TNOVA

NETWORK FUNCTIONS AS-A-SERVICE
OVER VIRTUALISED INFRASTRUCTURES

GRANT AGREEMENT NO. 619520

Deliverable D3.1

Orchestrator Interfaces

Editor José Bonnet (PTIn)

Contributors J. Carapinha, P. Neves, M. Dias, B. Parreira (PTIn), M. McGrath, G. Petralia, V. Riccobene (INTEL), P. Paglierani (ITALTEL*)*, J. Ferrer Riera, J. Batallé (i2CAT), M. Di Girolamo, P. Magli, L. Galluppi, G. Coffano (HP), A. Ramos, J. Melián (ATOS), P. Harsh (ZHAW)

Version 1.0

Date September 30th, 2015

Distribution PUBLIC (PU)

Executive Summary

The Orchestrator is the core sub-system of the T-NOVA Architecture. This deliverable describe its northbound, southbound and internal interfaces.

A REST-based API was adopted for all the Orchestrator's interfaces, both for external and internal (with JSON over HTTP), where information organised in resources, independently of its internal representation can be exposed, created, updated or deleted. This option for the interfaces naturally leads to the choice of separating the functionality of the Orchestrator into microservices, an approach that is becoming the de facto standard way of organising application, bringing advantages such as (programming) language-, platform- and operating system-agnosticism, as well as independent deployments, updates, replaces and scaling This level flexibility also results in operational challenges, which will be addressed in other deliverables ofWP3.

The fact that there is the possibility of some VNFs to bring their own specific manager raises some issues (e.g., would a Service Provider allow provisioning and monitoring of VNF instances managed externally in its own infrastructure?) that aren't yet clear enough to support the full development of such a feature. However, the microservice based design that has been adopted for the Orchestrator, together with RESTful APIs for the microservices (even those not directly exposed to external usage/interactions), should provide sufficient flexibility to support rapid implementation of support for new services with manageable impact existing modules.

We have opted to expose some of the initially considered internal interfaces for two reasons: first, as outlined the possibility of having to accommodate specific VNF Managers (i.e., turning an initial internal interface between the Orchestrator and the VNF Manager into an external interface); second, the need to provide those services (e.g., authentication and authorisation) to other modules that though being internal, were developed with a totally different technological stack and would benefit from sharing the service. These interfaces have been designed following the same rules as the external ones.

The **key takeaways** of this deliverable are:

- The T-NOVA Orchestrator is built with **RESTful APIs**, with **JSON** over **HTTP**, a state of the art approach;
- These interfaces support ETSI's notion of **specific VNF Managers**, but the implementation still has to mature due to the details like security, performance, etc., of such specific component that still have to be analysed;
- This deliverable also describe the **VNF Manager interfaces**, weather this component be generic (when those interfaces would be internal) or specific (when those interfaces would be internal), so that the exact implementation can be designed later, in a more knowledgeable state;
- As the Orchestrator architecture was designed following a **microservices** approach, the development has an extra degree of choice in terms of programming language, platform and Operating System to use in each one of

the microservices, as well as completely decoupling the form of deploying, scaling, etc. of each microservice.

Table of Contents

Figures

Tables

1. INTRODUCTION

The primary goal of Task 3.1, Orchestrator Interfaces, is to implement the external interfaces of the T-NOVA Orchestrator, specifically its Northbound interfaces with the Network Functions (NF) Store, the Marketplace and the Operational Support Systems (OSS), and its Southbound interfaces with the Virtual Infrastructure Manager (VIM) and the deployed Virtual Network Function (VNFs). These interfaces were first drafted in [1].

As the work in the task has progressed, and to support some of the Use Cases, it was necessary to have a third sub-system interacting with the Orchestrator through its Southbound interface, named Wide Area Network (WAN) Infrastructure Connection Management (WICM).

It was also concluded that interfacing with a generic OSS would be of a little value, and covering the available commercial offerings or choosing a sub-set of them would result a level of effort that would have more valuable covering other aspects of the Orchestrator that were not yet clear. Nevertheless, we believe that the architecture design and its interfaces will make it easy to interface with a specific OSS, once that OSS and the interfacing technologies are chosen.

Figure 1-1 represents a simplified view of some of the T-NOVA Orchestrator modules and the other sub-systems having interfaces with it (see Annex A for more details).

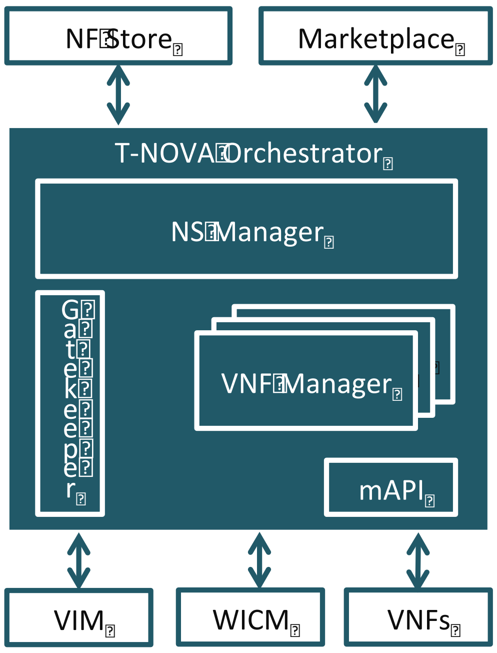


Figure 1-1: The T-NOVA Orchestrator Architecture, simplified.

Figure 1-1 shows the main T-NOVA Architecture sub-systems that interface with the Orchestrator:

- The **NF Store**, where **Function Providers** upload their VNFs;

- The **Marketplace**, where **Service Providers** compose Network Services (NSs) by using available VNFs and **Customers** purchase those services;
- The **VIM**, where the infrastructure supporting the VNFs execution for a given purchased NSs can be created;
- The **WICM**, a facade to the real network, where the connection between the requested service and the remaining network connecting the PoPs can be requested;
- And the **VNFs** themselves for supporting the T-NOVA VNF lifecycle.

Within the Orchestrator itself three modules are represented:

- The **VNF Manager**(s), representing VNFs that have their own (specific) VNF Manager (see [2]);
- The **Gatekeeper**, an authentication and authorisation module for all the modules of the Orchestrator;
- The **middleware API** (mAPI), which abstracts the interface(s) for different VNFs;

There are other internal modules, but since their interactions are kept fully internal, their interactions and interfaces are not detailed here.

This document describes the interactions and interfaces between all these systems. These interfaces can be grouped into two categories, as described in the next sub-section. Since the VNFM may be specific to a VNF, some options had to be taken, as outlined next.

1.1. Top-down vs. Bottom-up interfaces

The interfaces documented in this deliverable may be classified in two broad groups, in terms of amount of information conveyed and number of requests made:

- **Top-down** flows will typically have lengthier messages (VNFs, NSDs, HEAT Templates), but occur less frequently;
- **Bottom-up** flows will typically contain short messages (responses to the top-down requests mentioned above and monitoring data values), but occur with a (very) high frequency.

These two very distinct characteristics might impose an optimisation step later in the development process, namely for the bottom-up flows, due to the higher latency HTTP has (see [3]), when compared to other transport protocols (e.g., User Datagram Protocol, UDP, see [4]) used by the industry when low latency is a requirement (e.g., in Monitoring or Logging systems). As work progresses and we measure the performance of the different microservices, we may change the implementation of bottom-up interfaces that are less performing.

1.2. Specific VNF Managers

The design of the Orchestrator addresses the challenges of accommodating a VNF with its own (specific) VNF Manager (see [2]) when registering in the NF Store. In this case, there would be a 'generic' VNF Manager, which is being designed, and a specific VNF Manager, brought by some of the on-boarded VNFs. Although the exact consequences of such a possibility are not yet fully understood, there are already some issues that made us delay such design in detail, such as:

- **Catalogue integrity**: it is clear that the VNF Catalogue integrity must be kept, in order not to negatively affect the other subsystems (namely the Marketplace, where the Service Provider can build its NSs, from the available VNFs). So, even when there is one or more specific VNF Managers, it is necessary to have to be a single catalogue from the point of view of the other subsystems that have to be designed, even if and when strategies such as catalogue federation are adopted;
- **Security**: it is not at all clear how a Service Provider may allow access to a third party (the FP), allocating and managing resources in its infrastructure, monitoring them, etc., without negatively affecting the SPs' overall expected Quality-of-Service. Even if we restrict every specific VNF Manager to manage VNF instances resulting from the VNF it came with, this issue may still stand;
- **Performance**: if the external VNF Manager is going to be located away from the VNF instances it is responsible for managing, significant infrastructure may need to be allocated, in order to support the performance needed for the VNF Manager to adequately manage these instances.

The architecture design addresses these issues by maintaining a high degree of flexibility: the microservices (see [5, 6] and Sub-section 3.1) based architecture will support easy replacement of individual microservices (e.g., the VNF Manager), configuring it adequately (e.g., making the VNF Catalogue the generic one within the specific VNF Manager), through an adequate level of security (e.g., by using the Gatekeeper module to provide external VNF Managers with the adequate authentication and authorisation credentials).

1.3. Reading this Report

This report documents this work in the following chapters:

- Section 2, describes the **Orchestrator Interactions**, where the interactions between the different sub-systems with the Orchestrator is described, on top of which the interfaces with those systems were defined;
- Section 3, describes the **Orchestrator Interfaces**, where the interfaces are defined (in order to make this chapter as easy to read as possible, some of the details about those interfaces have been put in the Annexes – see below);
- Section 4, with the **Conclusions**;
- Section 5, with the list of used **Acronyms**.
- Section 6, with the used **References**;

Annexes detailed supporting information for the deliverable. These annexes are:

- Annex A, the detailed Orchestrator Architecture;
- Annex B, a complement to Section 3, with further details about the Orchestrator API;
- Annex C, an example of a VNF Descriptor (still a work in progress);
- Annex D, an example of a NS Descriptor (still a work in progress);
- Annex E, an example of a HEAT Template (still a work in progress).

2. ORCHESTRATOR INTERACTIONS

This section is devoted to a description of the different components in the T-NOVA Architecture that interact with the Orchestrator (see Figure 1-1).

2.1. Interactions with the NF Store

This section describes the interactions between the Orchestrator and the Network Function (NF) Store.

2.1.1. Virtual Network Functions

The NF Store notifies the Orchestrator every time a new or updated VNF is available or is deleted in the NF Store, using the VNF id generated within the NF Store, after a successful on-boarding process.

Before this VNF is available for purchase and every time a new VNF is accepted in the NF Store or there's any change in the existing ones, it needs to be validated. Registering it with the Orchestrator does this validation, mostly by parsing and validating the VNF Descriptor (for an example of a VNF Descriptor please see Annex C). It then notifies the NF Store, which must then mark the VNF as "available" (or "unavailable" otherwise). Only then a Service Provider can purchase it with certain guarantees and use that VNF in the composition of new Network Services.

The VNF lifecycle within the Orchestrator may have several (and, ideally, configurable) options, of which the following have been chosen in the first implementation.

An update of a VNF that has already been registered in the Orchestrator generates a new version of that VNF (within the Orchestrator). This is to cover the possibility of having a VNF update while instances of the previous version of that same VNF are part of Network Services that are still running. Clean up processes must be designed and implemented, in order to keep the VNF Catalogue free of old and unused versions of VNFs.

Deleting a VNF will delete all versions of that VNF. It remains to be decided if deleting a VNF with instances still running will fail (the simplest version), mark that VNF as deleted (thus not allowing new NS instances to be launched with it) or immediately shutting down all NSs using instances of that VNF and actually deleting it.

The interface is bidirectional, in the sense that the NF store notifies the Orchestrator and then the Orchestrator writes on the NF Store.

2.1.2. Virtual Network Functions Images

Each VNF component (or, more precisely, every Virtual Deployment Unit – VDU – on which every VNF component is based on) must have an 'image' (a file) as a basis, which the VIM uses to instantiate that component in the infrastructure. These images are provided by the Function Providers (FP) and are stored in the NF Store, to be later used in the provisioning of the related VNF. The URI of those images must be part of the VNFD.

Whenever a Customer buys a new instance of a Network Service (NS) in the Marketplace, the Orchestrator knows from the NS Descriptor (for an example of a NS Descriptor please see Annex D) which VNFs are part of that NS and requests the VNF manager to provision each one of those VNFs and the WICM to provision the connection(s) needed. The VNF Manager asks the VIM to provision each VNF Component, passing the URL of the image.

2.2. Interactions with the Marketplace

This section describes the interactions between the Orchestrator and the Marketplace.

2.2.1. Network Services

The Marketplace must notify the Orchestrator whenever the Service Provider successfully creates, updates or deletes a Network Service (NS) in the Business Service Catalogue (BSC). Any change in that catalogue is communicated to the Orchestrator by sending the descriptor associated to that service (NSD). The NSD is syntactically validated in the Orchestrator, and the result is communicated back to the Marketplace, making it "available" (or "unavailable" otherwise). Only available NSs can be sold to Customers and instantiated.

Since the NS is first created in the Marketplace, its unique ID (:ns_id) is generated there and must be part of the request body made to the Orchestrator.

The Orchestrator's behaviour when a new version of an already registered NS appears will be configurable, but we have for now adopted the following: an update of a NS that has already been registered in the Orchestrator generates a new version of that NS (within the Orchestrator). This is to cover the possibility of having a NS update while instances of the previous version of that same NS are still running. Clean-up processes must be designed and implemented, in order to keep the NS Catalogue free of old and unused versions of NSs.

It is assumed that a NS is deleted only when there aren't any instances of it running.

2.2.2. Network Service Instantiation

The Marketplace notifies the Orchestrator to instantiate and deploy a registered NS.

Through the Marketplace, a Customer selects one of the available services, purchases it and decides to start using it. Through the Marketplace's Service Selection module, the specific parameters of the service are configured and passed to the Orchestrator, along with the id of the NS to instantiate. As a result of the instantiation process, the Orchestrator returns the IDs of the newly instantiated service and the VNFs that will be stored in the Accounting module (also part of the Marketplace) in order to track them.

A successful NS Instance creation returns the unique :nsi_id (note the 'i' for 'instance'), generated by the Orchestrator, which must be used by the Marketplace for future requests. A successfully created NS instance is left in the 'Started' state. It is assumed that only NS instances in the 'Stopped' state can be deleted.

Instantiating a NS implies instantiating the VNFs that are part of that NS. New VNF instances will be created based on the most recent VNF version registered (see Manage VNFs, above). Since the VNF instance is a resource created within the Orchestrator, this creation returns a VNF instance id that has to be used in further interactions with the Orchestrator with respect to a specific VNF instance.

Like in deleting a VNF (above), deleting a running VNF instance might either:

- Always succeed, assuming that the NS instance that it is part of is already stopped and ready to be deleted (the simplest option);
- fail if the NS Instance it is part of is still running;
- imply immediately stopping the NS instance that it is part of and then be deleted;
- be tagged as 'to-be-deleted' and wait for the NS that it is part of to be stopped and then be deleted

Possible states of a VNF instance are:

- Requested
- Started
- Stopped

The VNF instance is in the Requested state while it is being provisioned in the VIM. It is expected that the newly and successfully instantiated NS starts its lifecycle in the 'started' state.

2.2.3. Network Service Instance Change of State

When the Customer changes the state of one of his/her NS instance, using the Marketplace's dashboard, this change must be communicated to the Orchestrator, so that the corresponding action must be taken. A successful change in the state of a NS is reflected in the VNFs that are part of it and the connections between them. This change is communicated back to the Marketplace.

Possible states are:

- Requested
- Started
- Stopped

The NS instance is in the Requested state while the VNF instances that are part of it are being provisioned in the VIM, the connections between the (possibly) different PoPs are being established, etc. If the change of state is a request from the Customer to stop the NS, the given NS instance is stopped (as well as all the VNF instances that composed the service and all the connections between those VNFs and the WAN) and all the resources released. This change of state is than communicated back to the Marketplace, for the correct actions (e.g., bill the service) to take place.

When this change is due to operational reasons (i.e., on the Orchestrator side, like when the agreed Service Level Agreement – SLA – is broken), the change is communicated to the Marketplace, which then must take the appropriate actions.

2.2.4. Network Service Instance Change of Configuration Parameters

Whenever the Customer needs to change any parameter in the current configuration of a NS (e.g., the connection points, etc.), the Marketplace notifies the Orchestrator about this change in the configuration parameters for an already deployed NS instance. The NS is modified and the result is returned to the Orchestrator.

2.2.5. Network Service Instance Request for Metrics

The Orchestrator maintains a record of all the variables that were specified to be monitored in the NS/VNF Descriptors. The SLA module in the Marketplace queries the monitoring system in the Orchestrator to collect metrics to show that the SLA is being met. This interface is used also when a user needs statistics on how a given NS instance is performing. These metrics are used to generate statistics in case the caller is the Dashboard or to evaluate the SLA of the service and its functions for later billing purposes in case the caller is the SLA module.

