Deliverable D2.3

Prototype and Report Framework for enabling the encapsulation of NFV and SDN Applications

Abstract

This deliverable presents the analysis, design and prototype implementation of SELFNET mechanisms and procedures for the encapsulation of Network Function Virtualization (NFV) and Software Defined Network (SDN) applications. These applications are SELFNET sensors and actuators that will support the four use cases defined in the project: self-optimization, self-healing, self-protection and a composed use case combining the previous three. Their encapsulation basically provides common and uniform primitives and procedures to manage their lifecycle (instantiation, configuration, termination, etc.) irrespectively of the specific application logic. In particular, two prototypes are released with this deliverable in support of Virtual Network Functions (VNFs) and SDN application lifecycle management.
Disclaimer
This document contains material, which is the copyright of certain SELFNET consortium parties, and may not be reproduced or copied without permission. All SELFNET consortium parties have agreed to full publication of this document.

The commercial use of any information contained in this document may require a license from the proprietor of that information.
Neither the SELFNET consortium as a whole, nor a certain part of the SELFNET consortium, warrant that the information contained in this document is capable of use, nor that use of the information is free from risk, accepting no liability for loss or damage suffered by any person using this information.

The EC flag in this press release is owned by the European Commission and the 5G PPP logo is owned by the 5G PPP initiative. The use of the flag and the 5G PPP logo reflects that SELFNET receives funding from the European Commission, integrated in its 5G PPP initiative. Apart from this, the European Commission or the 5G PPP initiative have no responsibility for the content.

The research leading to these results has received funding from the European Union Horizon 2020 Programme under grant agreement number H2020-ICT-2014-2/671672.

Impressum
Full project title: Framework for Self-Organized Network Management in Virtualized and Software Defined Networks
Short project title: SELFNET
Number and title of work-package: WP2 - SELFNET Network Management Framework
Number and title of task: T2.3 – NFV-Apps and SDN-Apps Encapsulation
Document title: Prototype and Report Framework for enabling the encapsulation of NFV and SDN Applications
Editor: Giacomo Bernini, Nextworks
Work-package leader: Pedro Neves, PTIN

Copyright notice
© 2016 Participants in SELFNET project
Executive summary

SELFNET targets a high degree of automation in network management as imposed by the challenging 5G requirements and KPIs. Deployment of new services in 5G networks needs to follow highly dynamic and flexible approaches to reduce the provisioning time from 90 minutes to 90 seconds.

In this context, Network Function Virtualization (NFV) and Software Defined Networking (SDN) are key technologies to enable a very dynamic management of 5G networks, in terms of provisioning and maintenance of services from a management perspective. Indeed, SELFNET envisages a Self-Organized Network (SON) management infrastructure where NFV and SDN applications can be quickly plugged into virtualized network infrastructures and automatically configured in support of the four use cases defined in the project: self-optimization, self-healing, self-protection and a combined use case including all the other three ones. This poses the requirement for common and homogeneous mechanisms and procedures aiming to manage the whole lifecycle of NFV and SDN applications (i.e. instantiation, configuration, start, stop, scale, termination, etc) irrespectively of their specific logic or function. This means that NFV and SDN applications need to be properly encapsulated to achieve a high degree of automation in the deployment of services.

In SELFNET, NFV applications refer to those Virtualized Network Functions (VNFs) implementing either control or data plane functionalities (i.e. sensor and actuators) on top of a virtualized infrastructure managed by a cloud management system. On the other hand, SDN applications implement control plane functions that are not VNFs, and can even possibly run directly on top of a physical infrastructure. SDN applications can either be deployed (and run) outside or within an SDN controller runtime environment.

This deliverable presents the design and prototype implementation of the SELFNET NFV and SDN applications encapsulation mechanisms and procedures. In practice, this document defines how SELFNET sensors and actuators applications are dynamically and automatically plugged into the SELFNET virtualized network infrastructure. The reference baseline for the encapsulation architectural approach has been the ETSI MANO framework for the NFV part, with the identification of the VNF Manager (VNFM) as the key component responsible for the lifecycle management of all the SELFNET sensor and actuator VNFs. For the SDN part, the ONF SDN architecture and above all the encapsulation frameworks offered by OpenDaylight and ONOS (i.e. the two most popular and deployed SDN controllers) have paved the way for the specification of the SELFNET lifecycle management of SDN applications, both in the case of applications deployed within and outside the SDN controller runtime environment.

Available open source tools and technologies suitable for NFV and SDN applications encapsulation have been also investigated and compared with the aim of selecting the starting points for the prototype implementation activity. Tools like OpenBaton, OpenMANO, Tacker for the ETSI MANO implementation, and Karaf and OSGi technologies for the SDN applications management are introduced in the document for their adoption and enhancement in support of SELFNET requirements.

As a result, the SELFNET NFV applications encapsulation prototype is implemented as an ETSI MANO compliant VNFM for homogeneous and common lifecycle management of SELFNET sensor and actuators VNFs. It has been developed on top of the OpenBaton open source framework and it is described in this document in terms of interfaces, information models and installation guidelines. For the SDN applications encapsulation part, the implemented prototype provides an SDN Application Manager which leverages on the OSGi framework and Karaf containers to deploy, install and configure SDN applications within and outside SDN controllers. A description of the produced software prototype is provided in this document, including interfaces and workflows specifications.
## List of authors

<table>
<thead>
<tr>
<th>Company</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nextworks</td>
<td>Giacomo Bernini</td>
</tr>
<tr>
<td>Nextworks</td>
<td>Elian Kraja</td>
</tr>
<tr>
<td>Nextworks</td>
<td>Gino Carrozzo</td>
</tr>
<tr>
<td>PT Inovação e Sistemas SA</td>
<td>Pedro Neves</td>
</tr>
<tr>
<td>PT Inovação e Sistemas SA</td>
<td>Rui Calé</td>
</tr>
<tr>
<td>PT Inovação e Sistemas SA</td>
<td>Bruno Parreira</td>
</tr>
<tr>
<td>University of the West of Scotland</td>
<td>Jose Alcaraz-Calero</td>
</tr>
<tr>
<td>University of the West of Scotland</td>
<td>Enrique Chirivella-Perez</td>
</tr>
<tr>
<td>University of the West of Scotland</td>
<td>Qi Wang</td>
</tr>
<tr>
<td>University of the West of Scotland</td>
<td>James Nightingale</td>
</tr>
<tr>
<td>University of the West of Scotland</td>
<td>Ricardo Marco Alaez</td>
</tr>
<tr>
<td>Creative Systems Engineering (C.S.E.)</td>
<td>Konstantinos Koutsopoulos</td>
</tr>
<tr>
<td>Monoprosopi EPE</td>
<td>George Giannopoulos</td>
</tr>
<tr>
<td>Creative Systems Engineering (C.S.E.)</td>
<td>Ricardo Maia</td>
</tr>
<tr>
<td>Monoprosopi EPE</td>
<td>José Santos</td>
</tr>
<tr>
<td>Proef</td>
<td>Rui Alheiro</td>
</tr>
<tr>
<td>Proef</td>
<td>Luís Andrade</td>
</tr>
<tr>
<td>Proef</td>
<td>Ricardo Preto</td>
</tr>
<tr>
<td>Ubiwhere Lda.</td>
<td>Tiago Teixeira</td>
</tr>
<tr>
<td>Alvarion Technologies Ltd.</td>
<td>Jesús Alonso</td>
</tr>
</tbody>
</table>
Table of Contents

Executive summary ................................................................. 3
List of authors ........................................................................ 4
Table of Contents ................................................................. 5
List of Figures .................................................................. 7
List of Tables .................................................................. 9
Abbreviations .................................................................. 10
1 Introduction .................................................................. 11
   1.1 Scope clarification ..................................................... 11
   1.2 Objectives of this document .................................... 15
   1.3 Structure of the document ......................................... 16
2 State-of-the-art and gap analysis .................................. 17
   2.1 Reference architectures ........................................... 17
      2.1.1 ETSI MANO ..................................................... 18
      2.1.2 ONF SDN ....................................................... 20
      2.1.3 Applicability to SELFNET ................................ 22
   2.2 Lifecycle management of NFV applications ............... 23
      2.2.1 NFV MANO implementations ............................... 23
      2.2.2 Dynamic provisioning tools ................................ 31
   2.3 Lifecycle management of SDN applications: SDN controllers .................................................. 34
      2.3.1 OpenDaylight ................................................... 35
      2.3.2 ONOS ............................................................ 37
      2.3.3 Applicability to SELFNET ................................ 38
3 Encapsulation of NFV applications ............................. 39
   3.1 High level architecture .................................................. 40
      3.1.1 SELFNET VNFM and encapsulated VNF functional decomposition ....................................... 42
   3.2 SELFNET VNFM lifecycle management services .......... 47
      3.2.1 VNF instantiation .................................................. 47
      3.2.2 VNF scale ............................................................ 48
      3.2.3 VNF operation (start/stop) ..................................... 50
      3.2.4 VNF modification .................................................. 51
      3.2.5 VNF software update .......................................... 51
      3.2.6 VNF termination .................................................... 52
   3.3 SELFNET VNFM prototype ........................................... 53
      3.3.1 OpenBaton SELFNET VNFM ................................ 53
      3.3.2 Ubuntu Juju SELFNET VNFM ................................. 70
   3.4 High level architecture .................................................. 75
3.5 Interfaces and services ................................................... 77
   3.5.1 SDN APP Case ....................................................... 78
   3.5.2 SDN Controller APP Case ....................................... 81
   3.5.3 Multitenancy Issues .................................................. 83
3.6 Prototype release notes ................................................... 85
4 Conclusions ................................................................. 89

© SELFNET consortium 2016
References .................................................................................................................. 91
Annex A - Example of VNFD .................................................................................. 93
Annex B – SELFNET VNFM validation .................................................................. 94
  B.0 - HSS VNFD ...................................................................................................... 94
  B.1 - hss_configure.sh .......................................................................................... 95
  B.2 - hss_start.sh .................................................................................................. 95
  B.3 - SELFNET VNFM logging snippets ............................................................... 96
List of Figures
Figure 1 SELFNET System Architecture .................................................................12
Figure 2 Virtualized Network Function Types ..........................................................13
Figure 3 SELFNET System Architecture – Network Functions Lifecycle Management ....15
Figure 4 ETSI MANO architecture ...........................................................................18
Figure 5 ONF basic SDN model (source ONF SDN architecture 1.1).............................20
Figure 6 SDN controller and contexts (source ONF SDN architecture 1.1) ....................21
Figure 7 OpenMANO’s components mapped in ETSI’s NFV Architecture (source https://github.com/nfvlabs/openmano) ..............................................................23
Figure 8 OpenBaton overview ..................................................................................24
Figure 9 OpenStack Tacker overview .................................................................25
Figure 10 Cisco NFV service orchestration portfolio (source: Cisco ESC datasheet) ......26
Figure 11 State diagram of an encapsulated service in Juju (i.e. a Juju Charm) ............32
Figure 12: Ansible Architecture [24] .....................................................................34
Figure 13 ODL architecture .....................................................................................35
Figure 14 Karaf Structure ......................................................................................36
Figure 15 ONOS’ architecture (source https://wiki.onosproject.org/) .......................37
Figure 16 ONOS’ services structure (source https://wiki.onosproject.org/) ................38
Figure 17: VNFs Lifecycle Management and Encapsulation on SELFNET Architecture ..39
Figure 18 Evolution of ETSI NFV in support of SELFNET encapsulation of NFV applications.................................................................41
Figure 19 SELFNET encapsulation of NFV applications within ETSI NFV and MANO.....42
Figure 20 Selfnet VNFM and encapsulated VNF functional split ...............................43
Figure 21 VNF states and transitions (source: ETSI NFV SWA [34]) .........................46
Figure 22 SELFNET VNFM VNF instantiation workflow ......................................47
Figure 23 SELFNET VNFM VNF scale out/up workflow ......................................48
Figure 24 SELFNET VNFM VNF scale in/down workflow .....................................49
Figure 25 SELFNET VNFM VNF operation workflow ............................................50
Figure 26 SELFNET VNFM VNF modification workflow ......................................51
Figure 27 SELFNET VNFM VNF software update workflow ..................................51
Figure 28 SELFNET VNFM VNF termination workflow .......................................52
Figure 29 Output of the Automation_bash script .....................................................67
Figure 30 SELFNET VNFM validation scenario .....................................................68
Figure 31 SELFNET VNFM subscribed ...................................................................69
Figure 32 HSS VNFR in ACTIVE status ...................................................................69
Figure 33 HSS Virtual Machine in the OpenStack dashboard ...................................70
Figure 34 Overview of the Juju architecture prototype for SELFNET .......................71
Figure 35 Example of the folder structure of the charm to perform the deployment of an eNodeB ......................................................................................................71
Figure 36. OpenStack infrastructure where the Juju Bootstrap has been performed and the associated VM is running ............................................................72
Figure 37. Example environment configuration for Juju over OpenStack .................73
Figure 38. Different versions of Operating Systems supported by the Juju EMS Service..73
Figure 39. The figure shows an example of a VNF statically provisioned in Juju ........74
Figure 40: SDN APPs and Controller APPs Lifecycle Management and Encapsulation on SELFNET Architecture

Figure 41: SDN APP Installation, Activation and Invocation workflow

Figure 42: SDN APP Container

Figure 43: SDN Controller Application Instance

Figure 44: SDN APP Access Control

Figure 45: SDNAppManager Activated in Karaf

Figure 46: SDN App Installation

Figure 47: SDN App Instantiation

Figure 48: SDN APP instance exposed as REST endpoint

Figure 49: Interaction with SDN APP
List of Tables

Table 1 NFV MANO open source tools comparison ................................................................. 30
Table 2 SELFNET VNFM functional components ................................................................. 45
Table 3 SELFNET VNF and VNF-C LCM entities ................................................................. 46
Table 4 SELFNET VNFM VNFD ........................................................................................... 55
Table 5 SELFNET VNFM lifecycle_event ............................................................................... 55
Table 6 SELFNET VNFM configurations ............................................................................. 55
Table 7 SELFNET VNFM VDU ............................................................................................. 56
Table 8 SELFNET VNFM deployment_flavour ...................................................................... 56
Table 9 SELFNET VNFM vnfc ............................................................................................. 56
Table 10 SELFNET VNFM VNFR ......................................................................................... 57
Table 11 SELFNET VNFM VNFR vdu ................................................................................... 58
Table 12 SELFNET VNFM VNFR vnfc_instance .................................................................... 58
Table 13 SELFNET VNFM OMLvnfm_SAOr REST API (server-side) ..................................... 62
Table 14 SELFNET VNFM OMLvnfm_SAOr REST API (client-side) ...................................... 63
Table 15 SELFNET VNFM VNF_OMLvnfm messages (configuration queues) ...................... 63
Table 16 SELFNET VNFM VNF_OMLvnfm messages (registration queue) ......................... 64
Table 17 SELFNET VNFM software structure ..................................................................... 65
Table 18 SELFNET VNFM software pre-requisites ............................................................... 65
Table 19 selfnet-vnfm.sh operations .................................................................................... 65
Table 20 SELFNET VNFM software pre-requisites ............................................................... 66
Table 21 SELFNET LCM agent python pre-requisites ......................................................... 66
Table 22 Interfaces in the SELFNET Juju architecture ......................................................... 74
Table 23: SDN APPs Lifecycle Management Operations ..................................................... 79
Table 24: SDN APPs Control Interface .................................................................................. 81
Table 25: SDN Controller APPs Management Operations .................................................. 83
Abbreviations

API – Application Programming Interface
APP – SELFNET Application
BSS – Business Support Systems
CL – Control Layer
DC – Data Center
DHCP - Dynamic Host Configuration Protocol
DNL – Data Network Layer
DNS – Domain Name Server
EMS – Element Management System
ETSI – European Telecommunications Standards Institute
FCAPS - Fault, Configuration, Accounting, Performance, Security
IL – Infrastructure Layer
LCM – Lifecycle Management
MAAS – Metal as a Service
MANO – Management and Orchestration
NMS – Network Management System
NFV – Network Functions Virtualization
OML – NFV Orchestration and Management Layer
OSS – Operations Support Systems
PGW – Packet Gateway
PNE – Physical Network Element
PNF – Physical Network Function
QoS – Quality of Service
SAC – SON Access Layer
SAU – SON Autonomic Layer
SDO – Standard Organization
SDN – Software Defined Networking
SDN-App – Software Defined Networking Application
SDK – Software Development Kit
SFC – Service Function Chaining
SON – Self Organizing Networks
VCNF – Virtual Control Network Function
VDNF – Virtual Data Network Function
VIM – Virtualized Infrastructure Manager
VM - Virtual Machine
VPN – Virtual Private Network
VNF – Virtual Network Function
VNE – Virtual Network Element
VNFM – Virtual Network Function Manager
VTN – Virtual Tenant Network
1 Introduction

1.1 Scope clarification

SELFNET is addressing the ambitious KPI of reducing the provisioning time of new services from 90 days to 90 minutes. The shifting in these new capabilities of the 5G networks requires very advanced mechanisms to provide automated ways of performing the deployment of services. This aspect of the management of the network is being critical for the successful deployment of new services in 5G networks. The encapsulation of NFV and SDN applications plays a critical role in achieving a high degree of automation in the deployment of services. The main purpose of encapsulating applications is to expose a homogeneous way of managing them, including aspects such as deployment and undeployment, configuration and reconfiguration of the application, start and termination of the application, and in summary primitives to control their whole life cycle.

In order to address the challenging objectives of SELFNET, the defined system architecture, as described in D2.2 [1] is rather complex and contains several layers and internal sublayers. Each sublayer represents one (or more) logical components of the architecture. The following set of layers and sublayers are provided by the architecture:

- **Infrastructure Layer (IL):** provides the physical (Physical sublayer) and virtual (Virtualization sublayer) resources for the SELFNET components to be deployed;
- **Data Network Layer (DNL):** contains all the data plane network functions (SON Data Plane sublayer), including the ones that only have data plane functionalities, as well as the ones that have both control and data plane capabilities. In the latter scenario, only the data plane part of the functions run on this layer;
- **Control Layer (CL):** encompasses the SELFNET control plane functions (SON Control Plane sublayer). As in the DNL case, it includes functions that are solely control-plane related, as well as functions that include both control and data plane capacities (only the control plane part of the functions run on this layer). SDN Controllers are also part of this layer (SDN Controllers sublayer), exposing northbound services to network functions that contain only control plane features;
- **NFV Orchestration & Management layer (OML):** orchestration (Orchestration sublayer) and lifecycle management of the virtual resources (VIM sublayer) and of the Virtual Network Functions (VNFs) (VNFM sublayer), either they are control and/or data plane capable, as well as SDN or non-SDN compliant;
- **SON Autonomic Layer (SAL):** provides the SELFNET SON-related functionalities on top of the abovementioned network functions and infrastructure. In this layer is located the sensors collection and analysis (Monitor & Analysis sublayer), machine-based decision making algorithms (Autonomic Management sublayer), as well as the orchestration functionalities (Orchestration sublayer) to enforce SON-decisions on the network domain;
- **SON Access Layer (SAC):** exposes all the SELFNET autonomic information and functionalities to external actors, such as the Business Support Systems (BSS), Operational Support Systems (OSS), Administration GUI, etc.

The several layers and sublayers above described are illustrated in Figure 1.

In terms of network functions, SELFNET accommodates both physical and virtual network functions, also known as PNFs and VNFs. For the context of this document, we will zoom in and focus on the VNFs and on the SELFNET layers that are impacted by these.
As referred in D2.2 [1], when the control and data plane functionalities are not separated, the naming proposed by SELFNET is **Non-SDN-Compliant Virtual Network Function (NSC-VNF)** due to the fact that these functions are not compliant with SDN paradigm – e.g. Packet Gateway – PGW. An evolution of this model, which is compliant with the SDN principles, is when the control and the data plane functionalities are separated into different logical elements – **Virtual Control Network Function (VCNF)** and **Virtual Data Network Function (VDNF)**, respectively. When the VCNF uses the northbound of the SDN Controller, it is usually known as **SDN Application (SDN App)** – e.g. Service Chaining Function (SFC) control. In this case, the SDN-App software counterpart required to be deployed on the SDN Controller framework is coined as **SDN Controller App**. – e.g. SFC Plugin. The SDN App itself can also be deployed as a non-VNF. And this is the case considered in the context of this document, where we consider as VNFs only those VCNFs and VDNFs applications running on top of a cloud infrastructure, usually in Virtual Machines. In particular we refer to:

- **NFV applications**, as VCNFs and VDNFs implementing control and data plane functions in Virtual Machines on top of a virtualized infrastructure managed by a cloud management system, like OpenStack
- **SDN applications**, as SDN-Apps and SDN Controller Apps implementing control plane functions that are not virtualized network functions, and can even possibly run directly on top of a physical infrastructure
When the VDNF is controlled by an SDN Controller, i.e., an SDN-Controllable VNF, the naming proposed within SELFNET is Virtual Network Element (VNE) – e.g., Deep Packet Inspection (DPI) or a virtual switch, like Open vSwitch (OVS).

Figure 2 illustrates the several types of VNFs supported in SELFNET, as well as the so-called “SELFNET APP”, which is the SELFNET terminology to all the network functions that can be onboarded to the SELFNET framework. It includes the VCNF, SDN-App, SDN Controller App and the VDNF.

![Figure 2 Virtualized Network Function Types](image)

It is important to mention that, although the SDN Controller App is represented in the previous figure, it is always not a virtualized network function. It represents a software component embedded within the SDN Controllers software framework and has its own lifecycle management procedures (apart from the remaining VNFs).

Based on the several types of network functions described, two types of functions can be identified in SELFNET from the lifecycle management perspective:

1. NFV applications, i.e. virtualized network functions – the lifecycle management of these functions is based on ETSI NFV MANO principles;
2. SDN-Apps and SDN Controller Apps, i.e. non-virtualized network functions – the lifecycle management of these functions is defined within SELFNET.

