



TNOVA

NETWORK FUNCTIONS AS-A-SERVICE
OVER VIRTUALISED INFRASTRUCTURES

GRANT AGREEMENT NO. 619520

Deliverable D3.41

Service Provisioning, Management and Monitoring – Interim

Editor Jordi Ferrer Riera (i2CAT)

Contributors J. Carapinha, P. Neves, M. Dias, B. Parreira (PTIn), M. McGrath, G. Petralia, V. Riccobene (INTEL), P. Paglierani (ITALTEL), 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 December 31st, 2015

Distribution PUBLIC (PU)

Executive Summary

The Orchestrator is the core sub-system of the T-NOVA Architecture, which has been mentioned several times. From now on, such core sub-system will be identified as TeNOR.

TeNOR is the name adopted in order to strengthen the potential outreach of the open-source system, once it is released during Y3. It was selected by the General Assembly of the project.

This deliverable provides an interim report of the developments done in terms of service management, monitoring, and provisioning in the T-NOVA orchestrator, in the context of task T3.4, which at the same time is integrating the developments produced within the rest of the work package three tasks.

In essence, this document contains a brief description of TeNOR, and all its internal components, i.e. micro-services. Due to its interim nature, not all the micro-services are finished, thus the manuscript only reports the current status of each one.

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

- **Micro-service based architecture** for the T-NOVA Orchestrator, developed within T3.4, and which integrates the interfaces (T3.1), the infrastructure repository (T3.2), and the service mapping (T3.3) outcomes.
- **TeNOR** as the elected branding name for the software system released within work package three.
- **Technologies** selected for the implementation of all the micro-services, the catalogues, the repositories, and all other tools composing the TeNOR system
- Current status of the data models utilized in the system
- **Sequence diagrams** of the most important TeNOR operations and interactions with other components, i.e. service provisioning and service monitoring

Table of Contents

1. INTRODUCTION	6
1.1. READING THIS REPORT	7
2. ORCHESTRATOR MICRO-SERVICES ARCHITECTURE	8
2.1. WHAT ARE MICRO-SERVICES	8
2.2. WHY MICRO-SERVICES	8
2.3. TENOR'S ARCHITECTURE.....	9
2.4. DETAILS ON IMPLEMENTATION OF EACH MICRO-SERVICE.....	9
2.4.1. <i>NS Manager</i>	10
2.4.2. <i>NS Catalogue</i>	12
2.4.3. <i>NS Provisioning</i>	12
2.4.4. <i>Service mapping</i>	14
2.4.5. <i>NS Monitoring</i>	14
2.5. SLA ENFORCEMENT.....	15
2.6. VNF MANAGER.....	16
2.7. VNF CATALOGUE.....	16
2.8. VNF MONITORING.....	17
2.9. VNF PROVISIONING.....	18
2.10. AUXILIARY SERVICES	18
2.10.1. <i>Management GUI</i>	18
2.10.2. <i>VNFD Validator</i>	18
2.10.3. <i>NSD Validator</i>	18
2.10.4. <i>HOT Generator</i>	19
2.10.5. <i>Gatekeeper</i>	19
2.10.6. <i>expression-evaluator</i>	20
3. CATALOGUES AND REPOSITORIES	21
3.1. CATALOGUES	21
3.1.1. <i>VNF catalogue</i>	21
3.1.2. <i>NS catalogue</i>	21
3.2. REPOSITORIES	21
3.2.1. <i>Provisioning repositories</i>	21
3.2.2. <i>Monitoring repositories</i>	22
4. PROVISIONING WORKFLOW	23
4.1. PROVISION OF A NETWORK SERVICE.....	23
5. MONITORING WORKFLOW	25
5.1. PARAMETER SUBSCRIPTION.....	25
5.2. PARAMETER READINGS	26
6. INTERACTING WITH TENOR.....	27
6.1. CONTEXT	27
6.2. THE WORKFLOW	27
7. CONCLUSIONS AND FUTURE WORK	29

8. ACRONYMS.....	30
9. REFERENCES	32
ANNEX A TENOR BRANDING	34
ANNEX B TENOR OVERALL ARCHITECTURE	36
ANNEX C TENOR NSD CURRENT VERSION.....	39
ANNEX D TENOR VNFD CURRENT VERSION	42
ANNEX E TENOR SUPPORTED REQUESTS (EXAMPLE).....	45

Figures

Figure 1-1: The T-NOVA Orchestrator Architecture, simplified.....	6
Figure 2-1: Gemfile: Dependencies declaration.....	10
Figure 2-2: NS Manager configuration file	11
Figure 2-3: The UML Sequence Diagram documenting the integration of the WICM with TeNOR.	13
Figure 2-4: TeNOR NS Provisioning micro-service internal structure	13
Figure 2-5: Service Mapping micro-service's UML Sequence Diagram.....	14
Figure 2-6: SLA enforcement sequence diagram	16
Figure 2-7: Registering a new service in Gatekeeper.....	19
Figure 2-8: An example of a workflow using the Gatekeeper micro-service.	20
Figure 4-1: Network Service provisioning's UML Sequence Diagram.....	23
Figure 5-1: Parameter subscription UML Sequence Diagram.	25
Figure 5-2: Parameter readings' UML Sequence Diagram.	26
Figure 6-1: The UML Sequence Diagram explaining how the Marketplace interacts with TeNOR.....	28
Figure 9-1: TeNOR Logo	34
Figure 9-2: TeNOR Logo Rationale	34
Figure 9-3: Postman Example for TeNOR Requests	45

1. INTRODUCTION

Following the previous WP3 deliverable objectives, the primary goal of Task 3.4, entitled Services Provisioning, Management and Monitoring, is to implement the T-NOVA Orchestrator, from now on called TeNOR¹. The TeNOR name was decided by the consortium GA in the meeting held in Limassol, Cyprus during Y2 by means of a closed voting as expressed in the Grant Agreement.

TeNOR is the system being developed in WP3, which will acquire and consolidate the outcomes of the different WP3 tasks, being T3.4 the major contributor to the system.

In terms of functionalities, as previously expressed in T-NOVA Deliverables D2.22 [1] and D3.01 [2], TeNOR is one of the core components of the architecture. It is responsible for Network Services (NSs) and virtual network functions lifecycle management operations over distributed and virtualized network/IT infrastructures. In fact, TeNOR is focused on addressing two of the most critical issues related to NFV operational environments:

- Automated deployment and configuration of NSs/VNFs;
- Control and monitoring of networking and IT resources for VNFs hosting

Figure 1-1 represents a simplified view of some of the T-NOVA Orchestrator (TeNOR) modules and the other sub-systems having interfaces with it. Refer to Deliverable D3.1 [3] for a complete specification of the interfaces utilized in order to communicate the different modules.

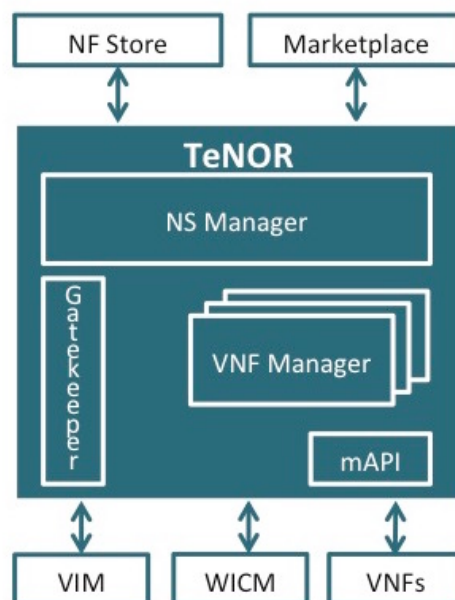


Figure 1-1: The T-NOVA Orchestrator Architecture, simplified.

¹ For further details on the TeNOR branding name refer to Appendix A.

1.1. Reading this Report

Considering the above depicted simplified architecture, this report contains the current implementation status of the different TeNOR components.

It is worth to remark at this point that this deliverable is meant to be an interim report, and thus it is not a software prototype release. In this sense, the report avoids to include specific software deployment instructions and/or software configurations, since this will be provided at the end of the WP3, during Y3.

Therefore, this report documents this work in the following chapters:

- Chapter 2, describes the **micro-services** of which TeNOR is composed, where the motivation on utilizing micro-services and their different functionalities is described;
- Chapter 3, describes the **Catalogues and Repositories** considered within TeNOR, where all the stored information of the system is permanently stored;
- Chapter 4, describes the designed **NS Provisioning workflows**; where the sequence diagram of the TeNOR components utilized for NS provisioning is presented;
- Chapter 5, similarly describes the designed **NS Monitoring workflow**; where the sequence diagram of the TeNOR components utilized for NS monitoring is presented;
- Chapter 6, with the high-level description of the **interaction** between the Marketplace and TeNOR.
- Chapter 7, with the **Conclusions**;
- Chapter 8, with the list of used **Acronyms**.
- Chapter 9, with the used **References**;

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

- Annex A, details on the **TeNOR branding** name and the new designed logo;
- Annex B, the detailed TeNOR updated **functional architecture**;
- Annex C and D, the current XSD and JSON formats for the **descriptors** (catalogues and validators);
- Annex E, **examples** on how different requests can be posted on the TeNOR system.

2. ORCHESTRATOR MICRO-SERVICES ARCHITECTURE

This section justifies our option for micro-services as a base for TeNOR's architecture, as well as it contains a detailed description of each micro-service, together with the current status of its implementation, since this deliverable is considered to be an interim report, and not the final outcome of Task T3.4.

2.1. What are micro-services

Micro-services [4] are *"an approach to **distributed systems** that promote the use of **finely grained services** with **their own lifecycles**, which collaborate together"*.

One micro-service usually provides one business-valued feature (as opposed to a number of them), interacting with other micro-services usually over HTTP and in a REST-based architecture style. Due to this more focused and simple approach, Micro-services are usually seen as being much more agile to develop and deploy. Furthermore, a single micro-service can evolve (in terms of scalability, for instance) independently of the other to which it provides or from which it consumes services.

2.2. Why micro-services

Micro-services [4] emerged when the previous related generation, Service-Oriented Architectures (SOA), began to fail to hold to its promises of making enterprise software development more agile [5]: basically, a service in a SOA does a lot of things, which implies that there are less of those services to manage, but evolving each one is slower than evolving a Micro-service, which does fewer things (preferably only one) but need more other micro-services to execute the same process.