An initial approach to the valid the states a NS instance may be in is as follows:

- Setup: the NS instance is still being set-up;
- Start: the NS instance is started;
- Stop: the NS instance is stopped;
- Terminate: the NS instance resources are still being released.

2.3. Interactions with the VIM

This section describes the interactions between the Orchestrator and the VIM.

2.3.1. Network Service Multitenancy

The T-NOVA Orchestrator will host multiple Network Service intances that will be requested by the Marketplace users (the Customers) and it should guarantee the proper level of isolation and security of each NS.

Each PoP has an OpenStack installation that includes the Keystone module. Keystone provides identity, authentication and authorization services for all other OpenStack modules, acting as a common authentication system across the cloud operating system.

The process of the instantiation of a NS starts with the Service Mapping microservice, which identifies the best PoPs for each VNF composing the NS. After that identification, the Orchestrator interacts with the PoP's Keystone to create a new Tenant and User that will be used by the VNFMs to request provisioning, scaling decommissioning operation to the VIM.

When the NS instance is terminated, the Orchestrator deletes users and tenants that are no longer be required.

2.3.2. Resource Allocation

For each data centre a Virtual Infrastructure Manager (VIM) guarantees the management and the allocation of the necessary virtual resources for the deployment of VNFs. These VNFs are composed of one or more virtual machines (VMs), which require the allocation of vCPUs, RAM and storage. The VIM is also responsible for creating and allocating the necessary network resources inside the Data centre.

VNF Provisioning is the Orchestrator microservice responsible for interacting with the VIM in order to request a deployment of a VNF, and HEAT is the OpenStack orchestration engine [7] responsible accepting requests and instantiating all the necessary infrastructure resources. HEAT provides a REST API [8] that the Orchestrator can use to request the creation of a VNF, the scaling of a VNF, information about a VNF and the termination of a VNF. Because HEAT only supports a specific kind of request in the form of a Heat Orchestration Template (HOT) to describe the required infrastructure, the Orchestrator must use the HOT Generator microservice to translate a VNFD to a HOT document. When HEAT receives this document it interacts with other OpenStack components to create the entire infrastructure in the correct order to launch the VNF. When infrastructure scaling is needed, the Orchestrator only needs to send the new HOT version to HEAT so that it updates the existing stack. For the termination of the VNF, the VNF Provisioning microservice sends a request to HEAT, which deletes all of the resources that have been allocated to the VNF.

The usage of the HEAT orchestration engine provides an advantage because it abstracts the interaction with all the other components of OpenStack however it does not have the concept of a NS.

2.3.2.1. Resource Allocation in Detail

From an infrastructure point of view, a generic VNF may be very complex, since it can be composed by several VNFCs, each of them implemented as a VM, connected through many different networks (both internal and external), not to mention additional storage/network resources (e.g. block storage volumes, load balancers, firewalls, etc.) and deployment policies (e.g. anti-affinity rules to guarantee HA deployment).

Therefore, if not properly governed, both first-time deployment and lifecycle management of a VNF can be cumbersome and subject to error. In order to overcome these problems:

1) The infrastructure characteristics of a VNF, e.g. its topology, required resources and deployment policies, are described by means of HEAT templates;
2) The required infrastructure for a given VNF instance is requested by the VNFM to the VIM by means of OpenStack HEAT REST APIs. VIM credentials will be made available to the VNFM by the Orchestrator.

HEAT is an orchestration engine, included in OpenStack, for provisioning and management of arbitrarily complex Infrastructure as a Service (IaaS) infrastructures. The infrastructure is described in a text file, called *template*. At provisioning time, the Heat engine interprets the template and deploys the corresponding infrastructure by orchestrating calls to the APIs of other OpenStack modules, as shown in Figure 2-1.

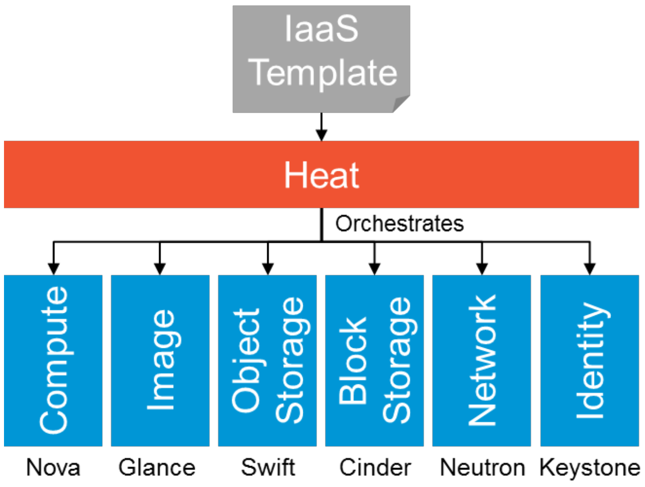


Figure 2-1: OpenStack modules.

An instance of a template is called a stack. Heat templates can be modelled in two different formats:

- **CFN** – Compliant with Amazon CloudFormation, written in JSON
- **HOT** – Heat Orchestration Template, written in YAML. Introduced in the Icehouse release, is the new de-facto standard and will replace the CFN format over time.

Independently from its format, a generic template is structured in three different sections, as shown in Figure 2-2.

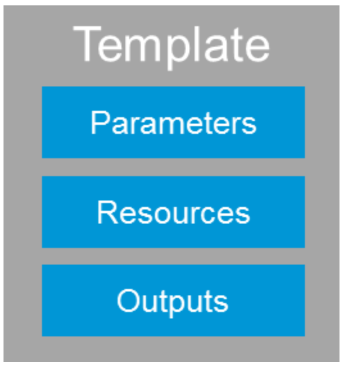
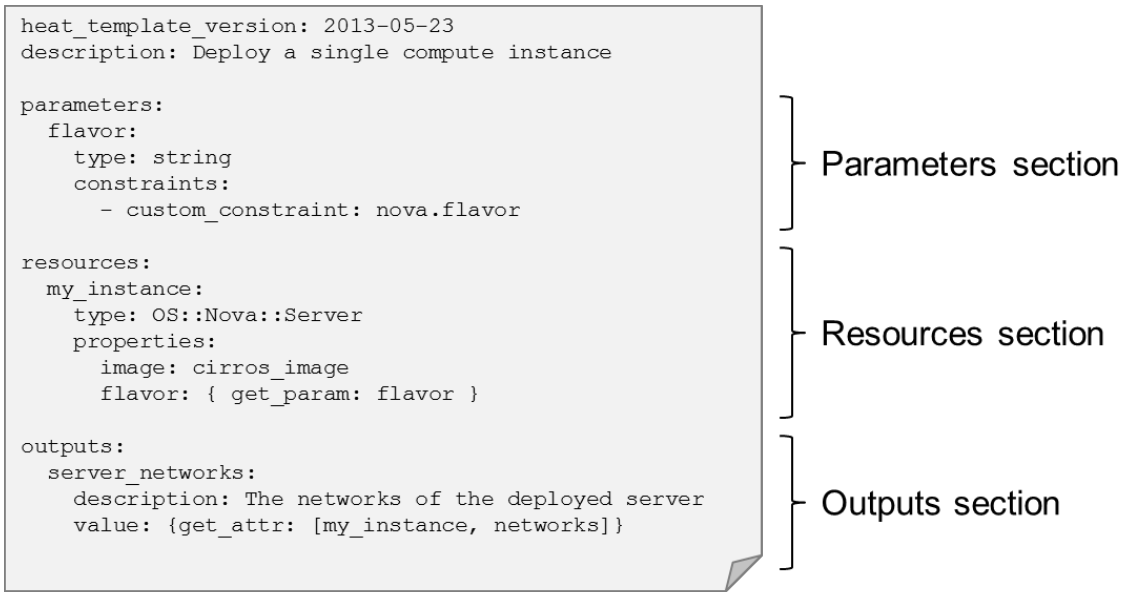


Figure 2-2: A HEAT template structure.

- **Parameters**: Declaration of input parameters that have to be provided when instantiating the template (optional)
- **Resources**: declaration of the resources (e.g. compute, network, storage, etc.) that make up the infrastructure template (mandatory)
- **Outputs**: declaration of output parameters (e.g. IP addresses of the VMs) available to the user once the template is instantiated (optional)

A very simple template, containing the definition of a single Virtual Machine is shown in Figure 2-3, highlighting also the three sections of the template:



```
heat_template_version: 2013-05-23
description: Deploy a single compute instance

parameters:
  flavor:
    type: string
    constraints:
      - custom_constraint: nova.flavor

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      image: cirros_image
      flavor: { get_param: flavor }

outputs:
  server_networks:
    description: The networks of the deployed server
    value: {get_attr: [my_instance, networks]}
```

Figure 2-3: Example of a HEAT file.

A more complete HEAT template is available in Annex Annex E.

The following paragraphs provide guidelines both for the definition and for the provisioning of VNF templates.

Template Format

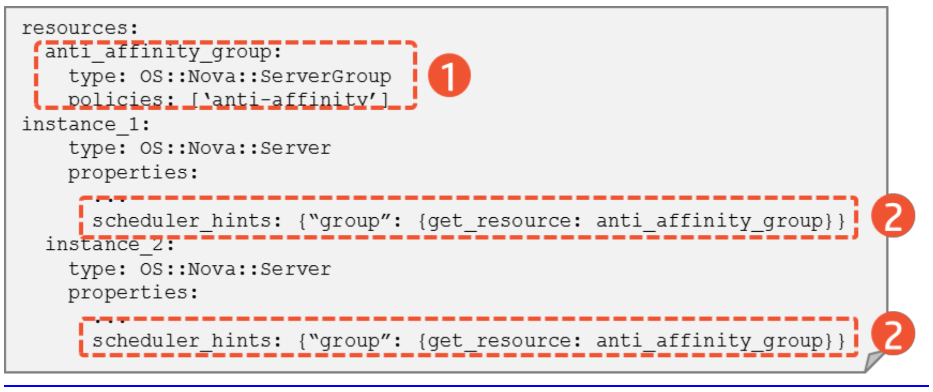
Application Templates shall be modelled using the HOT format [9], which is the new reference standard since the OpenStack Icehouse release [10], replacing the AWS-CloudFormation format. Although AWS-CloudFormation is (and will continue to be) supported for binary compatibility with the Amazon Public Cloud, the OpenStack team does not plan to invest on it, meaning that new and innovative features will be developed only for HOT.

The complete HOT reference guide is available on the OpenStack public website [9].

High Availability deployment of VNFs

High Availability (HA) deployment of VNFs can be enforced directly in the HOT template by recurring to anti-affinity rules. Anti-affinity rules, supported by OpenStack through nova-scheduler's ServerGroupAntiAffinityFilter, allow specification VM groups that should be placed on different physical hosts, so that, if a host fails and a VM is lost, the service will still be provided by VMs running on other hosts.

The usage of anti-affinity rules in HOT templates is quite simple, as shown in the following example:



```
resources:
  anti_affinity_group:
    type: OS::Nova::ServerGroup
    policies: ['anti-affinity']
instance_1:
    type: OS::Nova::Server
    properties:
      ...
      scheduler_hints: {"group": {get_resource: anti_affinity_group}}
  instance_2:
    type: OS::Nova::Server
    properties:
      ...
      scheduler_hints: {"group": {get_resource: anti_affinity_group}}
```

Figure 2-4: Sections in a HOT template.

In Figure 2-4:

1. Defines a new `Nova::ServerGroup`, named `anti_affinity_group`, with anti-affinity policy;
2. Place VMs `instance_1` and `instance_2` in the `anti_affinity_group` defined in the previous step. After provisioning, the two VMs are guaranteed to be running on two different compute nodes.

VNF Provisioning

Provisioning of the infrastructure needed by a VNF is done by the VNFM by invoking OpenStack Heat's "Create Stack" REST API on the VIM [7].

In order to simplify the integration between the VNFM and the VIM, Application Vendors may leverage existing SDKs for different programming languages, including Java, node.js, .NET, Ruby, PHP and Python, which are available and well documented on the OpenStack public website [11].

A very simple example, using the OpenStack command line interface (CLI), is shown hereafter:

```
$ heat stack-create MyVNF --template_url http://address:port/MyVNF.yaml
```

VNF Scale-in/out

Even if OpenStack HEAT has built-in support for auto-scaling [12], this is currently quite limited, since it can only be triggered through Infrastructure KPIs measured by Ceilometer (e.g. CPU consumption), but not through Application/Service KPIs (e.g. # of concurrent calls).

Since Application/Service KPIs are not taken into consideration, an automated scaling process can potentially lead to undesired and/or unnecessary scaling operations.

Therefore, the recommended scaling guideline is as follows:

1) A HEAT Template is defined for every flavour of the VNF (e.g. Small, Medium, Large)
2) The VNFM instantiates the VNF by requesting a "Stack-create" of one of the templates (e.g. Medium):

```
$ heat stack-create mystack --template_url http://address:port/MyVNF_Medium.yaml
```

3) When scale-in/out is required, the VNFM requests the VIM to update an existing stack, by invoking the "Stack-update" API and passing the corresponding template (e.g. Large), as shown in the following example, which uses the OpenStack CLI:

```
$ heat stack-update mystack --template_url http://address:port/MyVNF_Large.yaml
```

Template Parameterisation

VNF HEAT templates shall be generic and possible to instantiate in any Data centre. Thus, any environment-specific setting (e.g. networks) should not be hard-wired in the HEAT template, but rather declared as a parameter in the parameters section of the template.

In order to simplify the deployment of a VNF template in a specific Data centre, the VNFM may leverage Heat's Environment support, which makes it possible to provide default values for template parameters at instantiation time, by specifying them in separate text files, called environments [12], which can then by passed to the "Create Stack" request by means of the "environment" parameter. Here follows a simple example which makes use of the OpenStack CLI:

```
$ heat stack-create mystack --template_url http://address:port/MyVNF_Large.yaml\
> --environment http://address:port/DC1_env.yaml
```

2.3.2.2. VNF images

When we mention 'VNF images' we are really talking about VNF Components images, which are described by VDUs.

There are three possible approaches for the NF Store, the Orchestrator and the VIM to deal with these images:

1. The Orchestrator copies from the NF Store all images of all the components of a VNF, when that VNF is registered in the Orchestrator or is updated:
    a. Pros: the image, a file that maybe significant in size, is already kept on the Orchestrator's side, prior to any instantiation of the VNF, thus saving time for downloading it from the NF Store when an instance is requested;
    b. Cons: there must be sufficient storage space for all the images provided by the NF Store on the Orchestrator's side, this space is wasted (since the same images are also kept in the NF Store);
2. The Orchestrator does not copy any image from the NF Store, only downloads it from the NF Store when the first instance of that VNF is requested and passes it to the VIM whenever an instance of that VNF is requested:
    a. Pros: there is no storage space wasted in the Orchestrator with VNFs that are never instantiated;
    b. Cons: the VNF instantiation process will take longer, due to the download of the image;
3. The Orchestrator does not copy any image from the NF Store and passes the image URL that is part of the VNFD to the VIM, which executes the download:
    a. Pros: there is no storage space wasted in the Orchestrator with VNFs that are never instantiated;
    b. Cons: the VNF instantiation process will take longer, due to the download of the image;

Option #3 above is essentially a specialisation of #2. For #1, it is the VNF Catalogue component's responsibility to download and validate the VNF (components) images.

In option #2, it is the VNF Provisioning’s responsibility to download and validate those images. Option #3 delegates all responsibility to the VIM. Option #1 can also include loading the VIM with the images, but this will only increase the waste in storage space, making the VIM to store images of VNFs, even if they have never been instantiated. When multiple Points-of-Presence (PoPs) are considered, this waste is even more emphasized, with VNFC images copied to every PoP. We would also have to take care of versioning and maintenance of out-dated images.

It is also notable that instantiation time may not be the parameter to minimise here: if the Network Service to be instantiated includes a connection to the network at a specific point for the customer (see section on the WICM), this will typically take longer, thus making the time needed for downloading the VNF images irrelevant.

Our decision on this was option #3: delegate to the VIM the responsibility of fetching the VNF components images and using them on the instantiation.

2.3.3. Infrastructure Repository

The infrastructure resource repository delivers a single source of information on the infrastructure landscape of the T-NOVA system. The repository provides an interface to dependent components within the T-NOVA Orchestration layer through a middleware layer. The layer provides the Northbound REST API to the functional components of the T-NOVA Orchestration layer such as the Orchestrator manager, resource mapping module etc.

The middleware layer provides a common interface to all the PoP level databases within the T-NOVA system as shown in Figure 2-5. The PoP level databases store infrastructure information at each NFVI-PoP from the following sources:

1. Enhanced Platform Awareness Agents running on compute hosts
2. OpenStack service databases
3. OpenDaylight Controller

From the perspective of a component using the interface the location of the data and the underlying complexity in forming the query response is abstracted. In order to support common access to all PoPs, the relevant service endpoints needed are stored within the middleware layer in a database.

