D4.3 Service Platform First Operational Release and Documentation

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

Deliverable	D4.3 Service Platform First Operational Release and Documentation
Workpackage	WP4 Resource Orchestration and Operations repositories
Due Date	June 30th, 2017
Submission Date	June 29th, 2017
Version	0.1
Status	To be approved by EC
Editor	José Bonnet (AlticeLabs)
Contributors	José Bonnet, Alberto Rocha, Miguel Mesquita (AlticeLabs), Santiago Rodríguez (Optare), Felipe Vicens (ATOS), George Xilouris, Stavros Kolometsos, Christos Sakkas (NCSRD), Dario Valocchi (UCL), Theodore Zahariadis, Panos Trakadas, Panos Karkazis (SYN), Thomas Soenen (IMEC), Sharon Mendel-Brin (Nokia), Michael Bredel (NEC), Muhammad Shuaib Siddiqui, Daniel Guija (i2CAT), Manuel Peuster, Sevil Dräxler, Hadi Razzaghi Kouchaksaraei (UPB)
Reviewer(s)	George Xilouris (NCSRD), Geoffroy Chollon (THALES), Panos Trakadas (SYN)

Keywords:

Service Platform, orchestrator, gatekeeper, VIM

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

Executive Summary:

This deliverable is the last one to be produced in SONATA's Work Package 4, documenting the work done in the Service Platform since D4.2 was published (December, 2016).

During this period, we have concentrated our efforts in making the SONATA Service Platform a **carrier-grade solution**, capable of providing platform owners (not necessarily Telecom Providers) an adequate support for the extremely demanding scenarios of the **5G landscape**. Besides new or improved features, we have also introduced adequate **configuration mechanisms** that allow for a smoother adoption of a SONATA Service Platform into an existing infrastructure. We believe that our **micro-services based design and implementation** puts this platform in the front-line of **flexible solutions**, where the adoption of any monolithic product will prove to be too costly in engineering services.

Supporting the ability to **securely give back to the developer**, in near real time, **monitoring data** on his/her function is another feature that is a critical differentiator of our platform in the market. There are small glitches that can only occur in production, no matter how many hours of tests are made to new or existing features. And only with such approach we can allow developers to **close the cycle** between **idea**, **implementation** and **deployment**. And we even collect (internal) **key performance indicators** to demonstrate that.

By using a **DevOps** approach to build the platform itself, we keep its code and architecture really flexible and ready to be evolved.

We are therefore certain that the SONATA Service Platform can be a sound bet from those wanting to keep the pace of evolution towards 5G, without having to throw away years and years (and Euros!) of development and operations history, tools, processes, etc.

Contents

List of Figures

List of Tables

1 Introduction

This report is the last one for on the **Work Package 4, Resource Orchestration and Operations Repositories**, of the SONATA project.

In it we are describing the new or changed features since our last report ([7]). For example, the **authentication and authorization of users and micro-services**, which was mostly ready when we wrote D4.2, in which its design was presented, only now is fully integrated with the remaining Gatekeeper modules. This was also the case of the **KPIs Management** module, the **Licence Management** module, etc. Naturally, this integration has led to the need for some adjustments in the **GUI** (e.g., to show the KPIs) and the **BSS** (e.g., to support a simple Licence Store). We have also introduced a new module, the **Rate Limiter**, which allows for a more secure usage of the Service Platform, e.g., by avoiding Denial-of-Service [10] attacks. This module, together with the Licence Management module, also allows the implementation of monetisation mechanisms for the Service Platform API. This is a crucial aspect of the IT world, given the clear current trend of making services and data available through APIs; service and data owners may not always be open to just give away their assets for free, so any platform or system that provides an API must be prepared for such scenarios.

The **MANO Framework** got several updates, the most relevant ones being a clearer separation of the code between the **Service Lifecycle Management** component and the **Function Lifecycle Management** one. This clearer separation meets the ETSI's design of a two level design, comprising the **Network Service Manager** (NSM) on top and the **Virtual Network Function Manager** (VNFM) at the bottom.

Catalogues and Repositories got smarter in this new version: when a **package, service or function deletion** is requested, a validation on the possible re-use of the service or function is made, thus avoiding the introduction of inconsistencies in the Catalogues. This module was also the one we have chosen to make a reference implementation of the **micro-service authentication/authorisation**.

Monitoring now takes into consideration scalability aspects that we may have a multiple VIM infrastructure, thus enhancing the collection, processing and storage of data in a way that the remaining modules can efficiently access it. Monitoring is in this version able to provide a function's developer with **monitoring data**, either through a **web-socket** (current data, in near real time) or a common JSON/REST API call (past data), thus allowing for a **much more agile** way to improve the service's or function's performance. Finally, the developer is given the opportunity to modify monitoring metrics and alerting rules in real-time.

The **Infrastructure Abstraction** can now support a better and more flexible configuration of **WIMs** connecting the different **VIMs** of the whole virtual infrastructure, including the definition of a 'mocked' WIM (for simple scenarios with only one VIM, see the next paragraph on Configuration). To perform the complex flow of **VNF scaling** a OpenStack Mistral Workflow Engine is used in a completely standalone mode, decoupled from Openstack. The infrastructure abstraction layer was enhanced with an additional client. This Mistral Client implemented in Java is the foundation of a client that will be developed by OpenStack community members as part of OpenStack4J ([5]).

We have gathered all the platform's **configuration** strategy in a single section, thus making it easier for the developer who wants to download and deploy the SONATA Service Platform to understand the different aspects that are configurable and the impacts of each configuration. The

platform has two kinds of configuration parameters, one that defines each module's **default values** (e.g., IP addresses, ports, user names, passwords, etc.) and another kind that allows the switching off specific features or even entire modules (for example, when there is no need for managing users through the User Management module in a simple deployment).

2 Gatekeeper

The SONATA Gatekeeper got upgraded in most of its modules since D4.2 was written. Those that haven't been updated are mentioned only in the aspects that have changes, e.g. how they got integrated with the GK API.

The Gatekeeper modules that had some change since D4.2 are described next. Since the **Security Gateway** (see Section 2.6) is now part of the Gatekeeper, we describe the updates (namely the new **Rate Limiter** module) in a sub-section of this section.

2.1 Gatekeeper API

The Gatekeeper API has evolved towards the carrier-grade level we wanted to achieve in its last version: we have added user verification on the available operations, we are limiting the rate of usage of the API, etc.

This sub-section describes those upgrades. More detailed documentation can be found in the wiki of the GitHub's Gatekeeper repository.

2.1.1 User management

User management now allows for:

- **User registration**, the first step in any interaction a platform user must have with the SONATA SP. Registration can be done either in:
  - the **GUI** (see Section 2.4), for **developers** and **admins**;
  - the **BSS** (see Section 2.5), for **customers**.
- **User login** and **logout**, performed in the GUI and the BSS and also in the SDK. When the user login is done through the SDK, it uses the returned token to identify the user in sub-sequent calls;
- **User public key update**, allowing the user to update his/her public key, used in the platform to verify package's integrity.

2.1.2 Micro-services management

Micro-services management now allows for:

- **Micro-services registration**, to authenticate micro-services that are part of the Service Platform's architecture;
- **Micro-services login**, in which the micro-service gets a token with limited validity in time, that should be renewed periodically.

For the sake of simplicity and resource restrictions, we have opted to register, as an example, only one of the micro-services, the **Catalogue** (see Section 3). We consider that the impact of this decision in the overall platform security is negligible, since all micro-services are deployed behind the Service Platform's **Security Gateway** (see [7]).

2.1.3 Packages Management

Package management now allows for

- **User validation** (the package owner) is now performed for the following features:
  - Package on-boarding, now including the registration of the owner of this package in the Catalogue (see Section 3);
  - Package file downloading, which succeeds only if:
    1. that package is public;
    2. the package is private and the user requesting the package either:
       a) is the package owner;
       b) has a 'reuse' licence;
  - Package descriptor downloading, which succeeds only if:
    1. that package is public;
    2. the package is private and the user requesting the package either:
       a) is its owner;
       b) has a 'reuse' licence;
- **Package validation**, using the same micro-service developed for the SDK (see [8]);
- **Package signing**, also using the same micro-service developed for the SDK (see [8]);
- **Licence validation**, with each developer's **Licence Store** (a URL included in the descriptor of a private package) being queried whenever a private service is to be instantiated or the packages it is included in is requested to be downloaded (for reuse).

2.1.4 Services Management

Service management now allows for

- **Service owner** registration in the Catalogue (see Section 3), which is the user that has successfully submitted a package containing the service;
- **User validation** (the service owner) is now performed for the following features:
  - Service instantiation requests, which will only succeed if either:
    1. the service has been on-boarded in a **public** package;
    2. the service has been on-boarded in a **private** package and the user requesting the instantiation has an **instantiation** licence (checked in the **Licence Store**, see Section 2.5);
  - Service descriptor downloading, which will only succeed if either:
    1. the service has been on-boarded in a **public** package;
    2. the service has been on-boarded in a **private** package and the user requesting the downloading has a **reuse** licence (checked in the **Licence Store**, see Section 2.5);

2.1.5 Functions Management

Function management now allows for

- **Function owner** registration in the Catalogue (see Section 3), which is the user that has successfully submitted a package containing (a service referencing) the function;
- **User validation** (the function owner) is now done for the following features:
  - Function descriptor downloading, which will only succeed if either:
    1. the function has been on-boarded in a **public** package;
    2. the function has been on-boarded in a **private** package and the user requesting the downloading has a **reuse** licence (checked in the **Licence Store**, see Section 2.5);
  - Function instance monitoring, which will only succeed if either:
    1. the function has been on-boarded in a **public** package;
    2. the function has been on-boarded in a **private** package and the user requesting the monitoring has a **reuse** licence (checked in the **Licence Store**, see Section 2.5);
- **Function instance monitoring**, which can be divided in:
  - **Synchronous data**, with currently obtained monitoring data being provided through a **Web-Socket**;
  - **Asynchronous data**, with past obtained monitoring data being provided through a **file**;

2.1.6 Records Management

Record management now allows for

- **User validation** (the service or function owner) is now performed for the following features:
  - Service records downloading, which will only succeed if either:
    1. the service has been on-boarded in a **public** package;
    2. the service has been on-boarded in a **private** package and the user requesting the records, either:
       * is the **service owner** (i.e., has successfully on-boarded a package containing the service);
       * has a **reuse** licence (checked in the **Licence Store**, see Section 2.5);
  - Function records downloading, which will only succeed if either:
    1. the function has been on-boarded in a **public** package;
    2. the function has been on-boarded in a **private** package and the user requesting the records, either:
       * is the **function owner** (i.e., has successfully on-boarded a package containing the function);
       * has a **reuse** licence (checked in the **Licence Store**, see Section 2.5);

2.2 User Management

This section describes the enhancements and new features that have been implemented in the User Management module for the SONATA final release version. The User Management was a new component introduced in SONATA release version 2.1 and initially described in the Deliverable D4.2. This component is responsible for the authentication and authorization processes, managing the identity and the access to the Service Platform (SP) and controlling the permissions of the platform users and micro-services, allowing or denying the requested actions.

More detailed information about this component can be found in section 3.2 of Deliverable D4.2 [7].

2.2.1 Architecture updates

The User Management module's architecture has been slightly updated. It is currently composed by three key different sub-components, which are the Adapter, Keycloak and support Database. The following Figure 2.1, presents the current architecture for the User Management in the Service Platform.

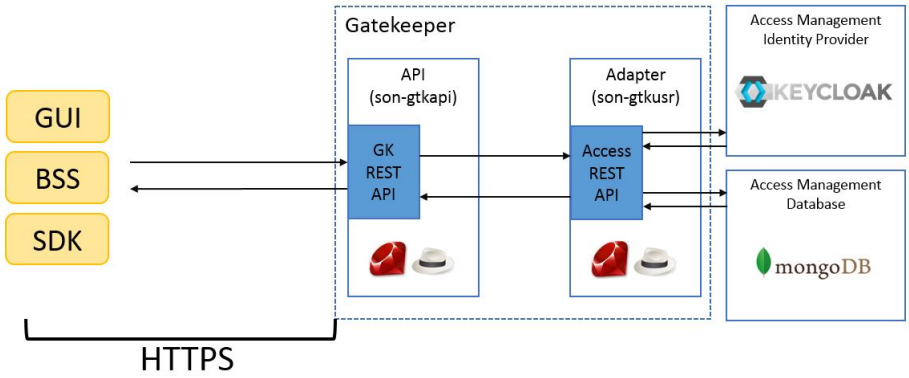

Figure 2.1: User Management architecture

Adapter (`son-gtkusr` micro-service):

This sub-component was already present in the initial version of the User Management architecture. It is a key sub-component that acts as an interface between the Gatekeeper API and the Keycloak tool framework. The Adapter is the main manager behind the Keycloak, as it is the administrator component responsible for running any operation related with the User Management component. In addition, the Adapter is implemented as a micro-service inside the Gatekeeper which exposes a well-defined REST API to the Gatekeeper API in order to perform authentication, authorization and identity management of the Service Platform. The main update in this sub-component is that it is now separated from the Keycloak tool framework, keeping communication through their Admin REST API.

Keycloak (`son-keycloak` micro-service):

This sub-component is the Identity and Access Manager tool, which in this updated version is now outside of the Adapter micro-service to be placed in its own micro-service `son-keycloak`.

This provides isolation for each sub-component and the advantage of dedicating granular containers for each micro-service, following the Gatekeeper architecture pattern, and allowing a better scalability. The Keycloak tool remains the main sub-component behind the User Management, as it is responsible for granting authentication and authorization capabilities to the Service Platform. It exposes an Admin REST API which is managed by the Adapter (`son-gtkusr` micro-service) sub-component.

MongoDB Database:

This is a new sub-component added to the User Management architecture. This MongoDB database enables support to store new meta-data to the User and Service Accounts. It introduces the possibility to enhance User Management functionalities and is currently responsible for functions such saving additional User Account information as the user's Public Key, and Protected Resources schemas, which states the SP resources requiring authorization checks in order to be accessed, the required roles and applied policies. In addition, the User Management communicates to the MongoDB through a REST API, using a model schema to enforce validation rules to the stored JSON data. These models are presented in the following list.

Users Account extra data:

```
field :username, type: String
field :id, type: String
field :public_key, type: String
field :certificate, type: String
```

Protected Resources data:

```
field :resource_owner_name, type: String
field :role, type: String
field :resources, type: Array
field :policies, type: Array
```

The defined model schemas are kept simple, as one of the advantages of MongoDB is the dynamic structure of the stored data. Furthermore, the User Management module exposes a REST API to manage the contents of the MongoDB.

2.2.2 Enhancements

This section lists the enhancements that have been implemented to the User Management. These enhancements basically add new options and parameters to existing features and they focus on improving some of the internal mechanisms, mainly the authorization mechanism used to process authorization checks and evaluations.

New Admin user type:

For the latest release, a new User type (`userType`) is being introduced to the already existing types. Along with the Developer and Customer User types, the new Admin User type comes into scene in order to provide to certain end-users the capabilities to manage the Identity and Access Manager of the Service Platform. This User will be able to add, update and remove roles, permissions and groups inside the User Management, plus other functionalities provided by

Keycloak. This type of user was found missing in previous versions and it was required given that Keycloak only creates an administrator user by default on deployment time, but this user was not meant to be used as a framework manager but for deployment processes. The Admin user type is attached to the administrator `realm-admin` role. As the other user types, the Admin type can also be combined with the other user types.

User Public Key and Certificate:

With the introduction of a support database as an architecture improvement, new user parameters can be stored within the User Management system. These parameters are currently focused on User Account meta-data, which currently are able to save the Public Key and Certificate for end-users of Developer user type. This enables a new feature in the Service Platform to receive digital signatures along with Package files from the SDK tools. These digital signatures are Hashes that can be used to evaluate the authenticity of the Packages using the author or owner Public Key. The Gatekeeper API is responsible for requesting user's Public Keys to the User Management in order to perform evaluations.

New Authorization model and mechanism:

A new Authorization model is being introduced to the final release version of the User Management. This new model comes to replace the model that was present since the initial release version. The old model presented a data structure based on Service Platform resources exposed by micro-services registered to the platform. This old model was loaded to the Service Platform on deployment time and it remained static. This model also was limited in terms of roles and permissions, and it also remained static with no option to updates. The new Authorization model grows in complexity but keeps the same resource design that was presented in the former model. It currently enables more granularity when exposing resources from micro-services, meaning that a micro-service can now expose diverse resources with different requirements attached to them in order to be accessed. In this new model, each micro-service can define different resources that can be accessed by users or micro-services. The micro-service defines many aspects for each resource, and the roles and permissions associated to each resource. Then, each associated permission is based on different roles defined by the User Management in order to grant access to the associated resource. The model is now defined in a JSON format (instead of a static YAML file in the User Management configuration). Thanks to the introduced MongoDB database, a new collection will be responsible for saving this new JSON model, allowing multiple definitions, making it also more dynamic and configurable.

An example of the new Authorization model is presented next, based on the SP Catalogue component and the "service" resource. As the example shows, the micro-service "catalogue" exposes a resource, in this case "services", which references the Network Service Descriptors in the SP Catalogue. In order to restrict access to this resource, the model defines the associated permissions to the resource. This resource is defined with different actions matching each associated permission. For the "services" resource, these permissions-action are "read-GET", "write-POST", etc. The permission is defined by the micro-service, but is attached to an action that refers to an HTTP method. It is important to define the URI of the resource, as the User Management needs to evaluate authorizations to the resources based on the URI and the HTTP method. Aside of the permissions, the model defines the policies allowed to each resource. The example shows to different policies, "developer" which is based on the user role, and "owner" which is defined by the User Management and only authorizes access for the author of the resource.

```
{
 "_id" : "592425ae9aaf0ef7d6be56ca",
 "ClientId" : "son-catalogue",
 "resource_owner_name" : "catalogue",
 "role" : "son-catalogue",
 "resources" : [
   {
     "resource_name": "services",
     "description": null,
     "type": "resource",
     "URI": "services",
     "owner" : "catalogue",
     "scopes": null,
     "associated_permissions": [
       {
         "name": "read",
         "description": "Read a catalogue service resource",
         "apply_policy": ["developer"],
         "action": "GET"
       },
       {
         "name": "write",
         "description": "Store a catalogue service resource",
         "apply_policy": ["developer"],
         "action": "POST"
       },
       {
         "name": "update",
         "description": "Update a catalogue service resource",
         "apply_policy": ["developer"],
         "action": "PUT"
       },
       {
         "name": "delete",
         "description": "Remove a catalogue service resource",
         "apply_policy": ["developer"],
         "action": "DELETE"
       }
     ]
   }
 ],
 "policies": [
   {
     "name": "developer",
     "description": "SONATA Realm role authorized to request the resource",
     "type": "role",
     "logic": "positive",
     "scopes": null
```

```
    },
    {
      "name": "owner",
      "description": "SONATA Realm user owner of the resource",
      "type": "user",
      "logic": "positive",
      "scopes": null
    }
  ],
  "scopes": null
}
```

Furthermore, in order to enable dynamic and configurable authorization models, a new API has been introduced. The API allows to register, update and delete resource entries per each micro-service. When a new micro-service registers to the Service Platform through the User Management, this micro-service can also register a new model with associated resources and permissions. This way, any incoming request to the Service Platform, that asks to access a protected resource, needs to be evaluated in order to be authorized to access the requested resource. The available endpoints to the API are

- Query: The endpoint `GET /resources?` allows to retrieve a list of stored micro-service component resources that are being evaluated by the User Management in order to authorize any request.
- Register: The endpoint `POST /resources/uuid` allows to register any new micro-service component resources to the User Management authorization process. It is recommended to register the component resources before the micro-service registers to the Service Platform and starts to run. Only one entry is recommended per micro-service, given that a single model can hold many resources per micro-service.
- Update: The endpoint `PUT /resources/uuid` allows to remove a micro-service component resource model previously stored in the User Management database and replace it by an updated version. It will create a new entry with a new unique identifier, however the micro-service should remain the same.
- Delete: The endpoint `DELETE /resources/uuid` allows to simply remove a micro-service component resource model previously stored in the User Management database. This action means that any exposed resource by a micro-service will become available to any incoming request.

2.2.3 New features

This section describes the new features that have been implemented for the final release of the User Management. These features are classified in different blocks based on the functionalities that they present. The new additions come to meet the planned features from the initial release and some new requirements that have emerged. Listed below are the features that are available in the final release:

Block I Authentication

- User registration: This feature is now able to create a new User Account for any platform end-user, assign roles, groups and policies (Developer, Customer and/or Admin)
- Micro-service (service) registration: This feature supports the creation of a new Service Account for any platform micro-service, assign roles and policies (A component example is the SP Catalogue)
- User Login: Enables User Accounts authentication to the platform through user credentials
- Service Login: Enables automated Service Accounts authentication to the platform through micro-services credentials
- Logout: Enables the access token invalidation for User and Service Accounts, ending any ongoing session
- Integration with SONATA BSS and SONATA GUI: The Authentication features have been fully integrated with the SONATA components GUI and BSS. The GUI component allows end-users to register new User Accounts to the SONATA Service Platform as Developers. Furthermore, it allows end-users to authenticate and access to the User Interface to manage their accounts and other options. On the other hand, the BSS allows end-users to register new User Accounts to the Service Platform as Customers. Through its User Interface, end-users can authenticate and manage different operations.

Block II Authorization

- Inspect end-user and micro-service provided JWT (Access Token): The authorization process always requires the end-user or micro-service to provide a granted Access Token. The Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client, where the end-user performs the required authentication prior to request the token. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the User Management that acts as authorization server. Once a end-user or micro-service has been authenticated to the platform, any request must provide the granted Access Token while it is active. The Service Platform will perform a series of inspections over the Access Token through the User Management, considering its status and lifespan
- Set of verifications: Token status, identity, roles, permissions Inspections are performed over the provided information in the Access Token for any request sent to the Gatekeeper API. The inspections currently evaluate certain conditions such token status and expiry time, the requester token integrity and identity, associated roles, permissions and claims. Once these inspections have been performed, the User Management is ready to perform the next step in the authorization evaluation to grant access to the request
- New Authorization mechanism based on new model: The authorization process evaluates the incoming request basically by the URI of the requested resource and the HTTP method. The new model defines the required roles and associated permissions in order to access the requested resource. The User Management checks if the requester (end-user or micro-service) meets the requirements. If the User Management gives green light to the requester, it is authorized and can proceed to access the requested resource
- Integration with SONATA Gatekeeper API: The User Management main communication channel is through the Gatekeeper API. A new User management API which is available to

the Gatekeeper API has been implemented and fully integrated in order to meet any Identity and Access requirement. The chart Table 2.1 presented below shows more details about this new API

Block III Management

- User and Service Account management: The User Management API allows to list and query for users and micro-services information, update and delete their information or accounts.
- Roles, groups and permissions management: Roles described in this deliverable are pre-defined roles that are created by the User Management installation script. However, this new feature enables to create new roles, update and delete them through the new management REST API Table 2.1. User Management roles follow the Role-based Access Control (RBAC) approach to restrict platform's resource access, however with the new Authorization model some resources can follow a limited User-based Access Control (UBAC) using a custom policy such as the "owner" policy presented in the model example. RBAC pattern is usually used for role associated permissions such as 'read/write' access to a group of resources, while the role of 'user' has a 'read access permission' associated to 'resource'. This approach tries to avoid having large number of roles that would quickly become unmanageable, requiring other support tools such an Access Control Lists (ACL).

New groups can also be created, updated and deleted through this REST API. Some initial groups are created as pre-defined groups through the installation script, which are attached to the pre-defined roles. New created groups can be also attached to other roles, using `attributes.roles` field when registering a new group form. In addition, permissions can also be managed through the new Authorization model, described in the Block II features.

All management can also be achieved manually through the Keycloak GUI, the Admin Console, which is a complete management asset for administrative tasks. However, the Keycloak Admin Console must be restricted to the Service Platform admins

- Support management: The new User Management API also offers means to manage the MongoDB database and other Adapter configuration. Endpoints such `/signatures` or `/attributes` allow to retrieve User Accounts meta-data as user's Public Keys or Certificates. The API configuration part exposes the Service Platform Public Key to the Gatekeeper, which is required by other components to deal with access token evaluations. It also provides endpoints to read User management server information and logs.

2.2.4 User Management API

This section presents the new User management API which improves the former API charts from Deliverable D4.2 [7]. It is available to the Gatekeeper API in order to implement authentication, authorization and other related operations. Table 2.1 collects the set of endpoints implemented in the new User Management API

Table 2.1: User Management API

Action	HTTP Method	Path	Response	User Token required
User Registration	POST	/api/v1/register/user	201 Created 400: Bad request	No

Action	HTTP Method	Path	Response	User Token required
Service Registration	POST	/api/v1/register/service	201 Created 400: Bad request	No
Service Registration	POST	/api/v1/register/service	201 Created 400: Bad request	No
User Log-in	POST	/api/v1/login/user	200: OK 401: Unauthorized	No
Service Log-in	POST	/api/v1/login/service	200: OK 401: Unauthorized	No
Log-out	POST	/api/v1/logout	204: No Content 401: Unauthorized	Yes
Token Authorization	POST	/api/v1/authorize	200: OK 401: Unauthorized	Yes
Userinfo (OIDC endpoint)	POST	/api/v1/userinfo	200: OK 401: Unauthorized	Yes
UM Public Key	GET	/api/v1/public-key	200: OK 400: Bad request	No
Translate username to User ID	GET	/api/v1/userid /api/v1/userid?username=	200: OK 400: Bad request	Yes
Check if token's status is active	GET	/api/v1/token-status	200: OK 401: Unauthorized	Yes
Check if token has expired	GET	/api/v1/token-check	200: OK 401: Unauthorized	Yes
List registered users	GET	/api/v1/users	200: OK 400: Bad request	No
List registered client services	GET	/api/v1/services	200: OK 400: Bad request	No
List available roles	GET	/api/v1/roles	200: OK 400: Bad request	No
List current user sessions	GET	/api/v1/sessions/users	200: OK 400: Bad request	No
List user's sessions	GET	/api/v1/sessions/users/:username	200: OK 400: Bad request	No
List current client service sessions	GET	/api/v1/sessions/services	200: OK 400: Bad request	No
Update user's Public key and/or certificate	PUT	/api/v1/signatures/:username	200: OK 400: Bad request	Yes
Get user data / List registered users	GET	/api/v1/users /api/v1/users?username= /api/v1/users?id=	200: OK 400: Bad request	No
Update user data	PUT	/api/v1/users?username= /api/v1/users?id=	204: No content 400: Bad request	No
Delete user	DELETE	/api/v1/users?username= /api/v1/users?id=	204: No content 400: Bad request	No
Get service data / List registered service clients	GET	/api/v1/services /api/v1/services?name= /api/v1/services?id=	200: OK 400: Bad request	No
Delete service client	DELETE	/api/v1/services?name= /api/v1/services?id=	204: No content 400: Bad request	No

2.2.5 Authentication and authorization

This describes the two major User Management entities, the User Accounts and Service Accounts. The User Management apply authentication and authorization to the Service Platform based on the actors and its associated account.

