and Security tests resources

Resources can be shown in Table 2.6

Table 2.6: Stress and performance test resources

Resource Name	CPU	Memory	Disk	Flavour
SONATA SP (Aveiro)	2	4G	40G	m1.medium

Associated tests:

1. POST-INSTALL test script
2. Check the certificates generated for the GUI and BSS
3. Massive package upload
4. Stress the Gatekeeper API
5. Stress the MANO framework
6. Stress monitoring
7. Stress catalogues and repos
8. Stress the IFTA with mocked VIM

2.3.1.3 Installation of the SP on multiple OS

The last environment illustrated Figure 2.5 is responsible for the installation of the service platform. This environment uses Athens and Aveiro testbeds, and has two components:

- The first one is the Jenkins+Ansible VM located in Athens testbed.
- The other component is a the Aveiro Openstack where the Service Platform will be used to perform installation tests.

Table 2.7 shows the resources.

Table 2.7: SP installation test resources

Resource Name	CPU	Memory	Disk	Flavour
SONATA SP Ubuntu	2	4G	40G	m1.medium
SONATA SP CentOS	2	4G	40G	m1.medium

Associated tests:

1. POST-INSTALL test script
2. Install SP on Centos7
3. Install SP on Ubuntu 16.04

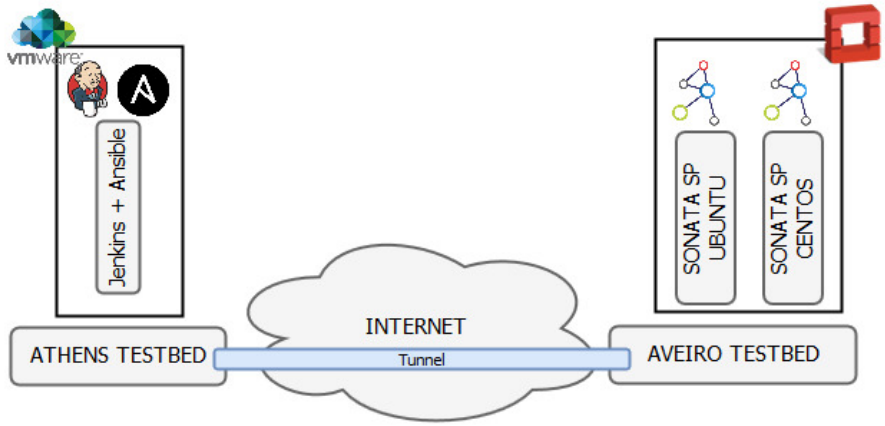

Figure 2.5: SP installation tests resources

2.4 Demonstration Infrastructure

In order to support final demonstration activities related to pilots and PoCs, a demonstration infrastructure comprised of all the provided testbeds is defined. Currently the partners that offer testbeds for experimentation and demonstrations are: i) NCSRD (Athens); ii) Altice Labs (Aveiro); iii) University College London - UCL (London); iv) University of Paderborn - UPB (Paderborn) and v) NOKIA (Tel-Aviv). All the testbeds are interconnected using a star topology to Athens VPN concentrator and expose different capabilities based on the available hardware and the resource sharing available in each testbed. As such the demonstration infrastructure of SONATA is able to provide adequate resources for evaluations related to both the Service Platform and the pilots which consume resources at various locations, with different configurations and network conditions. For example, instantiation of multiple SONATA SPs to configure and manage different segments of the infrastructure or instantiation of single SONATA SP managing multiple NFVI-PoPs in remote locations. Figure 2.6 illustrates the overall topology of Demonstration Infrastructure. It can be observed the participating testbeds provide different components of the entire SONATA framework (e.g. London testbed provides NFVI-PoPs based on Docker or Openstack), allowing a number of combinations for future experimentation campaigns.

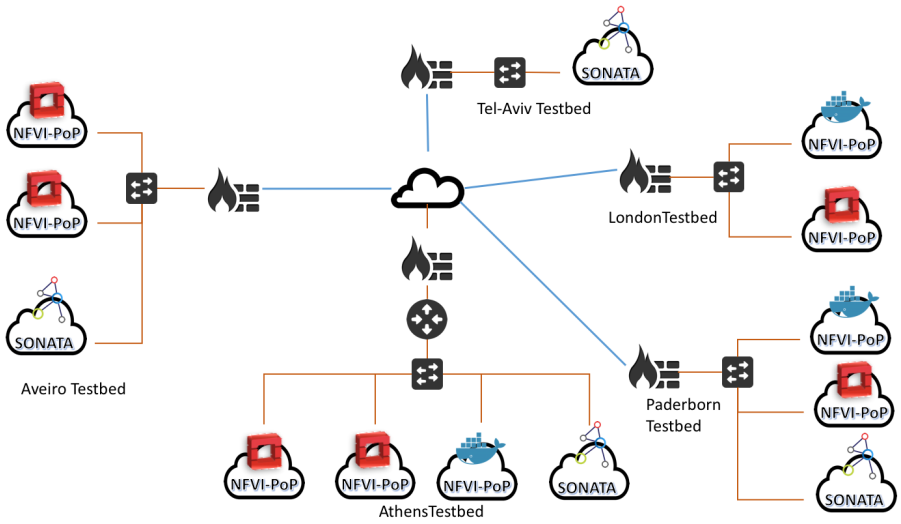

Figure 2.6: Overview of demonstration infrastructure

2.4.1 Athens Testbed

This section presents the available infrastructure at NCSRD premises. The hosting infrastructure of the SONATA deployment is based. As illustrated in (Figure 2.1), all the components of the SONATA deployment are also installed here. Besides the II and QI (Multi-Pop), the site hosts three NFVI-PoPs that will be used for the demonstrators as well as the whole DevOps framework (i.e. SDK, SP) of SONATA.

This section describes the computational, networking and storage resources use and topology.

2.4.1.1 Physical View

The logical view of Athens testbed is illustrated in (Figure 2.1). All the infrastructure elements are interconnected using Ethernet links in isolated L2 segments and routed by the core router (i.e. CISCO ISR 2800). Access to the Internet and also to the various tunnels interconnecting other testbeds is provided by CISCO ASA Firewall.

The detailed physical view (where applicable) and the resources assigned for each segment are presented is subsequent sections below.

NFVI-PoP 1 - Single-Node

The all-in-one deployment of Openstack is used as a very simple NFVI-PoP deployment at the very edge of the domain. The installation and setup for this NFVI-PoP is rather simple and allows quick deployment and testing, especially for the integration environment. However SFC issues arising from the multi-node deployments are not captured by this NFVI-PoP.

The All-in-one Openstack deployment is based on Mitaka release and is illustrated in Figure 2.7.

NFVI-PoP 2 - Multi-Node

The figure below (Figure 2.8) presents the multi-node NFVI-PoP deployed in Athens testbed. This PoP is based on Openstack Mitaka release. In order to allow better support for NFV network services deployment, SONATA specific SFC agent is also configured on this system. Moreover modification regarding specific configurations for neutron have been made. These related to firewall, anti spoofing, etc.

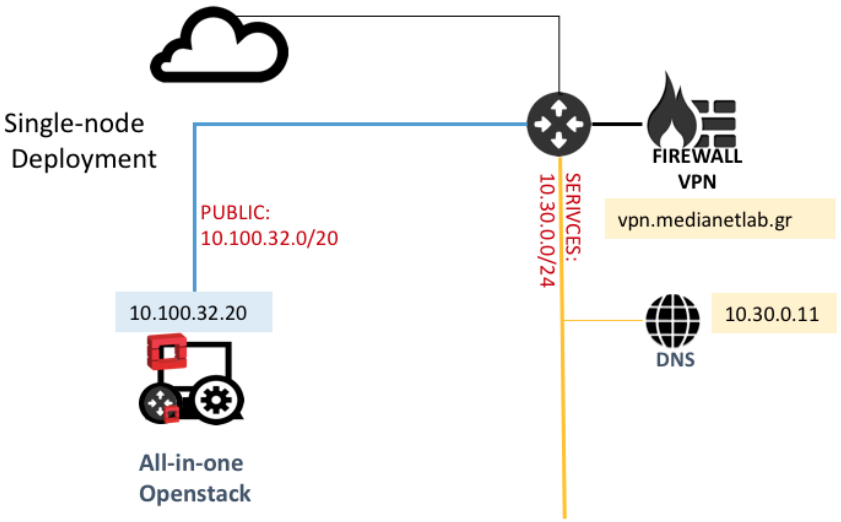

Figure 2.7: All-in-one NFVI-PoP

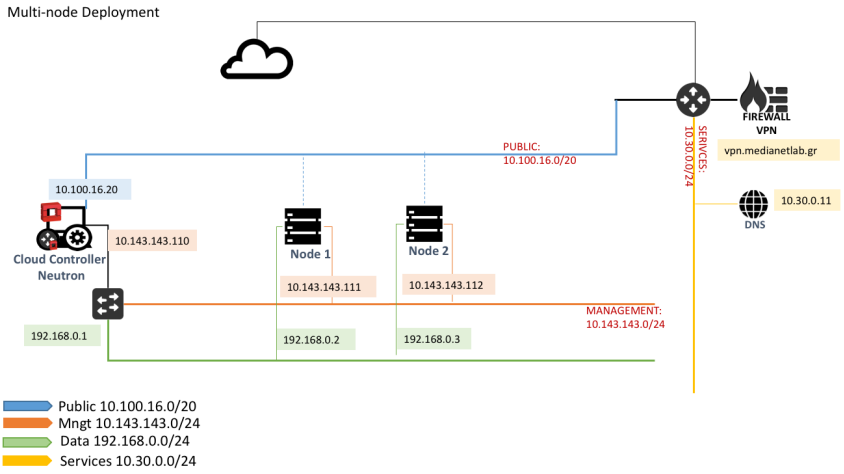

Figure 2.8: Multi-node NFVI-PoP

NFVI-PoP 3 - Docker Based

As SONATA is inherently designed to support alternative VIMs through wrappers that can be programmed at the Infrastructure Abstraction (IA) layer, means to test this capability are provided by the docker based NFVI-PoP. The implementation of this VIM is meant to be simplified and serve proof-of-concept purposes. As such not all Service Platform operations will be supported at this point. The PoP is comprised of a single server with Linux Ubuntu OS, configured to run Docker service and an agent that exposes interfaces with similar functionality as the Openstack VIM. The topology of the NFVI-PoP is illustrated in Figure 2.9.

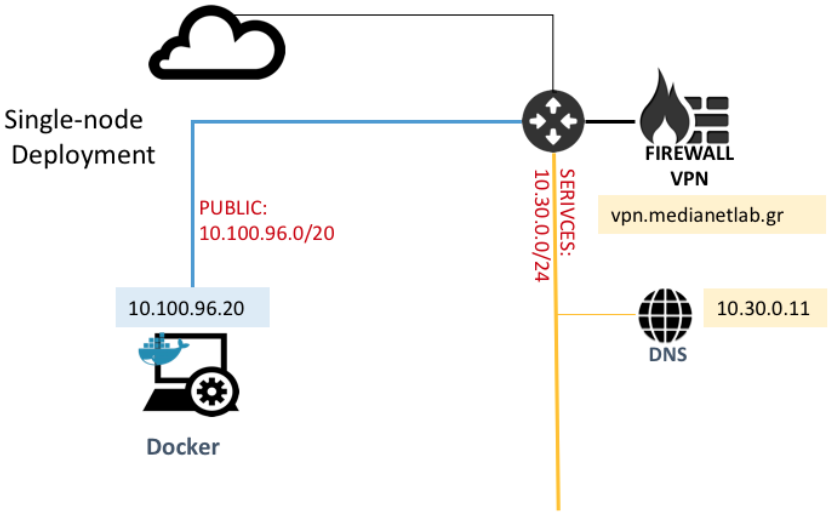

Figure 2.9: Docker based NFVI-PoP

Openstack Cloud - Service Platform

The following Figure 2.10 illustrates the infrastructure over which SONATA Service Platform is being deployed. The infrastructure is based on OpenStack environment, deployed using Mirantis FUEL installer bundled with OPNFV Danube. The latest release of OPNFV supports a number of NFV features such as Service Function Chaining, Interfacing with various Orchestrators, Network features supported at the Neutron service. These features allow refined control over infrastructure resources. This cloud will be used for instantiation of Service Platform for integration testing or to be used for conformance testing and demonstrations.

Computational Resources

The computational resources employed by the project in Athens have been described in previous sections, and for each infrastructure flavour.

Networking resources

The networking resources allocated for the NCSRD PoP Infrastructure are:

- A Dell PowerConnect 5524 Gigabit switch used for the storage network.

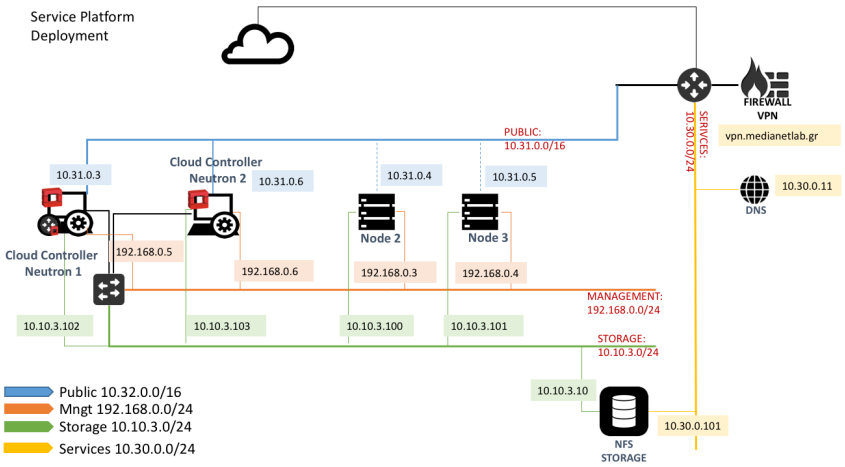

Figure 2.10: SONATA SP deployment infrastructure

- A Cisco SPS2024 Gigabit switch for Management,Public,Data networks.
- A Pica8 SDN Gigabit switch used for networking of users, servers and PoPs controlled by ODL and WIM.

The routing, firewall and access server equipment are based on Cisco ASA 5510 series.

2.4.1.2 Role in SONATA

Athens testbed role in SONATA infrastructure is critical. It is the central island where each and every other testbed is connected to. Since the start of the project, Athens testbed provides hardware and support for all the SONATA infrastructure flavours. As such Athens testbed participates in all Pilot demonstrations and evaluation campaigns.

2.4.2 Aveiro Testbed

The Altice Labs testbed located in Aveiro, Portugal, is a dedicated infrastructure based on **8** HP Proliant BL460c G1 bare metal servers running OPNFV 3.0 Openstack Mitaka deployed by Mirantis Fuel 9.0 with the following layout:

Table 2.8: Aveiro testbed

BL	Role	CPU	RAM	DISK	ETH
13	Mirantis Fuel	2x 2-core	16GB	146GB mirror	2x 1Gb ETH
06	Openstack Controller	2x 2-core	16GB	146GB mirror	2x 1Gb ETH
14	Openstack Controller	2x 2-core	16GB	146GB mirror	2x 1Gb ETH
07	Openstack Node	2x 4-core	24GB	4x 300GB RAID-5	2x 1Gb ETH
08	Openstack Node	2x 4-core	24GB	4x 300GB RAID-5	2x 1Gb ETH
05	Openstack LMA	2x 2-core	16GB	146GB mirror	2x 1Gb ETH
04	ODL SDN Controller	2x 2-core	16GB	146GB mirror	2x 1Gb ETH
03	OVS switch	2x 2-core	16GB	146GB mirror	2x 1Gb ETH

2.4.2.1 Physical Infrastructure view

Figure 2.11 shows the enclosure chassis with 8 blade servers allocated to SONATA Altice Labs PoP at Aveiro.

Figure 2.11: Physical infrastructure view

2.4.2.2 Logical Infrastructure view

Figure 2.12 shows the role played by each bare metal server on the Altice Labs infrastructure.

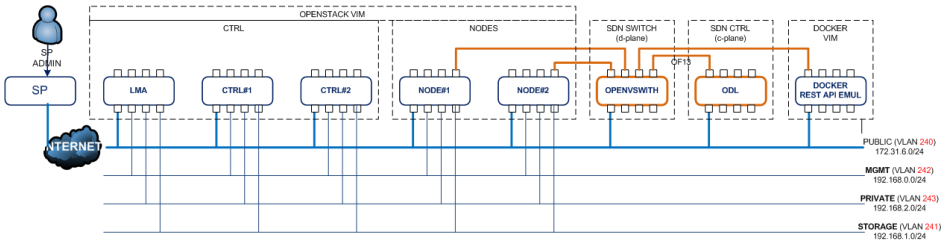

Figure 2.12: Logical Infrastructure view

2.4.2.3 Connectivity

Currently, Altice Labs has 2 redundant 1Gbps dedicated MPLS circuits to the Internet and a 50Mbps dedicated circuit for VPN remote access. As Figure 2.13 illustrates the Aveiro network topology comprises of :

- The Openstack public network segment “172.31.6.0/24” is NATed to c-class “194.65.128.0/24” address space.
- A permanent site-to-site layer 3 SSL tunnel is established between Aveiro and Athens PoP’s.
- A VPN Remote Access to Altice labs, using Cisco AnyConnect VPN client with credentials:

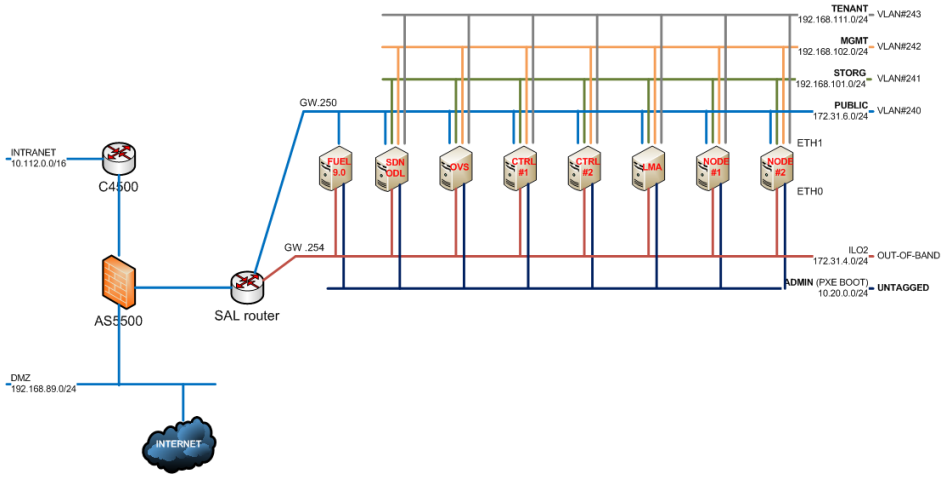

Figure 2.13: Aveiro PoP physical network topology

```
  URL: inovpn.ptinovacao.pt/SONATA
U: usersonata
P: **********
```

The external network segment is "194.65.138.0/24" is visible in the Internet via NAT to the SONATA public network segment.

2.4.2.4 Role in SONATA

The SONATA infrastructure provided by Altice Labs is dimensioned to be a VNFI deployment infrastructure. It has enough resources to run the Integration and Qualification as well. The Altice labs infrastructure for SONATA pilots is able to play the following roles:

- run a permanent public SONATA Service Platform - SP' URL at Aveiro PoP is: `http://sp.alabs.sonata-nfv.eu/` (IP address: 194.65.138.97; use your Github credentials to logon).
- provisioned VIM for the SONATA master SP located in Athens PoP.
- instantiation of vSA (virtual Security Appliance) NS.
- deployment of vCC (virtual Cache Content) VNF on behalf of vCDN NS running at Athens PoP.

2.4.3 Paderborn Testbed

The Paderborn testbed runs an installation of the SONATA service platform but does not offer infrastructure for service or VNF deployment. It is hosted in Paderborn University's vCluster running VMware vSphere 5.1. Figure 2.14 shows the testbed setup running at UPB.

2.4.3.1 Service Platform VM

The service platform VM has the following properties:

- 8x vCPU

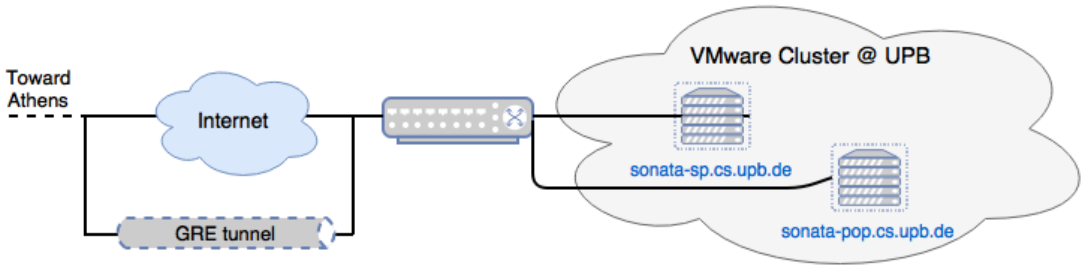

Figure 2.14: Service platform deployment inside UPB testbed

- 16 GB RAM
- 120 GB HDD
- Ubuntu 16.04 LTS
- Host: sonata-sp.cs.upb.de

2.4.3.2 NFVI PoP VM

The NFVI PoP VM has the following properties:

- 8x vCPU
- 32 GB RAM
- 200 GB HDD
- Ubuntu 14.04 LTS
- Host: sonata-pop.cs.upb.de

2.4.3.3 Connectivity

The service platform installation is directly accessible over the Internet (no firewall etc.) via the German DFG network through a 2GBit/s link. The interconnection to testbeds operated by other partners is established using GRE tunnels.