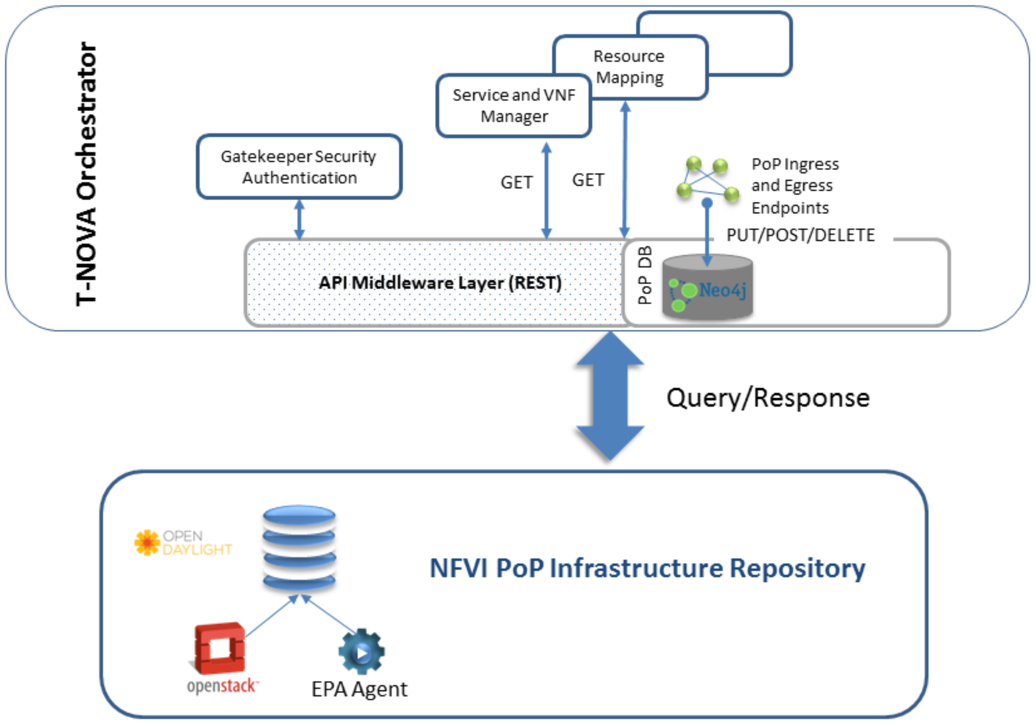


Figure 2-5: The Infrastructure Repository middleware layer design.

The primary function of the middleware APIs is to support retrieval of information from the repository databases located at each NFVI-PoP. The interface does not support other actions such as inserting, updating or deleting information in the NFVI-PoP level databases. To be compliant with the design decisions of Task 3-1 a REST type approach to the design of the interfaces was adopted. However additional requirements in the interface design were also considered. The middleware API also provides an agnostic repository implementation interface to the dependent Orchestrator components. The design of the interfaces therefore adopted an OCCI [13] approach to fulfil this requirement.

2.3.4. Monitoring Parameters

The Orchestrator bridges the gap between the NS instance-level monitoring parameters that are part of the Service Level Agreement (SLA) with the Customer and the monitoring data the Monitoring Framework (see [14]) makes available to the Orchestrator.

2.3.4.1. Network Service Monitoring Parameters

The NS Descriptor (see Annex D) has a section named `assurance_parameters` that holds the definition of the SLAs a given NS should follow (see also [1]).

```
…
"assurance_parameters": [
{
   "name": "availability",
   "value": "GT(0.99)",
   "formula": "min(vnfs[1].availability, vnfs[2].availability)",
   "violation": [
      {
         "breaches_count": "5",
                "interval": "120",
            }
        ]
    },
    …
```

Figure 2-6: Extract of an NSD, emphasizing the SLA parameters. For full NSD example, please see Annex D.

These assurance parameters each having:

- a **name**: in this example, availability;
- a **value**: a comparison with a numerical constant, in this case, greater than 0.99;
- a **formula**: to calculate the NS parameter with the values provided by the VNF instances parameters, as indicated in the dependencies section of the NSD – not shown here;
- A **violation** definition: in this example, the availability of the NS instance maybe less than 99% for 5 times within 120 seconds.

From this definition, whenever a new instance of this NS is asked purchased by a Customer, the Orchestrator (the NS Monitoring microservice) builds an SLA and asks the VNF Manager subscribe to the available parameters for each one of the VNF instances that are part of this NS instance. The VNF Manager gathers the name of the instances from the VNF Provisioning service and asks the VNF Monitoring microservice to subscribe to the required monitoring parameters published by the VIM's Monitoring Framework.

The sequence diagram in Figure 2-7 shows these interactions.

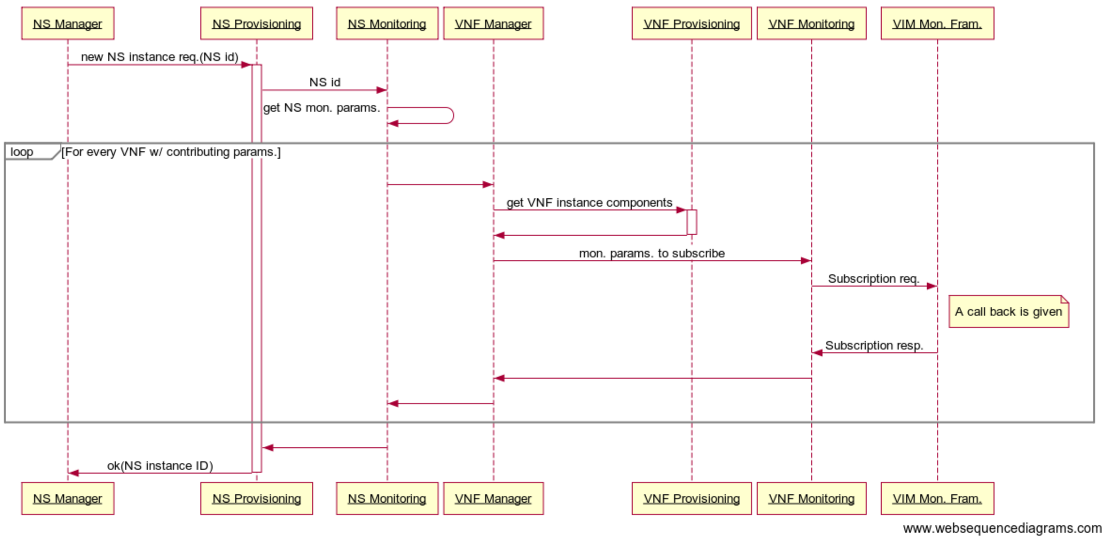


Figure 2-7: Sequence diagram of monitoring parameter subscription.

The subscription includes a call-back URL that the VIM's Monitoring Framework will use to report the subscribed monitoring parameters' readings (see below).

2.3.4.2. Subscribing Monitoring Parameters

Due to the high number of monitoring parameters available (and the predictably high volume of those parameters' published to the Orchestrator), we have implemented a subscription mechanism, which allows us to only collect, process and make available to the Orchestrator those parameters that are part of the NS SLA.

The subscribed monitoring parameters can then be sent to the Orchestrator, to be processed and compared with the agreed SLA. An example payload is:

```
{
   "SubRefId": 1cfbe83c,
    "metric": "27ad39af-0267-4f81-bdc6-deda0d64c9ac.vnf.totalflows.avg.600",
    "interval": 600,
    "CallbackURL": http://apis.t-nova.eu/orchestrator/readings/27ad39af-0267-4f81-
bdc6-deda0d64c9ac.vnf.totalflows.avg.600/
}
```

For further details on this mechanism please refer to [14].

2.3.4.3. Receiving Network Service Monitoring Parameters Readings

Whenever the VIM's Monitoring Framework has a monitoring parameter reading to send to the Orchestrator, it POSTs it and the whole process described above for subscriptions has to be repeated but in reverse order:

1. The VNF instance is found from the name of the parameter;
2. The NS instance to which the VNF instance belongs to is found in the VNF Instances Repository;
3. The SLA is found from the NS instance;
4. The NS instance level monitoring parameter is calculated from the SLA;
5. If there's a breach, the NS Manager must be notified, in order to take the appropriate action.

The sequence diagram in the next figure illustrates this flow.

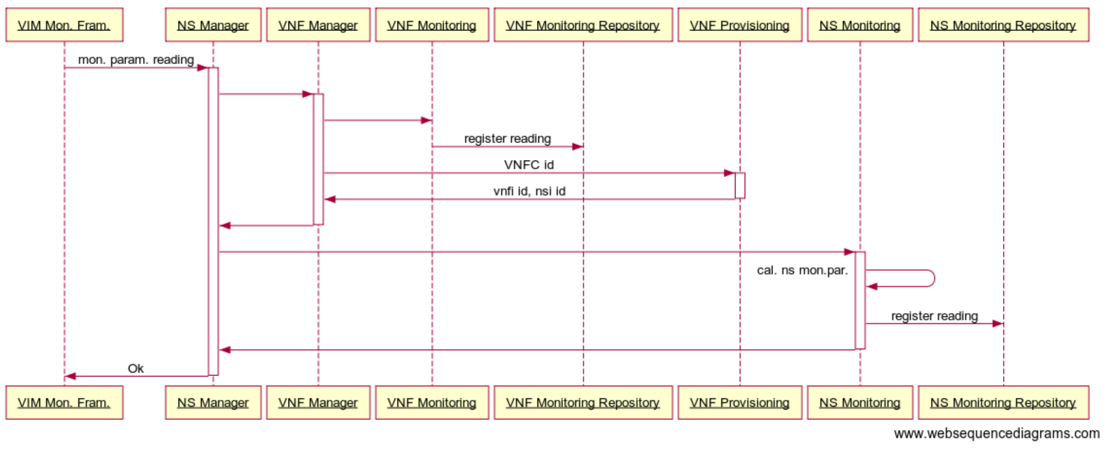


Figure 2-8: Sequence diagram of the reading of a monitoring parameter.

The sequence of actions illustrated by this diagram can later be optimised, since the reading already carries a lot of information in the monitoring parameter name (see above).

2.3.4.4. Missing Values

In the real world, VNF (component) monitoring parameters may be missing for a more or less significant interval of time, which will clearly impact the calculation of monitoring parameters at the NS level. There is a vast literature dealing with the subject.

In the T-NOVA Orchestrator we are assuming as valid the last value obtained from every monitoring parameter, knowing that this will somehow distort the NS level monitoring parameters readings but at a manageable level.

2.3.4.5. Possible Optimisations

HTTP's (or TCP's, to be precise) relatively high latency [3] has led some providers of monitoring tools to completely avoid the usage of this protocol, in favour of protocols like UDP [4].

We are aware of this fact but, in the name of speed of implementation, we have kept the interfaces in the monitoring 'column' of the architecture over HTTP. Nevertheless, if needed be, and due to the highly modular architecture that has been implemented, HTTP can be quickly replaced by UDP for example.

2.3.4.6. Big Data Approach

As previously stated in [1], a system approach that might be applicable in building the monitoring column that of Real-Time Stream Processing systems [15].

Figure 2-8, above, shows how complex the processing of monitoring parameters in the Orchestrator can be. A key focus in this task has been clearly understanding the needs of T-NOVA and implementing a flexible and modular architecture. We are now better positioned to evaluate such kind of systems and adopt one, in case the project feels so.

2.4. Interactions with the WICM

This section describes the interactions between the Orchestrator and the WAN Infrastructure Connection Management (WICM) sub-system of the T-NOVA architecture.

Whenever a Customer purchases a Network Service using the Marketplace, and that service needs to be connected to a specific point on the WAN (namely, where the Customer already has other services attached to), the Orchestrator has to allocate that connection by contacting the WICM. A successful request for this connectivity resource returns the VLAN ID to be used in configuration of the associated VNF instances.

Because the allocation of necessary the resources required for a requested NS Instance may take some time before they are available on the different PoPs, the Orchestrator must notify the WICM when this allocation ends successfully (see the PUT method on the API table in the Annexes). These interactions are shown in Figure 2-9.

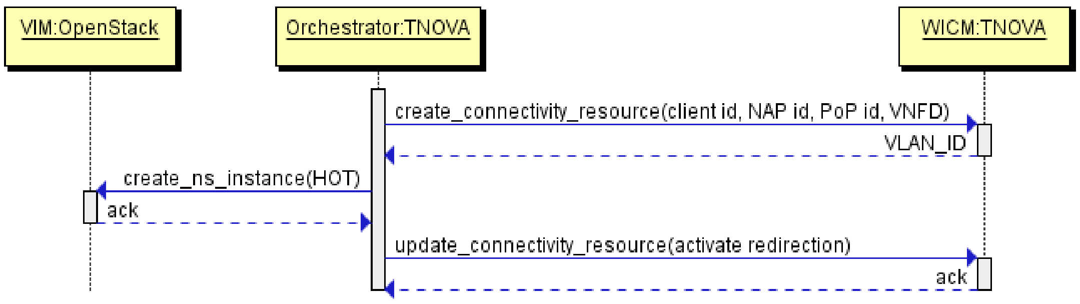

VIM:OpenStack
Orchestrator:TNOVA
WICM:TNOVA
create_connectivity_resource(client id, NAP id, PoP id, VNFD)
VLAN_ID
create_ns_instance(HOT)
ack
update_connectivity_resource(activate redirection)
ack

Figure 2-9: Sequence diagram illustrating the interactions between the T-NOVA Orchestrator, the VIM and the WICM.

2.5. Interactions with the VNFs

This section describes the interactions between the Orchestrator and the VNFs, considering the VNF Manager as an **internal** component, common to all VNFs (for further details please see [2]).

The manner in which the VNFM interacts with a VNF is shown in Figure 2-10 (for further details please see [16]).

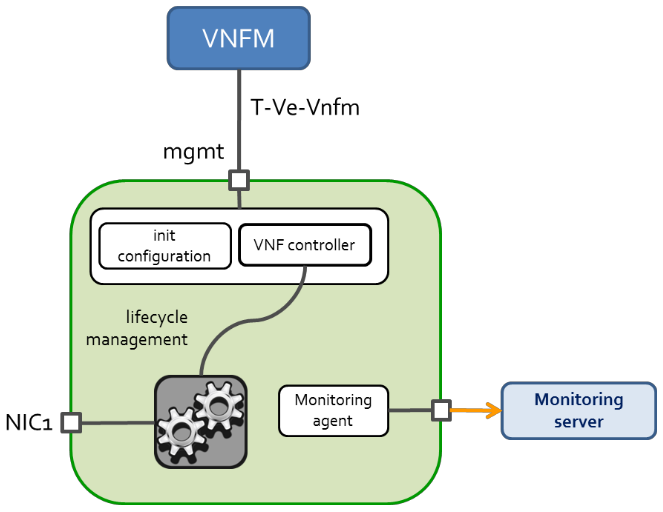


Figure 2-10: The VNFM and its VNFs.

In an effort to standardise the connection between the VNFM and the VNFs, a middleware API (mAPI) has been developed that is described in the next section.

2.5.1. The mAPI

During lifecycle of a VNF the VNFM will need to interact with the VNF to carryout configuration or reconfiguration actions. There are myriad of protocols and technologies available to perform configuration on network functions. To avoid selection of specific option a single one (especially when ETSI is still specifying the T-Ve-Vnfm reference point) it was decided to place a component between the VNFM and the VNFs. This component, the middleware API (mAPI), acts as a mediator between the entities and abstracts the T-Ve-Vnfm reference point from the VNFM. With this in mind, the mAPI exposes to the VNFM a single and common interface to perform configuration procedures on VNFs while supporting multiple technologies to interact with VNFs. For further details please see [17].

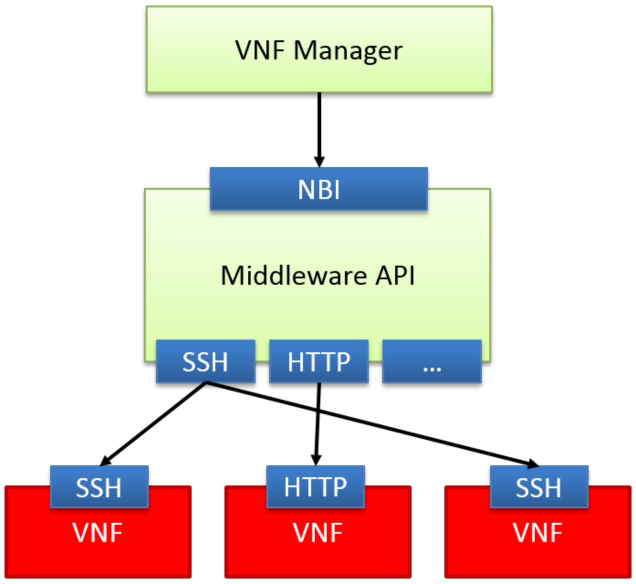


Figure 2-11- T-NOVA partial architecture with only the VNFM, mAPI and VNFs.