2.2.5.1 User Accounts

The main entity required by any end-user in order to access the Service Platform is the User Account, which is bound to the end-users, a kind of accounts that are currently classified by user types and roles. The User Management supports a variety of user types:

- Developer

- Customer
- Admin
- Multiple

These user types are pre-defined to the User Management module and the SONATA Service Platform. More user types can be added once the platform has been deployed. However, each type of user have main particularities, as they are associated to roles, and roles to permissions. In order to show some the differences between each user type, below there is a short list highlighting some characteristics of each user type for User Accounts:

Developers:

- Register through the SONATA GUI component
- Login through SONATA GUI component to manage profile information and other options
- Login through SDK son-Access component
- Submit and request SONATA packages and descriptors to the Service Platform
- Request monitoring of services to the Service Platform

Customers:

- Register through the SONATA BSS component
- Login through the SONATA BSS component
- Instantiate services
- Login through the SONATA GUI to manage profile information and other options

Admins:

- Need to be designated and registered by the Service Platform administrator or owner
- Login through the SONATA BSS and SONATA GUI components
- Login through User Management Admin Console
- Manage Service Platform configuration

Any end-user that owns a User Account requires of a client component that is part of the SONATA ecosystem in order to access the platform. These components are part of the User Clients scope, which are components that usually are found outside the Service Platform:

- User clients are micro-service components that act on behalf the user
- These components differ from other platform components because they do not register nor login themselves
- These components always use User's credentials for authentication and User's access token for authorization to the Service Platform
- Some examples are: SDK son-access, SONATA BSS, SONATA GUI

2.2.5.2 Service Accounts

The Service Platform presents interaction not only with end-users but also with other Service Platform components. The main entity required by any SONATA component inside the Service Platform, in order to interact with other Service Platform components is the Service Account, which is bound to the components or also called micro-services. Unlike User Accounts, this kind of accounts are not classified by any type or role. There is no type distinction and each Service Account owns its own role. Components or micro-services are registered to the platform in order to create a Service Account, then they are provided of unique roles, which are named as the component name, e.g.:

- Component: son-catalogue
- Role: son-catalogue

These roles must be defined by the component role in the platform architecture. The User Management can keep track of every registered micro-service and grant access tokens so the communication between micro-services is authenticated and can be authorized. Service Accounts are more complex than User Accounts. Any micro-service registration form need to meet Keycloak requirements in order to be properly registered to the platform. Next list present the mandatory fields required to create a Service Account:

- `clientId` : Micro-service name
- `clientAuthenticatorType`: Type of credentials, which must be set to `client-secret`
- `secret`: Service Account password value
- `standardFlowEnabled`: Enables standard token procedure
- `directAccessGrantsEnabled`: Enables Service Account authentication grant procedure
- `serviceAccountsEnabled`: Enables the Service Account for the micro-service
- `protocol`: States the authentication and authorization standard protocol. By default, it is set to `openid-connect`

When new components/micro-services are being added to the platform, they may opt to register the exposed resources, permissions and associated policies in order to enable Authorization evaluation. It is required to register them through the presented REST API along with the authorization model based JSON containing resources that are going to be protected by the User Management. Then, any micro-service needs to follow the same process in order to be authenticated and be part of the platform:

- Service account Registration
- Authenticate (Login) to retrieve the access token
- Add the access token to the micro-service to micro-service communication for authorization process

Similar to the Clients scope, these components are part of the Micro-services scope, which are components that usually are internal to the Service Platform. The particularities of this scope are

- Micro-services are Service Platform components that act on their own (there is no end-user behind)
- These components do need to automatically register and login themselves
- They always use their own credentials for authentication and access token for authorization to the Service Platform
- Some examples are: Service Platform internal components such SP Catalogue (`son-catalogue`)

The Micro-service scope also introduces two different authorization approaches that depends whether the Gatekeeper API component is involved or not in the communication flow between two components. These different approaches require from each micro-service to implement the necessary logics in order to be authorized to communicate with other micro-services.

Centralized approach authorization:

When a micro-service is directly connected to the Gatekeeper, it will implement a Centralized authorization approach, as the Gatekeeper API and User Management evaluate any on-going request from any end-user or micro-service to another micro-service. This process is presented in Figure 2.2, where SDK `son-access` sends a request to `son-catalogue` in **1.** but it is held to be evaluated in the Gatekeeper API and the User Management. Then, if the request has been successfully authorized, the Gatekeeper API forwards the request to `son-catalogue` in **2.**:

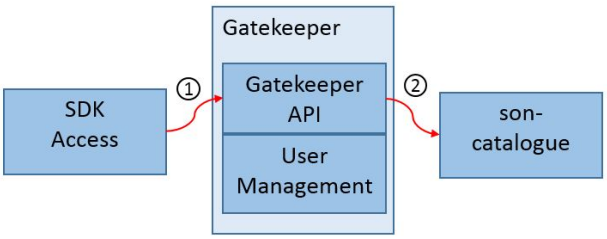

Figure 2.2: Centralized approach

On-demand approach authorization:

On the other side, if the communication between micro-services does not go through the Gatekeeper, but the micro-service is also connected to the Gatekeeper, then it must implement an On-demand authorization approach. The micro-service needs to ask the Gatekeeper to evaluate incoming request. The micro-service is responsible for requesting authorization checks through the Gatekeeper API for each received access token, which should be attached in the 'Authorization' header of the incoming requests. This process is presented in Figure 2.3, where `son-repositories` receives a request from SLM in **1.**, then `son-repositories` forwards the request along with the access token to the Gatekeeper API in **2.**. Finally, Gatekeeper evaluates the request and sends back a response to `son-repositories` in **3.**:

This is an example of the Service Platform Catalogue performing the logics according to the Centralized approach:

1. Son-catalogue-repos component starts, loads authentication settings and requests UM Public Key

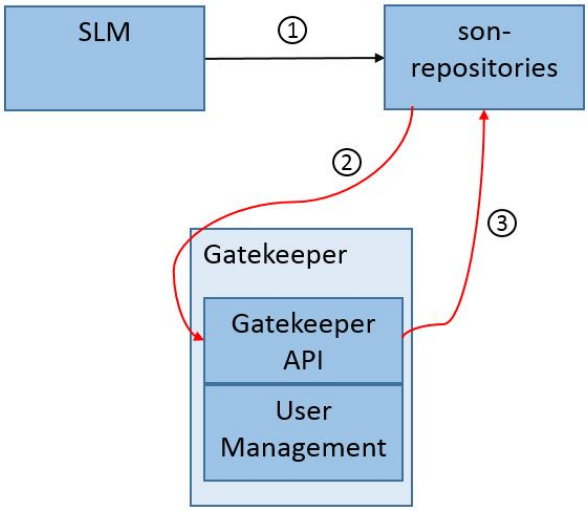

Figure 2.3: On-demand approach

2. Son-catalogue-repos keeps Public Key in memory
3. Son-catalogue (Son-catalogue-repos) registers to the GK API
4. GK API forwards registration data to the UM
5. UM registers son-catalogue, creates a Service account for it and returns a 201 response code
6. Son-catalogue authenticates itself to the GK API
7. GK API forwards the credentials to the UM
8. UM evaluates the credentials and returns a JWT
9. Son-catalogue keeps JWT (Access token) in memory
10. Son-catalogue is ready to operate
11. Son-catalogue authentication layer enters in the 'loop phase' where:
    a) For every request, it checks its token status
    b) When token expires, son-catalogue instantly logins again (new token)
    c) GK API and UM evaluates any request from any component to the son-catalogue (Centralized approach)

2.3 KPIs Management Module

The KPIs Management Module was implemented, as was anticipated in the proposal architectural possibilities in the Deliverable D4.2 [7], reusing the infrastructure from Monitoring Management that contains both Pushgateway and Prometheus components.

Figure 2.4 shows the metric registration process: when a relevant event occurs, the Gatekeeper's KPI module can create an associated metric pushing it to the Pushgateway component. Then, the

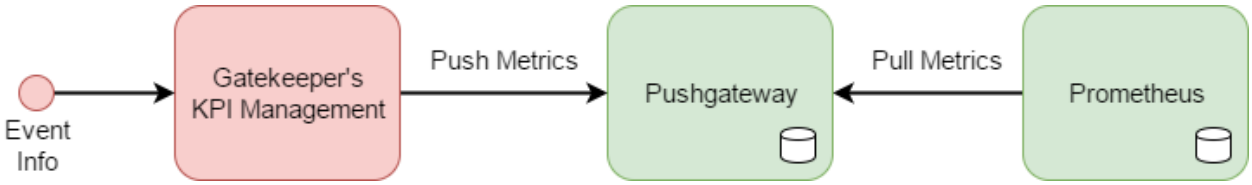

Figure 2.4: KPIs components

Prometheus server that looks periodically for metrics to scrape, retrieves the new information from the Pushgateway making possible the consult of these metrics.

As it will be explained in the GUI section (Section 2.4.1.1) the KPIs can be monitored in the GUI component.

It was implemented a REST API that allows every platform component to create/update counter and gauge metrics in the Prometheus server through the Pushgateway component, resulting as it looks in Figure 2.5

Figure 2.5: Pushgateway Metrics

To demonstrate the validity of this solution there will be three different KPIs registered and monitored in the system:

- Onboarded Packages: how many packages have been on-boarded.
- User Registrations: how many users have been registered.
- Service Instantiations: how many services have been instantiated.

2.4 GUI Module

This section describes the new features that have been developed and implemented in the SONATA GK-GUI from the previous Deliverable 4.2. These features can be divided in three main categories, as discussed below.

2.4.1 Extended GUI views

During the reporting period, the GUI views have been extended and updated, taking advantage of the APIs provided by other components of the SONATA Service Platform and more specifically the KPIs Management module of the Gatekeeper and the Infrastructure Abstraction component.

2.4.1.1 KPIs view

As shown in Figure 2.6, the new KPIs view presents information related to the registered users, Synch Requests and the Packages-on-Boarding in the form of timeline charts.

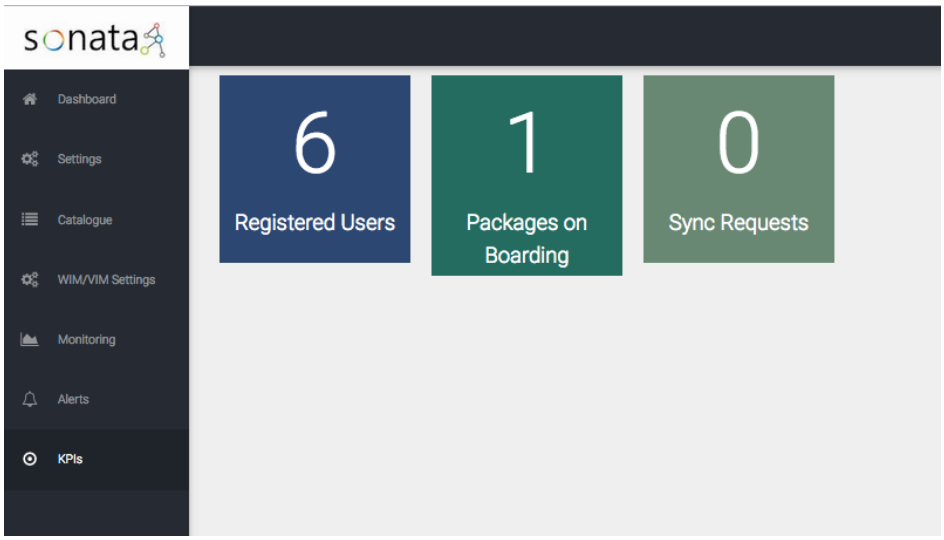

Figure 2.6: SONATA KPIs view

2.4.1.2 Add VIM

Following the modifications and additions that have been implemented on the SONATA IA component during the reporting period, GK-GUI has been extended its respective view to include new fields required by the SONATA Service Platform to support new network service deployment functionality (e.g. multi-PoP network service deployment) and add new resources on the SONATA Service Platform.

As it can be seen in Figure 2.7, in this view, the user can configure the general as well as the compute and networking characteristics of the newly added VIM.

2.4.1.3 Add WIM

The capability of adding a new WIM, in case of more than one WANs, has been added to the SONATA IA and GK-GUI visualizes the required information on a new view, as shown in Figure 2.8.

By the addition of these views, the user is given the capability to configure the IA layer by configuring the WIM(s) and the related PoPs with the proper configuration of the compute and networking VIM characteristics.

Also, the GUI takes advantage of the provided APIs to retrieve the list of WIMs, list of compute VIMs and list of Network VIMs, by pressing the REFRESH button.

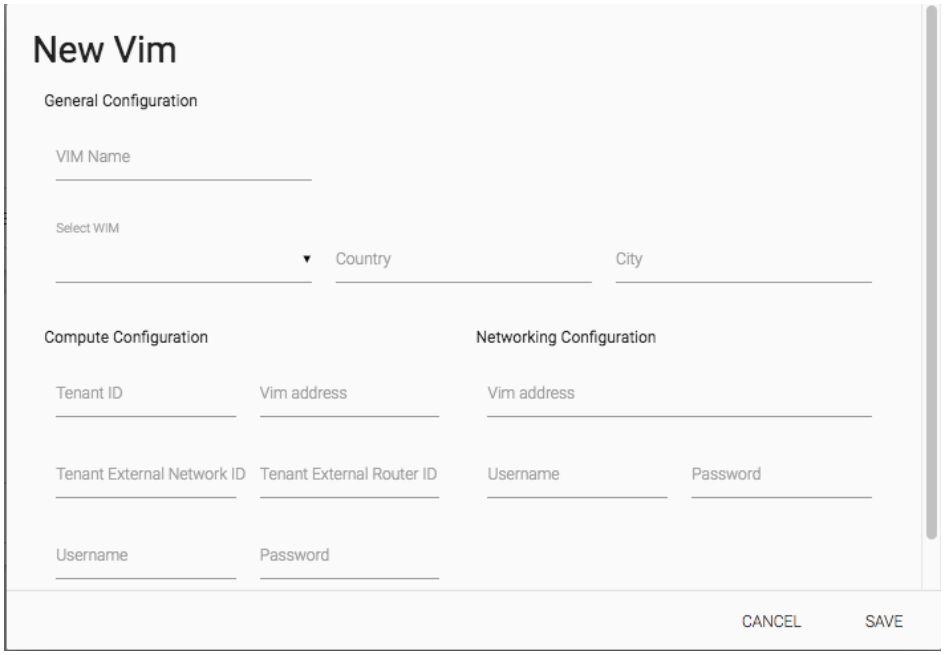

Figure 2.7: Add New VIM

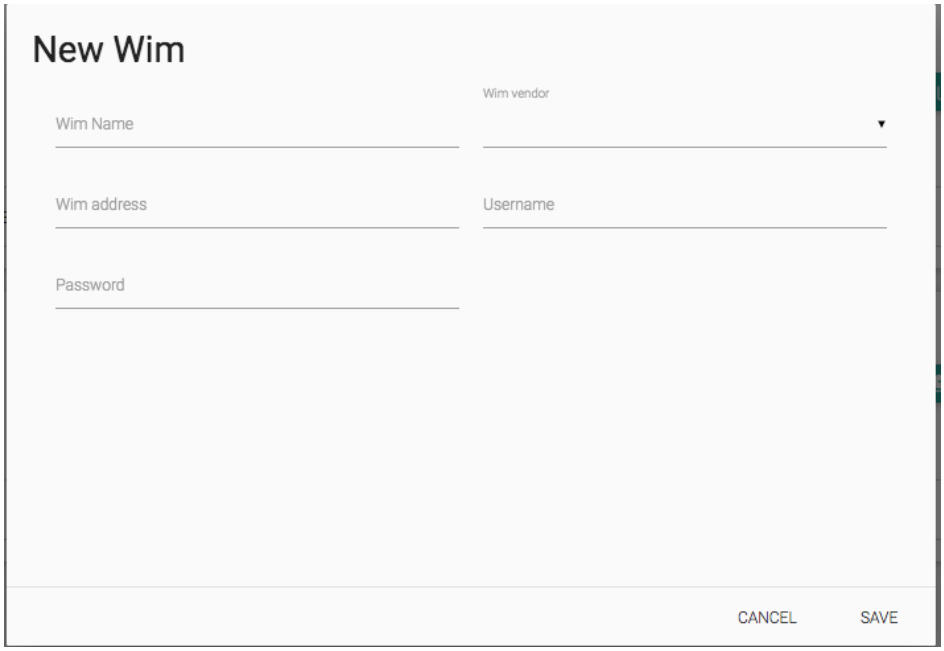

Figure 2.8: Add WIM

2.4.2 Integration with Security services and mechanisms

As explained in the User Management module section of this document, SONATA consortium decided to follow a stronger security management policy during the second year, adopting Keycloak solution for the authentication and authorization processes. Thus, in terms of user registration and login processes, GK-GUI functionality and views have been updated, taking advantage of the User Management OAuth-based API methods provided through Gatekeeper API. Moreover, the consortium decided to strengthen the authentication part of the Service Platform secure accessibility, implementing HTTPS protocol.

Figure 2.9 shows the registration form that a new user must fill in in order to gain access to the SONATA Service Platform.

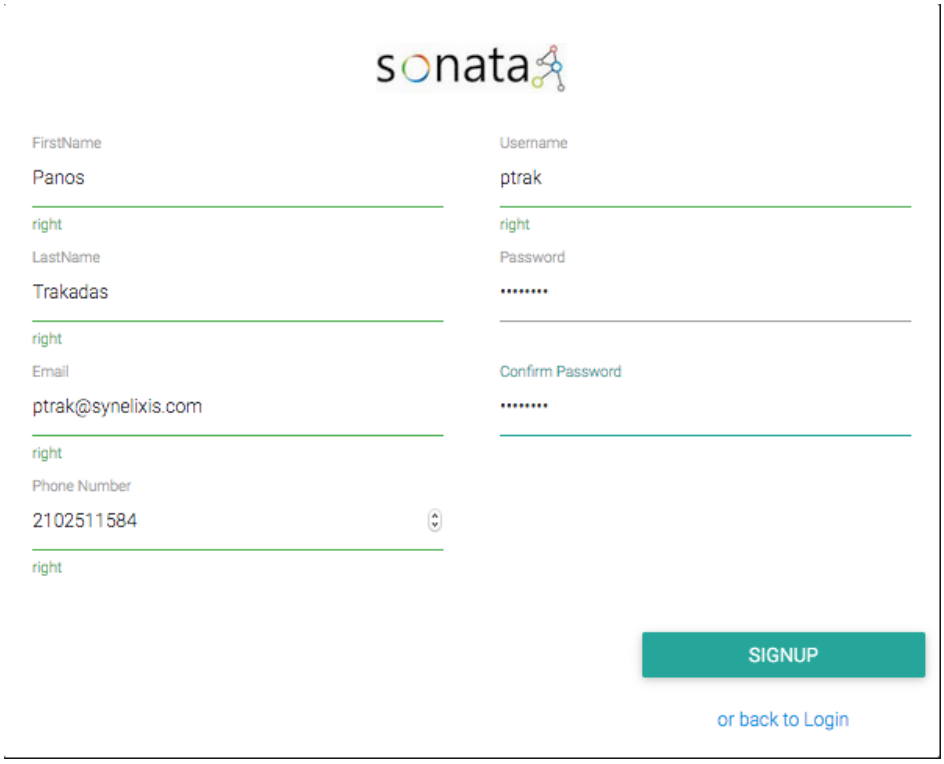

Figure 2.9: SONATA Registration Form

During the registration process, the user adds personal information and his/her role is determined.

2.4.3 Improved user friendliness

During this period, user friendliness of the GK-GUI has been enhanced, offering support for dynamic modifications on the views, such as zoom in/out, modification of the time duration, addition of a new graph on the user Dashboard, etc. as explained below. Also, libraries supporting better functionality, browser memory management and visualization tools have been added, while the code has been further polished and optimized to support views from the mobile application that has been developed during the reported period of time.

2.4.3.1 Create a new chart

The final release of the GK-GUI supports the addition of a new monitoring chart with specific characteristics, as shown in Figure 2.10.

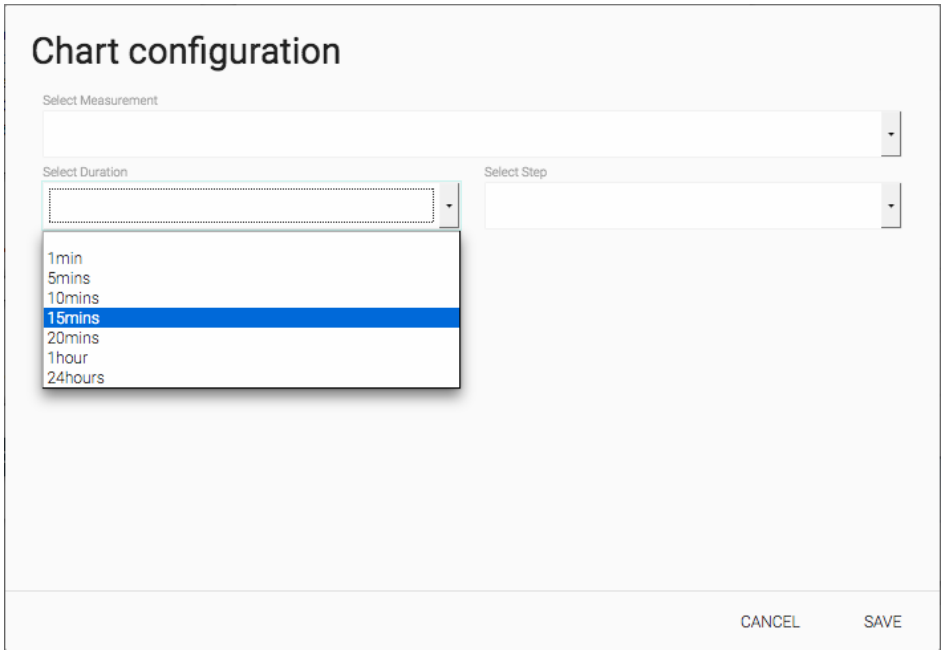

Figure 2.10: Add new chart

The user can select the (metric) measurement type, the duration of the measurement and the refresh rate of the chart.

2.4.3.2 Zoom in/out

Moreover, on existing graphs, the user can zoom in, in order to check details of a given graph and then zoom out by pressing the Reset zoom button, as shown in Figure 2.11.

Figure 2.11: Zoom-in chart

2.4.3.3 Mobile application

SONATA Service Platform can be viewed also on Android and iOS mobile devices, thanks to the proper modifications and adjustments performed on the source code in order to support (apart from both operating systems) all mobile devices screen sizes. The SONATA mobile application gives the opportunity to the user to have a quick look on his services from anywhere.

Figure 2.12 and Figure 2.13 shows the screens for the VIM and WIM settings on an Android phone.

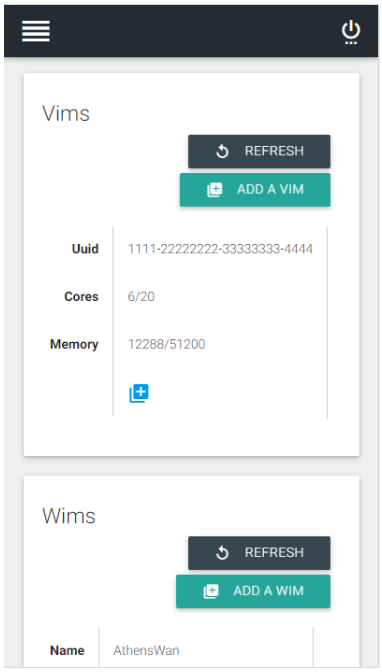

Figure 2.12: SONATA VIM settings

Figure 2.14 depicts the number of registered users, packages on boarded and websocket creation requests.

2.5 BSS Module

This section describes the new features that have been implemented in the BSS module in the last SONATA SP release. Attending to the nature of these new features we can classify them into two functional groups: the location info section and the license store.

2.5.1 Location Info

With the creation of multi-PoPs environments and multi-location testbeds (Athens, Aveiro, Paderborn, London, Tel Aviv) a new need appeared. SONATA needed to include the user's possibility to select the location in which the service would be deployed. That's the reason why it will be included in the service instantiation page the possibility to select this type of location fields.

When the "Instantiate" button is pressed (Figure 2.15) an updated modal view window will appear.

Figure 2.16 shows the new input fields that will be grouped into two sections, corresponding to the ingress and egress fields with which the network service will be configured (if no egress or

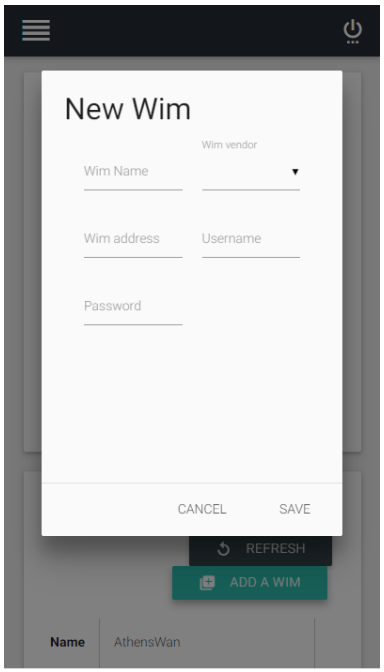

Figure 2.13: SONATA WIM settings

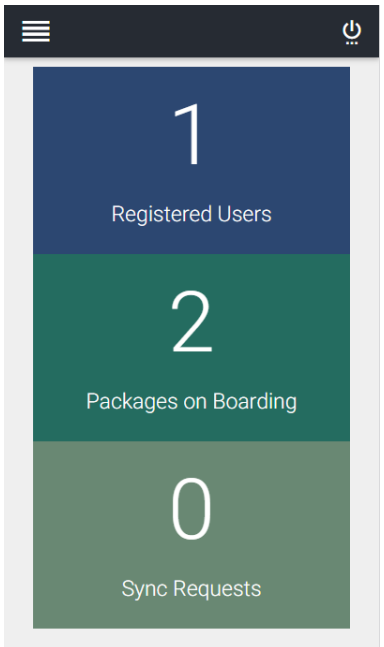

Figure 2.14: SONATA KPIs: registered users, packages on boarded and websocket creation requests

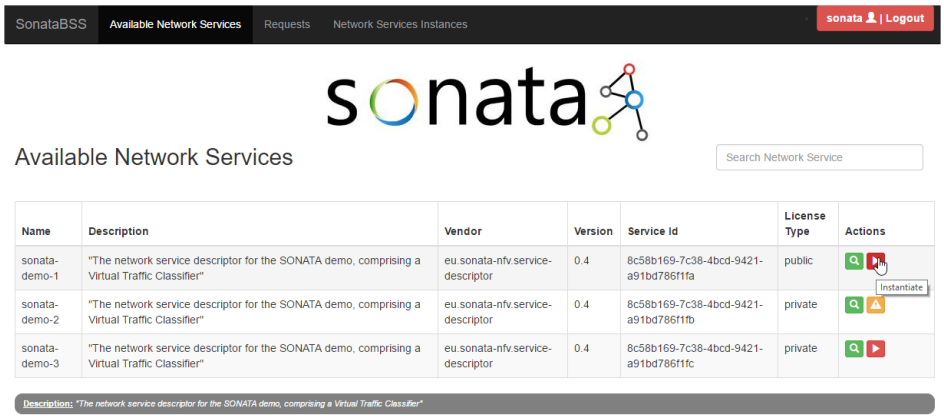

Figure 2.15: Service Instantiation

ingress fields are filled, the default location will be configured). These input fields will be:

- Location: SONATA's test beds are placed in different regions like Athens, Aveiro, Paderborn, London and Tel Aviv. The exact list of locations where a service can be configured will be retrieved from the Gatekeeper, so a new "GET locations" invocation will be implemented.
- Network Attachment Point (NAP): entry or exit point to configure the external interfaces of the network service.