2.4.3.4 Role in SONATA

The Paderborn University testbed differs from the other testbeds in the project because it offers only limited NFVI PoP facilities running in a VM. Even though this limits its usability for some test cases and experiment setups because of limited performance, it can still be used to validate some important properties of the SONATA architecture. As a result, the main use case for the Paderborn testbed is the HSP scenario in which the testbed is used to host a "master" service platform that controls "slave" platforms deployed in testbeds operated by other partners. This proves the deployability of the SONATA platform across spatially distributed facilities. It also shows how the SONATA approach can be used to implement virtual provider scenarios in which a virtual provider sells service platform capabilities but does not necessarily have its own NFVI available.

2.4.4 Tel-Aviv Testbed

Nokia's testbed will host a SONATA instance that will manage remote PoPs (it does not offer infrastructure for service or VNF deployment). It is hosted at Nokia IL (CloudBand) DMZ Lab. Figure 2.15 shows the testbed setup running at Nokia IL.

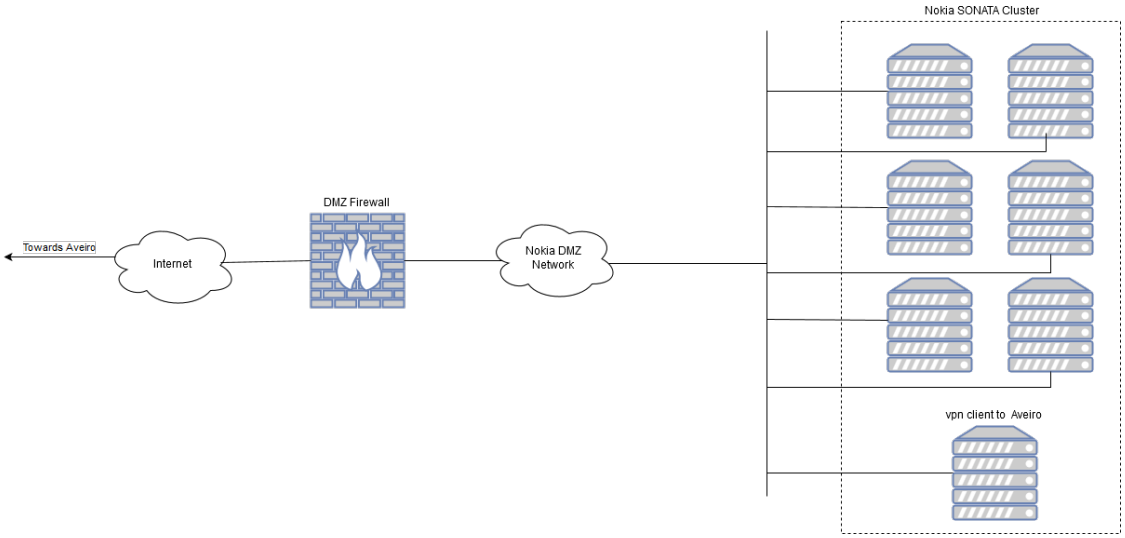

Figure 2.15: Service platform deployment at Nokia IL

2.4.4.1 Available Infrastructure

Nokia IL DMZ lab offers the following infrastructure for SONATA:

- 4 computes (servers) - HP ProLiant SL390s G7.
- CPU - 36 vCPU.
- RAM - 96GB.
- Storage - 20TB.

The server hardware is based on a HP ProLiant SL390s Generation 7 servers [11] with the following specification:

- The Nokia infrastructure is running Openstack as the Infrastructure Manager [18]. The OpenStack Installation on CloudBand CBIS distribution, supporting OpenStack Liberty [20].

2.4.4.2 Connectivity

- External access to this environment is possible via a 10G uplink internet connection using OpenVPN.
- The instance will be connected to the Athens Testbed using a VPN.

2.4.4.3 Role in SONATA

Similar to Paderborn University testbed, also Nokia's testbed does not provide infrastructure to deploy network services or VNFs. It will be used to validate some important properties of the SONATA architecture and functionality.

2.4.5 London Testbed

The UCL testbed located in London, United Kingdom, is based on the following bare metal server list:

- 1 Dell R730 servers with 1 Intel Xeon E5-2680 v3 (12 cores) running at 2.5GHz, with 192 GB memory, and 1 X 600 GB disc.
- 5 servers with 4 Intel Xeons E5520 (16 cores) running at 2.27GHz, with 32GB of memory, and 1 X 146 Gb disc.

In the context of SONATA, the testbed is used for experiment and testing for multi VIM support, and in general the interface between SONATA SP and the VIM. The testbed will be also configured as a Docker based NFVi-PoP.

2.4.5.1 Physical Infrastructure view

The physical network topology of UCL testbed is shown in Figure 2.16:

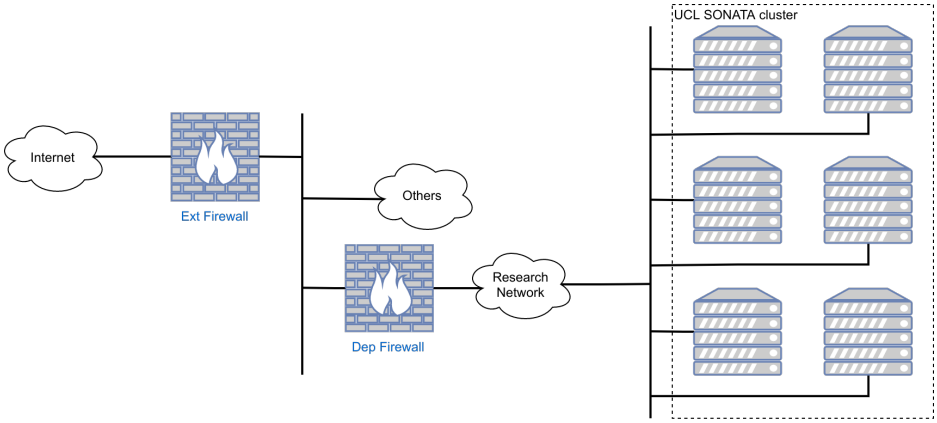

Figure 2.16: UCL testbed physical topology

2.4.5.2 Logical Infrastructure view

The logical network topology of UCL testbed, together with its interconnection to the overall SONATA testbed is shown in Figure 2.17:

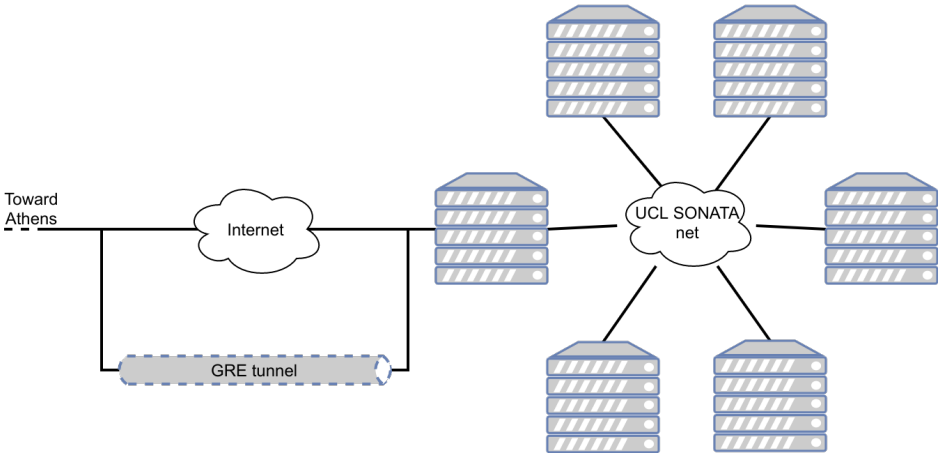

Figure 2.17: UCL testbed logical network topology

3 SONATA Deployment Recipes

In order to support the community involvement for the proper and efficient deployment of the SONATA framework as a whole, this section provides guidelines and lessons learnt from the deployment process. The content and the recipes for automatic deployment either via Ansible scripts or readily prepared virtual images of the components will be enriched an included in the SONATA web site.

In the above context two configurations are considered. The first one (i.e. Fully Virtualised) is for those that do not have the available hardware to install the components, while the second one (i.e. Bare Metal) is for those that have the means to install on a number of servers.

3.1 Virtualized Deployment I

This is the baseline deployment for SONATA, requiring the least amount of hardware resources. This setup is proposed for the developer that would like to use SONATA for instantiation of simple Network Services and focus mostly on either the SONATA Service Platform, or develop and validate simple functionalities related to FSM and SSM as well as VNFs.

In this context this deployment recipe installs the whole SONATA SP, a NFVI-PoP and 2 end nodes, one used as a source and the other used as a sink. The deployment is illustrated in Figure 3.1

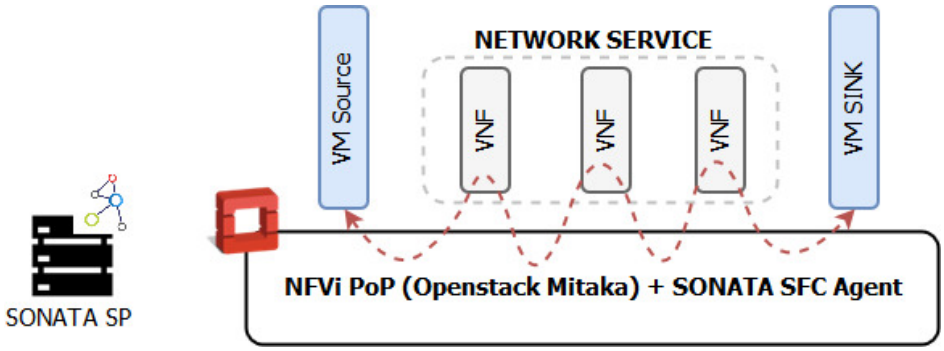

Figure 3.1: Fully virtualized deployment

A provided example Network Service, allows the interconnection of the two end nodes through the a testing VNF that simply forwards the packets between its interfaces. In the final version the deployment recipe will provide a series of simple Qualification Tests in order to validate the deployment.

Table 3.1 enumerates the components deployed with this recipe and the required hardware resources.

Table 3.1: Deployment of required resources

Component	Description/Function	Number of instances	Hardware specs	Guidelines
NFVI-PoP	Openstack Mitaka (single node installation) plus SONATA SFC component for VIM	1	1VM 8 CPU Cores, 32GB Ram, 2 NICs, 200GB Storage, nested virtualisation on the server	https://github.com/sonata-nfv/son-sp-infrabstract/blob/master/vim-adaptor/utils/README.md
Service Platform	The SONATA Service Platform	1	2 CPU cores, 8GB RAM, 20GB storage	https://github.com/sonata-nfv/son-install/blob/master/README.md
Test VNFs	Network Service descriptors and VNF	> 1	1 CPU cores, 1GB RAM, 20GB storage	https://github.com/sonata-nfv/son-qual/tree/master/qual-1VNF-1PoP/README.md
Sinks and Sources	VMs realising the edge nodes	2	min 2 VMs (m1.small flavour in case of Openstack) that will be used for testing and validation interconnected via the switch	For testing purposes, ping and iperf can be used.

3.2 Virtualized Deployment II

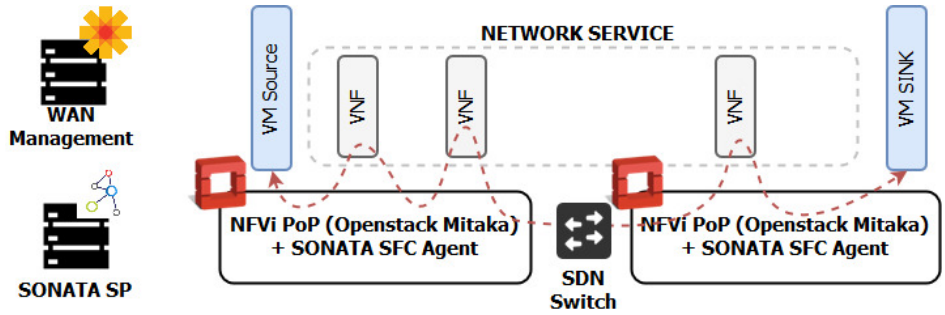

Figure 3.2: Medium size virtualized

As illustrated in figure Figure 3.2, the proposed deployment expands the previous version by incorporating more NFVI-PoPs plus the WAN part (in this deployment emulated by an OVS running in a VM) managed by SONATA WIM. The deployment allows the creation of a small deployment of SONATA framework for developer teams in order to be able to use the SONATA qualification environment and also include in their experimentation end-to-end networking concepts (i.e. SDN WAN). Table 3.2 enumerates the components and their requirements.

In detail this deployment comprises:

Table 3.2: Qualification components and requirements

Component	Description/Function	Number of instances	Hardware specs
NFVI-PoP	Openstack Mitaka (single node installation) plus SONATA SFC component for VIM	>= 2	1VM 8 CPU Cores, 32GB Ram, 2 NICs, 200GB Storage
Service Platform	The SONATA Service Platform	1	2 CPU cores, 8GB RAM, 20GB storage - may run on a virtualised environment i.e. VMware or Openstack
SDN Switch	An OVS installation on a VM	1	Supporting OF > 1.3

Component	Description/Function	Number of instances	Hardware specs
WAN Management	Deployed components: WIM, OpenDaylight	1	1VM 1 CPU core, 8GB RAM, 30GB storage (co-located with SDN Switch)
Test VNFs	Network Service descriptors and VNF	> 1	VNFs that allows packet forwarding. 1 CPU cores, 1GB RAM, 20GB storage.
Sinks and Sources	VMs realising the edge nodes	2	min 2 VMs (small flavour in case of Openstack) that will be used for testing and validation interconnected via the switch

3.3 Bare Metal Deployment

The Bare Metal deployment allow the creation of an actual, physical deployment of SONATA framework, including PoPs and WAN/SDN networking. As such provides better expandability and is more suitable for testing of complicated VNFs and NSs. Table 3.3 enumerates the components and their requirements.

Table 3.3: Bare Metal: components and requirements

Component	Description/Function	Number of instances	Hardware specs
NFVI-PoP	Openstack Mitaka (single/multinode installation) plus SONATA SFC component for VIM	>= 2	Xeon/i7 8 CPU cores, 32GB RAM, 200GB Storage Depends on the open stack deployment mode
Service Platform	The SONATA Service Platform	1	2 CPU cores, 8GB RAM, 20GB storage - may run on a virtualised environment i.e. VMware or Openstack
SDN Switch	An SDN capable SDN switch (i.e. Pica8)	1	Supporting OF > 1.3
WAN Management	Deployed components: WIM, OpenDaylight	1	1 CPU core, 8GB RAM, 30GB storage
Sinks and Sources	PCs realising the edge nodes	2	min 2 PCs that will be used for testing and validation interconnected via the switch

3.4 SONATA Deployment: Lessons Learned

In order to assess the external usability of the developed SONATA software, involving both SDK and SP components, PhD student Janos Elek from ELTE (Hungary) has tested different aspects in the first quarter of 2017: i) software installation, ii) software configuration, and iii) software usability targeting the implementation of a virtual Evolved Packet Core service on top of the SONATA platform.

This section summarizes the resulting findings. First we describe the targeted service which the user intends to develop and deploy (target), next we describe the overall impression of the user on the SONATA architecture, as well as comments regarding individual components, before going to the concluding remarks.

3.4.1 vEPC service to be tested on SONATA

Rather than focusing on the development from scratch, two existing EPC software implementations were considered:

1. OpenAirInterface EPC implementation (http://www.openairinterface.org)
2. vEPC implemented at IIT Bombay (link)

Some details related to the vEPC background is available at the ANNEX.

The main goal of this assessment is to wrap the existing software into a SONATA package using the SONATA SDK tools, which can be deployed on both the SONATA SDK emulator, as well as on the SONATA SP.

3.4.2 Global comprehensibility of the SONATA software components

Given the documentation available in README files on GitHub, together with the available deliverables from WP2 to WP5, the user has sufficient information available on the global structure of the project (although a more concise write-up might be beneficial). The following characteristics are perceived as very positive:

- Micro-service based architecture makes it easy to customize the different layers.
- Decoupling of states in databases (postgres, mongo) allows to test modification for the layers easily. For example, one does not need to reconfigure a VIM every time the infrastructure abstractor or the gatekeeper is restarted.
- Support for docker and docker-compose based installation is great, allows shorter development cycles.
- YAML description of packages (the schema of the description) is well-formulated, easy to use and understand.
- The usage of Jenkins is a good groundwork.

3.4.3 Global installation impression

As the external user did this exercise during the transition of SONATA release to SONATA release 2, several components were not fully compatible in this transitional phase. As a result the installation required a number of manual fixes, which made the process somehow painful. An easy solution to fix such a situation is to clearly indicate which versions of which components are well-able to cooperate successfully together.

Below, we shortly summarize the external usability findings on individual SONATA components.

3.4.3.1 son-install and global installation

The 1.0 version from Docker hub, was not perceived as stable at the time of testing. A public IP variable was hard coded in the implementation. This was not documented, that this needed to be changed. An issue was tracked, and meanwhile this has been fixed.

The postgresql database installation of the GK and the VIM did not automatically install correctly, and the user had to fix this manually.

As the global installation process of son-install had too many (small) issues, the external user decided to use the master versions of individual components of GitHub and manually install them. In many cases, this worked successfully as Docker scripts are available for most of them (resulting from the CI/CD process).

However, the dependency on the SONATA docker registry are perceived as blocking, as they are not always containing the latest component versions.

3.4.3.2 SDK: son-cli and son-emu

As the 1.1 version of son-cli has significant changes compared to the previous version, there was no support from son-install at the time of the testing. A side result is that the compatibility with the GK API was also broken, requiring the user to manually figure out which version at that time was compatible with the installed GK.

The lack of a feature to delete packages from son-cli either in the emulator or SP, is perceived as a drawback, as it forces the user to make new versions of his service all the time.

The emulator installation and usage worked flawless, and was perceived as very valuable. As a prior user of mininet, the interface also felt familiar. A feature which would be very helpful for the external user, would be the ability to play with SSMs, especially to ease the configuration process of a service consisting of VNFs which need information about each other.

3.4.3.3 SP: son-gkeeper and son-sp-infrabstract

At the time of the testing, the code of master branch of son-gkeeper was not always fully operational (consisting of syntax errors). A better procedure to avoid code errors would be advisable. Although the Jenkins procedures indicated successful test outcomes (e.g., integration with the GK), the tests were insufficiently detailed to catch certain errors.

The external user was able to successfully install the OpenStack VIM connecting to the SP, but only after checking and adapting the code manually regarding necessary parameters (e.g., names of ports and links, DNS server configuration, OVS configuration). This should be documented in a clear way.

3.4.4 Generic process aspects

If external users, such as Janos, are trying to debug the software themselves, the process to rebuild the software (and performing tests for integration, etc.) is not easily supported locally, as there are several dependencies on the dedicated CI/CD infrastructure of SONATA itself. As a result the external developer needs to push all changes to the central repositories, and wait for new images to be build and tests to be performed. Ideally the building and testing procedure of the CI/CD process could be performed quickly on the local user laptop, to speed up the external developer iterations in the code.

3.4.5 Concluding remarks

- The global architecture, project structure and workflow is very much appreciated and comprehensible for external users.
- The installation process of individual components using Docker is generally well, but should be better updated/integrated into son-install.
- Documentation might be bit more concise here and there, and if parameters need to be changed manually, this should be clear.
- If components only work with particular versions of other SONATA components, this should be clearly documented.
- The process for external developers should be made more friendly to local development and testing.

In order to solve the above glitches noticed in this exercise the project will refine and enrich the currently available documentation of the provided code base for the deployment of SONATA framework (SP, SDK, infrastructure, environments, etc). This task will involve two parts: (i) Expanding and completing the documentation available in the Github repository of the project, essentially enriching the available Readme files and (ii) Provide documentation and deployment recipes for the installation of virtual or physical infrastructure for the full SONATA framework deployment.

4 SONATA Pilots

The Pilot selection methodology was discussed in Deliverable D6.1. Taking into account the results of its application along with the comments from the Y1 review, the Pilots to be demonstrated by SONATA are:

- Virtual Content Delivery Network (vCDN)
- Personal Security Application (PSA)
- Service Platform to Service Platform (SP2SP)

The selected pilots' demonstrations will cover all the SONATA validation aspects from the validation of functional requirements to non-functional, plus verify the initial promise to provide a CI/CD framework for TELCOs to support multiple verticals and business scenarios. The added value of each Pilot is analysed within related subsections. The mapping of the pilots to the original objectives and coverage of SONATA aspects are discussed in Section 5.