Furthermore, the evolution from a purely micro-service based into a SOA based architecture is easier than the other way around.

Therefore, we are taking advantage of the faster approach that is developing micro-services, without compromising the future, in case a more traditional and SOA-like approach is needed. This justification is also provided in Deliverable D3.1 [3].

Furthermore, from a long-term perspective, the use of micro-services enables high modularity of the orchestration system. Although compromising overall complexity of the system in terms of integration, high modularity improves the potential survivability of the system, since it enables rapid updates and changes into individual micro-services, as long as the interfaces between them remain untouched.

Even, in the case of a sudden emergence of a novel technology for specific tasks, e.g. databases, the orchestration system could support a change in this component without the need to make any other modification to the rest of the micro-services.

2.3. TeNOR's Architecture

Initial TeNOR's architecture was contained in Deliverable D3.01 [2]. Since that, the architecture has been refined in order to accommodate the different workflows identified. TeNOR's architecture is depicted in detail Annex B.

In terms of software functionality, it is still a two-level functional architecture, split into a set of micro-services addressing the Network Service lifecycle management, and a set of micro-services addressing the Virtual Network Function lifecycle management. These sets of micro-services are complemented with a set of Catalogues and Repositories, fundamental for the permanent maintenance of information, as well as with a set of supporting micro-services, which provide enriched functionalities to TeNOR, such as logging, internal status monitoring, or even data model validations.

From the functional perspective, apart from the WICM component included in the orchestration system, there are no further major changes in comparison to the functional architectures presented both in WP2 and WP3 during Y1.

2.4. Details on implementation of each micro-service

This sub-section describes each of the designed and implemented micro-services, and its relationship with the remaining ones. Not all the micro-services are finished. Some of them are work-in-progress at the moment of writing this manuscript.

In general, the micro-services have been written using Ruby [6]. In detail, they run under Ruby 2.1.

Therefore, TeNOR utilizes also a set of Ruby Gems to be executed. Those Ruby Gems can be seen as dependencies of the micro-services required to run.

- **Sinatra**, Ruby framework
- **Thin**, Web server
- **json**, JSON specification
- **Sinatra-contrib**, Extensions for Sinatra
- **Nokogiri**, XML parser
- **JSON-schema**, JSON schema validator
- **Rest-client**, HTTP and REST client
- **Yard**, Documentation generator tool
- **rerun**, Restarts the app when a file change (used in development environment)

Figure below depicts a generic example of a micro-service declaring its dependencies of the different Gems.

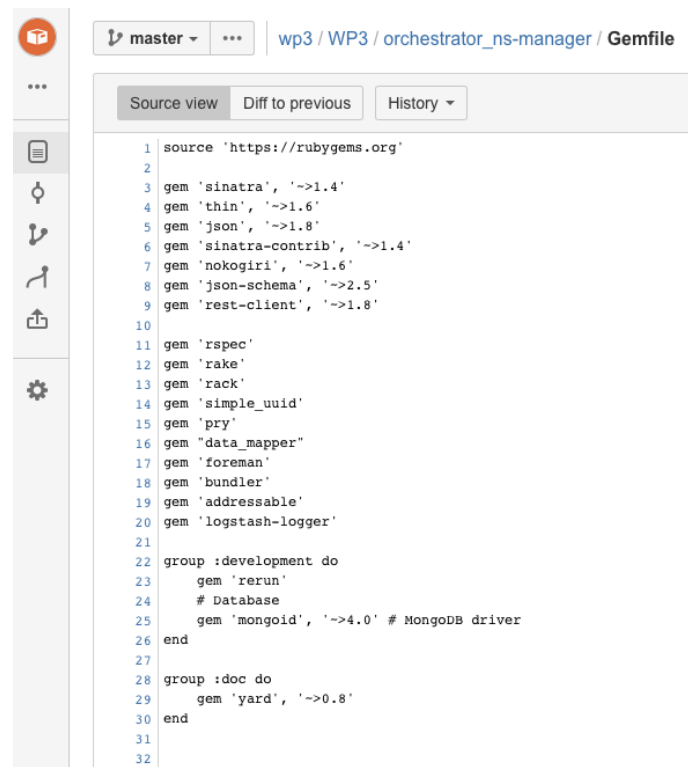


Figure 2-1: Gemfile: Dependencies declaration

In this case, it can be seen the set of required Gems and the versions in order to successfully run the given micro-service. In this sense, in order to download and install all the Gems required, it is needed to run the following command

```
bundle install
```

Each micro-service documentation is generated with yardoc, and can be executed with a rake task

```
rake yard
```

In order to visualize the documentation, there exists the need to start the server to browse the generated docs

```
yard server
```

Finally, in order to start the different micro-services, there exists the need to execute the following command

```
rake start
```

2.4.1. NS Manager

This micro-service is mostly a *façade* [7] to all the other micro-services: TeNOR's clients access it (see [8]) and then, depending on the request, it forwards it to the micro-service that will deliver the requested service.

This option was chosen mainly to simplify TeNOR clients' work, which use only one URI for most of the API calls. Details on the offered interface by the NS Manager can be found in deliverable D3.1 [3].

It is within the NS Manager that those users belonging to the Administrator role may configure all the micro-services the platform will provide.

Functionalities of the NS Manager are limited to act as an internal coordinator for the different micro-services as well as the coordinator of the interactions between TeNOR and the external systems (e.g. Marketplace). Figure below depicts an example configuration file for the NS Manager.

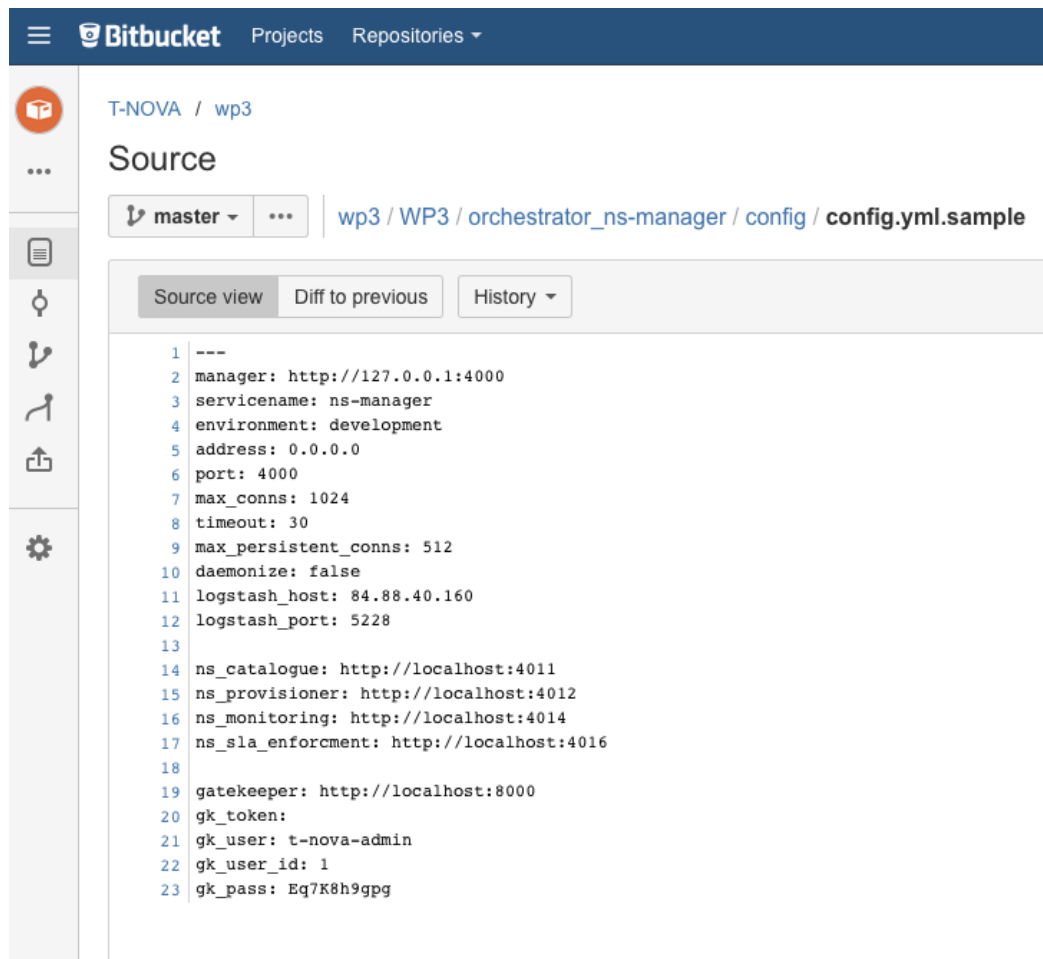


Figure 2-2: NS Manager configuration file

Apart from the generic information such as the URI where the interface is published, and the environment where the micro-service is deployed (i.e. production, or development), the configuration file also contains information about the logstash address, the gatekeeper credentials, and the URIs and ports of the micro-services registered into the NS Manager.

Further details of the current status of the implementation can be found in the private repository. During Y3 this micro-service will be released into the open GitHub project account.

http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator_ns-manager

2.4.2. NS Catalogue

This micro-service manages the Catalogue of Network Services, receiving requests from the Marketplace for both registration of a new NS and updating already registered micro-services.

NS Descriptors are described in JSON, and are stored in a **MongoDB** [9] (document-oriented) NoSQL database, taking advantage of the dynamics that this kind of NoSQL databases possesses in managing documents with a flexible structure.

Currently, at the time of writing the deliverable, the NS descriptor is not on its final version. It contains most of the fundamental parts to enable simple provisioning, and monitoring integrated within all the TeNOR micro-services.

It is worth to mention that only valid NS Descriptors go into the Catalogue.

The version utilized by the Catalogue and the validator micro-service can be found in Annex C.

2.4.3. NS Provisioning

This micro-service is responsible for the provisioning of the different Network Services available in the NS Catalogue.

From the general perspective, the NS Provisioning is composed of the following phases (refer to Figure 2-3):

1. If required, TeNOR creates a connectivity resource in the WICM, by means of sending the required information, such as the identifier, the Network Access Point, or the descriptor). It receives back the corresponding VLAN identifiers for both the ingress and egress traffic of the connection created.
2. TeNOR starts the provisioning process, which is materialized by means of a HEAT orchestration template to be sent to the VIM, which is the responsible to actually instantiate the different components expressed in the template.
3. After that, TeNOR updates the connectivity resource created before in order to adapt the traffic redirections to the instances created (both VMs and virtual networks) by the previous call.