As we mentioned before, the location and NAP fields can be empty, but if one of the fields is filled is mandatory to fill the other one, otherwise, the "Yes" button (to confirm the service instantiation request) will be disabled and an error message will be shown. In other words, the input fields will be always location and NAP couples.

The user will have the possibility to include so many location-NAP pairs as he/she wants. When a location/NAP pair is set, an "Add New Ingress/Egress" button will be enabled to allow the inclusion of new location/NAP pairs. If there are more than one location-NAP set, a "Remove Ingress" button will appear. This button will delete the current location-NAP block when is pressed.

The "Yes" button only will be active if the new form passes all validations:

- All Location and NAP fields are coupled by pairs (in both ingress and egress sections)
- The NAP pattern is met ([0-255].[0-255].[0-255].[0-255]/[0-32])

Currently, the information sent to the Gatekeeper in the instantiation request only includes the Service ID that it wants to be instantiated:

```
{"service_uuid": "8c58b169-7c38-4bcd-9421-a91bd786f1fa"}
```

With the mentioned changes, the information sent to the Gatekeeper will be like:

```
{
  "service_uuid": "8c58b169-7c38-4bcd-9421-a91bd786f1fa",
  "egress": [],
  "ingress": [
```

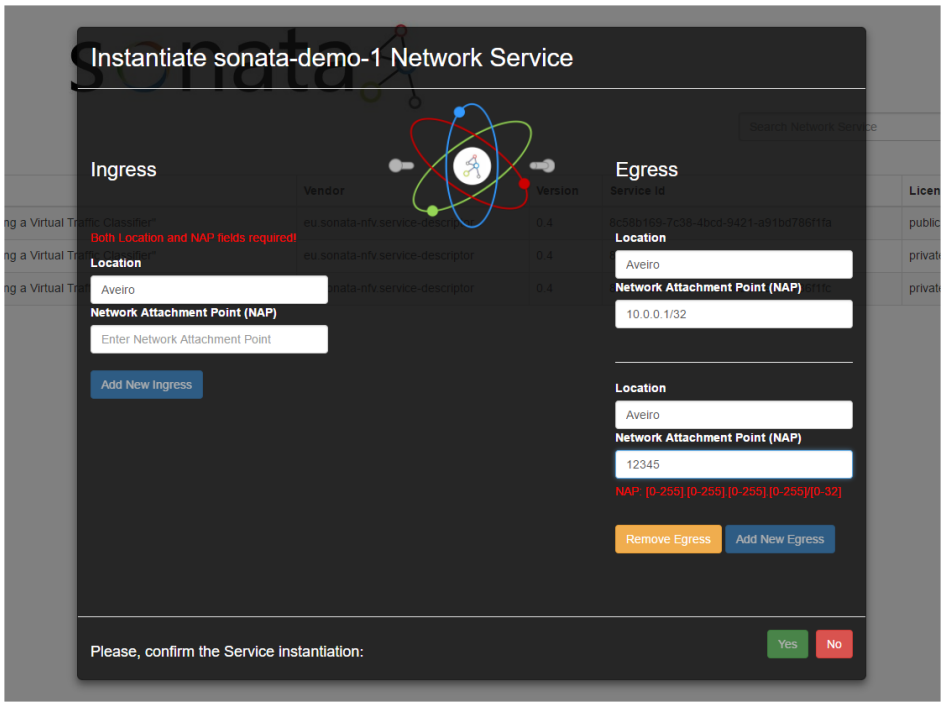

Figure 2.16: Location and Network Attachment Points Fields

```
    {
      "location": "Aveiro",
      "nap": "10.0.0.1/32"
    },
    {
      "location": "London",
      "nap": "20.10.0.2/24"
    }
  ]
}
```

2.5.2 License Store

This section describes the License Store, the module that allows the user to obtain use and instantiation service licenses.

The License Management process in SONATA point of view is very aligned with the TMForum's view and can be found in the annex Appendix A. Due to the shortage of time and resources (at this time of the project almost all efforts are focused on the pilots' phase) we are forced to simplify the license procurement model. That's the reason why SONATA will cover only the license acquisition by the platform user, assuming done the previous legal agreements and the platform's licensing importing process. Taking this into account, these are the assumptions:

- License type. The services will have two types of licenses:
  - public: with no restrictions. The user can instantiate or use the services and its functions to compose new services, attending to the user's role.
  - private: a license acquisition to use or instantiate the service is required .

- Users type. Based on the user role, there will be two types of users for the licenses:
  - developer: the license allows the use of the specific service/function in the creation of packages (Package Creation purpose).
  - customer: the license allows the service instantiation (Instantiation purpose).
  - both: the user is customer and developer at the same time, so can instantiate and use the service and its functions (Instantiation and Package Creation purposes).
- Service level. As the BSS works only at the service level, the procurement of a service license will imply the acquisition of the license of all the functions that integrates the service.

2.5.2.1 Architecture

Figure 2.17 shows the License Management Architecture, where a new Gatekeeper's License Management module appears. This module will manage the licenses and will storage the license definitions and license instances as well.

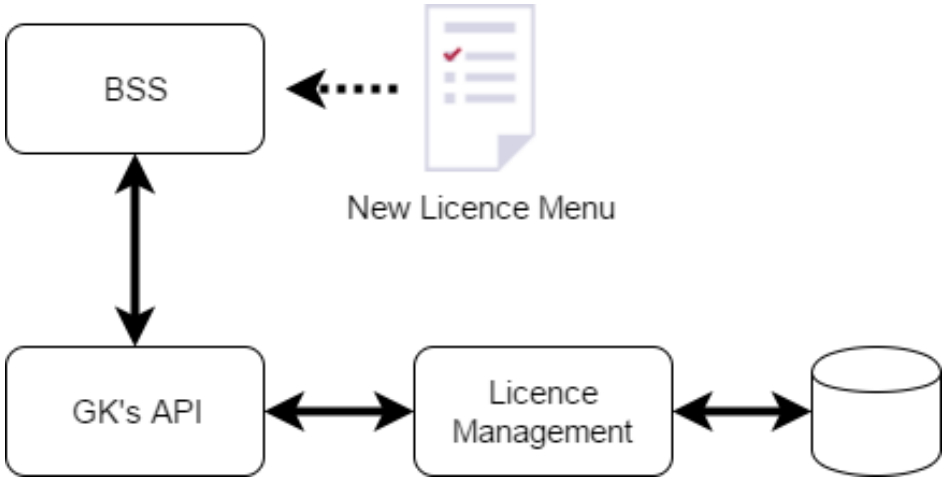

Figure 2.17: Licence Management Architecture

In this context, the BSS will have a new "License Store" menu who allows both types of users to obtain service use/instantiation licenses. In this way, the BSS will invoke the new Gatekeeper's REST API methods related to the license operations.

2.5.2.2 Graphical User Interface

This section describes how the licenses information will be shown to the user through the License Store in the BSS.

There will be a new option in the Menu bar navigation named "License Store". Under this menu, it will be possible to click on the next tabs:

- Service Licenses tab (Figure 2.18): it will show the service license information such as public/private license, purpose and if is already purchased by the user. Different options will be shown attending to:
  - user role: the logged in user can only see the service license related with its role. The developer user can see "Package creation" purpose licenses, the customer user can see the

Instantiation purpose licenses, and users that have the two roles can see both package creation and instantiation licenses.

– public licence or already purchased: if the license is public or is already purchased by the user, the only option that the user can perform is to press the "Details" button (to see license detailed information). Otherwise, a "Request License" button will appear that allows the user to procure the selected license.

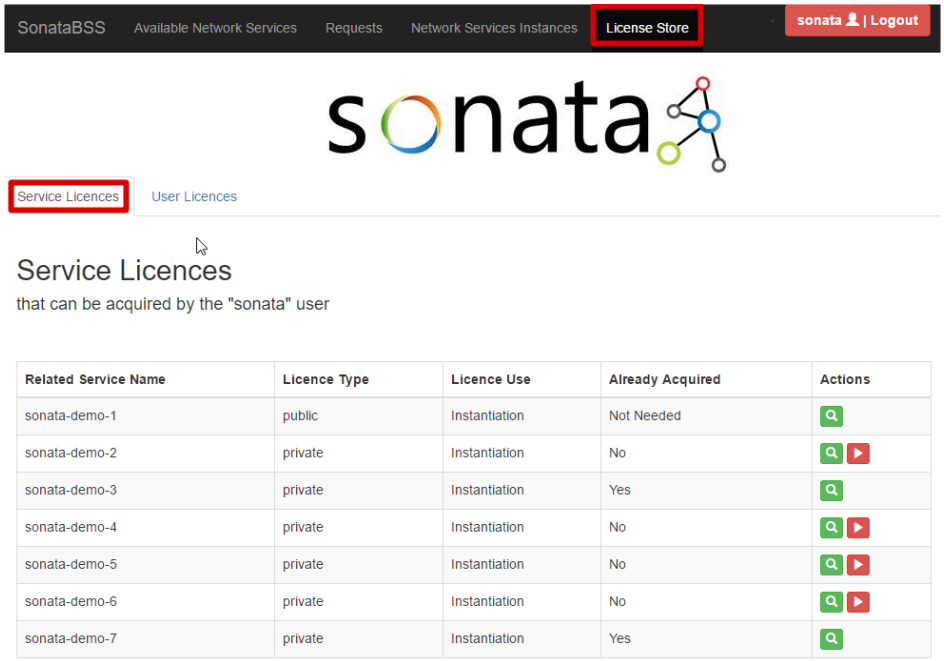

Figure 2.18: Service Licenses' View

- User Licenses tab (Figure 2.19): it will show basic information about the licenses owned by the logged in user, both public and purchased licenses. If the user needs more info about its licenses, a "Details" button can show the detailed information about the selected license.

As well as through the new "License Store" menu, instantiation licenses can be requested at instantiation time through as Figure 2.20 shown. Before showing the instantiation button the BSS will check if the service has a private license and if the user has it. If the user doesn't have the required private license a "Get License" button will appear instead of the "Instantiate" one and a licence request will be sent to the gatekeeper. Once the license is obtained the "Instantiate" button will appear.

2.5.2.3 Message Sequence Chart

This section will describe the message sequence chart of the license related processes performed in the BSS:

- Get available (i.e., that can be acquired) licenses;
- Get user (i.e., that have been acquired) licenses;

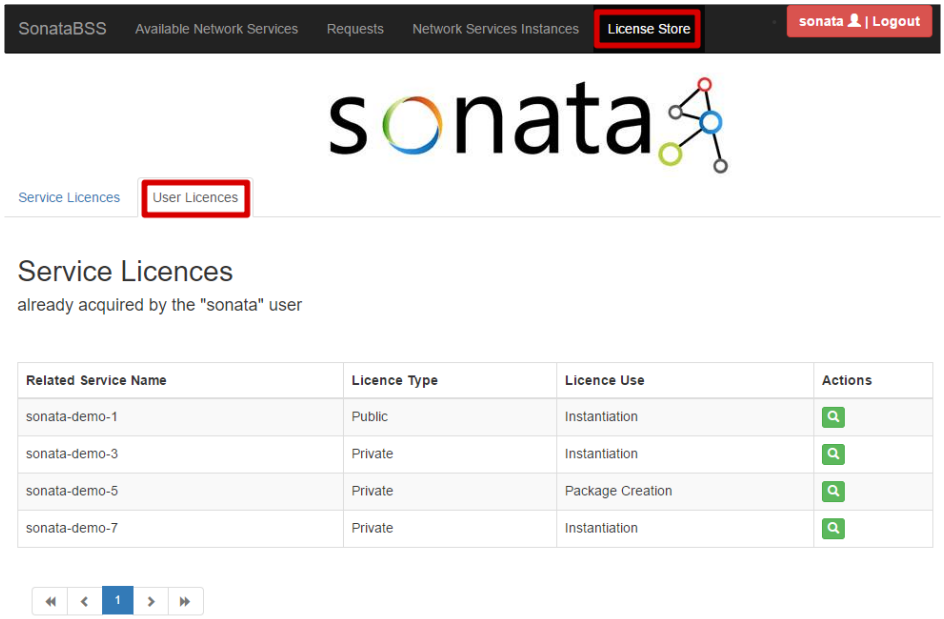

Figure 2.19: User Licenses' View

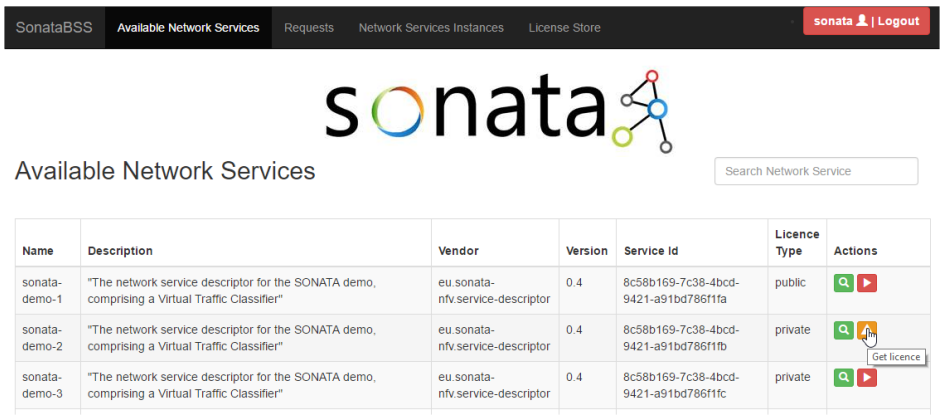

Figure 2.20: Request licence at instantiation time

- Request a license;
- Service request;

Get available licenses

Figure 2.21 shows the "Get licenses" process, a mechanism through which the user can obtain the list of licenses that can be acquired. In this way, a **customer user** will only see **instantiation licenses**, a **developer user** will only see **package reuse licences** or, if the user plays both developer and customer, all types of licenses will be shown. Public licenses will be shown whatever the user role is.

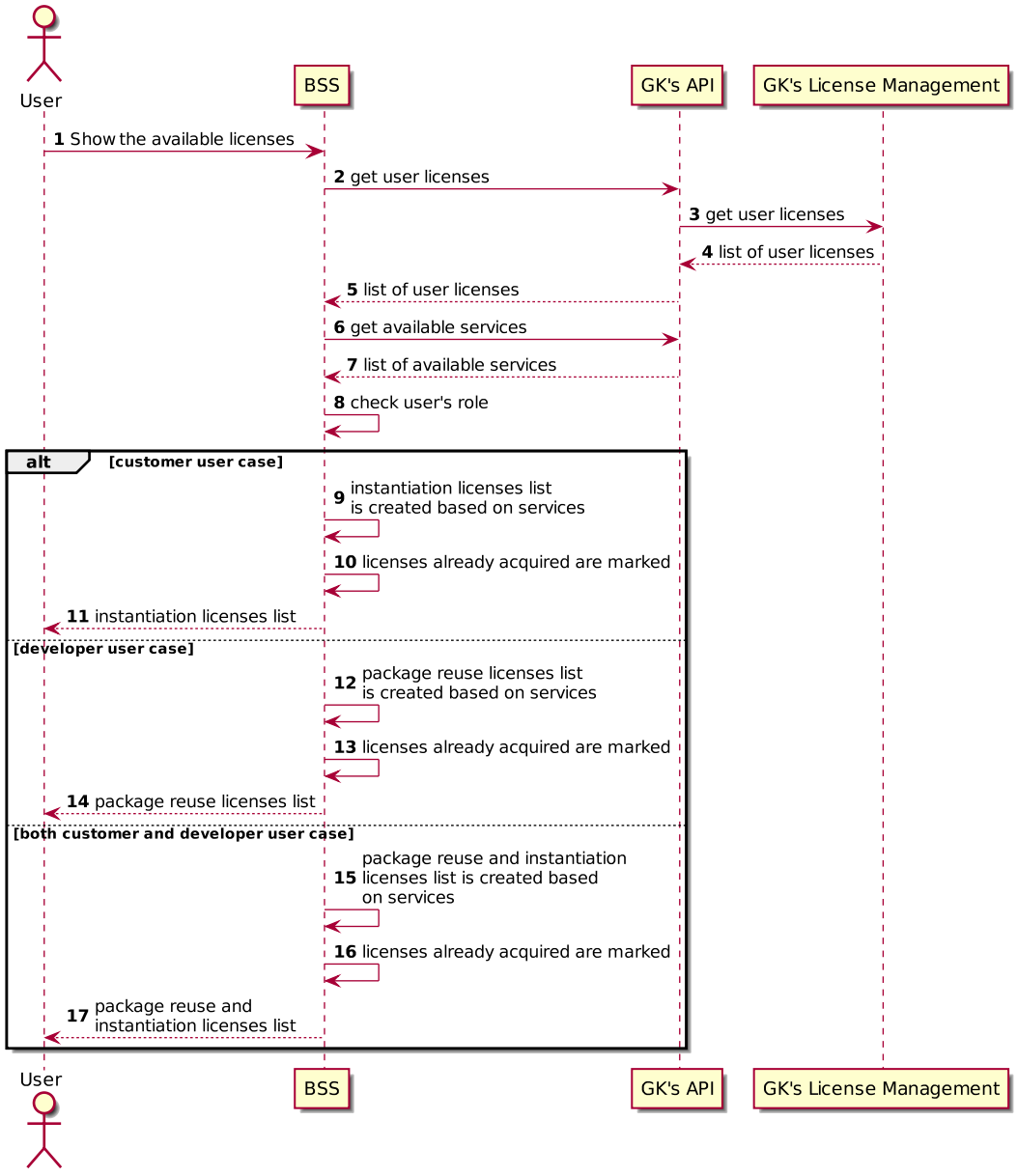

Figure 2.21: Get licenses by role

Get user licenses

Figure 2.22 shows the process which allows the user to retrieve the list of licenses that it owns (public licenses included).

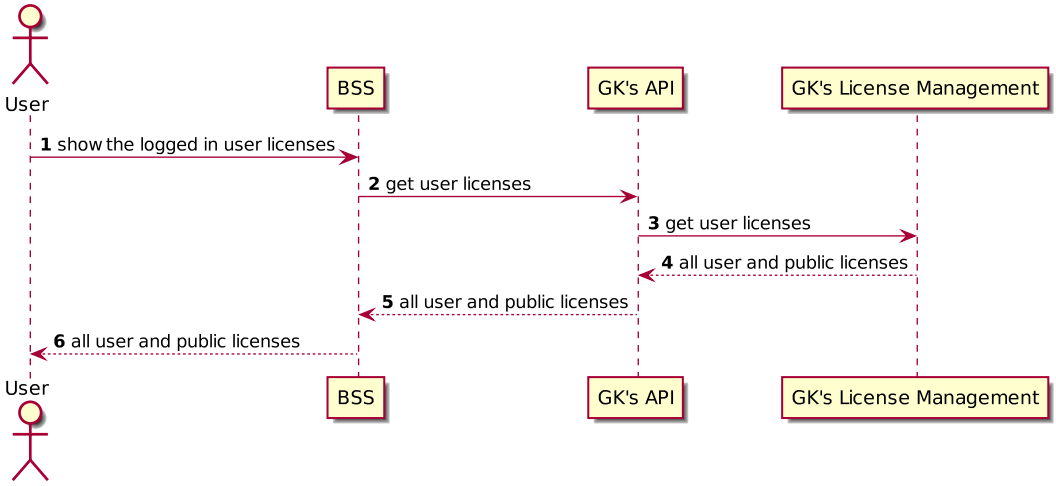

Figure 2.22: Get licenses by user

License request

Figure 2.23 shows the "License request" process, the way to obtain a private license since the "License Store" or through the Instantiation page.

Service request

Figure 2.24 shows the "Service Request" process from the license point of view: the user request a service and the platform checks if the user has a valid license (not expired) in the case of private services.

2.6 Security in the Service Platform

Security in the SP can be seen at two levels:

- the access control for DevOps using the SP web application (available at a typical FQDN like `https://sp.sonata-nfv.eu`)
- the access control for the Administrators that are deploying a new SP using command-line interface
- Built-in security implemented inside each micro-service and in the communication between micro-services

2.6.1 DevOps access control

The authentication and authorization to the former kind of users is controlled by the "Gatekeeper User Management" module based on the open source product Keycloak.

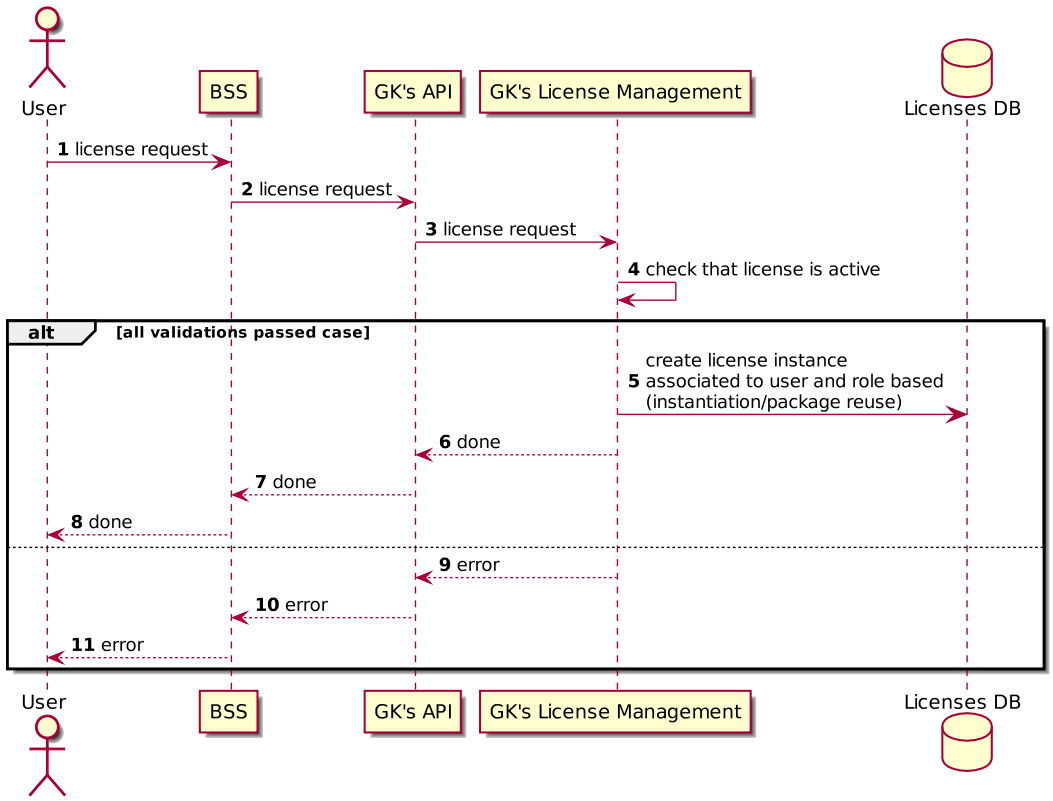

Figure 2.23: License Request

2.6.2 SP Admins access control

The latter group of users have privileged access to the platform, namely because they can change the default configuration and define passwords before deploying a new SP using 'son-install' Ansible tool.