For both NFV and SDN applications, the lifecycle management includes the following phases and actions:

- Instantiation, which covers the deployment phase of the application, and its installation into the SELFNET platform and infrastructure
- Configuration, which provides at least a basic configuration at the application level, and allows the application itself to properly start and interact with other SELFNET components. Possibly, full application configuration might be covered in this phase
• Modification, which basically consists into a modification of the application behaviour according to autonomic decisions taken by other SELFNET components in the SAL. This may involve a re-configuration of the application, a scale operation, or even a software update

• Start/Stop/Pause/Restart, which allows to apply basic operation actions, like start, stop, pause and restart of the application

• Termination, which is the counterpart of the instantiation, and covers the withdrawal of the application from the SELFNET platform and infrastructure.

It is also important to highlight that the onboarding phase is not covered in this document and is not part of the responsibilities of these lifecycle management procedures. Indeed, the onboarding refers to those mechanisms for uploading a new application into the SELFNET framework, in particular into the SELFNET APP catalogue maintained within the VNF onboarding sublayer. It is therefore a responsibility of the VNF onboarding sublayer to manage the uploading, retrieval, removal and update of NFV and SDN applications into the SELFNET APP catalogue.

From an architectural perspective, in order to accommodate the VNFs lifecycle management, and following the ETSI NFV MANO principles, the following architecture sublayers are impacted, as explicitly illustrated in Figure 3 (green boxes):

- VNF Management sublayer: responsible for managing the VNFs lifecycle, more precisely, the VNFs instantiation, configuration, start/pause/stop/restart and withdrawal;
- SON Control Plane sublayer: responsible for encapsulating the control plane VNFs, that is, exposing a common interface for the VNFM to manage the VNFs lifecycle management;
- SON Data Plane sublayer: responsible for encapsulating the data plane VNFs, that is, exposing a common interface for the VNFM sublayer to manage the VNFs lifecycle management.

Moreover, the VNFs lifecycle management can be addressed following two different approaches: dynamic and static. The dynamic approach is based on the fact that all the Virtual Machines hosting VNF applications are generic and then they are provisioned and configured with specific VNF content (software packages, configurations, etc.) dynamically when a new service need to be instantiated. It means that the assignment of such generic Virtual Machine resource, the copy of the VNF specific application/service in it, as well as the installation, the configuration and the execution of the VNF are done as part of its instantiation. As main advantage, it guarantees that only the VNF application content is provisioned which is incredibly fast when you compare with the static approach where the complete Virtual Machine image embedding a given VNF application need to be onboarded every time that any substantial update in the VNF application need to be done. Thus the dynamic approach makes easier updates, but as a counter side, it takes a significant time in installation and configuration. Both static and dynamic approaches have been considered and analysed in this document. In fact, it is the main intention of SELFNET to originally provide technical support for both of them in order to do not limit innovation and to be able to combine each other to get the better of these two approaches according to different use cases. Later on, it will be decided if both need to be integrated within the other components of the system or just one can provide the required capabilities but in any case, both functionalities are considered in SELFNET.

With respect to the non-VNFs, also illustrated in Figure 3 (in red), since these functions are not virtualized, the lifecycle management procedures do not involve the VNFM sublayer. In the SELFNET system architecture, the lifecycle management operations for this scenario are controlled by the Orchestration sublayer. Therefore, the following architecture sublayers are involved:
• Orchestration sublayer: responsible for managing the non-VNFs lifecycle, more precisely, the non-VNFs instantiation, configuration, start/pause/stop/restart and withdrawal;
• SDN Controllers sublayer: responsible for encapsulating non-virtualized SDN Controller Applications and/or SDN Applications, that is, exposing a common interface for the Orchestration sublayer to manage the non-VNFs lifecycle management.

Figure 3 SELFNET System Architecture – Network Functions Lifecycle Management

1.2 Objectives of this document

This document aims to:
• Provide a survey of reference architectures for ETSI NFV, MANO and SDN frameworks
• Provide an analysis of state of the art tools for NFV and SDN applications encapsulation and lifecycle management
• Design the functional architecture and components for the NFV applications encapsulation, including the specification of mechanisms and procedures for their uniform and common lifecycle management within the SELFNET framework
• Design the functional architecture and components for the SDN applications encapsulation, including the specification of mechanisms and procedures for their uniform and common lifecycle management within the SELFNET framework
• Report on the implementation of the NFV applications encapsulation prototype
1.3 Structure of the document

This deliverable is structured as follows:

- Section 2 first provides an overview of NFV and SDN reference architectures as defined in the relevant standardization bodies, then focuses on a brief description, analysis and evaluation of open source tools available in the state of the art for encapsulating NFV and SDN applications.

- Section 3 describes the SELFNET approach to encapsulation of NFV applications, including its high level architecture and functional decomposition, procedures and workflows for Virtual Network Functions lifecycle management. A detailed description of the implemented software prototypes is also provided, which includes interfaces specification and information models.

- Section 4 describes the SELFNET approach to encapsulation of SDN applications, differentiated between applications running outside and inside the SDN controller environment. The high level architecture approach is presented, along with interfaces and workflows to manage the lifecycle of SDN applications. A description of the implemented software prototype is also provided.

- Section 5 provides some concluding remarks.
2 State-of-the-art and gap analysis

This chapter aims to perform a state of the art analysis targeting encapsulation of NFV and SDN applications. SELFNET does not aim to implement or redesign NFV/SDN encapsulation technology from scratch. On the contrary, standard architecture proposals as well as the most community driven implementations of these proposals will be analysed and possibly adopted in order to promote SELFNET’s acceptance to the external community.

The deployment and management of Network Services is one of the main topics in discussion regarding 5G networks. Institutions like the European Telecommunications Standards Institute or the Open Network Foundation have already addressed the subject of NFV and SDN encapsulation, respectively, providing reference architectures to manage these components. Based on these reference architectures, new platforms have emerged to face the provision and management of NFVs and SDNs, such as OpenMano, OpenBaton, OpenStack Tacker or OpenDayLight. Despite following the same architecture, these platforms are developed independently, each one providing its own features and implementations.

Generic provisioning tools exist for quite some time and with increasing adoption. Deployments of complex IT systems like cloud services, during operations time, require performing maintenance and management with increasing complexity and time consuming. The exercise of these functions can incorporate tasks such as security updates across a large number of nodes. To manually perform this kind of chores in a considerable number of nodes may be achievable, but it is too much time consuming and error-prone to be desirable. Configuration management tools such as Puppet or Chef are commonly used to perform these tasks allowing its automation and reproducibility. Despite the fact that these tools have not been developed targeting VNFs or SDNs encapsulation, they may play a key role in the achievement of this assignment. Given the existence and the maturity of these provision and management tools, coupled with the fact that SELFNET’s goal is to reduce the provision time of new services, SELFNET can make use of some of these applications, using them as they currently exist or extending and combining them, in order to take advantages of their modus-operandi and avoid repeat issues already addressed by these platforms and common to the NFV/SDN encapsulation set of tasks.

The following sections provide an analysis of the state-of-the-art from different perspectives: section 2.1 takes care to summarize the NFV and SDN reference architectures as they are defined in the relevant standardization bodies, section 2.2 introduces the available tools (open source and commercial) for the management of NFV applications lifecycle, while section 2.3 provides an overview of the two more relevant SDN controllers currently used and deployed, with focus on their capabilities and approaches for applications encapsulation.

2.1 Reference architectures

When developing a concept or an idea it’s important to ensure references and standards so a common ground is provided and scattered efforts avoided. Network technologies are no different, promoting standards and safeguarding that every partner involved talks the same language and keeps the same terminology and building blocks, joining efforts across the same reference architecture.

In the area of NFV and SDN two bodies stand out when it comes to reference standards, for NFV the European Telecommunications Standards Institute, with architecture to manage and orchestrate VNFs and, for SDN, the Open Networking Foundation.
2.1.1 ETSI MANO

ETSI MANO refers to the proposal made by the European Telecommunications Standards Institute (ETSI) regarding the management and orchestration (MANO) activities of VNFs. MANO is a functional block responsible for the provisioning and lifecycle operations of VNFs and NSs as well as for the management of the NFVI by orchestrating the allocation of its resources (compute, storage, network). Figure 4 illustrates the NFV-MANO architecture proposed by the ETSI group [2].

![Figure 4 ETSI MANO architecture](image)

The proposed architecture foresees the existence of three main components:

- **VIM**: Manages the interaction with NFVI. It is responsible for discovering the available resources as well as managing its availability, allocation, connectivity and releasing.

- **VNFManager**: Responsible for managing the lifecycle of one or multiple VNFs performing its instantiation, scaling, updating, upgrading or termination.

- **NVFO**: This component manages the NS lifecycle operations such as on-boarding, instantiation, scaling, updating, topology definition and termination. It is also responsible to orchestrate NFVI resources across one or multiple VIMs.

The following complementary components are also proposed:

- **NS Catalogue**: Repository containing all the on-boarded NSs.

- **VNF Catalogue**: Repository containing all the on-boarded VNFs.

- **NFV Instances**: Repository holding information regarding all VNFs and NSs instances.

- **NFVI Resources**: Repository holding information about all the available/reserved/allocated NFVI resources.

Concerning the communication between these components the following interfaces are envisioned by ETSI group:

- **Os-Nfvo**: Used for the communication between OSS/BSS and the NFV Orchestrator (NFVO) allowing the NFVO to expose features such as: NS and VNF management;
NFVI’s resource allocation and release; reports accounting usage records or performance measurements.

- Or-Nvfm: Exposes VNF lifecycle management and configuration features provided by the NFVM to the NFVO.
- VeEn-Vnfm: Reference point between Element Management (EM) and VNFM allowing to perform lifecycle and configuration features in a specific VNF. This interface is only used if the VNFM cannot interact directly with the targeted VNF.
- VeNf-Vnfm: Interface between VNFM and VNF allowing to perform lifecycle and configuration operations in a specific VNF. This reference point is only used if it is possible for the VNFM to directly interact with a VNF and thus an EM is not available.
- Vnfm-Vi: Promotes the communication between the VNF Manager and the Virtualized Infrastructure Manager allowing NFVI resource reservation information, allocation and release.
- Nf-Vi: Allows the interaction between Virtualisation Infrastructure Manager (VIM) and NFV Infrastructure (NFVI) allowing VIM to allocate, migrate, update and terminate VMs provided by the NFVI. This interface also allows for the VIM to configure connections between VMs as receiving event information (failures, measurement results, usage records) from the NFVI.
- Or-Vi: Allows the NFV O to interact with NFVI’s resources through the VIM enabling it for instance to allocate, release or update a NFVI resource or add, delete and update a VNF software image. Furthermore, it enables NFVO to retrieve information regarding configuration, events, measurement results or usage records of NFVI resources.

Pertaining to the definition of a NS (and all the components that may comprise it) ETSI foresees the existence of five types of information elements (NS, VNF, PNF, VNFFG, VL) [2]. These information elements may be used in two different contexts [2]:

- as descriptors in a catalogue at the on-board stage
- as instance records in a runtime context at the instantiation stage.

As a matter of fact, to deploy an information element and accordingly to ETSI MANO, initially an information element will have to be on-boarded by the NFVO and its integrity and authenticity evaluated. Upon validation, an information element will be placed in a catalogue becoming available to be instantiated. In this stage (on-boarding), an information element will be defined by the following element descriptors:

- NSD (Network Service Descriptor). Deployment template for a NS containing or referencing all the components that are part of it.
- VNFD (Virtual Network Function Descriptor). Deployment template for a VNF describing its deployment and operational requirements.
- VNFFGD (Virtual Network Function Forwarding Graph Descriptor). Contains the topology of a NS describing how the VNFs, PNFs and VLS are connected.
- VLD (Virtual Link Descriptor). Describes the resource requirements needed for a link between VNFs, PNFs and endpoints of the NS.
- PNFG (Physical Network Function Descriptor). Contains the connectivity, Interface and KPIs requirements of Virtual Links to an attached Physical PNF.

Instantiation (second stage previously mentioned) will be triggered either by human intervention or by an autonomous routine. At this stage, previously on-boarded NS or VNF will be requested to be instantiated in the NFVI resources. To achieve this NFVO or VNFM will then receive instantiation parameters (from the entity initiating the instantiation) and fetch the descriptors that define the NSs or VNFs to instantiate from the catalogues. As a
result of the instantiation operation records are created to represent the newly created instances (NSR, VNFR, VNFFGR, VLR, PNFR). They are a result of combining NSD, VNFD, VNFFGD and VLD information elements with input/output parameters exchanged via different interfaces produced and/or consumed by NFV-MANO functional blocks [2].

2.1.2 ONF SDN

The Open Networking Foundation (ONF) [3], an industry-leaded group that aims at improving networking through the adoption of Software Defined Networking (SDN) [4] proposes an architecture [5] that enables decoupling of control and data planes, provides centralized control and exposes abstract network resources to external applications.

The SDN Architecture proposed by the ONF comprises three layers:

- The Application Plane is where SDN applications reside. They communicate their network requirements toward the Controller Plane via the application-controller plane interface (A-CPI).
- In the Controller Plane, the SDN controller orchestrates the applications’ demands for limited network resources according to policies.
- The Data Plane contains groups of resources which expose their capabilities toward the Controller Plane via the data-controller plane interface (D-CPI)

In Figure 5 we can see how these layers interact. A Service Consumer (client, user, customer) in the application plane starts a session and exchanges some management-control operations to invoke services from a SDN provider via the A-CPI of the SDN Controller. The main tasks of the SDN Controller are to virtualize and orchestrate the resources used by the Service Consumer onto resources from the Resource Groups under its control. Service Consumer perceives those virtual resources as its own resources.

ONF’s recommendation for a SDN architecture extends the SDN basic models with a few key extensions which include sharing resources a) among multiple clients, b) dynamically, c) in an optimum way.

The central entity is the SDN controller. SDN is modelled as a set of client-server relationships between SDN controllers and other entities that may themselves be SDN controllers. In its role as a server, an SDN controller may offer services to any number of clients, while an SDN controller acting as client may invoke services. Figure 6 illustrates this and also the different roles of users (administrator, user, provider) as well as the different type of contexts (client context, server context).
The SDN controller receives client requests via A-CPIs and satisfies them by virtualizing and orchestrating its underlying resources. The controller consumes the resources via the D-CPIs. When the network environment changes and client demands change, the SDN controller is responsible for continuously updating network and service state toward a policy-based optimum configuration. The A-CPIs and the D-CPIs are reference points for information hiding, traffic and namespace isolation, and policy enforcement.

With regards to roles, the administrator has authority to configure the SDN controller itself. The administrator configures the controller with server contexts to access underlying resources. He/she is also on charge of creating a client context for each of its clients, which includes allocation of underlying resources to the client. The administrator configures each client context with policies that defines the actions and bounds permitted to the client. An administrator may modify a client context during its lifetime, and may destroy a client context if the client relationship terminates. Clients or service consumers request services from SDN controllers that play the role in this case of providers and they achieve their data transfer and data processing objectives as users of the corresponding resources.

![SDN controller and contexts](source ONF SDN architecture 1.1)

Issue 1.1 of ONF’s SDN architecture incorporates the concept of management-control continuum (MCC) which follows the principle that the functions of management and of control are largely, if not entirely, the same. Therefore, traditional business and operations support system (BSS/OSS) functions could be merged with control. With regards to FCAPS functions (fault, configuration, accounting, performance, security), some of them may be partly or entirely outside the scope of a given SDN controller implementation. It is expected that, in the near term, much existing FCAPS and OSS/BSS functionality will remain with current OSS/BSS and Network Management System (NMS) entities.

Recursion is supported by the SDN architecture. It refers to the case where client consumes resources and services offered by a server, which may itself be a client of one or more additional servers. It supports: a) Hierarchical recursion in which higher-level SDN
controllers orchestrate a broad scope of resources and services across one or more lower-level SDN controllers with narrower scopes and less abstract resources; an example of this recursion can be seen in Figure 6: the so-called SDN Green Controller plays the role of service consumer b) Neighbour recursion in which SDN controllers peer to deliver services across SDN control domains, therefore, all of them expose comparable levels of abstraction and service, and any SDN controller could act as either client or server to its neighbours.

Applications in SDN model are client entities that request services from an SDN controller. This requires a session for implementing the required management-control interactions via the A-CPI interface. Since the SDN controller is the intelligent entity in the picture, the service consuming application can express its needs in terms of intents. These would be non-prescriptive description, intents describe what is needed (e.g. “low latency”, “high speed”, “20 Mbps”, etc.) and don’t prescribe how to provide it. Then, such descriptions are fully portable across different controller planes and renders application plane and controller plane implementations mutually independent. Intent descriptions can also support request conflict resolution by controller planes by including appropriate request context & excluding unnecessary request constraints.

BOULDER project [6] is developing an Intent-based NBI for ONF’s SDN architecture. It provides a scriptable runtime that maps to Intent structures that allows users to create portable applications across different underlying controllers.

2.1.3 Applicability to SELFNET

ONF SDN and ETSI NFV architectural approaches complement each other. While NFV offers the possibility to move the execution of network functions, traditionally implemented by dedicated hardware, into virtualized environments for highly improved flexibility and manageability of services, SDN provides those dynamic service composition and network connectivity provisioning that can help to chain VNFs into NFV network services (NS). In such a way, VNFs and NSs become SDN resources to compose and dynamically optimize services. Moreover, an SDN controller may be itself implemented as a VNF, and it may be expected that OSS/BSS/NMS/EMS will be or become VNFs, along with utility services such as Domain Name Server (DNS), Dynamic Host Configuration Protocol (DHCP), etc.

SELFNET is exactly targeting and addressing this combination of NFV and SDN concepts and solutions. Indeed, with its SON oriented architecture where high automation, dynamic management and provisioning of virtualized services are core functions, SELFNET implements a deep integration and coordination of SDN control applications on top of NFV infrastructures and functions. The encapsulation of NFV and SDN applications addressed by this document, at both design and prototype implementation level, is one of these combination aspects and offer to SON management and orchestration components those primitives to easily and uniformly manage the lifecycle of SELFNET sensors and actuators.
2.2 Lifecycle management of NFV applications

Management and orchestration of VNFs lifecycle is an important responsibility when related to provision NFV services. The following subsections provide an overview on state-of-the-art tools for NFV applications lifecycle management.

2.2.1 NFV MANO implementations

2.2.1.1 OpenMANO

OpenMANO [7] is Telefonica’s open source approach to the management and orchestration components of ETSI NFV ISG standardization. Figure 7 shows OpenMANO’s architecture and how the components are mapped in ETSI’s NFV Architecture. The former is composed by three components:

- openmano-gui – web based graphical user interface that uses OpenMANO’s RESTful API to interact with the system
- openmano – OpenMANO’s NFV Orchestrator that exposes the catalogue (e.g. VNF template creation) and service management (e.g. NS instantiation) through a REST based northbound API and uses southbound connectors to interface with VIMs
- openvim – Telefonica’s VIM implementation following ETSI’s NFV document on performance best practises [8]. openvim interfaces openmano through a REST based interface and allows management of NFV Infrastructures composed of compute nodes and a SDN controller.

OpenMANO’s latest version, 0.4, supports the creation and deletion of VNF descriptors using the Yaml format and which are based on ETSI’s specification. These VNF descriptors are more focussed on I/O performance through numas, CPU, memory and interface requirements, when compared with other platforms descriptors which are based on cloud-orchestration templates. Regarding network services, OpenMANO allows their design using the available VNFs and also service instantiation and deletion. Although OpenMANO has its own implementation of the VIM focussing on network performance and using Floodlight as the SDN controller, the latest version added support for OpenStack and OpenDayLight, which are more widespread within the open source community.
Moreover, the latest version also addressed one of the shortcomings in the first version, the support for multiple VIMs.

The only thing missing from ETSI specification in OpenMANO is the VNF Manager. This component is in charge of managing the VNF lifecycle and thus, this platform is unable to manage the VNF lifecycle and configuration. Recently, Telefonica participated with OpenMANO in a demo for the newly-created ETSI project, Open Source MANO [9], where the openmano and openvim components were used for the role of the Resource Orchestrator and VIM respectively [10].

2.2.1.2 OpenBaton

OpenBaton is an open source project by Fraunhofer FOKUS compliant with the ETSI MANO functional components [11]. Using ETSI MANO as a reference, Figure 8 provides a general view on how OpenBaton implements the specification.

![Figure 8 OpenBaton overview](image)

As seen in Figure 8, NFVO is the main component provided by OpenBaton, using Java and spring.io framework in its implementation. Currently OpenBaton provides a VIM plugin compatible with OpenStack allowing its seamless integration with the NFVO. The NFVO supports the addition of more than one VIM based on the same or different implementations. OpenBaton’s architecture encourages its extension, and alternative VIMs can be supported via a plugin mechanism and a dedicated SDK library for implementation of VIM-specific drivers to be easily integrated with the NFVO. OpenBaton provides a generic VNFM as well as a set of additional SDK libraries to be used for the development of a specific VNFM. The generic VNFM implementation follows the ETSI MANO specification, using an EMS to interact with a given VNF. The communication between the VNFM and the NFVO or EMS is handled through a message bus. If the current VNFM implementation does not address a specific use case scenario, a new VNFM may be developed using the SDKs provided by OpenBaton. The way that this VNFM interacts with each VNF has no restrictions from OpenBaton so the workflow and communication format is free to be defined by developers. The interaction between the developed VNFM and the NFVO is performed via a REST interface or AMQP protocol message system. OpenBaton follows the ETSI MANO proposal, using templated descriptors for the definition of the resources (compute, storage, and network) as well as the apps (VNFs, NSs) to be managed, allowing the exchange and persistence of this information.
OpenBaton’s modular implementation and its plugin based architecture allows extensibility and the integration of the solution without having to modify or understand its core implementation. The fact that this implementation follows ETSI MANO specifications pushes its adoption by the industry and provides a consensual roadmap on what features are to be implemented and how the interfaces will be implemented.

2.2.1.3 OpenStack Tacker

Tacker is a network service lifecycle management project within OpenStack that was started during the Juno cycle with contributors from Brocade, Cisco, Intel, and other companies [12]. It basically provides an Open NFV Orchestrator with in-built general purpose VNF Manager used to deploy and operate VNFs on an NFV Platform. It is based on the ETSI MANO Architectural Framework and provides a full functional stack for end-to-end VNF orchestration on top of OpenStack managed infrastructures. However, its compliance with ETSI MANO defined interfaces, workflows and procedures is still at an early stage, while its architecture is more aligned to the OpenStack framework.