This process enables the completely automated end-to-end service provisioning including one or more NFVI-PoPs and the transport network managed by the WICM, although, in this phase, we have considered only very simple Virtual Links (VLs) and Virtual Network Function Forwarding Graphs (VNFFGs). These simple versions of VLs and VNFFGs will be improved, when we interact with the Service Forwarding Graph (SFG) that is being developed in the WP4 ([10]).

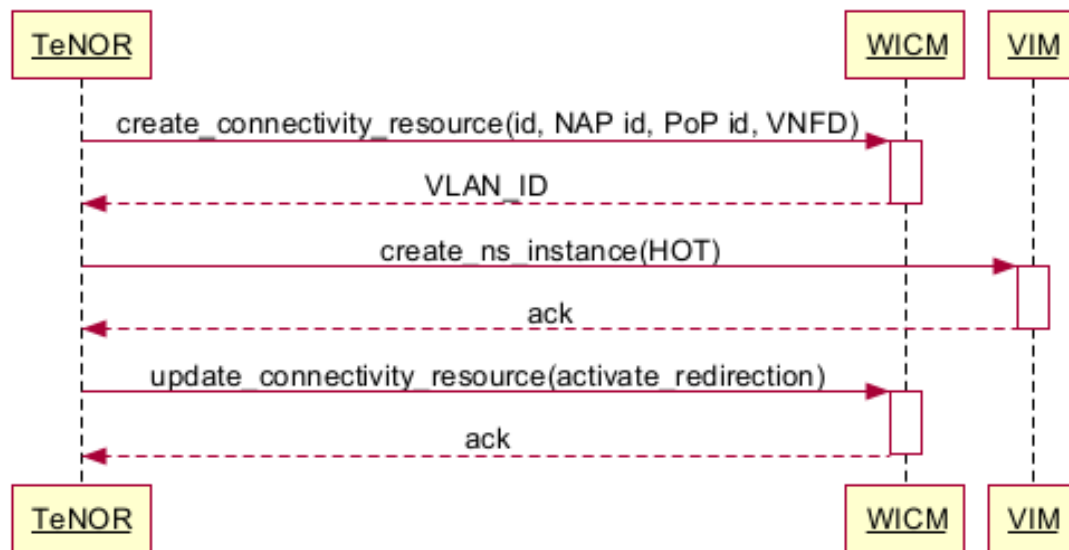


Figure 2-3: The UML Sequence Diagram documenting the integration of the WICM with TeNOR.

However, from the detailed perspective, the second step (*create_ns_instance(HOT)*) involves not only one but several core micro-services in the process. Details of the designed **NS Provisioning** workflow and the interactions between the different TeNOR micro-services are detailed in Section 4.

Detailed status of the micro-service can be found in the corresponding repository.

http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator_ns-provisioning

Figure below depicts an example of the folder structure for the micro-service

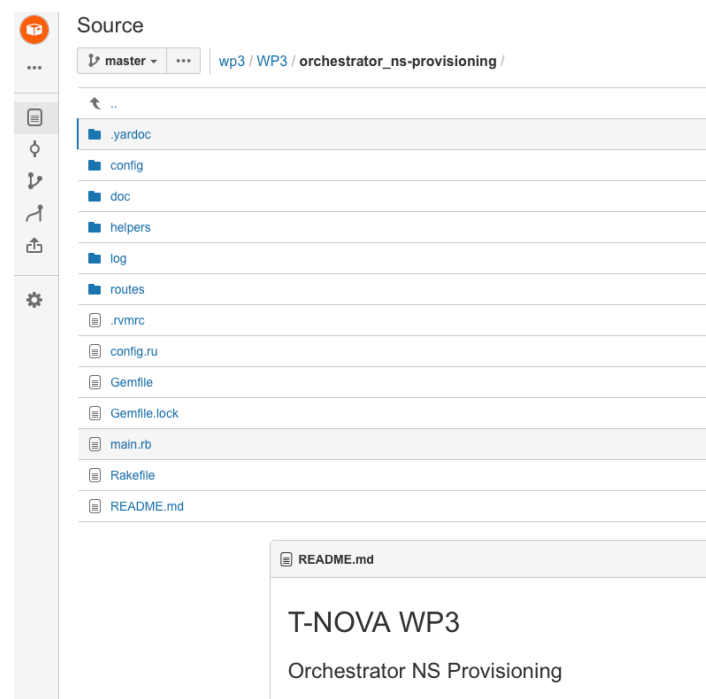


Figure 2-4: TeNOR NS Provisioning micro-service internal structure

2.4.4. Service mapping

This micro-service encapsulates the optimization algorithms that have been designed and developed in Task 3.3.

Its integration with the remaining components of TeNOR is shown in Figure 2-5.

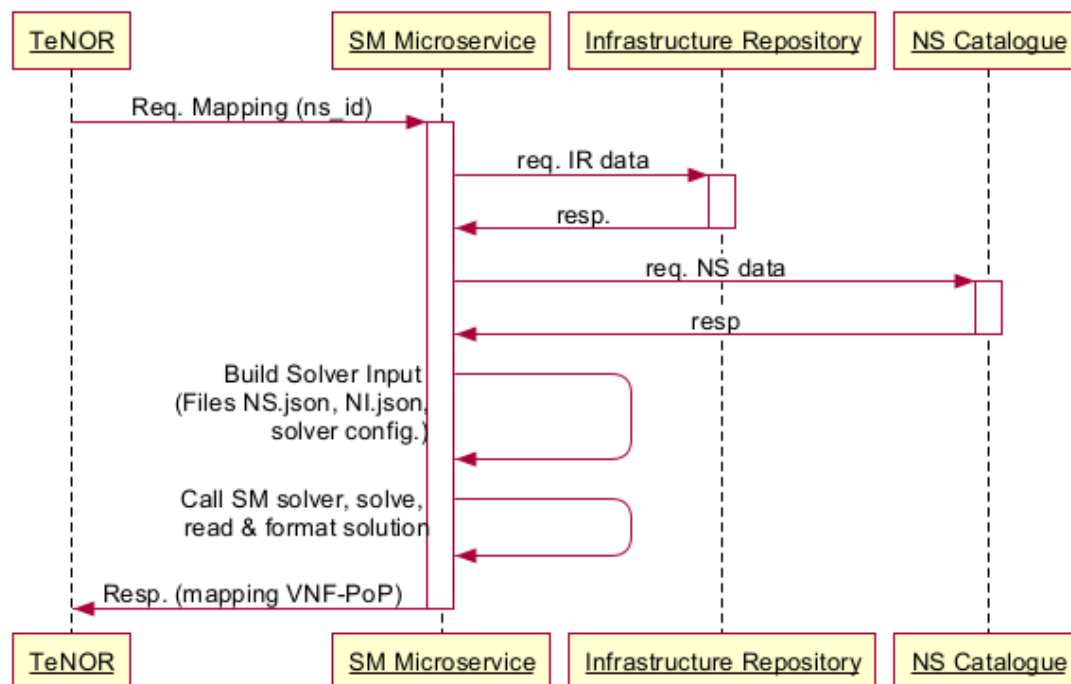


Figure 2-5: Service Mapping micro-service's UML Sequence Diagram.

Basically, every time there is a new request to instantiate a Network Service, the Service Mapping is called, to determine which infrastructure to be used for that instance, i.e. on which infrastructure resources to deploy the VNFs composing the NS.

In order to achieve this, the Service Mapping requires communication with both the Infrastructure Repository and the NS Catalogue. Basically, it uses the information available in the Infrastructure Repository (to know which resources are free or being used), as well as the NS Catalogue (to know exactly which VNFs the Network Service is composed of).

It then creates the internal data structures needed to calculate the optimal solution and calculates this solution, returning an ordered list of possible PoPs (first is the best) where that service can be deployed.

Finally, some performance indication about the service mapping is stored in the micro-service; e.g. received requests, successful mappings, performance time, etc. This information is showed later on the Management UI.

2.4.5. NS Monitoring

This micro-service is responsible of managing the monitoring of the different service metrics within the TeNOR system. Considering the characteristics of the NSs, which

are composed of VNFs, and due to the large amount of data to be monitored (i.e. transmitted from the VIM towards the Orchestrator and stored in the data repositories), we have designed a publish-subscribe mechanism.

Thus, for each NS instance that is provisioned, the NS monitoring subscribes the monitoring parameters defined in the NSD. Further details on the complete sequence diagram for NS monitoring are provided later in Section 5.

Finally, in order to store all the acquired and calculated data, the NS Monitoring is responsible of filling and managing the so-called *ns-monitoring repository*. The ns-monitoring repository is basically a Cassandra database storing all the information related to the subscribed metrics for each one of the instantiated NSs.

Considering Cassandra is an excellent tool for time series-based databases, the schema of the database utilized for storing the values of the metrics can be found in the repository in

```
http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator_ns-monitoring-repository/db/schema.txt
```

In essence, the model is as follows

```
CREATE KEYSPACE tnova_monitoring WITH REPLICATION = { 'class' : 'SimpleStrategy',  
'replication_factor' : 1 };  
  
use tnova_monitoring;  
  
CREATE TABLE nsmonitoring ( instanceId text, date int, metricName text, value text,  
primary key(instanceId, metricName, date));
```

There is one common keyspace, with replication factor, automatically managed by the Apache Cassandra toolset. Within this space, used by only one service provider, as T-NOVA requirements were based on single-provider, each entry represents a value of a given metric, identified by the ID of the service instance (not the ID of the NSD), the name of the metric, the value of the metric, and the timestamp / date of the metric.

Thus, the monitoring stores metric value entries, which can later on be easily queried and joined by means of the service instance ID and the metric name.

2.5. SLA Enforcement

At the time of writing this deliverable, this micro-service is the only one on this list that is still being designed and developed, which means will be integrated in the final release of TeNOR (due in Y3). It will concentrate the actions TeNOR will have to take to keep the agreed SLAs, as indicated by the Marketplace when instantiating the service.

Taking into consideration the SLA thresholds, the actions that could be taken imply for example being able to scale a VNF composing the NS, scale the connections between VNFs, or even scaling both.