The `'clouds.yaml'` contains the Openstack API URL and credentials to deploy resources to an Openstack VIM. It can be saved to one of the following paths:

- `'/etc/openstack/clouds.yaml'` (needs `'root'` privileges)
- `'~/.config/openstack/clouds.yaml'` (hidden local account)
- at the base of the repo directory (not used in order to avoid sending confidential info to Github)

The `'clouds.yaml'` file was encrypted using Ansible vault, a tool to encrypt files with symmetric `AES` key:

```
$ ansible-vault encrypt ~/.config/openstack/clouds.yaml
```

Then, the playbook using `'clouds.yaml'` must be invoked with passing `'--ask-vault-pass'`, eg:

```
$ ansible-playbook son-cmud.yml \
    -e "ops=create plat=docker-vpsa pop=ncsrd proj=qual distro=xenial" \
    --ask-vault-pass
```

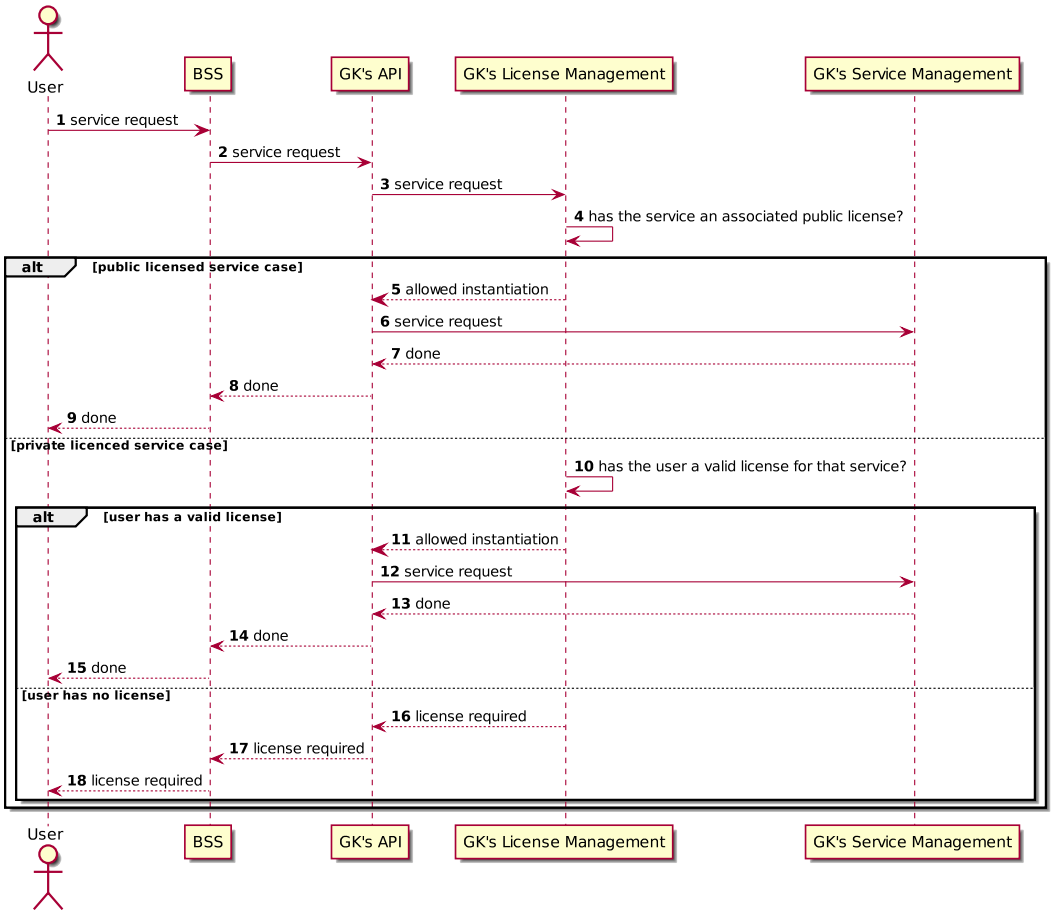

Figure 2.24: Service Request

The default password for user 'sonata' is provided as an hash in `"roles/sp/defaults/main.yml"`. Before deploying a new SP, the administrator has the opportunity to change the default `'sonata'` password:

```
$ echo 'sonata' | openssl passwd -1 -stdin
```

2.6.3 Built-in security in micro-service

Some micro-services need to establish a connection to their own databases: in Docker, the credentials are typically passed as environmental variables. In order to not explicitly show database connection values, they are passed as parameters. The effective declaration of those variables are in "group_vars/sp/vault.yml". This file is encrypted using Ansible Vault and the "--ask-vault-pass" directive must be used, as explained in the previous section.

To change the default password used by Vault ('sonata') and set a new one for a specific environment then:

```
$ ansible-vault decrypt group_vars/sp/vault.yml
$ vi /.config/openstack/.vault_pass
// change password and save file
$ ansible-vault encrypt group_vars/sp/vault.yml
$ ansible-playbook . . . --ask-vault-pass
```

The path to the password file can be declared in 'ansible.cfg':

```
[defaults]
vault_password_file = ~/.config/openstack/.vault_pass
```

To avoid having to provide the path to the password file, you can set an environmental variable:

```
$ export ANSIBLE_VAULT_PASSWORD_FILE=~/.config/openstack/.vault_pass
```

NOTE: since Ansible 2.3 (end of second quarter of 2017), Vault applies also to a single variable and not only to a file (as in the previous versions). Actually, the latter technique is being used in SONATA (as 'son-install' version 2.1). The former should be considered in next 'son-install' release.

Each micro-service REST API can be invoked using one of the following protocols:

- HTTP/HTTPS
- WS/WSS

2.6.4 External secured connections

In order to provide a security layer on front of SONATA service platform, son-sec-gateway component was develop to handle the HTTP/HTTPS connections from the external to internal components.

The Figure 2.25 shows the components involved in the connection schema. It is divided in two parts. On the left hand, is represented the secure path, starting from the User and its connections to both son-security gateway and BSS. The BSS is a plug-and-play component since sonata gatekeeper expose a set of APIs that can be used to integrate others BSS. Sonata BSS can use https if the service platform operator provide the certificates.

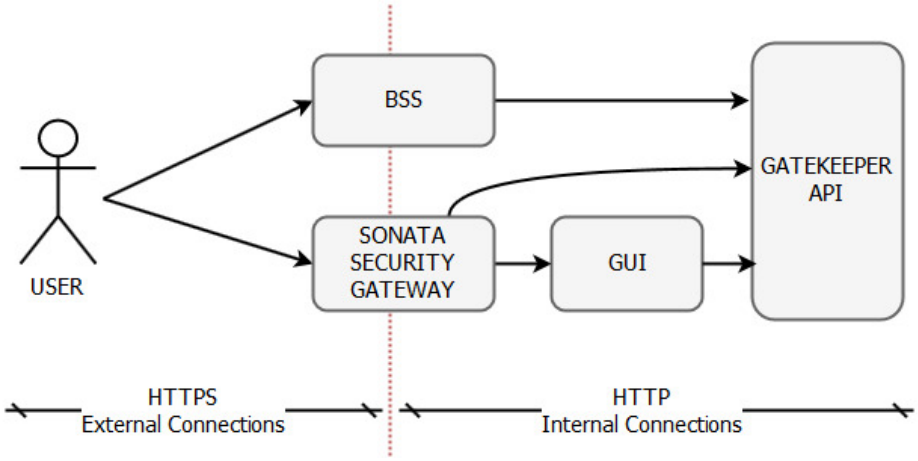

Figure 2.25: son-sec-gw connection schema

The other secure connection from the user is through SONATA security gateway. This security gateway provides an auto-configuration mechanism where if the certificates are present in the server that hosts the SONATA Service platform, then the module will allow https connections, and all the traffic coming from http will be redirected to https. If the certificates are not present then the module will be configured automatically to accept only http. In the Figure 2.26 is shown the auto-configuration process.

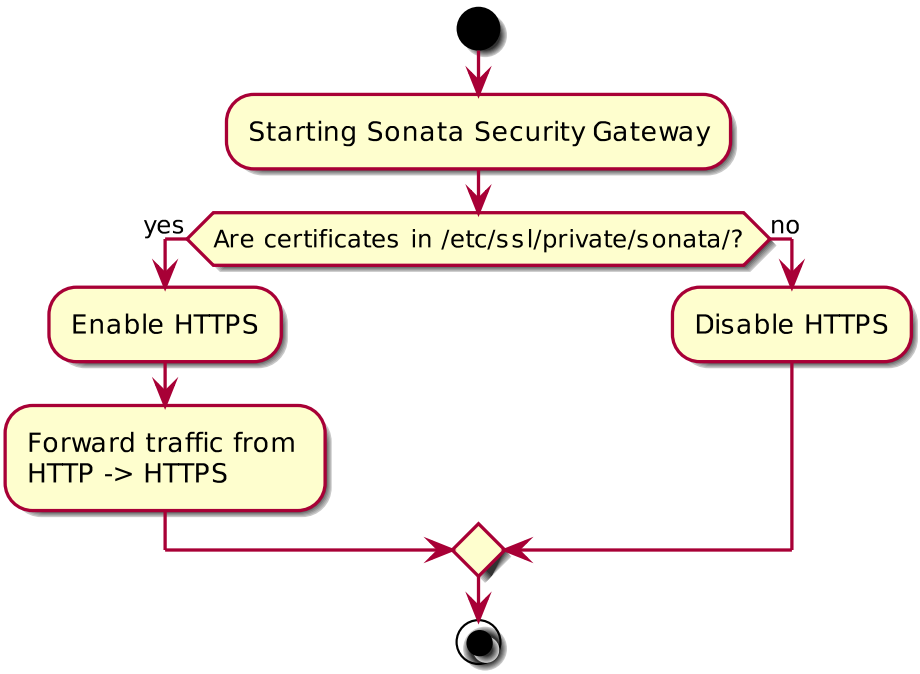

Figure 2.26: SONATA Security Gateway Auto-configuration

Starting from the SONATA security gateway component and BSS to the right side of the Figure 2.25 the connections are without security (http). This is because the links between internal components are placed in a trust domain. Also to avoid the overhead that implies the use of SSL.

The development of the son-sec-gw is available in this link: son-sec-gateway repository

2.6.5 Security within micro-services

The User Management module supports both (human) users and services authentication/authorization. Because we have introduced this feature when most of the micro-services were already developed, the impact of introducing these security steps in all of them would be rather high. We have therefore opted to introduce this in the **Catalogues**, since this would be a service that would most probably have to have authentication/authorization to work with external catalogues.

2.6.6 Message Broker security strategy

The message broker, that is one of the central pieces in terms of communication within the Service Platform, is configured to separate the communication between Service Platform and SSM/FSM components. The reason is to add a security layer in order to avoid connections from SSM/FSM that tries to control the Service Platform. The strategy implemented is based on the use of three virtual hosts:

- **Virtual host for the Service Platform**
  - `/sonata`
  - User and Password authentication
- **Virtual hosts for SSMs**
  - `/ssms/{uuid}`
  - No User and Password authentication
- **Virtual hosts for FSMs**
  - `/fsms/{uuid}`
  - No User and Password authentication

In order to manage the virtual hosts, user and permissions, SONATA Service Platform had have used the rabbitmq-management API [15]:

- **List Virtual hosts:**

```
curl -i -u guest:guest http://localhost:15672/api/vhosts
```

- **Create New Virtual host:**

```
curl -i -u guest:guest -H "content-type:application/json" \
-X PUT http://son-broker:15672/api/vhosts/foo
```

- **List new users:**

```
curl -i -u guest:guest -H "content-type:application/json" \
-X GET http://son-broker:15672/api/users
```

- **Create a new user:**

```
curl -i -u guest:guest -H "content-type:application/json" \
-X PUT http://son-broker:15672/api/users/name \
-d '{"password":"secret","tags":"administrator"}'
```

- **Delete a user:**

```
curl -i -u guest:guest -H "content-type:application/json" \
-X DELETE http://son-broker:15672/api/users/name
```

- **Set user permissions:**

```
curl -i -u guest:guest -H "content-type:application/json" \
-X PUT http://son-broker:15672/api/permissions/vhost/user \
-d '{"configure":".*","write":".*","read":".*"}'
```

In terms of configuration, the Service Platform is configured by Ansible that is the SONATA configuration manager. The son-broker is configured with the virtual host `/sonata` and a random user/password. Next the Service Platform containers that use the message broker will be initiated, passing as environment variables the configuration made in the son-broker.

For the SSM and FSM, the SMR is the component of the Service Platform that manages the lifecycle of SSM/FSM. SMR will create the vhosts and will pass as environment variable to the SSM/FSM containers the configuration made in the son-broker.

2.6.7 Rate limiter

The Rate Limiter (also known as **throttle** in some of the available APIs) is a new module, introduced to protect the platform against excessive (and sometimes abusive) usage from its clients.

2.6.7.1 Algorithm and monetisation

The rate limiter is based on the **leaky bucket algorithm** ([13]), on which each client/limit has an empty bucket assigned with a given capacity (according to a limit definition). Each request goes to the bucket, that is leaking requests at a given rate (again according to a limit definition). A client is allowed to perform the request if there are remaining capacity in the bucket.

As an example, suppose that we have a limit that allows at maximum **6 requests per minute**. Then our bucket can hold at maximum 6 requests. However we'll start draining our bucket after **60 seconds** at a rate of **6reqs/60sec**, that is removing one request from the bucket every 10 seconds.

This allows the SP to just accept the pre-defined number of requests, thus protecting it against attacks such as the **Denial of Service** [10]. It also allows, together with the **Licence Management** module, to **monetise** the SP, i.e., the implemented rate of usage of the API is the same for every customer, except if some of the customers pay the SP own to be allowed higher rates of usage.

2.6.7.2 High level view of integrating the Rate Limiter module

The Rate Limiter module will be integrated as shown in Figure 2.27.

The sequence shows every request being authenticated and passed through the rate limiter, which either accepts the request if the limit has not been achieved (and executes the request) or denies it.

2.6.7.3 Special case: user creation request

There is a special case of the **user creation request**, for which there is no user requesting it yet. This case is illustrated in Figure 2.28.

In this case, the Gatekeeper API module generates a dummy user for the **user creation** request, that is then used in the Rate Limiter validation. This makes it work just like in any other operations.

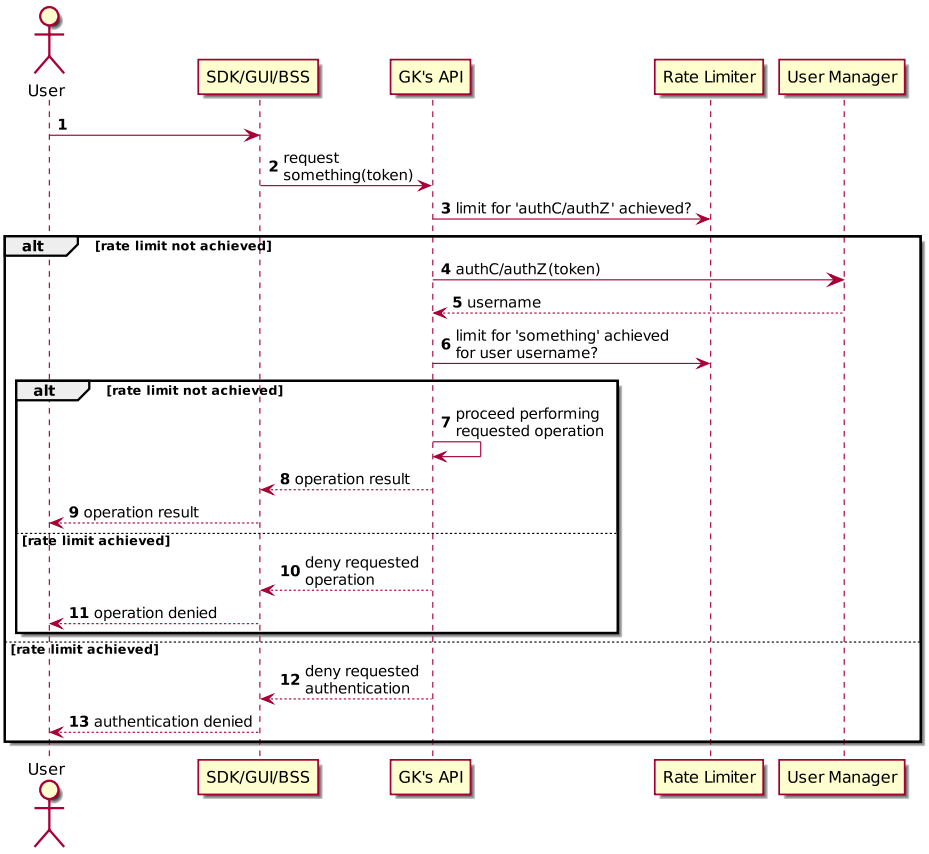

Figure 2.27: Rate limiter high level integration

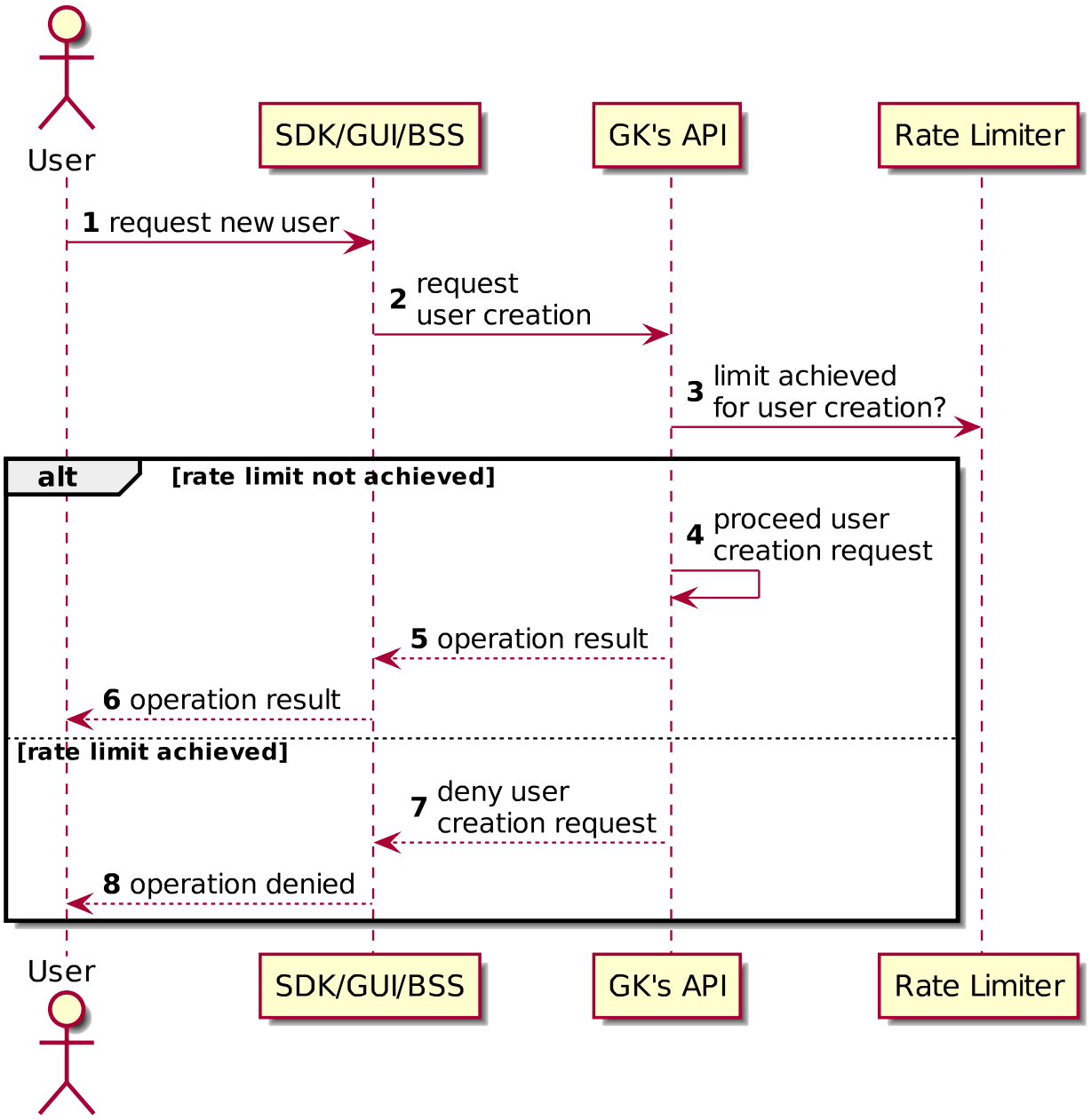

Figure 2.28: Rate limiter when user creation is requested

2.6.7.4 Protection against (D)DOS attacks

This solution protects the Service Platform against (Distributed) Denial of Service ((D)DOS, [10]) in the following way.

This kind of attack may be done using any of the following scenarios (or a combination of both):

1. **Creating users:** since this request uses the internal (dummy) user created upon Platform Start, requests for this operation will be limited according to the schema described above;
2. **Using any other operation:** there has to be a user previously created, so the rate limit is applied.

3 Catalogues and repositories

Catalogues and Repositories are two different key modules that share their implementation in a single component (`son-catalogue-repos`) in the Service Platform architecture, which includes RESTful APIs and the databases. For more detailed information about the Catalogues and Repositories, see section 5.1 from Deliverable D4.1 [6] and section 4 from Deliverable D4.2 [7].

3.1 Service Platform Catalogue

This section describes the new features that have been implemented in the Service Platform (SP) Catalogue module for the SONATA final release. As it has been explained in previous deliverables, this component is responsible for storing NSDs, VNFDs, PDs and SONATA packages (files called son-packages). The SP Catalogue is only accessible through the SP Gatekeeper component.

Aside from these new features, between SONATA release version 2.1 and version 3.0, many enhancements have been introduced in the component, which are also described in this section.

3.1.1 Enhancements

This section lists the enhancements introduced to the SP Catalogue in the latest version:

Owner Username:

This a new meta-data field responsible for storing the username of the end-user that has submitted the package to the SP Catalogue. It is not necessarily the owner of the package, service or function (for that matter there is a `vendor` field), however this field keeps track of the User account that was responsible of the submission. This field requires to have authentication capabilities enabled in the Service Platform, and if the authentication information is not provided, this field will be left blank.

Package signature:

This a new meta-data field responsible for storing the SONATA package file (son-package) signature when it is provided by the developer who has made the submission to the Service Platform. This signature contains the data required by the Gatekeeper in order to make required evaluations over the provided package. As package signing is an optional new introduced feature in the SONATA SDK side (see Deliverable D3.3 [8]), this field can left blank when no signature is provided by the owner or author.

3.1.2 New features

This section shows new added features or that are under development.

- **Descriptor dependencies mapping:** Create a mapping for descriptors that bind them when dependencies are found between NSDs, VNFDs and PDs. This mapping can be helpful to add consistency on the SP Catalogues;

- **Intelligent Delete method update:** Currently the Delete method does not process any dependency check and directly removes a descriptor from the catalogues. This method is planned to be improved in order to check dependencies before doing any change on a descriptor stored in the catalogues. When all checks are passed, then a descriptor can be safely removed from the catalogues.

3.1.2.1 Package-Descriptor dependencies mapping

The Catalogue introduces a new collection in its database in order to implement the new features. This collection stores the data structure that all the stored Package files present, and creates a mapping that can be used to evaluate dependencies between different packages and their content. When the Catalogue needs to find element dependencies, it can perform several searches through the mapping to collect required information.

As the Catalogue can just store unique Services and Functions, duplicates are not allowed. For that reason, there is a "binding" between PDs, NSDs and VNFDs, which result in dependencies between elements. This binding follows the SONATA descriptors schema and as an example it can be defined as:

- PD1 -> PD2, PD3, ... (can depend of)
- PD1 -> NSD1, VNFD1, VNFD2, ... (includes)
- NSD1 -> PD1, PD2, ... (can be included in)
- VNFD1 -> NSD1, NSD2, ... (can be included in)

A Package Descriptor (PD) contains references to all the subset elements it includes. This will help to create a mapping to resolve dependency checks. The mapping will be implemented with a new MongoDB collection that will include all the bindings between descriptors. Currently, a Package descriptor includes the next information for its contents:

- Package descriptor

```
{
  "package_content": [
    {
      "name": "/service_descriptors/sonata-demo.yml",
      "content-type": "application/sonata.service_descriptor",
      "md5": "2fd08cff8e56c6ee6621962ae37fcc4a"
    },
    {
      "name": "/function_descriptors/firewall-vnfd.yml",
      "content-type": "application/sonata.function_descriptor",
      "md5": "17ce9e07f3d215528ab8ecc98cd97d09"
    }
  ]
}
```

A Network Service descriptor (NSD) includes the next information about the Service and its Functions:

- Network Service descriptor


```
{
  "nsd": {
    "descriptor_version": "1.0",
    "vendor": "eu.sonata-nfv.service-descriptor",
    "name": "sonata-demo"
    "version": "0.2",
    "description": "\"The network service descriptor for the SONATA demo...\"\n",
    "author": "Michael Bredel, NEC Labs Europe",
    "network_functions": [
      {
        "vnf_name": "firewall-vnf",
        "vnf_vendor": "eu.sonata-nfv",
        "vnf_version": "0.2",
        "vnf_id": "vnf_firewall"
      }
    ],
  }
}
```


The Catalogue does not have enough visibility between descriptors as they are received separately and they can use different names. Package descriptors do not provide enough information (the **"name-trio"** convention) about included descriptors. Then, to overcome these limitations, the mapping collection focus on solving this issue.

When a new Package file arrives to the Catalogue API, it will first store it in the `son-package` collection, then the Catalogue will rebuild it to evaluate its contents:

- Opens `MANIFEST.MF` file, which contains Package descriptor data
- Gets the Package descriptor `vendor` + `name` + `version` information
- Checks `package_content` data and for each path that points to a YAML (`.yml`) file, it opens them and gets the `vendor` + `name` + `version` information. In addition, it needs to know if the YAML is a NSD or VNFD. This can be achieved by searching for `network_functions` key field. If this field is found, then it means that it is a NSD, and it will include the VNFDs `vnf_vendor`, `vnf_name` and `vnf_version` information.
- Checks `package_dependencies` and gets `vendor` + `name` + `version` information, in case it is available.

A summary of the retrieved mapping information from Package file (son-package):

- Package descriptor: `vendor` + `name` + `version`
- Network service descriptor(s): `ns_vendor` + `ns_name` + `ns_version`
- Virtual network function descriptor(s): `vnf_vendor` + `vnf_name` + `vnf_version`
- Package dependencies: `dep_vendor` + `dep_name` + `dep_version`

A Mapping item sample is shown below:

```
{
  "_id": "472f1f6f-31e4-45ae-931d-15c442db539d",
  "pd": {
    "version": "0.3",
    "vendor": "eu.sonata-nfv.package",
    "name": "sonata-demo"
  },
  "nsds": [
    {
      "version": "0.3",
      "vendor": "eu.sonata-nfv.service",
      "name": "sonata-demo"
    }
  ],
  "vnfds": [
    {
      "version": "0.3",
      "vendor": "eu.sonata-nfv.function",
      "name": "sonata-demo"
    }
  ],
  "deps": [
    {
      "version": "0.3",
      "vendor": "eu.sonata-nfv.package",
      "name": "sonata-demo-dep"
    }
  ],
  "son_package_uuid": "e71e55f4-83f9-4e23-a23e-b50f5607845e",
  "created_at": "ISODate(\"2017-05-16T13:16:44.470Z\")",
  "updated_at": "ISODate(\"2017-05-16T13:16:45.543Z\")"
}
```


3.1.2.2 New Delete mechanism

A new feature called **Intelligent Delete** is introduced in the SONATA release version 3.0. The SP Catalogue is improved with a new and smarter way to delete contents form the descriptor database that replaces the former delete methods. These former methods simply delete SP Catalogue's content without any dependency or status check. The Intelligent Delete is responsible for checking current status of the element to delete and the dependencies it might present.