Figure 2-11 shows the partial architecture of the T-NOVA platform with only the VNFM, mAPI and VNFs. This figure illustrates the role of the mAPI as mediator in the communication between the VNFM and the VNFs. Although the mAPI northbound interface (NBI) could be seen as an MANO external interface (from the VNFM point-of-view), the real external interface it is only realised in the mAPI southbound plugins.

With this implementation, the platform can easily be extended to support new technologies without needing to change a complex component such as the VNFM.

2.5.2. Configuration Procedures (VNFM point of view)

From the VNFM point of view, the connection to VNFs is realised in a single and common interface realised in the NBI of the mAPI. This interface is related to the VNF lifecycle. It exposes the following operations:

1. Register VNF: this operation allows the on boarding of the VNFD in the mAPI. Afterwards the mAPI will use the lifecycle events description in the VNFD to deploy the configuration resource[1].
2. initial_configuration: this operation allows the VNFM to provide the initial VNF configuration.
3. start: this operation is used to ask the VNF to start providing its services.
4. stop: this operation is used to ask the VNF to stop providing its services. Two types of stop are supported: hard and soft. A soft stop ensures a graceful cessation of the VNF. In particular, it asks to complete any in-progress operation before stopping the VNF. The hard stop asks to immediately cease

[1] In fact there is not an explicit operation for this: the information about the VNF manager is derived from the VNFD.

all the operations of the VNF. The VNF remains active after the stop operation is completed, i.e. the MV is up and running; only the service provisioning is interrupted.

5. scale_out, scale_in: these operations are used to ask the VNF to perform scaling out/in procedures. Then the VNF will reconfigure its internal resources in order to accommodate new resources or release them.
6. terminate: this operation is used by VNFM to signal the termination/destruction of the given VNF. Moreover, the configuration resource is destroyed and can no longer be used to manage the VNF. To pause/stop a VNF the *start/stop* operations are used and should not be mistaken with the *terminate VNF* operation.

Figure 2-12 shows a sequence diagram of an example interaction between the VNFM and the mAPI, when updating the configuration of a VNF.

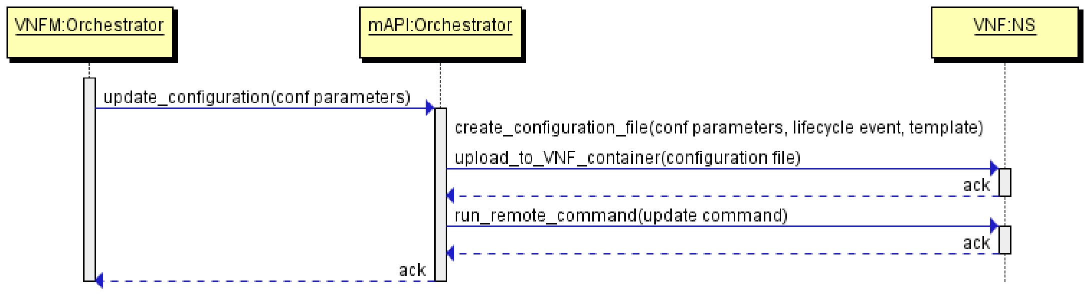

VNFM:Orchestrator
mAPI:Orchestrator
VNF:NS
update_configuration(conf parameters)
create_configuration_file(conf parameters, lifecycle event, template)
upload_to_VNF_container(configuration file)
ack
run_remote_command(update command)
ack
ack

Figure 2-12: Example of an interaction between the VNFM and the mAPI for updating a VNF configuration.

2.5.3. External Interface Specification

The implementation and specification of the plugins available in the mAPI is still a work in progress and will be presented in [17] (which will be released in M27 of the T-NOVA project).

2.6. Interactions with the VNFM

This section describes the interactions between the Orchestrator and the VNFM, when it is considered as an **external** component, specific to a VNF.

The VNFM, as its name indicates, is the element responsible of coordinating the different microservices implemented for the VNFs (e.g. VNF monitoring, or VNF provisioning). Therefore, this component will act within the orchestrator as the point of entry of the different requests, which should be forwarded to the corresponding microservice. Basically, this component within the orchestrator will require an interface that allows it to proxy the different VNF lifecycle related requests.

Figure 2-13, below shows the different interactions of the VNFM within the orchestrator, considering it is a generic module deployed for all the different VNFs (no specific VNF Manager comes with the VNFD in this case. Refer to next section for such case).

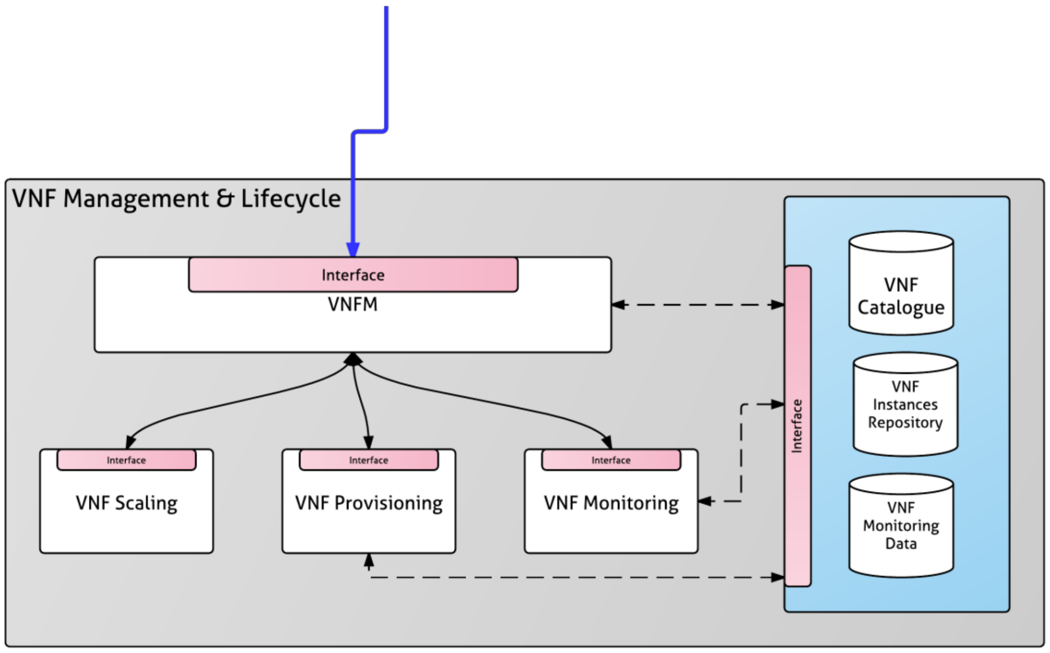


Figure 2-13: interactions of the VNFM within the orchestrator.

As Figure 2-13 shows, the VNFM interacts with the VNF-Catalogue in order to on-board newly available VNFs, as well as to retrieve information related to existing VNF in the system, or to manage versioning of the VNFDs. It also interacts with the VNF-Provisioning in order to instantiate the different VNFs available in the Catalogue; at the same time it controls the status of those VNFs, since it is in charge of managing the different deployment, update, or destroy requests of VNF instances.

The VNFM also requires an interface with the VNF Monitoring module, which supports both reading and writing of data in the corresponding VNF Monitoring repository. The data is received from the VIM.

The VNFM will interact with the scaling microservice, which responsibility for scaling procedures within the VNF lifecycle.

The VNFM must allow the configuration of at least some of the services supporting it. This configuration is done for a limited set of pre-defined parameters. These configuration ids are one of (e.g.):

- vnf-catalogue
- vnf-provisioning
- vnf-monitoring
- vnf-scaling

These configuration ids are the 'keys' of which values are the endpoints of each one of those services.

2.7. Interactions with other Modules/Microservices

The orchestrator is a complex, distributed system, which basically performs lifecycle management of network services composed of virtual network functions. However, in order to deal with the complexity of the internal architecture, and enable maximum performance of such a system, there is a set of additional microservices that need to be utilised. Those microservices provide the system with required functionalities for its internal mechanisms. There are three major modules considered as additional functionalities:

- Gatekeeper, which provides security functionalities both with the external orchestrator entities, and the internal microservice architecture.
- Management User Interface, which enables the human administrator to interact with the different orchestrator functionalities.
- Internal monitoring and logging, which is utilised as an internal control mechanism to detect errors or anomalies in the microservices.

These modules are detailed next.

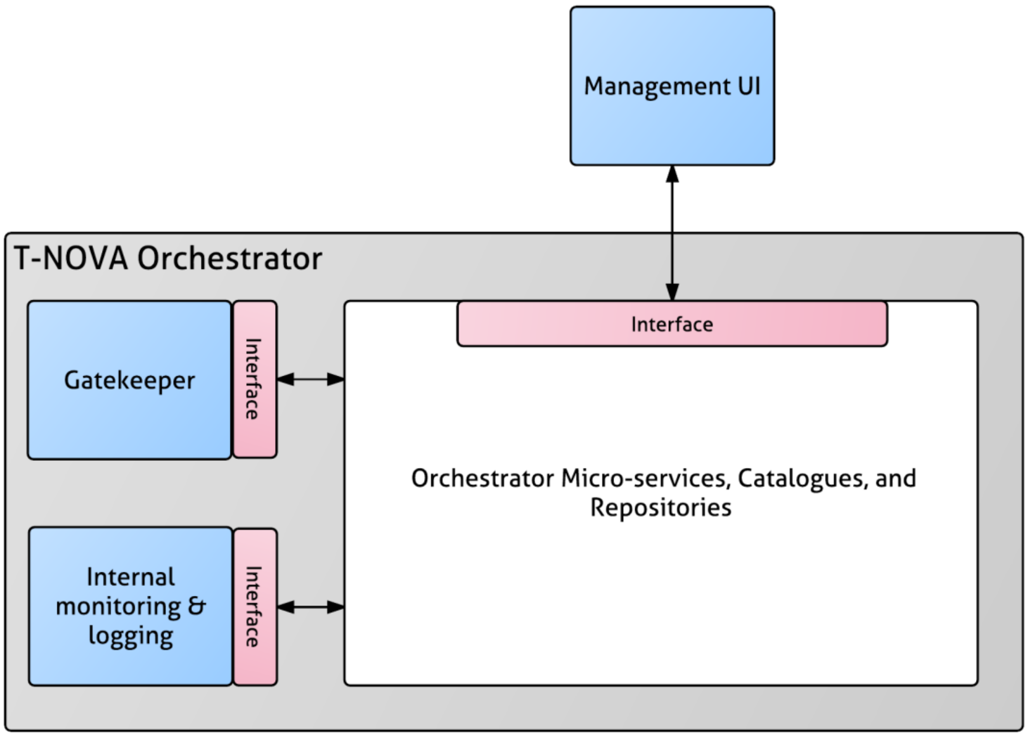

Management UI
T-NOVA Orchestrator
Gatekeeper
Interface
Interface
Orchestrator Micro-services, Catalogues, and Repositories
Internal monitoring & logging
Interface

Figure 2-14: Internal modules of the Orchestrator.

2.7.1. Authentication and Authorisation: Gatekeeper

The Gatekeeper is a simple microservice providing authentication and authorisation service to any other micro (or monolithic) service(s). The module provides simple but complete RESTful interface that supports authentication and ACL based authorisation flow between users, orchestrator-services and Gatekeeper. This functionality is achieved through an efficient token management system employed within the Gatekeeper service.

2.7.1.1. Interfaces by role

The Gatekeeper supports interactions via HTTP REST APIs. The APIs enable users, services, and administrators to interact with it to fulfil their need for a proper authentication and authorisation process. The interfaces it exposes can be categorized into these three categories:

- used by users (or their proxies);
- used by registered services for authorisation needs;
- used by administrators to manage access to users and services to this Gatekeeper service.

Interfaces for these roles are detailed next.

Administrator's Gatekeeper interfaces

There are certain capabilities that can be accessed only by system administrators: these are managing user accounts (create, update and delete), and registration of services. For accessing these capabilities, the administrators must request first a token, and using such a token, API requests can be performed. A summary of these APIs is provided in Table 2-1.

Table 2-1: Gatekeeper administrator's interface.

Endpoint	Header Fields	Method	Description
`/admin/user/`	X-Auth-Token	GET, POST	List of users, registration of new users
`/admin/user/:user_id`	X-Auth-Token	GET, PUT, DELETE	Read, Update details of user account, delete user account
`/admin/service/`	X-Auth-Token	GET, POST	List services, register a new service
`/admin/service/:service_id`	X-Auth-Token	GET, PUT, DELETE	Read, update service details, delete service

Appropriate HTTP response-codes are returned along with descriptive messages and other parameters as part of API call response.

Users' Gatekeeper interfaces

Any user, admin or non-admin can user Gatekeeper to authenticate and generate token(s) to be presented to the orchestrator service(s) for authorisation. The tokens generated have the same capabilities as granted to the user. These permissions are maintained in a capability list internally managed by Gatekeeper, which can only be modified by system administrators. The tokens that are generated have a default expiry set to 6 hours, and upon expiry, the user, or a process acting a proxy for the user must generate a new token to continue using the orchestrator services. The Gatekeeper APIs accessible to all users are shown below.

Table 2-2: Gatekeeper's users interface.

Endpoint	Header Fields	Method	Description
`/auth/:user_id`	`X-Auth-Password`	`GET`	Authentication, no token is generated
`/token/`	`X-Auth-Uid, X-Auth-Password`	`POST`	Authenticates user and generates a token
`/token/validate/:token_uuid`	`X-Auth-Uid`	`GET`	Validating the token against given :user_id

Services' Gatekeeper interfaces

When a system administrator registers a service with Gatekeeper, a unique service-key is generated for that service. When service(s) makes a call for token authorisation to Gatekeeper, they are required to provide their *service-key* as part of the request.

Table 2-3: Gatekeeper's service interface.

Endpoint	Header Fields	Method	Description
`/token/validate/:token_uuid`	`X-Auth-Service-Key`	`GET`	Validating the user token against given :service_key

If the user has permissions to access the *service*, the response will authorise the service to proceed against the token it has, otherwise the authorisation request will be denied.

2.7.1.2. Workflows

The design of Gatekeeper allows several workflows to be implemented, but the most common workflows are shown here.

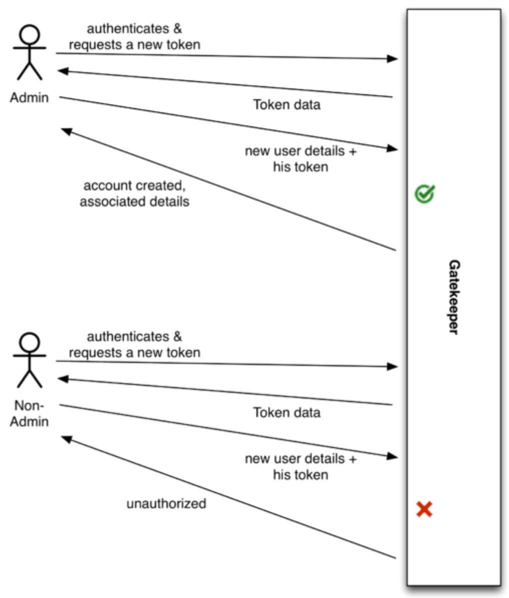

Admin
authenticates & requests a new token
Token data
new user details + his token
account created, associated details
Non-Admin
authenticates & requests a new token
Token data
new user details + his token
unauthorized
Gatekeeper

Figure 2-15: Creating a new user.

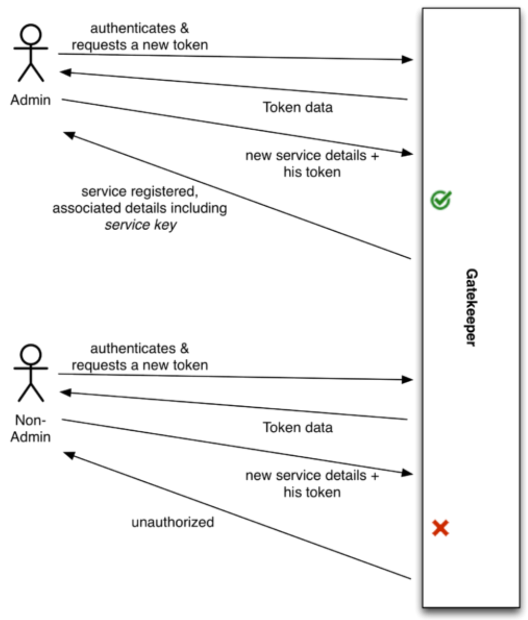

Admin
authenticates & requests a new token
Token data
new service details + his token
service registered, associated details including *service key*
Non-Admin
authenticates & requests a new token
Token data
new service details + his token
unauthorized
Gatekeeper

Figure 2-16: Registering a new service.