Figure below depicts the internal SLA enforcement mechanism sequence diagram.

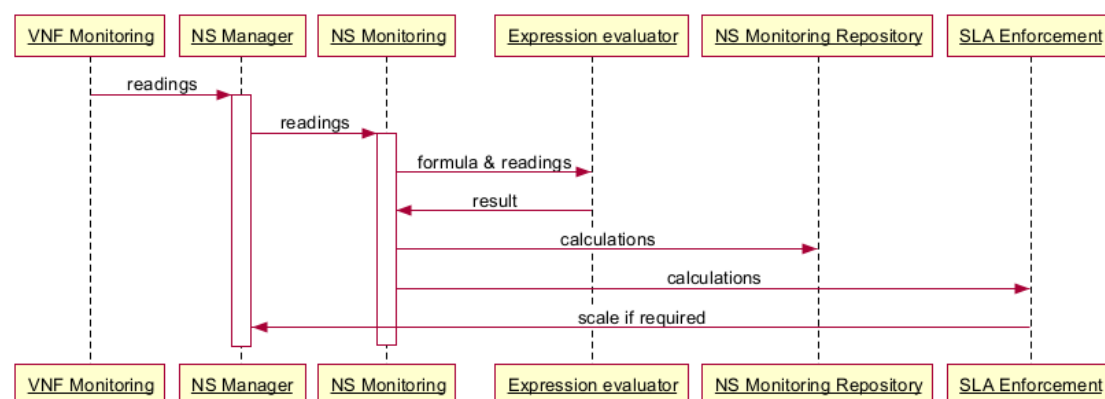


Figure 2-6: SLA enforcement sequence diagram

Basically, the values of the metrics get the NS Monitoring micro-service through the NS Manager. The expression evaluator is responsible to analyse the formulas and provide the result for all the metrics at the service level. These obtained metrics (calculations in the diagram) are then stored in the corresponding data repository, and analysed by the SLA enforcement component.

In detail, SLA and scaling will be included in the last WP3 deliverable in Y3, deliverable D3.4, which will contain the final TeNOR prototype release.

The current status of the micro-service can be found in the software repository

<http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator-sla-enforcement>

2.6. VNF Manager

Like the NS Manager (see above), this micro-service acts as mostly a *façade* [7] to all the micro-services that deal with a VNF at a time.

The VNF Manager passes the right micro-service the requests it receives, and orchestrates their answers back to the requester.

Besides being a façade, the VNF Manager has been designed with the possibility of being replaced by the specific VNF Manager that may come with certain VNFs, a feature where the micro-service based architecture will shine. This feature will be developed until the end of the task, in April, 2016.

The current status of the micro-service can be found in the software repository.

http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator_vnf-manager

2.7. VNF Catalogue

This micro-service manages the Catalogue of Virtual Network Functions (VNFs), receiving VNF Descriptors (VNFDs) from the NF Store for both registration of a new VNF and updating an already registered VNF. The interaction of updating a VNF

creates a versioned document, this way it is possible to keep tracking of what were the changes that were introduced to update an existing VNFD.

VNFDs are JSON format documents that are stored in a **MongoDB** [8] (document-oriented) NoSQL database, taking advantage of the dynamics that this kind of NoSQL databases possesses in managing documents with a flexible structure.

At the time of writing this deliverable, the VNFD is not on its final version. It contains most of the fundamental parts to enable provisioning and monitoring, integrated with all the other relevant TeNOR micro-services.

Only valid VNF Descriptors go into the Catalogue and this validation is done by the VNFD Validator micro-service.

As an example, the source code below depicts the VNF model ruby class we utilize for the VNF Catalogue micro-service.

```
module BSON
  class ObjectId
    def to_json(*args)
      to_s.to_json
    end

    def as_json(*args)
      to_s.as_json
    end
  end
end

class Vnf
  include Mongoid::Document
  include Mongoid::Timestamps
  include Mongoid::Pagination
  include Mongoid::Versioning

  field :name, type: String
  field :vnf_manager, type: String
  field :vnfd, type: Hash

  validates :name, :vnfd, :presence => true
end
```

The current model can be seen in Annex D.

The current status of the micro-service can be found in the software repository

http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator_vnf-catalogue

2.8. VNF Monitoring

This micro-service is responsible for managing the monitoring at the VNF level.

Whenever a new VNF instance is created, as a consequence of a request for a new NS instance coming from the Marketplace, the relevant (to the service) monitoring parameters are passed to the **VNF Monitoring** (from the **NS Monitoring** micro-service, see above) and are subscribed in the **VIM Monitoring Framework** (see [9]).

As a consequence of a successful subscription, readings of that parameter start reaching TeNOR. These readings are stored in the **VNF Monitoring Repository**, the related NS instance identified, and the reading is passed to the **NS Monitoring** micro-service (see above).

See also Section 5 about the monitoring workflow below.

The current status of the micro-service can be found in the software repository

http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator_vnf-monitoring

2.9. VNF Provisioning

This micro-service is responsible for the provisioning of the different VNFs that compose a given NS, that are available in the VNF Catalogue.

The provisioning of each VNF is done by dynamically generating a HEAT Orchestration Template (HOT), using the **HOT Generator** micro-service (see below). This HOT file is then sent to the VIM, which is the responsible to actually instantiate the different components expressed in the template.

The current status of the micro-service can be found in the software repository

http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator_vnf-provisioning

2.10. Auxiliary services

This section describes the micro-services that were needed but that do not directly represent a business value.

2.10.1. Management GUI

TeNOR provides a Graphical User Interface (GUI) tool that allows authorized user(s) to manage it. This is being a work in progress, which is expected to be completed by the end of T3.4 during Y3 of the project.

2.10.2. VNFD Validator

This micro-service is called by the **VNF Catalogue** micro-service whenever a new VNF, or a new version of one existing, is provided by the **NFStore**.

It uses a JSON schema to validate the content of the VNF Descriptor (in JSON). TeNOR also supports VNF Descriptors in XML, using an XML schema for the validation.

2.10.3. NSD Validator

This micro-service is called by the **NS Catalogue** micro-service whenever a new Network Service, or a new version of one existing, is provided by the **Marketplace**.

Like the **VNFD Validator** (see above), it uses a JSON schema to validate the content of the NS Descriptor (in JSON). TeNOR also supports NS Descriptors in XML, using an XML schema for the validation.

2.10.4. HOT Generator

This micro-service receives, from the VNF Provisioning, the VNFD relative to a single VNF along with the T-NOVA deployment flavour and generates the Heat Orchestration Template (HOT) that is necessary for the VIM (OpenStack).

2.10.5. Gatekeeper

This micro-service implements TeNOR's Authentication and Authorization mechanism that allows simple actions such as user-registration/authentication and inter-services authorization based on tokens.

The registration of a new service is illustrated in Figure 2-7 (registering a new user is a similar flow).

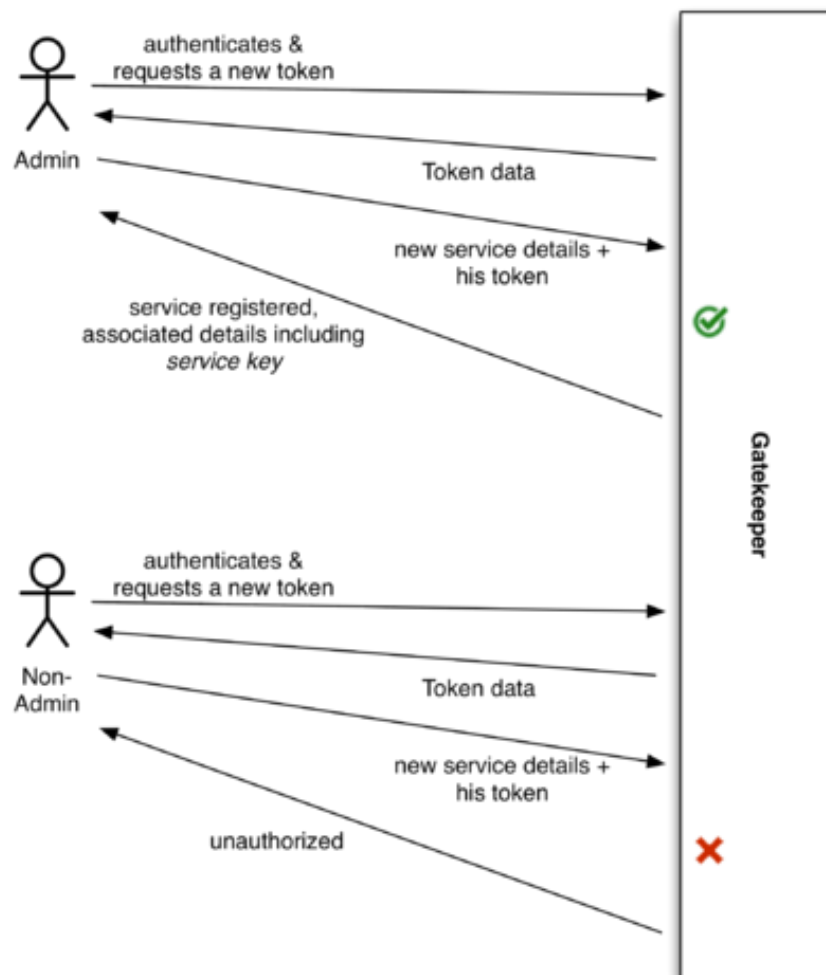


Figure 2-7: Registering a new service in Gatekeeper.

The workflow of authenticating and authorizing the usage of a service is shown in Figure 2-8.