Current SP Catalogue is capable of storing different element types and each one has an API associated to:

- Package files (son-packages)
- Package descriptors (PD)
- Network Service descriptors (NSD)
- Virtual Network Function descriptors (VNFD)

The Catalogue uniquely identifies each element using `uuid` and the "name-trio" convention (except package files that only use `uuid`). This "name-trio" convention merges the `vendor`, `name` and `version` fields of an element into one identifier name such `name.vendor.version`. Each element API offers various methods to delete contents. A list of supported APIs is shown below:

Package files API delete method:

- `HTTP DELETE /son-packages/:uuid/`

Package descriptors API delete methods:

- `HTTP DELETE /packages?vendor=:vendor&name=:name&version=:version`
- `HTTP DELETE /packages/:uuid/`

Network Service descriptors API delete methods:

- `HTTP DELETE /network-services?vendor=:vendor&name=:name&version=:version`
- `HTTP DELETE /network-services/:uuid/`

Virtual Network Function descriptors API delete methods:

- `HTTP DELETE /vnfs? vendor=:vendor&name=:name&version=:version`
- `HTTP DELETE /vnfs/:uuid/`

The `HTTP DELETE` operation will be just enabled to Package descriptors and Package files. As the only accepted input by the Gatekeeper API are Package files (son-packages), the `DELETE` operation will be applied over Package files contents. `DELETE` operation over other granular elements will not be allowed from external Gatekeeper requests, and they will only be deleted if they belong to a package set. For management and compatibility reasons, these operations are kept in the Catalogue code. E.g.:

- `HTTP DELETE /packages/:uuid/`:

This operation will delete the package file, the PD and all NSDs and VNFDs that belong to the package descriptor.

- `HTTP DELETE /network-services/:uuid/`:

This operation will not be allowed, as it is a subset of a greater element (PD and Package file). However, this API is still available for management and maintenance of the Catalogue.

The workflow and conditions

It is important to state that this new delete mechanism works along with the new **Package-Descriptor dependencies mapping** feature.

A delete request is received in the SP Catalogue, then the Intelligent Delete must follow the next process in order to evaluate if the requested package is candidate to be deleted from the Catalogue:

1. First check if any of the subset elements of the package (all the related NSD and VNFDs of the package) is instantiated using the Repositories records API. If any element is found instantiated, then the delete operation stops and returns a response message "ERROR: Instanced elements cannot be deleted".

2. If no element is found instantiated, then the dependency check must be applied. Each subset element of the package is searched in the mapping collection database (presented in the lines above). If only a reference of the subset element is found in another PD, NSD or VNFD subset, it means that there is a dependency and the subset element is bound to another subset. It is only necessary to find it twice (one for the element to be removed and another for the dependency) to know that this element is a dependency or bound element and cannot be removed. In any case, the process can continue.

3. Next step checks the status of all subset elements of the package that does not belong to another package (which do not present a dependency). The status can be active or inactive, e.g.: {...`"status": "active",` ...}. If the status of the non-dependent elements is found active, it will change to status `inactive`. The delete process will end. However, if the status of these elements are already marked as `"status": "inactive"`, then they can be deleted definitively from the Catalogue.

4. When the Package file, and its descriptors are deleted, the mapping entry in the Catalogue must be updated accordingly.

The dependencies check basically is responsible for the search of any element in the mapping structure, evaluating if a item is referenced in another mapping item. A sample case is presented in Figure 3.1 graphically:

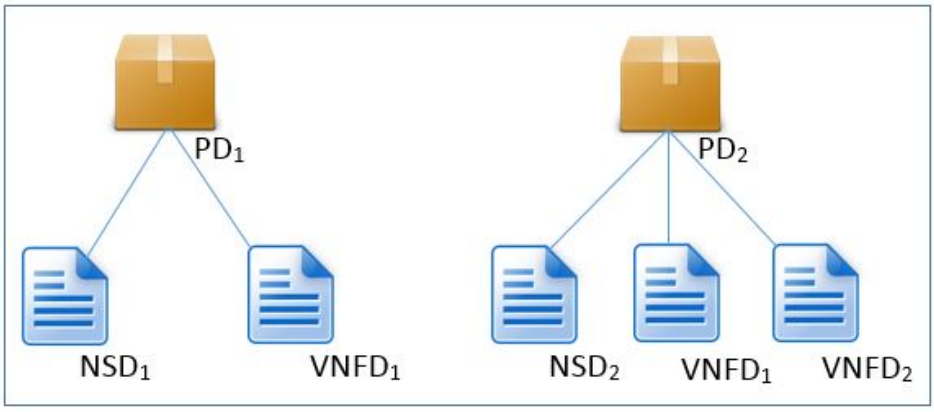

Figure 3.1: Mapping graphical representation

The Package PD1 and PD2 are stored in the SP Catalogue. Then, a `HTTP DELETE` request is received, asking to remove PD1. Package PD1 contains NSD1 and VNFD1, and Package PD2 contains NSD2, VNFD2 and VNFD1 which is part of both packages. Assuming that both packages' status are `inactive`, the Intelligent Delete will proceed to remove the elements that do not share a dependency, in this case, PD1 and NSD1, as can be seen in Figure 3.2:

3.1.3 Authentication and authorization

The communication between SONATA Service Platform micro-services and SP Catalogues can be authenticated in order to increase the security within the Service Platform. The SP Catalogue feature the possibility to enable or disable the authentication and authorization process at start time.

When the authentication and authorization process is enabled, the SP Catalogue triggers a sub-process at start time in order to register to the User Management component through the

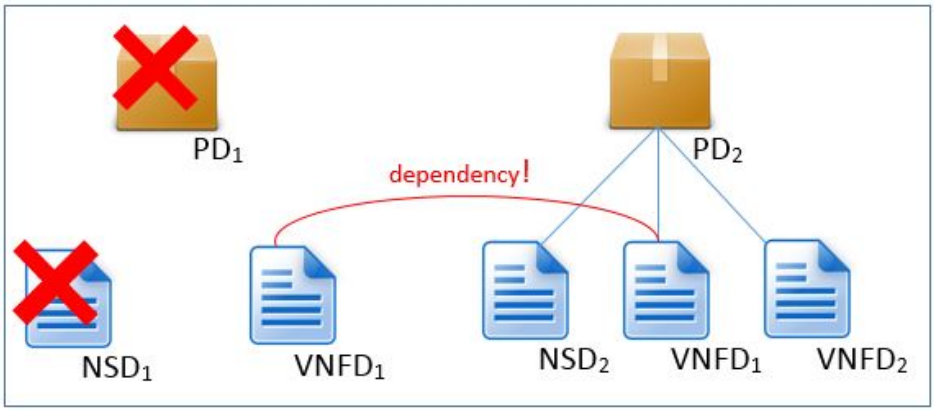

Figure 3.2: Delete and dependencies mapping graphical representation

Gatekeeper API. The registration operation is a `HTTP POST` that includes a JSON configuration file with the required Catalogue information in order to create a **Service Account**, such the component **unique name** and **credentials**. Once the registration is successful, the SP automatically logs-in to the User Management to retrieve a JSON Web Token (JWT) which grants secure access. The JWT includes an access that can be used by the SP Catalogue in the `authorization` header of any request. More details about authentication and authorization can be found in Section 2.

3.2 Service Platform Repositories

This section describes the additional functionalities of the SONATA Repositories that will be added due to the SONATA final release.

3.2.1 Authentication and authorization

The communication between SONATA Service Platform Gatekeeper, the Repositories and other micro-services include the possibility to be authenticated in order to increase the security within Service Platform.

As the Repositories and Catalogues modules share their implementation in a single component (`son-catalogue-repos`) in the Service Platform architecture, their RESTful APIs and the database can incorporate the authentication and authorization procedure. However this feature can be enabled or disabled for the two modules at start time, so they can independently adopt one approach or another. The Repositories can register itself to the User Management through the Gatekeeper API, just as the Catalogue can do.

The Repositories component does not include any new features to the final release.

3.3 Resolver Component

As introduced and described in deliverable D2.3, the SONATA Resolver component allows to resolve artifact dependencies, namely VM image dependencies, within the SONATA Service Platform. Thus, it downloads potentially large VM image and stores them within the Service Platform. Any VIM component can - maybe via the Infrastructure Abstraction component - access the Resolver and, thus, retrieve the VM image and install it within the VIM, say OpenStack. To this end,

the SONATA Resolver allows an end-to-end workflow where a SONATA Package contains all the information and artifacts needed to deploy a Network Service potentially across multiple VIMs.

3.3.1 Package Descriptor Enhancements

In order to allow for artifact dependencies, we enhanced the package descriptor and added a section containing the required information. To this end, we may add a list of dependencies which contain the URL and access credential of the respective artifact. In addition, a dependency contains metadata like the vendor-name-verion tuple that is used to identify the artifact within the resolver component and, thus, the service platform.

```
artifact_dependencies:
  description: "Artifacts, such as VM images, that are not part of this package,
                but have to be downloaded. The artifacts are then identified by
                the vendor-name-version tuple."
  type: "array"
  items:
    type: "object"
    properties:
      vendor:
        description: "The vendor of the artifact."
        type: "string"
        pattern: "^[A-Za-z0-9\\-_./]+$"
      name:
        description: "The name of the artifact."
        type: "string"
        pattern: "^[A-Za-z0-9\\-_./]+$"
      version:
        description: "The version of the artifact."
        type: "string"
        pattern: "^[0-9.]+$"
      url:
        description: "The URL where the artifact can be downloaded from."
        type: "string"
        pattern: "^[A-Za-z0-9\\-_./:]+$"
      md5:
        description: "An MD5 hash of the artifact."
        type: "string"
        pattern: "^[A-Fa-f0-9]{32}$"
      credentials:
        description: "Credentials needed to download the artifact."
        type: "object"
        oneOf:
          - $ref: "#/definitions/credentials"
    required:
      - "vendor"
      - "name"
      - "version"
      - "url"
```

```
      additionalProperties: false
    uniqueItems: true
```

In addition to the dependency information, the Package Descriptor may also contain a list of resolvers. This resolvers have to implement the resolver API and can be accessed by the Service Platform in order to retrieve the respective artifact.

```
  package_resolvers:
    description: "An array of artifacts contained in the package."
    type: "array"
    items:
      description: "The different package resolvers, i.e. catalogues, where
                    packages can be retrieved from."
      type: "object"
      properties:
        name:
          description: "The resolver name."
          type: "string"
          pattern: "^[A-Za-z0-9\\-_./:]+$"
        credentials:
          description: "The credentials needed to access the resolver."
          type: "object"
          oneOf:
            - $ref: "#/definitions/credentials"
      required:
        - "name"
      additionalProperties: false
    uniqueItems: true
```

3.3.2 Resolver Mode of Operation

The Resolver component is implemented as a micro-server and consists of multiple sub-components such as a database and storage. Moreover it interacts with several other components in the SONATA Service Platform, like the Gatekeeper and the Infrastructure Adapter. This imposes several challenges in order to achieve a stable and consistent system. In the following, we provide a high level view on the Resolver's mode of operation:

Resolve and Download Images

In order to trigger an image download to the SONATA Service Platform, the Gatekeeper (or any other component) has to POST a message to the Resolver. This message MUST contain a valid Package Descriptor containing at least a non-empty list of artifact dependencies. Moreover, the message MAY contain a call-back address to inform the calling component when the download has been finished and the image is ready to use, or about any errors. If the message is valid, the Resolver responses immediately with a SUCCESS, or with an ERROR otherwise.

One the valid trigger message has been received, the Resolver adds the meta-data of the image to the database, but marks it as *Not Ready*. It then starts to download the image and stores it to disk. Once the download has finished, it goes back to the database and marks the meta-data as *Complete*. Using the multi-step approach, the Resolver assures consistency or can at least recover if a download fails.

Once a download has been completed, the Resolver informs all components that have registered a call-back. Thus, the Resolver works asynchronously in a well-defined micro-service based approach.

Provide Images to other Components

In order to retrieve an image from the Resolver, any component having the right credentials can access the Resolver API using the vendor-name-version tuple that uniquely identifies the image. To this end, the retrieving component sends a GET message to the Resolver to start downloading the image. Thus, the Infrastructure Adapter component can download and push the image to a VIM, or it can instruct the VIM to download the image directly.

As the image is available at the Service Platform level and can potentially be used by all the Service Platform VIMs, this allows and end-to-end solution where the Service Package is onboarded and installed once, the image is downloaded by the Resolver, and the image becomes available at each VIM.

4 MANO Framework

This section present the MANO Framework's progress since D4.2.

Changes were mostly around the actual separation of the **FLM** code from the **SLM** code. This separation reflects a better separation of concerns of the two-level managers, the service lifecycle manager and the function lifecycle manager. We have also elaborated on the design and implementation of the **Placement Plugin**, as well as improved the Specific Managers infrastructure.

4.1 SLM

In the final phase of SONATA, we finalize the **Service Lifecycle Manager** (SLM) transition into a workflow based task manager. The SLM supports a set of workflows, and translates those workflows into a set of tasks. This makes the SLM highly dynamic, and allows network service developers to manipulate the behaviour of the SLM through **Service Specific Managers** (SSMs) that can customize this set of tasks and their service's workflows.

In Deliverable 4.2 ([7]), we defined a set of workflows that should be supported by the SLM, such as instantiation or terminating a service. In this section, we go through these workflows and detail which tasks they include in their default setup and which are the actors for the specific tasks.

4.1.1 Instantiating a service

The SLM needs to support a **service instantiation workflow**, which the Gatekeeper triggers by sending a message on the `service.instances.create` topic. This message contains the service descriptor (NSD) of the service that needs instantiating, as well as the function descriptors (VNFD) of all the involved VNFs. Once the SLM receives such a message, it starts the service instantiation workflow, which comprises out of the following ordered tasks:

1. Validate that the received payload is correctly formatted;
2. Inform the GK that the instantiation workflow has been started;
3. Onboard Service Specific Managers (SSM) if required;
4. Instantiate SSMs if required;
5. Consult with a task SSM, if available, about the tasks in this task list;
6. Request the topology from the Infrastructure Adaptor (IA);
7. Calculate the placement of the service;
8. Instruct the IA to prepare for a service instantiation;
9. Instruct the Function Lifecycle Manager (FLM) for each VNF in the service to deploy it;
10. Instruct the IA to chain the VNFs together;

11. Instruct the IA to configure the WAN;
12. Generate and store the service record;
13. Instruct the Monitoring Manager to start monitoring the VNFs and the traffic;
14. Inform the GK of a successful service instantiation.

The upcoming subsections will provide further detail for each of these tasks. The Figure 4.1 shows these tasks and their involved actors.

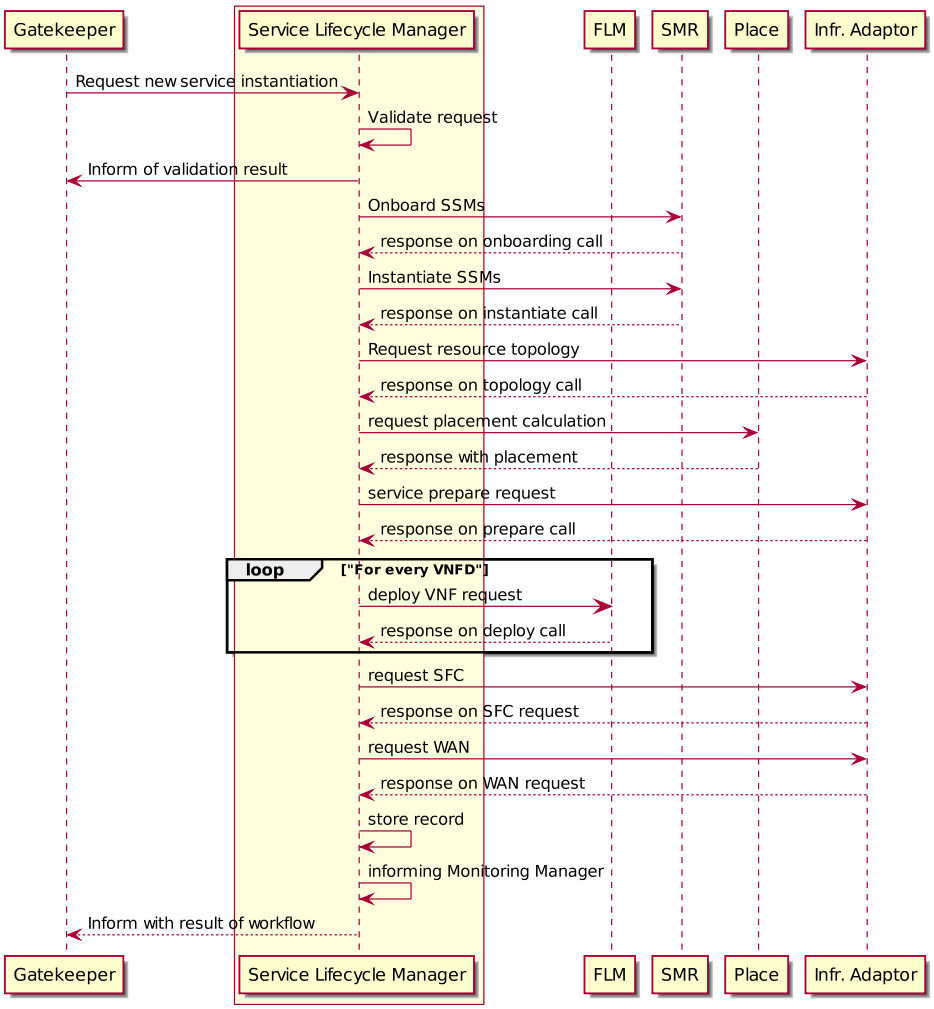

Figure 4.1: Instantiating a service

4.1.1.1 Validate the payload of the GK request

As a first step of the instantiation workflow, the SLM makes sure that the received payload is correctly formatted, and all the required data is present. Next to the presence of the NSD and the correct amount of VNFDs, the SLM also checks the type of the content and whether the GK included the uuid for each of the descriptors. The API for the GK - SLM interaction can be found in Deliverable 4.2 ([7]).

4.1.1.2 Inform GK of the status of the workflow

Once the SLM has validated the instantiation request sent by the GK, it will inform the GK of this result. If the request message is rejected, the GK gets a response from the SLM with an 'ERROR' status, accompanied with an error message and the instantiation workflow in the SLM is aborted. Otherwise, the GK receives a response with an 'INSTANTIATING' status, and the SLM continues with the tasks in the workflow.

4.1.1.3 SSM onboarding

The SLM checks in the NSD whether SSMs should be used during the lifecycle of the service that is being instantiated. If so, the SLM will send an onboarding request to the Specific Manager Registry (SMR) for these SSMs. More information on the API for this call can be found in the section on SMR (Section 4.5). If the onboarding of the SSMs fails, the instantiation workflow is aborted.

4.1.1.4 SSM instantiation

Once the SSMs are onboarded, the SLM sends a request to the SMR to instantiate them. Once the SLM receives a response on this call, the involved SSMs are up and running, and can be communicated with. More details on this call in its API can be found in the section on SMR (Section 4.5). If the instantiation of the SSMs fails, the instantiation workflow is aborted.

4.1.1.5 Consult with task SSM

The developer can customize the service instantiation workflow through a task SSM. Once the SSMs are instantiated, the SLM will check if one of them is a task SSM. If so, the SLM will ask the task SSM if it wants to change to task list of this workflow. The task SSM can then respond with an update for the task list. The SLM will then execute this updated task list instead of the generic list. An example of such a change could be that after the WAN is configured, the SLM should consult with a configuration SSM to perform some configuration tasks (such as configuring a VNF by setting up an SSH connection with it, and informing it with the IP addresses of the other VNFs). The SLM will check if the proposed updated task list makes sense, and doesn't make the service instantiation meaningless.

4.1.1.6 Request the topology

To be able to instantiate a new service, the SLM requires a view of the topology of available compute, storage and memory nodes. For this, the SLM requests the IA for the topology of available resources. If this request fails, the instantiation workflow is aborted. The API for this call can be found in Deliverable 4.2 ([7]).

4.1.1.7 Calculate the Placement

Once the SLM has the topology of the available resources and their availability, and it knows the service chain and required resources from the descriptors, a placement calculation can take places that maps the VNFs on specific PoPs. The placement calculation can be done in multiple ways. The SP has a placement plugin that can be contacted by the SLM to request the placement from. More information on this placement plugin and its API can be found in section Section 4.3. If the developer of the service has specified a placement SSM in the NSD, which is already onboarded and

instantiated by this time, this SSM is contacted to calculate the placement, instead of the in-house plugin. If the placement calculation fails, the instantiation workflow is aborted.

4.1.1.8 Prepare the IA

Once the placement calculations have been completed, the SLM informs the IA that a new service is to be instantiated, and provides a list of the PoPs that will be used for this service. The API for this call can be found on . If the IA responds to this call with an error, the instantiation workflow is aborted. The API for this call can be found in Deliverable 4.2 ([7]).

4.1.1.9 Deploy the VNFs

The SLM instructs the FLM for each individual VNF in the service start its lifecycle. If the FLM response on any of the VNF deploy calls with an error, the instantiation workflow is aborted. The API for this call can be found in Deliverable 4.2 ([7]).

4.1.1.10 Create the service function chain

Once the FLM has indicted for each VNF that it is deployed, the SLM instructs the IA to create the service function chain for this service. If the IA responds to this call with an error, the instantiation workflow is aborted. The API for this call can be found in Deliverable 4.2 ([7]).

4.1.1.11 Configure the WAN

After the service function chaining has been completed, the SLM instructs the IA to create the WAN, which will ensure that traffic flows from the source, through the VNFs and ultimately to the incoming traffic's destination. If the IA responds to this call with an error, the instantiation workflow is aborted. The API for this call can be found in Deliverable 4.2 ([7]).

4.1.1.12 Generate and store the record

After the different aspects of the service have been deployed, the SLM generates the service record. This record is then validated and stored by the Repositories. If storing the record fails, the instantiation workflow is aborted. The API for this call can be found in Deliverable 4.2 ([7]).

4.1.1.13 Inform the Monitoring Manager

The SLM generates and sends a message for/to the Monitoring Manager, to indicate that a new service is deployed and where the functions can be found for monitoring purposes. This message also includes any rules for metric thresholds/alarms that were added to the descriptors. If the Monitoring Manager responds with an error, the instantiation workflow is aborted. The API for this call can be found in Deliverable 4.2 ([7]).

4.1.1.14 Inform the GK of the result of the workflow

The workflow ends with the SLM informing the GK of the result of the deployment. The status in this message is either 'READY' or 'ERROR', depending on the success of the workflow. If the workflow was aborted due to a failure in any of the above tasks, the status becomes 'ERROR' and is accompanied with an error message. The API for this call can be found in Deliverable 4.2 ([7]).

4.1.2 Terminating a service

To be able to stop a running service, the SLM requires a termination workflow. This termination workflow can be triggered by sending a message on the 'service.instance.terminate' topic. This message should include the instance id of the service that needs to be terminated. The API for this call is shown in Table 4.1.

Table 4.1: Termination API between the GK and the SLM.

Type	Topic	Sender	Receiver	Body
request	service.instance.terminate	GK	SLM	• service_id
response	service.instance.terminate	SLM	GK	• status • error • timestamp

Once the SLM receives a termination request, it starts the termination workflow by executing the following, ordered, tasks:

1. Inform the Monitoring Manager to stop monitoring VNFs or traffic related to this service. This is done by making a DELETE REST API request to the following url: `‘monitoring_url‘ + ‘services/‘ + ‘<service_instance_uuid>‘`.
2. Consult the task SSM, if available, whether the task list of this workflow needs to be customized (like in section Section 4.1.1).
3. Instruct the IA to de-configure the WAN. The API for this task can be found in Deliverable 4.2 ([7]).
4. Instruct the IA to remove the service function chain. The API for this task can be found in Deliverable 4.2 ([7]).
5. Instruct the FLM to terminate each specific VNF. The API for this call can be found in the FLM section (Section 4.2).
6. Perform final clean up for this service. This step is VIM dependant. For example, if OpenStack was used to deploy this service, the stack for this service needs to be removed.
7. Instruct the SMR to terminate the SSMs. The API for this call be found in the section on the SMR (Section 4.5).
8. Update the service record so it reflects the 'terminated' status. The API for this task can be found in Deliverable 4.2 ([7]).
9. Inform the GK whether the termination was successful or not.

The Figure 4.2 shows these tasks and their involved actors.

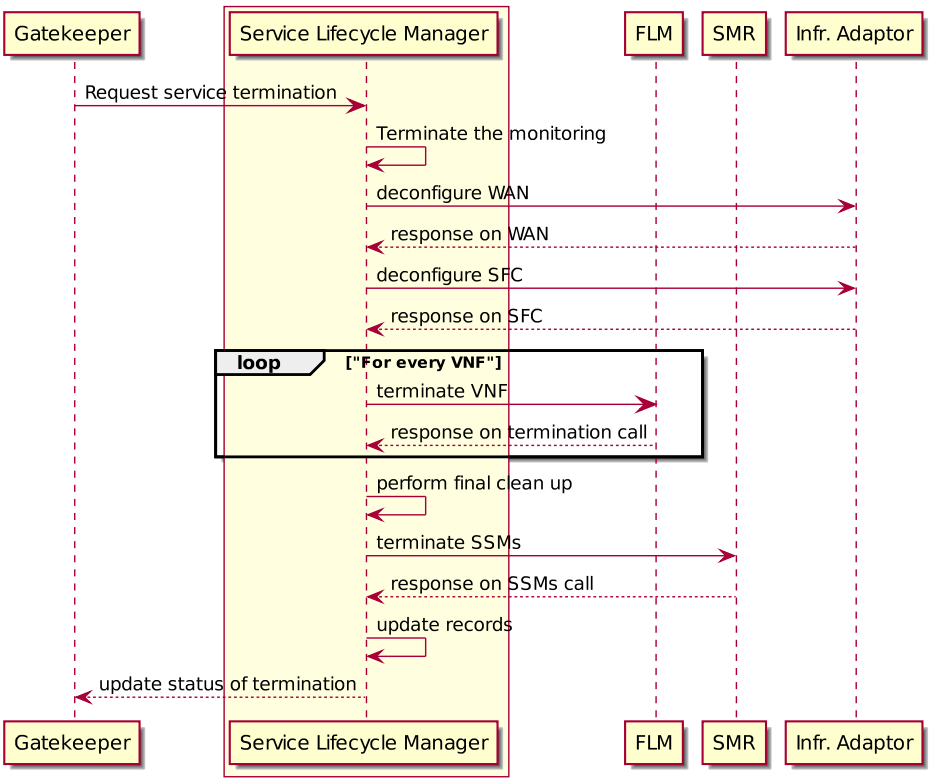

Figure 4.2: Terminating a service

4.1.3 Pausing and resuming a service

Due to its high complexity, pausing and resuming functionalities for a service will not be included in release 3.0 of SONATA. What follows is a list of issues that arise when thinking about such functionalities.