4.1 Virtual Content Delivery Network Pilot

As presented previously in D2.1 (initial use case discussion) and D6.1 (pilot discussion), this use case focuses on showcasing and assessing the SONATA system capabilities in order to enhance a virtual CDN service with features like elasticity and programmability. The business case of Content Delivery Networks is well established in the current telecommunications environment. A series of business relationships are affected by various deployment scenarios that are possible within the current setting. SONATA is building upon the aforementioned status in order to allow for an enhanced vCDN service, focused around enablers provided by the Service Platform that allow high levels of programmability and flexibility. Two scenarios are anticipated for this pilot, namely:

- Classic vCDN mode: Content originates from a single content provider or multiple ones, distributed across the vCaches and eventually delivered to a huge number of subscribers. This scenario will be used to highlight placement and scaling functionalities of the SONATA SP.
- User Generated Content (UGC) based vCDN mode: Content also originates from the end-users (allowing various sub-cases of social networking content exchange). The SONATA SP allows the flexibility of dynamically extending the vCDN service, accommodating additional sources from alternative Content Providers. The twist of this scenario is that the UGC content is identified and cached at the edges, allowing resource optimisation at the edges. This scenario, reveals the interaction of the Service Platform with information that stems from the network (either as traffic information or content information or end-user information) in order to dynamically configure and optimise the CDN for an improved user experience.

An extension to the above functionalities can be seen by the introduction of an additional functionality for the vCDN. As non-linear editing tools normally produce non-adaptive MP4 media,

which need to be transcoded and segmented to provide best user experience (QoE) for the available bandwidth, the vCDN service will optimise the QoE for the End Users of the service by introducing in the forwarding graph of the service a vTranscoder.

SONATA, through the SSM/FSM structure and the DevOps approach, allows developers to reuse and ingest external sources and components in addition to theirs, for the development of a functional composed Network Service. In this context the SONATA SP offers the unique capability to have a fully composed service allowing interaction between the various VNFs in order to enrich their functionalities and implement value added NS. To our knowledge, no other Orchestration Platform that assumes composition of NS with third party VNFs offers this capability. In this context, it is interesting for SONATA to implement and demonstrate this particular Use Case. Assessing the implementation feasibility of the discussed Use Case, most of the components that are considered will not be developed from scratch rather than readily available VNFs or modified versions of existing ones will be used. On the contrary, SSM and FSM plugins need to be be developed as the SONATA paradigm affects the way a Virtual Network Function Manager (VNFM) or a Service Orchestrator will be implemented. So instead of a consolidated management SONATA uses plugable functionalities that in total provided those proposed by NFV MANO is proposing.In this view, the implementation is realistically feasible.

The overall deployment of vCDN Use Case is illustrated in Figure 4.1. Groups of End Users are connected at the edges of the network (connectivity is out of the scope of SONATA), some End-Users are also able to generate content (UGC-green group). In the same figure, the upper orchestration and management layer is illustrated. For the whole service, an SSM is used in order to manage the placement of new VNFs as well as any service scaling or re-configuration decisions. In addition, for each VNF (i.e. instances located at various PoPs), FSM plugins deal with the placement, reconfiguration or scaling of the VNF Components (VNFC) of each VNF. For example, additional instances of Virtual Traffic Classifier (vTC) DPI engine might be required to cope with the traffic load at certain PoPs. At the same time, new edge locations demand the deployment of additional vCDN VNFs (i.e. vCaches). All the interactions are driven by the monitoring and traffic analysis capabilities at the locations part of the Network Service. The metrics are both generic (i.e. CPU, interface traffic, memory, etc.) and VNF specific (i.e. hit ratio, content classification, etc.).

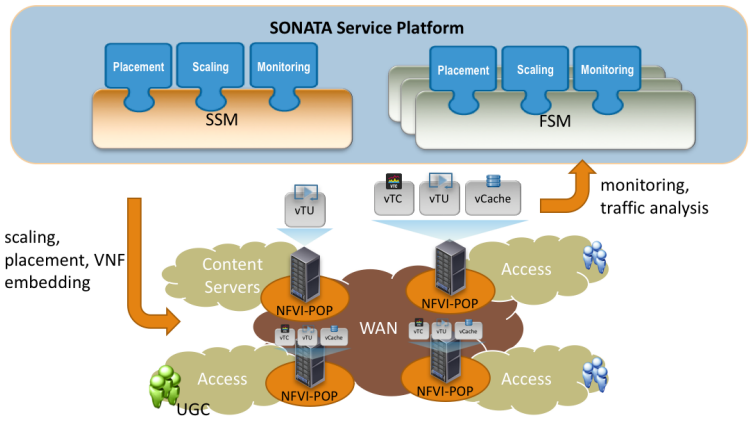

Figure 4.1: vCDN network service deployment

4.1.1 Deployment Scenarios

In the following subsections the various deployment scenarios for vCDN are discussed.

4.1.1.1 Scenario 1 - Network Service reconfiguration

- The NS is instantiated by the SP on top of the already provisioned network slice. (Assumption 2).
- SSM and the FSMs are instantiated.
- SSM placement plugin is deciding on the proper placement of the VNFs to the available NFVI-PoPs, taking into account the explicit placement and also resource availability.
- The VNFs are instantiated and signaling to the FSMs is established.
- The SFC is established and traffic from the content servers is now passing through the deployed VNFs.
- Content is now received on end users' terminals.
- monitoring information is collected by the VNFs and the infrastructure elements.
- alerts are issued by SONATA monitoring framework and collected by the ssm-mon plugin (SSM Monitoring plugin).
- In another end point of the network service, end users start to consume content however there is no vCache for them.

Case 1 : Manual triggering of the new placements

- Customer decided that a new vCache is needed so he reconfigures his service.
- SSM Placement plugin receives a request (external) that a new vCache is to be instantiated at a prefered (explicit) location to accommodate the traffic at the edge.
- The ssm-place (SSM Placement) commands the instantiation of the new VNF and modifies the established chain.
- The new SFC might have two branches originating from the same Content Servers segment and directed to two edges PoP2 and PoP3.
- The service at the edge where the new users reside is a) better quality or b) just available depending on the objectives set by this scenario.

Case 2 : Automatic triggering of new placement

- Alerts are received by ssm-mon (related to usage of the service on all branches towards the edge locations).
- ssm-place detects that at a particular edge new users are connected, thus increase of the aggregate traffic towards that particular edge is detected.
- ssm-place checks if that edge is served locally by a vCache.

- When alerts surpass the configured threshold, automatic placement of vCache is requested.
- The PoP in proximity to the edge location is identified (ssm-place).
- the SSM coordinated the required lifecycle operations in order to deploy the vCache and update the NS.

4.1.1.2 Scenario 2 - Scaling

- New load is gradually introduced at some of the edges of the provisioned slice.
- FSM for the running VNFs at those locations sends alerts for certain metrics that are used for triggering the scaling lifecycle either at VNF or NS level.
- SSM receives request for certain actions regarding to the scaling, i.e. by requesting the permission to spawn an additional vNFC for scaling out the VNF, or by instantiating a new VNF in order to load balance the traffic at certain edge locations.

4.1.1.3 Scenario 3 - User Generated Content Classification

- NS is instantiated, and VNFs are deployed in all network edges where end-users and content servers reside.
- In one of those edges end-users turned into content providers share content (e.g. live coverage of a live event).
- Offered content is requested from an increasing number of end-user, spanning beyond the local ones.
- information on the requests per second for the same content is monitored by the vTC located at each network edge.
- however the UGC streams are not getting cached into the vCaches.
- Since the SSM NS reconfiguration plugin is not yet activated the received QoS of the UGC content fluctuates depending on the network status for some end-users.
- A new SSM plugin that allows reconfiguration of NS subject of UGC flow information is installed.
- When then number of req/sec reaches a configured threshold the SSM is notified.
- SSM (reconfigures) instructs that the UGC content is cached in the vCaches available at the edges in order to facilitate the delivery of the content with optimal QoS.
- VTC redirects identified UGC traffic to the local vCaches.
- Local content is cached and optimal use of resources is achieved as the connections of the PoP to the content servers is less utilised.

4.1.1.4 Scenario 4 - QoE enhancement

This scenario is an extension of the vCDN service including a DASH transcoding unit. The transcoding functionality can produce new content per combination of elements (available bandwidth, terminal information, etc.). By choosing the best suitable transcoding and segmentation, it ensures the best Quality of Experience (QoE). The vTranscoder will be exploited on-demand according to the situation and customer needs.

- Upon user request of a content format or quality that is not available
- vTC forwards request to the vTU
- vTU ??transcodes?? the content based on the user request
- As soon as the initial segments are transcoded, they are made almost immediately available to the content server
- Upon new request for the new content format the content server streams the content to the user.
- vTC is monitoring the whole process.

4.1.2 Virtual Network Functions of vCDN Pilot

The vCDN Network Service is built of the following VNF's:

- vCC virtual Content Cache
- vTC virtual Traffic Classifier
- vTU virtual Transcoding Unit

4.1.2.1 Virtual Content Cache (vCC)

vCC is a virtual Content Cache server. Despite supporting most authentication backends, on the scope of vCDN it acts as a transparent proxy: it deliver content without user authentication, i.e., behaves as a pure caching server.

The vCC for vCDN is available in two flavours:

- QCOW2 image ready to deploy on a provisioned Openstack VIM
- Docker image on Github or SONATA Registry

Along with the instantiation of the vCC VNF, it must then mount an external volume used for the cache storage.

Architecture

The main functionality of the VNF is provided by a single VNFC running squid proxy. The proxy is configured in transparent configuration. This way there is no need for configuration of the end-user devices. As Figure 4.2 illustrates, End-User requests are delivered at connection point 1 (CP1) while the requests made by the proxy and the incoming content are going through via CP2. The last connection point (CP0) interface conveys management information. Steering decisions are based on the configured policies at the vTC.

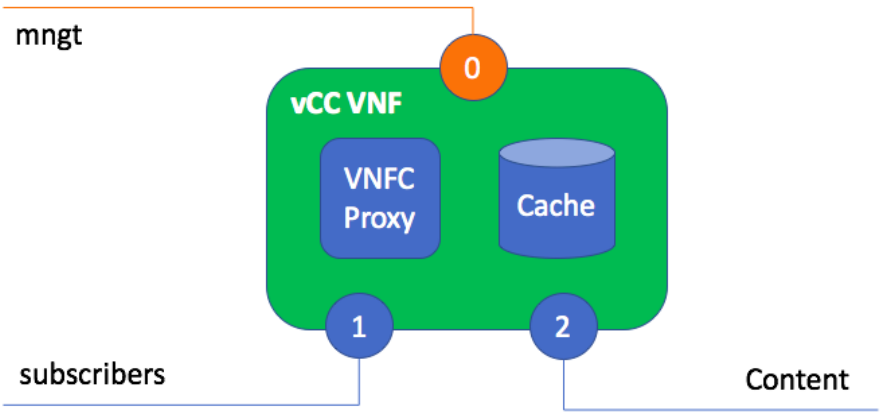

Figure 4.2: Virtual Content Cache architecture

VNF Descriptor

The initial VNF descriptor is provided in this document ANNEX and is subject of further experimentation and refinements. At least one of the data_plane interfaces that vCC provides (i.e. Content) should be assigned a static IP address with access to port 3128 (TCP), where the PROXY is listening. Additionally, a Management interface is also assigned static public IP in order to be reachable by the FSM.

Lifecycle Management (FSM)

An FSM will be developed for this VNF in order to support basic lifecycle events for the operation specifically for this VNF. In order mostly to highlight the SONATA plugin infrastructure and less to implement new algorithms or functionalities for this particular component, support for common events for all the VNFs will be implemented once and adapted to serve all the VNFs. For the vCC the following functionalities will be covered by FSM plugins (see Table 4.1)

Table 4.1: vCC FSM plugins

FSM Plugin Name	Description/Function	Unique	Mandatory
fsm-config	Send initial configuration required for communication with VNFCs and management/monitoring configuration	No	Yes
fsm-start	Start the functionality promised by the VNF	No	Yes
fsm-stop	Stop the functionality promised by the VNF	No	Yes
fsm-mon	Receive VNF specific metrics in case the developer asks for direct access to the monitoring offered by the VNF, configuration of metric collectors	No	No
fsm-pol	Send policies for the vCC (i.e. user groups, flushing, caching strategy etc)	Yes	No
fsm-fetch	Initiate pre-caching of content from the content servers	Yes	No

Technologies

The component used in the PSA pilot is call vPROXY and is based on Squid open source proxy (aka, web content acceleration and authorization service).

4.1.2.2 Virtual Traffic Classifier (vTC)

The proposed Traffic Classification solution (vTC) is based upon packet inspection techniques that vary from shallow to deep packet inspection. The vTC uses heuristics and specific application protocol dissectors that allow for fast traffic identification. In order to make the functionality of the vTC seamless for the actual forwarded traffic, two possible deployments can be supported. The first one useful for monitoring purposes is to mirror original traffic towards the vTC, the identification process this way does not interfere with the traffic forwarding, hence no latency is introduced. The second deployment case is in-line to traffic. In this deployment latencies in forwarding cannot be avoided, however the trade-off is access to functionalities as traffic classification, Constrained Based Routing etc. In order to alleviate the latencies, use of direct path technologies may be employed (i.e. SR-IOV) or other software based acceleration (i.e. DPDK). In addition the analysis is flow based, i.e. a small number of initial packets from each flow is used in order to identify the traffic. After the flow identification step no further packets are inspected. The Traffic Classifier follows the Packet Based per Flow State (PBFS) in order to track the respective flows. This method uses a table to track each session based on the 5-tuples (source address, destination address, source port, destination port, and the transport protocol) that is maintained for each flow.

Architecture

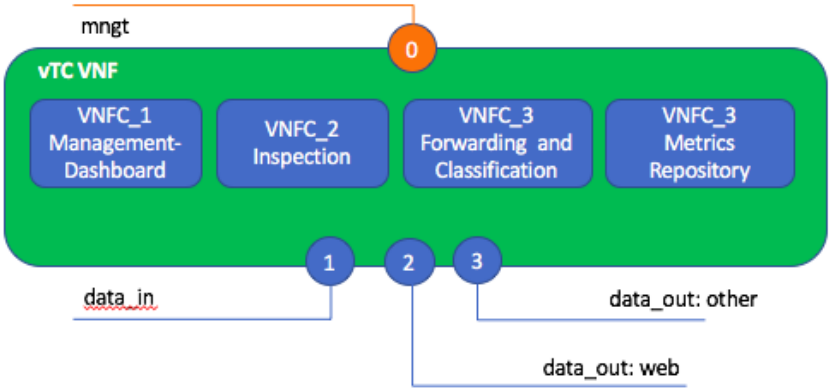

Figure 4.3: vTC architecture

Originally the vTC full fledged implementation comprises of four VNF components as illustrated in Figure 4.3.

- Management dashboard - This VNFC provides the Monitoring Dashboard, exposes an API for external access and data sharing. Furthermore via this VNFC a web GUI is exposed allowing direct monitoring and metric selection. The VNFC uses visualisation tools that query the Metrics Repository in order to provide the monitoring data.
- Inspection - This VNFC implements the filtering and packet matching algorithms and is the necessary basis to support additional forwarding and classification capabilities. It is a key component for the successful implementation of the vDPI and the most computationally intensive. The component includes a flow table and an inspection engine.
- Forwarding and Classification: This VNFC handles routing and packet forwarding. It accepts incoming network traffic and consults the flow table for classification information for

each incoming flow. Traffic is forwarded using default policies until it is properly classified and alternate policies are enforced. It is often unnecessary to mirror packet flow in its entirety in order to achieve proper identification. Since a smaller number of packets may be utilized, the expected response delay can therefore be close to negligible. In a case where the Inspection, Forwarding and Classification VNFCs are not deployed on the same compute node, traffic mirroring may introduce additional overhead. A classified packet can be redirected, marked/tagged, blocked, rate limited, and also reported to a reporting agent or monitoring/logging system within the network.

- Metrics Repository - This VNFC includes the internal metrics repository that acts as local storage (usually a time-series database e.g. InfluxDB).

The initial implementation currently experimented uses two VNFC in order to provide the same functionalities with the original, minus traffic classification support (i.e. QoS) that is not required by the Pilot. Hence, the vTC VNF comprises of two Virtual Network Function Components (VNFCs), namely the Management Dashboard and the Inspection engine. The aforementioned VNFCs are implemented in respective VMs as illustrated in Figure 4.4.

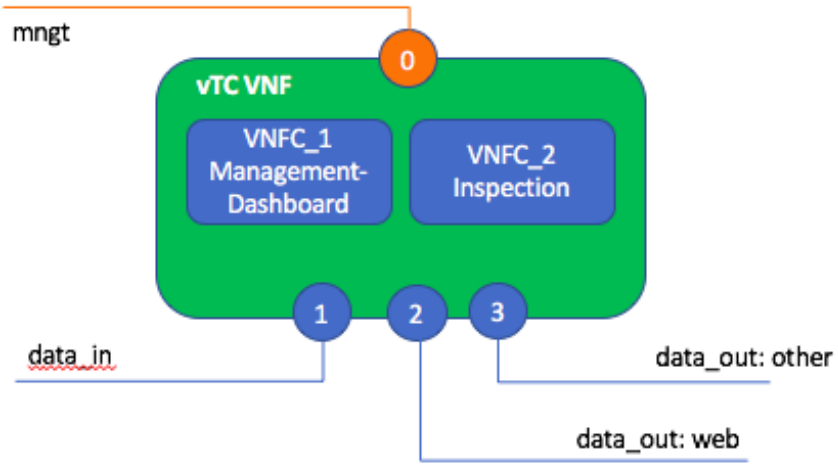

Figure 4.4: Simplified vTC architecture

The Traffic Inspection VNFC is the most processing intense component of the VNF. It implements the filtering and packet matching algorithms in order to support the enhanced traffic forwarding capability of the VNF. The component supports a flow table (exploiting hashing algorithms for fast indexing of flows) and an inspection engine for traffic classification. The same VM is also responsible for routing and packet forwarding. It accepts incoming network traffic, consults the Flow Table for classification information for each incoming flow and then applies pre-defined policies i.e. selection of output interface, chain selection etc. It is assumed that the traffic is forwarded using the default policy until it is identified and new policies are enforced. The expected response delay is considered to be negligible, as only a small number of packets are required to achieve the identification.

The VNF provides four interfaces:

- A management interface used to communicate with the FSM.

- A data_in interface used for the incoming traffic.
- A data_out:web interface for traffic characterised as web traffic that is forwarded to vCC.
- A data_out:other interface for traffic not filtered as web traffic.

VNF Descriptor

The initial VNF descriptor is provided in this document ANNEX and is subject of further experimentation and refinements. The most important fields in the VNF descriptor are related to (i) monitoring metrics and thresholds; (ii) connection points and virtual links description and (iii) floating IP assignment. The later are used by the NSD in order to specify the VNFFG and essentially the NS end-points. Finally management interfaces should be assigned static IP addresses. In the data plane the vTC is considered as a L2 device.

Lifecycle Management (FSM)

The FSM plugins that will be developed for vTC VNF are summarised in Table 4.2.

Table 4.2: vTC FSM plugins

FSM Plugin Name	Description/Function	Unique	Mandatory
fsm-config	Send initial configuration required for communication with VNFCs and management/monitoring configuration	No	Yes
fsm-start	Start the functionality promised by the VNF	No	Yes
fsm-stop	Stop the functionality promised by the VNF	No	Yes
fsm-mon	Receive VNF specific metrics in case the developer asks for direct access to the monitoring offered by the VNF, configuration of metric collectors	No	Yes
fsm-pol	Send policies for the vTC (i.e. protocols to be analysed, collected data, dashboard)	Yes	Yes
fsm-class	Configure and start/stop classification process	Yes	Yes

Technologies

The vTC utilizes various technologies in order to offer a stable and high performance VNF compliant to the high standards of legacy physical network functions. The implementation for the traffic inspection used for these experiments is based upon the open source nDPI library [REFnDPI]. The packet capturing mechanism is implemented using various technologies in order to investigate the trade-off between performance and modularity. The various packet handling/forwarding technologies are:

- PF_RING: PF_RING is a set of library drivers and kernel modules, which enable high-throughput, packet capture and sampling. For the needs of the vTC the PF_RING kernel module library is used, which is polling the packets through the LINUX NAPI. The packets are copied from the kernel to the PF_RING buffer and then they are analysed using the nDPI library.
- Docker: Docker is a platform using container virtualization technology to run applications. In order to investigate the pros and cons of the container technology, the vTC is developed also as an independent container application. The forwarding and the inspecting of the traffic are also using PF_RING and nDPI as technologies, but they are modified to fit and function in a container environment.