In practice, the Tacker project is still in progress and not yet available in the official release, and implements an NFVO with an integrated general purpose VNFM, operating over the other components of OpenStack used as VIM, as depicted in Figure 9. The Tacker NFVO includes initial functions to provide an end-to-end orchestrated set of VNFs, interconnected through a service chain described in a VNF Forwarding Graph (VFNFG) and possibly deployed across multiple VIMs. It also includes mechanisms for VIM resource check and allocation with VNF placement policies to guarantee an efficient positioning of the VFN over the available resources. At the generic VNFM level, it supports the basic VNF life-cycle (load descriptor, start, stop), its initial configuration and performance and health monitoring. The VNF catalogue uses descriptors compliant with the TOSCA NFV profile.

Currently, Tacker has been mostly deployed in support of a set of proof-of-concept use cases targeting service providers’ NFV environments for the virtualization of Customer Equipment (CE), Customer Premise Equipment (CPE) and Provider Edge (PE) functionalities.

![Figure 9 OpenStack Tacker overview](image_url)
2.2.1.4 Commercial implementations

In addition to the open source implementations described above, a number of NFV MANO commercial solutions are available from vendors like Cisco, Brocade, HP, Juniper among the others.

Taking Cisco as an example, its service portfolio includes a set of commercial NFV MANO solutions, thus being well aligned with the current trends and requirements imposed by network and service providers [13].

In particular, the core of the Cisco NFV orchestration portfolio is built by the combination of the Elastic Service Controller (ESC) and the Network Services Orchestrator (NSO), as depicted in Figure 10. The Cisco ESC provides an advanced lifecycle management platform for VNFs and is built as an open and modular system. It provides a single point of control to manage all aspects of VNF lifecycle for generic VNFs in a dynamic environment. Leveraging on industry standards and open APIs, it enables to control the full lifecycle of all the virtualized resources, whether using Cisco or third-party VNFs, allowing the integration of best-of-breath industry solutions. Cisco ESC provides VNF provisioning and configuration to create, read, and delete VNFs including day-zero configurations (management IP, gateway, smart-licensing parameters, and any other VNF supported bootstrap configurations). A dedicated engine takes care of monitoring VMs resource usage and status by means of SNMP, ICMP and customizable KPI collection. Moreover, the Cisco ESC performs dynamic provisioning and scaling of individual or groups of VNFs within multitenant environments. It support the creation of Tenants, Flavors, and Networks in Openstack as well as the orchestration of VNFs within a virtual infrastructure domain.

On top of the Cisco ESC, the NSO enabled by Tail-f simplifies the process of provisioning and controlling applications and services in both physical and virtual networks. It decouples network services from specific components, while automatically configuring the network according to the service specifications. The Cisco NSO communicates with Cisco ESC using the open NETCONF protocol and YANG based data models. While ESC manages VNFs at a device level, NSO takes care of the whole service lifecycle. Together, they build a complete orchestration solution that spans across both physical and (NFV) virtual infrastructure.

Therefore, Cisco ESC and NSO provide advanced VNF lifecycle management capabilities through an open, standards-based platform compliant with the ETSI NFV MANO reference architecture thus enabling the interoperation with any standards-based NFV orchestration system.

![Figure 10 Cisco NFV service orchestration portfolio (source: Cisco ESC datasheet)](image-url)
2.2.1.5 Applicability to SELFNET

Table 1 below provides a deeper technical overview of the NFV MANO open source tools introduced in the previous sections. Each of them (OpenMANO, OpenBaton, Tacker) are evaluated in the table against a set of crucial features and functionalities for VNF lifecycle management. The main aim here is to identify which is the most advanced, aligned to standards and mature tool for adoption, extension and use in SELFNET.

According to this evaluation, the SELFNET project has selected OpenBaton as the reference open source tool for the implementation of the software prototype for NFV MANO based VNFs lifecycle management. The main reason behind this choice is that OpenBaton is currently in a mature state, with most of the ETSI MANO functionalities supported and implemented. In particular, NFVO and VNFM roles are clearly separated in different and stand-alone components. On the other hand, OpenMANO suffers of a number of proprietary implementations, data structures, formats and choices in general that limit its full compliance with the standards. Also, the VNFM component is not available, as well as an interface or communication channel with VNFs. Similarly, Tacker is still in its incubation phase. Even if it is an OpenStack project, thus with high impact potential in a large community, most of key ETSI MANO features are still in the development roadmap.

Moreover, OpenBaton is highly extensible and offers various SDKs to ease the integration with external applications, as well as the development of new drivers and plugins for specific VIMs. This has been considered as a key feature, especially in terms of additional extensions and improvements to be possibly applied either during or after the SELFNET project.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>OpenMano</th>
<th>OpenBaton</th>
<th>Tacker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Service Composition</td>
<td>Does the NFVO support the composition of network services involving multiple VNFs?</td>
<td>Yes</td>
<td>Yes</td>
<td>This feature is still at an early stage and it is not fully implemented. It's part of the Tacker short-term roadmap.</td>
</tr>
<tr>
<td>Network Service Catalogue and Descriptor</td>
<td>The NFVO has a network service catalogue? If yes, what is the interface to onboard the network service? What is the format of the network service?</td>
<td>Yes, onboarding of network services is available through the GUI, REST API and the CLI. The network service descriptor uses a proprietary format compliant with the ETSI specification</td>
<td>Yes, it has a catalogue available through GUI or REST API. The network service descriptor uses a proprietary format compliant with the ETSI specification</td>
<td>Not yet.</td>
</tr>
<tr>
<td>Network Service</td>
<td>Does the NFVO support the deployment of</td>
<td>Yes</td>
<td>Yes</td>
<td>Not yet as a full function of the NFVO. The</td>
</tr>
<tr>
<td><strong>Deployment and Management</strong></td>
<td><strong>VNF Catalogue and Descriptor</strong></td>
<td><strong>VNF Manager</strong></td>
<td><strong>VNF Lifecycle Management</strong></td>
<td><strong>External VNF Management Control</strong></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------</td>
<td>-----------------</td>
<td>-----------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>network services? Can they be deployed in a multiple VIM scenario? Does the NFVO support network service lifecycle management? start and stop. the network service descriptors. NFVO support service lifecycle management.</td>
<td>The NFVO has a VNF catalogue? If yes what is the interface to onboard the VNFs? What is the format of the VNF descriptor? Yes, the onboarding of VNFs is done using the CLI and REST API. It uses a proprietary format compliant with ETSI specification</td>
<td>Yes and the use of external VNF managers is not contemplated There is a generic VNF manager but it is possible to extend or re-implement it using the vnfm-sdk</td>
<td>Not available</td>
<td>Not available</td>
</tr>
<tr>
<td>deployment on multiple VIMs is enabled by the VNFM anyway.</td>
<td>Yes, the onboarding of VNFs is done using the GUI or REST API. The VNF descriptor is compliant with the ETSI specification and is in JSON format</td>
<td>No, the current Tacker version embeds NFVO and VNF manager functionalities</td>
<td>Yes it’s possible to provide scripts for different lifecycle events. These actions can be performed using the GUI.</td>
<td>Yes, it exposes a REST API and an sdk is also available to use NFVO from java applications</td>
</tr>
<tr>
<td>Yes, the onboarding is done through REST APIs. VNFDs are TOSCA based</td>
<td>No, the current Tacker version embeds NFVO and VNF manager functionalities</td>
<td>Yes, The Tacker VNFM supports instantiation, termination, healing and auto-scale. It can be done either through REST APIs or CLI</td>
<td>Yes, Tacker supports auto-scale at the VNFM</td>
<td></td>
</tr>
<tr>
<td><strong>Monitoring System</strong></td>
<td><strong>Does the NFVO has its own monitoring system? If yes, does it also has analytics? Does it support external monitoring systems?</strong></td>
<td><strong>No and VNF monitoring is not contemplated</strong></td>
<td><strong>Not available.</strong></td>
<td><strong>Not available</strong></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------</td>
<td>------------------</td>
<td>------------------</td>
</tr>
<tr>
<td><strong>VNF Monitoring</strong></td>
<td><strong>Does the NFVO monitor VNFs? Support for resource and application monitoring?</strong></td>
<td><strong>Not Available</strong></td>
<td><strong>Openbaton allows the monitoring of the VNFs via Zabbix. To use this feature it is required to install and configure Zabbix server. <a href="http://openbaton.github.io/documentation/nfvo-installation/">http://openbaton.github.io/documentation/nfvo-installation/</a></strong></td>
<td><strong>Yes, a monitoring driver is available for the VNFM. The current version only performs VMs network connectivity checks</strong></td>
</tr>
<tr>
<td><strong>Supported IaaS Platforms</strong></td>
<td><strong>What IaaS platforms are support by the NFVO?</strong></td>
<td><strong>OpenVIM (Telefonica implementation of NFV VIM) and Openstack</strong></td>
<td><strong>Openstack, but it’s possible to write other plugins for other VIMs allowing its integration.</strong></td>
<td><strong>OpenStack</strong></td>
</tr>
<tr>
<td><strong>Multiple VIM support</strong></td>
<td><strong>Does it support a distributed VIM scenario?</strong></td>
<td><strong>Yes</strong></td>
<td><strong>Yes, it is possible to register multiple Points of Presence.</strong></td>
<td><strong>Yes</strong></td>
</tr>
<tr>
<td><strong>NFVO – VIM Interface</strong></td>
<td><strong>Details on the NFVO – VIM interface</strong></td>
<td><strong>The interface with OpenVIM is proprietary and with Openstack is through its API</strong></td>
<td><strong>The NFVO uses a plugin mechanism for interacting with VIMs.</strong></td>
<td><strong>It uses Heat to interact with other OpenStack services (Nova, Neutron, etc.)</strong></td>
</tr>
<tr>
<td><strong>VNFM – VIM Interface</strong></td>
<td><strong>Details on the VNFM - VIM interface</strong></td>
<td><strong>Not available</strong></td>
<td><strong>The generic VNFM does not interface with the VIM. The VIM plugins offered by OpenBaton can be however imported and used by the VNFM if necessary</strong></td>
<td><strong>It uses Heat to interact with other OpenStack services (Nova, Neutron, etc.)</strong></td>
</tr>
<tr>
<td>VNFM – VNF Interface</td>
<td>Details on the VNFM – VNF interface</td>
<td>Not Available</td>
<td>Achieved through an EMS responsible for the encapsulation of a VNF. It interacts with Generic VNFM through RabbitMQ queues.</td>
<td>A Management Driver Framework is embedded in the VNFM. No much details are available anyway.</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NFVO – Monitoring</td>
<td>Details on the NFVO – Monitoring System interface</td>
<td>Not Available</td>
<td>Not Available</td>
<td>Not available</td>
</tr>
<tr>
<td>NFVO Northbound Interface</td>
<td>Details on the NFVO NBI</td>
<td>It has GUI, REST API and CLI</td>
<td>Mainly GUI and REST. A CLI is also available for plugin management</td>
<td>Currently REST and GUI</td>
</tr>
<tr>
<td>User Access and Management Control</td>
<td>Does it support authentication and authorization of users? Does it support multi-tenancy?</td>
<td>It supports authentication but there’s no authorization mechanisms. It also supports multi-tenancy.</td>
<td>NFVO supports authentication and also multi-tenant environments.</td>
<td>Leverage on OpenStack features.</td>
</tr>
<tr>
<td>Security Features</td>
<td>The NFVO has any security features?</td>
<td>Besides authentication nothing particularly relevant</td>
<td>Besides authentication nothing particularly relevant</td>
<td>Leverage on OpenStack features.</td>
</tr>
<tr>
<td>License</td>
<td>License for the NFVO and its components</td>
<td>Open Source (Apache v2.0)</td>
<td>Open Source (Apache 2.0)</td>
<td>Open Source (Apache 2.0)</td>
</tr>
<tr>
<td>Programming Language</td>
<td>Python</td>
<td>Java</td>
<td>Python</td>
<td></td>
</tr>
<tr>
<td>Documents Availability, Community Support</td>
<td>What is the availability and access to documentation? Is there any community actively supporting it?</td>
<td>There is some documents available but the community is still small</td>
<td>There are some documents describing how to work with OpenBaton, but the community is quite small.</td>
<td>Documentation is quite poor and it’s not clear how much Tacker is used in/by the OpenStack community</td>
</tr>
<tr>
<td>Last Update</td>
<td>Date</td>
<td>March, 2016</td>
<td>March, 2016</td>
<td>March 2016</td>
</tr>
</tbody>
</table>

Table 1 NFV MANO open source tools comparison
2.2.2 Dynamic provisioning tools

In addition to pure ETSI MANO based implementations and tools, Ubuntu Juju, Puppet, Ansible or Chef, are open source applications which, even though developed out of NFV scope, can enable a dynamic management of VNFs lifecycle.

2.2.2.1 Ubuntu Juju

Ubuntu Juju [14] is a software solution for dynamic provisioning of services. The following are the execution environments supported by Juju:

- Metal-as-a-Service (MaaS): It enables Juju to employ the physical resources for dynamic provisioning of services from scratch.
- Bare Metal: It enables Juju to provision services directly over an operating system. It works primarily over Linux but it has preliminary support for Windows deployments as well.
- Light Containers (LXC, Vagrant): It enables Juju to make use of the lightweight virtual layer in order to perform the dynamic provisioning of services over virtual containers.
- Virtual Environments (VMWare vSphere, KVM): It enables Juju to make use of the virtual layer in order to perform the dynamic provisioning of services over virtual resources.
- Private Clouds (OpenStack, Scaleway): It enables Juju to employ the virtual layer of a private cloud provider in order to perform the dynamic provisioning of services over tenant-aware virtual resources.
- Public Clouds (Azure, HP Cloud, Juvent, DigitalOcean): Similarly, it enables Juju to use the virtual layer of a public cloud provider for dynamic provisioning of services over tenant-aware virtual resources.

Regarding the encapsulation of applications, Juju refers to its encapsulation method as a Juju Charm. A Charm is a set of files that represents the encapsulated version of a service application. It requires the creation of a number of files that will be executed as part of the actions to be invoked in the different states that a service is holding during the provisioning of the service. Figure 11 shows the basic state diagram managed internally by the latest version of Juju 1.25.
As can be seen in Figure 11, an encapsulated Juju application invoked an installation script. Consequently, the orchestration among different components is managed by means of the creation of relationships among different encapsulated applications. These relationships are used to control the orchestration of the services and to link configuration options between different services, thereby enabling the share of information among these encapsulated services. After a service has fulfilled all its requirements in terms of relationships with other applications, all the configurations are applied. The configuration of values is indicated using a standard mechanism to define key/value pairs to represent the configuration parameters associated to a given service. Specifically, Juju utilizes the YANG model to represent such configurations. These configurations are passed to Juju upon the instantiation of a new service. Then, the Charm will take this configuration as input parameters to invoke all the actions implemented in the encapsulation. These actions cover installation, configuration, reconfiguration, upgrade and uninstallation. Juju also enables an optional way to define actions over a service. A well-known subset of actions could be used in order to provide a common subset of management capabilities for a VNF/PNF application, such as start, stop, restart, and reload configuration.

2.2.2.2 Puppet

Puppet is a declarative, model-based configuration management solution that uses Puppet language in the definition of the state of IT infrastructure enabling the use of templates for repetitive tasks. Puppet is available under Apache 2.0 license and support all major Linux and Unix platforms as well as Mac OS X and Windows [15]. Puppet usually runs in an agent/master architecture, where a Puppet master server controls the configuration information and managed agent nodes only request their own configuration catalogues [16]. In this architecture, managed nodes run the Puppet agent application, usually as a background service while one or more servers run the Puppet master application, usually in the form of Puppet Server. Periodically, Puppet agents will send facts to the Puppet master and request a catalogue. The master will compile and return that node’s catalogue. Once it receives a catalogue, Puppet agent will apply it by checking each resource the catalogue describes and ensuring that its local state matches the state specified in the catalogue. Puppet agent nodes and Puppet masters communicate via HTTPS with client-verification. To promote knowledge sharing, Puppet Forge provides more than three
thousand modules (reusable, sharable pre-written Puppet code for automating admin tasks) created by Puppet Labs and by Puppet community members [17]. Puppet is currently also addressing the topic of deployment/management Openstack. A wide set of modules are currently under development aiming to manage Openstack components such as keystone, nova, ceilometer, heat or horizon [18].

Focusing on the encapsulation of NFV and SDN applications, Puppet can be used both at the instantiation as well as further down the road for configuration when the NFV/SDN app is already running. At the instantiation phase, by integrating a Puppet agent, it is possible to correctly download, install and configure all the packages required by a SDN/NFV. This will be achieved by installing a Puppet agent responsible for autonomously contact the Puppet master, request its catalogue and apply it to the NFV’s/SDN’s resources. The link between the Puppet agent and the Puppet master will persist after the instantiation and so provides a communication link with an instantiated SDN/NFV. Using the ETSI MANO as a reference, Puppet agent may assume the role of an EM (Entity Manager) and Puppet master the role of VNFM and thus may enable the performance of lifecycle management on VNFs and SDNs.

2.2.2.3 CHEF

Chef is a platform that allows to automate how applications are configured, deployed and managed across a network, no matter its size using a Ruby-based DSL [19]. It handles configuration by packing details into an artefact that it calls recipes. Chef provides a way to quickly deploy entire environments instead of only single applications. Chef operates as a master-client model being different roles assumed by different machines/resources [20]:

- Chef Server: This is the central location that stores configuration recipes, cookbooks, and node and workstation definitions. It is the central machine that every other machine in the organization will use for deployment configuration.
- Chef Nodes: Chef nodes are the deployment targets that are configured by Chef. Each node represents a separate, contained machine environment that can be on physical hardware or virtualized. These operating system environments each contain a Chef client application that can communicate with the Chef Server.
- Chef Workstations: Chef workstations are where Chef configuration details are created or edited. The configuration files are then pushed to the Chef server, where they will be available to deploy to any nodes.

It is worth to mention that it is possible for one machine to adopt two or all of the previously mentioned roles. Chef is currently one of the automation tools being used to manage the deployment of Openstack [21]. It provides a considerable set of recipes that allows to deploy different flavours of Openstack or specific components that comprise it.

Regarding the encapsulation of NFV and SDN applications, it enables an approach like the one previously described for Puppet. Chef Nodes may be installed in VNF and SDN applications allowing the configuration and control of its environment by providing a communication link with these entities. Centralized Chef components (Chef Server) may assume the role of a VNFM and thus provide lifecycle management features to each VNF/SDN that interacts with if through an active Chef Node.

2.2.2.4 Ansible

Ansible [22] is an IT automation tool. It can configure systems, deploy software, and orchestrate more advanced IT tasks such as continuous deployments or zero downtime rolling updates [23].

Ansible’s main goals are simplicity and ease-of-use. It also has a strong focus on security and reliability, featuring a minimum of moving parts, usage of OpenSSH for transport (with
an accelerated socket mode and pull modes as alternatives), and a language that is designed around auditability by humans—even those not familiar with the program.

Ansible manages machines in an agent-less manner. There is never a question of how to upgrade remote daemons or the problem of not being able to manage systems because daemons are uninstalled. Because OpenSSH is one of the most peer-reviewed open source components, security exposure is greatly reduced. Ansible is decentralized—it relies on your existing OS credentials to control access to remote machines. If needed, Ansible can easily connect with Kerberos, LDAP, and other centralized authentication management systems.

Configuration files are mainly written in the YAML data serialization format due to its expressive nature and its similarity to popular markup languages. Ansible can interact with clients through either command line tools or through its configuration scripts called Playbooks.

![Ansible Architecture](image)

**Figure 12: Ansible Architecture [24]**

### 2.2.2.5 Applicability to SELFNET

Among the dynamic tools investigated in section 2.2.2, the SELFNET project has selected Juju for prototyping validation. This is because Juju has significant advantages over its competitors. Firstly, Juju offers a common and standard abstraction of heterogeneous execution environments relevant to SELFNET, like MaaS, bare metal, containers and virtualized environments as well as private and public clouds, and it has implemented support for a significant number of them. Juju enables any user to use the same encapsulated service in order to be deployed in various heterogeneous execution environments.

Secondly, the combination of the actions imposed by the Juju charms to be encapsulated and the optional actions to be provided in a Charm can be used to cover all the set of requirements of SELFNET with respect to the dynamic provisioning of services.

Lastly, it is particularly important to mention that Juju enables its usage over MaaS, thus enabling to employ physical resources for dynamic provisioning of services from scratch, which is a key feature for management architectures as the one proposed in SELFNET towards the achievement of the 5G KPI related to the reduction of the service provisioning time.

### 2.3 Lifecycle management of SDN applications: SDN controllers

SDN platforms allow hosting and management of SDN applications. These applications bring intelligence and versatility in the network. Over this section two platforms stand out, OpenDaylight, from the Linux Foundation and ONOS.
2.3.1 OpenDaylight

OpenDaylight (ODL) project [25] is an industry consortium hosted by The Linux Foundation that is building an open source framework and platform to accelerate the adoption of SDN and create as well a solid foundation for NFVs. In this way, ODL project main goal is to enable interoperability among products and stimulate SDN application development by providing a direction for future SDN technologies.

Nowadays, with SDN it is possible to optimize existing networks to fit our needs and it is easier to adapt them if our requirements change. ODL is building a common platform that can be configured in multiple ways to solve all network challenges. Moreover, ODL integrates open standards and open APIs in order to deliver a SDN platform that makes networks today more programmable, configurable and intelligent.

ODL follows a model-driven approach to describe the network and all the functions to be performed on it. Moreover, ODL by using YANG data structures in a common data store and messaging infrastructure, allows for fine-grained services to be created and then combined together to solve more complex problems.

ODL supports multiple set of protocols, such as Openflow, NETCONF, BGP/PCEP, CAPWAP and interfaces with Openstack and Openvswitch platforms, that improve the programmability of modern networks and solves a wide range of common network problems according to the user’s needs.

In the Figure 13 it is illustrated the architecture of the ODL Beryllium platform.

![Figure 13 ODL architecture](image)

Regarding SDN Apps encapsulation and implementation, OpenDaylight uses software tools such as Java interfaces, Maven, Karaf and OSGi. Java Interfaces are used for event listening, specifications and forming patterns. On the other hand, Maven is hosted by Apache Software Foundation [26] that allows developers to build and manage their applications for ODL by creating a standard way to build and publish projects. ODL uses Karaf [27] to deploy different kind of applications created by using Maven.