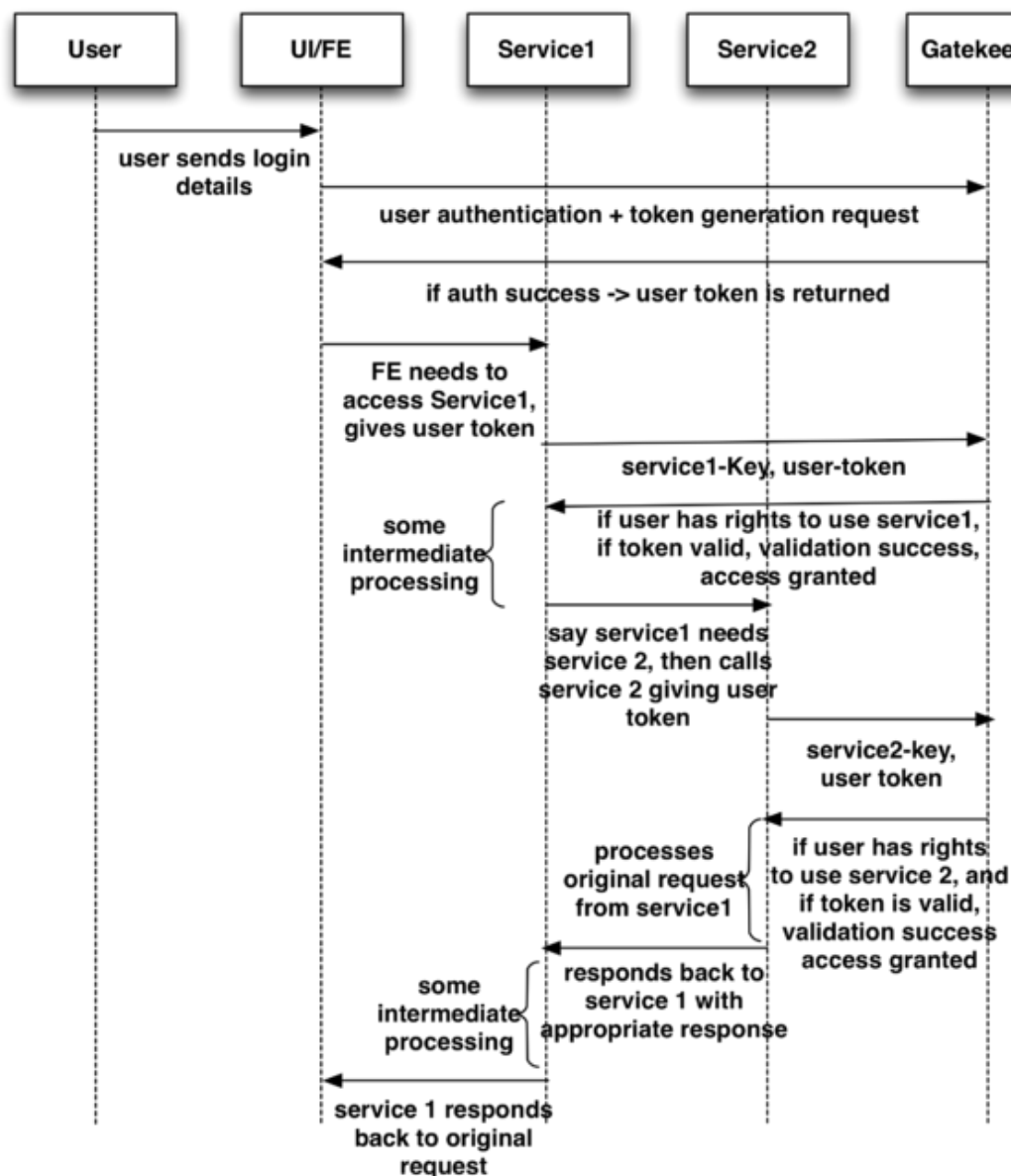


Figure 2-8: An example of a workflow using the Gatekeeper micro-service.

2.10.5.1. Using a ruby gem to access the micro-service

In order to further abstract the usage of the micro-service, we have built a ruby gem, **sinatra-gkauth-gem**.

Using this generated gem the gatekeeper functionalities can be directly utilized by the Sinatra framework, which enables the REST-based communication of the different micro-services.

2.10.6. expression-evaluator

This micro-service is called by the **NS Provisioning** micro-service with the mathematical expression that allows the calculation of the monitoring parameters at the NS level. The expression is validated and 'compiled' to be later used in calculating the values of those parameters, whenever new readings are provided.

3. CATALOGUES AND REPOSITORIES

This section describes the options taken when designing TeNOR's catalogues and repositories.

We separate the two because they are very different in nature, both in the kind of data that they have to store, and the frequency of readings and writings, among other characteristics.

3.1. Catalogues

Catalogues hold data describing the main entities that can be instantiated.

3.1.1. VNF catalogue

This catalogue holds the successfully on-boarded VNF descriptors, submitted by the NFStore.

3.1.2. NS catalogue

This catalogue holds the successfully on-boarded NS descriptors, submitted by the Marketplace.

3.2. Repositories

Repositories hold data resulting from instantiating the entities that are part of the Catalogues.

We have chosen two different kinds of databases for TeNOR's repositories.

3.2.1. Provisioning repositories

Provisioning repositories hold all the information resulting from a successful provisioning of a Network Service and all its related components: VNFs, VNFFGs, etc.

For this kind of repository we initially used the **PostgreSQL** [13] relational database. The reason for this option was that the information we had to store was distributed into related tables, the optimal use case for a Relational Database.

Since service instances are not created every second, writings in these repositories are fairly distributed in time, while readings are rather frequent.

However, we finally opted for utilizing a NoSQL database, i.e. **MongoDB**, due to the constant changes in the NSDs and VNFDs during the development stage, which implied several changes in the relational database model.

The source code of the model follows, although the complete repository code can be found in the repository.

```
module BSON
class ObjectId
```

```

def to_json(*)
  to_s.to_json
end
def as_json(*)
  to_s.as_json
end
end
end

module Mongoid
  module Document
    def serializable_hash(options = nil)
      h = super(options)
      h['_id'] = h.delete('_id') if(h.has_key?('_id'))
      h
    end
  end
end

class NsInstance
  include Mongoid::Document
  include Mongoid::Timestamps
  include Mongoid::Attributes::Dynamic
  field :vnfs, type: Array

  field :nsr_instance, type: Array
  field :ns_id, type: String
  field :status, type: String
  field :version, type: String
  field :vnfrs, type: Array
  field :marketplace_callback, type: String
end

```

3.2.2. Monitoring repositories

Monitoring repositories hold all the information resulting from monitoring the provisioned Network Services

For this kind of repository we have used the **Cassandra** [10] NoSQL (column oriented) database. The reason for this option was that the information we had to store (monitoring data) is better modelled as columns (as opposed to rows in a SQL database). The information to be retrieved is a series of (monitoring) values, which, if stored in a SQL database, would imply first to retrieve all the rows and then choose the relevant column. By using a column-oriented database, data is store in columns, therefore accelerating its retrieval.

These repositories will have to support a high frequency of writings, with monitoring data coming in to be stored. This data will arrive in small chunks (the value read and some more metadata). Readings, on the other hand, will be less frequent, but made in chunks: typically, a set of (parameter) readings between two time stamps.

4. PROVISIONING WORKFLOW

This section describes the provisioning workflow whenever a new instance of a Network Service is requested.

4.1. Provision of a Network Service

Provisioning a Network Service implies provisioning every VNF that compose that service, as well as the connections between them.

Figure 4-1 shows a UML Sequence Diagram of a typical NS provisioning request.

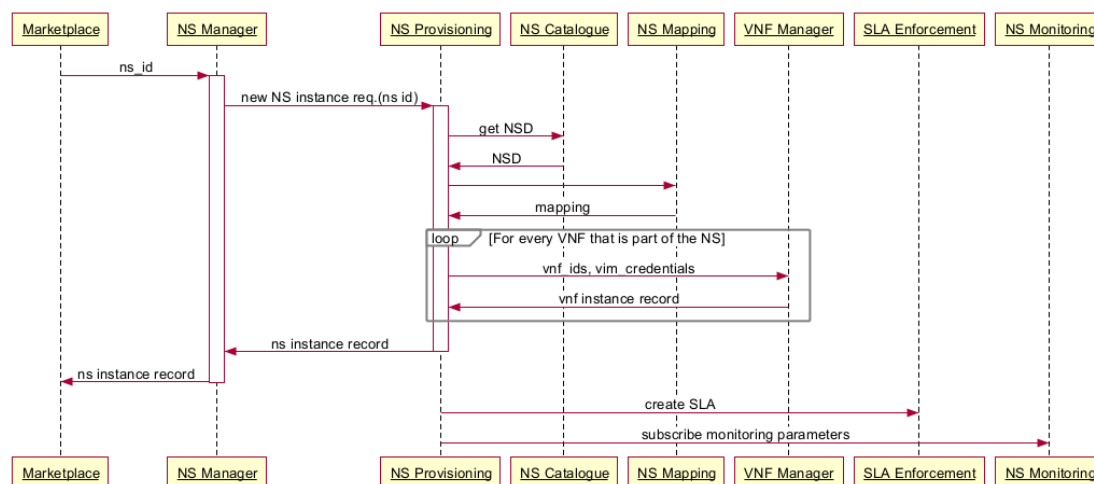


Figure 4-1: Network Service provisioning's UML Sequence Diagram.

The sequence of messages is the following:

1. Whenever a **new NS instantiation request** reaches TeNOR coming from the **Marketplace**, it is passed to the **NS Provisioning** micro-service (see 2.4.3. NS Provisioning, above);
2. The **NS Provisioning** micro-service grabs the NS id from that request and grabs its data from the **NS Catalogue** micro-service (see 2.4.2. NS Catalogue above);
3. With that information, merged with the request information, the **NS Mapping** micro-service (see 2.4.4. Service mapping, above) is called; which returns a list of possible PoP's where to allocate the required resources, sorted from best to worst;
4. With the PoP location, a loop is started and each VNF that is part of the NS is then passed to the VNF Manager, for it to provision them. With the instantiation data is received, the VNF repositories are filled, as well as the NS Repositories;
5. For each successful NS instantiation request an SLA is created (through the **SLA Enforcement** micro-service, see 2.5. SLA Enforcement, above) and the necessary monitoring parameters are subscribed (through the **NS Monitoring** micro-service, see 2.4.5. NS Monitoring, above).

After this the **Marketplace** has all the data on the new NS instance available, namely in terms of monitoring data (see below).

5. MONITORING WORKFLOW

This section describes the monitoring workflow that allows TeNOR to dynamically, for each NS instance that is provisioned, follow its behaviour and adjust it according to the agreed SLA. Furthermore, monitoring data is also made available to the **Marketplace**.

Since monitoring data, collected at the VNF component level, can be overwhelming in volume, we opted for a publish/subscribe schema, such that only the required monitoring parameters are received.