4.1.2.3 Virtual Transcoding Unit (vTU)

In the context of the vCDN Pilot, the player will send a request to the vTU when it can not consume the content. That happens when the player can not handle any of the available transcoded formats for that particular content. All the requests the player sends, will pass through the vTC, who in turn will redirect them either to:

- the vTU
- the vCache going towards the content server

When the player request a new transcoding, that request will contain the appropriate resolution (the one the player can handle) and the particular content to be transcoded. vTU will take care of producing a new MPEG-DASH transcoding with the requested resolution. The master content to be transcoded is in the content server. When the transcoding is ready, the new produced content will be stored in the content server, together with all the other transcodifications for that same content.

Architecture

The virtual Transcoding Unit (vTU) VNF is divided in 3 different modules: (i) a Database, (ii) an API REST and (iii) the Transcoder. The database and the transcoder are connected through a REST API. The REST API is the manager of all the petitions for consumption and transcoding of content. The client has to send an HTTP POST to the REST API in order to trigger transcoding sessions, this will add a new transcoding job to the database. The database is used in order to save everything related to the transcoding jobs and content information. Figure 4.5 illustrates the architecture of the vTU and its interfaces with other components. In order that the transfer of the content from the vTU to the DASH server is immediate, the DASH server is exposing storage via NSF, which is mounted over the network by the vTU.

VNF Descriptor

The initial VNF descriptor is provided in this document ANNEX and is subject of further experimentation and refinements. The VNF exposes a single network interface that is used for receiving transcoding requests. Then the content is transcoded and placed in the shared storage. Apart from the connection points and the networking, proper selection of CPU and memory options needs to be done in order to be able to handle multiple concurrent transcoding requests. The case of scaling of the VNF in order to accommodate increased load when required will be explored.

Lifecycle Management (FSM)

vTU lifecycle events will be managed by either the default lifecycle management (i.e FLM) or the plugins described in the Table 4.3. The later will be used in case of supporting scaling at the VNF level.

Table 4.3: vTU FSM plugins

FSM Plugin Name	Description/Function	Unique	Mandatory
fsm-config	Send initial configuration required for communication with VNFCs and management/monitoring configuration	No	Yes
fsm-start	Start the functionality promised by the VNF	No	Yes
fsm-stop	Stop the functionality promised by the VNF	No	Yes

FSM Plugin Name	Description/Function	Unique	Mandatory
fsm-mon	Receive VNF specific metrics in case the developer asks for direct access to the monitoring offered by the VNF, configuration of metric collectors	No	Yes
fsm-pol	Send policies for the vTC (i.e. protocols to be analysed, collected data, dashboard)	Yes	Yes
fsm-class	Configure and start/stop classification process	Yes	Yes

Technologies

The technologies used for the API REST are Node.js + Express.js. MongoDB is used as a database. The Transcoding Unit has been developed with Python. A transcoding job is divided in three steps. Firstly, the original media file is transcoded to MP4 using ffmpeg. Secondly this transcoded file is fragmented to MPEG-DASH with the GPAC multimedia framework. Finally the new resolution, along with other information like the bit rate, is added to the MPEG-DASH MPD file. The vTU in order to transcode the content needs the content as close as possible, this is the main reason why between the vTU and the Content Server there is a folder using Network File System (NFS). The NFS allows to access files over a computer network as if it were local storage access.

4.1.3 Service Orchestration and Lifecycle Operations

SONATA offers refined control over the Service Orchestration and lifecycle operation of each Network Service. In fact although there are simple and default lifecycle operation for all the services, the developer is able to replace them and infuse new operations based on more dynamic orchestration patterns. With this approach the Service Lifecycle Management (SLM), substitutes the default operation in case plugins are available for any given Network Service. The anticipated Network Service Service Specific Management plugins and their functionality are presented in the following table Table 4.4:

Table 4.4: vCDN SSM plugins

SSM Plugin Name	Description/Function	Unique	Mandatory
ssm-config	Send initial configuration required for initial management/monitoring configuration	No	Yes
ssm-start	Start the Network Service	No	Yes
ssm-stop	Stop the Network Service	No	Yes
ssm-mon	Receive monitoring alerts via the message bus, configuration of thresholds and monitoring intervals	No	Yes
fsm-place	Control the placement of NS components based on the criteria expressed by the descriptors and the preferences of the developer	Yes	Yes
fsm-pol	Send policies to the respective FSM plugins for altering the behaviour of the service	Yes	Yes

4.1.3.1 Network Service

The following UML Diagram provides an overview of the operation of vCDN network service.

The physical deployment of a vCDN service is illustrated in Figure 4.7. It can be observed that 3 PoP will be used as end-points for the NS. On one of them only the vTC will be deployed in order to facilitate the Scenario 1. The vTU will be deployed on the root of the NS, close to the Content Server while the vCCs are deployed close to the End-Users. The ssm-place plugin makes sure that these restrictions are met during the initial deployment or any re-deployment.

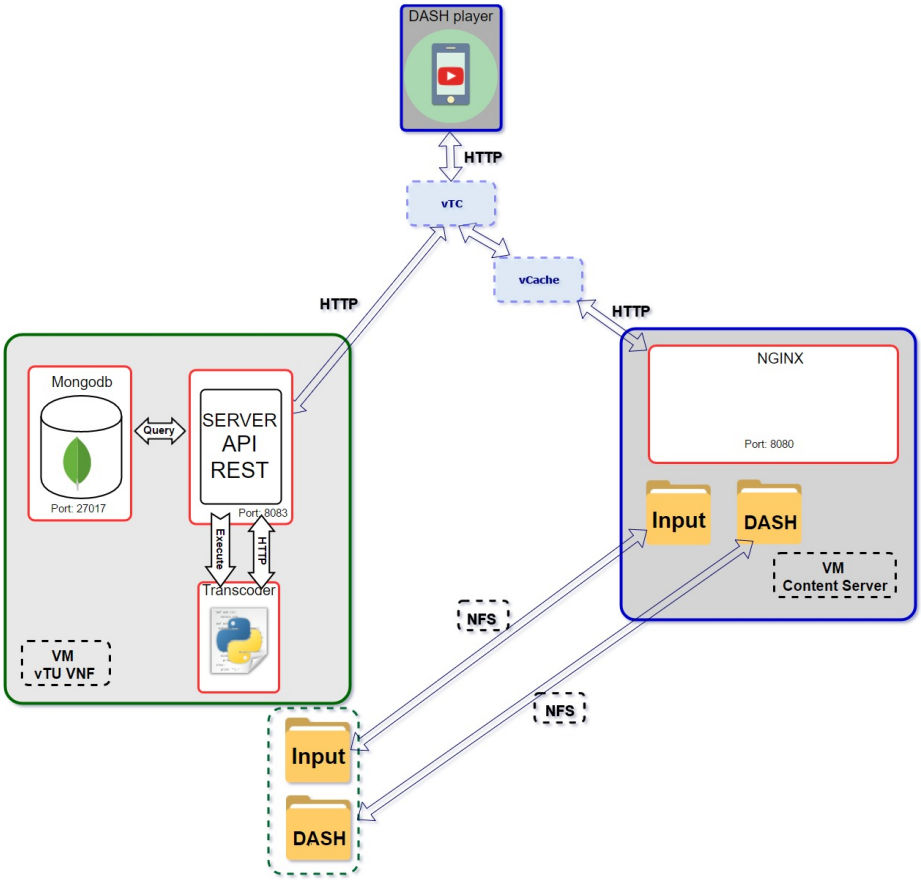

Figure 4.5: Player, vTU and content server

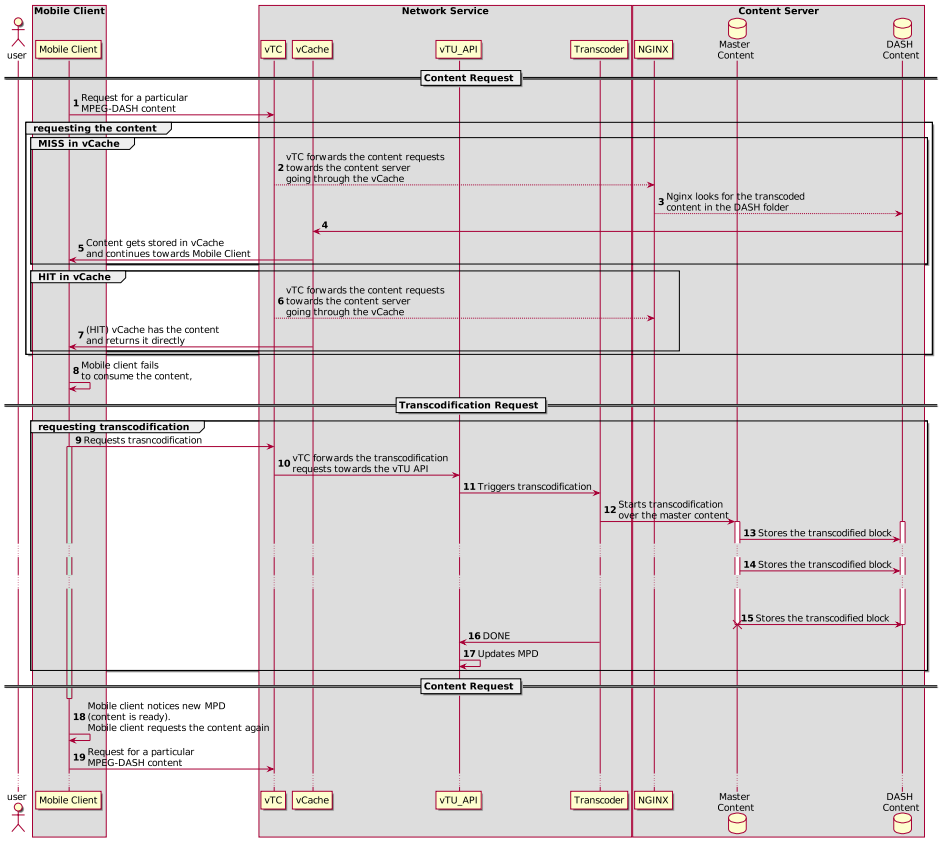

Figure 4.6: vCDN network service

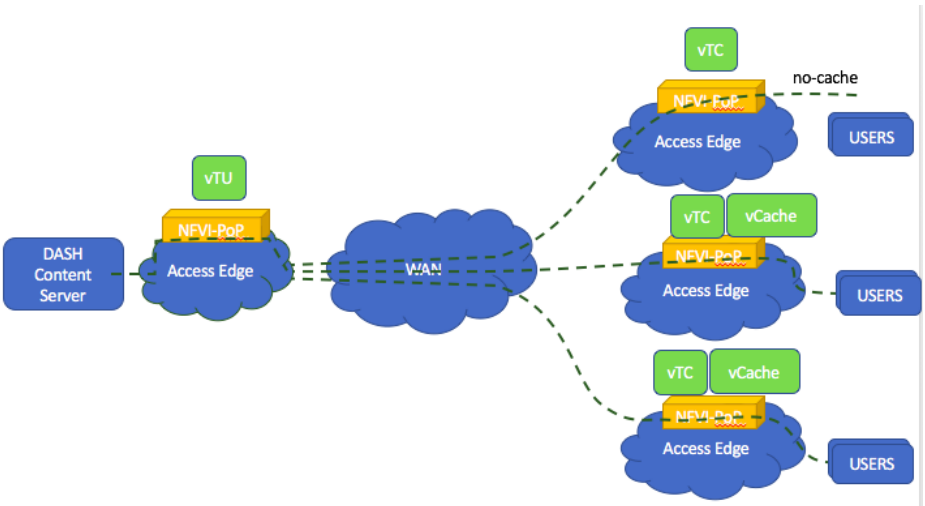

Figure 4.7: vCDN physical deployment

The vCDN Network Service schematic can be observed in Figure 4.8. The interconnection of VNFs in this network service is achieved via three Virtual Links, used for data-plane traffic and one Virtual Link, used for management signalling used by FSM. The direction of traffic for the data-plane in the NS is always uni directional from connection point A to connection point B. The VNF that controls the Forwarding Path or the chain to be followed is the vTC VNF. It should be noted that SONATA does not include in the descriptor files any intelligence that could enable classification inherently supported by the Service Platform. Instead it establishes the chains as those are described by the NSD and the actual selection of which traffic goes to which chain is decided by the vTC or any other VNF that the developer selects to play this role.

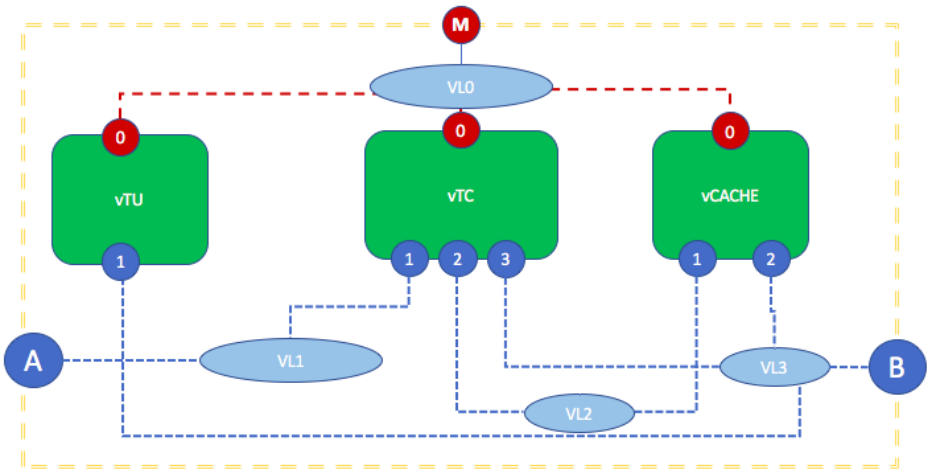

Figure 4.8: vCDN network service

Network Service Descriptor (NSD)

The manifestation of the schematic presented in Figure 4.8 in a script used for the deployment of the service is the NS descriptor (NSD). The descriptor provides information for all the Virtual Links (interconnecting VNF connection points), end-point information of the NS and last but not least the VNF Forwarding Graph that expresses the sequence and conditions for Service Function Chaining (SFC) agent to enforce. Currently the NSD is still under consideration, mainly due to refinements in the service deployment and validations that are on-going.

4.1.4 Preliminary Deployment and Results

The Figure 4.9 illustrates a preliminary deployment configuration for the vCDN NS. This deployment is using Athens POP, at this point is used for finalising the configuration of the NS components (i.e. VNFs, FSMs and SSMs) and refinement of the descriptors for the automated deployment by the SP. Furthermore performance issues related with service specificities and forwarding graph variations are also examined.

In the above context the service has been deployed in Athens testbed, using a single PoP for the deployment of vTC and vCC. The traffic is chained through vTC as already discussed in the previous section. Figure 4.10 below is a snapshot from the End User desktop, depicting the service as consumed by the End User. The Service is provided by an nginx web server using DASH streaming. At the end user side AKAMAI DASH player is used to consume the service.

Figure 4.9: vCDN preliminary deployment

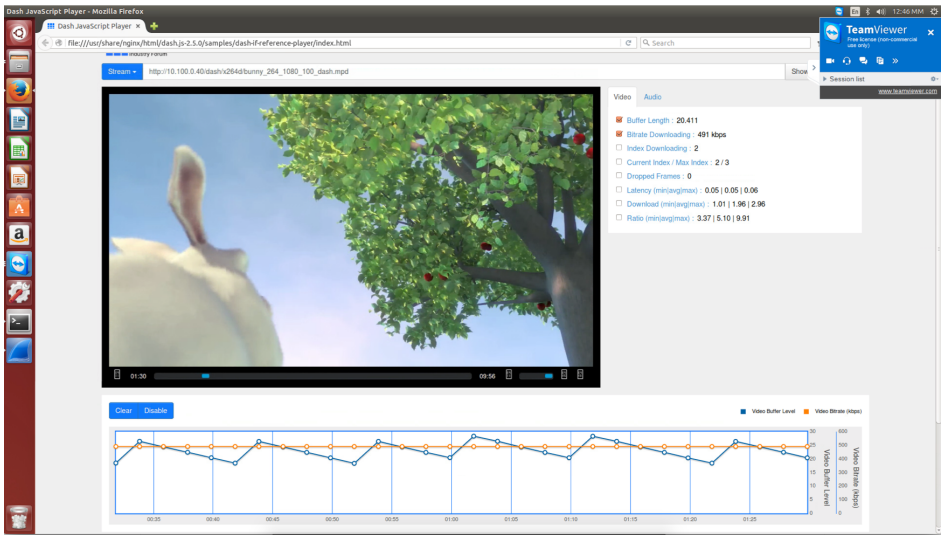

Figure 4.10: End User service consumption

4.2 Personal Security Application Pilot

As presented previously in D2.1 (initial use case discussion) and D6.1 (pilot discussion), the personal security application (PSA) use-case focuses on showcasing and assessing the SONATA system capabilities in order to enhance a service provider based personal security application. To this end, a security application comprising several different security components such as a firewall, a virtual private network service, and an intrusion detection system, is executed in the virtual network infrastructure of the service provider. It is embedded in the data path of a user and assesses and filters its network traffic and thus protects its devices connected to the Internet. Using a self-service portal, a user can connect to the personal security application and adapt the actual composition of the network functions that constitute the service. Thus, a user might add a firewall or an intrusion detection system to its data path on demand.

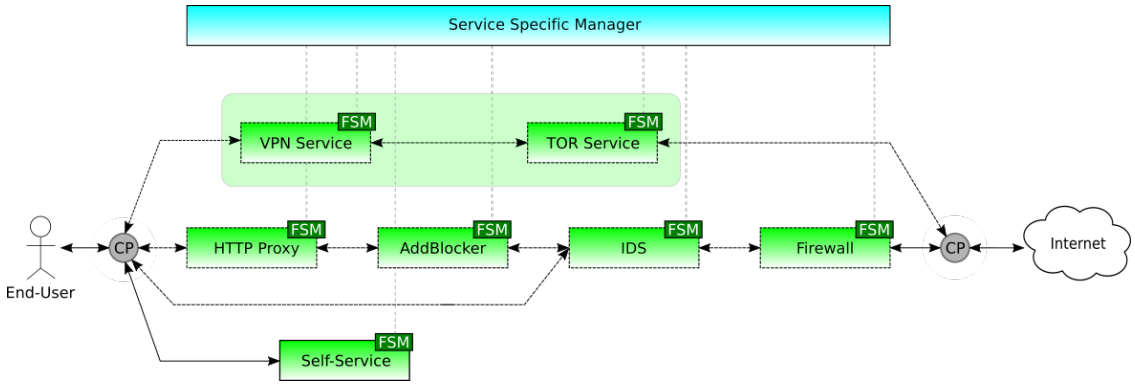

Figure 4.11: The PSA service deployment

Figure 4.11 shows how the PSA network service looks like in its initial setup only comprising the self-service port VNFs (bold lines) and in its full deployment comprising all available VNFs (dashed lines). The user connects to the network service, running on the SONATA Service Platform via a Connection Point (CP). Likewise the Service Platform is connected to the Internet via a CP. All the VNFs ship with a related FSM that connects the VNF with the SSM and the MANO framework of the service platform. Thus, every running VNF can trigger and consume events and share information with the SSM. The self-service portal VNF, for instance, generates a create-new-instance event, when the user wants to add a new PSA component to the data path. This event is propagated to the SSM using the portal-FSM. The SSM reacts to this event by instantiating the requested VNF and modifying the service function chain accordingly. Since we focus on showcasing SONATA Service Platform capabilities rather then VNF capabilities, we made the assumption that there will be one PSA network service instance per end-user. In fact, a network service could also serve multiple users. And, evidently, this also offers room for improvements. However, it would require more complex VNFs.

Assessing the implementation feasibility of the discussed Use Case, most of the VNFs that are considered will not be developed from scratch but available VNFs or modified versions of existing ones will be used. A detailed description of the VNFs can be found in Sect. *Virtual Network Functions*. Moreover, SSM and FSM plugins will be developed in order to implement the required functionalities at the management level. It is worth to note that complementary to the other pilots, the PSA pilot does not focus on the VNFs and their functionalities as such, but uses the PSA use-case to demonstrate features and the flexibility of the SONATA Service Platform.

4.2.1 Deployment Scenarios

In the following we briefly describe the deployment scenarios of the Personal Security Application.

4.2.1.1 Dynamic Network Service Reconfiguration

The goal of this deployment scenario is to showcase the deployment of a network service that can be modified on the fly. To do so, we first define the network service including all possible VNF components. However, at instantiation time, only an initial VNF, i.e. the self-service portal VNF, is deployed in the SONATA Service platform. The network service end-user can connect to the self-service portal and select additional VNFs. Once selected, the additional VNFs will be deployed and the service function chain will be modified dynamically. The SFC modification will be triggered and performed by the PSA SSM.

This scenario includes the following steps:

- On-board the network service on the Service Platform.
- Instantiate the initial service comprising only the self-service portal.
- Provide the portal gateway (IP) to the end-user.
- Let the end-user log in to the self-service portal.
- Let the end-user configure the personal security application by adding additional security VNFs to the service.
- The service platform inserts the added VNFs to the service function chain of the user.
- Finally, the user can use the personal security application, and traffic gets filter or modified accordingly.