After the development of an SDN Application, the developer will load the SDN App in the OSGi (Open Service Gateway Interface) environment, which is a framework that provides a place for the modularization of applications into smaller bundles. ODL implements an OSGi environment by using Karaf.
Karaf, as shown in Figure 14, is based in OSGi which provides a lightweight container where various components and applications can be deployed [5]. Therefore, by using karaf, any user can upload their own features and build their own versions of the ODL controller. Running Karaf starts all the Java bundles installed as jar files in the OSGi environment. In this way, the Karaf shell is our main portal for managing all SDN applications/Java bundles. Karaf has important features, such as:

- **Hot deployment** (handle deployment of bundles automatically);
- **Dynamic configuration** (new configurations will be implemented on services);
- **Logging System** (centralized logs);
- **Provisioning**;
- **Native OS integration** (Karaf integration on our Operating System);
- **Extensible Shell console**;
- **Remote access**: (SSH);
- **Managing instances**;
- **Security framework**.

![Figure 14 Karaf Structure](image)

Moreover, on this new version of ODL, developers made an effort to evolve the controller so that it could support intent-based networking (IBN) by providing a Network Intent Composition that offers wide flexibility for intent-based management. Also, there is a new project inside ODL called NetIDE [28] that their main goal is to provide an intent-based network modelling. IBN makes networks more scalable, since the developers do not need to be aware of the infrastructure environment. Therefore, the introduction of new SDN applications becomes much faster and flexible, because the developer focuses more on application development instead of focusing on a further understanding of how applications work with the infrastructure. It also should be highlighted that IBN is portable and vendor-agnostic, because an SDN application developed for a certain SDN environment can be easily ported to another SDN environment without involving the app developer. Besides, this means that an SDN application for a certain SDN controller can be run on another vendor’s controller.

IBN implementation will reduce as well problems on running SDN applications inside SDN controllers. In the past, there have been always problems when multiple applications have to push commands to a SDN controller. Therefore, when it was needed to run multiple applications on the network, the SDN controller was unable to understand the real intent of each SDN application, but with the introduction of IBN this will change. IBN will bring coherence to SDN and will remove most of the conflicts that are generated from running multiple apps.

Nowadays, everyone, including SDN developers and Network vendor’s, knows that IBN is not an option, it’s a must, because without it SDN will not be flexible, scalable and portable enough to make SDN applications widely acceptable on the market. Therefore, IBN is crucial for the SDN success.
2.3.2 ONOS

The Open Network Operating System (ONOS) [29] project is an open source community hosted by The Linux Foundation. Its main goal is to create an operating system for telecommunications service providers. ONOS is based on the software-defined networking (SDN) paradigm and is designed with scalability, high performance and high availability in mind in order to make easy to create apps and services.

ONOS uses the layered architecture [30] shown in Figure 15. In the lowest layer of the stack there are protocol-aware network-facing modules called Providers that interact with the network using different protocols. In the middle a protocol-agnostic Core tracks and serves information about network state. Finally, in the top layer, there are Applications that consume services and act upon the information provided by the core.

The Core offers two interface APIs. The southbound interface enables the Core to interact with the network via the Providers. It also defines a protocol-neutral way to relay network state information to the Core. The northbound interface provides Applications with abstractions that describe network components and their properties.

ONOS is agnostic with regards of the configuration and management protocols used. There are modules that implement OpenFlow, Netflow, SNMP and OVSDB. IF support FOR a new protocol, it is possible to build a new network-facing module against the southbound API as a plugin that may be loaded into the system.

In ONOS terminology, a service is a unit of functionality that is comprised of multiple components that create a vertical slice through the aforementioned tiers as a software stack. ONOS provides several primary services which provide high-level abstractions, through which the applications can learn about the state of the network and through which they can control the flow of traffic through the network. For instance, the Topology service provides a network graph abstraction with the view of the entire network. This global network information is presented as logically centralized, even though it is physically distributed across multiple servers. The Path service computes/finds paths between infrastructure devices or between end-station hosts using the most recent topology graph snapshot. The Intent service provides a network-centric abstraction which allows application developers to control the network by specifying what they wish to accomplish (in the form of a policy) instead of how they want to accomplish it. Of course, this simplifies application development but at the same time the lets the platform to solve freely potentially conflicting requests.
All services follow ONOS' layered architecture. The internals of a service can be seen in Figure 16. In the core, each component has a manager that receives information from the network from providers using different protocols. Components’ managers serve this information to the applications which can received it synchronously by querying the service or asynchronously as an event listener. Stores are also part of the core layer and have the role of indexing, persisting, and synchronizing the information received by the manager with stores of other ONOS instances in order to ensure consistency and robustness of information.

New applications and core extensions can be developed in Java [32] ONOS provides a REST API and a GUI to load/unload new applications and core extensions. This can be done dynamically and ONOS application management subsystem distributes the application artifacts throughout the cluster to assure that all nodes are running the same application software.

ONOS applications platform is built on top of Apache Karaf [27] which is an OSGi container [31] and therefore benefits from OSGi bundle management services. Karaf is then able to activate/deactivate a collection of OSGi bundles as a single feature. An ONOS app is an artifact (an OAR file) composed by several OSGi bundles which provide features defined in features.xml files. The OAR file is used by the application management subsystem to deliver the application software across the entire ONOS cluster.

2.3.3 Applicability to SELFNET

ODL and ONOS offer capabilities and functionalities compatible with the encapsulation of SDN applications envisaged and required in SELFNET. In particular, both ODL and ONOS follow the same approach for SDN applications management services based on the combination on Apache Karaf and OSGi, thus offering flexible primitives and procedures to deploy, install and configure SDN applications in the controller runtime environments.

For these reasons, the mechanisms and procedures defined in this document for encapsulation of SDN applications have to be considered not dependent on a specific SDN controller (i.e. ODL or ONOS). At the time of writing, the selection of the reference SELFNET SDN controller is still open, and the consortium is actually evaluating both ODL and ONOS for their usage. SELFNET will continue exploring both SDN controllers in their capabilities offered at different levels and for different types of functionalities, including monitoring, network virtualization and multi tenancy support, performance, scalability, performance in the control plane, scalability, etc.
3 Encapsulation of NFV applications

In SELFNET, following the nomenclature defined in deliverable D2.2 [1], and reported for sake of completeness of this document in section 1.1, NFV applications refer to those SELFNET APPs implemented as virtual network functions to cover either control, data, or mixed control and data network functions in support of the three use cases envisaged in the project: self-healing, self-optimization, self-protection. These NFV applications are therefore sensor and actuator VNFs to be dynamically deployed and managed in the SELFNET Control Layer.

Figure 17: VNFs Lifecycle Management and Encapsulation on SELFNET Architecture

In this context, the encapsulation of NFV applications is composed by those infrastructure management functions responsible for the encapsulation of the SELFNET actuators and sensors VNFs into common virtualized containers. The goal is to provide a unified abstraction of the VNFs while exposing common lifecycle management primitives to be used for automated instantiation, configuration, re-configuration and termination of any SELFNET VNF. In practice, a set of mechanisms and tools to containerize the developed SELFNET actuators and sensors into encapsulated VNFs, thus enabling their management by means of common APIs. The reference baseline for the SELFNET encapsulation of NFV applications is the ETSI MANO architecture presented in section 2.1.1, and in particular how this standard tackles and addresses the lifecycle management of VNFs.
Figure 17 maps the encapsulation of NFV applications into the SELFNET architecture, highlighting the impacted sublayers where the VNF encapsulation mechanisms and tools will be implemented, namely: the VNF Management sublayer, the SON Control Plane sublayer, and the SON Data Plane sublayer.

The encapsulation of NFV applications is provided in SELFNET by leveraging on ETSI MANO functionalities and components for the lifecycle management of VNFs. Indeed, the ETSI MANO specifications [2] offer mechanisms, workflows and dedicated components responsible for the management of VNFs irrespectively of the specific service or application they implement.

3.1 High level architecture

The encapsulation of NFV applications aims to provide a unified and common approach for the lifecycle management of sensor and actuator VNFs, thus exposing primitives to instantiate and control the VNFs towards other SELFNET components and layers, like the Orchestration sublayer. As said, this is aligned with some of the functions provided within the ETSI MANO architecture. Therefore, the encapsulation of NFV application is implemented in SELFNET as an evolution of the ETSI MANO and ETSI NFV architecture, as depicted in Figure 18.

In the ETSI MANO architecture, the VNFM is responsible for the lifecycle management of all the VNF instances deployed within the virtualized infrastructure. The VNFM functions can be considered as generic and common functions applicable to any type of VNF, like instantiation and configuration, software upgrade, modification and termination. Therefore, the VNFM is natively the most suitable candidate to implement all those mechanisms needed in SELFNET to encapsulate sensor and actuator VNFs and expose towards upper SON layers unified VNF management primitives.

While the VNFM is responsible of exposing unified VNF lifecycle management APIs towards upper orchestration layer and components, thus hiding the specific peculiarities of VNF services and applications, the VNFs themselves need to be containerized and offer unified management primitives towards the VNFM. As depicted in Figure 18, SELFNET proposes to have a common container for all the VNFs implementing sensor and actuator functionalities that embeds those ETSI NFV Element Management System (EMS) functionalities needed to offer unified VNF management functions. Indeed, the ETSI NFV and MANO specifications present the EMS as a lifecycle management (LCM) agent responsible for FCAPS management functionality for a VNF, thus including configuration, performance collection, fault management operations. In practice, this LCM agent can be considered as a thin software layer embedded in each sensor or actuator VNF implementing the unified interface between the VNFM and the SON Control Layer specified in deliverable D2.2.
Figure 18 Evolution of ETSI NFV in support of SELFNET encapsulation of NFV applications

In summary, the SELFNET encapsulation of NFV applications translates into two major enhancements to the ETSI NFV and MANO architectures, as depicted in Figure 19:

- **Enhancement of the ETSI MANO VNFM functionalities towards a multi-tenant aware management of sensor and actuator VNFs lifecycle.** The SELFNET VNFM is the component responsible for the enforcement of VNF lifecycle management actions as requested by the upper layer Orchestration components. It directly interacts with the SELFNET actuator and sensor VNFs by means of unified procedures and interfaces.

- **Containerization of VNF and EMS into encapsulated VNFs.** The SELFNET actuator and sensor VNFs expose a common interface that allows the SELFNET VNFM to operate basic lifecycle management actions, like VNF configuration, re-configuration, start, stop irrespectively of the given VNF type. This is achieved by embedding a common EMS software layer within each sensor and actuator VNF.

It is important to highlight that the encapsulation of NFV applications implemented in SELFNET and released with this deliverable as a software prototype basically covers the VNFM functionalities within the ETSI MANO architecture. All the other MANO components, including the NFVO, the VIM and different catalogues depicted in Figure 19 are to be considered as fundamental components that will interact with the SELFNET VNFM and that will be provided separately within the project. The NFVO functionalities will be covered within the SELFNET Orchestration and Management Layer as part of the SON Autonomic Layer functions to coordinate the deployment, instantiation and provisioning of virtual network services composed by chains of VNFs. On the other hand, the VIM functionalities have been already implemented in SELFNET as part of the portable testbed development as described in deliverable D2.4 [33].

In particular, OpenStack is used in SELFNET as VIM to manage the NFVI infrastructure where all the sensor and actuator VNFs will be deployed and operated. Moreover, part of the catalogues, especially an extended version of the VNF catalogue, will be implemented in WP3 as a repository where all the SELFNET APPs (thus including VNFs, SDN applications and SDN controller applications) will be onboarded following the procedures described in deliverable D2.2.
3.1.1 SELFNET VNFM and encapsulated VNF functional decomposition

As anticipated in section 1.1, the VNF lifecycle management is addressed in SELFNET by taking into account and being open to support two different approaches: dynamic and static. In terms of NFV applications encapsulation and lifecycle management architecture, the SELFNET VNFM supports both of them. This means that in the static approach, the SELFNET VNFM takes care of instantiating, configuring, starting/stopping and terminating Virtual Machines already including VNF specific applications and contents (e.g. software packages). On the other hand, the dynamic case entails a more flexible VNF lifecycle management implementation where VNF specific applications (i.e. its software package) and contents are dynamically provisioned and configured (at the time of VNF instantiation) into generic Virtual Machines during instantiation. This will basically depends on how the release format of SELFNET sensors and actuator software will be specified in the VNF deployment template onboarded in the SELFNET APP repository (i.e. full complete Virtual Machine image vs. software package). The SELFNET VNFM is therefore agnostic of the lifecycle management approach.

It is important to highlight that the SELFNET VNFM depicted in Figure 20 is compliant with the interfaces specified and described in deliverable D2.2. However, with particular reference to the OMLvnfm_SAOr and VNF_OMLvnfm reference points (that is the interface implementing the combination DNLdp_OMLvnfm and CLcp_OMLvnfm D2.2 interfaces), the SELFNET VNFM provides its own implementation with adoption of specific technologies and semantics, as described in section 3.3.

Given the fact that the specification of the ETSI MANO interfaces is still in progress and includes only high level descriptions of messages and contents exchanged between the different components, it is common that the implementation of these interfaces differs from different VNFM implementations (especially the Or-Vnfm interface shown in Figure 19, i.e. the interface between the NFV Orchestrator and the VNFM, that is the OMLvnfm_SAOr in SELFNET according to the D2.2 nomenclature) with the adoption of different technologies and protocols, e.g. based on REST, NETCONF, etc. Thus the specific selection of technologies for the SELFNET VNFM is aligned with the current status of the standards.
In summary, the SELFNET VNFM implements and enhances the following VNF lifecycle management mechanisms and workflows specified by the ETSI MANO:

- **VNF instantiation**, including VNF configuration according to the VNF deployment template (VNFD), which describes attributes and requirements necessary to realize such VNF and instantiate it in the NFV infrastructure. The VNF configuration at this stage of the VNF lifecycle covers at least basic aspects, such as IP addresses, execution of lifecycle scripts.

- **VNF instance software update/upgrade**, that allows to perform modifications to the software running within the VNF (e.g. specific software packages, VNF application)

- **VNF instance modification**, that basically consists into an update of the VNF configuration

- **VNF instance scale out/in and up/down**, consisting into a structural modification of the VNF instance, in terms of virtual resources allocated to the given VNF
  - scale out: used to allocate additional Virtual Machines in support of the existing ones that implement the VNF application. It can be somehow considered as a special case of instantiation
  - scale in: used to terminate one or more Virtual Machines implementing the VNF application. It can be somehow considered as a special case of termination
  - scale up: used to increase the amount of virtual resources (CPU, storage) for a given VNF, due to performance reasons, possibly with a live procedure (at least for the CPU)
  - scale down: used to decrease the amount of virtual resources (CPU, storage) for a given VNF, due to performance reasons, possibly with a live procedure (at least for the CPU)

- **VNF instance-related collection of performance measurements from the VNF**

- **VNF instance termination**

- **Management of integrity and consistency of the VNF instance through its lifecycle**

![Diagram](https://via.placeholder.com/150)

**Figure 20 Selfnet VNFM and encapsulated VNF functional split**

Figure 20 illustrates the high level SELFNET VNFM functional split and highlights its main functional entities and the interactions with external components in the SELFNET NFV OML and SON autonomic layer. This functional split also includes those lifecycle management entities needed at the VNF side in support of the SELFNET VNFM.
operations. Here, following the ETSI NFV principles, each SELFNET VNF is generally considered as composed by a set of VNF components (VNF-C). Indeed, VNF developers and vendors may structure their VNFs into software components each one deployed within an independent Virtual Machine. A VNF-C is therefore associated with a VNF software component, and a 1:1 correspondence exists between VNF-Cs and Virtual Machines. The way VNFs are structured into VNFCs is out of the scope of this document and generally depends on specific VNF vendor and developer factors. For each SELFNET VNF-C, a lifecycle management (LCM) agent is embedded in the correspondent Virtual Machine to enable the communication with the SELFNET VNFM and implement the lifecycle actions on the VNF. This agent enables the containerization of VNF and ETSI MANO EMS components into encapsulated VNFs, providing a common interface towards the SELFNET VNFM.

On the VNFM side, the following Table 2 lists the SELFNET VNFM functional modules and for each of them it provides a high-level description of its main functionalities, roles and responsibilities.

<table>
<thead>
<tr>
<th>Functional component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMLvnfm_SAUF driver</td>
<td>This component provides the implementation of the OMLvnfm_SAUF interface as defined in deliverable D2.2. It is the main SELFNET VNFM entry point to access and use its lifecycle management logic. The OMLvnfm_SAUF driver implements the server side of the interface, being used by the SELFNET Orchestration sublayer to request for VNF lifecycle management actions. In particular, the OMLvnfm_SAUF interface extends the ETSI MANO Or-Vnfm reference point in support of multi-tenancy, exposing primitives for:</td>
</tr>
<tr>
<td></td>
<td>• VNF instantiation</td>
</tr>
<tr>
<td></td>
<td>• VNF scale</td>
</tr>
<tr>
<td></td>
<td>• VNF operation (start/stop)</td>
</tr>
<tr>
<td></td>
<td>• VNF modification (i.e. re-configuration)</td>
</tr>
<tr>
<td></td>
<td>• VNF software update</td>
</tr>
<tr>
<td></td>
<td>• VNF termination</td>
</tr>
<tr>
<td></td>
<td>Each request coming from the SELFNET Orchestration sublayer is processed and validated by this components, and then passed to the VNF Lifecycle Engine to implement the proper actions according to the given state of the VNF.</td>
</tr>
<tr>
<td>SAUvo_OMLvnfm driver</td>
<td>This component provides the implementation of the client side of the SAUvo_OMLvnfm interface as it is specified in deliverable D2.2. This interface is used by the SELFNET VNFM to retrieve information about VNF template descriptors (VNFD) and metadata to properly instantiate VNFs, in terms of virtual resources to be allocated as well as attributes and requirements for its operation. This information is needed during the VNF instantiation phase. It is important to highlight that the support of this interface is not mandatory and the retrieval of VNF template descriptor (VNFD) information may occur by means of the OMLvnfm_SAUF interface (as described in section 3.3.1.3).</td>
</tr>
<tr>
<td>VNF Lifecycle Engine</td>
<td>This component provides the core functions of the SELFNET VNFM and implements the per-VNF lifecycle management logic and intelligence. It takes care to maintain the state and integrity of each</td>
</tr>
</tbody>
</table>
VNF, and for each request coming from the driver it applies and perform the proper action.

To do this, each VNF is maintained within this lifecycle engine with its own finite state machine to properly manage the evolution of its lifecycle. The SELFNET VNFM manages VNF states and lifecycle transitions according to the ETSI NFV VNF software architecture (SWA) specification [34], as depicted in Figure 21. In particular, the VNF finite state machine implemented by the SELFNET VNFM includes the following states:

- Null: the VNF does not exist and is about to be instantiated
- Instantiated Not Configured: the VNF has been instantiated (i.e. virtual resources allocated) but is not configured for service
- Instantiated Configured – Inactive: the VNF is allocated and configured for service, but it does not yet participates to the service.
- Instantiated Configured – Active: the VNF has application has been started and it participates to the related service.
- Terminated: the VNF has been terminated and the related virtual resources released

Software update and scale actions translates into finite state machines transitions which keep the VNF state either in the Instantiated Not Configured or Instantiated Configured state.

<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMLvim_OMLvnfm driver</td>
<td>This component provides the implementation of the client side of the OMLvim_OMLvnfm interface as it is specified in deliverable D2.2. This is used to request to the VIM the allocation of virtual resources (network, storage, CPU) for each Virtual Machine composing VNFs and VNF-Cs according to the information contained in the VNF descriptor template (VNFD). This is valid when for VNF instantiation and scale the VNFM performs the resource allocation on the VIM. As detailed below in section 3.2, the SELFNET VNFM mostly supports the other ETSI MANO approach for VNF resource allocation, that entail it is performed directly by the NFV Orchestrator (i.e. the SELFNET Orchestration sublayer in our case through the OMLvim_SAUor interface). Therefore, the support of this interface is not mandatory.</td>
</tr>
<tr>
<td>VNF_OMLvnfm driver</td>
<td>This component implements the combination of two interfaces defined in deliverable D2.2: the DNLdp_OMLvnfm and CLcp_OMLvnfm, which are respectively responsible for the configuration of data plane and control plane VNF types. With reference to the ETSI MANO, the VNF_OMLvnfm interface combines the Ve-Vnfm-Nf and Ve-Vnfm-Em reference points. This component basically provides the client side of the VNF_OMLvnfm interface for VNF configuration, start, stop, software update and performance query purposes. It is directly invoked by the VNF Lifecycle Engine and in particular in the context of each VNF finite state machine according to the given lifecycle event and transition. Its direct counterpart implementing the server side of the VNF_OMLvnfm interface is the LCM agent deployed within each VNF-C.</td>
</tr>
</tbody>
</table>

Table 2 SELFNET VNFM functional components
On the VNF and VNF-C side, the following Table 3 describes functionalities and roles of the LCM entities deployed within each VNF-C.

<table>
<thead>
<tr>
<th>Functional component</th>
<th>Description</th>
</tr>
</thead>
</table>
| LCM Agent            | This component is a software agent deployed within each SELFNET VNF and VNF-C for their encapsulation and exposes common primitives for their lifecycle management. Each SELFNET actuator and sensor VNF and VNF-C will embed this thin software layer. The LCM Agent is directly interfaced with the SELFNET VNFM VNF_OMLvnfm driver and provides the following primitives:  
  - VNF configuration  
  - VNF start  
  - VNF stop  
  - VNF software update  
  - VNF performance query (e.g. in terms of utilization of VNF resources like CPU)  

The deployment of the LCM agent within the each SELFNET VNF-C (i.e. each Virtual Machine) can be performed following two approaches:  
- LCM agent embedded in each VNF/VNF-C onboarded in the SELFNET APP repository, thus already installed as a software package  
- LCM agent dynamically deployed during VNF/VNF-C instantiation, e.g. implementing a cloud-init [35] and metadata procedures in general when allocating Virtual Machines in the VIM |

Table 3 SELFNET VNF and VNF-C LCM entities
3.2 SELFNET VNFM lifecycle management services

This section provides a description of the lifecycle management services supported by the SELFNET VNFM. For each service, a sequence diagram is provided along with a step-wise description of the main interactions between the involved actors: the SELFNET VNFM, the SELFNET Orchestration sublayer, the VIM, the VNFs and VNF-Cs.