TeNOR is also interested in monitoring the NS instance performance end-to-end, given that the (customer) agreed SLA is at that level. We therefore map the collected, VNF Component level monitoring data to service level monitoring data in this workflow.

The remaining of this section further details the two main processes of this workflow: the parameter subscription and the parameter readings.

5.1. Parameter subscription

As described above, given the (high) volume of monitoring data that is usually available, we have decided to constrain a bit the amount that we want to receive. Therefore, we have opted to use a publish-subscribe mechanism to restrict that amount. This section describes the subscription part.

The UML Sequence Diagram describing the subscription of a monitoring parameter is shown in Figure 5-1.

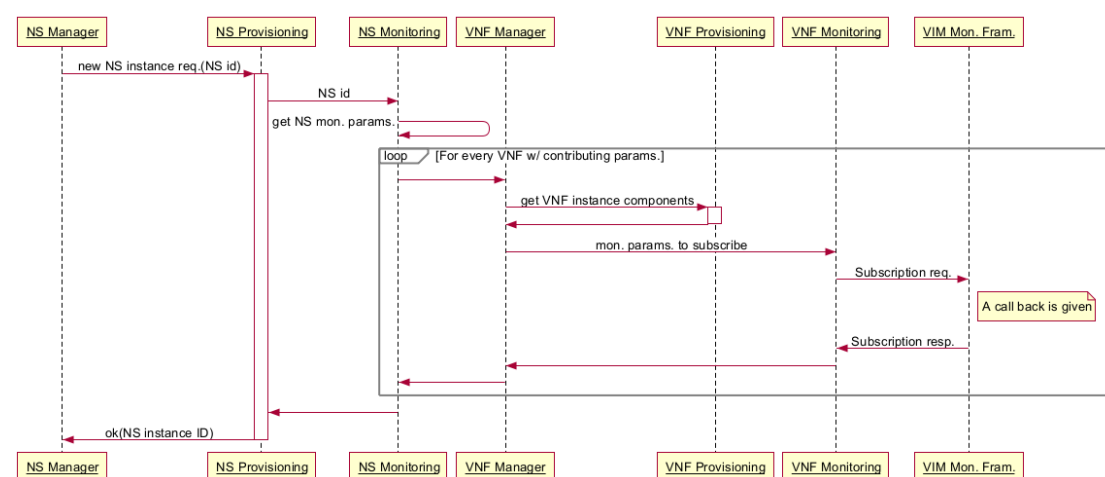


Figure 5-1: Parameter subscription UML Sequence Diagram.

The sequence of messages is the following:

1. For each NS instance that is successfully provisioned (see section 4. Provisioning workflow, above) the **NS Provisioning** micro-service (see 2.4.3. NS Provisioning, above) asks the **NS Monitoring** micro-service (see 2.4.5. NS

- Monitoring, above) to subscribe the monitoring parameters that are specified in the **NSD**;
2. Every VNF instance that is part of this NS instance is then passed to the **VNF Manager** micro-service (see 2.6. VNF Manager, above)
 3. The **VNF Manager** finds out which parameters are needed and asks the **VNF Monitoring** micro-service (see 2.8. VNF Monitoring, above) to subscribe them on the **VIM Monitoring Framework** [9].

This finishes the subscription sequence. It's only from this moment on that the subscribed parameters may reach TeNOR and start being processed. The subscription is a callback URL that will later be called whenever there's a reading to report.

5.2. Parameter readings

This section explains how (monitoring) parameter readings reach TeNOR and how they get transformed from VNF-based reading into a NS-based reading.

The UML Sequence Diagram that shows this workflow is shown in Figure 5-2.

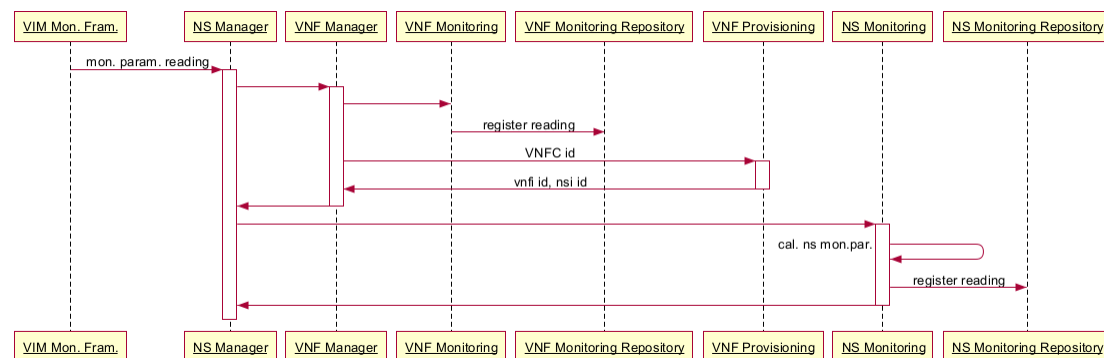


Figure 5-2: Parameter readings' UML Sequence Diagram.

The sequence of messages is the following:

1. Whenever the **VIM Monitoring Framework** has a parameter reading to report, it calls the URL provided in the subscription phase (see sub-section 5.1. Parameter subscription, above) with that data;
2. The **NS Manager** micro-service, being a *façade*, receives the request and passes its data to the **VNF Manager** micro-service, another *façade* (this one exists due to the possibility of the VNFM being proposed together with a (set of) VNF(s)), which passes to the **VNF Monitoring** micro-service (see 2.8. VNF Monitoring, above) for storing in the **VNF Monitoring Repository**;
3. From the VNF Component ID that comes with the data reading, the **VNF Manager** gets the **NS ID** and the **VNF ID** from the **VNF Provisioning** micro-service, which is passed back to the **NS Manager**;
4. The **NS Manager** micro-service passes these IDs to the **NS Monitoring** micro-service, which allows it to store the reading data (and the corresponding relationships) in the **NS Monitoring Repository**.

The step following these, which is still part of the on-going work, will be to check the relevant SLA and see if the collected data is within what is acceptable, or start a **migration** or **scaling** process.

6. INTERACTING WITH TeNOR

This section briefly describes how TeNOR can be accessed from the Marketplace, one of the systems in T-NOVA's architecture.

6.1. Context

In T-NOVA there is a constant feedback between the Marketplace and TeNOR. The Marketplace needs an orchestrator in the lower layers to provision, run and manage the network services that are defined in its catalogue since it is totally unaware of the infrastructure below, but TeNOR is. In exchange, TeNOR provides back information related to the running instances, updated status and monitoring data so the Marketplace can show its users how the instances are running and bill them accordingly.

6.2. The workflow

The workflow between the Marketplace and TeNOR, illustrated in Figure 6-1, is as follows.

A Function Provider (FP) that wants to upload a VNF to T-NOVA accesses the Marketplace and defines it by means of a form based on the ETSI VNFD information model: this will be the T-NOVA VNFD, that is stored, along with the VNF images in the NF Store, where TeNOR parses and runs tests over the VNFD and marks the new VNF as 'available' on success.

The NSs the provider offers in the Business Service Catalogue are created by means of the combination of the available functions in the NF Store. The form to define the NS in the Marketplace follows the ETSI NSD specifications as well. Again, the Marketplace interacts with TeNOR to store a copy of this NSD and it is checked for potential availability of resources.

Once a Customer selects a NS from the catalogue in the Marketplace, the first step is to connect it to the local network and to do so, the Customer provides a connection point, that it's sent to the Orchestrator along with the id of the selected service as an instantiation request. This request is processed by TeNOR, which decides the amount of resources to be assigned to provision the NS and where, to be close enough to the connection point provided by the Customer in order to meet the SLA.

After the mapping of the NS to the infrastructure is done, the NS components (VNFs) are deployed in the assigned POPs and linked as specified in the forwarding graph of the NSD. At this point, the NS is ready to start running and the monitoring starts as well.

Each VNF provides monitoring of its own metrics and those are gathered and aggregated by TeNOR with two goals: Feed the Marketplace with it so the billing of the services and functions is accurate and to guarantee the instances are running as expected according to the indicated SLA parameters.

In case this is not happening, a readjustment is needed and the NS needs to be scaled out.

All this underground functioning is transparent to the Customer, which only sees that the platform is stable and the NSs run smoothly thanks to TeNOR efforts.

The communication that ends this process is the service termination request, which is sent from the Marketplace to TeNOR on user command: all resources are released and the Marketplace can bill the Customer and send a revenues report to the Provider.