- How to ensure that the VNFs in the service are paused/stopped in the correct order, so that they all have the same/correct state.
- What needs to happen with the traffic that normally goes through the service. Should it be stopped, buffered, or should it flow to the destination without going through the service.
- Are the allocated resources freed when the service is paused. If so, how is state migrated in a generic way to newly allocated resources when the service resumes.

Due to resource restrictions, these questions will not be addressed in the scope of SONATA.

4.1.4 Updating a service

As described in Deliverable 4.2 ([7]), we identified and are integrating some workflows that allow to update a service. The first service update workflow, that was already introduced in the first SONATA SP prototype, is updating the SSMs and FSMs that are being used by the SLM and FLM. If a newer version of an SSM or FSM is available, the customer can request to update the running service. This will trigger the GK to make an update request. The API for this request can be found in Deliverable 4.2. This request will trigger an interaction between the SLM, FLM and

SMR to update the respective SSMs and FSMs, during which the SLM and FLM will buffer all traffic intended for those SSMs or FSMs. The communication between the SLM, FLM and SMR for this workflow is documented in section Section 4.5.

A second manner in which a running service can be updated, is based on the received monitoring data. Section Section 4.1.6 describes how the developer can instruct the SLM to change the lifecycle of a running service based on monitoring information. Such lifecycle changes could include the termination of a service, deploying a new VNF or scaling a running VNF.

4.1.5 Configuring a service

In some use cases, certain configuration instructions should be executed to get the service to behave correctly. Informing the involved VNFs with the IP addresses of the other VNFs, or triggering the start workflow of the involved VNFs in a specific order are both examples of this. These configuration instructions are service specific and can not be generalized into any of the generic workflows. Therefore, they are encapsulated in an SSM.

When the SLM receives a trigger to configure a service, it will contact the configuration SSM associated to this service. In section Section 4.5, it is described how both SLM and SSM know on which topic to communicate with each others. As only FSMs, and thus not SSMs, can communicate directly with the VNFs, and most configuration instructions are VNF related, the configuration SSM does its configuration by sending a task list to the SLM. This list contains tasks of which the SLM knows how to execute them, such as 'start a VNF', 'configure a VNF' and 'terminate a VNF' (See section Section 4.2 for more information on these workflows). These tasks are calls from the SLM to the FLM. The FLM has a workflow associated with these calls, which might use FSMs (which are in direct communication with the FLM). Once the configuration is completed, the SLM responds to the request with an indication on the result of the configuration.

The workflow that configures a workflow has the following tasks:

1. Contact the configuration SSM when a configuration request is received.
2. Perform the task list that is received from the configuration SSM.
3. Respond to the actor that made the configuration request whether the workflow finished correctly or not.

4.1.6 Monitoring a service

When a service is correctly deployed, it is being monitored by the Monitoring Manager, and the SLM is receiving messages from the Monitoring Manager that give indications on the performance of the service and its components. This section will describe how the SLM deals with these messages. The SLM divides these messages in two groups:

1. The messages that require a generic response from the SLM. An example is a message that indicates that a VNF is no longer running.
2. The messages that require a customized response. An example is a message that indicates that a VNF is using more than 80% of its provisioned resources.

Messages from the first category will result in a generic response, i.e. the behavior of the SLM upon receiving such a message is identical for all deployed functions. For example, if the SLM receives a message that indicates that a VNF is no longer running, the SLM will:

1. Collect the VNFR of the VNF and the NSR of the associated service;
2. Update the status of the VNF and the service to “offline” or “interrupted”;
3. Store the new records.

Once the records are updated, the customer that is using the service will be able to see on the BSS that the service is no longer running. At this point, the customer can terminate the service, or take another action.

Messages from the second category require a customized response, which in SONATA is done through SSMs. So, if the SLM receives a monitoring message that can't be handled in a generic way, it will forward it to a Monitoring SSM. This Monitoring SSM can be seen as a special case of a Configuration SSM (see section Section 4.1.5). The Monitoring SSM will respond to the SLM with a task list, which contains a set of tasks that the SLM should execute in response to receiving this monitoring message. Such tasks could include “Deploy a VNF”, “Terminate a VNF” or “Terminate the service”.

4.2 FLM

The next release of the SONATA SP will contain the **Function Lifecycle Manager** (FLM) as individual component (i.e. Docker container). The FLM is part of the MANO framework, and its concept was already introduced in Deliverables 4.1 ([6]) and 4.2 ([7]). Before this release, the FLM functionality was incorporated inside the SLM, but from SONATA release 3.0 onwards, the FLM is integrated as a separate component.

Whereas the SLM manages life cycles on the service level, the FLM manages life cycles on the function level. Where the SLM makes use of SSMs to allow the customization of these life cycles by the service developer, the FLM uses Function Specific Managers (FSM) to provide this feature. Due to these parallels between the SLM and the FLM, the FLM is tailored after the SLM. Like the SLM, it is implemented as a workflow based task manager. It supports a set of workflows, which are translated into a set of ordered tasks. This set of ordered tasks associated with a workflow is customizable based on or by FSMs.

The remainder of this section is used to describe how the different FLM supported workflows are built.

4.2.1 Deploy a VNF

Since the FLM is managing life cycles of VNFs, it needs a workflow to deploy a VNF. This workflow is triggered by sending a message on the `mano.function.deploy` topic over the message broker. The message is a YAML encoded dictionary. All the keys shown in the Table 4.2 should be provided in this dictionary.

Table 4.2: Payload for a VNF deploy request.

Key	Value
vnfd	The entire VNFD in dictionary format, augmented with the key 'instance_uuid', which has the instance id of this VNF as value.
vim_uuid	The id of the VIM on which this VNF should be deployed. This id is obtained by the SLM by calculating the placement of each VNF after it received the topology from the IA.
service_instance_id	The id of the service that the VNF will be part of.

Once the FLM receives such a message on the correct topic, the following ordered task list is executed:

1. Requesting the SMR to onboard the FSMs. Based on the presence of an FSM in the VNFD, the FLM will request the SMR to onboard them. For more information on this API, see Section 4.5.

2. Instantiation of the FSMs. Based on the presence of an FSM in the VNFD, the FLM will request the SMR to instantiate them. For more information on this API, see Section 4.5.

3. Consulting the task FSM, if available, whether the tasks in this workflow should be updated or not. More information on this process can be found in section Section 4.1.1. A possible update could be that after the deployment of the VNF is done, a configuration FSM is used to configure it in an additional task, instead of just executing the default tasks.

4. Deployment of the VNF. The FLM will request the IA to deploy the VNF. The API for this call can be found in Deliverable 4.2 ([7]).

5. Generation and storage of the function record. This record is generated in accordance with the function record schema which can be found in Deliverable 3.3 ([8]). The record is stored in the repository, of which the API is described in Deliverable 4.2 ([7]).

6. Reporting the status of the request to the SLM. The payload of this response is a YAML dictionary, of which the content is shown in the Table 4.3.

Table 4.3: Payload for a VNF deploy response.

Key	Value
vnfr	The function record formatted as a dictionary
status	The status of the VNF deployment. This is a direct copy of the status the FLM received from the IA deployment call.
error	Any errors that might have occurred during the deployment. This value is 'None' if no errors occurred.

The Figure 4.3 shows the different actors and their communication for the function deploy workflow.

4.2.2 Terminate a VNF

When a service is terminated, the FLM is required to provide a clean VNF termination workflow. The termination workflow is triggered by the SLM through sending a message on the `mano.function.kill` topic. The payload for this message is a YAML encoded dictionary, constructed as described in the Table 4.4.

Table 4.4: Payload for a VNF terminate request.

Key	Value
vnf_uuid	The instance id that references the VNF that needs to be terminated.

The termination workflow is composed by these tasks:

1. Request the IA to terminate the VNF. The FLM makes this request on the topic:

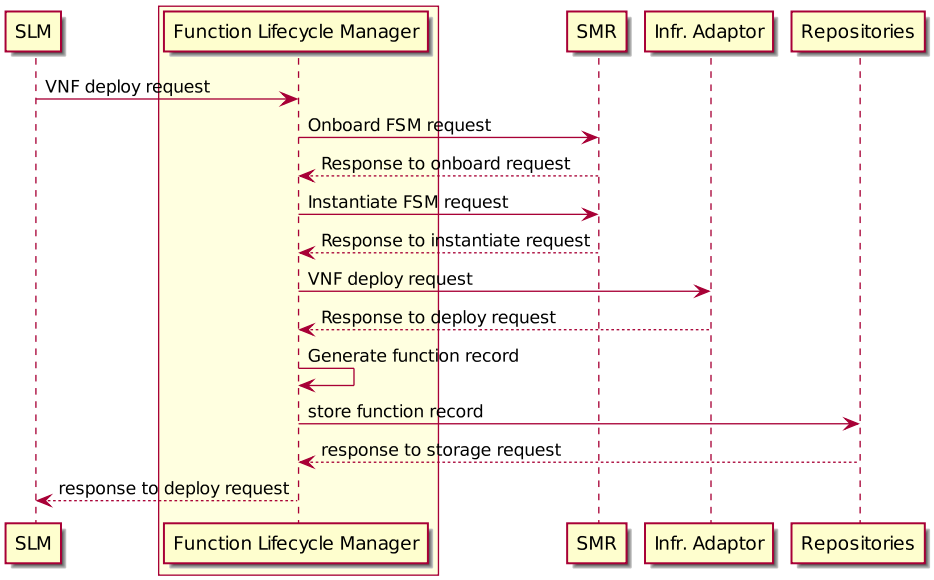

Figure 4.3: Deploying a VNF

`infrastructure.function.terminate`

The payload of this message is a YAML encoded dictionary, shown in the Table 4.5.

2. Update the function record. The API for this call can be found in Deliverable 4.2 ([7]).
3. Terminate the FSMs. For more information on this API, see Section 4.5.
4. Report the termination status to the SLM. The payload for this response is a YAML encoded dictionary, as shown in the Table 4.6.

Table 4.5: Payload for a VNF terminate request from FLM to the IA .

Key	Value
vnfr	The function record
vim_uuid	The VIM id that is hosting the function.

Table 4.6: Payload for a VNF terminate response.

Key	Value
vnfr	The updated function record.
status	The status of the termination workflow.
error	An error message that indicates what errors have occurred. This value is 'None' if no error occurred.

The different actors and their communication for the termination workflow are shown in the Figure 4.4.

4.2.3 Start, Stop or Configure a VNF

Some VNFs require some further instructions after they are deployed, to get them running correctly or to stop them. These function specific inputs can be implemented in an FSM. Since such an FSM

is described in the VNFD, it is onboarded and instantiated during the VNF deploy workflow and available from that point onwards. We have identified three categories of instructions:

- Instructions to start the VNF. This could be something like setting up an SSH connection with the VNF and starting a process.
- Instructions to stop the VNF. This could be collecting some data from the VNF and then stopping a process.
- Instructions to configure the VNF. This could be something like setting up an SSH connection with the VNF and inserting configuration information, such as relevant IP addresses for the other VNFs, for the Monitoring Server, ...

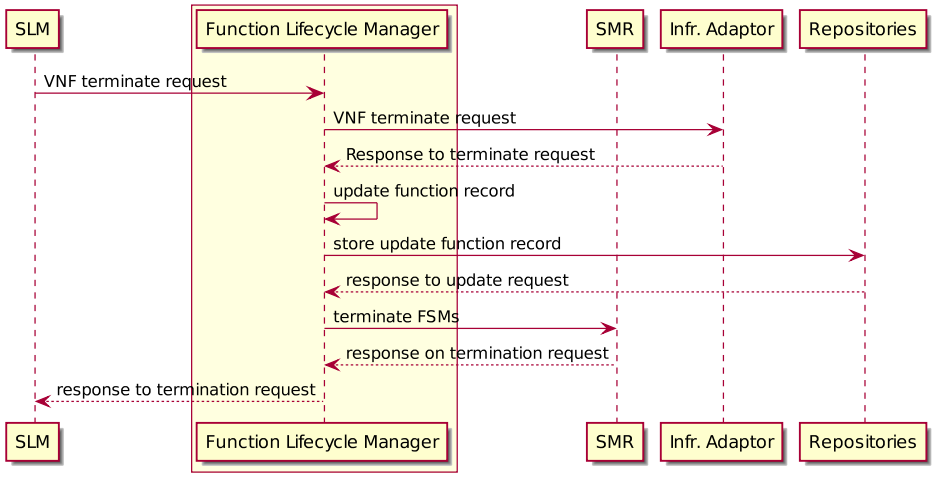

Figure 4.4: Terminating a VNF

When the FLM receives a request to perform any of those instructions, it will forward the relevant payload of the request to the corresponding FSM, which will then be responsible to execute the instructions. The FLM is monitoring the following topics for these requests:

- `mano.function.start`
- `mano.function.stop`
- `mano.function.configure`

These workflows can also be triggered from within the FLM. For example, an FSM can customize the function deploy workflow so that the start workflow is executed as one of the tasks. The payload for all three requests is identical. It is a YAML encoded dictionary with two keys, as shown in the Table 4.7.

Table 4.7: Payload for a VNF start/stop/configure request.

Key	Value
vnf_instance_id	The VNF instance id that the FLM can use to determine which deployed VNF this request belongs to.
data	The content that needs to be included in the request to the FSM.

After the FLM makes a request to the corresponding FSM, it awaits a response. The Section 4.5 describes how both FLM and FSM figure out which topic they should use to communicate. The response from the FSM to the FLM should contain a YAML encoded dictionary with two keys, as shown in the Table 4.8.

Table 4.8: Payload for an FSM response.

Key	Value
status	Either 'SUCCESSFUL' or 'ERROR'
error	Either 'None' or a message that indicates the error.

The FLM will then forward this message as a response to the request it received.

4.2.4 Scale a VNF

The FLM can, instructed by an FSM, use the scaling API that is provided by the IA (see section Section 5.1.1) to scale a single VNF. More information on this scaling process/workflow can be found in section Section 4.4.

4.3 Placement Plugin

The placement plugin is a new plugin that we introduced in our MANO framework since Deliverable 4.2. It can be consulted by other MANO plugins to perform placement calculations, i.e. mapping VNFs on the available resources. For example, looking at the service instantiation workflow that is described in Section 4.1, the SLM consults the placement plugin when it requires an in-house mapping. The placement plugin receives a topology of resources and a topology of the service as input, and maps the components of the service on the resource topology by using some placement algorithm. As a wide variety of placement algorithms is available, one can simply change the mapping behaviour of the SP by replacing the plugin with a new placement plugin that implements a different placement algorithm. Since the Placement Plugin is a general plugin of the MANO Framework, this replacement can only be done by the SP owner. Customization of placement logic by the service developer is done through the use of SSMs.

In most cases, the mapping algorithm requires both the NSD and the VNFDs, as the NSD contains the service chain, and the VNFDs contain the resource requirements (e.g. required cores, required memory, etc.) of the individual VNFs.

The placement plugin listens for placement requests on the 'mano.service.place' topic. The payload of such messages should be a YAML encoded dictionary, of which the key-value pairs are shown in the Table 4.9.

Table 4.9: Payload for a service placement request.

Key	Value
nsd	The NSD of the service, represented as a dictionary.
functions	A list of the VNFDs of all the different VNFs in the service. Each VNFD is represented as a dictionary, augmented with the 'id' key which has the function instance id as value.
topology	The topology of available resources. This is a dictionary with the ids of the available VIMs as keys and a dictionary with the availability of the resources of that VIM as value. This resource availability dictionary has keys like 'core_used', 'core_total', 'memory_used' and 'memory_total'.

Once the placement is calculated, the placement plugin responds to the request on the same topic. In case the placement could not be performed due to any reason (e.g. not enough resources available to accommodate the service, incorrect format of the request payload, etc.), the payload of the response is 'None'. If the placement is successful, the response payload is a YAML encoded dictionary. This dictionary has each VNF instance id of a mapped function as keys, with the VIM id they should be mapped on as value.

4.4 Scaling

In this section we describe the workflows that the SP has to support to be able to provide scaling capabilities.

The service scaling process can be triggered, for example, as a result of an alert issued by the monitoring system. This trigger can indicate, e.g., that a VNF is exhausting its resources. The first step is to check whether this resource scarcity can be overcome by scaling the components in the VNF. This scaling is VNF-specific and should be done by an FSM. In case the FSM cannot resolve the issue by scaling the components in the VNF, or if no FSM for this VNF is available, the SP will try a service-level scaling, i.e., by adding new instances of the VNF.

We believe the way service components can handle horizontal scaling is highly service-specific and depend on factors like statefulness of the VNF, if an application-specific load balancing is required, what optimization requirements the service has, etc. Therefore, we do not plan to offer default scaling mechanisms embedded in the SP, meaning that the scaling description of a function/service should be described and implemented by the developer of the function/service in FSMs/SSMs. Both changing the components of the VNF and changing the service chain should be translated into instructions for the IA.

Currently, we are planning to support scaling at the function level, using a Mistral-based function lifecycle management process.

4.4.1 Scaling at the Function Level

Figure 4.5 shows the workflow between MANO and IA, in case there is a FSM available that can scale the VNF that is running out of resources. Changing the VNF components (VNFCs) architecture of a VNF is done by the FSM. The APIs that the MANO framework requires from the IA to support this are:

- Add VNF
- Remove VNF
- Add forwarding path
- Remove forwarding path

The FSM uses the VNFR and the VNFD to calculate a scaled version of the VNF, i.e., a VNF with a different VNFC architecture. The FSM outputs the updated function description to the FLM. This description is the input when the FLM requests the IA to change the architecture of the VNF. The new VNF architecture needs to be included in the new VNFR, this gives the FSM the current view when it needs to scale the VNF again.

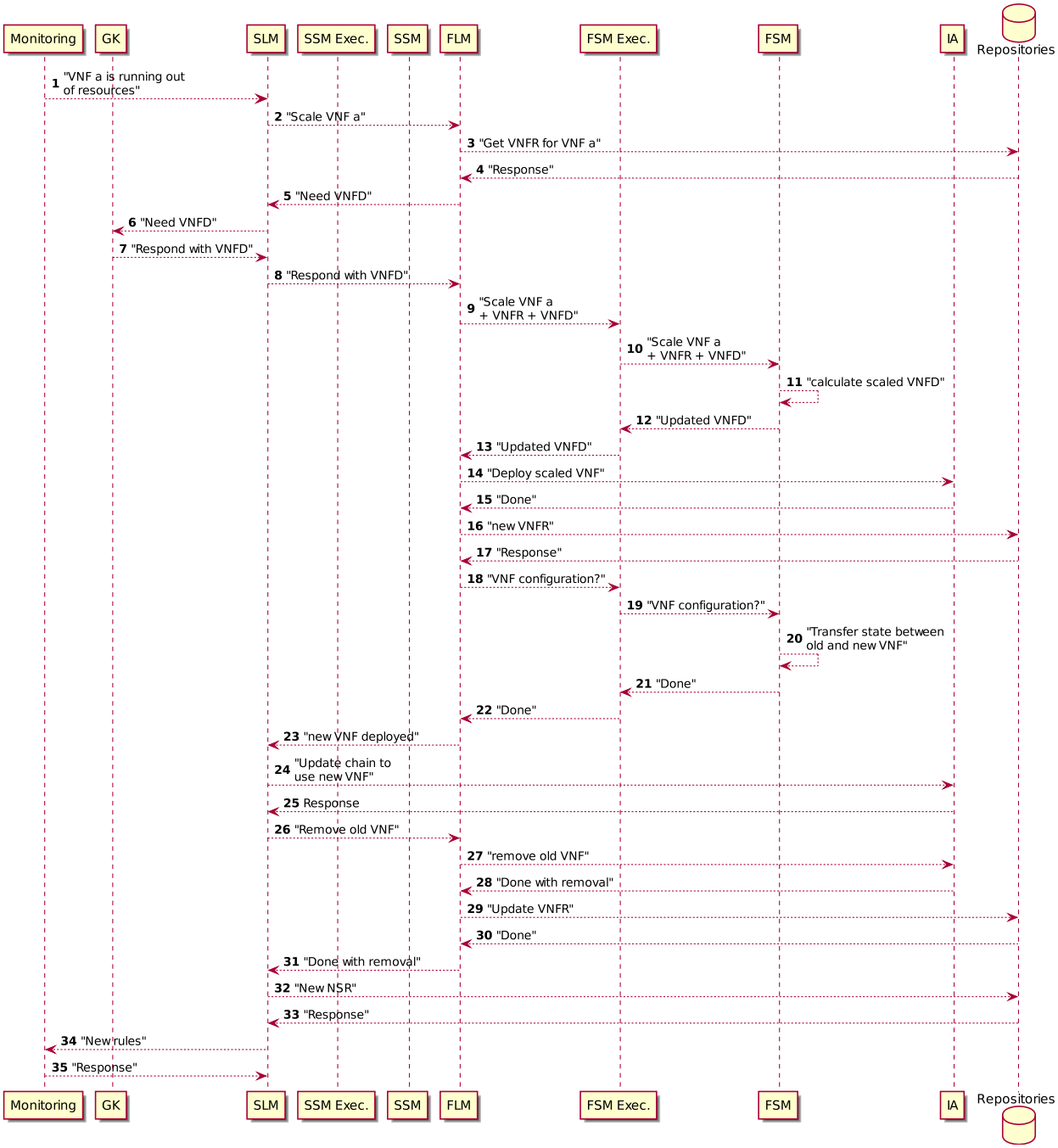

Figure 4.5: Scaling using a FSM

4.4.2 Scaling on Service Level

Figure 4.6 shows the workflow between MANO and IA, in case there is no FSM that can scale the VNF.

In case additional instances of a VNF are added to the service, a load balancing mechanism is required to distribute the traffic among instances. The load balancer can be configured by the FSM to split the traffic between its exit paths, or the load balancer can use the network so that the network makes sure that the packages arrive at the right instance. The load balancing logic depends on the type of the VNFs and the running application, so it should be included in the original service description.

The SSM is responsible for constructing the new forwarding graph for the service. The SSM requires the most recent NSR, the original NSD and topology information for these calculations. The new forwarding graph will be the input when the SLM requests the IA to construct the new service function chain. The new forwarding graph needs to be included in the NSR, which gives the scaling SSM the current view when it needs to scale the service again later.

4.5 Specific Manager Infrastructure

This section explains the infrastructure that is provided in SONATA's MANO framework to support FSMs and SSMs plugins.

4.5.1 Specific Manager Registry

As explained in the previous deliverable, Specific Manager Registry (SMR) is a MANO framework plugin that is responsible for managing the lifecycle of FSMs/SSMs including onboarding, instantiating, updating and terminating. The workflows of SMR functionalities have been updated since the last deliverable in order to support the management of a large number of FSMs/SSMs. The following describes the updated that is provided in the SMR workflow.

4.5.1.1 FSM/SSM deployment

The FSM/SSM deployment workflow consists of two major steps including the FSMs/SSMs onboarding, which downloads the FSMs/SSMs Docker containers from the corresponding repository, and instantiation, that is responsible for starting FSM/SSM containers. All FSMs/SSMs belonging to a service are deployed during the service deployment at once and will be triggered later on by their associated events.

To do the SSM deployment, SLM forwards the service descriptor (NSD) within an onboarding request message to the SMR. Then, SMR on-boards all the SSMs that are described in the NSD and sends a response back to the SLM. Once the SLM generates the service UUID, it sends an instantiation request to the SMR containing the NSD and the service UUID. When SMR receives the request message, it instantiates all the SSMs included in the NSD and send the request response back to SLM.

The same workflow is performed for FSM deployment with the difference that the FLM is responsible for sending the request messages, and the messages contain VNFD and VNF UUID instead of NSD and service UUID, respectively. Figure 4.7 and Figure 4.8 show the workflow of SSMs and FSMs deployment, respectively.