For the SELFNET VNFM it is not mandatory to take care of the allocation of virtual resources (network, storage, CPU) for each Virtual Machine on the VIM. Indeed, according to the ETSI MANO specifications, two different approaches are defined: resource allocation performed by the NFV Orchestrator, and resource allocation performed by the VNFM. SELFNET mostly supports the first one, therefore the allocation of virtual resources is delegated to the Orchestration sublayer

As already anticipated in section 1.1, it is also important to highlight that VNF onboarding phase is not covered in this document and is not part of the responsibilities of the SELFNET VNFM. It will be a responsibility of the VNF onboarding sublayer to manage the uploading, retrieval, removal and update of VNFs into the SELFNET APP catalogue.

3.2.1 VNF instantiation

- The SELFNET Orchestration sublayer requests to the SELFNET VNFM for the instantiation of a new VNF. The message includes all the attributes, parameters and requirements needed for the instantiation, including the VNFD and additional information for the specific VNF to be instantiated (e.g. deployment flavour, deployment location, etc.)

- The SELFNET VNFM validates the instantiation request (e.g. proper format, missing parameters, etc.), assigns a unique VNF instance identifier and requests back to the SELFNET Orchestration sublayer for the allocation of virtual resources as described in the VNFD. This is compliant with the ETSI MANO approach where the VNF virtual resource allocation are performed by the NFV Orchestrator directly to the VIM.

- At this point, the SELFNET Orchestration sublayer requests to the VIM for the allocation of the needed virtual resources to build the given VNF, in terms of virtual networks, virtual CPU, virtual storage, and all the Virtual Machines needed to implement the VNF-Cs composing the VNF.

- The VIM performs the allocation of the resources, by taking care to first create the needed network resources to enable connectivity for the VNF, and then allocates the needed compute and storage resources for each Virtual Machine.

Figure 22 SELFNET VNFM VNF instantiation workflow

© SELFNET consortium 2016
When the SELFNET Orchestration sublayer receives the positive feedback from the VIM about the virtual resources allocation, it provides to the SELFNET VNFM the allocate request acknowledgment along with all the identifiers and properties of the resources allocated by the VIM (e.g. hostnames, addresses, resources ids, location, etc.)

The SELFNET VNFM can now take care of the VNF configuration part, which means sending individual configuration messages to each VNF-C composing the VNF, according to the information stored in the VNFD. Indeed, the VNFD includes, for each VNF-C, a set of configuration parameters and a set of lifecycle scripts to be run in the VNF-C itself (see section 3.3.1.1.1). These scripts are referenced in the VNFD and can be either pre-deployed in the VNF-Cs, or copied to the VNF-C within the configuration message, or be cloned from repositories during the instantiation. This configuration part of the instantiation workflow is where the differentiation between the static and dynamic lifecycle management approach occurs:

- **Static approach**: the VNF-Cs/Virtual Machines allocated by the VIM at the previous step already embed the VNF-C application and content (i.e. the software package), and the lifecycle scripts (with parameters) to be applied according to the VNFD provide just its configuration and possibly the start of VNF-C services

- **Dynamic approach**: the VNF-Cs/Virtual Machines allocated by the VIM at the previous step are generic, and the lifecycle scripts (with parameters) to be applied according to the VNFD provide the deployment, installation and configuration of the VNF-C application and content (i.e. the software package) and possibly the start of the VNF-C service

Once that all the VNF-Cs have been properly configured, the SELFNET VNFM sends back to the SELFNET Orchestration sublayer the positive feedback that the VNF instantiation has been completed successfully.

### 3.2.2 VNF scale

#### Scale out/up

![Figure 23 SELFNET VNFM VNF scale out/up workflow](image)

- The SELFNET Orchestration sublayer requests to the SELFNET VNFM to scale out or up a VNF previously instantiated. The message includes all the information related to the new virtual resources to be allocated (e.g. increase of virtual CPU in case of scale up, or a full new Virtual Machine in case of scale out)
The SELFNET VNFM validates the scale request (e.g. proper format, missing parameters, etc.) and also checks that the scale request is compliant with the scaling properties described in the VNFD. Then it requests back to the SELFNET Orchestration sublayer for the allocation of the additional virtual resources for the VNF.

The SELFNET Orchestration sublayer requests to the VIM for the allocation of the needed additional virtual resources to scale the VNF, according to the attributes to be improved (full new VM, storage, compute).

The VIM performs the allocation of the requested new resources and bind them to the proper existing Virtual Machines building the instantiated VNF.

When the SELFNET Orchestration sublayer receives the feedback from the VIM, it provides to the SELFNET VNFM the allocate request acknowledgment along with all the identifiers and properties of the new resources.

In case of scale out (i.e. new Virtual Machines created for the VNF) the SELFNET VNFM take cares of the configuration of the new resources, thus sending individual configuration messages to each new VNF-C (similarly to the instantiation workflow). In case of scale up, the SELFNET VNFM may send configuration messages to VNF-Cs either affected or not affected by the scaling according to the rules and properties set in the VNFD. As per instantiation, the lifecycle scripts and parameters are managed according to the lifecycle management static or dynamic approach.

Once that all the new VNF-Cs have been properly configured, the SELFNET VNFM sends back to the SELFNET Orchestration sublayer the positive feedback that the VNF scale has been completed successfully.

**Scale in/down**

![Diagram of SELFNET VNFM VNF scale in/down workflow](image)

The SELFNET Orchestration sublayer requests to the SELFNET VNFM to scale in or down a VNF previously instantiated. The message includes all the information related to the virtual resources to be deleted (e.g. decrease of virtual CPU in case of scale down, or a full Virtual Machine in case of scale in).

The SELFNET VNFM validates the scale request (e.g. proper format, missing parameters, etc.) and also checks that the scale request is compliant with the scaling properties described in the VNFD.
• In case of scale in (i.e. Virtual Machines to be deleted for the VNF) the SELFNET VNFM can optionally take care of a de-configuration of the Virtual Machines to be removed, thus sending individual configuration messages to each new VNF-C. This may help to implement a graceful shutdown of Virtual Machines that are going to be deleted.

• Once that all the VNF-Cs have been properly de-configured, the SELFNET VNFM requests back to the SELFNET Orchestration sublayer for the deletion of a set of virtual resources for the VNF.

• The SELFNET Orchestration sublayer requests to the VIM for the deletion of the virtual resources according to the attributes to be downgraded (full Virtual Machine, storage, compute)

• The VIM performs the deletion of the requested resources and unbind them from the existing Virtual Machines building the instantiated VNF.

• When the SELFNET Orchestration sublayer receives the feedback from the VIM, it forwards to the SELFNET VNFM a message that acknowledges the completion

• The SELFNET VNFM internally processes the operation to store the new state of the VNF and then sends back to the SELFNET Orchestration sublayer the positive feedback that the VNF scale has been completed successfully

3.2.3 VNF operation (start/stop)

The SELFNET Orchestration sublayer requests to the SELFNET VNFM to start or stop a VNF previously instantiated. This operation is intended to be applied at the VNF application level. The message can include information related to the individual VNF-Cs to be started or stopped.

• The SELFNET VNFM validates the request (e.g. proper format, missing parameters, etc.) and also checks that it is compliant with the properties described in the VNFD.

• The SELFNET VNFM take cares of sending individual start/stop messages to each VNF-C, including information of scripts and parameters to be used according to the information originally included in the VNFD.

• Once that all the VNF-Cs have been properly started or stopped, the SELFNET VNFM sends back to the SELFNET Orchestration sublayer the positive feedback that the VNF operation has been completed successfully
3.2.4 VNF modification

The SELFNET Orchestration sublayer requests to the SELFNET VNFM to modify a VNF previously instantiated. This operation is intended to be provide a re-configuration of the VNF at its application level. The message includes information related to the new configuration parameters to be enforced in the VNF.

The SELFNET VNFM validates the request (e.g. proper format, missing parameters, etc.) and also checks that it is compliant with the properties described in the VNFD.

The SELFNET VNFM thus sends individual re-configuration messages to each involved VNF-C, including information of scripts and new configuration parameters to be applied according to the information originally included in the VNFD and in the VNF modification request.

Once that all the VNF-Cs have been properly re-configured, the SELFNET VNFM sends back to the SELFNET Orchestration sublayer the positive feedback that the VNF re-configuration has been completed successfully.

3.2.5 VNF software update

The SELFNET Orchestration sublayer requests to the SELFNET VNFM to modify a VNF previously instantiated. This operation can be seen as a special case of the VNF modification described above, where the modification is intended to be at the level of one or more software packages running with the VNF/VNF-Cs. The message may include information related to the new software configuration to be enforced in the VNF, possibly in the form of parameters to be enforced in the VNF.
• The SELFNET VNFM validates the request (e.g. proper format, missing parameters, etc.) and also checks that it is compliant with the properties described in the VNFD.

• The SELFNET VNFM thus sends individual software update messages to each involved VNF-C, including information of lifecycle scripts and new parameters (when available) to be applied according to the information originally included in the VNFD and in the VNF software update request.

• Once that all the VNF-Cs have been properly updated, the SELFNET VNFM sends back to the SELFNET Orchestration sublayer the positive feedback that the VNF update has been completed successfully.

### 3.2.6 VNF termination

![Figure 28 SELFNET VNFM VNF termination workflow](image)

- The SELFNET Orchestration sublayer requests to the SELFNET VNFM to terminate a VNF previously instantiated. The message includes a reference to the VNF instance to be terminated.

- The SELFNET VNFM validates the request (e.g. proper format, missing parameters, etc.) and also checks that the request is compliant with the current status of the VNF.

- The SELFNET VNFM sends individual termination messages to each VNF-C building the VNF, including information of scripts and parameters (when available) to be applied according to the information originally included in the VNFD. This phase is intended to implement a graceful deletion of the VNF and its VNF-Cs; indeed, it allows to perform the termination at the VNF application level first and avoid to leave the application in unstable conditions.

- Once that all the VNF-C applications have been properly terminated, the SELFNET VNFM sends back to the SELFNET Orchestration sublayer the positive feedback that the VNF has been successfully terminated.

- The SELFNET Orchestration sublayer requests to the VIM for the deletion of all the virtual resources composing the VNF (full Virtual Machines, storage, compute, network).

- The VIM performs the deletion of the requested resources. For resources optimization purposes, the deletion of Virtual Machine images may be considered optional in order to allow a quick reaction of the system for a later new instantiation.

- When the SELFNET Orchestration sublayer receives the feedback from the VIM, the VNF termination has been completed successfully.
3.3 SELFNET VNFM prototype

Starting from the high level architecture described in section 3.1, the SELFNET consortium has developed a SELFNET VNFM prototype, covering both the static and dynamic lifecycle management approaches.

The SELFNET VNFM prototype has been implemented on top of the OpenBaton opensource project introduced in section 2.2.1.2. The reason behind the selection of the OpenBaton platform as starting point for the implementation of this SELFNET VNFM is that OpenBaton is highly compliant with the ETSI MANO specifications and workflows for VNF lifecycle management, and it can be easily integrated with existing cloud platforms and adapted to different types of VNFs. In particular, at the VIM level, OpenBaton supports the integration with (multi-site) OpenStack environments and it also provides an SDK to implement VIM-specific drivers. On the VNF side, it implements a generic VNFM, but it can also interoperate with external VNF-specific VNFM via REST APIs. Details about this OpenBaton based SELFNET VNFM software prototype, including VNF information models, interfaces and APIs specification and release format are provided in section 3.3.1. In addition, as a further NFV applications encapsulation prototype activity, Ubuntu Juju (see section 2.2.2.1) has been also validated as VNFM as described in section 3.3.2.

3.3.1 OpenBaton SELFNET VNFM

The SELFNET VNFM developed on top of the OpenBaton implements the static VNF lifecycle management approach described in section 1.1. This means that VNFs are instantiated, configured, upgraded, updated, started, stopped, scaled and terminated according to detailed virtual resource and Virtual Machine information, attributes and requirements included in VNF descriptor templates (VNFD).

As anticipated in section 2.2.1.2, OpenBaton is a Java open source project released under the Apache 2.0 License which implements an NFV Orchestrator with an integrated and general purpose VNFM. A complete Java library providing ETSI MANO compliant data structures, messages, and utilities for parsing and formatting JSON contents is also available. In addition, a set of SDK libraries are also offered to implement vendor specific VNFM. In particular, two different SDKs are available: a `vnfm-sdk-amqp` provides automatic background implementation of communication between VNFM and NFV Orchestrator using RabbitMQ message queues [36], while a `vnfm-sdk-rest` uses REST.

The Generic VNFM [37] offers a basic support for the management of VNFs lifecycle and interacts with the NFV Orchestrator through a RabbitMQ message queue interface (using the `vnfm-sdk-amqp`) for instantiation, modification, starting and stopping of VNFs. The interaction of the generic VNFM with VNFs is mediated through the generic OpenBaton EMS, which is a software application running within each VNF/VNF-C that enables a communication based on RabbitMQ message queues. This interaction is mostly used for VNF configuration purposes and to trigger the execution of lifecycle management scripts on the VNF side.

The SELFNET VNFM has been developed starting from the OpenBaton Generic VNFM [37], which has been enhanced with the following functionalities:

- **Usage of `vnfm-sdk-rest` instead of `vnfm-sdk-amqp`**: this enables the SELFNET VNFM to expose REST APIs at the OMLvnfm_SAUor reference point, thus easing the integration with the SELFNET Orchestration sublayer.
- **Enhancement of the VNF instantiation procedure and interface to enable tenant awareness in the SELFNET VNFM**: In particular the OMLvnfm_SAUor REST API for VNF instantiation has been extended to include information about the tenant to which the VNF belongs. This information is then managed and kept by the SELFNET VNFM to isolate VNFs belonging to different tenants.
• Implementation of VNF stop operation lifecycle management service defined in 3.2.3: the generic VNFM natively supports the only start operation
• Implementation of VNF modification lifecycle management service as defined in 3.2.4, that was not originally supported by the generic VNFM and has been implemented from scratch
• Implementation of VNF software update lifecycle management service defined in 3.2.5, that not originally supported by the generic VNFM and has been implemented from scratch

Moreover, the Openbaton EMS has been enhanced in support of SELFNET LCM agent functionalities described in Table 2. In particular the retrieval of VNF performance statistics has been implemented.

This SELFNET VNFM follows the ETSI MANO lifecycle management approach where resource allocation during VNF instantiation and scale is performed by the NFV Orchestrator (i.e. the SELFNET Orchestration sublayer in our case). Therefore the OpenBaton based SELFNET VNFM does not interface directly with the SELFNET OpenStack VIM. However, OpenBaton already provides a full complete OpenStack VIM plugin that could be imported and easily integrated in the SELFNET VNFM if needed.

### 3.3.1.1 VNF information model

#### 3.3.1.1.1 VNF Descriptor

According to the ETSI MANO specifications, the instantiation and operational behaviour of each VNF is described in a template called VNF Descriptor (VNFD). The VNFD is stored in the SELFNET APP repository, where it is uploaded during the VNF onboarding. The OpenBaton SELFNET VNFM uses VNFDs to create VNF instances and to manage their lifecycle according to the attributes specified. In particular, during instantiation, virtual resources (VMs, storage, network, CPU) are assigned to VNFs and VNF-Cs based on requirements and parameters included in the VNFD, which also contains resource allocation criteria, description of VNF composition (i.e. number and types of VNF-Cs), VNF functional scripts for specific lifecycle events.

Different versions of a VNFD corresponds to different implementations of the same function, different versions to run in different execution environments (e.g. on different hypervisors, dependent on NFVI resources availability information, etc.), or different release versions of the same software. In SELFNET, VNFDs are stored and maintained by the VNF Onboarding sublayer.

The OpenBaton based SELFNET VNFM VNFDs are in JSON format and are based on a detailed information model which includes most of the attributes defined in the ETSI MANO specification.

The following tables provides a description of the most relevant information included in the SELFNET VNFM VNFD, which has been inherited from the OpenBaton information model. For reference, a simple example of VNFD is reported in Annex A.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Cardinality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>1</td>
<td>The name of the VNFD</td>
</tr>
<tr>
<td>version</td>
<td>1</td>
<td>The version of the VNFD</td>
</tr>
<tr>
<td>type</td>
<td>1</td>
<td>The type of VNF application (e.g. “virtual-ids”)</td>
</tr>
<tr>
<td>vendor</td>
<td>1</td>
<td>The owner/developer of the VNF</td>
</tr>
<tr>
<td>endpoint</td>
<td>1</td>
<td>The type of VNFM to be used to deploy VNFs based on this VNFD (e.g. “selfnet”)</td>
</tr>
</tbody>
</table>
Provides the list of scripts to be applied into the VNF for specific lifecycle events (instantiation, scale, modification, termination, etc.). These are scripts at the VNF level and thus valid for all VNF-Cs. See Table 5.

List of configuration parameters defined by (key, value) that can be used as environment variables in the lifecycle_event scripts. See Table 6.

It is the Virtual Deployment Unit (VDU). It includes the deployment attributes and requirements, in terms of virtual resources, for all the VNF-Cs building the VNF. See Table 7.

List of possible deployments choices for this VNFD. Each deployment flavour includes information on number and size of virtual resources to be allocated. See Table 8.

List of VNF internal and external virtual links (i.e. virtual networks to be allocated in the VIM). Internal links refer to interconnections among VNF-Cs.

<table>
<thead>
<tr>
<th>Table 4 SELFNET VNFM VNFD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Identifier</strong></td>
</tr>
<tr>
<td>event</td>
</tr>
<tr>
<td>lifecycle_events</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5 SELFNET VNFM lifecycle_event</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Identifier</strong></td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>configurationParameters</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 6 SELFNET VNFM configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Identifier</strong></td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>vm_image</td>
</tr>
<tr>
<td>lifecycle_event</td>
</tr>
<tr>
<td>virtual_memory_</td>
</tr>
<tr>
<td>Identifier</td>
</tr>
<tr>
<td>--------------------</td>
</tr>
<tr>
<td>id</td>
</tr>
<tr>
<td>constituent_vdu</td>
</tr>
<tr>
<td>flavour_key</td>
</tr>
</tbody>
</table>

Table 7 SELFNET VNFM VDU

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Cardinality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>1</td>
<td>VNF-C identifier. It has to be unique within the namespace of the VNF</td>
</tr>
<tr>
<td>connection_point</td>
<td>1…N</td>
<td>It provides the list of references to VNF internal virtual links (as provided in the VNFD virtual_link data structure) to which the VNF-C is attached to.</td>
</tr>
</tbody>
</table>

Table 9 SELFNET VNFM vnfc

3.3.1.1.2 VNF Record

When a VNF is instantiated, the SELFNET VNFM creates a VNF Record (VNFR) to index the virtual resources allocated to the given VNF instance and its VNF-Cs. Following the ETSI MANO specifications, the VNFR is created to model the VNF instance, and it is built based on the content of the related VNFD augmented with additional runtime VNF-C information depending on the given instantiation (e.g. virtual resource identifiers, lifecycle history, etc.). It also includes information to operate changes to the deployed VNF instance, e.g. in the case of a scalability update. Moreover, information related to the VNF-C instances associated to VDUs included in the VNFD represent the core part of the VNFR.

The VNFR is created by the SELFNET VNFM during the VNF instantiation but it is owned and maintained by the SELFNET Orchestration sublayer. It enables the traceability and integrity of VNF instances across the different components forming the SELFNET OML.

After the VNF instantiation, in each interaction between the SELFNET Orchestration sublayer and the SELFNET VNFM (e.g. for VNF scale, start, stop, scale, modification, termination), the VNFR is exchanged within the lifecycle management messages, as detailed in the OMLvnfm_SAUor interface description in 3.3.1.1.2.
Similarly to the VNFD, the OpenBaton based SELFNET VNFM VNFRs are in JSON format and are based on a detailed information model which includes most of the attributes defined in the ETSI MANO specification.

The following tables provides a description of the most relevant information included in the SELFNET VNFM VNFR, which has been inherited from the OpenBaton information model.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Cardinality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>1</td>
<td>The ID of the VNF instance</td>
</tr>
<tr>
<td>vendor</td>
<td>1</td>
<td>The owner/developer of the VNF in the reference VNFD</td>
</tr>
<tr>
<td>version</td>
<td>1</td>
<td>The version of the VNF in the reference VNFD</td>
</tr>
<tr>
<td>deployment_flavour_key</td>
<td>1</td>
<td>The selected deployment flavour among those included in the reference VNFD</td>
</tr>
<tr>
<td>lifecycle_event</td>
<td>0…N</td>
<td>The copy of the <code>lifecycle_event</code> in the reference VNFD</td>
</tr>
<tr>
<td>configurations</td>
<td>0…N</td>
<td>The copy of the <code>configurations</code> in the reference VNFD</td>
</tr>
<tr>
<td>parent_ns_id</td>
<td>1…N</td>
<td>The reference to the service instances identifier using this VNF instance</td>
</tr>
<tr>
<td>descriptor_reference</td>
<td>1</td>
<td>The reference to the reference VNFD name</td>
</tr>
<tr>
<td>vnfm_id</td>
<td>1</td>
<td>The identification of the VNFM entity managing this VNF</td>
</tr>
<tr>
<td>status</td>
<td>1</td>
<td>The current operational status of the VNF instance (failed, normal operation, degraded operation, offline through management action)</td>
</tr>
<tr>
<td>vdu</td>
<td>1…N</td>
<td>It includes the VNF attributes in terms of allocated virtual resources for all the VNF-Cs instances in this VNF. See Table 11</td>
</tr>
<tr>
<td>lifecycle_event_history</td>
<td></td>
<td>The record of lifecycle events occurred to this VNF (instantiate, terminate, scale, etc.)</td>
</tr>
<tr>
<td>vnf_address</td>
<td>1…N</td>
<td>The VNF address(es) configured for the management access to the VNF instance (and possibly internal VNF-Cs)</td>
</tr>
<tr>
<td>virtual_link</td>
<td>1…N</td>
<td>List of VNF internal and external virtual links allocated for this VNF instance in the VIM (i.e. virtual networks)</td>
</tr>
</tbody>
</table>

Table 10 SELFNET VNFM VNFR

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Cardinality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>1</td>
<td>The name of the reference VDU in the VNFD</td>
</tr>
<tr>
<td>vm_image</td>
<td>1</td>
<td>The reference to the Virtual Machine image name in the VNFD</td>
</tr>
<tr>
<td>lifecycle_event</td>
<td>0…N</td>
<td>The copy of the <code>lifecycle_event</code> in the VNFD VDU. See Table 5</td>
</tr>
</tbody>
</table>
The copy of memory requirements for the allocation VNF-Cs from the reference VNFD VDU

The copy of computation requirements for the allocation VNF-Cs from the reference VNFD VDU

List of VNF-Cs instantiated from this VDU. See Table 12

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Cardinality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vc_id</td>
<td>1</td>
<td>The identifier of the virtual container (i.e. the Virtual Machine) in the VIM associated to this VNF-C instance</td>
</tr>
<tr>
<td>vim_id</td>
<td>1</td>
<td>The VIM identifier where this VNF-C instance has been allocated</td>
</tr>
<tr>
<td>hostname</td>
<td>1</td>
<td>The hostname associated to the Virtual Machine implementing this VNF-C instance</td>
</tr>
<tr>
<td>state</td>
<td>1</td>
<td>The current operational status of the VNF-C instance</td>
</tr>
<tr>
<td>vnfComponent</td>
<td>1</td>
<td>Additional attributes and parameters identifying the VNF-C instance. See Table 9</td>
</tr>
</tbody>
</table>

Table 11 SELFNET VNFM VNFR vdu

3.3.1.2 SELFNET LCM Agent

The SELFNET LCM Agent prototype has been developed starting from the OpenBaton EMS [37]. It is basically a thin python software running in each VNF-C that enables the execution of the lifecycle scripts described in 3.2 and 3.3.1.1.1 for VNF-C configuration and operation purposes, including software updates.