4.2.1.2 Scaling

This second scenario extends the first one and introduces elasticity to it. The scaling scenario uses a similar configuration as described for the Dynamic Network Service Reconfiguration scenario. However, now a single VNF, i.e. the AddBlocker VNF, can be scaled out. This scaling can be triggered either automatically by the monitoring system or - for demo purposes - manually using the self-service portal.

This scenario includes the following steps:

- Onboard, deploy, and configure a more complex PSA network service comprising at least the AddBlocker VNF in the data path.
- Create a trigger to scale out the AddBlocker VNFs. This trigger might either be manual, using the self-service portal, or automatic by using the related Function Specific Manager.
- In case of the automatic option, the FSM checks the number of concurrent connections to the AddBlocker and adds additional resources based on the number of connections.
- Using the scaled-out version of the AddBlocker, we show that dynamic scaling may increase the overall performance on the fly - also in a dynamic network service.

4.2.2 Virtual Network Functions of PSA Pilot

In the following, we describe the Virtual Network Functions used in this pilot. We present a brief overview of the VNF architecture and the technologies used by the VNF. Moreover, we briefly describe the Function Specific Managers and their main functionalities in supporting the VNF life-cycle management as well as the pilot as such.

4.2.2.1 Proxy

The proxy VNF, caches requested content in order to accelerate the content delivery when the same content is requested by a number of end-users. Moreover, it can control via user authentication or other filters the access to the content.

The SONATA PSA Network Service implements one service per subscriber. This implies the instantiation of one PSA per client. In this topology, there are a local authentication backend per client and one cache volume per client as well. This suggests a low resources flavour guest machine or container per client and doesn't not require scaling feature.

Architecture

To fulfill the Parental Control capability inside a dedicated PSA box, the following assumptions are made:

- A local password authentication back-end is available (and with local management) to support the subscriber family or employees members.
- A small cache volume is allocated.

This simple scenario is illustrated in Figure 4.12:

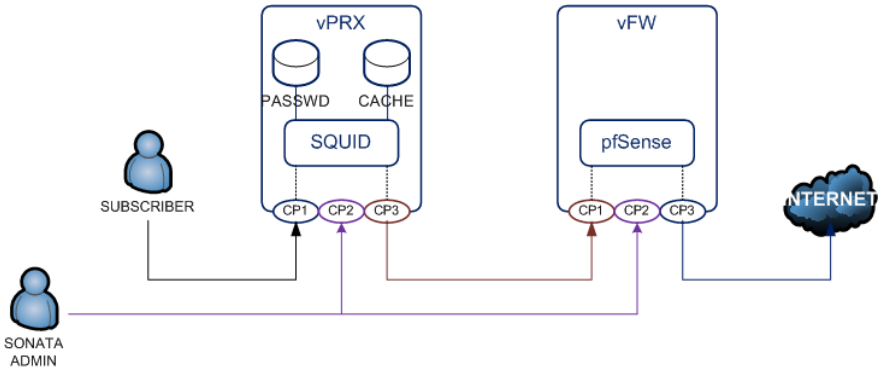

Figure 4.12: The PSA service deployment

An alternative topology is the implementation of one centralized service for all clients based on a shared cache and external authentication backed (e.g., accessed via LDAP protocol). In a large scale scenario (lets say, thousand users), it provides a better cache optimization (due to a shared volume for all content), requiring however scaling features to support pick hours.

Function Specific Manager

The Function Specific Manger for the Firewall VNF takes care of a) the regular life-cycle events, such as starting and stopping the VNF as well as b) the initial configuration of the VNF. Moreover, the FSM communicates with the Service Specific Manager and sends event notifications to the SSM.

Technologies

The proxy service is build on Squid [10], which is a 'de facto' open source tool for the web content caching and filtering. It has a wide variety of uses, from speeding up a web server by caching repeated requests; to caching web, DNS and other computer network lookups for a group of people sharing network resources, to aiding security by filtering traffic.

4.2.2.2 Firewall

In the PSA pilot, the firewall acts as the first line of defense against malicious traffic. Thus, it can be configured to filter out traffic and act as a gateway deciding what goes in and out. The Firewall VNF is based on pfSense [17]. It can be configured via the self-service portal and filters all traffic transparently. By default, outgoing traffic is always permitted, where's incoming traffic is usually blocked.

Architecture

The Firewall VNF is designed as a single-VDU VNF comprising a single QCOW2 image as Virtual Deployment Unit. It requires at least on vCPU core, 1 GB RAM, and 1 GB Storage. The VNF offers 3 external Connection Points, namely: mgmt (management), input, and output. By default, the VNF monitors the CPU utilization, the memory utilization, and the NIC utilization. The VNF Descriptor that specifies the VNF metadata can be found in the annex at Section A.2.2.

Function Specific Manager

The Function Specific Manger for the Firewall VNF takes care of a) the regular life-cycle events, such as starting and stopping the VNF as well as b) the initial configuration of the VNF. Moreover, the FSM communicates with the Service Specific Manager and sends event notifications to the SSM.

Technologies

The Firewall VNF is based on pfSense [17] which is a well-known open-source firewall based on FreeBSD's packet filter (*pf*). PfSense can be easily installed on virtual machines to make a dedicated firewall/router for a network. Moreover it has been noted for its reliability and offering a range of features. To this end, pfSense is commonly deployed as a perimeter firewall, router, wireless access point, DHCP server, DNS server, and as a VPN endpoint. Thus, we can re-use the pfSense image in various occasions. Evidently, this makes it a perfect candidate to act as a VNF in our Personal Security Application pilot.

4.2.2.3 Intrusion Detection System

The IDS VNF is based on Snort [19]. If inspects all incoming and outgoing traffic and raises alarms - to the user and to the FSM/SSM system - whenever it detects any anomaly.

Architecture

The Intrusion Detection VNF is designed as a single-VDU VNF comprising a single QCOW2 image as Virtual Deployment Unit. It requires at least on vCPU core, 1 GB RAM, and 1 GB Storage. The VNF offers 3 external Connection Points, namely: mgmt, input, and output. By default, the VNF monitors the CPU utilization, the memory utilization, and the NIC utilization. The VNF Descriptor that specifies the VNF metadata can be found in the annex at Section A.2.3.

Function Specific Manager

The Function Specific Manger for the Intrusion Detection VNF takes care of a) the regular life-cycle events, such as starting and stopping the VNF as well as b) the initial configuration of the VNF. Moreover, the FSM communicates with the Service Specific Manager and sends event notifications to the SSM. In particular, the FSM emits and forwards events based on triggers from the Snort IDS as such.

Technologies

The Intrusion Detection VNF is based on Snort [19], which is a free and open source network intrusion prevention system. In general, Snort performs protocol analysis, content searching and matching. It can be configured to passively sniff packages, work as packet logger, and actively perform network intrusion detection. In our pilot, Snort acts as an intrusion detection system. It monitors network traffic and analyses it against a rule set defined by the user. If a rule is matched it emits events to the FSM analyses and forwards the events to the SSM that can make Network Service life-cycle decisions based on these events.

4.2.2.4 Virtual Private Network Server

The VPN VNF is based on OpenVPN [15] and is used in conjunction with the Anonymizer Service. Thus both services are always added together. A user can connect to the VPN service to ensure traffic to be encrypted. The VPN forwards all that traffic to the Anonymizer Service which makes sure the user cannot be traced back.

Architecture

The Virtual Private Network VNF offers an easy, encrypted connection point to the Anonymizer VNF. Thus, it is tightly coupled with the TOR VNF and always operates in parallel to that function. The coupling of the two functions is done using the service specific life-cycle management, which is implemented in the SSM. To this end, whenever the users triggers the instantiation of the TOR service, the SSM makes sure the VPN VNF is brought up as well.

The VPN VNF as such is designed as a single-VDU VNF comprising a single QCOW2 image as Virtual Deployment Unit. It requires at least on vCPU core, 1 GB RAM, and 1 GB Storage. The VNF offers 3 external Connection Points, namely: mgmt, input, and output. By default, the VNF monitors the CPU utilization, the memory utilization, and the NIC utilization. The VNF Descriptor that specifies the VNF metadata can be found in the annex at Section A.2.4.

Function Specific Manager

The Function Specific Manger for the Intrusion Detection VNF takes care of a) the regular life-cycle events, such as starting and stopping the VNF as well as b) the initial configuration of the VNF. Moreover, the FSM communicates with the Service Specific Manager and sends event notifications to the SSM. In particular, the FSM emits and forwards information on its network configuration and thus makes the connection point, i.e. the IP address and the port, available to the SSM which can use that information for further configuration decisions and present that information to the user.

Technologies

The Virtual Private Network VNF is based on OpenVPN [15], that allows creating secure point-to-point or site-to-site connections in routed or bridged configurations and remote access facilities.

To this end, a user can leverage a VPN client to connect to the VPN service. Thus, all traffic that is sent to the VPN service is encrypted. In general, OpenVPN can create either a layer-3 based IP tunnel (TUN), or a layer-2 based Ethernet TAP that can carry any type of Ethernet traffic. In our case, we configure OpenVPN to work on layer-3 and provide IP tunnels.

4.2.2.5 Anonymizer Service

The Anonymizer VNF is based on TOR [21] and is always used in conjunction with the VPN Service. Thus both services are always added together. The TOR VPN sends all traffic to the TOR network and makes sure the user cannot be traced back. The Anonymizer Service directs Internet traffic through a free, worldwide, volunteer network consisting of more than seven thousand relays[9] to conceal a user's location and usage from anyone conducting network surveillance or traffic analysis. As part of the Personal Security Application, this makes it more difficult for Internet activity to be traced back to the user. This includes visits to Web sites, online posts, instant messages, and other communication forms. Thus, it offers additional security service for the user.

Architecture

The Anonymizer VNF is designed as a single-VDU VNF comprising a single QCOW2 image as Virtual Deployment Unit. It requires at least on vCPU core, 1 GB RAM, and 1 GB Storage. The VNF offers 3 external Connection Points, namely: mgmt, input, and output. By default, the VNF monitors the CPU utilization, the memory utilization, and the NIC utilization. The VNF Descriptor that specifies the VNF metadata can be found in the annex at Section A.2.5.

An alternative design for the Anonymizer VNF is a two-VDU VNF comprising the QCOW2 image of the TOR service as well as the QCOW2 image of the VPN service. This VNF also offers 3 external Connection Points but hides a more complex internal network, where the TOR service VDU(s) and the VPN service VDU(s) are interconnected and share an E-LAN connection for management purposes.

Function Specific Manager

The Function Specific Manger for the Anonymizer VNF takes care of a) the regular life-cycle events, such as starting and stopping the VNF as well as b) the initial configuration of the VNF. Moreover, the FSM communicates with the Service Specific Manager and sends and receives event notifications to the SSM. In particular, the FSM consumes information from the SSM configure the network forwarding rules such that all traffic is transferred directly to the firewall.

Technologies

The Anonymizer VNF is based on TOR [21], which adds encryption to the application layer of a communication protocol stack, nested like the layers of an onion. Tor encrypts the data, including the next node destination IP address, multiple times and sends it through a virtual circuit comprising successive, randomly selected Tor relays. From a VNF point of view, the TOR service operates transparently for the user. All traffic that is send via the VPN service is forwarded to the TOR service and thus anonymized. To this end, TOR operates on layer-3 and only required an IP connection point that must be offered to the user.

4.2.2.6 Self-Service Portal

The Self-Service Portal VNF offers a simple Web interface where the user can configure its Personal Security Application by adding and removing VNFs to the service. Moreover, the Web interface

offers some basic information, like the IP address of the VPN endpoint, to the user.

Architecture

The Self-Service Portal VNF is designed as a single-VDU VNF comprising a single QCOW2 image as Virtual Deployment Unit. It requires at least on vCPU core, 1 GB RAM, and 1 GB Storage. The VNF offers 2 external Connection Points, namely: mgmt and input. By default, the VNF monitors the CPU utilization, the memory utilization, and the NIC utilization. The VNF Descriptor that specifies the VNF metadata can be found in the annex at Section A.2.6.

Function Specific Manager

The Function Specific Manger for the Self-Service Portal VNF takes care of a) the regular life-cycle events, such as starting and stopping the VNF as well as b) the initial configuration of the VNF. Moreover, the FSM communicates with the Service Specific Manager and sends event notifications to the SSM. In particular, the FSM collects the proprietary triggers send by the self-service portal Web applications and translates them to events which can be understood by the SSM. Moreover, it collects the events, like a "VNF started" event, emitted by the SSM to update the Web application, e.g. to present the current status of the PSA service components.

Technologies

The Self-Service Portal VNF runs a Web application that offers a REST API which allows to retrieve the available services (GET /services), to start services (POST /services/{service-id}), and to stop services (DELETE /services/{service-id}). Moreover, it ships a JavaScript front-end application to interact with the backend application. Whenever an application is started or stopped, respectively, the backend application emits an event to the VNF's Function Specific Manager, which forwards the event to the SSM. The SSM interprets the event and triggers the related life-cycle mechanism to start or stop the related VNF. Evidently, the Service Portal backend application, the corresponding FSM, and the SSM have to be tightly integrated.

4.2.2.7 Traffic Splitter and Merger

For the Personal Security Application, traffic has to be steered to various VNFs depending on specific traffic characteristics. The encrypted VPN traffic, for instance, has to be forwarded to the VPN service, the HTTP traffic might be forwarded to the HTTP proxy, whereas non-HTTP traffic might bypass that proxy, etc. Thus, traffic has to be split and merged at various occasions, based on the current configuration of the PSA service. To this end, a Traffic Splitter and Merger (TSM) VNF is introduced. The TSM VNF can be configured by the SSM and supports and enriches the native VIM networking capabilities.

Architecture

The Traffic Splitter and Merger VNF is designed as a single-VDU VNF comprising a single QCOW2 image as Virtual Deployment Unit. It requires at least on vCPU core, 1 GB RAM, and 1 GB Storage. The VNF offers 3 external Connection Points, namely: mgmt, input, and output. By default, the VNF monitors the CPU utilization, the memory utilization, and the NIC utilization. The VNF Descriptor that specifies the VNF metadata can be found in the annex at Section A.2.7.

Function Specific Manager

The Function Specific Manger for the Self-Service Portal VNF takes care of a) the regular life-cycle events, such as starting and stopping the VNF as well as b) the initial configuration of the VNF. Moreover, the FSM communicates with the Service Specific Manager and sends event notifications to the SSM. In particular, the FSM collects the events and forwarding configurations from the service life-cycle management and translates it into commands specific to the Traffic Splitter and Merger VNF.

Technologies

The Traffic Splitter and Merger implementation is based on Open VSwitch [16], which is a software implementation of a virtual multilayer network switch, designed to enable effective network automation through programmatic extensions, leveraging control protocols like OpenFlow. In the Personal Security Application pilot, we make use of OVS capability to identify flows based on OpenFlow matches and forward them to specific output ports based on specific forwarding rules. This gives us the flexibility to steer traffic according to the PSA configuration without the necessity to rely on VIM features. In fact, Open VSwitch offers us the flexibility to steer traffic, i.e. split and merge traffic, as needed.

4.2.3 The PSA Network Service and Service Orchestration

The Personal Security Network service is comprised of all the VNFs described above. In contrast to other service, especially the ones in the pilots, the PSA network service can change continuously based on user generated events. That is, users can dynamically add and remove PSA components implemented as VNFs. Obviously, this behavior has to be reflected in the Network Service Descriptors, the life-cycle events, and monitoring. Thus, we describe the related parts next.

4.2.3.1 Network Service Descriptor

The NSD combines the various VNFs to the actual network service. In case of the PSA service, the NSD makes use of various flavours with different service function chains to reflect the different configurations comprising different sets of VNFs. Whenever a VNF is added to (or removed from) the current network service configuration, the service life-cycle management sends an event containing a new service function chain to the SONATA MANO framework and the Infrastructure Adapter. The Infrastructure Adapter then starts (or stops) the related VDUs and updates the service function chain and the forwarding graph respectively.

4.2.3.2 Lifecycle Events

The PSA service relies largely on the flexibility offered by the SONATA Service Platform. The pilot makes use of the FSM/SSM infrastructure and the plugable architecture. All the features of the PSA pilot, such as adding and removing VNFs on the fly and changing the service function chain accordingly, are based on the message exchange of life-cycle events. To this end, the PSA pilot leverages the following existing and new life-cycle events. These events are exchanged between the FSMs and the SSM as well the SSM and the Service Platform.

- Deploy VNF (VNF-ID)
- Remove VNF (VNF-ID)
- Update SCF (SFC-Flavour-ID)

- Scale up VNF (VNF-ID)
- Scale down VNF (VNF-ID)

4.2.4 Preliminary Deployment and Results

The Personal Security Application network service can be deployed on a regular SONATA Service Platform installation, possibly with across multiple VIMs. It has to be placed between the end-user and the regular Internet. Thus, for the purpose of the PSA demo, the NFVI that hosts the PSA service has to connect the service Connection Points to the Internet on the one hand, and the end-user on the other hand, and the end-user on the other hand. We may use a VPN connection to connect an end-user device, like a CPE, to the NFVI and the PSA related Connection Point. To this end, we may need the following additional components and services:

- A CPE that connects the end-user equipment to the PSA network service, by creating a VPN to the PoP that runs the service. Moreover, the CPE itself might showcase things like traffic statistics.
- An NFVI that can connect the PSA service to the Internet.
- End-user equipment, such as a laptop with a Web browser, that can be used to connect to the PSA self-portal service to configure the PSA service.
- Several services on the Internet that can be used to demonstrate the PSA capabilities. A Web server that delivers content which can be cached by the HTTP cache as well a content that is filtered by the PSA parental control system. Moreover, a service that generate 'malicious' traffic which is filtered by the Firewall. Lastly, a service that benefits from the TOR service offered by the PSA service.

4.3 Hierarchical Service Providers Pilot

In this Pilot we demonstrate hierarchical service providers by extending our vCDN Pilot.

4.3.1 Description and motivation

In the Hierarchical Service Providers (HSP) scenario, one SP provides a Network Service (NS) to the other SP. We term these the lower-SP and upper-SP. Essentially, as far as the lower-SP is concerned, the upper-SP is just another customer requesting service; similarly, from the upper-SP's perspective, the lower-SP is providing a component in their overall network service in a similar manner to the NFVI.

In the Pilot, we consider a SP that offers a CDN service to content providers. The (upper-)SP performs the content caching; it cooperates with a network operator that provides connectivity for end users and also provides the firewalling aspect of the CDN service, on behalf of the upper-SP and so it acts as a lower-SP, because this is best done nearer the end users.

As another motivating example, consider an operator selling services to end customers. Instead of deploying all of the VNFs and network infrastructure, it sub-contracts to a "wholesale operator" to provide the infrastructure and the VNFs, perhaps apart from one specialist VNF that it providers itself. By doing so, the operator may be able to provide its customer with a better performance (latency, reliability), at lower cost and at an earlier time than it could do by providing the entire service itself. A similar scenario could occur when different departments in an enterprise want to supplement the basic, enterprise-wide service with some specific extra functions.

The interface between the two SPs is essentially identical to the interface between a SP and a customer (which Sonata has already defined). This approach allows the details of each SP to be hidden from the other. Hence the lower-SP could alter the way it implemented the service (as long as the service presented at the interface is maintained unchanged), and similarly the upper-SP could shift to using a different lower-SP. As an Orchestrator of an SP already has a mechanism to interact with Infrastructure Managers - namely, the wrappers for VIMs and WIMs - we can reuse this functionality and create a wrapper for another Orchestrator. Note that each SP does its own MANO functionality, including lifecycle management.

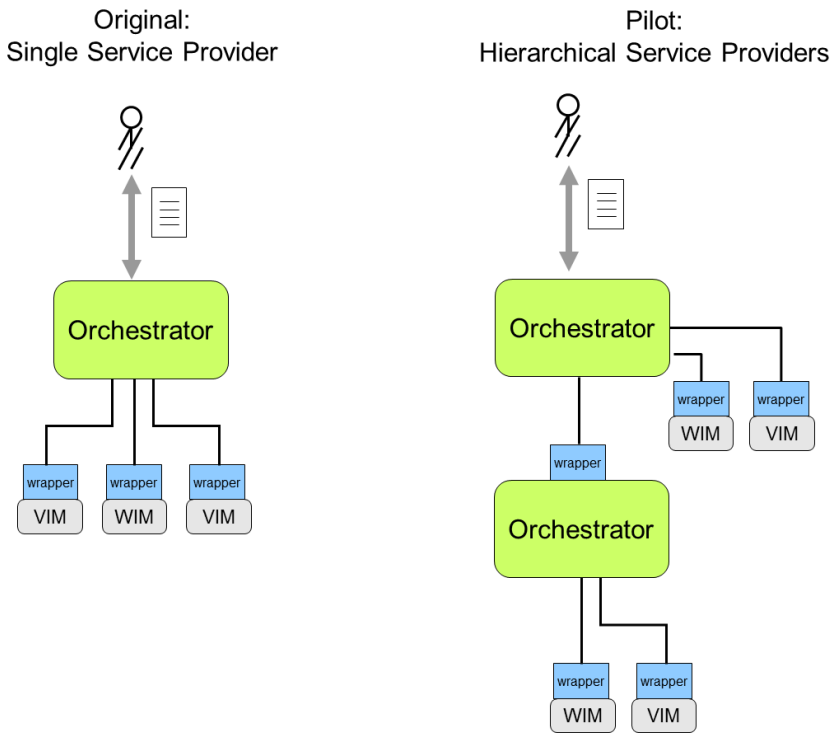

Figure 4.13: Hierarchical service platforms interconnected through wrappers

We don't want the architecture to be limited to just an upper-SP and lower-SP – we want to be able to have a stack of SPs. This allows us to combine or extend the scenarios above (imagine the operator above (#1) is bought by another operator, which wants to resell operator #1's NSs). Such a recursive approach simplifies operational management and the creation of new services. The most general case of this recursive architecture is a tree. Although this may not appear in many real deployments, it is beneficial to have such a flexible and generic model, rather than a specially defined but limited lower-SP / upper-SP model.