The process shown in Figure 2-16 shows the sequence of API calls to be made for registering a new user account, or a new service with Gatekeeper by an admin. Once the services are registered, and users with appropriate capabilities are registered with the system, then a typical authentication, authorisation workflow from users request to cooperation and coordination among orchestrator microservices in order to fulfil the user's request via Gatekeeper will typically resemble the sequence steps shown below.

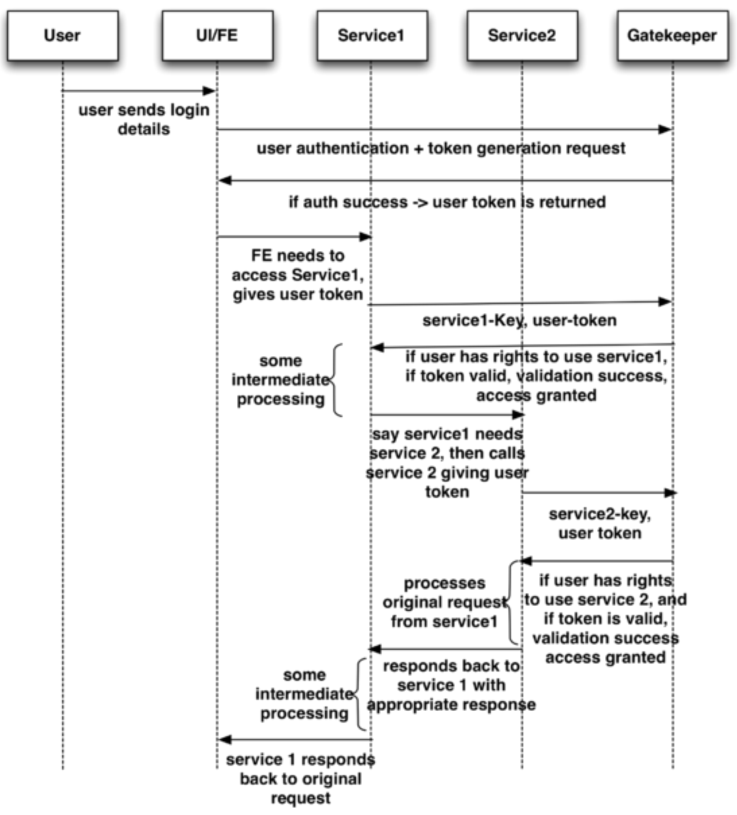

User
UI/FE
Service1
Service2
Gatekeeper
user sends login details
user authentication + token generation request
if auth success -> user token is returned
FE needs to access Service1, gives user token
service1-Key, user-token
some intermediate processing
if user has rights to use service1, if token valid, validation success, access granted
say service1 needs service 2, then calls service 2 giving user token
service2-key, user token
processes original request from service1
if user has rights to use service 2, and if token is valid, validation success access granted
responds back to service 1 with appropriate response
some intermediate processing
service 1 responds back to original request

Figure 2-17: Sequence diagram of a user request.

The above sequence diagram describes this simple use-case: user logs in to a dashboard (UI), makes a request to service1. Service1 needs to call service2 to satisfy the request. The user has been allowed access to service1 and service2 from the administrator during account creation step, and stored as capability list for this user by Gatekeeper.

2.7.1.3. Roadmap

In order to make this simple authentication/authorisation service more secure, it is planned to implement SSL encrypted communication links between Gatekeeper and various actors using it. It is important to specify that Gatekeeper already takes several steps to minimise attack possibilities, for instance no user passwords are stored, but the cryptographically secure hashes are stored. To further strengthen the interfaces, it is planned to allow access from actors with approved X.509 certificates only. The certificates would be generated by consortium controlled CA and certificates issued by any other CA will be rejected by default.

2.7.2. Management UI

The Web-based User Interface will be used as the management and configuration point of entry for the orchestrator. Will enable two major actions: (i) to visualise all the information stored in the different catalogues and repositories in a centralised manner, as well as to monitor all orchestrator system metrics; and (ii) to configure specific options for the orchestrator software system itself. But the major function of the UI is consuming the different APIs of the Manager in order to read all the stored data.

2.7.3. Internal Monitoring and Logging

Considering the multi-element nature of the orchestrator (i.e. implemented by means of different microservices), it becomes a requirement to define an internal mechanism in order to look for problems and/or events in each one of the modules. The challenge of this internal monitoring and logging module is to aggregate together the different logs of each component, including aggregation of the information, and the possibility to retrieve such information. A clear example of tools to be used for such microservice is LogStash [18], together with ElasticSearch [19] or Kibana [20].

Additionally, this component is also in charge of controlling the status (registered, up, down, etc.) of each microservice, as well as the coherence of the interfaces running.

3. ORCHESTRATOR INTERFACES

This section focuses on the Orchestrator's External interfaces. As stated in [1], the Orchestrator interfaces use the following technologies:

- HTTP
- REST
- JSON

A microservices-based architecture (see [1]), has be adopted whose advantages are briefly addressed in the next section. Using as a basis the previous section, this section contains the specification of the orchestrator interfaces as they are in the NS Manager and VNF Manager, as the entry points of the orchestrator.

3.1. Why a Microservices-based architecture?

As first described in [1], we have adopted a microservices way of organising the several modules of the Orchestrator.

Microservices were first used by Fowler and Lewis [5] as a way to decompose monolithic applications. A microservice is therefore responsible for a single part of the whole functionality, using the Single Responsibility Principle [21]. It has an independent lifecycle (deployment, update, replace and scale), thus making it easier to evolve than its equivalent monolith.

The main reason for adopting this form of organisation was an independent scaling advantage. Some of the Orchestrator's interfaces and supporting microservices may be subject to a heavy load while others will just have to handle the basic throughput of requests a normal web application of this kind is subject to. We can speculate which of the services might need to scale or not, and design them accordingly from scratch, but approach may waste precious resources, needed to address other important features. Therefore keep the initial design is kept simple and uniform, but ready to scale later.

The main disadvantage of such an architectural organisation is the plethora of microservices that must be tracked. However the usual mechanisms that provide mitigation the operation of such kind of systems, such as monitoring and logging are being implemented. This subject will be elaborated on in the remaining deliverables of the Work Package.

3.2. External Orchestrator Interfaces

This sub-section details the external interfaces of the Orchestrator itself. The internal structure of the Orchestrator is completely hidden from the outside: 'client' systems just have one point of access to the Orchestrator. It is the Network Services' Manager's (NS Manager) role to redirect the requests to the intended microservice that has the responsibility to answer that request.

All the Orchestrator's interfaces have its URL started by:

```
http://apis.t-nova.eu/orchestrator
```

All the calls are then routed to the correct internal component of the architecture, as explained in [1].

Most of the REST APIs mentioned below follow this pattern, unless stated. For example, a call from the Orchestrator to one of the Marketplace's REST API is written in the form:

```
/marketplace/<api>
```

3.2.1. Interfaces with the NF Store

The interface between the Orchestrator and the NF Store is for managing VNFs.

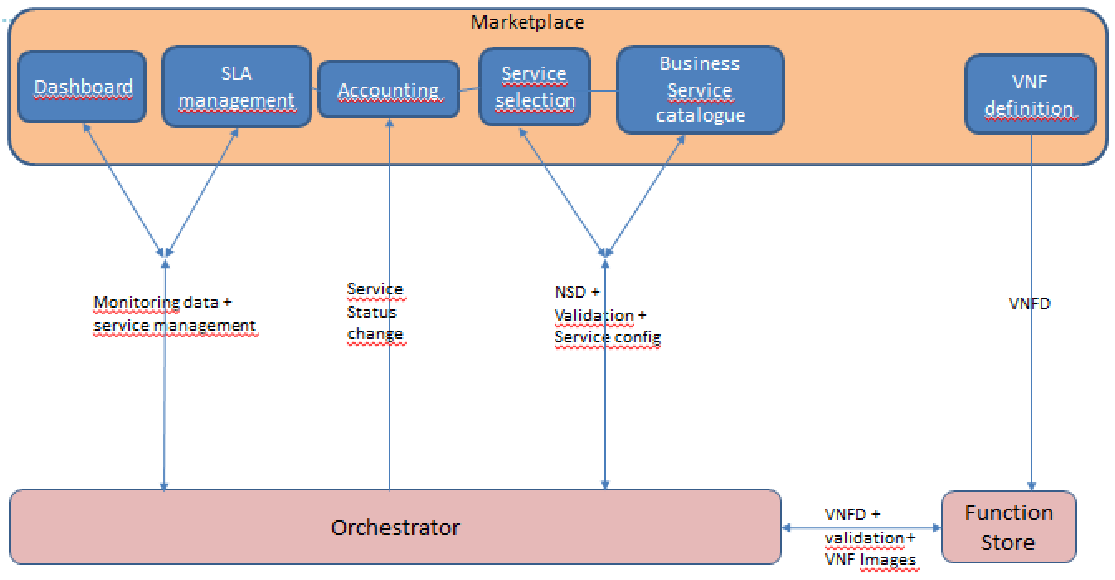


Figure 3-1: The interactions between the Orchestrator and the NF Store/Marketplace.

3.2.1.1. Accept Virtual Network Function Definitions

Virtual Network Functions (VNFs) must be present in the Orchestrator's VNF Catalogue before being part of any Network Service definition.

Table 3-1: VNFs interface with the NF Store.

Endpoint	Header Fields	Method	Description
/vnfs	X-Auth-Service-Key	POST	Registers the new VNF in the VNF Catalogue. The actual VNF Definition (VNFD) is part of the VNF creation request, but is not the request in it self. Since the VNF is first created in the NF Store, its unique ID (:vnf_id) is generated there and must be part of the request body.
/vnfs/:vnf_id	X-Auth-Service-Key	PUT	No specific use case was found for deleting (DELETE operation) and reading (GET operation)

			VNFs that have already been registered in the Orchestrator.

Since VNFs must be dealt with the VNFM, these requests are then forwarded to that microservice.

3.2.2. Interfaces with the Marketplace

The interface between the Orchestrator and the Marketplace is for managing Network Services (NSs) and Network Service Instances (NSIs).

3.2.2.1. Accept Network Services definitions

Network Services (NSs) must be present in the Orchestrator's NS Catalogue before being available for customers to purchase.

Table 3-2: NSs interface with the Marketplace.

Endpoint	Header Fields	Method	Description
`/network-services`	`X-Auth-Service-Key`	`POST`	Registers the new NS in the NS Catalogue. The actual NS Definition (NSD) is part of the NS creation request, but is not the request in itself.
`/network-services/:ns_id`	`X-Auth-Service-Key`	`PUT, DELETE`	No specific use case was found for reading (GET operation) NSs that have already been registered in the Orchestrator.

These requests are then forwarded to the **NS Catalogue** microservice.

3.2.2.2. Accept Network Services Instantiations Requests

Network Services Instances (NSIs) are requested whenever a Customer purchase a NS in the Marketplace.

Table 3-3: NS Instance interface with the Marketplace.

Endpoint	Header Fields	Method	Description
`/ns-instances`	`X-Auth-Service-Key`	`POST`	The :ns_id from which the instance is being requested must be present in the request body.
`/ns-instances/:nsi_id`	`X-Auth-Service-Key`	`PUT, GET, DELETE`	

/ns-instances/:nsi_id?state	X-Auth-Service-Key	GET	See possible states above.
/ns-instances/:nsi_id?state=stopped	X-Auth-Service-Key	GET	A query for NS instances in a certain state
/ns-instances/:nsi_id/:new_state	X-Auth-Service-Key	PUT	See possible states above.

These requests are then forwarded to the **NS Provisioning** microservice.

3.2.2.3. Request NS and VNF Instances Monitoring Data

The Marketplace can request monitoring data from the Orchestrator, either at the Network Service instance level or at the VNF instance level. The type of the instance must be part of the request, as well as the id, as well as the start and end dates of needed data. A maximum number of registers to return is optional, having a small preconfigured number.

Table 3-4: Monitoring data interface with the Marketplace.

Endpoint	Header Fields	Method	Description
/monitoring?instance_type=:it&instanceId=:it_id&metric=:name&from=:from &until=:until&maxResults=:max	X-Auth-Service-Key	GET	Accept NS instance monitoring data requests for displaying in the Marketplace.

3.2.2.4. Notify NS Instance Change of State

To notify the Marketplace about an update (PUT) to a NS instance's state, the following endpoint is available:

Table 3-5: Notification of change of state of a NS instance with the Marketplace.

Endpoint	Header Fields	Method	Description
/marketplace/accounting/servicestatus/:ns_instance_id/:state	X-Auth-Service-Key	PUT	

3.2.3. Interfaces with the WICM

While provisioning new NS instances, and when the customer wants the new instance connected to a specific network (WAN) point (e.g., for redirecting traffic for a given NFVI-PoP), this network connectivity must be established.

The WICM API with the Orchestrator is summarised in the following table.

Table 3-6: Interface with the WICM.

Endpoint	Header Fields	Method	Description
`/vnf-connectivity`	`X-Auth-Service-Key`	`POST`	The NS instance ID must be part of the request's body, since VLAN ID might not be unique.
`/vnf-connectivity/:ns_instance_id`	`X-Auth-Service-Key`	`PUT, GET, DELETE`	DELETE destroys the connectivity resource, thus ending the traffic redirection that had been put in place.

3.2.4. Interfaces with the VNFM

The interface between the Orchestrator and the VNFM concentrates all the features needed at the VNF level. Its URLs have the following common start:

```
http://apis.t-nova.eu/orchestrator/vnf-manager
```

3.2.4.1. Configure the VNFM

As described above, the ETSI MANO predicts the possibility of a certain VNFs providing its own VNF Manager. Therefore this interface must considered as 'external' (a better wording might be 'externally visible'), although some restrictions be must applied. For the reasons stated above, not all operations of the default (or generic) VNF Manager may be executed by VNF-specific VNF Managers.

Table 3-7: Interface with the VNF Manager.

Endpoint	Header Fields	Method	Description
`/configs/:config_id`	`X-Auth-Service-Key`	`PUT, GET`	Allows the configuration of supporting services
`/configs`	`X-Auth-Service-Key`	`GET`	Accessing all existing configurations

3.2.4.2. Manage VNFs

It is the VNFM's responsibility to manage all VNFs the Orchestrator is requested to manage.

Table 3-8: Interface for managing VNFs with the VNFM.

Endpoint	Header Fields	Method	Description
`/vnfs`	`X-Auth-Service-Key`	`POST, GET`	Due to the above explained reasons, this endpoint must only be used by the **NS**

			Manager. Due to the possibly high number of registered VNFs, the information returned when querying for all VNFs will be paginated (see [2] for details).
`/vnfs/:vnf_id`	`X-Auth-Service-Key`	`PUT, GET, DELETE`	

All these endpoints are supported by the **VNF Catalogue** microservice.

3.2.4.3. Manage VNF Instantiations Requests

Table 3-9: Interface for managing VNF instances with the VNFM.

Endpoint	Header Fields	Method	Description
`/vnf-instances`	`X-Auth-Service-Key`	`POST, GET`	The vnf external id of which this resource is an instance of must be part of the request body. Due to the possibly high number of VNF instances, the information returned when querying for all VNF instances will be paginated (see [1] for details).
`/vnf-instances?state=:state`	`X-Auth-Service-Key`	`GET`	Returns data on all the VNF instances that are currently in the indicated state. Due to the possibly high number of VNF instances, this information will be paginated (see [1] for details).
`/vnf-instances/:vnf_instance_id`	`X-Auth-Service-Key`	`PUT, GET, DELETE`	
`/vnf-instances/:vnf_instance_id?state`	`X-Auth-Service-Key`	`GET`	Returns the state on which the indicated VNF instance is in.
`/vnf-instances/:vnf_instance_id/:new_state`	`X-Auth-Service-Key`	`PUT`	Puts the VNF instance in the requested state.

All these endpoints are supported by the **VNF Provisioning** microservice.

3.2.4.4. Receive VNF Migration or Scaling Requests

The new infrastructure onto which the VNF should migrate must be given in the body of the request. This endpoint is used by the NS Manager when an SLA breach is reported and the VNF migration is found to be the solution to close that breach.

Migrating a VNF instance implies:

1. allocating new resources;
2. starting all the newly allocated VNF components;
3. reconfigure the new VNF and the connections it has with other VNFs, if existent;
4. stopping all the old VNF components;
5. releasing all the VNF allocated resources;

The general case for step #3 above might be very complex to implement, especially if state management (e.g., the VNF instance to be migrated has a database of significant size) is involved, when migrating that state it may take unacceptable time and significant (network) resources might have to be consumed. For executing a VNF instance migration the Network Service instance it is part of has to be stopped.

The kind of scaling to be executed must be given in the body of the request and can be out or in[2]. It also must have been defined by the Function Provider in the VNF Descriptor.