The interface with the SELFNET VNFM is implemented by a RabbitMQ message queue service [36]. RabbitMQ is a messaging broker that acts as an intermediary for messaging services across software applications. RabbitMQ gives applications a common platform to send and receive messages, based on a message broker concept. The communication with the SELFNET VNFM is enabled by a subscription procedure; indeed, soon after the LCM agent starts, it connects to the RabbitMQ server running in the SELFNET VNFM and subscribes as a consumer to the dedicated queue with its name (based on the hostname of the Virtual Machine) and sends a registration message to the specific registration queue hosted by the SELFNET VMFM.

Then, the LCM agent waits for messages coming from the SELFNET VNFM in its dedicated queue to execute the correspondent lifecycle actions on arrival. The messages exchanged over these queues are in json format, as detailed in section 3.3.1.3.2, and trigger the execution of actions to: clone scripts from a repository, update scripts from repository, and execute scripts with the arguments and environmental variables sent as parameters in the message.

Details for RabbitMQ connection, including reachability of RabbitMQ server running within the SELFNET VNFM, are provided with a dedicated configuration file.

As said, the SELFNET LCM agent enhances the OpenBaton EMS, in particular to enable of VNF statistics and performance query from the SELFNET VNFM, according to the main functionalities offered by the SELFNET VNFM and described in section 3.1. A new message “GET_STATISTICS” has been implemented, as detailed in section 3.3.1.3.2, to
let the SELFNET VNFM retrieve the utilization (in percentage) CPU, memory and disk space within each VNF-C.

### 3.3.1.3 Interfaces specification

With reference to the high level architecture described in section 3.1 and depicted in Figure 20, the OpenBaton SELFNET VNFM provides the implementation of the following interfaces:

- OMLvnfm_SAUor, both server and client side in support of the lifecycle management services described in section 3.2
- VNF_OMLvnfm, for VNF and VNF-C configuration purposes
- SAUvo_OMLvnfm, to retrieve and query VNFDs

The OMLvim_OMLvnfm is not directly provided by the SELFNET VNFM, in line with the ETSI MANO principle of virtual resource allocation performed by the NFV Orchestrator (i.e. SELFNET Orchestration sublayer). However, OpenBaton offers in its libraries a ready-to-be-integrated OpenStack VIM plugin that if needed can be easily imported and used at a later stage of the project to allow the SELFNET VNFM to provide an implementation of the OMLvim_OMLvnfm interface described in deliverable D2.2.

#### 3.3.1.3.1 OMLvnfm_SAUor REST API

The OMLvnfm_SAUor is the interface between the SELFNET Orchestration sublayer and the SELFNET VNFM, and regulates their communication for instantiation, termination, scale, modification of VNFs. The SELFNET VNFM, according to the deliverable D2.2 specifications and the lifecycle management services introduced in section 3.2, provides a REST based implementation of both the client and server side of this interface.

The server side of the interface is implemented by means of a REST controller running within the SELFNET VNFM, and provides the following primitives:

- VNF instantiation
- VNF operation (start/stop)
- VNF configuration (i.e. modification)
- VNF scale
- VNF software update
- VNF termination
- VNF query

The messages over the REST interface are in JSON format. Details are provided in Table 13.

On the other hand, the client side of the OMLvnfm_SAUor interface is implemented by means of a REST client embedded in the SELFNET VNFM and provides the following functionalities:

- VNF registration, to let the SELFNET VNFM subscribe to the NFV orchestrator (i.e. the SELFNET Orchestration sublayer)
- VNF allocate request (for allocation of virtual resources to the VIM)

The messages over this REST interface are in JSON format. Details are provided in Table 14.

The correspondent server part has to be provided by the SELFNET Orchestration sublayer, as detailed in section 3.2. It is important to highlight that this part of the REST interface is inherited from OpenBaton, in particular from its NFV Orchestrator.
<table>
<thead>
<tr>
<th>REST Endpoint</th>
<th>LCM action</th>
<th>Method</th>
<th>Request message (JSON content)</th>
<th>Response message (JSON content)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/vnfm-rest-core/</td>
<td>VNF instantiation</td>
<td>POST</td>
<td>{</td>
<td>{</td>
<td>The full VNFD (see Table 4) is passed within the JSON request message, together with the name of the deployment flavour to apply (vnfd). The extension field includes the tenant id, the reference to the service this VNF belongs to as well as the geographical location where to deploy the VNF (e.g. edge or core). The list of external virtual links is also passed (vlrs). The request includes either a link to a git repository for all the lifecycle scripts, or the full scripts encoded in base64 payload. The response message includes the VNFR (see Table 10) created by the SELFNET VNFM and the reference to the action “INSTANTIATE” successfully performed</td>
</tr>
<tr>
<td></td>
<td>VNF operation (start)</td>
<td>POST</td>
<td>{</td>
<td>{</td>
<td>The full VNFR (see Table 10) is passed in the JSON request message. The response message includes the VNFR and the reference to the action “START” successfully performed (i.e. correspondent lifecycle scripts applied in the VNF-Cs).</td>
</tr>
<tr>
<td></td>
<td>VNF operation (stop)</td>
<td>POST</td>
<td>{</td>
<td>{</td>
<td>The full VNFR (see Table 10) is passed in the JSON request message. The response message includes the VNFR and the reference to the action “STOP” successfully performed</td>
</tr>
</tbody>
</table>

© SELFNET consortium 2016
| VNF scale in | POST | { "vnfr": { ...},
                "vnfcInstance": { ...},
                "action": "SCALE_IN" } |
|-------------|------|------------------------------------------------------------------|
| VNF scale out | POST | { "vnfr": { ...},
                "scriptsLink":"linktogroup",
                "scripts":"scriptsfiles",
                "vnfc": { ...},
                "action": "SCALE_OUT" } |
| VNF modification | POST | { "vnfr": { ...},
                "configuration": { ...},
                "action": "CONFIGURE" } |
| VNF software update | POST | { "vnfr": { ...},
                "action": "SOFTWARE_UPDATE" } |

(i.e. correspondent lifecycle scripts applied in the VNF-Cs).

The full VNFR (see Table 10) is passed in the JSON request message, together with the VNF-C instance (see Table 12) to be removed.

The response message includes the VNFR and the reference to the action “SCALE_IN” successfully performed.

The full VNFR (see Table 10) is passed in the JSON request message, together with the new VNF-C (see Table 9) to be instantiated. As per instantiation, the lifecycle scripts to be applied are passed.

The response message includes the VNFR and the reference to the action “SCALE_OUT” successfully performed.

The full VNFR (see Table 10) is passed in the JSON request message, together with the new configuration parameters (see Table 6) to be applied in the correspondent configuration lifecycle scripts.

The response message includes the VNFR and the reference to the action “CONFIGURE” successfully performed.

The full VNFR (see Table 10) is passed in the JSON request message. The response message includes the VNFR and the reference to the action “SOFTWARE_UPDATE” successfully performed (i.e. correspondent lifecycle scripts applied in the VNF-Cs).
### VNF termination

**Request message (JSON content):**
```
{  
  "vnfr": {...},  
  "action":  
    "RELEASE_RESOURCES"
}
```

**Response message (JSON content):**
```
{  
  "vnfr": {...},  
  "action":  
    "RELEASE_RESOURCES"
}
```

The full VNFR (see Table 10) is passed in the JSON request message. The response message includes the VNFR and the reference to the action “RELEASE_RESOURCES” successfully performed (i.e. correspondent lifecycle scripts applied in the VNF-Cs).

### VNF query

**Request message (JSON content):**
```
{  
  "vnfr": {...}
}
```

This method returns a JSON message including the full VNFR (see Table 10) associated to the requested vnfd. It therefore provides information about VNF status and related virtual resources allocated.

### Table 13 SELFNET VNFM OMLvnfm_SAUor REST API (server-side)

<table>
<thead>
<tr>
<th>Orchestrator REST Endpoint</th>
<th>LCM action</th>
<th>Method</th>
<th>Request message (JSON content)</th>
<th>Response message (JSON content)</th>
<th>Description</th>
</tr>
</thead>
</table>
| /admin/v1/vnfm-subscribe   | SELFNET VNFM registration | POST   | {  
  "version":"1.0",  
  "type":"selfnet",  
  "endpointType":"REST",  
  "endpoint":"http://localhost:8081"}   | none                             | When starting, the SELFNET VNFM sends a subscription message to the Orchestrator. The message includes parameters to uniquely identify the SELFNET VNFM, like the type of VNFM, the type of endpoint for the OMLvnfm_SAUor interface and the endpoint URI. |
| /admin/v1/vnfm-core-allocate | VNF allocate request | POST   | {  
  "vnfr": {...},  
  "action":  
    "ALLOCATE_RESOURCES"
}   | {  
  "vnfr": {...},  
  "action":  
    "ALLOCATE_RESOURCES"
}   | The full VNFR (see Table 10) is sent in the JSON request message. The response message includes the VNFR with the virtual resources identifiers. |
### Table 14 SELFNET VNFM OMLvnfm_SAUR or REST API (client-side)

<table>
<thead>
<tr>
<th>Message</th>
<th>LCM action</th>
<th>Request message (JSON content)</th>
<th>Response message (JSON content)</th>
<th>Description</th>
</tr>
</thead>
</table>
| SAVE_SCRIPTS    | VNF instantiation      | `{ "action":"SAVE_SCRIPTS", "payload": {...}, "script-path":"path", "name":"scriptname" }` | `{ "out": "scriptlist" }` {
|                 | VNF scale out          |                                                                                               | "err": "error", "res": "result" }                                                                 | This request message is sent by the SELFNET VNFM to the LCM agent during VNF instantiation to save lifecycle scripts in the VNF-C Virtual Machine. The message include the path where to locally save the scripts (script-path) and the name of the script (name). The payload contains the script encoded in base64. The response message includes the list of scripts currently saved in that path, and in case of error an error string. |
| CLONE_SCRIPTS   | VNF instantiation      | `{ "action":"CLONE_SCRIPTS", "payload": "url", "script-path":"path" }`                     | `{ "out": "scriptlist" }` {
|                 | VNF scale out          |                                                                                               | "err": "error", "res": "result" }                                                                 | This request message is sent by the SELFNET VNFM to the LCM agent during VNF instantiation to clone lifecycle scripts in the VNF-C Virtual Machine from a git repository. The message include the url of the repository (payload) and the path where to clone the scripts (script-path). The response message includes the list of scripts currently saved in that path, and in case of error an error string. |
| EXECUTE         | all                    | `{ "action":"EXECUTE" "env": {...} "payload": "scriptname" }`                                 | `{ "out": "stdout" }` {
|                 |                        |                                                                                               | "err": "error", "res": "result" }                                                                 | This request message is sent by the SELFNET to force the execution of a given script name (payload) into the VNF-C Virtual Machine. The scripts has to be previously saved or cloned. The message includes a list of parameters (env) to be used in the script as environment variables. The response message includes the std out of the script execution, and in case of error an error string. |
| GET_STATISTICS  | VNF query              | `{ "action":"GET_STATISTICS" }`                                                              | `{ "out": "stats" }` {
|                 |                        |                                                                                               | "err": "error", "res": "result" }                                                                 | This message is sent by the SELFNET VNFM to retrieve VNF-C performance statistics. The response message includes the percentage of usage of Virtual Machine CPU, RAM and disk space, in a string format like "cpu:<%>,ram:<%>,disk:<%>". In case of error an error string. |

### Table 15 SELFNET VNFM VNF_OMLvnfm messages (configuration queues)
3.3.1.3.2 VNF_OMLvnfm

The SELFNET VNFM prototype provides an implementation of the VNF_OMLvnfm interface for VNF configuration, start, stop, software update and performance query purposes. As anticipated in section 3.3.1.2, the interface is implemented as a RabbitMQ message service between the SELFNET VNFM and the LCM agent based on the OpenBaton EMS.

The RabbitMQ message service is set up when the SELFNET VNFM starts, and a message exchange broker is created, initialized and started (broker name: openbaton-exchange). Two types of message queues are used in this message service:

- **LCM agents registration queue**: it is a unique queue created by the SELFNET VNFM to let LCM agents of different VNF-Cs to register to the SELFNET VNFM. Messages used over this queue are detailed in Table 16.

- **VNF-C configuration queues**: these are queues created by LCM agents running within VNF-Cs to accept messages from the SELFNET VNFM for lifecycle management purposes. Messages used over this queue are in the direction SELFNET VNFM -> LCM agents. They are detailed in Table 15.

<table>
<thead>
<tr>
<th>Message</th>
<th>Request message (JSON content)</th>
<th>Response message (JSON content)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>{ “hostname”:”host_name” }</td>
<td>none</td>
<td>This message is sent by each LCM agent to subscribe to the SELFNET VNFM. In the request JSON message there is a unique parameter, i.e. the VNF-C hostname, that will be used by the SELFNET VNFM to identify the proper configuration queue where to post messages according to Table 15.</td>
</tr>
</tbody>
</table>

Table 16 SELFNET VNFM VNF_OMLvnfm messages (registration queue)

3.3.1.3.3 SAUvo_OMLvnfm

This SELFNET VNFM prototype does not provide a direct implementation of this interface, which is conceived to retrieve VNFDs from the SELFNET APP repository during VNF instantiations. Indeed, as described in section 3.3.1.3.1, the complete VNFD is passed from the SELFNET Orchestration sublayer to the SELFNET VNFM in the context of the VNF instantiation operation. As a consequence, the SAUvo_OMLvnfm is implicitly implemented in the VNF instantiation interface. Moreover, a local copy of VNFDs related to instantiated VNFs is kept within the SELFNET VNFM; this way, the SELFNET VNFM can easily retrieve information about VNFD attributes, parameters, virtual resource requirements, VNF lifecycle operation properties and constraints, as well configuration and lifecycle scripts details.

3.3.1.4 Release format and installation procedures

The SELFNET VNFM prototype has been developed as an extension of the OpenBaton Generic VNFM [37]. Following the OpenBaton project features and guidelines, the SELFNET VNFM software is released under the Apache 2.0 license as a self-contained Gradle [39] project. The SELFNET VNFM software structure is described in Table 17.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gradle</td>
<td>Folder containing Gradle binaries to compile and build the SELFNET VNFM environment</td>
</tr>
<tr>
<td>src</td>
<td>This folder contains the SelfnetVnfm main Java class implementing the logic of the SELFNET VNFM on top of the OpenBaton framework and libraries.</td>
</tr>
</tbody>
</table>
The enhanced OpenBaton base library including ETSI MANO information models, messages and interfaces implementation

Table 17 SELFNET VNFM software structure

For the SELFNET VNFM, the following openbaton-libs features are installed when the software package is compiled following instructions in section 3.3.1.4.1 below:

- MANO catalogue with VNFD, VNFR, NFV messages data and interfaces
- vnfm-sdk to import the OpenBaton abstract VNFM interfaces and background logic
- vnfm-sdk-rest to import the OpenBaton VNFM REST interface skeleton implementation

3.3.1.4.1 SELFNET VNFM installation

The bootstrap is a non-interactive bash script included in the SELFNET VNFM software package that simplifies the initial installation of the application, also taking care of checking the pre-requisites.

When getting the SELFNET VNFM software, the bootstrap script is the first to be used to prepare and check the environment and have a first compilation of the software stack.

The SELFNET VNFM main pre-requisites are summarized in Table 18.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>openjdk-7-jdk</td>
<td>Java development kit</td>
</tr>
<tr>
<td>rabbitmq-server</td>
<td>RabbitMQ service</td>
</tr>
</tbody>
</table>

Table 18 SELFNET VNFM software pre-requisites

3.3.1.4.2 SELFNET VNFM operation

The selfnet-vnfm.sh is a non-interactive bash script included in the SELFNET VNFM software package that simplifies the compilation, application start/stop procedures.

The selfnet-vnfm.sh supports the options described in Table 19.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Starts the SELFNET-VNFM application</td>
</tr>
<tr>
<td>stop</td>
<td>Stops the SELFNET VNFM application</td>
</tr>
<tr>
<td>kill</td>
<td>Forces the kill of the SELFNET VNFM application</td>
</tr>
<tr>
<td>compile</td>
<td>Compile the software stack and produces the jar file</td>
</tr>
<tr>
<td>clean</td>
<td>Clean the built environment</td>
</tr>
</tbody>
</table>

Table 19 selfnet-vnfm.sh operations

3.3.1.4.3 SELFNET LCM Agent

The SELFNET LCM Agent prototype is released as a python thin service to be installed in each VNF-C Virtual Machine. It has been developed enhancing the OpenBaton EMS. It is basically a thin python software running in each VNF-C that enables the execution of the lifecycle scripts described in 3.2 and 3.3.1.1.1 for VNF-C configuration and operation purposes, including software updates.

The SELFNET LCM agent software structure is very simple and is described in Table 21.

<table>
<thead>
<tr>
<th>Source file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>It is the main class of the LCM agent application and includes the</td>
</tr>
</tbody>
</table>
parsing of the configuration file and instantiation of the RabbitMQ service

| receiver | It implements the consumer service for the RabbitMQ message queue established with the SELFNET VNFM |
| utils | It provides some utils to parse messages and files |

Table 20 SELFNET VNFM software pre-requisites

The SELFNET LCM agent python pre-requisites are summarized in Table 21.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>python 2.7</td>
<td>Reference python version</td>
</tr>
<tr>
<td>pika 0.10.0</td>
<td>Package to access RabbitMQ services</td>
</tr>
<tr>
<td>git</td>
<td>Package to access git repositories</td>
</tr>
<tr>
<td>psutil</td>
<td>Package for collection performance statistics</td>
</tr>
</tbody>
</table>

Table 21 SELFNET LCM agent python pre-requisites

Moreover, for a proper operation of the SELFNET LCM agent is recommended to use debian 64bit image as VNF-C Virtual Machine.

The LCM agent software can be installed in the VNF-C Virtual Machine following two approaches:

- Embed the software in each Virtual Machine that is onboarded in the SELFNET APP repository, thus already installed as a software package
- Dynamic deployment during VNF instantiation procedure, e.g. implementing a cloud-init [35] with metadata procedure when the Virtual Machine is allocated in the VIM

3.3.1.5 Validation and integration with NFV LTE Services

The OpenAirInterface (OAI) open-source LTE software stack has been chosen in SELFNET for the NFV LTE prototype. OAI is essentially a software-based LTE system that provides a full protocol stack implementation of the 3GPP standard both in E-UTRAN and EPC [40][41]. Moreover, OAI allows methods for performance evaluation and pre-deployment system testing of LTE (Rel-8) and LTE-Advanced (Rel-10) [41]. Therefore, OAI will be used in SELFENT to emulate an LTE base station and a core network in a NFV infrastructure, where multiple UEs could be connected in order to test and evaluate different network setups.

OAI EPC software provides MME, S+PGW and HSS functions of the LTE core EPC architecture. Nowadays, SGW and PGW are still merged together. Our main goal with OAI is to achieve a deployment with 5 Virtual Machiness fully operational implementing Virtual Data Network Functions (VNFDs):

- Simulated UE + eNB NFV,
- Emulated eNB NFV with USRP B210 Software-Defined Radio Hardware
- HSS NFV
- MME NFV,
- SGW+PGW NFV

However, the consortium has not yet been able to provide the separation of the MME and the SGW+PGW NFVs. The consortium will continue working with OAI software so that the separation could be achieved in order to enhance the automation and management of the infrastructure and to better reflect an operational scenario.
Despite the missing separation, an automatic building script that runs all the required commands in order to instantiate a full LTE NFV Network (Simulated UE with eNB NFV, HSS NFV, MME + S/PGW NFV) has been developed. The LTE NFV Network is composed by three VNFs:

- Simulated UE with eNB NFV
- HSS NFV
- MME with S/PGW NFV

```
[<root@localhost scripts]>./automation_bash hss mme enb 2
Creating HSS
Allocating 'hss.img' | 50 GB 02:57
Clone 'hss1' created successfully.
Domain hss1 started
... 
Creating eNB
Allocating 'enb2.img' | 50 GB 02:52
Clone 'enb2' created successfully.
Domain enb2 started
... 
```

Figure 29 Output of the Automation_bash script.
This script is divided in two steps, first it is about the creation of the LTE NFV Virtual Machines by cloning a LTE Virtual Machine template hosting the full OAI software previously prepared. A DHCP server is used to assign an IP address to each LTE NFV Virtual Machine. Then, after the creation of the Virtual Machines, the script executes the second step that consists in establishing a ssh connection with all the Virtual Machine by using the IP address provided by the DHCP server and then prepare all configuration files from all the OAI modules to be built and later instantiated. In the Figure 29, it is shown the output of the creation of a full LTE NFV Network by using such script.