The use case was mentioned in D3.1 and D6.1, and recursive architectures were discussed in D2.2 and D2.3. This is not an East-West multi-domain peer-to-peer interaction, rather it is a North - South interface. We already have a mechanism which is a North - South interface - the wrapper to a VIM or a WIM, and as stated for symmetry we can extend the North - South interface to have a wrapper for an Orchestrator. We have to consider how does an Orchestrator informs another Orchestrator what VNFs and NS are available. In this recursive architecture, there needs to be a Capability Exposure mechanism upwards, i.e. an Orchestrator can expose a set of VIMs and a set

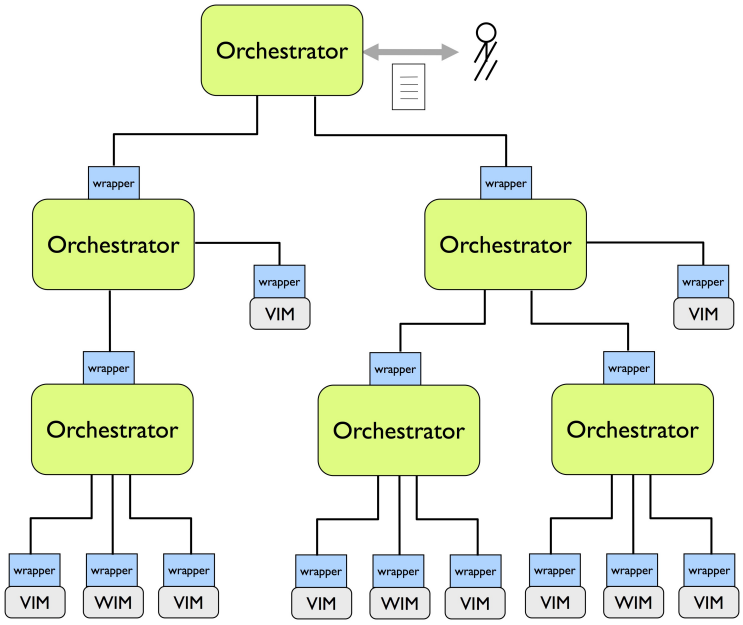

Figure 4.14: Large-scale example for recursive deployment

of WIMs upwards, and this is essentially the existing catalogue /registry.

4.3.2 Architectural approach for the pilot

We make a few choices about the implementation of the Pilot (mostly to make it simpler):

- Two levels of SP – the recursive architecture allows an arbitrary number of levels of the hierarchy of SPs. In the pilot, we implement a scenario with two, i.e. an upper-SP and a lower-SP.

- The NS provided to the customer consists of two VNFs chained together. The upper-SP provides one VNF itself, and the lower-SP provides one VNF and the NFVI. (It would be simpler if the upper-SP devolved the entire service provision to the lower-SP (like a virtual operator), but our choice allows us to explore more challenging issues.)

- The two SPs identify services in the same way (in fact as a triple of: vendor name, service name and version number). In a real world situation, this corresponds to the assumption that the SPs share a catalogue of services (perhaps it is global). (In situations where this is not possible, then the best approach is probably for the lower-SP to publish its catalogue, most likely with a publish-subscribe model, so upper-SPs are aware of changes in the lower-SP's catalogue; the upper-SP would also be responsible for putting a request in the format so that the lower-SP can understand it.)

- The SP API is a request /response for service. This is also likely in the real world, as it is simple and fits with the Openstack approach. One could imagine more complicated models, where for a NS requiring several elements, several tentative requests are made and, if they're all positive, then a firm request is sent. In the simple approach, if one request fails then roll-back (i.e. delete) requests must be sent to the other elements.

- The requests are sent at the same time. It is possible that, for speed reasons, the upper-SP would make the request to the lower-SP and then to its own VNF. This is a second-order issue that we don't plan to explore in the pilot.
- Resources are always granted. This is for simplicity. In the real world, each SP tracks its own resources and would check that it has sufficient to meet the request.
- The two SPs are pre-authenticated and pre-authorised. Again, this is for simplicity. In the real world, requests and responses would have to be checked for validity. In the pilot, the gatekeeper sends requests and receives responses, so steps involving AAA (and perhaps the BSS) could easily be added.
- There needs to be a way for the packets to be forwarded between the VNFs that are run by the two SPs. In the Pilot, this is done using Layer 2 addressing, with a bridging protocol (virtual tenant networking) ensuring the two layer 2 domains are fully joined up, and IP addresses are manually coordinated. In the real world, these approaches aren't scalable. The most promising approach seems to be that, as part of the orchestration, the upper-SP tells the lower-SP what IP address is to be used for the service - we are raising the issue in the standards bodies (OSM and ETSI).

4.3.3 Deployment scenario

The goal of this deployment scenario is to showcase the deployment of a network service in a recursive SONATA service platform environment. To do so, two full service platforms, each having its own NFVI infrastructure available, are deployed and connected in a recursive manner. The first platform, referred as *upper-SP*, is the entry point for the service instantiation requests done by the BSS. The second platform, referred as *lower-SP*, has its northbound interface connected to the southbound interfaces of the *upper-SP* to which it offers service deployment capabilities as shown in Figure 4.15. The network service used for this scenario is a simplified version of the network service used in the vCDN pilot. It is comprised of two chained VNFs namely a virtual firewall (vFW) and a virtual cache (vCache). Like in the vCDN pilot, the service is deployed between a content provider and the end users.

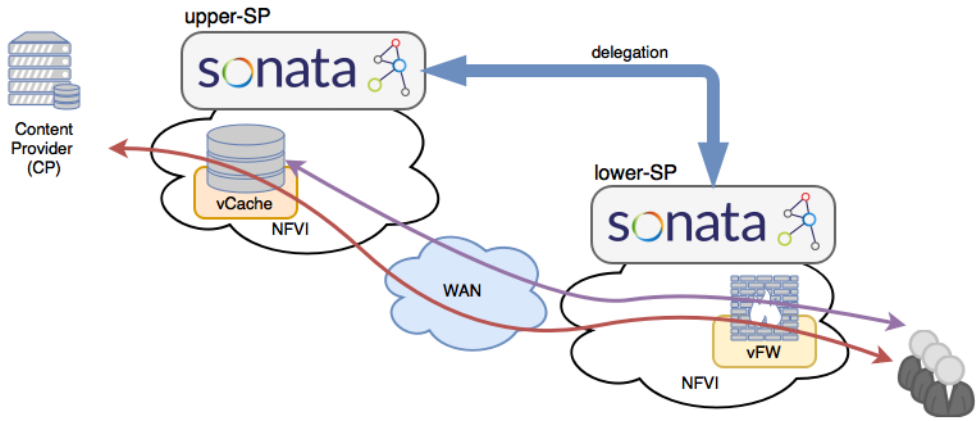

Figure 4.15: Pilot scenario 1: HSP NS deployment

Figure 4.15 shows how this network service is deployed across the two recursively connected, service platforms. More specifically, the vCache VNF is deployed by the *upper-SP* and the deployment, instantiation and management of the vFW VNF is delegated to and performed by the

lower-SP. The two platforms also take care to setup the service chaining between the two platforms that could even be deployed in different locations, for example, London and Athens. In this scenario, end user requests access the network service through the vFW deployed at the *lower-SP*, traverse the service to the vCache in the *upper-SP* and leave the service chain towards the content provider (CP). If the requested content is already available in the vCache (a *hit*), the users are directly served from the cache VNF.

This scenario includes the following steps:

- On-board the network service on the *upper-SP*. This includes a NSD and the VNFD for the vCache. This NSD references the vFW that should be provided by the *lower-SP*.
- The vFW in the *lower-SP* is assumed to be already on-boarded and ready to be instantiated.
- The BSS requests the instantiation of the service from the *upper-SP* through the standard SONATA northbound interface, i.e., Gatekeeper.
- The *upper-SP* starts to instantiate the service. During this process the *upper-SP* requests the instantiation of a vFW service from the *lower-SP* which immediately instantiates it.
- Both platforms configure the service chaining, including the link between both platforms, in a cooperative process under control of the *upper-SP*.
- The instantiated network service is up and running and the end user can access the content, e.g., playing a streaming video.

4.3.4 Virtual Network Functions for HSP pilot

The VNFs for this pilot will be re-used from the vCDN pilot. Their description can be found in Section 4.1.2.

4.4 Monitoring

SONATA Monitoring framework fulfills the requirements of service developers to accommodate 5G network services under the SDN/NFV paradigm, offering a cloud-technology agnostic framework able to provide real-time data collection and alert notifications based on several critical performance metrics per Virtualized Network Function (VNF) and/or Network Service (NS). The implemented monitoring framework defines an open-source, non-intrusive, flexible and modular mechanism able to follow rapidly changing VNFs and NSs and produce event-driven alerts at different customizable aggregation levels/rules.

In particular, the SONATA Monitoring Framework supports the use case scenarios realization and evaluation process in the following four aspects:

- Defining, in a static or dynamic way, the metrics to be collected and how/when alerts will be generated.
- Providing the ability to collect information from several components utilized for the use case scenario via generic and specific plugins.
- Storing metrics and related data to be visualized either real-time through websocket plugin or off-line via the GK-GUI.

- Sending notifications and alerts to the users/developers/services when rule thresholds are violated, through synchronous and asynchronous ways.

In this sense, it turns out that the Monitoring Framework is the "bonding material" among the instantiation of a service and the reaction of the user in case where conditions change, achieved through the continuous monitoring of metrics.

4.4.1 Architecture

Taking into account the diverse operational specifications and critical requirements that have to be fulfilled in network services (NSs) monitoring, SONATA Monitoring Framework architecture aims at providing an interactive monitoring framework capable of offering personalized, real-time data collection and alerting to all stakeholders of a SDN/NFV-enabled service platform, i.e. service developers, service platform operators and end-users, under heterogeneous cloud-enabled computing environments, such as VMs and containers.

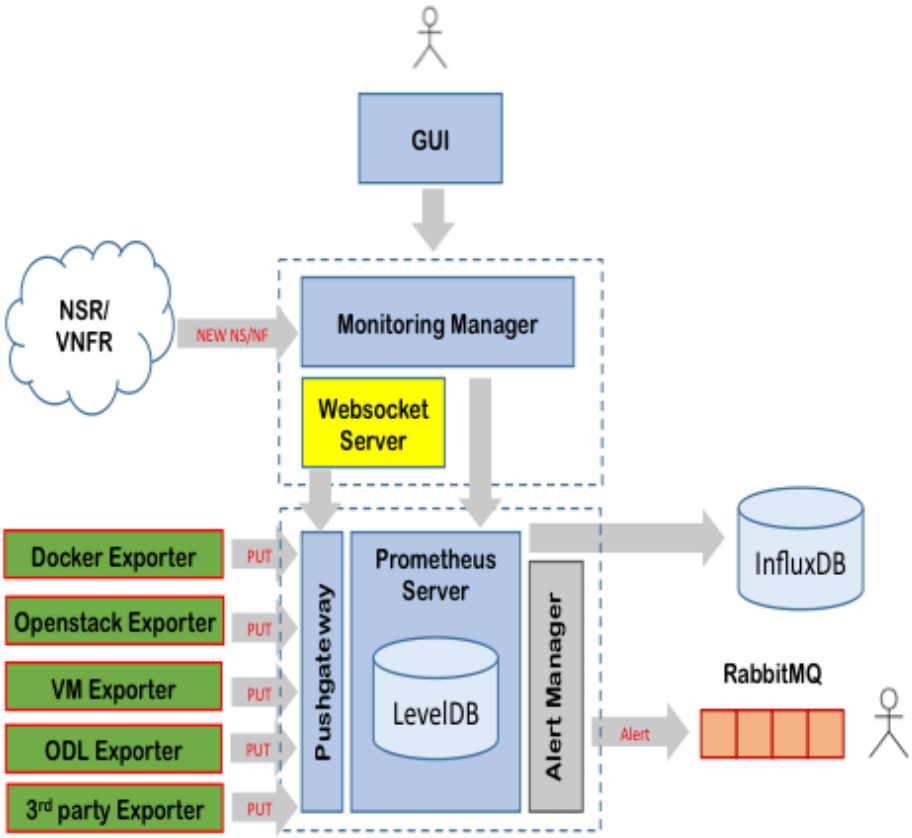

Figure 4.16: SONATA monitoring framework components

As depicted in Figure 4.16, to achieve this goal, the developed framework utilizes a number of flexible exporters to periodically push critical VNF metrics to a time-series monitoring server (Prometheus), where data are systematically organized to provide enriched VNFs' and NS's profiling. Through flexible (Django framework-based) API and Gatekeeper GUI implementation and message brokering (RabbitMQ) tools, developers and users are able to query for associative relations and mixed statistics and set/configure their own monitoring and alerting rules in order to monitor, control and optimize the performance of the NSs of interest.

Monitoring manager is a Django/rest-framework server combined with a PostgreSQL DB. The usage of the server allows the relation of each metric in Prometheus DB with only one label, while also to store all the relational information in Monitoring Manager. This configuration overcomes some restrictions coming from the adption of the time series DB and keeps the rule engine configuration simplest. In particular, Monitoring Manager provides the following abilities:

- Stores/Updates the relations between end-users, NSs, VNFs.
- Stores/Updates metrics configuration per VNF.
- Stores/Updates the active rules per NS.
- Reconfigures Prometheus server each time that NS/VNFs are created/deleted.
- Provides a restful API interface for the above actions.

Despite the fact that the default approach for Prometheus is to retrieve the metrics data by performing HTTP GET requests to exporters, in our design, we decided to follow the "push" approach, where exporters use HTTP POST methods to a PushGateway. The usage of "push" method has the following advantages:

- There is no need for exporters to implement a web socket in order to be reached from Prometheus server.
- Monitoring entities may be protected by a Firewall.
- There is no need for reconfiguring Prometheus server each time a VNF is instantiated or reconfigured (e.g. IP address reallocation, etc.).

In order to support this functionality, Prometheus provides an extra plugin called PushGateway, which allows to push monitoring data to an intermediary job which Prometheus can scrape. Finally, an alert manager is implemented, where alerting rules in monitoring server generates and sends alerts to the Alert manager which produces real time notifications sent to a message queue and received by the subscribed end-users.

In the developed framework, there are five different types of exporters regarding each monitoring entity (containers, VMs, Openstack and OpenFlow enabled servers).

- Container exporter is a software which runs inside the container of each VNF. This software collects data from some default metrics (like RAM usage CPU usage, disk usage etc.) or it can use custom metrics by running executable files which will be provided by the developer.
- VM Exporter aims to collect monitoring data from VMs running container engine which hosts NSs/VNFs. VM exporter is a process which periodically collects metric data by running executable files/scripts and sends them to Prometheus through PushGateway.
- Moreover, Prometheus Monitoring Server can collect monitoring data from an OpenFlow Exporter. In the current release, we developed a Python software that uses OpenDayLight API to collect data from the controller, forwarding them to a PushGateway.
- An OpenStack Exporter has also been developed as a software module (in Python language) that uses OpenStack API to collect data from all OpenStack components (Nova, Glance, Neutron, Swift, etc.). The data is again forwarded to the PushGateway and collected by the Monitoring Server.

In this point it has to be highlighted that the monitoring framework provides tools that can be easily extended in order to support any other custom metric and its integration with the rest of the regular metrics already provided by SONATA plugins.

Another aspect that use case scenarios will take advantage of is the federated architecture that monitoring framework supports, as illustrated in fig:monarch.

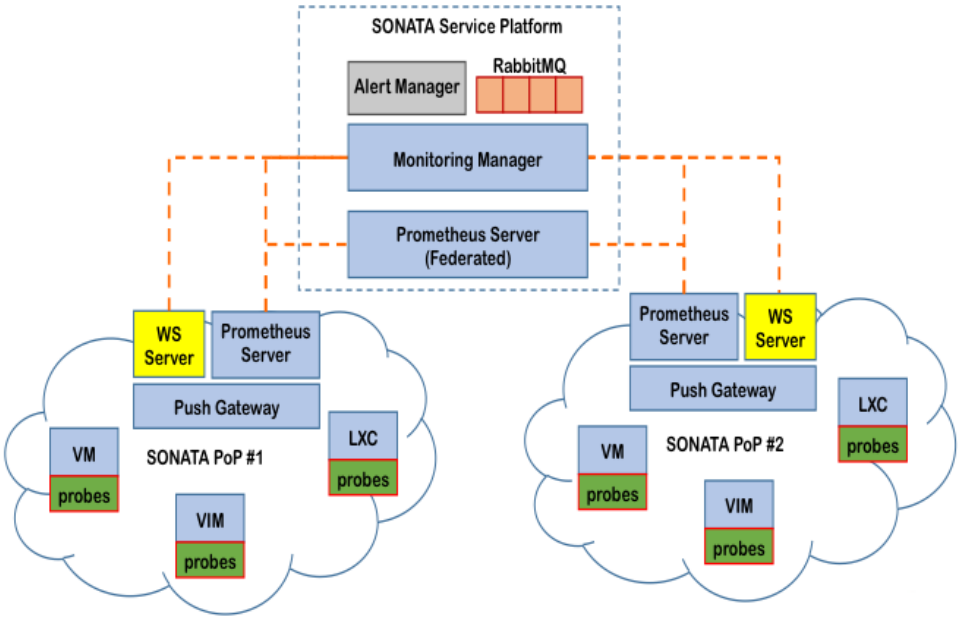

Figure 4.17: SONATA monitoring framework architecture

The main motivation behind the adoption of this federated architecture is the concept that a developer must be able to deploy a network service that consists of several components deployed in different SONATA PoPs. To support it, modifications have been made on the information exchanged between distributed Prometheus instances and the federated one, while also WebServices must be distributed to each PoP, as well. Most importantly, in this concept, alerting rules and notifications must be based on monitoring data collected in different PoPs and thus the decision must be made on a federation level, that is tailored to support stringent requirements from use case scenarios and especially those related to SP2SP use case.

4.4.2 Technologies

For the development and implementation of the SONATA monitoring framework, several technologies have been utilized. More specifically, the Monitoring Manager component capitalizes on the Django REST Framework (as well as many of its extensions/plugins) which is a free and open-source web framework, following the model-view-template (MVT) architectural pattern. Taking advantage of the reusability and "pluggability" of Django components, SONATA monitoring framework has developed the API for the users through the communication with Gatekeeper Graphical Use Interface. Additionally, the monitoring framework has extended Prometheus, which is an open-source service monitoring system, based on time series database that implements a highly dimensional data model, where time series are identified by a metric name and a set of key-value pairs. Prometheus provides a flexible query language, allowing slicing and dicing of collected time series data in order to generate ad-hoc graphs, tables, and alerts. In order to cover the requirements of SONATA, specific and generic exporters have been developed that allow bridging of third-party

data into Prometheus, including cAdvisor and collected in a “push” fashion (through the PushGateway). Also, the monitoring framework supports real-time monitoring information to the developer through websocket technology. In order to accomplish these requirements, a new component has been implemented that it is based on Tornado that is a Python web framework and asynchronous networking library that uses non-blocking network I/O in order to be capable of scaling to tens of thousands of open connections, making it ideal for long polling applications, such as WebSockets.

5 Pilot Selection and Evaluation Strategy

The following sections revisit the Deliverable 6.1 in order to provide an overview on the final selection of the Pilots for the SONATA project. Furthermore, these sections present the added value of SONATA with respect to current practices regarding Pilot deployments. Finally, we discuss the strategy regarding the evaluation which is about to start.

5.1 Pilot Selection

This section attempts a first mapping of the selected pilots, based on the initial categorisation done in Deliverable 6.1. The actual evaluation campaign and its results is subject of the forthcoming Deliverable 6.3. In D6.1, an assessment of all the initially proposed Use Cases against various criteria was elaborated. Those criteria were:

Project KPIs Check against the main KPIs for the project as laid out in the SONATA DoW.

Project Objectives Check against the main project objectives as laid out in the SONATA DoW.

SONATA functionalities coverage Check against the coverage of SONATA functionalities and identify gaps.