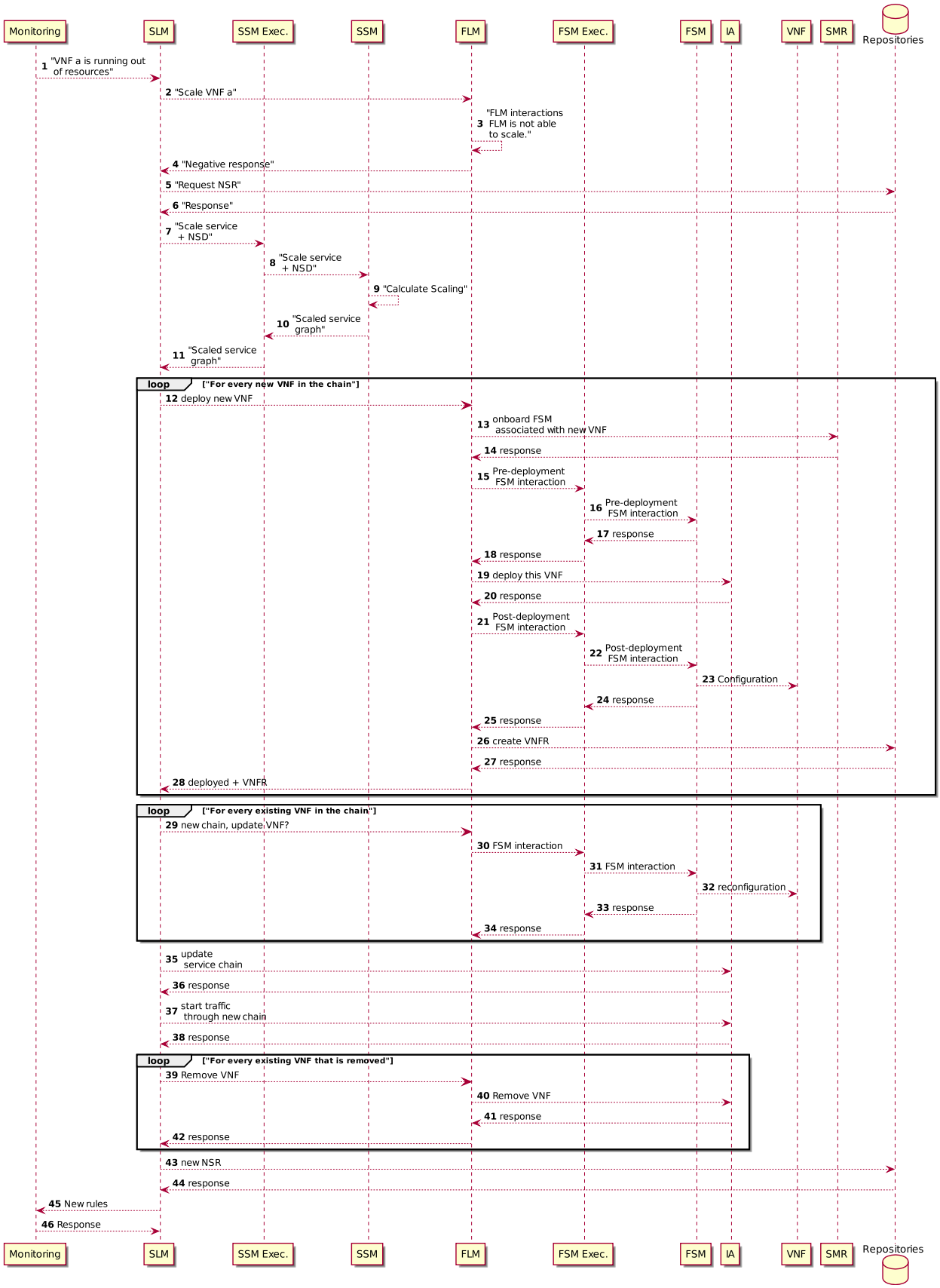

Figure 4.6: Scaling using a SSM

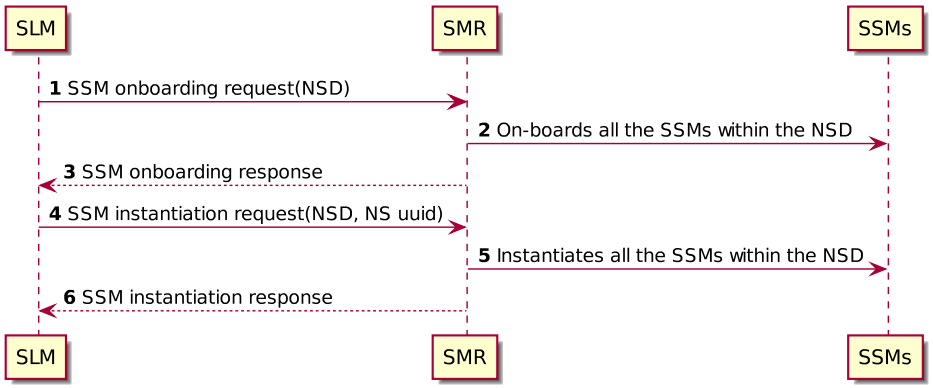

Figure 4.7: Sequence diagram for SSMs deployment

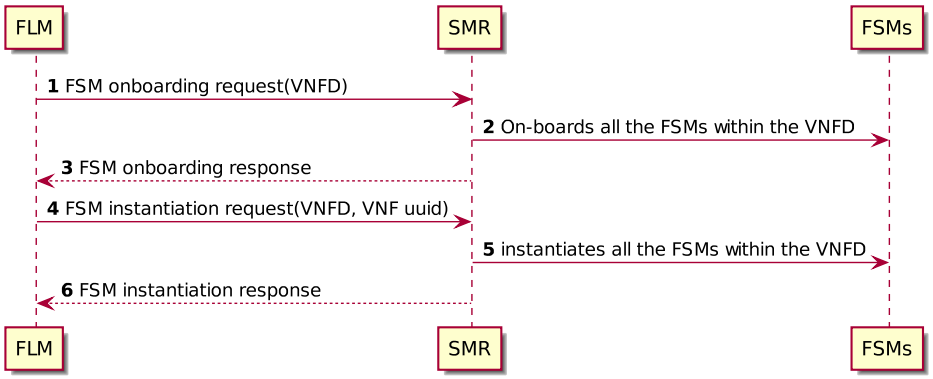

Figure 4.8: Sequence diagram for FSMs deployment

4.5.1.2 FSM/SSM Updating

FSM/SSM updating is the second use case that is supported by SMR. To update an SSM/FSM, the developer needs to update the NSD/VNFD with a new version of the SSM/FSM. The new version of SSM/FSM should be linked to the current version so that the SMR can find out which SSM/FSM must be updated. The developer can provide this linking by using the key-value pair item within the SSM/FSM section of the descriptors. The following is an example that associates the new SSM/FSM with the current one.

```
service_specific_managers:
  - id: "sonfsmservice1firewallplacement2"
    description: "Placement FSM"
    image: "hadik3r/sonfsmservice1firewallplacement2"
    options:
      - key: "currentId"
        value: "sonfsmservice1firewallplacement1"
      - key: "currentImage"
        value: "hadik3r/sonfsmservice1firewallplacement1"
```

To update FSMs/SSMs two cases are considered including: (i) when the new FSM/SSM has the exact same "id" as the current FSM/SSM and (ii) the new SSM has a different "id" than the current FSM/SSM. For the first case, SMR instantiates the updated FSM/SSM with a random id and then rename it to its original id once the current FSM/SSM is successfully terminated. However, since in the second case, ids are different, and no name collision happens, SMR simply instantiates the updated SSM/FSM with its id and then terminate the current one. The workflow for updating SSMs and FSMs are shown in Figure 4.9 and Figure 4.10, respectively.

4.5.1.3 SSM/FSM Termination

The SSM/FSM termination can also be done through updating the descriptor. Using the key value pairs provided in the descriptor, the developer can specify an SSM/FSM to be terminated. An example is shown in the following description:

```
service_specific_managers:
  - id: "sonfsmservice1function1monitoring1"
    description: "Monitoring FSM"
    image: "hadik3r/sonfsmservice1function1monitoring1"
    options:
      - key: "termination"
        value: "true"
```

Figure 4.9 and Figure 4.10 show the workflow for terminating SSMs and FSMs, respectively.

4.5.2 SSM/FSM Executives

SSMs/FSMs are service developer-made components that can change the behaviour of the service platform. This raises security concerns since SSMs/FSMs also enable service developers to inject any malicious operation into the service platform. To tackle this issue, we developed plugins, called SSM/FSM executives, that are responsible for inspecting all the messages originated from

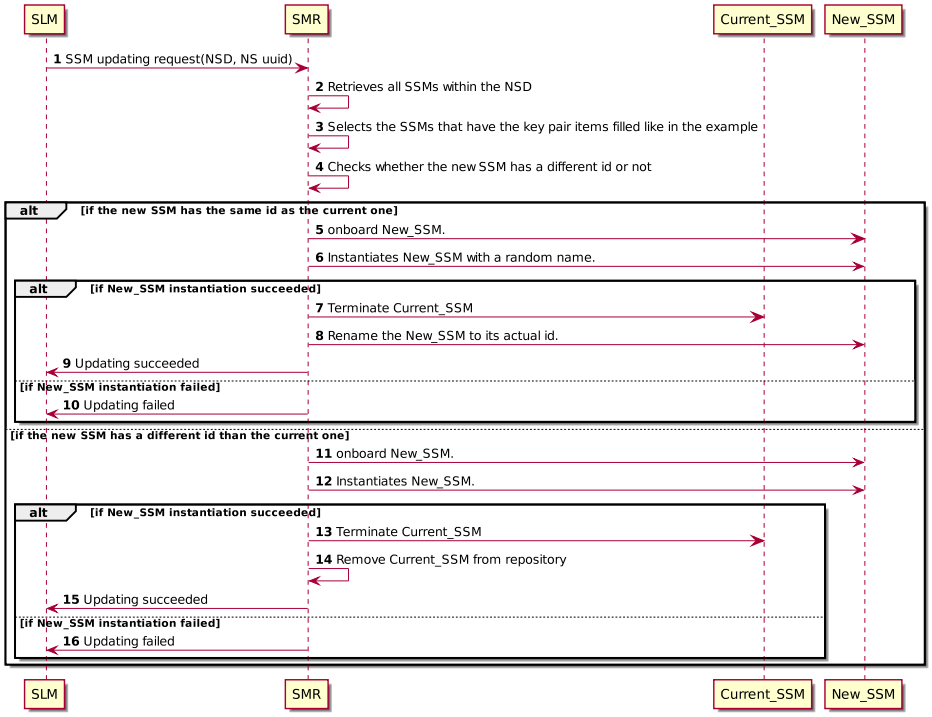

Figure 4.9: Sequence diagram for SSM updating

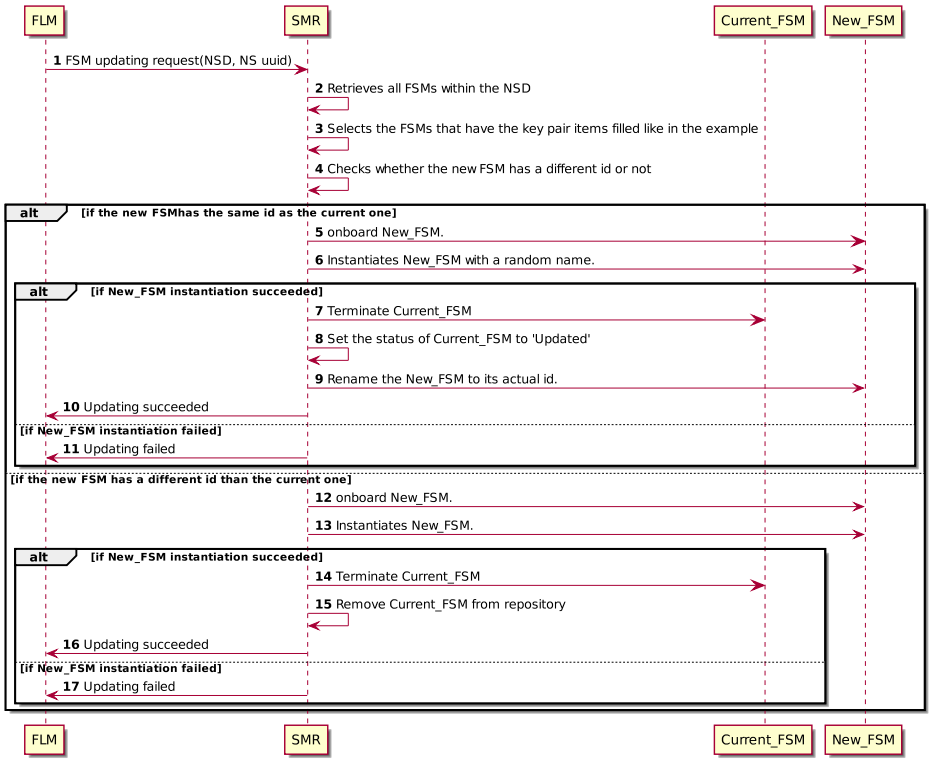

Figure 4.10: Sequence diagram for FSM updating

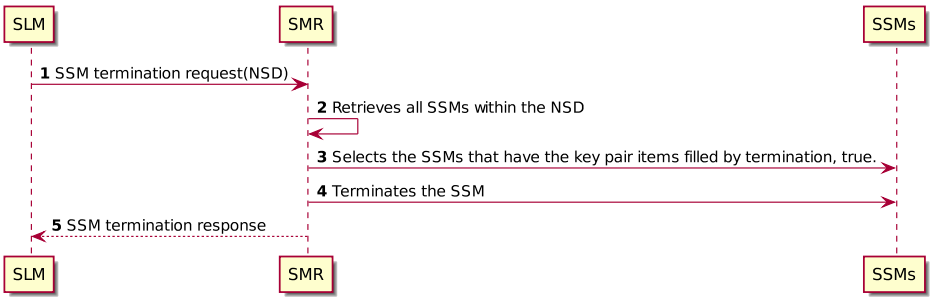

Figure 4.11: Sequence diagram for SSMs termination

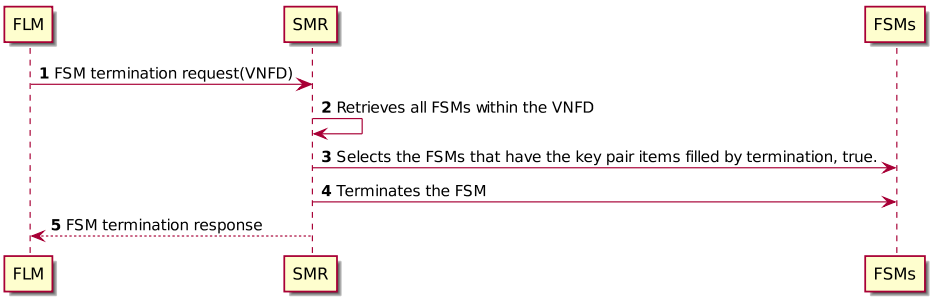

Figure 4.12: Sequence diagram for FSM termination

SSMs/FSMs or destined for the SSMs/FSMs. These plugins avoid injecting malicious operation to the service platform and also exposing sensitive infrastructure data to the SSMs/FSMs. The followings are the executives that we developed in this regard.

4.5.2.1 Placement executive

Placement executive inspects the messages related to placement SSMs. It exposes an API that will be used by placement SSMs to send and receive messages. The API is a RabbitMQ topic which is as follow:

Placement topic: `placement.ssm.[sevice_uuid]`

`service_uuid` is the uuid of the service that the SSM belongs to. Using the service uuid in topics, we isolate messages owned by different services.

4.5.2.2 Scaling executive

Scaling executive is provided to inspect messages originated from scaling FSMs or destained for the scaling FSMs. The topic exposed by this executive is the following:

Scaling topic: `scaling.fsm.[function_uuid]`

`function_uuid` is the uuid of the VNF that the FSM belongs to. Using the function uuid in topics, we isolate messages owned by different VNFs.

4.5.2.3 FLM

Besides other tasks that have already mentioned for FLM, it also takes care of inspecting the messages of any FSM that does not correspond to scaling FSMs. The topics exposed by this plugin is as follow:

- FLM generic topic:

- `generic.fsm.[function_uuid]`

4.5.2.4 SLM

Besides other tasks that have already mentioned for SLM, it also takes care of inspecting the messages of any SSM that does not correspond to placement SSMs. The topics exposed by this plugin is as follow:

- SLM generic topic:
  - `generic.ssm.[service_uuid]`

5 Infrastructure Abstraction

This section describes the features and advances included in the latest version of the Infrastructure Abstraction layer. The section is divided into two sub-sections, dealing respectively with the VIM-Adaptor and the WIM-Adaptor.

5.1 VIM Adaptor

With respect to the description provided in D4.1[6] and D4.2[7], Several improvements have been introduced in the generic part of the Vim-Adaptor, which deals with the messages decoding, requests handling and wrappers calls, and NFVI abstract data model.

- A caching system for Wrapper has been introduced so to avoid retrieving relevant information from the database at each request, which allows better performance in terms of request handling and a more fine-tuned internal management of parallel call handling. With the introduction of such system, the VIM-Adaptor is able to deploy VNF in parallel across different NFVI-PoP. Anyway, some limitation can exist depending on the specific VIM that is called to deploy a VNF, therefore, to keep the model generic, parallel VNF spawning for the same Network Service on the same NFVI-PoP are handled sequentially. More specifically, if the MANO framework requests in parallel the instantiation of two VNFs for the same NS to be deployed on two separate NFVI-PoP, the requests are handled completely in parallel, and as well as the actual VNFs spawning on the VIM. On the contrary, if the MANO framework requests in parallel the instantiation of two VNFs for the same NS and to be deployed on the same NFVI-PoP, the requests are again handled in parallel, but the actual spawning of the VNFs on the VIM is synced, queued and handled serially. This is necessary because some VIMs (like OpenStack) are not ensuring the consistency of their manifest for virtual resources (e.g. the Heat Template) when parallel request for an update are issued. Therefore the parallel deployment of two VNFs on the same VIM will issue two concurrent requests for the virtual resource manifest to be first retrieved and then updated. The first request could edit the manifest after the slowest request retrieved the current manifest status, leaving the slowest request on the race with an inconsistent manifest to update, and thus generating the wrong update request. For this reason this process has been synchronised at the lowest possible abstraction level in the IA, that is the VIM-specific function for VNF deployment offered by the VIM Compute Wrapper interface.
- The NFVI-PoP data model has been extended to include further information on the location of the computational and networking resource, which allows the development of location aware placement plug-ins.
- The VIM-Adaptor northbound interface has been integrated with the MANO framework for the lifecycle steps involving the VM/Container images pre-provisioning on the NFVI-PoP involved in a service instance deployment. Using the results of VNFs placement phase together with the information provided in the NSD and VNFDs, SLM provides to the IA a list of the images that needs to be pre-provisioned in each NFVI-PoP. The IA use this list

to retrieve the relevant images from the image-store, or in general from the provided URL, and pushes them to the specified NFVI-PoP image repository (i.e. Glance or Docker image repository).

5.1.1 VIM Adaptor API Reference

The VIM Adaptor API has been improved to support the features just described.

Add Compute VIM

Register a Compute VIM to the VIM-Adaptor.

topic: `infrastructure.management.compute.add`

data: { `vim_type: String, configuration: { tenant_ext_router: String, tenant_ext_net: String, tenant: String }, city: String, country: String, vim_address: String, username: String, pass: String }`

return: {`request_status: String, uuid: String, message: String`}, when *request_status* is "COMPLETED", *uuid* fields carries the UUID of the registered VIM and *message* field is null, when *request_status* is "ERROR", *message* field carries a string with the error message, and the *uuid* field is empty.

Add Network VIM

Register a Network VIM to the VIM-Adaptor, also specifying the Compute VIM to which it is attached.

topic: `infrastructure.management.network.add`

data: { `vim_type: String, vim_address: String, username: String, city: String, country: String, pass: String, configuration: { compute_uuid: String } }`, compute_uuid must contain the UUID of a compute VIM already registered to the platform.

return: {`request_status: String, uuid: String, message: String`}, when *request_status* is "COMPLETED", *uuid* fields carries the UUID of the registered VIM and *message* field is null, when *request_status* is "ERROR", *message* field carries a string with the error message, and the *uuid* field is empty.

List Compute VIM

Return a list of the registered compute VIMs, along with basic information on the resource totally available and consumed in this VIM.

topic: `infrastructure.management.compute.list`

data: `null`

return: {`[{vim_uuid: String, vim_city: String, vim_name: String, vim_endpoint: String, memory_total: int, memory_used: int, core_total: int, core_used: int}]`}

List Network VIM

Return a list of the registered Network VIMs.

topic: `infrastructure.management.network.list`

data: `null`

return: {`[ { vim_uuid: String, vim_city: String, vim_name: String, vim_endpoint: String } ]`}

Remove VIM

Remove a VIM from the the VIM-Adaptor, being it a Network VIM or a Compute VIM. If a compute VIM is removed, also the Network VIM attached to it will be removed.

topic: `infrastructure.management.{network,compute}.remove`

data: {`uuid:String`}

return: {`request_status: String, message: String`}, when *request_status* is "COMPLETED", *message* field is empty, when *request_status* is "ERROR", message field carries a string with the error message.

Prepare NFVI for service deployment

Prepare a list of NFVI-PoP for the deployment of a NS instance. This includes pre-deploying VNF images in the relevant image repository (e.g. Glance or Docker image repository) and creating network facilities to which VNFs will be attached.

topic: `infrastructure.service.prepare`

data: {`instance_id: String, vim_list: [{uuid: String, vm_images: [{image_uuid: String, image_url: String}]]}]`}

return: {`request_status: String, message: String`}, when *request_status* is "COMPLETED", *message* field is empty, when *request_status* is "ERROR", message field carries a string with the error message.

Deploy a VNF instance for a Service

Deploy a VNF given the relevant VNFD, the UUID of the compute VIM in which the VNF must be deployed and the service instance ID of the relevant network service instance. This call returns an incomplete VNF Record containing the field that are generated at the infrastructural level, such as VNFC number and identifiers, IP and MAC addresses, etc.

topic: `infrastructure.function.deploy`

data: {`vim_uuid: String, service_instance_id: String, vnfd: SonataVNFDescriptor`}

return: { `instanceName: String, instanceVimUuid: String, vimUuid: String, request_status: String, vnfr: SonataVNFRecord` }

Scale a VNF instance for a Service

Scaling a VNF given the VNF instance ID and a list VDUs and the updated number of instances per VDU. This call returns an incomplete VNF Record containing the updated fields that are generated at the infrastructural level.

topic: `infrastructure.function.scale`

data: {`vnf_instance_id: String, vdus: [{ vdu_id: String, updated_instances_number: String }]`}

return: {`request_status: String, message: String, vnfr: SonataVNFRecord`} when *request_status* is "COMPLETED", *message* field is empty, when *request_status* is "ERROR", *message* field carries a string with the error message.

Configure intra-PoP chaining

For each NFVI-PoP involved in the deployment of a network service, the function chaining between VNFs deployed in the same PoP is configured following the specification contained in the NS Forwarding graph and configured for the flows specified in the *ingress_nap* and *egress_nap* list, which is retrieved from the information specified by the SP user during the service instantiation request described in the Section 2.5.

topic: `infrastructure.chain.configure`
data:

```
{
 service_instance_id: String,
 nsd: SonataNSDescriptor,
 vnfds: [{ SonataVNFDescriptor }],
 vnfrs: [{ SonataVNFRecord }],
 ingress_nap: [{ segment:String }],
 egress_nap: [{ segment:String }]
}
```

return: {`request_status: String, message: String`}, when *request_status* is "COMPLETED", *message* field is empty, when *request_status* is "ERROR", message field carries a string with the error message.

Deconfigure intra-PoP chaining

Remove the SFC rules in all NFVI-PoP involved in the deployment of a specific network service instance.
topic: `infrastructure.chain.deconfigure`
data: {`service_instance_id: String`}
return: {`request_status: String, message: String`}, when request_status is "COMPLETED" message field is empty, when request_status is "ERROR", message field carries a string with the error message.

Remove Service instance

Remove all the VNFs instances of a given Network Service instance from the whole NFVI.
topic: `infrastructure.service.remove`
data: {`instance_id: String`}
return: {`request_status: String, message: String`}, when request_status is "COMPLETED" message field is empty, when request_status is "ERROR", message field carries a string with the error message.

5.1.2 VNF Scaling API and Mistral Integration

To perform the (usually) complex flow required for scaling a VNF, a workflow engine is used. The workflow used is Mistral [4]. It is important to note that despite Mistral being an OpenStack initiative, it is a completely stand alone workflow engine that can be installed separately and perform a very rich set of operations (see [4]). Obviously it supports out of the box many OpenStack operations required for scaling.

The Mistral Workflow Engine is running on a docker container which is part of the service platform installation. Mistral uses a MYSQL data base, which is running on another docker container which is also part of the service platform installation.

The infrastructure abstraction layer was enhanced with an additional client that communicated with Mistral via REST-API. This Mistral Client implemented in Java is the foundation for a future client that will be developed by OpenStack community members as part of OpenStack4J [5].

Figure 5.1 presents in high level the flow of scaling by using Mistral. In part of the flow, when the IA is requested by the FLM to perform a VNF scaling operation, it produces the relevant workflow and requested Mistral to execute it.

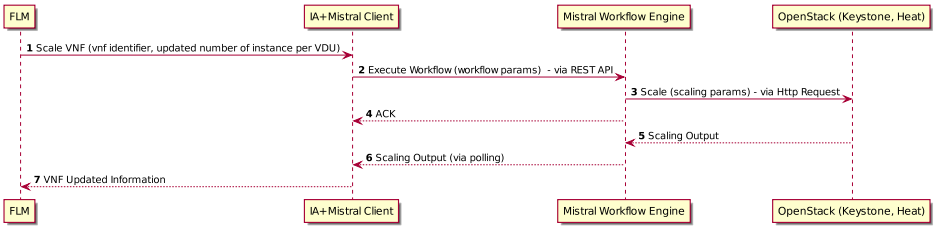

Figure 5.1: Scaling a VNF Using Mistral

5.1.3 OpenStack Compute wrapper V3

The Vim Wrapper for OpenStack has been improved in several parts.

1. The Java implementation of the OpenStack API client has been further extended to improve performances when handling multiple requests in parallel and to automatically set endpoints of the several OpenStack micro-services (i.e. Nova, Heat, Glance, etc.) after authenticating with Keystone.

2. The translation model has been further refined to avoid redundant Neutron Port belonging to the same VM and connected to the same Neutron sub-net. This design was introduced to maintain a one-to-one mapping between connection points listed in the VNFD and actual network interfaces configured in a VM. Anyway, given the advances in the SFC model introduced in [7] and further extended in the following section, this mapping results redundant, and it is possible to limit the number of ports for each VM connected to a given network to one, this optimising the number of Heat Resources deployed and also avoiding performance problem that could arise when multiple interfaces of the same VM are connected to the same network.

3. The translation model has been also extended to use Heat *ResourceGroup* resource type when mapping VDUs to VMs. Whether the previous model directly maps a VDU to a single VM, wrapping the VM into Heat ResourceGroup allows the IA to easily specify the number of instance of the VM to be deployed. This nicely maps to the relation existing between VDU and VNFC in the ETSI NFV model, and also allows to easily scale in and out VNFs by simply changing the number of instances created using the template resource provided. Therefore, this extension simplifies the implementation of the IA Scaling API, and also allows an easier integration between the IA and the Mistral workflow manager.

5.1.4 Docker Compute wrapper

In deliverable D4.2 [7] we presented a first translation model between the SONATA NFV data model and the Kubernetes Container Orchestrator[2]. Further investigation and prototyping has lead us to the conclusion that the networking facilities currently available in the Kubernetes data model do not allow an acceptable mapping of the entities of SONATA model, and in general of the NFV paradigm. In fact, containers grouped in pods cannot expose more than one interface, and the network data model used by kubernetes network agent does not allow to bind more than one network to a given container. This makes impossible at the current status of the kubernetes development to map VNFs into kubernetes entities. Nonetheless, the aim of running SONATA services on Docker based VIMs persists, therefore we moved the focus of our investigation from kubernetes to a native Docker engine (It is worth noting that, given the extensible nature of the IA, future versions of kubernetes with improved features in terms of networking capabilities could

be easily integrated with SONATA following and extending the model presented in [7]). Therefore this section presents a translation model, along with a description of the workflows of a Compute wrapper based on Docker-engine VIM. The first important difference between the Docker and the OpenStack wrapper is that Docker does not offer the equivalent of a stack orchestration facility (i.e. Heat), therefore the entities that compose the service at the VIM level (containers, ports, networks, volumes) must be instantiated and managed separately. Anyway, the deployment model introduced in version two of the IA easily maps to handle this fine-tuned deployment. Moreover, a Java library that embeds the docker native API is available on maven repository [3] to facilitate the integration with the Docker engine endpoint. The deployment flow illustrated in D4.2 could be implemented in the following way:

1. `infrastructure.service.prepare`
    - Container images listed under the PoP id of the Docker VIM are pulled from the given *image_url* and pushed in the Docker VIM image repository, using the *image_uuid* provided by the SLM as name for the image.
    - **Two** Docker networks are created for the service data plane and management plane using the *service_instance_id* to build consistent naming for these networks, using the same rationale followed in the OpenStack Wrapper, which has been described in [7].
2. `infrastructure.function.deploy`
    - For each VDU listed in the VNFD, the Docker VIM deploys the relevant container image and connects it both to the data plane and the management plane. Again, naming is used to map the container to the VNF instance ID combined to an index to allow multiple VNFC based on the same VDU, so to facilitate scaling in and out.
3. `infrastructure.chain.*`
    - Since the Docker engine networking is essentially based on OpenVSwitch, the integration with the OVS Network Wrapper used together with the OpenStack VIM is seamless, as the same configuration of OVS will allow traffic to flow through container in the same way it flows through the VMs.