Table 3-10: Interface with the VNFM for scaling and migrating.

Endpoint	Header Fields	Method	Description
/vnf-instances/:vnf_instance_id/migrate	X-Auth-Service-Key	PUT	
/vnf-instances/:vnf_instance_id/scale	X-Auth-Service-Key	PUT	

All these endpoints are supported by the **VNF Scaling** microservice.

3.2.4.5. Receive VNF Instance Monitoring Parameters Definition

One of the steps in provisioning a new NS Instance is indicating the **NS Monitoring** microservice (through an internal interface) which monitoring parameters that particular NS instance will have. The NS Monitoring microservice decomposes the service into its constituent VNF monitoring parameters and uses this interface to subscribe them into the VIM's Monitoring Framework.

[2] Scaling up or down implies that features at the VIM level also support it.

Table 3-11: Interface with the NS Manager for accepting monitoring parameters subscriptions.

Endpoint	Header Fields	Method	Description
`/vnf-instances/:vnf_instance_id/monitoring-parameters`	`X-Auth-Service-Key`	`PUT`	The body of the request must have all the defining data for all the monitoring parameters at the VNF level.

This definition must be processed by the **VNF Monitoring** microservice.

3.3. External VNFM Interfaces

This sub-section details the external (southbound) interfaces of the Virtual Network Function Manager (VNFM) itself. Northbound interfaces are only with the Orchestrator, and have been described in the previous section.

As it happens for the Orchestrator, the internal structure of the VNFM is completely hidden from the outside: 'client' systems just have see one point of access. It is the VNF (NS Manager) role to redirect the requests to the microservice that has the responsibility to answer it.

The VNFM has one particularity, though: it might be provided with a certain VNF. This raises some issues of security, performance, etc., since it is in the Service Provider's infrastructure that required resources will be allocated on.

3.3.1. With the VIM

The interface between the VNFM and the VIM serves three purposes:

- to ensure the proper level of security and isolation;
- to allocate the needed infrastructure in the VIM;
- to know about which infrastructure there is (allocated or not);
- to subscribe monitoring parameters;
- to receive monitoring data about the instantiated VNFs and NSs.

These five kinds of interfaces are further described below.

3.3.1.1. Multitenancy

The multitenancy required for a given NS instance is requested to the VIM using KEYSTONE REST API [7].

Table 3-12: Interface with the VIM for multitenancy infrastructure.

Endpoint	Header Fields	Method	Description
`/v2.0/tenants`	`X-Auth-Token`	`POST`	Create a Tenant
`/v2.0/tenants/{tenantId}`	`X-Auth-Token`	`DELETE`	Delete a tenant
`/v2.0/users`	`X-Auth-Token`	`POST`	Create a new user.

/v2.0/users/{userId}	X-Auth-Token	DELETE	Delete a user
/v2.0/tenants/{tenantId}/users/{userId}/roles/OS-KSADM/{roleId}	X-Auth-Token	PUT	Grant roles to user on tenant

3.3.1.2. Allocate Infrastructure

The required infrastructure for a given VNF instance is requested to the VIM using HEAT REST API [7].

Table 3-13: Interface with the VIM for allocating infrastructure.

Endpoint	Header Fields	Method	Description
/v1/:tenant_id/stacks	X-Auth-Token	POST, GET	For POSTs, the body of the request must contain the HOT. For GETs, the body of the request is empty
/v1/:tenant_id/stacks/:stack_name/:stack_id	X-Auth-Token	DELETE	The body of the request is empty
/v1/:tenant_id/stacks/preview	X-Auth-Token	POST	The body of the request must have the stack name.

3.3.1.3. Manage the Infrastructure Repository

The interface between the Orchestrator and the VIM in terms of infrastructure implemented through the Infrastructure Repository subsystem, which had its own development Task and Deliverable ([22]). We therefore just summarise it in this deliverable.

OCCI is a RESTful protocol and API for various kinds of management tasks. These APIs support the development of interoperable tools for common tasks including deployment, autonomic scaling and monitoring. The middleware interface was implemented using the pyssf package. In accordance with the OCCI specification each resource in the repository is characterised by a kind. The kind is defined by a category in the OCCI model. This kind is immutable and specifies a resource's basic set of characteristics. This includes its location in the hierarchy, attributes, and applicable actions as shown in Table 3-14. The table only provides a sample of the available kinds. Further details are in section 4.5.1 of [22].

Table 3-14: Interface (partial) with the VIM for the Infrastructure Repository.

Endpoint	Methods	Description
/pop/	GET, POST, PUT, DELETE	Point of presence (PoP)

/pop/link/	GET, POST, PUT, DELETE	Link between two POPs (PoP Link)
/pop/:pop_id/stack/	GET	OpenStack Stack (Stack)
/pop/:pop_id/stack/link/	GET	Link between a stack and its resources (Stack Link)
/pop/:pop_id/vm/	GET	Virtual Machine (VM)

Finally the middleware implementation support the common service authentication mechanism (Gatekeeper) used by the T-NOVA Orchestration layer. All API calls received by the middleware layer must be authenticated before execution.

3.3.1.4. Request the VIM for the subscription of a monitoring parameter

The VIM Monitoring Framework (see [14]) supports a subscription mechanism, as described above, in sub-section 2.3.4). This subscription is supported using the following API.

Table 3-15: Interface with the VIM's Monitoring Framework for subscribing monitoring parameters.

Endpoint	Header Fields	Method	Description
/<vimmm>/metrics-push		POST	The body of the request must have all the defining data for all the monitoring parameters at the VNF level.

3.3.1.5. Receive Monitoring Parameters Readings from the VIM

For the subscribed VNF-level parameters, and only for those, the VNF Manager must be able to accept (POST operation) the reading the VIM Monitoring Framework provides. This is done by using the endpoint of Table 3-16.

Table 3-16: Interface with the NS Manager for accepting monitoring parameters readings.

Endpoint	Header Fields	Method	Description
/vnf-manager/readings/:parameter_name	X-Auth-Token	POST	The body of the request must contain the value and time-stamp of the parameter's reading.

3.3.2. With the VNFs

The interface between the VNFM and the VNFs is mainly for configuring those VNFs and it includes two phases: registering the VNF instance and then configuring the VNF instance.

Table 3-17: Interface with the VNFs (through the mAPI).

Endpoint	Header Fields	Method	Description
`/vnf_api`	`X-Auth-Token`	`POST`	If successful, a VNF API ID is returned.
`/vnf_api/:vnf_api_id`	`X-Auth-Token`	`PUT, DELETE`	PUT is used to pause/stop the VNF. DELETE destroys the VNF instance.
`/vnf_api/:vnf_api_id/config`	`X-Auth-Token`	`POST, PUT, GET`	The POST triggers the start command/script associated with that VNF instance. PUT is for pausing/stopping the VNF instance. The GET is used to read the configuration of a given VNF API

3.4. Extensibility

Independent microservices represent an ideal implementation in order to provide a modular solution for the service and VNF lifecycle management. Through this modularity, and considering the rapid evolution in the NFV realm, it becomes feasible to extend the Orchestrator in order to integrate or accommodate novel functionalities implemented also as microservices.

In fact, all the microservices within the orchestrator follow a registration procedure (including authentication and authorisation through the Gatekeeper). The registration process of a microservice requires the insertion of the path where to forward the requests together with the corresponding address where the service resides (IP and port). For example, a new component for Service Mapping could be registered into the NS Manager in order to let the orchestrator utilise multiple mapping algorithms depending of each request.

The process of adding a new microservice into the orchestrator comprehends the following logical steps:

1. Implement and deploy the microservice;
2. Request registration to the corresponding manager with the path and address of where the service runs;
3. Registration into the Gatekeeper (with the corresponding credentials).

For a specific example, let us consider the case of where a new Service Mapping algorithm has to be registered. Once the microservice is up and running, and with available access, it connects to the corresponding manager utilising the following POST method

```
/ns-provisioning/configs/registerService
```

with a body that contains the following information:

```
{
 "name": "serviceMapping1",
 "path": "/service-mapping",
 "host": "127.0.0.1",
 "port": "4060"
}
```

Once the new mapping algorithm is registered at the provisioning module, it will thus appear in the list of the corresponding registered microservices, which is available through the following GET method

```
/ns-provisioning/configs/services
```

4. Conclusions

The Orchestrator is the central sub-system of the whole T-NOVA system. It therefore interacts with all the other sub-systems (the NF Store, the Marketplace, the VIM, the WICM and the VNFs), through a substantial number of different interfaces, which have been designed and described in this report.

To implement these interfaces the most up-to-date practices and technologies have been adopted: a Representational State Transfer (REST)-based API, using JavaScript Object Notation (JSON) as the message format and the HyperText Transfer Protocol (HTTP) as the transport protocol.

These interfaces can be classified in two broad groups, in terms of the amount of information conveyed and the number of requests:

- Top-down flows will typically have lengthier messages (VNFs, NSDs, HEAT Templates), but occur less frequently;
- Bottom-up flows will typically contain short messages (responses to the top-down requests mentioned above, monitoring data values), but occur with a (very) high frequency.
- As we experiment further with the implementation, this may lead to optimisation of the bottom-up flows, namely due to the higher latency HTTP has, when compared to other transport protocols (e.g., User Datagram Protocol, UDP) used by the industry when low latency is a requirement (e.g., in Monitoring or Logging systems).
- The design of the whole Orchestrator API was influenced by the adoption of a microservice-based architecture for the Orchestrator itself. The manner in which the architecture is organised around 'resources' significantly lowers barriers for integrating both internal microservices and the external interfaces. We have seen this when the Gatekeeper, an internal authentication and authorisation microservice, was made available to the Infrastructure Repository (though an internal module of the Orchestrator's architecture, it had its own roadmap and technology stack). The Gatekeeper itself is made in Go, while Ruby is the chosen language for the Orchestrator, which shows the potential of usage of REST-based APIs, even internally.
  We have also chosen to expose some of what initially were thought as internal interfaces (beyond the Gatekeeper's one, already mentioned), namely the ones that support the interaction between the Orchestrator and the VNF Manager, due to the fact that ETSI (see [2]) sees as a possibility certain VNFs to bring their own VNF Manager. The Orchestrator's team feels that it is currently not clear how this possibility can be implemented in the general case, due to all the impacts it may bring to the Service Provider's infrastructure management, in terms of security, performance, etc. As further interactions with the Function Providers happen, we are sure that knowledge on this feature will increase up to the level where the best decisions might be made.

5. ACRONYMS

Acronym	Explanation
ACL	Access Control List
API	Application Program Interface
BSC	Business Service Catalogue
CPU	Central Processing Unit
FP	Function Provider
HA	High Availability
HOT	HEAT Orchestration Template
HTTP	Hypertext Transfer Protocol
IVM	Infrastructure virtualisation and management
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
mAPI	Middleware API
NBI	Northbound Interface
NF(Store)	Network Function (Store)
NS	Network Service
OCCI	Open Cloud Compute Interface
ODL	OpenDaylight (SDN Controller)
OSS	Operations Support System
PoP	Point-of-Presence
RAM	Random Access Memory
REST	Representational State Transfer
SBI	Southbound Interface
SDN	Software Defined Networking
SLA	Service Level Agreement
SP	Service Provider
SSH	Secure Shell
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
UDP	Universal Datagram Protocol

UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally unique identifier (
vCPU	Virtual Central Processing Unit
VDU	Virtual Deployment Unit
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VNF	Virtualised Network Function
VNFC	Virtualised Network Function Component
VNFM	Virtualised Network Function Manager
WAN	Wide Area Network
WICM	WAN Infrastructure Connection Manager

6. References

[1] J. Bonnet et. al., "T-NOVA Deliverable D3.01 Interim Report on the Orchestrator Platform Implementation"

[2] ETSI GS NFV 002 (http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf)

[3] G. Banga , F. Douglis, M. Rabinovich, "Optimistic Deltas for WWW Latency Reduction" (https://www.usenix.org/legacy/publications/library/proceedings/ana97/full_papers/banga/banga_html/usenix.html)

[4] "TCP vs. UDP" (http://www.diffen.com/difference/TCP_vs_UDP)

[5] M. Fowler, J. Lewis, "Microservices" (http://martinfowler.com/articles/microservices.html)

[6] C. Richardson, "Microservice architecture patterns and best practices" (http://microservices.io/)

[7] OpenStack Orchestration API (http://developer.openstack.org/api-ref-orchestration-v1.html)

[8] OpenStack REST API (http://developer.openstack.org/api-ref.html)

[9] OpenStack Template guide (http://docs.openstack.org/developer/heat/template_guide/index.html)

[10] OpenStack Icehouse Release Notes (https://wiki.openstack.org/wiki/ReleaseNotes/Icehouse)

[11] Resources for application development on private and public OpenStack clouds (http://developer.openstack.org/)

[12] Heat Environments (http://docs.openstack.org/developer/heat/template_guide/environment.html)

[13] OCCI. (2015). *The Open Cloud Computing Interface*. Available: http://occi-wg.org/

[14] G. Gardikis et. al., "T-NOVA Deliverable D4.42 Monitoring and Maintenance -- Final"

[15] M. Stonebraker , U. Çetintemel, S. Zdonik, "The 8 Requirements of Real-Time Stream Processing" (http://cs.brown.edu/%7Eugur/8rulesSigRec.pdf)

[16] P. Comi et al., "T-NOVA Deliverable D2.42 Specification of the Network Function framework and T-NOVA Marketplace"

[17] B. Parreira et al., "T-NOVA Deliverable D5.2 Function Deployment, Configuration and Management"

[18] LogStash (https://www.elastic.co/products/logstash)

[19] ElasticSearch (https://www.elastic.co/products/elasticsearch)

[20] Kibana (https://www.elastic.co/products/kibana)

[21] "Single Responsibility Principle" (http://www.oodesign.com/single-responsibility-principle.html)
[22] M. McGrath et al., "T-NOVA Deliverable D3.2 Infrastructure Resource Repository"

Annexes

Annex A The Orchestrator Architecture

The T-NOVA Orchestrator Architecture is presented in Figure A-1.

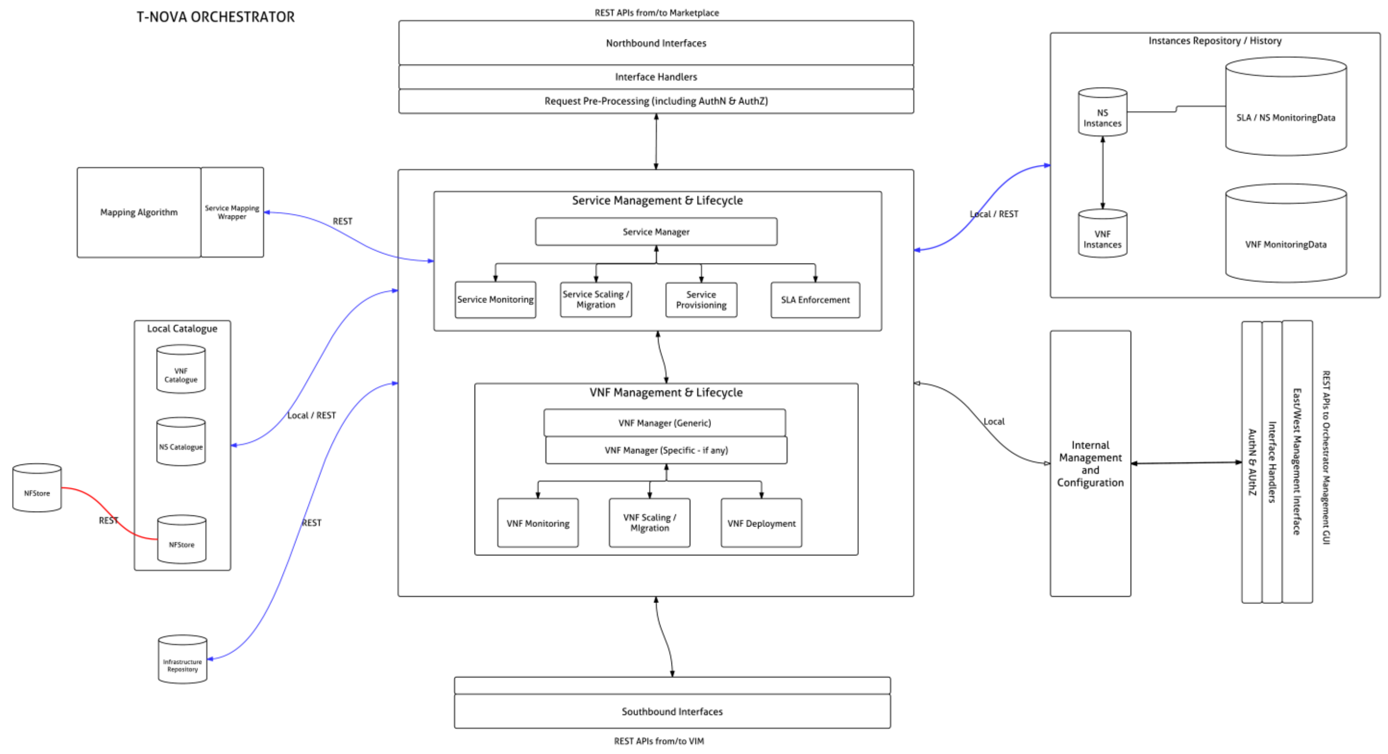


Figure A-1: The Orchestrator Architecture.