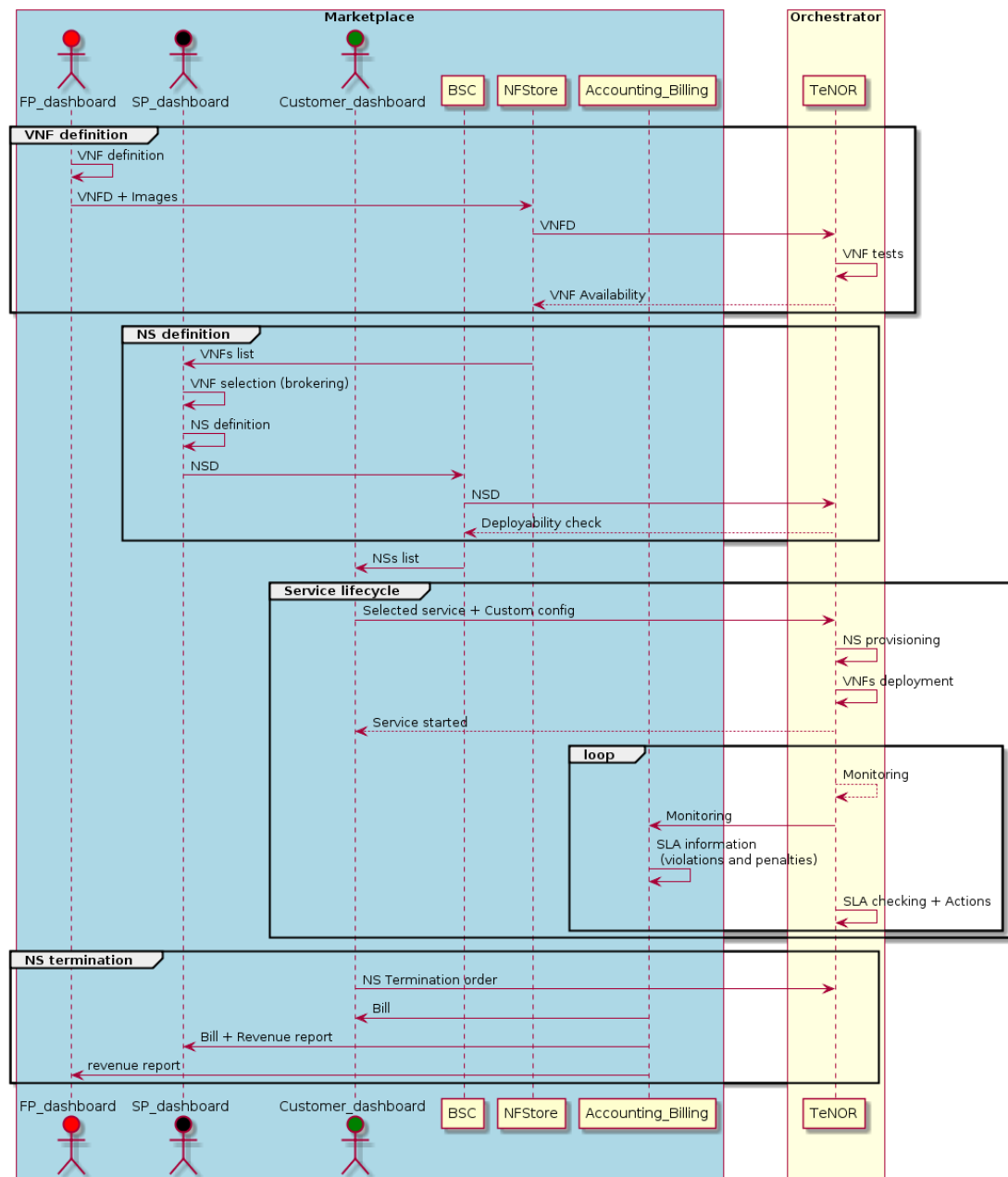


Figure 6-1: The UML Sequece Diagram expalining how the Marketplace interacts with TeNOR.

7. CONCLUSIONS AND FUTURE WORK

The Orchestrator is the central sub-system of the whole T-NOVA system. From now on, the system is known as TeNOR.

The interim report provides a snapshot of the current development activities within work package three. In essence, the report describes the micro-services based architecture of TeNOR, and the function assigned to each one of the micro-services. It initially introduces the technologies utilized to implement all the functionalities, catalogues, and repositories. Furthermore, the

However, this being an interim report, we would like to emphasize that there are still developments under way, namely:

- Scaling an end-to-end service: letting a VNF scale out or in automatically is a problem that has been solved and is currently part of the most advanced Datacentres. A different problem, however, is scaling an end-to-end Network Service that may be composed by more than one interconnected VNFs. We are still working and experimenting for solving this problem;
- Supporting specific VNF Managers: as precluded by ETSI [11], each VNF may bring its own VNF Manager, which (as previously reported, see [2, 12]) brings a new set of problems to the infrastructure owner. TeNOR was designed to support this, but we would like to try to effectively on-board a specific VNF Manager that could exist in parallel with the generic one;

Moreover, open-source software release and integration of TeNOR in the T-NOVA pilot test-bed in Athens is planned to happen in the third year of the project.

8. 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

9. REFERENCES

- [1] G. Xilouris, et. al., "T-NOVA Deliverable D2.22 Overall System Architecture and Interfaces"
- [2] J. Bonnet et. al., "T-NOVA Deliverable D3.01 Interim Report on the Orchestrator Platform Implementation"
- [3] J. Bonnet et. al., "T-NOVA Deliverable D3. 1 Orchestrator Interfaces"
- [4] S. Newman, Building Microservices, O'Reilly, 2015;
- [5] M. Richards, "Microservices vs. Service-Oriented-Architecture", O'Reilly, 2015
- [6] <https://www.ruby-lang.org/> Programming language
- [7] E. Gamma et. al., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994;
- [8] <https://www.mongodb.org/>
- [9] G. Gardikis et. al., "T-NOVA Deliverable D4.42 Monitoring and Maintenance -- Final"
- [10] <http://cassandra.apache.org/>
- [11] "TCP vs. UDP" (http://www.diffen.com/difference/TCP_vs_UDP)
- [12] G. Banga , F. Dougli, M. Rabinovich, "Optimistic Deltas for WWW Latency Reduction"
(https://www.usenix.org/legacy/publications/library/proceedings/ana97/full_papers/banga/banga_html/usenix.html)
- [13] <http://www.postgresql.org>

Annex A TeNOR BRANDING

During Y3 the orchestrator source code will be completely released. In order to provide it with fully-functional identity outside the T-NOVA realm, the consortium decided to provide a branding name to the system. Thus, the T-NOVA Orchestrator, after a voting process held during the GA in Limassol, Cyprus, decided TeNOR as the name for the T-NOVA Orchestrator.

As part of the branding activities, during this period within T3.4 we designed the following logo, to be used in public presentations, and public releases of the system.



Figure 9-1: TeNOR Logo

The term **TeNOR**, is mostly known for being a type of classical male singing voice whose vocal range is one of the highest of the male voice types. The analogy of the logo with an actual TeNOR is depicted in Figure below.



Figure 9-2: TeNOR Logo Rationale

However, in systemic functional linguistics, the term **TeNOR** refers to the participants in a discourse, the relationships to each other, and their purpose [see [https://en.wikipedia.org/wiki/Tenor_\(linguistics\)](https://en.wikipedia.org/wiki/Tenor_(linguistics))]. In fact, looking at the proposed

architecture, the different micro-services, and all the relationships between them, this meaning of the TeNOR term perfectly applies to the system. The TeNOR term refers to the distinct micro-services composing the orchestrator, the relationships between them (as defined by the already presented sequence diagrams and materialised through the interfaces defined in D3.1), and their purpose, being the main purpose of the system:

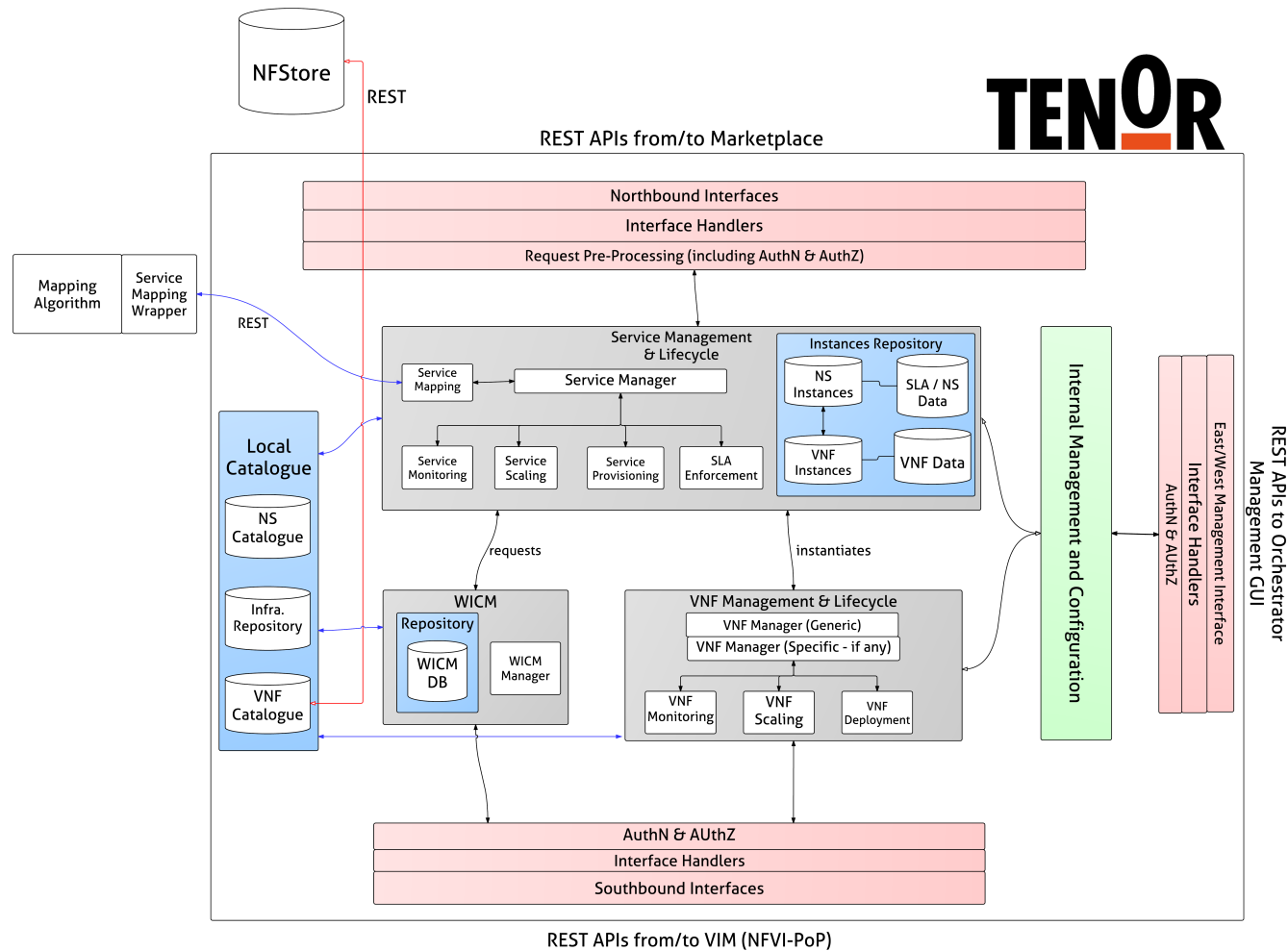
- to automate the deployment and configuration of NSs/VNFs; and
- to control and monitor network and IT resources devoted to VNF hosting

Annex B TeNOR OVERALL ARCHITECTURE

The following figure depicts the updated functional architecture of the TeNOR system.

The architecture is a two-layer based functional architecture. On the one hand, the upper layer, which is mainly devoted to the service lifecycle management; on the other hand, the lower part, devoted to virtual network function lifecycle management, as described in D3.01 [2].

It is important to mention that the figure contains a functional schema of the architecture. The software architecture is not exactly the same. For example, the interfaces are depicted as a single functional entity both on the north and the southbound levels of TeNOR, while in reality every micro-service holds an operational interface (utilized either internally or externally by other components).