In SELFNET, this manual procedure described is exactly what is overcome by the adoption of an ETSI MANO based approach for the lifecycle management of VNFs. The usage of the SELFNET VNFM to instantiate the LTE NFV Network VNFs enable to have a full automatic instantiation and configuration of the three Virtual Machines (each implementing a VNF) mentioned above.

Indeed, the OpenBaton SELFNET VNFM prototype has been validated to implement an automated instantiation of an MME VNF based on the OAI Virtual Machine. This validation has been carried in the Nextworks lab, deploying the SELFNET VNFM on top of an OpenStack Liberty VIM controlling a single server NFV lab infrastructure.

To enable the full lifecycle management operations, the OpenBaton NFV Orchestrator has been also deployed on top of the SELFNET VNFM to take care of the Virtual Machines and virtual networks allocation in the OpenStack VIM. The validation scenario is depicted in Figure 30.

According to the SELFNET VNFM operation, a dedicated VNFD has been prepared for the HSS VNF, as shown in the Annex B. This VNFD includes the reference to two lifecycle scripts to be executed during the VNF instantiation that are the same used in the manual procedure described above to install and configure the HSS application running in the VNF. These HSS VNFD scripts are reported in Annex B.1 and B.2.

The OpenBaton NFV Orchestrator, in particular by means of its dashboard [42], has been used to trigger the HSS VNF instantiation that from a SELFNET VNFM perspective follows the procedure described in the instantiation workflow of section 3.2.1. As a first preparatory step however, the HSS VNFD with the lifecycle scripts have been onboarded to the NFV Orchestrator in the form a VNF package [42].

When the SELFNET VNFM is started, it subscribes to the OpenBaton NFV Orchestrator following the procedure described in section 3.3.1.3.1. A snippet of the related SELFNET VNFM log is reported Annex B.3, and the successful registration on the OpenBaton dashboard is shown in Figure 31.
A set of snippets of SELFNET VNFM logs are reported in Annex B.3 to show the evolution of the HSS VNF instantiation (i.e. instantiate request, resource allocation request and response, VNF configuration, instantiate response), according to the workflows and interfaces supported. As a final validation of the successful HSS VNF instantiation, Figure 32 shows the status ACTIVE of the HSS VNFR in the OpenBaton dashboard, and Figure 33 shows the HSS Virtual Machine up and running in the OpenStack dashboard.
3.3.2 Ubuntu Juju SELFNET VNFM

In Task 2.4 of SELFNET, Juju has been prototypically used to perform the deployment of services in both MaaS and Bare Metal. In this task, the support for Juju has been prototypically validated in other execution environments, focusing on its suitability for OpenStack. The target is to enable Juju to act conceptually as a VNF Manager according to the ETSI MANO architecture, interfacing directly over the VIM. In that case, the VIM functionality is provided by OpenStack.

Juju, as a tool provided by Ubuntu, is able to install, configure and manage services. SELFNET project has extended Juju capabilities to provide support for the SELFNET infrastructure and use cases. It enables the deployment of hundreds of services in the different locations of the Mobile Edge infrastructure. To do so, Juju provides a service catalogue with templates ready to be deployed automatically, including installation and configuration.

Juju service catalogue is composed by both the VNF and the Network Services utilized within the SELFNET framework. In fact, Juju facilitates the onboarding of new VNF and Network Services by means of an encapsulation mechanism, i.e., the so-called charm. A charm encapsulates the files and the deployment logics necessary to deploy the services over a given infrastructure. A charm is implemented by mean of a well-known structure of files. A complete tutorial about this encapsulation method is available in [43]. Charms are depicted as SELFNET VNF Catalogue available in Figure 34, which shows an overview of the Juju architecture prototype for the SELFNET framework.
It is worthy to remark that a complete prototypical validation of Juju over OpenStack has been conducted as part of this deliverable. The rest of this section explains different aspects of the implemented prototype, whilst the conceptual approach employed in Juju is further described referring to Figure 34 and additional illustrations. Figure 35 shows the basic layout of one example of a charm being used in the SELFNET prototyping of Juju to deploy a new eNodeB node in the infrastructure.

Figure 34 Overview of the Juju architecture prototype for SELFNET

Figure 35 Example of the folder structure of the charm to perform the deployment of an eNodeB
Juju has the capability to be able to perform the deployment of services over different heterogeneous infrastructures. In Juju’s terminology, each type of infrastructure is referred to as an environment. An environment defines an execution environment where VNF and services need to be deployed. This capability enables administrators to use the same software to perform the control of service over different types of execution platforms. The environments are also depicted in Figure 34 indicating the fact that multiple environments can be managed by Juju.

In SELFNET, Juju is explored for two different purposes. Firstly, Juju is used to perform the deployment of services over physical infrastructures. Secondly, Juju is also employed to deploy services in virtual infrastructures, mainly interacting with OpenStack and other similar public and private cloud infrastructures. The latter case is the one explored by SELFNET as an implementation of a VNF Manager within the SELFNET Framework. For more details of the SELFNET infrastructure design, Deliverable 2.4 provides a comprehensive description. In summary, Juju has been used in SELFNET as VNFM implementation and configured to interact with OpenStack as VIM.

### 3.3.2.1 OpenStack as Virtual Execution Environment

In the prototyping for this SELFNET task, Juju has been configured to control the OpenStack virtual infrastructure. As part of the bootstrapping of the new Juju environment, a new Virtual Machine is created inside OpenStack in order to act as a management point where to perform the control, deployment, configuration and lifecycle control of the VNF services deployed by the VIM. Figure 36 shows the “juju-openstack-machine-0” representing such a bootstrapping Virtual Machine.

![Figure 36. OpenStack infrastructure where the Juju Bootstrap has been performed and the associated VM is running](image)

The integration of Juju into OpenStack provides SELFNET with the capability to quickly deploy services at large-scale and on-demand. The configuration of a new environment is performed via the environments.yaml file. This file defines the interaction with the VIM and enables Juju to take the control of the infrastructure. Figure 37 shows an excerpt of the configuration used in the prototyping.
3.3.2.2 Juju Agent as EMS Interface

In the ETSI NFV-MANO architecture, the EMS components are in charge of enabling the remote control of VNFs. As depicted in Figure 34 a Juju Agent is equivalent to the EMS component. In order to make this environment work, the prototype has been implemented using a metadata repository where all the different versions of Juju agents are stored to be consequently installed inside the new instances of the VNF in order to gain the control over such a VNF. The number of EMS versions is very significantly in Juju providing support for a number of different operating systems such as Win7, Win8, Win2012, Ubuntu, CentOs, etc. Figure 38 shows an excerpt of the folder containing all the different EMS agents currently supported in the SELFNET prototyping.

```
openstack:
  auth-mode: userpass
  auth-url: http://10.1.7.209:5000/v2.0/
  http-proxy: http://10.1.0.2:8000
  https-proxy: http://10.1.0.2:8000
  network: <Name of the network>
  tools-metadata-url: <Path of metadata tools>
  image-metadata-url: <Path of metadata images>
  no-proxy: localhost,10.1.0.2
  password: <Password>
  region: <name of Region>
  tenant-name: admin
  type: openstack
  use-default-secgrou n: true
  use-floating-ip: false
  username: admin
```

Figure 37. Example environment configuration for Juju over OpenStack

Figure 38. Different versions of Operating Systems supported by the Juju EMS Service

3.3.2.3 VNF Onboarding and Descriptor

The onboarding of a new VNF can be realized through two different approaches. The first approach covers a use case where a VNF has been statically provisioned into the VIM by the administrator. In that case, after registering this new VNF in Juju, the SELFNET framework will be able to deploy this VNF when needed by extracting the ID of the VNF registered in OpenStack and executing a Juju command line “metadata image tool” that generates the VNF Descriptor of this image in Juju. This descriptor is then registered in Juju, thereby having the image ready for its usage. Figure 39 shows a snapshot of the descriptor of a new VNF registered using the static provisioning approach. It is noted that
in the description there is information about the VM ID registered in OpenStack, the region where the VNF needs to be deployed, and the name used inside Juju among other useful configuration files.

![Example](image)

**Figure 39.** The figure shows an example of a VNF statically provisioned in Juju

The other approach to perform the deployment of new services is by means of the interaction directly with the EMS agent. This agent allows performing not only the control of the lifecycle of the services allocated in the VNF but also deploying and configuring new services therein. Deliverable D2.4 [33] of the SELFNET project explains in detail the prototype implemented where a dynamic provisioning of services has been performed in Juju. The way to carry out such installation, deployment, re-deployment, or undeployment of new services is analogous to the methodology explained in deliverable D2.4. The only practical difference is that the new installation is performed over a VNF rather than over the MaaS execution environment in the context of D2.4.

### 3.3.2.4 Interfaces

As shown in Figure 34, the interfaces defined in Deliverable 2.2 of the SELFNET project are presented in the SELFNET Juju architecture. These interfaces mainly include CLcp_OMLvnfm, OMLvnfm_SAUor, OMLvim_OMLvnfm, SAUvo_OMLvnfm, and DNLdp_OMLvnfm. Table 22 shows a summary of these interfaces and what is the approach followed by Juju in order to provide these interfaces.

<table>
<thead>
<tr>
<th>Interface name</th>
<th>Description</th>
<th>Juju approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLcp_OMLvnfm</td>
<td>Basic installation, configuration, deployment and operational management of VNFs in the control plane</td>
<td>Client-Server approach using a YAML model</td>
</tr>
<tr>
<td>OMLvnfm_SAUor</td>
<td>Virtualized compute and network resource management</td>
<td>Via command line tools or API (See Deliverable 2.4)</td>
</tr>
<tr>
<td>OMLvim_OMLvnfm</td>
<td>Virtualized compute and network resource management</td>
<td>Via command line tools or API (See Deliverable 2.4)</td>
</tr>
<tr>
<td>SAUvo_OMLvnfm</td>
<td>Onboard new VNFs and Services</td>
<td>Via command line tools</td>
</tr>
<tr>
<td>DNLdp_OMLvnfm</td>
<td>Basic installation, configuration, deployment and operational management of VNF in the data plane</td>
<td>Client-Server approach using a YAML model</td>
</tr>
</tbody>
</table>

**Table 22 Interfaces in the SELFNET Juju architecture.**
4 Encapsulation of SDN applications

4.1 High level architecture

The concept presented in this chapter is aligned with the ONF proposed SDN architecture as presented in 2.1.2 and more specifically with the aspects identified with respect to the Applications Plane and the interaction foreseen for the A-CPI interface. The proposed solution and early prototype have focused on the sharing of resources among multiple clients as well as the dynamic nature of the environment with respect to the lifecycle management of the SDN Applications. The approach has taken into account Orchestration requirements.

An analysis of the aspects of SDN applications is presented in the following paragraphs. The following definitions are used extensively in this analysis and are, therefore, presented here for clarity.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
</table>
| Application Type Identifier (or APP Type Identifier) | A well-defined identifier (it can be an Object Identifier – OID) of the application type. It refers to the functionality that is provided by the application along with all the possible parameters that the application can handle in any phase (bootstrapping, configuration update, etc.)

The type identifier is extensible and any extension inherits and augments the parent functionality and parameters set by applying more granularity. E.g.:

Type 0.1.2.3 \(\rightarrow\) Load Balancing with configurable Algorithm A and B

Type 0.1.2.3.5 \(\rightarrow\) The same with 0.1.2.3 offering also algorithms C and D. It also allows prioritization of three different traffic types based on destination port.

It is assumed that there is registry with application type identifiers that can be used for lookup of the identifiers and resolving type descriptors and parameter sets. |

| Application Identifier or (APP Identifier) | An application is implementing the functionality described by an APP type identifier. Once onboarded it is available for being selected for activation by the Orchestration Sublayer. When the application is injected in the system it is assigned an identifier that is based on the concatenation of the APP type identifier along with an index indicating the order of insertion among similar APP type implementations. E.g.:

Application 0.1.2.3.5_23 \(\rightarrow\) The 23rd application in the system implementing APP Type 0.1.2.3.5

Apart of the order, the nature of the application (if it is SDN Application or VNF Application) along with any other parameters (like resource or performance requirements) are maintained so that they can be used during the selection process in the Orchestration Sublayer. The nature of the application is used so that the Orchestration Sublayer is aware of the provisioning process it has to follow. |

| Application Instance Identifier | Once an application is instantiated the Application Instance Identifier is created by appending the instance order to the Application Identifier. E.g.: |
There are two kinds of SDN Applications in SELFNET depending on the runtime they are deployed in as well as the way they interact with the SDN Controllers (see Figure 40):

**SDN APPs**

These are software packages that are deployed in a runtime environment outside the SDN Controller and utilise either the Northbound Interface of the SDN Controller or any other network application that is also deployed outside the SDN Controller (this regards cases like the OpenDaylight VTN Coordinator in contrast to the OpenDaylight VTN Manager, an SDN APP may invoke either the VTN Manager or the VTN Coordinator depending on the services the APP is actually implementing). In practical terms an SDN APP is implementing a network application logic that is carried out through a number of transactions with the Northbound interfaces of the SDN Controller plugins focusing on specific high level tasks such as extracting topology information or network metrics, applying any forwarding rules, etc.

**SDN Controller APPs**

These are software packages deployed directly in the SDN Controller runtime environment utilising directly the services provided by other components of the SDN Controller or by southbound protocol plugins. Potentially and SDN Controller APP can be supplied in the form of a protocol plugin.

VNFs that interact with the SDN Controller via the NB interface can be also considered as SDN APPs. However, since these entities require a virtual infrastructure to be provisioned they are considered, in SELFNET, as NFV APPs and they are managed according to the NFV MANO practices. Therefore, in these paragraphs (Chapter 0) the focus is on the lifecycle management of applications that are considered as not so demanding in terms of resources and thus deployable in application containers like Karaf.
4.2 Interfaces and services

An SDN application is subject to be instantiated and activated according to an autonomic plan that will be forwarded to the Orchestration Sublayer by the Autonomic Sublayer. This plan is subject to be composed as a result of policy enforcement which in turn has been previously composed on the basis of potential situations and the way these situations have to be handled. The autonomic plan is expected to focus on certain action types that may potentially mapped also to SDN application types. The availability of an application type is subject to publication upon insertion in the SELFNET ecosystem of any not yet activated application implementing the specific type. A type is also expected to be associated with any configuration parameters the values of which are to be supplied by the Autonomic Sublayer during the composition of the action plan. The processing of the action plan by the Orchestration Sublayer may lead to the instantiation of an SDN application for the fulfillment of a specific action.

According to this approach an SDN application implementing a certain application type, registers such capability once it is inserted in SELFNET ecosystem.

A typical workflow of SDN App installation, activation and invocation is presented in the following figure (Figure 41). The SDN App Manager is the service entity that administers the SDN APP Karaf Runtime. It is configured to bootstrap new bundles with information regarding the SDN Controller with which the APPs have to interact and potentially any monitoring endpoints to which APPs acting as sensors may invoke for publishing.
monitoring data. The SDN App Manager is able to maintain a registry with application types under which it activates the supplied OSGi Bundles that implement the APP Type functionality. Once the information for a new bundle is posted to the App Manager, the package is analysed so that information about the implementation can be extracted and published (type, vendor, and configuration data) to the Orchestration and Autonomic Layers. During this phase the OSGi bundle activation mechanism can be used for creating a single App Management Object from the classes provided in the bundle. This object is used at least for retrieval of information that has to be published (as stated earlier) so that the APP is a candidate solution for the SON Policy and Action Plan. Once a published type becomes part of an Autonomic Plan the orchestration may select to activate an SDN App to fulfill the request (the App type is not restricted to SDN Apps only and may be implemented by VNFs onboarded in the system). At this stage the SDN App Manager is queried for available type implementations and in case the Orchestrator selects a specific SDN App implementation of the indicated type the SDN App Manager is requested to instantiate a special class that implements the service interface for the implemented service logic. In all cases the SDN App Service Class is implementing a SELFNET defined model so that all Service Object instances can be exposed as REST endpoints to be invoked by the Orchestrator for receiving configuration requests. The configuration set passed to these objects follow a generic schema and the object is responsible for enforcing it through the interaction with the SDN Controller.

4.2.1 SDN APP Case

4.2.1.1 SDN APP Lifecycle Management

An SDN Application that operates on top of an SDN controller in a separate runtime environment by interacting with the controller through the SDN controller’s Northbound interface is implemented as an OSGi bundle. The OSGi environment hosting SDN APPs is based on the Apache Karaf Container and it is administered by a special bundle that will be thereafter called SdnAppManager. The SdnAppManager exposes a number of REST

Figure 41: SDN APP Installation, Activation and Invocation workflow
endpoints supporting CRUD operations that are intended to be used in the context of the operation of the Application Manager in Orchestration Sublayer. These endpoints are presented in Table 23. Conceptually the SdnAppManager may be a component of the WP6 Application Manager, however, we analyse its operation separately in order to highlight the encapsulation and onboarding features of the SDN APP approach.

<table>
<thead>
<tr>
<th>REST Endpoint</th>
<th>Lifecycle Management Context</th>
<th>Method/Operation</th>
<th>Sent Information</th>
<th>Return Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>Type registration and registration of the first implementation of the type</td>
<td>POST/create</td>
<td>APP Type Identifier</td>
<td>The APP Type Identifier supplied in the call as verification</td>
<td>Adds a new APP type ID.</td>
</tr>
<tr>
<td>/{app type id}/</td>
<td>Enumeration of registered types</td>
<td>GET/read</td>
<td>APP Identifiers Available</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUT/update</td>
<td>Not used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DELETE/delete</td>
<td>Not Used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/{app type id}/</td>
<td>Registration of an additional type implementation</td>
<td>POST/create</td>
<td>Bundle locator</td>
<td>APP Identifier (APP Type ID + Order Index)</td>
<td>Adds a new APP type ID and registers the first app implementing it. If the APP type is already created it just registers the implementation.</td>
</tr>
<tr>
<td></td>
<td>Enumeration of a specific type implementations</td>
<td>GET/read</td>
<td>APP Identifiers Available for specific type</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUT/update</td>
<td>Not used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DELETE/delete</td>
<td>ACK/NACK</td>
<td></td>
<td>Drops the APP type and all implementing bundles and their instances</td>
<td></td>
</tr>
<tr>
<td>/{app type id}/</td>
<td>SDN APP instantiation</td>
<td>POST/create</td>
<td>Tenant ID</td>
<td>APP instance identifier (APP Type ID + Order Index + instance number)</td>
<td>Instantiates the service object of the application for the indicated Tenant and assigns to it a REST endpoint.</td>
</tr>
<tr>
<td></td>
<td>Enumeration of instances of a specific APP Type implementation</td>
<td>GET/read</td>
<td>Optional Tenant ID</td>
<td>APP instances available of specific APP identifier, filtered per Tenant if ID is provided</td>
<td></td>
</tr>
<tr>
<td>/{app type id}/</td>
<td>SDN APP software update</td>
<td>PUT/update</td>
<td>Bundle locator</td>
<td>Update a specific APP</td>
<td>Stops all instances of the specific bundle, refreshes the bundle and restart all the instances</td>
</tr>
<tr>
<td></td>
<td>SDN APP termination</td>
<td>DELETE/delete</td>
<td>Optional APP instance identifier</td>
<td>ACK/NACK</td>
<td>If APP instance identifier is provided the APP instance is terminated otherwise all the instances of a bundle are terminated</td>
</tr>
</tbody>
</table>

Table 23: SDN APPs Lifecycle Management Operations
The SDN APP is implemented as a Java OSGi bundle that can be installed in the Karaf container administered by SdnAppManager. All SDN APP bundles provide:

- the implementation class of a generic SDN APP Interface that defines the four CRUD methods annotated with the corresponding HTTP methods, the content type of consumed and produced information. This class is subject to several instantiations (SDN APP Service Object) by the SDNAppManager on the basis of the enforcement of the autonomic action plans.

- the implementation of a management interface class that can be queried for the APP Type Identifier that the SDN APP is implementing and the service parameters that the service objects can produce or consume. This class is instantiated only once (SDN App Management Object) for every SDN APP bundle.

The type registration that associates the application type implementation with the application identifier is achieved either by the instantiation of the management class or by the bundle installation. Either the SDNAppManager interacts with the SDN App Management Object or this object triggers an event that is captured by the SDNAppManager. During bundle installation the service class, which performs the service tasks of the application type, is registered in the runtime for future instantiations. The service class is instantiated on demand and is associated with one tenant for whom it performs the envisaged networking functionality applying configurations to or extracting information from the tenant’s networking resources. An instantiated service object exposes a REST endpoint supporting the four HTTP methods (POST, GET, PUT, DELETE) that correspond to CRUD operations (create, read, update, delete) so that it can be properly integrated with other components (e.g. monitoring, orchestrator).

The following Figure 42 presents this concept:

**Figure 42: SDN APP Container**
4.2.1.2 SDN APP Operation

All SDN APPs can be accessed by the Orchestration Sublayer through a common REST interface that produces and consumes information according to a generic schema. The received requests are processed by the APP logic and mapped onto specific SDN Controller NB invocations. Similarly, any information returned by the SDN APP (e.g. in the case of monitoring) is also structured according to the generic schema. In both cases the information is based on a sequence of the following elements:

```
<item>
  <paramDescription>Parameter_1</paramDescription>
  <paramIdentifier>0.1.2.4</paramIdentifier>
  <paramValue>300</paramValue>
</item>
```

Either these parameters are pushed to or pulled from the SDN APP, it is up to the APP implementation to map it with the proper information exchanged with the SDN controller.

As indicated above every parameter is uniquely identified by an OID. These OIDs are associated with the APP type identifier and are common among all APPs implementing the type independently of the SDN or VNF nature of the APP. Moreover, every extension to a specific APP Type has to support the existing parameter identifiers and potentially augment the set with new or other existing ones. Both APP type and parameter registration is taking place upon population of the SELFNET registry. This process is expected to proceed in parallel with the onboarding of the APPs. APP development should take care of already existing object identifiers (types and parameters) and adapt the development of the APP as far as the processing and exchange of the information is concerned.