Annex B The Orchestrator API

This Annex lists the options and standards supporting the Orchestrator's APIs, as well as the APIs themselves.

B.1 NS Manager

Table B-1: The NS Manager's API

URI	Method	Purpose
`/`	GET	REST API Structure and Capability Discovery
`/configs/registerService`	POST	Register a service configuration
`/configs/unRegisterService/{microservice}`	POST	Unregister a service configuration
`/configs/services`	GET	List all services configuration
`/configs/services`	PUT	Update service configuration
`/configs/services/{name}/status`	PUT	Update service status
`/{path}`	GET	Redirects a GET request to the specified microservice
`/{path}`	POST	Redirects a POST request to the specified microservice
`/{path}`	DELETE	Redirects a DELETE request to the specified microservice

B.1.1 Return codes

Table B-2: The NS Manager's API return codes.

Code	Description
200	OK
400	Bad Request
404	Not Found
415	Unsupported Media Type
500	Internal Server Error

B.2 VNF Manager

Table B-3: The VNF Manager's API

URI	Method	Purpose

`/`	GET	REST API Structure and Capability Discovery
`/vnfs`	GET	List all VNFs
`/vnfs/{external_vnf_id}`	GET	List a specific VNF
`/vnfs`	POST	Store a new VNF
`/vnfs/{external_vnf_id}`	PUT	Update a stored VNF
`/vnfs/{external_vnf_id}`	DELETE	Delete a specific VNF
`/vnf-instances`	POST	Request the instantiation of a VNF
`/configs`	GET	List all services configurations
`/configs/{config_id}`	GET	List a specific service configuration
`/configs/{config_id}`	PUT	Update a service configuration
`/configs/{config_id}`	DELETE	Delete a service configuration

B.2.1Return codes

Table B-4: The VNF Manager's API return codes

Code	Description
200	OK
400	Bad Request
404	Not Found
415	Unsupported Media Type
500	Internal Server Error

B.3Interface between the VNFM and the VIM

B.3.1Infrastructure allocation

Table B-5: The VNF Manager's AP with the VIMI

URI	Method	Purpose
`/v1/:tenant_id/stacks`	POST, GET	Provision a VNF. The body of the request must contain the body
`/v1/:tenant_id/stacks/:stack_name/:stack_id`	PUT, DELETE	

/v1/:tenant_id/stacks/preview	POST	The body holds the stack name

In the response body of a POST the client receives the stack URL that contains the stack ID and the stack name to use in subsequent calls to HEAT:

```
{
    "stack": {
        "id": "3095aefc-09fb-4bc7-b1f0-f21a304e864c",
        "links": [
            {
                "href":
"http://192.168.123.200:8004/v1/eb1c63a4f77141548385f113a28f0f52/stacks/simple_stack/3
095aefc-09fb-4bc7-b1f0-f21a304e864c",
                "rel": "self"
            }
        ]
    }
}
```

In the response body of a GET the client receives something like::

```
{
    "stacks": [
        {
            "creation_time": "2014-06-03T20:59:46Z",
            "description": "sample stack",
            "id": "3095aefc-09fb-4bc7-b1f0-f21a304e864c",
            "links": [
                {
                    "href":
"http://192.168.123.200:8004/v1/eb1c63a4f77141548385f113a28f0f52/stacks/simple_stack/3
095aefc-09fb-4bc7-b1f0-f21a304e864c",
                    "rel": "self"
                }
            ],
            "stack_name": "simple_stack",
            "stack_status": "CREATE_COMPLETE",
            "stack_status_reason": "Stack CREATE completed successfully",
            "updated_time": "",
            "tags": ""
        }
    ]
}
```

In the response body of a preview the client receives something like::

```
{
    "stack": {
        "capabilities": [],
        "creation_time": "2015-01-31T15:12:36Z",
        "description": "HOT template for Nova Server resource.\n",
        "disable_rollback": true,
        "id": "None",
        "links": [
            {
                "href":
"http://192.168.122.102:8004/v1/6e18cc2bdbeb48a5basad2dc499f6804/stacks/test_stack/Non
e",
                "rel": "self"
            }
        ],
        "notification_topics": [],
        "parameters": {
            "OS::project_id": "6e18cc2bdbeb48a5basad2dc499f6804",
            "OS::stack_id": "None",
            "OS::stack_name": "teststack",
            "admin_user": "cloud-user",
            "flavor": "m1.small",
            "image": "F20-cfg",
            "key_name": "heat_key",
            "server_name": "MyServer"
        },
        "parent": null,
        "resources": [
            {
                "attributes": {},
                "description": "",
                "metadata": {},
                "physical_resource_id": "",
                "properties": {
                    "description": "Ping and SSH",
                    "name": "the_sg",
                    "rules": [
                        {
                            "direction": "ingress",
                            "ethertype": "IPv4",
                            "port_range_max": null,
                            "port_range_min": null,
                            "protocol": "icmp",
                            "remote_group_id": null,
                            "remote_ip_prefix": null,
                            "remote_mode": "remote_ip_prefix"
                        },
                        {
                            "direction": "ingress",
                            "ethertype": "IPv4",
                            "port_range_max": 65535,
                            "port_range_min": 1,
                            "protocol": "tcp",
                            "remote_group_id": null,
                            "remote_ip_prefix": null,
                            "remote_mode": "remote_ip_prefix"
                        },
                        {
                            "direction": "ingress",
                            "ethertype": "IPv4",
                            "port_range_max": 65535,
                            "port_range_min": 1,
                            "protocol": "udp",
                            "remote_group_id": null,
                            "remote_ip_prefix": null,
                            "remote_mode": "remote_ip_prefix"
                        }
                    ]
                },
                "required_by": [
                    "server1"
                ],
                "resource_action": "INIT",
                "resource_identity": {
                    "path": "/resources/the_sg_res",
                    "stack_id": "None",
                    "stack_name": "teststack",
                    "tenant": "6e18cc2bdbeb48a5b3cad2dc499f6804"
                },
                "resource_name": "the_sg_res",
                "resource_status": "COMPLETE",
                "resource_status_reason": "",
                "resource_type": "OS::Neutron::SecurityGroup",
                "stack_identity": {
                    "path": "",
                    "stack_id": "None",
                    "stack_name": "teststack",
                    "tenant": "6e18cc2bdbeb48a5b3cad2dc499f6804"
                },
                "stack_name": "teststack",
                "updated_time": "2015-01-31T15:12:36Z"
            },
            {
                "attributes": {
                    "accessIPv4": "",
                    "accessIPv6": "",
                    "addresses": "",
                    "console_urls": "",
                    "first_address": "",
                    "instance_name": "",
                    "name": "MyServer",
                    "networks": "",
                    "show": ""
                },
                "description": "",
                "metadata": {},
                "physical_resource_id": "",
                "properties": {
                    "admin_pass": null,
                    "admin_user": "cloud-user",
                    "availability_zone": null,
                    "block_device_mapping": null,
                    "config_drive": null,
                    "diskConfig": null,
                    "flavor": "m1.small",
                    "flavor_update_policy": "RESIZE",
                    "image": "F20-cfg",
                    "image_update_policy": "REPLACE",
                    "key_name": "heat_key",
                    "metadata": {
                        "ha_stack": "None"
                    },
                    "name": "MyServer",
                    "networks": [
                        {
                            "fixed_ip": null,
                            "network": "private",
                            "port": null,
                            "uuid": null
                        }
                    ],
                    "personality": {},
                    "reservation_id": null,
                    "scheduler_hints": null,
                    "security_groups": [
                        "None"
                    ],
                    "software_config_transport": "POLL_SERVER_CFN",
                    "user_data": "",
                    "user_data_format": "HEAT_CFNTOOLS"
                },
                "required_by": [],
                "resource_action": "INIT",
                "resource_identity": {
                    "path": "/resources/hello_world",
                    "stack_id": "None",
                    "stack_name": "teststack",
                    "tenant": "6e18cc2bdbeb48a3433cad2dc499sdf32234"
                },
                "resource_name": "hello_world",
                "resource_status": "COMPLETE",
                "resource_status_reason": "",
                "resource_type": "OS::Nova::Server",
                "stack_identity": {
                    "path": "",
                    "stack_id": "None",
                    "stack_name": "teststack",
                    "tenant": "6e18cc2bdbeb48a3433cad2dc499sdf32234"
                },
                "stack_name": "teststack",
                "updated_time": "2015-01-31T15:12:36Z"
            }
        ],
        "stack_name": "test_stack",
        "stack_owner": "admin",
        "template_description": "HOT template for Nova Server resource.\n",
        "timeout_mins": null,
        "updated_time": null
    }
}
```

B.3.2Return codes

Table B-6: The VNF Manager's API with the VIM return codes

Code	Description
200	OK
400	Bad Request
401	Unauthorized
404	Not found
409	Conflict
500	Internal Server Error

B.4Interfaces compilation, installation and deployment Guide

This section has all the information on how to access the source code for the interfaces, how to install the prototype, and how to deploy it if required.

All the software developed is located in the corresponding repository

```
http://stash.i2cat.net/
```

To access and install the code the following user and password are required:

```
User: tnova_reviewer
Password: 2ndreview
```

You need to run the following commands to download the code:

First, download all the orchestator microservices from source code

```
$ git clone http://username@git.i2cat.net/scm/TNOV/wp3.git
$ cd wp3/WP3/
```

Download, resolve and install all the dependencies required for each microservice:

```
$ cd orchestrator_{microservice_name}
$ bundle install
```

Run each microservice without follow any order:

```
$ cd orchestrator_{microservice_name}
$ rake start
```

Some microservices require external applications for execute the run command. These external applications are databases: PostgreSQL and Apache Cassandra.

In the case that the PostgreSQL is required, the following command will create the database and the corresponding tables:

```
$ rake db:create
$ rake db:migrate
```

In the case of Apache Cassandra, the database schema should be loaded directly into Cassandra:

```
$ cd orchestrator_ns-monitoring-repository
$ apache-cassandra/bin/cqlsh -f db/schema.txt
```

B.4.1Gatekeeper

The Gatekeeper code is located in the same repository:

```
http://stash.i2cat.net/
```

You need to run the following commands to install the code:

Install

```
$ cd wp3/WP3/orchestrator_gatekeeper
$ go get
$ go install
```

Run

```
$ go run*.go
```

B.4.2Management UI

The management user interface is located in the same repository:

```
http://stash.i2cat.net/
```

You need to run the following commands to install the code:

Install

```
$ cd wp3/WP3/orchestrator_mgt-gui
$ mvn clean install -DskipTests
```