Workload characteristics variety Check against the workload characteristics of main VNFs embedded in the use case scenario.

Feasibility Check against the availability of UC components and the consortium experience in supporting the scenario described.

As the project has nominated three Use Cases to be implemented and demonstrated as Pilots, the *Feasibility* criterion is not relevant any more. In the subsections below we revise the initial analysis only for the selected Pilots.

5.1.1 Project KPIs

Table 5.1 presents an overview of the technical KPI coverage of the three Pilots: Virtual CDN (vCDN), Personal Security Application (PSA), and Hierarchical Service Platform (HSP). The provided list of technical KPIs will be refined and expanded in the upcoming Deliverable D6.3.

Table 5.1: SONATA KPIs

SONATA KPIs	Pilot1: vCDN	Pilot2: PSA	Pilot3: HSP
Resource usage	x	o	x
Service performance	x	o	x
SLA fulfillment	x	x	x
Service mapping	x	x	x
Service Setup Time	x	x	x
Scalability	x	x	x

In the table, (x) *denotes KPIs that are part of the Pilot* whereas (o) *denotes KPIs that are not part of the Pilot*

5.1.2 SONATA functionalities coverage

Table 5.2 presents the SONATA functionalities as introduced by Work Packages 3, 4, and 5 versus the coverage achieved by the original Use Cases as defined in WP2.

Table 5.2: SONATA functionalities

SONATA Functionality	Pilot1: vCDN	Pilot2: PSA	Pilot3: HSP
VNF development	x	x	o
SSM development	x	x	x
FSM development	x	x	x
Package (VNF/NS) On-boarding	x	x	x
Network Slicing	x	x	o
Service Function Chaining	x	x	x
VNF Placement (initialization)	x	x	x
VNF Placement (scaling)	x	o	-
Network scaling	x	x	-
single-PoP deployment	x	x	x
multi-PoP deployment	x	x	x
monitoring	x	x	x
NS/VNF updating	x	x	-
NS/VNF upgrade	o	x	-
multi (heterogeneous) VIM	-	-	-
WIM	x	x	x

In the table, (x) *denotes functionalities that are part of the UC*, (o) *denotes functionalities that are is not part of the UC*, and (-) *denotes functionalities that are not applicable to the UC.*

5.1.3 Workload characteristics

Figure 5.1 presents a mapping of the VNFs used in the Pilots to the taxonomy as provided by the ETSI NFV ISG [12]. As the VNFs that are participating in the Pilots are defined and implemented now the analysis in this deliverable is more detailed.

5.1.4 Mapping to collected requirements

This section presents the mapping of the collected requirement to the selected Pilots. For a detailed description of these requirements we refer to the previous deliverables [1] and [2]. Table 5.3 below lists the name of the requirement and the Pilot where the requirement validation will occur.

Table 5.3: Requirements mapping

Requirement Name	Pilot	Functional Verification	Non-functional Verification
VNF Catalogue	1, 2	On-boarding of functions from the VNF Catalogue	Time to on-board (measure time(sec) required to on-board a VNF to be available by the platform)
VNF Placement	1, 2, 3	Placement of the various NS components to specific locations in the underlying infrastructure	Time (sec) to calculate the placement decision (depends on the complexity of the infrastructure and of the NS)
Service Function Chaining	1, 2, 3	Chaining of VNFs components of the same NS in a single end-to-end chain (inside the PoP)	Time (sec) to setup the chain in the PoP; Latency (sec) caused by the chained path; Number of rules applied for the SFC to apply

Requirement Name	Pilot	Functional Verification	Non-functional Verification
VNF Placement	1, 2, 3	Placement of the various NS components to specific locations in the underlying infrastructure	Time (sec)to calculate the placement decision (depends on the complexity of the infrastructure and of the NS)
VNF Specific Monitoring	1, 2	Monitoring of VNF specific metrics via SONATA monitoring framework	Overall metrics transmission overhead (bps); Latency for the transmission of alert (sec)
NS/VNF Scaling	1	Scaling a VNF or a Network Service based on monitoring and alert thresholds	Time (sec) to scale
NS reconfiguration	1, 2	Reconfiguration of NS (i.e. update of VNFFG or extension)	Time (sec) to reconfigure
SP Recursiveness	3	SP to SP communication for deployment of NS on separate administrative domains	Time (sec) to deploy; Security features

5.2 SONATA Added Value

The most prominent feature of the SONATA framework is the tools it provides for the developers to use CI/CD methodology in novel NS/VNF deployment. The process includes all the CI/CD stages through integration/qualification and demonstration environments. However the pilots, slightly deviate from the demonstration of the above process. Instead, the development of all the components for the realisation of the Pilots, including the development of the Service Platform has been done following the proposed methodology. However it is difficult to include that process into the pilots. In other words the Pilots mostly focus on the actual novel functionalities SONATA provides and how those functionalities expand or enhance the usual deployments of similar to the pilots cases.

The functionality that can be highlighted by the first two pilots (i.e. vCDN and PSA) is the ability to influence the default life-cycle operations (i.e. FLM/SLM) in order to build in more intuitive and refined management operations. This ability resembles a lot with the current hype of supporting both Generic VNFM and function specific VNFM in all contemporary NFV MANO implementations. The very important difference is that the structure of the function specific VNFM in SONATA is using a plugable and highly modular architecture operating in a sandboxed environment allow the developer flexibility and at the same time isolation from other tenants of the Service Platform. At the same time in the service orchestration domain, the SSM plugins having the full operational picture and deployment knowledge may apply efficient orchestration algorithms in order to optimise resource usage and increase quality of experience for the NS users.

The above are manifested at the proposed pilots, in summary:

- In the vCDN pilot the developer can control the placement of vCCs given certain conditions are met via the ssm-placement plugin. Intelligence infused in the ssm-placement plugin may influence initial VNF placement or even alter the service during run-time so that it provides better services. This ability is not only due to policies (that most of the times are static) but via algorithms that can be developed, deployed and monitored during the NS lifetime. The vCDN pilots will demonstrate this ability by, (i) re-configuring service when requests are increased in an unserviced location and (ii) scale the vTU when the demand increases; (iii) monitor and fine-tune vCC parameters depending on the load.

- In the PSA pilot the developer enables the End User to control the functionalities that are provided by his NS. This is achieved dynamically through a Management GUI where the end-user may select the functionalities and even apply simple initial policies. The SSM will orchestrate the NS for each user so that the appropriate VNFs are instantiated and new

Pilot	Virtual Network Function	Data Plane			Control Plane				Signal Processing	Storage	
		Edge VNF	Intermediate NFs	NF supp Encryption	Routing	Authentication	Session Management	Scaling/Placement		Non-intensive	R/W Intensive
vCDN											
	vTC	✔	✔	✘	✔	✘	✔	✘	✘	✔	
	vTU		✔	✘	✘	✘	✔	✔	✔	✔	
	vCache	✔		✘	✔	✔	✘	✘	✘		✔
PSA											
	vProxy	✔	✘	✘	✔	✔	✘	✘	✘	✔	
	vFirewall	✔	✘	✘	✔	✔	✘	✘	✘	✔	
	vIDS	✔	✔	✘	✘	✘	✘	✘	✘	✔	
	vVPN Srv	✔	✘	✔	✔	✔	✘	✘	✘	✔	
	vAnon Srv	✔	✘	✔	✔	✔	✔	✘	✘	✔	
	SelfSrv	✔	✔	✘	✘	✔	✔	✔	✘	✔	
	TSM	✔	✔	✘	✔	✔	✘	✘	✘	✔	

Figure 5.1: Workload mapping

functionalities are supported. New VNFFG need to be applied before the NS the End-User traffic is again steered through the new chain. On top of that new algorithms that may correlate metrics received by the SONATA monitoring are possible to be inserted by the developer in order to strengthen the security provided by the NS.

- In the SP2SP pilot, the highlighted feature is the operation of the SP in a recursive mode. This innovative feature approaches what is considered as enabler for Service Provider to Service Provider cooperation. Of course basic assumptions are set in order to manage demonstration of this pilot within the time-frame of the project. The SP2SP will demonstrate the ability of SONATA framework to support splitting of the NS and deployment on slave SP that control remote resources. This allows the expansion of SP reach beyond resources that are readily controlled using a dynamic and flexible way.

5.3 Evaluation Strategy

The last phase of the project plan includes evaluation campaign that will validate and evaluate the functional and non-functional requirements as well as experiment with variations of the framework and features that are not supported in the proposed pilots (i.e. multi VIM support). In summary the following evaluations are anticipated:

- Functional level testing - These tests will be conducted in order to validate certain operations under the SONATA framework. The detailed tests expand beyond test that are executed automatically in Integration and Qualification phases. The tests include all three pilots and validate each non-functional requirement according to Table 5.3.

- Non-Functional level testing - These tests require deployment of instrumentation and ancillary tools in order to gather the required metrics for the evaluation of the non-functional attributes. It is important to set certain KPIs prior to the evaluation in order to assess the achievements and evaluate their maturity in terms of optimisations and implementation.

- System level testing - These test will involve more than one component of the SONATA SP and will be focused on the selected Pilots. Again certain KPIs will be appropriately set before the experiments.

The methodology to be followed for the aforementioned tests is described in ETSI NFV documents like [13] and [14]. In this context, all the recommended test methods (e.g. functional testing, performance testing etc.) address a certain target to be validated and a test environment enabling the test execution. A test target in the context of the present document is considered to be the System Under Test (SUT) which comprises one or more Functions Under Test (FUT). Each test specification validates one SUT where the SUT is one or more functional components of the NFV architecture. The SUTs considered for pre-deployment validation are the NFV Infrastructure (NFVI), a Virtualised Network Function (VNF), a Network Service (NS) or the Management and Orchestration (MANO). In general the approach will be as follows:

- Define a high level scope
- Define the Functions Under Test (FUT) in scope
- Define event-specific requirements
  - hosting
  - integration
  - event duration, etc.

The above information will be inserted into test sheets that present all the relative information and results of each test.

6 Conclusion

This Deliverable updates and extends the information of Deliverable D6.1 on the SONATA deployed infrastructures that support the Integration and Qualification environments. Moreover it introduces the SONATA demonstration infrastructure as a sum of a core infrastructure and remote testbeds. The deliverable provides deployment recipes and guidelines for developers interested in deploying a SONATA compatible infrastructure. SONATA Pilots are presented with more technical details regarding their deployment and the components (i.e. VNFs used). Finally, the SONATA evaluation strategy is preliminary discussed, setting the ground for the final deliverable of WP6 regarding the evaluation campaign.

A Pilot VNF Descriptors

A.1 vCDN Pilot VNF descriptors

A.1.1 virtual Traffic Classifier (vTC)

```
##
## vTC VNF descriptor.
##
descriptor_version: "vnfd-schema-02"
vendor: "eu.sonata-nfv"
name: "vtc-vnf"
version: "0.1"
author: "George Xilouris, Stavros Kolometsos :@ NCSRD"
description: "VNF implementing a vTC for traffic inspection and classification"
##
## The virtual deployment unit.
##
virtual_deployment_units:
  - id: "1"
    vm_image: "vtc-XXXXX.img"
    vm_image_format: "qcow2"
    resource_requirements:
      cpu:
        vcpus: 2
      memory:
        size: 4
        size_unit: "GB"
      storage:
        size: 20
        size_unit: "GB"
    monitoring_parameters:
      - name: "vm_cpu_perc"
        unit: "Percentage"
      - name: "vm_mem_perc"
        unit: "Percentage"
      - name: "vm_net_rx_bps"
        unit: "bps"
      - name: "vm_net_tx_bps"
        unit: "bps"
      - name: "mbits_packets_all"
        unit: "bps"
      - name: "mbits_packets_apple"
        unit: "pps"
```

```
      - name: "mbits_packets_bittorrent"
        unit: "pps"
      - name: "mbits_packets_dns"
        unit: "pps"
      - name: "mbits_packets_dropbox"
        unit: "pps"
      - name: "mbits_packets_google"
        unit: "pps"
      - name: "mbits_packets_http"
        unit: "pps"
      - name: "mbits_packets_icloud"
        unit: "pps"
      - name: "mbits_packets_skype"
        unit: "pps"
      - name: "mbits_packets_twitter"
        unit: "pps"
      - name: "mbits_packets_viber"
        unit: "pps"
      - name: "mbits_packets_youtube"
        unit: "pps"
    connection_points:
      - id: "vdu01:eth0"
        interface: "ipv4"
        type: "internal"
      - id: "vdu01:eth1"
        interface: "ipv4"
        type: "internal"
      - id: "vdu01:eth2"
        interface: "ipv4"
        type: "internal"

##
## The virtual links that interconnect
## the different connections points.
##
virtual_links:
  - id: "mgmt"
    connectivity_type: "E-LAN"
    connection_points_reference:
      - "vdu01:eth0"
      - "mgmt"
  - id: "input"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth1"
      - "vnf:input"
  - id: "output"
    connectivity_type: "E-Line"
```

```
    connection_points_reference:
      - "vdu01:eth2"
      - "vnf:output"

##
## The VNF connection points to the
## outside world.
##
connection_points:
  - id: "vnf:mgmt"
    interface: "ipv4"
    type: "management"
  - id: "vnf:input"
    interface: "ipv4"
    type: "external"
  - id: "vnf:output"
    interface: "ipv4"
    type: "external"

 mbits_packets_all kai mbits_packets_http
monitoring_rules:
  - name: "mon:rule:mbits_packets_all"
    description: "Trigger events if network load is greater than 1500."
    duration: 10
    duration_unit: "s"
    condition: "vdu01:mbits_packets_all > 1500"
    notification:
      - name: "notification01"
        type: "rabbitmq_message"
monitoring_rules:
  - name: "mon:rule:mbits_packets_http"
    description: "Trigger events if HTTP load is more than 10000."
    duration: 10
    duration_unit: "s"
    condition: "vdu01:mbits_packets_http > 10000"
    notification:
      - name: "notification02"
        type: "rabbitmq_message"
```

A.1.2 virtual Content Cache (vCC)

```
##
## vCC VNF descriptor.
##
descriptor_version: "vnfd-schema-02"
vendor: "eu.sonata-nfv"
name: "vprx-vnf"
version: "0.1"
author: "Alberto Rocha, Miguel Mesquita :@ Alabs"
```

```
description: "VNF implementing a vProxy (Squid based with local cache and authentication)"
##
## The virtual deployment unit.
##
virtual_deployment_units:
  - id: "1"
    vm_image: "squid-3.5.12-img"
    vm_image_format: "qcow2"
    resource_requirements:
      cpu:
        vcpus: 1
      memory:
        size: 2
        size_unit: "GB"
      storage:
        size: 20
        size_unit: "GB"
    monitoring_parameters:
      - name: "vm_cpu_perc"
        unit: "Percentage"
      - name: "vm_mem_perc"
        unit: "Percentage"
      - name: "vm_net_rx_bps"
        unit: "bps"
      - name: "vm_net_tx_bps"
        unit: "bps"
    connection_points:
      - id: "vdu01:eth0"
        interface: "ipv4"
        type: "internal"
      - id: "vdu01:eth1"
        interface: "ipv4"
        type: "internal"
      - id: "vdu01:eth2"
        interface: "ipv4"
        type: "internal"

##
## The virtual links that interconnect
## the different connections points.
##
virtual_links:
  - id: "mgmt"
    connectivity_type: "E-LAN"
    connection_points_reference:
      - "vdu01:eth0"
      - "mgmt"
  - id: "input"
```

```
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth1"
      - "vnf:input"
  - id: "output"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth2"
      - "vnf:output"

##
## The VNF connection points to the
## outside world.
##
connection_points:
  - id: "vnf:mgmt"
    interface: "ipv4"
    type: "management"
  - id: "vnf:input"
    interface: "ipv4"
    type: "external"
  - id: "vnf:output"
    interface: "ipv4"
    type: "external"

monitoring_rules:
  - name: "mon:rule:vm_cpu_usage_85_perc"
    description: "Trigger events if CPU load is above 85 percent."
    duration: 10
    duration_unit: "s"
    condition: "vdu01:vm_cpu_perc > 85"
    notification:
      - name: "notification01"
        type: "rabbitmq_message"
```

A.1.3 virtual Transcoding Unit (vTU)

```
---
##
## Some general information regarding this
## VNF descriptor.
##
descriptor_version: "vnfd-schema-01"
vendor: "eu.sonata-nfv"
name: "vtu-vnf"
version: "0.1"
author: "Javier Fernandez Hidalgo, Einar Meyerson :@ i2CAT"
description: "VNF implementing a vTU (virtual Transcoding Unit)"
##
```

```
## The virtual deployment unit.
##
virtual_deployment_units:
  - id: "1"
    vm_image: "sonata-vtu"
    vm_image_format: "qcow2"
    resource_requirements:
      cpu:
        vcpus: 4
      memory:
        size: 8
        size_unit: "GB"
      storage:
        size: 15
        size_unit: "GB"
    monitoring_parameters:
      - name: "vm_cpu_perc"
        unit: "Percentage"
      - name: "vm_mem_perc"
        unit: "Percentage"
      - name: "vm_net_rx_bps"
        unit: "bps"
      - name: "vm_net_tx_bps"
        unit: "bps"
    connection_points:
      - id: "vdu01:eth0"
        interface: "ipv4"
        type: "internal"
      - id: "vdu01:eth1"
        interface: "ipv4"
        type: "internal"
      - id: "vdu01:eth2"
        interface: "ipv4"
        type: "internal"

##
## The virtual links that interconnect
## the different connections points.
##
virtual_links:
  - id: "mgmt"
    connectivity_type: "E-LAN"
    connection_points_reference:
      - "vdu01:eth0"
      - "mgmt"
  - id: "input"
    connectivity_type: "E-Line"
    connection_points_reference:
```

```
      - "vdu01:eth1"
      - "vnf:input"
  - id: "output"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth2"
      - "vnf:output"

##
## The VNF connection points to the
## outside world.
##
connection_points:
  - id: "vnf:mgmt"
    interface: "ipv4"
    type: "management"
  - id: "vnf:input"
    interface: "ipv4"
    type: "external"
  - id: "vnf:output"
    interface: "ipv4"
    type: "external"

monitoring_rules:
  - name: "mon:rule:vm_cpu_usage_85_perc"
    description: "Trigger events if CPU load is above 85 percent."
    duration: 10
    duration_unit: "s"
    condition: "vdu01:vm_cpu_perc > 85"
    notification:
      - name: "notification01"
        type: "rabbitmq_message"
```

A.2 The VNFDs of the PSA Pilot

Below we present the current status of the VNF Descriptors used by the various VNFs of the Personal Security Application pilot. All descriptors are still under development at the time of writing. However, up-to-date descriptors can be found in the SONATA GitHub repositories.