Based on the previously described operations of SdnAppManager the class that implements the generic SDN APP Interface is instantiated and the instance is registered as the serving bean reachable via a specific path created as indicated in the Table 24 below along with the REST Method/CRUD operations provided.

<table>
<thead>
<tr>
<th>REST Endpoint</th>
<th>Method/Operation</th>
<th>Exchanged Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ {app id instance}</td>
<td>POST/create</td>
<td>All exchanged information is formatted according to the generic schema presented above and the methods are implementation specific. Orchestration Sublayer is expected to invoke mainly the PUT/update functionality whereas monitoring modules are expected to invoke GET/read.</td>
</tr>
<tr>
<td></td>
<td>GET/read</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUT/update</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DELETE/delete</td>
<td></td>
</tr>
</tbody>
</table>

Table 24: SDN APPs Control Interface

In order to avoid interactions among instances so as to maintain multi-tenant support, the SdnAppManager does not allow the existence of static fields and functions as well as separate activators and exporting of OSGi services as far as the service classes are concerned. Dependencies on other generic bundles and Karaf features are manifested by use of the Karaf feature mechanisms.

4.2.2 SDN Controller APP Case

4.2.2.1 SDN Controller APP Lifecycle Management

SDN Controllers, like ONOS and OpenDaylight, are based on the Apache Karaf OSGi environment that supports intrinsically the on-the-fly activation of additional plugins and application in the same runtime with the core bundles that provide the functionality of the controllers. In practice, there are several projects around the controller mainstream
development that contribute to the controller application, protocol and plugin ecosystem. This feature offers, as already mentioned, the possibility to create SDN Controller Applications that are activated in the same runtime with the rest of the controller functionalities. In SELFNET architecture, the SDN controller plays a prominent role targeting both infrastructure (virtual resources and service chaining) management needs but also a big part of the tenant based autonomic operation. Consequently, the injection, activation and operation of additional custom SDN controller bundles beyond those that are already offered by the controller repositories has to be handled with increased attention for the following reasons:

- A controller bundle may potentially operate inappropriately, intentionally or not, ruining in this way either the control or forwarding plane operation that in turn may harm the overall architecture.
- A bundle activated in the same runtime with controller may potentially get access to all southbound protocols as well as to all controller exposed services and therefore to all forwarding plane resources independently of any tenant based restrictions.

Taking into account the support of the Role Based Access Control in Karaf runtime, the first case may be potentially restricted by use of this mechanism. Thus, SDN Controller Apps may be activated in Karaf under a specific role for which access to other services is clearly defined in the corresponding lists. This approach is subject to further analysis once the type of the SDN controller is identified.

As far as the second case is concerned, we assume that any SDN Controller APP is subject to Northbound configuration. The type of the controller identifies the model to be followed (YANG, MD-SAL, XML, JSON, NETCONF, RESTCONF). In this way the configuration items intended to be applied to the SDN Controller APPs can be subject to analysis on the basis of a Tenant based separation of resources. According to this feature no configuration that may potentially mixing resources belonging to different tenants will be allowed to be sent to the NB interface of the controller. This requires advance knowledge of the actual configuration model that is supported by the controller. Once an SDN Controller APP is deployed in the SDN Controller runtime it augments the NB interface by the addition of the extra endpoints that this application implements. The following figure (Figure 43) presents this concept.

![Figure 43: SDN Controller Application Instance](image.png)
Unlike the SDN APPs, the SDN Controller APPs are not subject to multiple instance management as there is no SdnAppManager concept in the SDN Controller to manage the various instances and associate these with their corresponding endpoints. Implementing such concept for the SDN Controller OSGi environment is not considered as an option – at least at this phase of the project – as there may be imposing additional complexity with respect to the application development but it also may be in conflict with the existing NB endpoint management. However, in order to maintain a certain level of consistency the SdnAppManager can be also used for the lifecycle management of the SDN Controller APPs. Obviously, not all functionalities are meaningful in this case. The following table (Table 25) presents the subset of the SdnAppManager operations that is meaningful in the SDN Controller APP case along with the required adaptation in the description.

<table>
<thead>
<tr>
<th>REST Endpoint</th>
<th>Lifecycle Management Context</th>
<th>Method/Operation</th>
<th>Sent Information</th>
<th>Return Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>SDN Ctrl APP installation</td>
<td>POST/create</td>
<td>Bundle locator</td>
<td>APP Type Identifier</td>
<td>The APP type may be potentially used to identify that the APP is to be loaded on the SDN Controller. The manager loads the new bundle on the SDN Controller. The Order Index might not be meaningful in this case.</td>
</tr>
<tr>
<td></td>
<td>SDN Ctrl APP Enumeration</td>
<td>GET/read</td>
<td>SDN Controller APP Type Identifiers Available</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUT/update</td>
<td>Not used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DELETE/delete</td>
<td>Not Used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/{app type id}/</td>
<td>SDN Ctrl APP instantiation</td>
<td>POST/create</td>
<td>empty info set (i.e. {})</td>
<td>APP Type ID</td>
<td>Instructs the SDN Ctrl Karaf to activate the bundle. Nothing happens if it is already activated</td>
</tr>
<tr>
<td></td>
<td>GET/read</td>
<td>OSGi status of the bundle.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SDN Ctrl APP update</td>
<td>PUT/update</td>
<td>Bundle locator</td>
<td>Update a specific APP</td>
<td>Stops the specific bundle, refreshes the bundle and restarts all the instances</td>
</tr>
<tr>
<td></td>
<td>SDN Ctrl APP termination</td>
<td>DELETE/delete</td>
<td>ACK/NACK</td>
<td>Drops bundle</td>
<td></td>
</tr>
</tbody>
</table>

Table 25: SDN Controller APPs Management Operations

The SdnAppManager utilises any control interface offered by the SDN Controller runtime (CLI, SSH, etc.)

4.2.3 Multitenancy Issues

Multitenancy is at the heart of SELFNET principles and this aspect has to be enforced with respect to the transactions between the SDN APPs and the SDN Controller. This should address both the information sent to the controller and the information retrieved from the
controller. For example, an SDN APP instantiated for a specific Tenant should not be able to retrieve information from the controller that may be listing data plane resources allocated to a different tenant. Similarly, configuration information sent to the controller endpoints should not include any identifiers or parameters that may be associated with a different tenant.

Normally, an SDN Controller has a single pair of administrative credentials which implies that no multi-tenant separation can be enforced. For this purpose, an intermediate component is introduced to restrict the access of SDN APPs only to the information and configuration of subnets and networks allocated to the tenant for whom the policy rules have led to their instantiation. The actual design of the access control component will evolve in the context of WP6, however, it is foreseen that this SELFNET defined component will maintain a number of proxy objects, one for every SDN controller northbound endpoint, which will be responsible to examine resource identifiers addressed in every request originating from the SDN APP Service Objects. The objects will proxy only those requests to the actual SDN Controller endpoints that address only resources that have been identified (potentially collected through the WP6 Resource Manager where tenant related lists can be maintained) as belonging to the identified tenant.

The SDN APP instance is configured with a token to be used in all REST transactions with the SDN Controller this token is associated with the specific tenant for whom the application instance has been launched during orchestration. The transactions, however, are served through a proxy instance that depending on the resources addressed in the requests (network, subnets) is rejecting or allowing the request to reach the SDN Controller service endpoints. In the case of information retrieval from the SDN controller, the reply is stripped off any information belonging to any tenant which the application instance has not been linked with. The SDN Access Resolution component is expected to interact with the Resource Manager within WP6 components so as to get information

![Figure 44: SDN APP Access Control](image-url)
regarding the resource identifiers relating to every tenant. The functionality of the proxy instance depends on the SDN controller selection.

4.3 Prototype release notes

In the current phase the development with respect to the SDN APPs has focused on the concept of the SDNAppManager and its operation inside a Karaf runtime environment. Taking into account that this component will be further enhanced and completed during WP6 in view also of the progress and availability of SDN App, the current prototype is a proof of concept implementation that provides the know-how with respect to the following:

- Apache Karaf installation and maintenance (features, blueprints and hot-plug mechanisms)
- Integration and verification of Apache CXF with Karaf
- Verification of dynamic publication of REST endpoints based on a common Java Interface model
- Management and layering of App Types, App Type Implementations and App Instances
- Dynamic onboarding of bundles through the SDNAppManager and analysis of Java archives and enclosed classes.

The prototype consists of four packages:

- The Apache Karaf runtime, properly configured with Apache CXF for REST support
- An OSGi library bundle that provides the common model for SDN Apps
- The OSGi blueprint of the SDNAppManager
- A Sample OSGi bundle as SDN App example.

Once the Karaf environment is up and running, the jar file of the SDNAppManager can be copied in the deploy folder of Karaf.

```
cp target/com.osg.sdn.app.manager-1.0-SNAPSHOT.jar .././apache-karaf-4.3.4/deploy/]
```

The blueprint jar is detected and installed along with the library bundle that provides the App model.

```
Karbon@root()$ list
START LEVEL 100 , List Threshold: 50
ID | State  | LVL | Version | Name
---|--------|-----|---------|---
144 | Active | 80 | 1.1.1   | geronimo-ۃاط_1.1-spec
145 | Active | 80 | 2.0.9   | Apache MINA Core
174 | Installed | 80 | 1.0.0-SNAPSHOT | SdnAppCommon OSGi Bundle
Karbon@root()$ Started SDN APP MANAGER
```

```
Karbon@root()$ list
START LEVEL 100 , List Threshold: 50
ID | State | LVL | Version | Name
---|-------|-----|---------|---
144 | Active | 80 | 1.1.1   | geronimo-ۃاط_1.1-spec
145 | Active | 80 | 2.0.9   | Apache MINA Core
174 | Resolved | 80 | 1.0.0-SNAPSHOT | SdnAppCommon OSGi Bundle
175 | Active | 80 | 1.0.0-SNAPSHOT | com.cse.sdn.app.manager Blueprint Bundle
Karbon@root()$ []
```

The SDNAppManager exposes the set of REST endpoints presented in Table 23.
The SDNAppManager can be invoked to install a new bundle under a specific service type (Figure 46). The bundle locator is supplied and the information is posted at the envisaged APP Type path. The bundle is installed in Karaf and its service class is registered for activation that is requested by posting a tenant identifier to the REST endpoint identified by the concatenation of App Type and App Identifier (Figure 47).
Configuration information can be sent thereafter to the SDN App instance by posting content to the proper URL denoting the instance identifier.

Available SOAP services:

Available RESTful services:

Endpoint address: http://localhost:9000/1.2_0_0
WADL: http://localhost:9000/1.2_0_0?wadl

Endpoint address: http://127.0.0.1:8181/ctf/manager
WADL: http://127.0.0.1:8181/ctf/manager?wadl
In the above examples the exchanged information is not structured using any markup or object notation techniques. It is expected that these structures will be specified during WP6.

The design and early developments that have been presented in this chapter are a sufficient proof of concept with respect to the envisaged life cycle management of SDN Applications. Certain aspects such as dynamic deployment, application type identification and per tenant application activation have been clarified and resolved from a technical point of view, whereas certain interactions with WP6 components for orchestration and access control mechanisms have been also addressed.

![Figure 49: Interaction with SDN APP](image-url)
5 Conclusions

This deliverable has presented the design and prototype implementation of the SELFNET NFV and SDN applications encapsulation. In practice, this document defines how SELFNET sensors and actuators applications are dynamically and automatically plugged into the SELFNET virtualized network infrastructure in response to specific needs coming from self-optimization, self-healing, self-protection and composed use cases. The released prototypes provide common lifecycle management primitives for VNFs and SDN applications instantiation, configuration, start, stop, modification and termination.

The design and specification of the SELFNET mechanisms and procedures for encapsulation of NFV and SDN applications have not been defined from scratch. Indeed, reference architectures from relevant standardization bodies have been investigated and evaluated to find suitable solutions to be adopted, reused and enhanced where needed in support of the SELFNET requirements. First, the ETSI MANO framework has been identified as the reference baseline for the NFV applications encapsulation architecture approach. In particular, the VNFM component is the one selected for the implementation of the SELFNET VNF lifecycle management primitives. Encapsulation of VNFs is implemented in SELFNET by means of a dedicated LCM agent embedded in each sensor or actuator VNF which exposes a uniform interface towards the SELFNET VNFM for instantiation, configuration, modification, start, stop and termination purposes.

On the SDN side, the latest ONF SDN architecture specification released in February 2016 has been used as a reference to define and separate roles and scopes of applications running within and outside an SDN controller environment. In addition, the application encapsulation solutions adopted by the two most used and deployed SDN controllers at the time of writing, i.e. OpenDaylight and ONOS, have been investigated for reuse and adoption for the SELFNET encapsulation of SDN applications. Here, the OSGi framework combined with Karaf application containers for dynamic deployment, configuration, modification and termination of SDN applications have been selected as the reference SELFNET encapsulation technologies. It is important to highlight that the encapsulation solution adopted in SELFNET, based on a dedicated SDN Application Manager running as a lifecycle management OSGi bundle within a Karaf container is independent from the SDN controller used (i.e. ODL or ONOS). Indeed, the consortium is continuing its deep exploration of ODL and ONOS capabilities and functionalities with the aim of selecting the reference SELFNET SDN controller. The possibility to support both of them is still an option under evaluation.

As per NFV and SDN applications encapsulation design, the software development has also not been carried out from scratch. Open source tools available in the state-of-the-art have been evaluated as reference technologies to be used as starting point for the prototypes implementation.

Tools like OpenBaton, OpenMANO, Tacker have been investigated and compared for the SELFNET VNFM implementation. The selected reference baseline is OpenBaton, since it natively supports most of the ETSI MANO functionalities and workflows, with VNFM and NFVO roles clearly separated. Moreover, OpenBaton is highly extensible and offers a set of SDKs to integrate with external applications and develop new drivers and plugins for specific VIMs. The ETSI MANO compliant SELFNET VNFM prototype implemented on top of the OpenBaton framework is released with this deliverable, with interfaces, information models and installation guidelines fully described. A preliminary validation of this VNFM prototype has been also carried out to instantiate and configure a VNF based on the OpenAirInterface LTE software stack. As a further prototype activity in the context of NFV applications encapsulation, the open source Ubuntu Juju tool has been also validated as an ETSI MANO compliant VNFM directly interfaced with an OpenStack managed infrastructure.
For the SDN applications encapsulation part, as said, OSGi framework and Apache Karaf containers have been selected as reference technologies for the development of the SDN Application Manager lifecycle management functionalities. The prototype released with this deliverable provides mechanisms and interfaces for deployment, instantiation, configuration, modification and termination of SDN applications running within and outside SDN controllers runtime environments. This SDN Application Manager prototype has to be considered as a first version that will be improved and refined in the context of WP6 activities due to its affinity with the SELFNET Orchestration functionalities in the SAL.

As part of the next steps associated to the outcomes of this deliverable, the software prototypes released for VNF and SDN applications lifecycle management will be integrated with other SELFNET components in support of the use cases workflows defined in deliverable D2.2. In particular, these integration activities will mostly involve the WP6 Orchestration sublayer components (which will provide NFV Orchestration functionalities according to the ETSI MANO principles) and WP3 for the SELFNET APP repository that will manage VNF and SDN applications on boarding.
References


[34] ETSI GS NFV-SWA 001 V1.1.1, “Network Functions Virtualisation (NFV); Virtual Network Functions Architecture”, ETSI NFV ISG, December 2014


[38] OpenBaton Generic EMS, https://github.com/openbaton/ems


[43] Ubuntu Juju Charms, https://jujucharms.com/docs/1.25/authors-charm-writing
Annex A - Example of VNFD

```json
{
    "name": "virtual-ips",
    "version": "1.0",
    "type": "virtual-ips-small",
    "vendor": "nextworks",
    "endpoint": "selfnet",
    "lifecycle_event": [{
        "event": "INSTANTIATE",
        "lifecycle_events": [
            "setup.sh"
        ]
    }, {
        "event": "TERMINATE",
        "lifecycle_events": [
            "shutdown.sh"
        ]
    }],
    "configurations": {
        "name": "vips-parameters",
        "configurationParameters": [{
            "confKey": "param1",
            "value": "value1"
        }, {
            "confKey": "param2",
            "value": "value2"
        }]
    },
    "vdu": [{
        "name": "vips",
        "vm_image": [
            "ubuntu-14.04-server-cloudimg-amd64-disk1"
        ],
        "lifecycle_event": [{
            "event": "CONFIGURE",
            "lifecycle_events": [
                "config.sh"
            ]
        }],
        "virtual_memory_resource_element": "1 GB",
        "computation_requirement": "2",
        "vimInstanceName": "selfnet-openstack",
        "scale_in_out": 2,
        "vnfc": [{
            "id": "vips-01",
            "connection_point": [{
                "floatingIp": "random",
                "virtual_link_reference": "test1"
            }, {
                "floatingIp": "random",
                "virtual_link_reference": "test2"
            }]
        }],
        "deployment_flavour": [{
            "id": "flavour-test",
            "constituent_vdu": [{
                "number_of_instances": 1,
                "constituent_vnfcs": [{
                    "vnfc-id": "vips-01"
                }],
                "vdu_reference": "vips"
            }],
            "flavour_key": "m1.small"
        }],
        "virtual_link": [{
            "name": "test1"
        }, {
            "name": "test2"
        }]
    }]
}
```
Annex B – SELFNET VNFM validation

B.0 - HSS VNFD

```json
{
    "vendor": "PROEF",
    "version": "0.1",
    "name": "hss",
    "type": "hss-test",
    "endpoint": "selfnet",
    "vdu": [{
        "vm_image": [
            "OpenAirVM"
        ],
        "computation_requirement": "",
        "virtual_memory_resource_element": "1024",
        "virtual_network_bandwidth_resource": "1000000",
        "lifecycle_event": [
        ],
        "vimInstanceName": "vim-selfnet",
        "vdu_constraint": "",
        "scale_in_out": 2,
        "vnfc": [{
            "connection_point": [{
                "virtual_link_reference": "private"
            }
        }],
        "monitoring_parameter": [
            "cpu_utilization"
        ],
        "virtual_link": [{
            "name": "private"
        }],
        "connection_point": [
        ],
        "lifecycle_event": [{
            "event": "INSTANTIATE",
            "lifecycle_events": [
                "hss_configure.sh",
                "hss_start.sh"
            ]
        }],
        "monitoring_parameter": [
            "cpu_utilization"
        ],
        "deployment_flavour": [{
            "flavour_key": "m1.small"
        }]
    }
}
```
B.1 – hss_configure.sh

#!/bin/sh

#VARIABLES
hssName="hss"
user=root
password=osboxes.org

#PATHS
hssConf=/usr/local/etc/oai
hssBuildRun=/home/osboxes/openair-cn/SCRIPTS
hssRunOutput=/nfs/output

#BUILD
echo making build for hss
cd $hssBuildRun
./build_hss
sleep $(15))

#CONFS
echo changing hss configuration file
sed -i -e 's/hssadmin/'$user'/g' -e 's/admin/'$password'/g' $hssConf/hss.conf
sleep $(1))

exit

B.2 – hss_start.sh

#!/bin/sh
cd `dirname /home/osboxes/releases/openair-cn-master/SCRIPTS`
/bin/sh ./run_hss
B.3 - SELFNET VNFM logging snippets

SELFNET VNFM subscription

```java
--- [main] o.o.c.vnfm_sdk.rest.VnfmRabbitHelper : Initialization of VnfmRabbitHelper
--- [main] o.o.common.vnfm_sdk.AbstractVnfm : creating VnfmManagerEndpoint for vnfm endpointType: REST
--- [main] o.o.common.vnfm_sdk.rest.VnfRestHelper : Waiting for Orchestrator to accept Selfnet VNFM registration ... (0 secs)
--- [main] o.o.common.vnfm_sdk.rest.VnfRestHelper : BODY is: {"version":0,"type":"selfnet","endpointType":"REST","endpoint":"http://localhost:8081"}
--- [main] s.w.s.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice:
```
VNF allocate request message

```
DEBUG 1768 ... [pool-3-thread-4] o.o.common.vnf_sdk.rest.VnfRestHelper
        : body is: ("virtualNetworkFunctionRecord":{"version":0, "auto_scale_policy":[]}, "connection_point_key": "mi.small", "configurations": [{"version":0, "event": "INSTANTIATE", "lifecycle_events": ["i_start.sh"]}], "lifecycle_event_history": [], "monitoring_parameter": [{"cpu_utilization": [], "vdu": [{"version":0, "vm_image": ["OpenNFD", "82e7faa-8dcf-4146-949b-1014e9a7d03f", "lifecycle_event": [], "vdu_constraint": [], "fault_management_policy": []}], "connection_point": [{"virtual_link_reference": ["private", "version":0]}, ["vnf_instance"]: [{"connection_point_key": "mi.small", "virtual_link": ["private", "version":0]}, ["virtual_link_reference": ["private", "version":0]}, ["virtual_link": ["private", "version":0]}
```

VNF allocate response message

```
DEBUG 1768 ... [pool-3-thread-4] o.o.common.vnf_sdk.rest.VnfRestHelper
        : Received from ALLOCATE: VirtualNetworkFunctionRecord[audit_log='null', id='1ab640a9-dcd4-4a06-82f8-2aa8b90d5899', version=0, event='INSTANTIATE', lifecycle_event_history=[], localisation=null, monitoring_parameter={"cpu_utilization": [], "vdu": [{"version":0, "vm_image": ["OpenNFD", "82e7faa-8dcf-4146-949b-1014e9a7d03f", "lifecycle_event": [], "vdu_constraint": [], "fault_management_policy": []}], "connection_point": [{"virtual_link_reference": ["private", "version":0]}, ["virtual_link": ["private", "version":0]}
```

Save lifecycle scripts into LCM agent (EMS)

```
DEBUG 1768 ... [simpleAsyncTaskExecutor-1] o.o.vmf.selfnet.util.EmrsRegistrator
        : [HTTP]Request received (Hostname: "hss-hss")
```

© SELFNET consortium 2016
Execution of lifecycle scripts for INSTANTIATE event into LCM agent (EMS)

VNF instantiation response message

<table>
<thead>
<tr>
<th>Lines</th>
<th>Description</th>
</tr>
</thead>
</table>