Annex C TeNOR NSD CURRENT VERSION

The following listings represent the .xsd and .json versions of the descriptors utilized by both the service catalogues and the validator.

It is worth to mention that this is not the final version, which will be commonly created between the Marketplace and the TeNOR system.

Additionally, the catalogues will inherently support any change in the descriptor due to its NoSQL nature, where the inserted tuples do not necessarily need to hold the same internal structure.

The basic NSD holds in essence the following structure, as initially identified in Deliverable D3.01 [2]

- Identifier of the NS
- Vendor
- Version
- Published to customers
- Availability
- List of VNFDs composing the service
- Virtual network function forwarding graph
- Virtual links description
- Lifecycle events
- VNF dependencies, in case there is any specific dependency in terms of instantiating order between the VNFs.
- List of monitoring parameters for the service
- SLA parameters

.xsd version

The .xsd version can be found also in the stash repository

http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator_nsd-validator/assets/schemas/nsd.xsd

As an example, we include the complete document here in the Appendix.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="nsd">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="id" type="xs:integer" />
        <xs:element name="vendor" type="xs:string" />
        <xs:element name="version" />
        <xs:element name="published_to_customers" />
        <xs:element name="availability" />
        <xs:element name="vnfd" maxOccurs="unbounded" />
        <xs:element name="vnffgd" maxOccurs="unbounded" />
        <xs:element name="vld" maxOccurs="unbounded" />
        <xs:element name="lifecycle_events" maxOccurs="unbounded">
        <xs:element name="vnf_dependencies" />
        <xs:element name="monitoring_parameters" minOccurs="0" maxOccurs="unbounded">
```

```

<xs:complexType>
  <xs:sequence>
    <xs:element name="id" type="xs:integer" />
    <xs:element name="name" type="xs:string" minOccurs="0" />
    <xs:element name="description" type="xs:string" minOccurs="0" />
    <xs:element name="definition" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="service_deployment_flavour" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id" type="xs:integer" />
      <xs:element name="flavour_key" type="xs:string" />
      <xs:element name="constituent_vnf" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="vnf_reference" type="xs:integer" />
            <xs:element name="vnf-flavour_id_reference" type="xs:string" />
            <xs:element name="redundancy_model" type="xs:string" />
            <xs:element name="affinity" type="xs:string" />
            <xs:element name="capability" type="xs:string" />
            <xs:element name="number_of_instances" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="sla_specification" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="sla_id" type="xs:integer" />
      <xs:element name="service_deployment_flavour_reference" type="xs:string" />
      <xs:element name="assurance_parameters" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="assurance_parameter" type="xs:integer" />
            <xs:element name="limit" type="xs:string" />
            <xs:element name="value" type="xs:string" />
            <xs:element name="unit" type="xs:string" />
            <xs:element name="violation" type="xs:string" />
            <xs:element name="penalty" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="billing_model" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="id" type="xs:integer" />
            <xs:element name="type" type="xs:string" />
            <xs:element name="period" type="xs:string" />
            <xs:element name="price" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="auto_scale_policies" minOccurs="0" maxOccurs="unbounded" />
<xs:element name="connection_points" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id" type="xs:integer" minOccurs="1" maxOccurs="1" />
      <xs:element name="type" minOccurs="1" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>

```



```
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="pnfd" minOccurs="0" />
<xs:element name="nsd_security" />
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

.json version

The .json version can be found in the stash repository. For the sake of readability of the document it is not included here.

```
http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator_nsd-
validator/assets/schemas/nsd_schema.json
```

Annex D TeNOR VNFD CURRENT VERSION

The following listings represent the .xsd and .json versions of the descriptors utilized by both the VNF catalogue and the validator.

It is worth to mention that this is not the final version, which will be commonly created between the VNF Developers, the NFStore, and the TeNOR system.

Additionally, the catalogue will inherently support any change in the descriptor due to its NoSQL nature, where the inserted tuples do not necessarily need to hold the same internal structure.

The VNFD is mainly composed of the following structure:

- Identifier of the VNF
- Name of the VNF
- Description of the VNF
- Provider information
- Type of the VNF
- List of VDUs composing the VNF
- Virtual links description
- List of deployment flavours
- List of lifecycle events
- Billing model

.xsd version

The .xsd version can be found also in the stash repository

http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator_vnfd-validator/assets/schemas/vnfd.xsd

As an example, we include the complete document here in the Appendix

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="vnfd">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="xs:byte" name="id"/>
        <xs:element type="xs:string" name="vendor"/>
        <xs:element type="xs:string" name="descriptor_version"/>
        <xs:element type="xs:string" name="version"/>
        <xs:element name="vdus">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="vdu" maxOccurs="unbounded" minOccurs="1">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element type="xs:byte" name="id"/>
                    <xs:element type="xs:string" name="vm_image"/>
                    <xs:element type="xs:string" name="computation_requirement"/>
                    <xs:element name="virtual_memory_resource_element">
                      <xs:complexType>
                        <xs:simpleContent>
                          <xs:extension base="xs:byte">
                            <xs:attribute type="xs:string" name="unit" use="required"/>

```

```

        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
</xs:element>
<xs:element type="xs:string" name="lifecycle_events"/>
<xs:element type="xs:byte" name="high_availability"/>
<xs:element type="xs:byte" name="scale_in_out"/>
<xs:element type="xs:string" name="OpenStack_Flavour"/>
<xs:element name="vnfc">
    <xs:complexType>
        <xs:sequence>
            <xs:element type="xs:byte" name="id"/>
            <xs:element name="connection_point" maxOccurs="unbounded" minOccurs="1">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element type="xs:byte" name="id"/>
                        <xs:element type="xs:string" name="virtual_link_reference"/>
                        <xs:element name="virtual_network_bandwidth_resource">
                            <xs:complexType>
                                <xs:simpleContent>
                                    <xs:extension base="xs:byte">
                                        <xs:attribute type="xs:string" name="unit" use="required"/>
                                    </xs:extension>
                                </xs:simpleContent>
                            </xs:complexType>
                        </xs:element>
                        <xs:element type="xs:string" name="type"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="cpu">
    <xs:complexType>
        <xs:sequence>
            <xs:element type="xs:string" name="cpu_instruction_set_extension"/>
            <xs:element type="xs:string" name="cpu_model"/>
            <xs:element type="xs:byte" name="cpu_core_reservation"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="memory">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="memory_parameter">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element type="xs:byte" name="number_of_large_pages_required_per_vdu"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="hypervisor">
    <xs:complexType>
        <xs:sequence>
            <xs:element type="xs:string" name="hypervisor_type"/>
            <xs:element type="xs:string" name="hypervisor_version"/>
            <xs:element type="xs:string" name="hypervisor_address_translation_support"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:sequence>

```

```

    </xs:complexType>
  </xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="deployment_flavour">
  <xs:complexType>
    <xs:sequence>
      <xs:element type="xs:byte" name="id"/>
      <xs:element type="xs:byte" name="flavour_key"/>
      <xs:element name="constituent_vdu" maxOccurs="unbounded" minOccurs="1">
        <xs:complexType>
          <xs:sequence>
            <xs:element type="xs:byte" name="vdu_reference"/>
            <xs:element type="xs:byte" name="number_of_instances"/>
            <xs:element type="xs:byte" name="constituent_vnfc"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element type="xs:string" name="auto_scale_policy"/>
<xs:element type="xs:string" name="manifest_file"/>
<xs:element type="xs:string" name="manifest_file_security"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

```

.json version

The .json version can be found in the stash repository. For the sake of readability of the document it is not included here.

http://stash.i2cat.net/projects/TNOV/repos/wp3/browse/WP3/orchestrator_vnfd-validator/assets/schemas/vnfd_schema.json

Annex E TeNOR SUPPORTED REQUESTS (EXAMPLE)

TeNOR supports a huge amount of requests, as defined in their interfaces. Considering this document as an interim report, we provide in this Annex an example of some of the HTTP methods (REST-based calls) that can be performed over the TeNOR system.

In this example, the methods are forwarded from the Postman application, provided by Google Chrome.

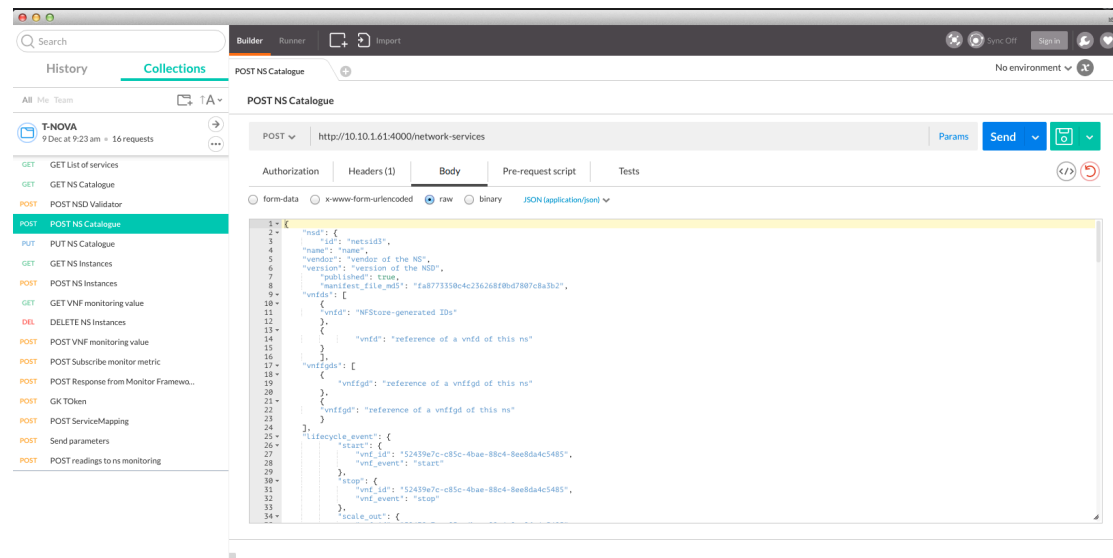


Figure 9-3: Postman Example for TeNOR Requests

Figure depicts the POST operation over the NS Catalogue. Basically, it can be seen the NSD utilized to onboard a new NS into the corresponding catalogue.