Run

```
$ mvn jetty:run
```

Annex C VNF DESCRIPTOR

This annex shows an example of a full VNF Descriptor, as it is being used by the FPs, at the time of writing. The content of this kind of file will probably change until the end of the project.

```
{
   "vnfd": {
      "release": "T-NOVA v.0.1",
      "id": "id number",
      "vendor": "vendor generating this VNFD",
      "provider_id": "function provider id",
      "description": "description of the function of the VNF",
      "description_version": "version of this VNFD",
      "version": "version of VNF software",
      "type": "TC / should an ontology be defined or this is free text input?",
        "__comment": "maybe include some keyword selection i.e storage and be included
here",
      "__comment": "in order to be used efficiently deployed",
      "date_created": "date created e.g 2015-06-11T13:10:00Z",
      "date_modified": "date modified",
      "trade": "TRUE",
      "billing_model":{
         "model": "billing model informantion e.g. PAYG",
         "period": "billing period e.g. P1W",
         "price" :{
   "unit":"billing unit e.g. EUR",
   "min_per_period":"5",
   "max_per_period":"10",
   "setup":"0"
         }
      },
      "vdu": [
         {
            "id": "vdu uuid1",
            "vm_image": "image reference uri",
            "computation_requirement": {
               "vcpus": "number of virtual cpus"
            },
            "virtual_memory_resource_element": "virtual memory needed eg 10M",
            "virtual_network_bandwidth_resources": "virtual bandwidth eg 10Mbit",
            "lifecycle_event": {
               "driver": "SSH",
               "Authentication": "private_key.pem",
               "Authentication_Type":"private key",
               "__comment": "information about IP resides in the VNFR not VNFD",
               "VNF_Container":"path/to/container e.g. /home/tnova/container/",
               "start": {
                    "command": "service vnf start",
                    "template_file_format": "json",
                    "Template_file": {
                      "controller": "get_attr: [controller, floating_ip]",
                      "__comment": "we should identify the vdu that is the
controlling/managing vdu for all the VNFCs, this implies that a tag should be placed
in that vdu",
                      "__comment": "assumption that that vdu is the first to start and
delegates all the other config commands to the rest",
                      "vdu1":"get_attr: [instance1, mngt_network_ip]",
                      "vdu2":"get_attr: [instance2, mngt_network_ip]"
                    }
               },
               "stop": "path/to/script",
               "restart": "path/to/script"
            },
            "constraint": "placeholder for other constraints",
            "high_availability": "ActiveActive or ActivePassive",
            "scale_in_out": {
               "minimum": "minimum number of instances",
               "maximum": "maximum number of instances"
            },
            "vnfc": [
               {
                    "id": "vnfcid1",
                    "connection_point": [
                      {
                          "id": "connectionpointidvnfc1",
                          "vitual_link_reference": "virtual link reference",
                          "type": "virtual/physical port/nic or vpn endpoint ip"
                      },
                      {
                          "id": "connectionpointidvnfc2",
                          "vitual_link_reference": "virtual link reference",
                          "type": "virtual/physical port/nic or vpn endpoint ip"
                      }
                    ]
               },
               {
                    "id": "vnfcid2",
                    "connection_point": [
                      {
                          "id": "connectionpointidvnfc3",
                          "vitual_link_reference": "virtual link reference",
                          "type": "virtual/physical port/nic or vpn endpoint ip"
                      },
                      {
                          "id": "connectionpointidvnfc4",
                          "vitual_link_reference": "virtual link reference",
                          "type": "virtual/physical port/nic or vpn endpoint ip"
                      }
                    ]
               }
            ],
            "monitoring_parameters": [
               {
                    "monitoring_parameter": "memory-consumption"
               },
               {
                    "monitoring_parameter": "CPU-utilization"
               },
               {
                    "monitoring_parameter": "bandwidth-consumption"
               },
               {
                    "monitoring_parameter": "VNFC-downtime"
               }
            ],
            "cpu_instruction_set_extension": "",
            "cpu_model": "",
            "cpu_model_specification_binding": "",
            "cpu_min_clock_speed": "",
            "cpu_core_reservation": "",
            "cpu_simultaneous_multi_threading_hw_thread_specification": "",
            "cpu_core_oversubscription_policy": "",
            "cpu_core_and_hw_thread_allocation_topology_policy": "",
            "cpu_last_level_cache_size": "",
            "cpu_direct_io_access_to_cache": "",
            "cpu_translation_look_aside_buffer_parameters": {
               "TLB_size": "",
               "TLB_large_page_support": "",
               "IOTLB_size": "",
               "IOTLB_large_page_support": ""
            },
            "cpu_hot_add": "",
            "cpu_support_accelerator": "",
            "memory_parameters": {
               "type": "",
               "speed": "",
               "channels": "",
               "size": "",
               "error_correction_codes": "",
               "oversubscription_policy": "",
               "bandwidth_required": "",
               "large_pages_required": "",
               "NUMA_allocation_policy": ""
            },
            "memory_hot_add": "",
            "platform_security_parameters": {
```

```
            "random_number_generation": "",
            "measure_launch_environment": ""
         },
         "hypervisor_parameters": {
            "type": "",
            "version": "",
            "second_level_address_translation": "",
            "second_level_address_translation_with_large_page_support": "",
            "second_level_address_translation_for_io": "",
            "second_level_address_translation_for_io_with_large_page": "",
            "support_for_interrupt_remapping": "",
            "support_for_data_processing_acceleration_libraries": ""
         },
         "platform_pcie_parameters": [
            {
                 "type": "type of pcie device e.g. NIC",
                 "general_capabilities": "",
                 "bandwidth": "",
                 "device_pass_through": "True/False",
                 "SR-IOV": "",
                 "device_assignement_affinity": ""
            }
            {
                 "type:" "type of pcie device e.g. NIC"
                 "general_capabilities": "",
                 "bandwidth": "",
                 "device_pass_through": "True/False",
                 "SR-IOV": "Info rgarding SR-IOV deployment",
                 "device_assignement_affinity": ""
            }
         ],
         "pcie_advanced_error_reporting": "",
         "platform_acceleration_device": "",
         "network_interface_card_capabilities": {
            "LSO": "",
            "LRO": "",
            "checksum": "",
            "RSS": "",
            "flow_director": "",
            "mirroring": "",
            "availability": "",
            "jumbo_support": "",
            "VLAN_tag": "",
            "RDMA": "",
            "SR-IOV": ""
         },
         "network_interface_bandwidth": "eg 1GBit",
         "data_processing_acceleration_library": "eg DPDK v1.0",
         "vswitch_capabilities": {
            "type": "ovs",
            "version": "2.0",
            "overlay_tunnel": "GRE"
         },
         "corrected_error_notification": "number of correctable errors",
         "uncorrected_error_notification": "number of error raising exceptions",
         "storage_requirements": {
            "size": "size required eg 30GB",
            "KQI1": "IOPS limit if applicable",
            "KQI2": ""
         },
         "rdma_support_bandwidth": "rdma bandwidth"
      }
   ],
   "virtual_link": [
      {
         "id": "vlinkid1 number",
         "connectivity_type": "E-Line, E-LAN or E-Tree",
         "connection_points_references": [
            {"id": "connection_point_id1"},
            {"id": "connection_point_id2"}
         ],
         "root_requirement": "root bandwidth",
         "leaf_requirement": "leaf bandwidth",
         "qos": "qos options, eg latency, jitter",
         "test_access": "none, passive monitoring, active monitoing"
      },
      {
            "id": "vlinkid2 number",
            "connectivity_type": "E-Line, E-LAN or E-Tree",
            "connection_points_references": [
               {"id": "connection_point_id1"},
               {"id": "connection_point_id2"}
            ],
            "root_requirement": "root bandwidth",
            "leaf_requirement": "leaf bandwidth",
            "qos": "qos options, eg latency, jitter",
            "test_access": "none, passive monitoring, active monitoing"
         }
      ],
      "connection_point":[
         {
            "id": "connectionpointid1",
            "vitual_link_reference": "virtual link reference",
            "type": "virtual/physical port/nic or vpn endpoint ip"
         },
         {
            "id": "connectionpointid2",
            "vitual_link_reference": "virtual link reference",
            "type": "virtual/physical port/nic or vpn endpoint ip"
         }
      ],
      "lifecycle_event": {
         "__comment": "when VNFM initiates a start for this VNF, the VNFM should
communicate with",
         "__comment": "the VIM in order to spin on the VMs (NVFCs),",
         "__comment": "then sends the start command to the controll VM (VNFC)
indicated by a specific tag (TBD)",
         "__comment": "The controll VNFC reiterates the programmed actions to the
other VNFCs",
         "__comment": "Is this a valid assumption or should the VNFM communicate with
each and every VNFC in order",
         "__comment": "to configure them?",
         "driver": "driver to be used for accessing the management VNFC, e.g. SSH",
         "Authentication": "path/to/private_key",
         "Authentication_Type": "Authentication type e.g. PrivateKey/ Digest ",
         "__comment": "information about IP resides in the VNFR not VNFD",
         "VNF_Container":"path/to/container e.g. /home/tnova/container/",
         "__comment": "we should identify the vdu that is the controlling/managing vdu
for all the VNFCs",
         "__comment": "this implies that a tag should be placed in that vdu",
         "__comment": "assumption that that vdu is the first to start",
         "__comment": "and delegates all the other config commands to the rest",
         "events": [
            {
                   "lifecycle_event": "start",
                   "controller": "get_attr: [controller, floating_ip]",
                   "vdu1":"get_attr: [instance1, mngt_network_ip]",
                   "vdu2":"get_attr: [instance2, mngt_network_ip]"

            },
            {
               "lifecycle_event": "stop",
               "controller": "get_attr: [controller, floating_ip]",
               "vdu1":"get_attr: [instance1, mngt_network_ip]",
               "vdu2":"get_attr: [instance2, mngt_network_ip]"

          },
            {
               "lifecycle_event": "restart",
               "controller": "get_attr: [controller, floating_ip]",
               "vdu1":"get_attr: [instance1, mngt_network_ip]",
               "vdu2":"get_attr: [instance2, mngt_network_ip]"
          },
            {
               "lifecycle_event": "scale-in",
               "controller": "get_attr: [controller, floating_ip]",
               "vdu1":"get_attr: [instance1, mngt_network_ip]",
               "vdu2":"get_attr: [instance2, mngt_network_ip]"
          },
            {
               "lifecycle_event": "scale-out",
               "lifecycle_event": "restart",
               "controller": "get_attr: [controller, floating_ip]",
               "vdu1":"get_attr: [instance1, mngt_network_ip]",
            "vdu2":"get_attr: [instance2, mngt_network_ip]"
         }
        ]
      },
      "dependency": [
         {
            "source_vdu": "sourcevduid",
            "target_vdu": "targetvduid"
         },
         {
            "source_vdu": "sourcevduid",
            "target_vdu": "targetvduid"
         }
      ],
      "monitoring_parameters": [
         {
            "monitoring_parameter": "memory-consumption",
            "description": "Maximum memory consumed by the VNF"
         },
         {
            "monitoring_parameter": "CPU-utilization",
            "description": ""
         },
         {
            "monitoring_parameter": "bandwidth-consumption",
            "description": ""
         },
         {
            "monitoring_parameter": "VNFC-downtime",
            "description": ""
         }
      ],
      "deployment_flavour": [
         {
            "id": "vnfflavourid1",
            "flavour_key": "calls5k",
            "constraint": "specific hardware constraint",
            "constituent_vdu": {
               "vdu_reference": "vduid for this deployment",
               "number_of_instances": "number of VDU instance required",
               "constituent_vnfc": "references vnfc id"
            },
            "assurance-params":[
               {
      "param-id":"memory-consumption",
      "value":"1",   "unit":"MB",    "formula": "memory-consumption LT 1",
      "violation": [
         {
            "breaches_count": 2,
            "interval": 30
         },
         {
                       "breaches_count": 5,
                       "interval": 120
                   }
                 ],
                 "penalty": {
                   "type" : "discount",
                   "value": 5,
                   "unit": "%",
                   "validity": "P1D"
                 }
      },
               {
      "param-id":"CPU-utilization",
      "value":"70",
      "unit":"percentage",
      "formula": "CPU-utilization GT 70",
      "violation": [
         {
            "breaches_count": 2,
            "interval": 30
         }
                 ],
                 "penalty": {
                   "type" : "discount",
                   "value": 5,
                   "unit": "%",
                   "validity": "P1D"
                 }
      }
            ]
         },
         {
            "id": "vnfflavourid2",
            "flavour_key": "calls10k",
            "constraint": "specific hardware constraint",
            "constituent_vdu": {
               "vdu_reference": "vduid for this deployment",
               "number_of_instances": "number of VDU instance required",
               "constituent_vnfc": "references vnfc id"
            },
            "assurance-params":[
               {
      "param-id":"memory-consumption",
      "value": 1.5,
      "unit":"MB",   "formula": "memory-consumption LT 1.5",
      "violation": [
         {
            "breaches_count": 2,
            "interval": 30
         },
         {
                       "breaches_count": 5,
                       "interval": 120
                   }
                 ],
                 "penalty": {
                   "type" : "discount",
                   "value": 0.05,
                   "unit": "percentage",
                   "validity": "P1D"
                 }
      },
               {
      "param-id":"CPU-utilization",
      "value": 0.8,
      "unit":"percentage",
      "formula": "CPU-utilization LT 0.8",
      "violation": [
         {
            "breaches_count": 2,
            "interval": 30
         }
                 ],
                 "penalty": {
                   "type" : "discount",
                   "value": 0.05,
                   "unit": "percentage",
                   "validity": "P1D"
                 }
      }
            ]
         },
         {
            "id": "vnfflavourid1",
            "flavour_key": "users50k",
            "constraint": "specific hardware constraint",
            "constituent_vdu": {
               "vdu_reference": "vduid for this deployment",
               "number_of_instances": "number of VDU instance required",
               "constituent_vnfc": "references vnfc id"
            },
            "assurance-params":[
               {
      "param-id":"memory-consumption",
      "value": 1,
      "unit":"MB",
      "formula": "memory-consumption LT 1",
      "violation": [
         {
            "breaches_count": 2,
            "interval": 30
         },
         {
                         "breaches_count": 5,
                         "interval": 120
                     }
                   ],
                   "penalty": {
                     "type" : "discount",
                     "value": 0.05,
                     "unit": "percentage",
                     "validity": "P1D"
                   }     },
               {
      "param-id":"CPU-utilization",
      "value": 0.85,
      "unit":"percentage",
      "formula": "CPU-utilization LT 0.85",
      "violation": [
         {
            "breaches_count": 2,
            "interval": 30
         }
                   ],
                   "penalty": {
                     "type" : "discount",
                     "value": 0.05,
                     "unit": "percentage",
                     "validity": "P1D"
                   }     }
            ]
         }
      ],
      "auto_scale_policy": [
         {
            "criteria_parameter": {
               "type": "monitoring parameter name",
               "threshold": "threshold"
            },
            "action_type": "scale-out to different flavour ID"
         },
         {
            "criteria_parameter": {
               "monitoring_parameter": "monitoring parameter name",
               "threshold": "threshold"
            },
            "action_type": "scale-out to different flavour ID"
         }

      ],
      "manifest_file": "path/to/file",
      "manifest_file_security": "manifest file md5 hash"
   }
}
```

Annex D NS DESCRIPTOR

This annex shows an example of a full NS Descriptor, as it is being used by the Marketplace, at the time of writing. The content of this kind of file will probably change until the end of the project.

```
{
   "nsd": {
      "id": "network service id",
      "name": "name",
      "vendor": "vendor of the NS",
      "version": "version of the NSD",
      "vnfds": [
         "__comment": "NFStore-generated IDs",
         {
            "vnfd": "reference of a vnfd of this ns"
         },
         {
            "vnfd": "reference of a vnfd of this ns"
         }
      ],
      "vnffgds": [
         {
            "vnffgd": "reference of a vnffgd of this ns"
         },
         {
            "vnffgd": "reference of a vnffgd of this ns"
         }
      ],    "lifecycle_event": {
         "__comment": " /* to be decided later */"
      },
      "vnf_dependency": [
         {
            "source_vnf": "sourcevnfid",
            "target_vnf": "targetvnfid"
         },
         {
            "source_vnf": "sourcevnfid",
            "target_vnf": "targetvnfid"
         }
      ],
      "monitoring_parameters": [
         {
            "monitoring_parameter": "availability"
         },
         {
            "monitoring_parameter": "ram-consumption"
         },
      ],
      "service_deployment_flavour": [
         {
            "id": "nsflavourid1",
            "flavour_key": "callspersecond5k",
            "constituent_vnf": {
               "vnf_reference": "vnfid for this deployment",
               "vnf_flavour_id_reference": "reference of vnfd:deployment_flavour:id",
               "redundancy_model": "active or standby",
               "affinity": "placement policy between instances",
               "capability": "eg instance capacity, 50% * NS capacity",
               "number_of_instances": "number of vnf instances required"
            },
            "assurance_parameters": [
      "__comment": "the values are calculated based on the values of the VNF selected
flavour",
      {
         "__comment": "",
         "name": "availability",
         "value": "GT(0.99)",
         "formula": "min(vnfs[1].availability, vnfs[2].availability)",
         "violation": [
            {
               "breaches_count": "5",
               "interval": "120",
               "penalty": "not included, as they're not relevant to the Orchestrator"
            }
         ]
      },
      {
         "name": "ram-consumption",
         "value": "LT(add(vnfs[1].memory-consumption, vnfs[2].memory-consumption,
100))",
         "__comment": "/* allow 100MB extra over the combined consumption by the VNFs
*/",
         "formula": "add(vnfs[1].memory-consumption, vnfs[2].memory-consumption,
100)",
         "violation": [
            {
               "breaches_count": "5",
               "interval": "120",
               "penalty": "not included, as they're not relevant to the Orchrstrator"
            },
            {
               "breaches_count": "10",
               "interval": "300",
               "penalty": "not included, as they're not relevant to the Orchrstrator"
            }
         ],
      },
      ],
         },
         {
            "id": "nsflavourid2",
            "flavour_key": "callspersecond10k",
            "constituent_vnf": {
               "vnf_reference": "vnfid for this deployment",
               "vnf_flavour_id_reference": "reference of vnfd:deployment_flavour:id",
               "redundancy_model": "active or standby",
               "affinity": "placement policy between instances",
               "capability": "eg instance capacity, 50% * NS capacity",
               "number_of_instances": "number of vnf instances required"
            },
            "assurance_parameters": [
      "__comment": "the values are calculated based on the values of the VNF selected
flavour",
      {
         "__comment": "",
         "name": "availability",
         "value": "GT(min(vnfs[1].availability, vnfs[2].availability))",
         "formula": "min(vnfs[1].availability, vnfs[2].availability)",
         "violation": [
            {
               "breaches_count": "5",
               "interval": "120",
               "penalty": "not included, as they're not relevant to the Orchestrator"
            }
         ]
      },
      {
         "name": "ram-consumption",
         "value": "LT(add(vnfs[1].memory-consumption, vnfs[2].memory-consumption,
100))",
         "__comment": "/* allow 100MB extra over the combined consumption by the VNFs
*/",
         "formula": "add(vnfs[1].memory-consumption, vnfs[2].memory-consumption,
100)",
         "violation": [
            {
               "breaches_count": "5",
               "interval": "120",
               "penalty": "not included, as they're not relevant to the Orchestrator"
            },
            {
               "breaches_count": "10",

               "interval": "300",

               "penalty": "not included, as they're not relevant to the Orchrstrator"
            }
         ],
      },
      ],
         }
      ],
   "t_nova_service_deployment_flavour": [
         {
      "name": "gold"
         }
      ],
      "billing": {
         "type": "billing model",
         "period": "billing period",
         "price":{
            "currency:": "",
            "setupCost": "",
            "price_per_period": ""
         }
      },
      "auto_scale_policy": [
         {
            "criteria_parameter": {
               "type": "monitoring parameter name",
               "threshold": "threshold"
            },
            "action_type": "scale-out to different flavour ID"
         },
         {
            "criteria_parameter": {
               "monitoring_parameter": "monitoring parameter name",
               "threshold": "threshold"
            },
            "action_type": "scale-out to different flavour ID"
         }
      ],
      "connection_points": [
         {
            "id": "connectionpointid1",
            "type": "virtual/physical port/nic or vpn endpoint ip"
         },
         {
            "id": "connectionpointid2",
            "type": "virtual/physical port/nic or vpn endpoint ip"
         }
      ],
      "pnfds": [
         {
            "pnfd": "reference of a pnfd of this ns"
         },
         {
            "pnfd": "reference of a pnfd of this ns"
         }
      ],
      "nsd_security": "MD5 hash of the NSD",
   },
}
```

Annex E OPENSTACK'S HEAT TEMPLATE

This annex shows an example of a full HEAT Template, as it is being used by the Orchestrator to allocate the needed infrastructure in the VIM, at the time of writing. The content of this kind of file will probably change until the end of the project.

```
---
heat_template_version: '2014-10-16' # HEAT version
description: 'forwarder, classificaion, DPI' # VNF description

resources: # In this section the resources are defined
  # Create the 4 types of networks: Management, Monitoring, Datapath and storage
  VTC_0:
    type: OS::Neutron::Net
    properties:
      name: mngt
  VTC_1:
    type: OS::Neutron::Net
    properties:
      name: monitoring
  VTC_2:
    type: OS::Neutron::Net
    properties:
      name: datapath
  VTC_3:
    type: OS::Neutron::Net
    properties:
      name: storage

  # Each network resource described above would need a correspoding subnet resource.
  # This information is not present in the VNFD
  # Example for Management network:
  mngt_subnet:
  type: OS::Neutron::Subnet
  properties:
   network_id: { get_resource: VTC_0 }
   cidr: 192.168.40.0/24
   dns_nameservers:
    - 8.8.8.8

  # Create the resource to upload the image to Openstack
  VTC_4:
    type: OS::Glance::Image
    properties:
      container_format: bare
      disk_format: qcow2 # Image type
      location: https://api.t-nova.eu/v1/nfstore/vnfs/123/image # URI to download
image

  # VNF Instance flavour (VDU flavour)
  VTC_5:
    type: OS::Nova::Flavor
    properties:
      disk: 30 # 30GB
      ram: 2048 # 2GB
      vcpus: 2

  # VNF instance (VDU)
  VTC_6:
    type: OS::Nova::Server
    properties:
      flavor:  {get_resource: VTC_5} # ID of the flavour after created in OpenStack
      image:  {get_resource: VTC_4} # ID of the image after uploaded to Openstack
```