A.2.1 Proxy VNFD

```
---
##
## vPRX VNF descriptor.
##
descriptor_version: "vnfd-schema-02"
vendor: "eu.sonata-nfv"
name: "squid"
```

```
version: "0.1"
author: "Alberto Rocha, Miguel Mesquita :@ Alabs"
description: >
    "VNF implementing a vProxy (Squid based with local
     cache and authentication)"

##
## The virtual deployment unit.
##
virtual_deployment_units:
  - id: "1"
    vm_image: "squid"
    vm_image_format: "qcow2"
    resource_requirements:
      cpu:
        vcpus: 1
      memory:
        size: 2
        size_unit: "GB"
      storage:
        size: 20
        size_unit: "GB"
    monitoring_parameters:
      - name: "vm_cpu_perc"
        unit: "Percentage"
      - name: "vm_mem_perc"
        unit: "Percentage"
      - name: "vm_net_rx_bps"
        unit: "bps"
      - name: "vm_net_tx_bps"
        unit: "bps"
    connection_points:
      - id: "vdu01:eth0"
        interface: "ipv4"
        type: "internal"
      - id: "vdu01:eth1"
        interface: "ipv4"
        type: "internal"
      - id: "vdu01:eth2"
        interface: "ipv4"
        type: "internal"

##
## The virtual links that interconnect
## the different connections points.
##
virtual_links:
  - id: "mgmt"
```

```
    connectivity_type: "E-LAN"
    connection_points_reference:
      - "vdu01:eth0"
      - "mgmt"
  - id: "input"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth1"
      - "vnf:input"
  - id: "output"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth2"
      - "vnf:output"

##
## The VNF connection points to the
## outside world.
##
connection_points:
  - id: "vnf:mgmt"
    interface: "ipv4"
    type: "management"
  - id: "vnf:input"
    interface: "ipv4"
    type: "external"
  - id: "vnf:output"
    interface: "ipv4"
    type: "external"

##
## The monitoring rules.
##
monitoring_rules:
  - name: "mon:rule:vm_cpu_usage_85_perc"
    description: "Trigger events if CPU load is above 85 percent."
    duration: 10
    duration_unit: "s"
    condition: "vdu01:vm_cpu_perc > 85"
    notification:
      - name: "notification01"
        type: "rabbitmq_message"
```

A.2.2 Firewall VNFD

```
---
##
## Some general information regarding this
## VNF descriptor.
```

```
##
descriptor_version: "vnfd-schema-02"
vendor: "eu.sonata-nfv.nec"
name: "pfsense"
version: "0.1"
author: "Michael Bredel, NEC, michael.bredel@neclab.eu"
description: "The PFSense firewall VNF for the PSA pilot"

##
## The virtual deployment unit.
##
virtual_deployment_units:
  - id: "1"
    vm_image: "pfsense"
    vm_image_format: "qcow2"
    resource_requirements:
      cpu:
        vcpus: 1
      memory:
        size: 1
        size_unit: "GB"
      storage:
        size: 1
        size_unit: "GB"
    monitoring_parameters:
      - name: "vm_cpu_perc"
        unit: "Percentage"
      - name: "vm_mem_perc"
        unit: "Percentage"
      - name: "vm_net_rx_bps"
        unit: "bps"
      - name: "vm_net_tx_bps"
        unit: "bps"
    connection_points:
      - id: "vdu01:eth0"
        interface: "ipv4"
        type: "external"
      - id: "vdu01:eth1"
        interface: "ipv4"
        type: "external"
      - id: "vdu01:eth2"
        interface: "ipv4"
        type: "external"

##
## The virtual links that interconnect
## the different connections points.
##
```

```
virtual_links:
  - id: "mgmt"
    connectivity_type: "E-LAN"
    connection_points_reference:
      - "vdu01:eth0"
      - "vnf:mgmt"
  - id: "input"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth1"
      - "vnf:input"
  - id: "output"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth2"
      - "vnf:output"


##
## The VNF connection points to the
## outside world.
##
connection_points:
  - id: "vnf:mgmt"
    interface: "ipv4"
    type: "management"
  - id: "vnf:input"
    interface: "ipv4"
    type: "external"
  - id: "vnf:output"
    interface: "ipv4"
    type: "external"
```

A.2.3 Intrusion Detection System VNFD

```
---
##
## Some general information regarding this
## VNF descriptor.
##
descriptor_version: "vnfd-schema-02"
vendor: "eu.sonata-nfv.nec"
name: "snort"
version: "0.1"
author: "Michael Bredel, NEC, michael.bredel@neclab.eu"
description: "The SNORT DPI VNF for the PSA pilot"

##
## The virtual deployment unit.
```

```
##
virtual_deployment_units:
  - id: "1"
    vm_image: "snort"
    vm_image_format: "qcow2"
    resource_requirements:
      cpu:
        vcpus: 1
      memory:
        size: 1
        size_unit: "GB"
      storage:
        size: 1
        size_unit: "GB"
    monitoring_parameters:
      - name: "vm_cpu_perc"
        unit: "Percentage"
      - name: "vm_mem_perc"
        unit: "Percentage"
      - name: "vm_net_rx_bps"
        unit: "bps"
      - name: "vm_net_tx_bps"
        unit: "bps"
    connection_points:
      - id: "vdu01:eth0"
        interface: "ipv4"
        type: "external"
      - id: "vdu01:eth1"
        interface: "ipv4"
        type: "external"
      - id: "vdu01:eth2"
        interface: "ipv4"
        type: "external"

##
## The virtual links that interconnect
## the different connections points.
##
virtual_links:
  - id: "mgmt"
    connectivity_type: "E-LAN"
    connection_points_reference:
      - "vdu01:eth0"
      - "vnf:mgmt"
  - id: "input"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth1"
```

```
      - "vnf:input"
  - id: "output"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth2"
      - "vnf:output"


##
## The VNF connection points to the
## outside world.
##
connection_points:
  - id: "vnf:mgmt"
    interface: "ipv4"
    type: "management"
  - id: "vnf:input"
    interface: "ipv4"
    type: "external"
  - id: "vnf:output"
    interface: "ipv4"
    type: "external"
```

A.2.4 Virtual Private Network Server VNFD

```
---
##
## VPN VNF Descriptor
##
descriptor_version: "vnfd-schema-02"
vendor: "eu.sonata-nfv"
name: "vpn-vnf"
version: "0.1"
author: "Luis Conceicao, UBIWHERE"
description: "Implementation of OpenVPN used in conjunction with TOR function"

##
## The virtual deployment units.
##
virtual_deployment_units:
  - id: "vdu01"
    vm_image: "http://files.sonata-nfv.eu/son-psa-pilot/vpn-vnf/sonata-vpn.qcow2"
    vm_image_format: "qcow2"
    resource_requirements:
      cpu:
        vcpus: 1
      memory:
        size: 2
        size_unit: "GB"
```

```
      storage:
        size: 20
        size_unit: "GB"
    monitoring_parameters:
      - name: "vm_cpu_perc"
        unit: "Percentage"
      - name: "vm_mem_perc"
        unit: "Percentage"
      - name: "vm_net_rx_bps"
        unit: "bps"
      - name: "vm_net_tx_bps"
        unit: "bps"
    connection_points:
      - id: "vdu01:eth0"
        interface: "ipv4"
        type: "external"
      - id: "vdu01:eth1"
        interface: "ipv4"
        type: "external"
      - id: "vdu01:eth2"
        interface: "ipv4"
        type: "external"

##
## The VNF connection points to the
## outside world.
##
connection_points:
  - id: "mgmt"
    interface: "ipv4"
    type: "management"
  - id: "input"
    interface: "ipv4"
    type: "external"
  - id: "output"
    interface: "ipv4"
    type: "external"

##
## The virtual links that interconnect
## the different connections points.
##
virtual_links:
  - id: "mgmt"
    connectivity_type: "E-LAN"
    connection_points_reference:
      - "vdu01:eth0"
      - "mgmt"
```

```
    dhcp: True
  - id: "input"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth1"
      - "input"
    dhcp: True
  - id: "output"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth2"
      - "output"
    dhcp: True

##
## The monitoring rules that react
## to the monitoring parameters
##
monitoring_rules:
  - name: "mon:rule:vm_cpu_perc"
    description: "Trigger events if CPU load is above 10 percent."
    duration: 10
    duration_unit: "s"
    condition: "vdu01:vm_cpu_perc > 10"
    notification:
      - name: "notification01"
        type: "rabbitmq_message"
  - name: "mon:rule:vm_mem_perc"
    description: "Trigger events if memory consumption is above 10 percent."
    duration: 10
    duration_unit: "s"
    condition: "vdu01:vm_mem_perc > 10"
    notification:
      - name: "notification02"
        type: "rabbitmq_message"
```

A.2.5 Anonymizer VNFD

```
---
##
## TOR VNF Descriptor
##
descriptor_version: "vnfd-schema-02"
vendor: "eu.sonata-nfv"
name: "tor-vnf"
version: "0.1"
author: "Luis Conceicao, UBIWHERE"
description: "Implementation of TOR function to be used as a transparent proxy"
```

```
##
## The virtual deployment units.
##
virtual_deployment_units:
  - id: "vdu01"
    vm_image: "http://files.sonata-nfv.eu/son-psa-pilot/tor-vnf/sonata-tor.qcow2"
    vm_image_format: "qcow2"
    resource_requirements:
      cpu:
        vcpus: 1
      memory:
        size: 2
        size_unit: "GB"
      storage:
        size: 20
        size_unit: "GB"
    monitoring_parameters:
      - name: "vm_cpu_perc"
        unit: "Percentage"
      - name: "vm_mem_perc"
        unit: "Percentage"
      - name: "vm_net_rx_bps"
        unit: "bps"
      - name: "vm_net_tx_bps"
        unit: "bps"
    connection_points:
      - id: "vdu01:eth0"
        interface: "ipv4"
        type: "internal"
      - id: "vdu01:eth1"
        interface: "ipv4"
        type: "internal"
      - id: "vdu01:eth2"
        interface: "ipv4"
        type: "internal"

##
## The VNF connection points to the
## outside world.
##
connection_points:
  - id: "mgmt"
    interface: "ipv4"
    type: "management"
  - id: "input"
    interface: "ipv4"
    type: "external"
  - id: "output"
```

```
    interface: "ipv4"
    type: "external"

##
## The virtual links that interconnect
## the different connections points.
##
virtual_links:
  - id: "mgmt"
    connectivity_type: "E-LAN"
    connection_points_reference:
      - "vdu01:eth0"
      - "mgmt"
    dhcp: True
  - id: "input"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth1"
      - "input"
    dhcp: True
  - id: "output"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth2"
      - "output"
    dhcp: True

##
## The monitoring rules that react
## to the monitoring parameters
##
monitoring_rules:
  - name: "mon:rule:vm_cpu_perc"
    description: "Trigger events if CPU load is above 10 percent."
    duration: 10
    duration_unit: "s"
    condition: "vdu01:vm_cpu_perc > 10"
    notification:
      - name: "notification01"
        type: "rabbitmq_message"
  - name: "mon:rule:vm_mem_perc"
    description: "Trigger events if memory consumption is above 10 percent."
    duration: 10
    duration_unit: "s"
    condition: "vdu01:vm_mem_perc > 10"
    notification:
      - name: "notification02"
        type: "rabbitmq_message"
```

A.2.6 Self-Service Portal VNFD

```
---
##
## Some general information regarding this
## VNF descriptor.
##
descriptor_version: "vnfd-schema-02"
vendor: "eu.sonata-nfv.nec"
name: "squid"
version: "0.1"
author: "Michael Bredel, NEC, michael.bredel@neclab.eu"
description: "The Self-Service Portal VNF for the PSA pilot"

##
## The virtual deployment unit.
##
virtual_deployment_units:
  - id: "1"
    vm_image: "self-service-portal"
    vm_image_format: "qcow2"
    resource_requirements:
      cpu:
        vcpus: 1
      memory:
        size: 1
        size_unit: "GB"
      storage:
        size: 1
        size_unit: "GB"
    monitoring_parameters:
      - name: "vm_cpu_perc"
        unit: "Percentage"
      - name: "vm_mem_perc"
        unit: "Percentage"
      - name: "vm_net_rx_bps"
        unit: "bps"
      - name: "vm_net_tx_bps"
        unit: "bps"
    connection_points:
      - id: "vdu01:eth0"
        interface: "ipv4"
        type: "external"
      - id: "vdu01:eth1"
        interface: "ipv4"
        type: "external"

##
## The virtual links that interconnect
```

```
## the different connections points.
##
virtual_links:
  - id: "mgmt"
    connectivity_type: "E-LAN"
    connection_points_reference:
      - "vdu01:eth0"
      - "vnf:mgmt"
  - id: "input"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth1"
      - "vnf:input"

##
## The VNF connection points to the
## outside world.
##
connection_points:
  - id: "vnf:mgmt"
    interface: "ipv4"
    type: "management"
  - id: "vnf:input"
    interface: "ipv4"
    type: "external"
```

A.2.7 Traffic Splitter and Merger VNFD

```
---
##
## Some general information regarding this
## VNF descriptor.
##
descriptor_version: "vnfd-schema-02"
vendor: "eu.sonata-nfv.nec"
name: "squid"
version: "0.1"
author: "Michael Bredel, NEC, michael.bredel@neclab.eu"
description: "The Traffic Splitter and Merger VNF for the PSA pilot"

##
## The virtual deployment unit.
##
virtual_deployment_units:
  - id: "1"
    vm_image: "tsm"
    vm_image_format: "qcow2"
    resource_requirements:
      cpu:
```

```
        vcpus: 1
      memory:
        size: 1
        size_unit: "GB"
      storage:
        size: 1
        size_unit: "GB"
    monitoring_parameters:
      - name: "vm_cpu_perc"
        unit: "Percentage"
      - name: "vm_mem_perc"
        unit: "Percentage"
      - name: "vm_net_rx_bps"
        unit: "bps"
      - name: "vm_net_tx_bps"
        unit: "bps"
    connection_points:
      - id: "vdu01:eth0"
        interface: "ipv4"
        type: "external"
      - id: "vdu01:eth1"
        interface: "ipv4"
        type: "external"
      - id: "vdu01:eth2"
        interface: "ipv4"
        type: "external"

##
## The virtual links that interconnect
## the different connections points.
##
virtual_links:
  - id: "mgmt"
    connectivity_type: "E-LAN"
    connection_points_reference:
      - "vdu01:eth0"
      - "vnf:mgmt"
  - id: "input"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth1"
      - "vnf:input"
  - id: "output"
    connectivity_type: "E-Line"
    connection_points_reference:
      - "vdu01:eth2"
      - "vnf:output"
```

```
##
## The VNF connection points to the
## outside world.
##
connection_points:
  - id: "vnf:mgmt"
    interface: "ipv4"
    type: "management"
  - id: "vnf:input"
    interface: "ipv4"
    type: "external"
  - id: "vnf:output"
    interface: "ipv4"
    type: "external"
```

A.3 Aveiro Configuration openrc

The Resource Configuration file (openrc) contains the credentials and the URL' API for the tenant to connect to the Openstack VIM and can be retrieved from the tenant Horizon portal:

- **Access and Security --> API Access --> Download Openstack RC file v3**

Openstack command line clients
To connect to NCSRD or Altice Labs Openstack API using CLI tools (previously installed), you should load the authentication variables like

```
$ source ncsrd-sonata.qual-openrc-v3.sh
 #!/usr/bin/env bash
 export OS_AUTH_URL=http://10.100.32.10:5000/v3
 export OS_PROJECT_ID=27b4af8cdbe24eb8b59e2dc86be072f2
 export OS_PROJECT_NAME="sonata.qual"
 export OS_USER_DOMAIN_NAME="default"
 if [ -z "$OS_USER_DOMAIN_NAME" ]; then unset OS_USER_DOMAIN_NAME; fi
 unset OS_TENANT_ID
 unset OS_TENANT_NAME
 export OS_USERNAME="sonata.qual"
 export OS_PASSWORD="***********"
 export OS_REGION_NAME="RegionOne"
 if [ -z "$OS_REGION_NAME" ]; then unset OS_REGION_NAME; fi

$ source os-alabs-son_qual-openrcv3.sh
 #!/usr/bin/env bash
 export OS_AUTH_URL=http://172.31.6.9:5000/v3
 export OS_PROJECT_ID=a0bf2553a97d42ccbdb02aa364e6d577
 export OS_PROJECT_NAME="son-qual"
 export OS_USER_DOMAIN_NAME="Default"
 if [ -z "$OS_USER_DOMAIN_NAME" ]; then unset OS_USER_DOMAIN_NAME; fi
```

```
unset OS_TENANT_ID
unset OS_TENANT_NAME
export OS_USERNAME="son-qual"
export OS_PASSWORD="********"
export OS_REGION_NAME="RegionOne"
if [ -z "$OS_REGION_NAME" ]; then unset OS_REGION_NAME; fi
export OS_INTERFACE=public
export OS_IDENTITY_API_VERSION=3
```

Ansible Openstack cloud module

To connect to NCSRD or Altice Labs Openstack services using Ansible cloud module, use the following file:

- **~/.config/openstack/clouds.yml**

```
os_ncsrd:
  auth:
    auth_url: 'http://10.100.32.200:5000/v3'
    username: 'sonata.qual'
    password: '***********'
    project_name: 'sonata.qual'
    project_domain_name: 'default'
    user_domain_name: 'default'
  auth_type: password
  identity_api_version: '3'
  #region_name: 'RegionOne'

os_alabs:
  auth:
    auth_url: 'http://172.31.6.9:5000/v3'
    username: 'son-demo'
    password: '********'
    project_name: 'son-demo'
    project_domain_name: "default"
    user_domain_name: "default"
  auth_type: password
  identity_api_version: "3"
  region_name: RegionOne
```

The EPC is the mobile core network responsible for connecting mobile devices with external packet networks. The modern EPC network is IP-focused. The main components of these EPC implementations are:

- Home Subscriber Server (HSS), as a central user and subscription database component (an SQL database with REST API)
- Mobility Management Entity (MME), as a control process responsible for attachment and handovers of mobile user equipment

- Serving/Packet Gateway (S/PGW), as forwarding devices of the bearer between the mobile user equipment and the external networks

As the findings in this experiment are rather agnostic to the selected service, we won't further go into detail on the functionality of the EPC, and it is sufficient to assume 3 to 4 VNFs (depending on the separation of the gateways) required in order to implement and deploy this service.

B Abbreviations

API Application Programming Interface

CI/CD Continuous Integration / Continuous Deployment

ETSI European Telecommunications Standards Institute

FSM Function-Specific Manager

GUI Graphical User Interface

HDD Hard Disk Drive

HSP Hierarchical Service Providers

HW Hardware

IA Infrastructure Abstraction

IDE Integrated Development Environment

II Integration Infrastructure

KPI Key Performance Indicator

MANO Management and Orchestration

NF Network Function

NFV Network Function Virtualisation

NFVI-PoP Network Function Virtualisation Points of Presence

NFVO Network Function Virtualisation Orchestrator

NFVRG Network Function Virtualisation Research Group

NS Network Service

NSD Network Service Descriptor

ODL OpenDayLight SDN Controller

OSS Operations Support System

OS Operating System

PSA Personal Security Applications

QI Qualification Infrastructure

REST Representational State Transfer

SDK Software Development Kit

SDN Software-Defined Networking or Software-Defined Network

SLA Service Level Agreement

SNMP Simple Network Management Protocol

SP Service Platform

SSM Service-Specific Manager

SW Software

vCDN Virtual Content Distribution Network

VDU Virtual Deployment Unit

vEPC Virtualised Evolved Packet Core

VIM Virtual Infrastructure Manager

VLD Virtual Link Descriptor

VM Virtual Machine

VN Virtual Network

VNF Virtual Network Function

VNFD Virtual Network Function Descriptor

VNFFGD VNF Forwarding Graph Descriptor

VNFM Virtual Network Function Manager

VPN Virtual Private Network

WAN Wide Area Network

WIM WAN Infrastructure Manager

XMPP Extensible Messaging and Presence Protocol

C Bibliography

[1] SONATA consortium. D2.1: Use cases and requirements. Website, October 2015. Online at http://www.sonata-nfv.eu/content/d21-use-cases-and-requirements.

[2] SONATA consortium. D2.2 architecture design. Website, December 2015. Online at http://www.sonata-nfv.eu/content/d22-architecture-design-0.

[3] SONATA consortium. D5.1 continuous integration and testing approach. Website, December 2015. Online at http://www.sonata-nfv.eu/content/d51-continuous-integration-and-testing-approach.

[4] SONATA consortium. D2.3 updated requirements and architecture design. Website, December 2016. Online at http://www.sonata-nfv.eu/.

[5] SONATA consortium. D3.1: Basic sdk prototype. Website, May 2016. Online at http://www.sonata-nfv.eu/content/d31-basic-sdk-prototype.

[6] SONATA consortium. D4.1: Orchestrator prototype. Website, May 2016. Online at http://www.sonata-nfv.eu/content/d41-orchestrator-prototype.

[7] SONATA consortium. D5.2: Integrated lab based sonata platform. Website, June 2016. Online at http://www.sonata-nfv.eu/content/d52-integrated-lab-based-sonata-platform.

[8] SONATA consortium. D6.1: Definition of the pilots, infrastructure setup and maintenance report. Website, June 2016. Online at http://www.sonata-nfv.eu/content/d61-definition-pilots-infrastructure-setup-and-maintenance-report.

[9] SONATA consortium. D5.3: Integrated and qualified public release of sonata platform. Website, December 2017.

[10] Duane Wessels et al. Squid. Website. Online at http://www.squid-cache.org/.

[11] Quickspecs - hp proliant sl390s generation 7 (g7). Website, November 2014. Online at http://www8.hp.com/h20195/v2/GetPDF.aspx/c04123322.pdf.

[12] ETSI NFV ISG. Nfv performance portability best practices, etsi, v1.1.1, 2014. Online at http://www.etsi.org/deliver/etsi_gs/NFVPER/001_099/001/01.01.01_60/gs_nfv-per001v010101p.pdf.

[13] ETSI NFV ISG. Network functions virtualisation (nfv); pre-deployment testing; report on validation of nfv environments and services v1.1.1, 2016. Online at http://www.etsi.org/deliver/etsi_gs/NFV-TST/001_099/001/01.01.01_60/gs_NFV-TST001v010101p.pdf.

[14] ETSI NFV ISG. Network functions virtualisation (nfv); testing methodology; report on nfv interoperability testing methodology v1.1.1, 2016. Online at http://www.etsi.org/deliver/etsi_gs/NFV-TST/001_099/002/01.01.01_60/gs_nfv-tst002v010101p.pdf.

[15] Inc OpenVPN Technologies. Openvpn. Website. Online at https://openvpn.net/.

[16] Openvswitch - openflow virtual switch. Online at http://openvswitch.org/.

[17] The pfSense Project. pfsense. Website. Online at https://www.pfsense.org/.

[18] The OpenStack Project. OpenStack: The Open Source Cloud Operating System. Website, July 2012. Online at http://www.openstack.org/.

[19] Sourcefire. Snort. Website. Online at http://snort.org/.

[20] Nokia CloudBand Team. Nokia cloudband infrastructure software , data sheet. Website, Data sheet, July 2016. Online at https://tools.ext.nokia.com/asset/200058.

[21] Inc The Tor Project. The tor project. Website. Online at https://www.torproject.org/.