5.1.5 OVS Network wrapper

As it was presented in deliverable D4.2 [7] Service Function Chaining or SFC is a vital part of the NFV concept. Since our last prototype SFC agent deployment there have been multiple improvements on many of its aspects. In addition to the various updates on code functionality, making the agents more robust and more customizable, a few new features have been introduced. The agent now is able to assign the network flow rules in the reverse order also. Previously, the SFC agent was only redirecting the incoming flow from the user to the server without handling the returning traffic. After these improvements, when the network traffic returns, in order to go back to the user, there are network flow rules deployed, that guide the traffic through the desired VNFs. Essentially, the traffic follows the same network graph as provided before, but in this case the SFC agent has reversed the traffic source and traffic destination options in the flow rules, so that it is able to capture traffic outgoing from the server back to the user. Besides this, we have introduced a new feature that now rewrites a MAC destination every time the traffic is directed inside a VNF with the VNF's ingress port MAC, keeping the same destination IP. Most of the VNFs involved need this modification or else they can't perform the functions required from them.

5.2 WIM Adaptor

The WIM adaptor has been substantially extended from SONATA release 2.0. The data model used to represent WIM was refined to include multiple NFVI-PoP and multiple WAN area (i.e. multiple WIMs). In deliverable D4.2 we described the data model used to represent a NFVI-PoP. The following image Figure 5.2 shows an abstract view of the overall NFVI model.

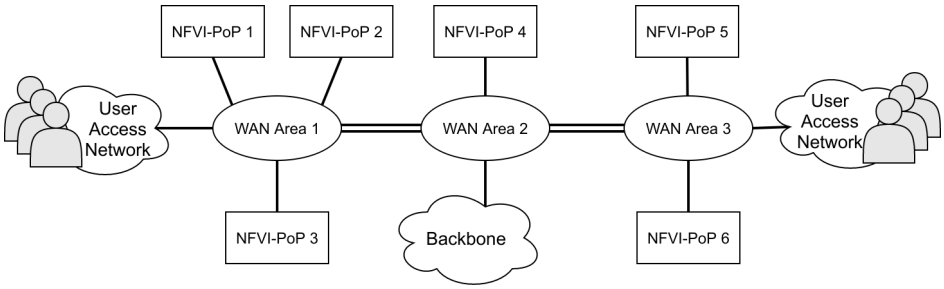

Figure 5.2: An abstract view of the overall NFVI model used by the Infrastructure Abstraction, including the multi-WIM/multi-VIM abstraction

The second major improvement of the WIM-Adaptor is the introduction of the WAN configuration for network services which are deployed across multiple NFVI-PoP, making use of the ingress and egress NAP specified by the user during the service instantiation (See Section Section 2.5 for more details). A pair of ingress/egress points, specified as an IP address in CIDR notation, is used by the WIM-Adaptor to identify the flows that need to be steered through the subset of the available NFVI-PoP where relevant VNFs are deployed. In order to provide this service to the overlaying MANO framework, the WIM-Adaptor offers an an interface through which the MANO can specify a pair of ingress and egress NAP, as well as an ordered list of the public endpoints(in the form of IP addresses) of the NFVI-PoPs (i.e. VIMs) where the VNFs of the relevant service are deployed. The interface is documented in the following Section 5.2.1, and some implementation details are given in Section 5.2.2.

5.2.1 WIM Adaptor API reference

The WIM Adaptor API is listed below.

Add a WIM

Register a WIM to the WIM-Adaptor.

topic: `infrastructure.management.wan.add`

data: {`wim_vendor: String, wim_address: String, username: String, pass: String`}, allowed values for *wim_vendor* is "VTN" and "MOCK", when one wants a deployment without WAN configuration.

return: {`request_status: String, uuid: String, message: String`}, when *request_status* is "COMPLETED", *uuid* fields carries the UUID of the registered WIM and *message* field is null, when *request_status* is "ERROR", *message* field carries a string with the error message, and the *uuid* field is empty.

List WIMs

Retrieve a list of the WIMs registered to the WIM-Adaptor.

topic: `infrastructure.management.wan.list`

data: `null`
return: `[{uuid: String, name: String, attached_vims: [Strings]}]`

Link VIM to WIM

Store in the WIM repository the connection information of a NFVI-PoP (i.e. Compute VIM) to a specific WIM.

topic: `infrastructure.management.wan.attach`
data: {`wim_uuid: String, vim_uuid: String`}
return: return: {`request_status: String, message: String`}, when *request_status* is "COMPLETED", *message* field is empty, when *request_status* is "ERROR", *message* field carries a string with the error message.

Configure WAN for service instance

Configure the WAN to allow traffic flow defined by the pair (*ingress_nap*,*egress_nap*) to be steered through the NFVI-PoP where VNFs of the relevant service instance are deployed.

topic: `infrastructure.management.wan.configure`
data: {`wim_uuid: String, ingress_nap: String, egress_nap: String, vims: [String]`}
return: {`request_status: String, message: String`}, when *request_status* is "COMPLETED", *message* field is empty, when *request_status* is "ERROR", message field carries a string with the error message.

5.2.2 VTN Wrapper V3

In order to control the WAN, we are using VTN Manager, a component of OpenDaylight, that allows us to manage the network using SDN technologies. As it was already mentioned in past deliverable D4.2 [7], with the VTN we are able to set traffic flow rules, controlling the network traffic, allowing, prohibiting or redirecting the packets that match the conditions applied. Since the last release a new version of the VTN WIM Wrapper has been implemented with extended functionalities. A RESTful API has now been developed, as well as a SQLite DB, located both in the same host PC that OpenDaylight+VTN Manager are installed. These two parts, the API and the WIM_info compose the new SONATA custom WIM, catered to the projects needs. The RESTful API in practice, is integrated upon the already existent VTN REST API and acts as a wrapper for the IA component, combining multiple VTN functions in a single call. The WIM_info database provides information about the hardware configuration of the infrastructure owner. The owner needs an SDN switch that he will have configured and registered to be controlled by ODL+VTN. This information can only be provided by the owner and as a result when the SONATA custom WIM is installed, the owner also has to populate the database with the required data. This data includes a particular segment of their deployment, the port id and virtual bridge name of where this segment is attached to the SDN switch, as well as an approximate physical location.

Below we list the REST API calls available and their functions:

Add Service Configuration

Add a network traffic flow rule to our VTN-controlled WAN. The call is send along with information on the various pair-redirections we want our network traffic to have. The ingress and egress segments are defined, as well as the PoPs that we need the traffic to be redirected into. The WIM uses the database to find out to which physical ports of our SDN switch these segments and PoPs are allocated. Afterwards, the WIM sets, through the VTN, the redirection rules from the source to the PoPs and finally to the final destination. Finally the service or "flow", is saved locally.

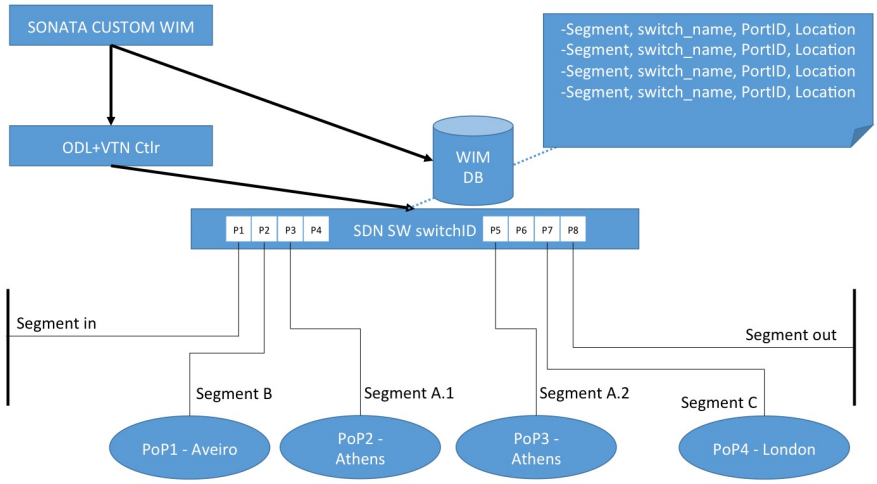

Figure 5.3: SONATA multi WIM approach

uri: `5000/flowchart/`
data: { `"instance_id": String, "in_seg": String, "out_seg": String, "ports": [ { "port": String, "order": Integer } ]` }, *int_seg*, *out_seg*, as well as the *port* field identifying the NFVI-PoP, must be expressed in CIDR form (e.g. xxx.yyy.zzz.hhh/pp)
return: 200

List Flows

Calls for a list of all the flows or services that have been set so far.
uri: `5000/flowchart/`
data: `null`
return : { `"flows": []` }

List Specific Flow

List a specific existing flow
uri: `5000/flowchart/instance_id_name`
data: `null`
return: {`"data": {"in_seg": String, "instance_id": String , "out_seg": String, "ports": [{"order": Integer, "port": String}]}`}, *int_seg*, *out_seg*, as well as the *port* field identifying the NFVI-PoP, are expressed in CIDR form (e.g. xxx.yyy.zzz.hhh/pp)

Delete Specific Flow

Delete a specific flow, if it exists
uri: `5000/flowchart/instance_id_name`
data : `null`
return : 200

List Location

Lists the locations available in the Database
uri: `5000/location/`

data : `null`
return : { `"locations": []` }

6 Monitoring

With respect to the SONATA Monitoring Framework, during the reporting period, the following enhancements have been achieved.

6.1 Distributed monitoring

From the beginning of the SONATA project, the target of the Service Platform, including the monitoring system, was to develop a solution that would be able to cope with carrier-grade requirements in terms of deployment, reconfiguration, migration, monitoring, etc., in a multi-PoP environment. In the reporting period, monitoring framework solution has been refactored (according to the plan provided from Deliverable 2.2 in the beginning of the project) based on the concept that a developer/user must be able to design/develop/deploy a network service that consists of VNFs that must be possible to be deployed in different SONATA PoPs.

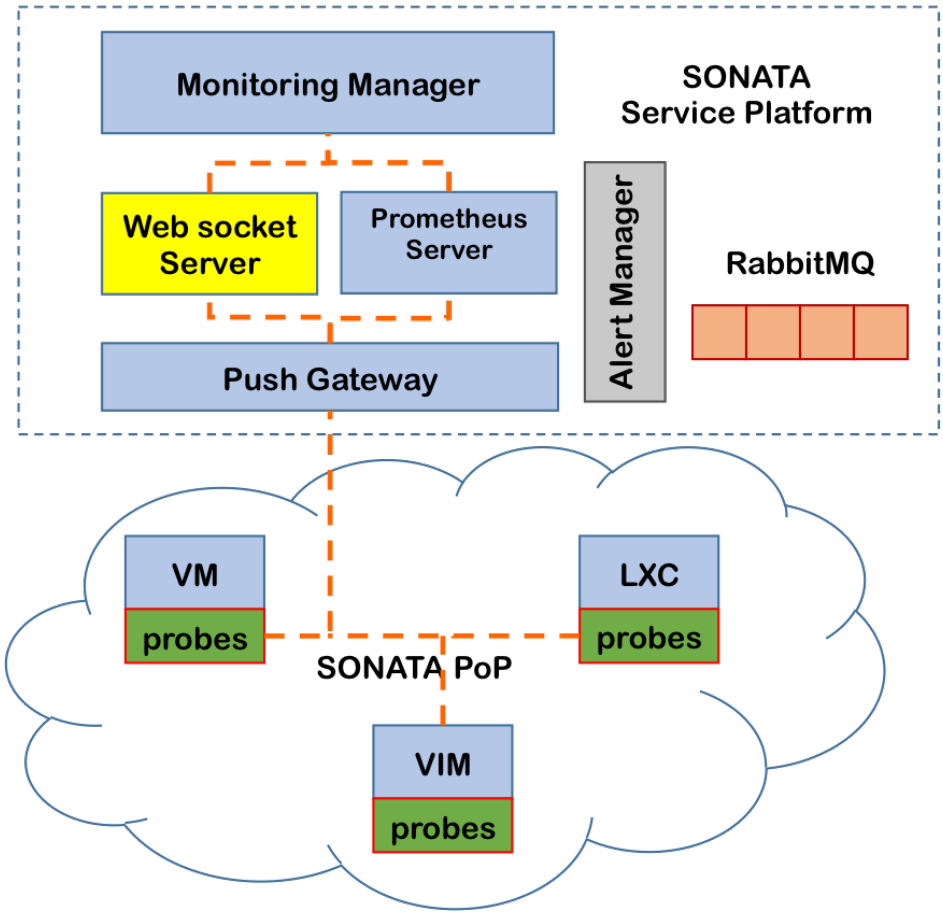

Figure 6.1: SONATA Centralized Monitoring Y1

In Figure 6.1 the centralized Monitoring approach (implemented during the first year of the

project) is depicted, while Figure 6.2 gives an overview of the distributed architecture implemented in this final software release.

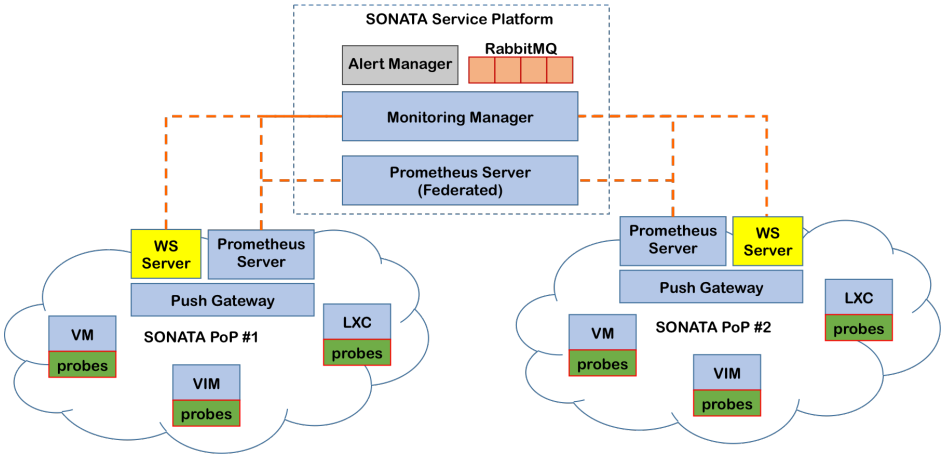

Figure 6.2: SONATA Distributed Monitoring Y2

As it is noticed, several components of the Monitoring Framework have to be distributed across the SONATA PoPs. First, each PoP must have its own websocket server to accommodate developers demands for streaming data, although as it will be described in the next section, the management of websocket is handled by the Monitoring Manager instance in a centralized way. So, the developer uses the appropriate APIs provided by the federated Monitoring Manager to get the list of metrics of a particular VNF. Then, the developer asks from the Monitoring Manager to create a new websocket deployed in the PoP where the VNF is instantiated.

Second, Prometheus servers follow a distributed (cascaded) architecture for scalability purposes. The local Prometheus servers collect and store metric data from the VNFs deployed in the PoP, while only the alerts are sent to the federated Prometheus server for further processing (Alert Manager) and forwarding to the RabbitMQ to be published to the subscribed users. In this distributed architecture, the alerting rules and notifications must be based on monitoring data collected in different PoPs and thus the decision must be made on a federation level. It is highlighted that each POP, apart from one Prometheus server instance, also includes an InFluxDB in order to store the monitoring data, while the Monitoring Framework also gives the ability to the SP and POP administrators to set thresholds to monitoring metrics beyond which the data are also forwarded to the federated Prometheus server, providing high flexibility.

The deployment of Prometheus servers in this cascaded fashion allows the monitoring system to scale to environments with tens of data centres and millions of nodes.

For the realization and support of the aforementioned functionality, a number of APIs has been developed and implemented:

- `GET /api/v1/prometheus/pop/{popID}/metrics/list`
- `GET /api/v1/prometheus/pop/{popID}/metrics/name/{metricName}/`
- `POST /api/v1/prometheus/pop/{popID}/metrics/data`
- `GET /api/v1/prometheus/pop/{popID}/configuration`
- `POST /api/v1/ws/pop/{popID}/new`

6.2 Dynamic and real-time reconfiguration of Prometheus and monitoring rules

Two important features that have been scheduled for the final version of the monitoring framework are:

- the ability to reconfigure Prometheus `yaml` configuration file, allowing for the addition of new PoPs, modifications of IP addresses of components in local or remote PoPs, etc,
  - `GET /api/v1/prometheus/configuration`
  - `GET /api/v1/prometheus/pop/{popID}/configuration`
- the ability to provide reconfiguration of monitoring rules per service id in a dynamic and real-time way. This has been implemented and can be utilized by the following API method:
  - `POST /api/v1/alerts/rules/service/{srvID}/configuration`

6.3 Streaming monitoring data via websockets

The implementation of the websocket extension has been finalized during the reporting period, providing a valuable tool to the SONATA developers. Figure 6.3 provides an overview of the integration of websocket in the SONATA Monitoring Framework.

The developer can communicate with the Monitoring Manager to get the list of his deployed service instances (`GET /api/v1/services/user/{usrID}/`) and the list of monitored metrics (`GET /api/v1/prometheus/metrics/list`).

Then, the developer can ask for the creation of a new websocket returning values from the specified metric (`POST /api/v1/ws/new{"metric":"mem","filters":["id='xxx'","type='vnf'"]}`).

Finally, the developer can connect to the websocket server, collecting the requested metric values (`ws://<ws_server_url>/ws/7f8bb6ecedad4f7090591f66d1f1cce0`).

6.4 Enhancement of monitoring sources

During the implementation of the SONATA use cases, and especially the vCDN use case, there was a need for collecting monitoring data from the virtual Traffic Classifier (vTC) time-series database (InfluxDB), store them in Prometheus database, process them in relation with other data and trigger alerts on the RabbitMQ.

The integration of the vTC data to the Monitoring framework was straightforward and showed the simplicity of integrating data from 3rd party sources to the SONATA Monitoring Framework, as shown in the figure above.

6.5 Other API extensions

There are few other APIs that are supported in the final release of the monitoring framework code related to the collection of data related to a specific user id, the management of PoPs and Service Platforms, as shown below:

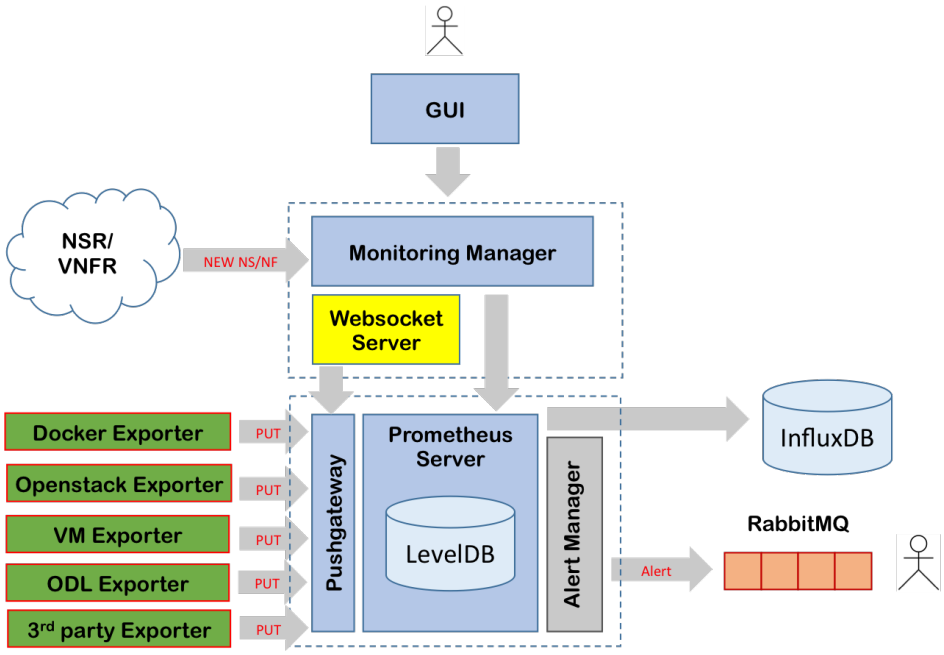

Figure 6.3: SONATA websocket architecture

6.5.1 Add/Retrieve/Delete POPs

The following PoP management API was made available:

- `GET /api/v1/pop`
- `POST /api/v1/pop`
- `GET /api/v1/pop/splatform/{spID}/`
- `DELETE /api/v1/pop/{sonata_pop_id}/`

6.5.2 Add/Retrieve/Delete Service platforms

The following Service platforms management API was made available:

- `GET /api/v1/splatform`
- `POST /api/v1/splatform`
- `DELETE /api/v1/splatform/{pk}/`

6.5.3 Retrieve services per User id

The following services per User id API was made available:

- `GET /api/v1/services/user/{usrID}/`

7 Platform configuration

The SONATA Service Platform has grown into a rather complete system and, although its architecture can be split into very fine grained components, or **micro-services**, each one being rather simple to understand, the whole got rather complex, specially for newcomers. The combination of every possible configuration of each one of the different modules/micro-services is a hard problem to solve generically, but we thought of having this section to explain what has been our strategy in terms of Service Platform Configuration.

We have grouped this into the following distinct sub-sections:

- **Default values:** to make the life of the developer that first downloads and tries to instantiate and experiment with our platform, there's a list of values that are there and make the whole coherent;
- **Feature toggles:** to make the platform flexible, we can enable/disable (toggle) a set of features (or even entire modules or even infrastructure blocks);

7.1 Default values

The first level of configuration is each module's **default values**. Ideally, no default value should be set within the code (hard-coded, see [14]), except those that can be super-seeded by values defined through alternative ways (e.g., **environment variables**).

The following sub-sections explain how default values are set for each one of the different modules and how they can be changed an played together.

7.1.1 Gatekeeper

Most of the Gatekeeper's modules are written in ruby, which has the convention of configuring by using `YAML` files in a `<root>/config` folder. Take, for instance, the configuration of the `son-gtkvim`, the micro-service that is responsible for managing the VIMs and the WIM (see `https://github.com/sonata-nfv/son-gkeeper/blob/master/son-gtkvim/config/services.yml.erb`):

```
development: &common_settings
  mqserver_url: <%= ENV['MQSERVER'] %>
  logger_level: <%= ENV['LOGGER_LEVEL'] %>

test:
  <<: *common_settings

integration:
  <<: *common_settings

qualification:
  <<: *common_settings
```

```
demonstration:
  <<: *common_settings
```

A configuration block, named `common_settings`, is defined for the `development` environment (lines `1-3`), and is then reused on each one of the different environments (`test`, `integration`, `qualification` and `demonstration`). These environments are defined by an environment variable called `RACK_ENV`. These values can be overloaded in each environment, for example like in

```
    ...
demonstration:
  <<: *common_settings
  logger_level: error
    ...
```

Expressions like ” are `ruby` code that get interpreted by a pre-processor -- `ERB` [11], in this case (therefore the `.erb` file extension of the file).

This is then used within the code like it is shown below.

```
    ...
    set :mqserver_vims_compute_list, MQServer.new(
      COMPUTE_LIST_QUEUE,settings.mqserver_url, logger)
    ...
```

So, the value defined in the `services.yml.erb` file above for the `mqserver_url` reaches the code and gets used.

Since this somehow overlaps with the more operational configuration done by using Ansible, we’re passing this kind of information to that tool, as can be seen in the following file:

```
---

# Populate pgSQL database VIM
- name: create data model and populate db
  shell: /usr/bin/docker run --net={{ docker_network_name }} -i \
    -e DATABASE_HOST=son-postgres -e MQSERVER=amqp://guest:guest@son-broker:5672 \
    -e RACK_ENV=integration -e DATABASE_PORT=5432 -e POSTGRES_PASSWORD=sonata \
    -e POSTGRES_USER=sonatatest --rm=true sonatanfv/son-gtkvim:{{ sp_ver }} \
    bundle exec rake db:migrate

# Start GTK VIM (Docker container)
- name: GATEKEEPER VIM
  docker_container:
    name: son-gtkvim
    image: "sonatanfv/son-gtkvim:{{ sp_ver }}"
    env:
      RACK_ENV: integration
      MQSERVER: amqp://guest:guest@son-broker:5672
      DATABASE_HOST: son-postgres
      DATABASE_PORT: 5432
```

```
    POSTGRES_PASSWORD: sonata
    POSTGRES_USER: sonatatest
  network_mode: bridge
  networks:
    - name: "{{ docker_network_name }}"
      aliases:
        - son-gtkvim
  state: started
  restart_policy: "unless-stopped"
  published_ports: "5700:5700"
```

Further improvements will be done using Ansible's `vault` mechanism to store sensitive information, such as `POSTGRES_PASSWORD`, above.

7.1.2 MANO Framework

Most MANO framework plugins are written in Python and are deployed as Docker containers. They all communicate through a central RabbitMQ message broker. Each of the MANO framework plugins needs can be configured through environment variables. The most important one is available to all plugins and used to configure the message broker to be used for communication:

```
ENV broker_host amqp://guest:guest@broker:5672/%2F
ENV broker_exchange son-kernel
```

The first variable `broker_host` specifies the URL to the broker which contains its host name, port and user credentials. The ones given in the example will work for the default installation of RabbitMQ. Further, a second variable named `broker_exchange` is used to define on which exchange of the broker the MANO plugins communicate. This can be any value but must be the same for all plugins that should talk with each other.

7.1.2.1 Plugin Manager

The plugin manager does not require special configurations except of information about the Mongo database to be used to store the information about the registered plugins:

```
ENV mongo_host mongo
ENV mongo_port 27017
ENV mongo_user <empty>
ENV mongo_pass <empty>
```

Further, does the plugin manager provide a RESTful management interface which is exposed to port 8001 as default. This can be changed by alternating the container's port mapping configuration:

```
-p 8001:8001
```

7.1.2.2 Specific Manager Registry

In the case of using Docker network for communication among plugins, the network_id needs to be configured by the name of Docker network like the following:

```
ENV network_id sonata-plugins
```

Otherwise, network_name should be configured as the following:

```
ENV network_name broker,broker
```

DOCKER_HOST is also needed to be configured like what you see in the following box:

```
ENV DOCKER_HOST unix://var/run/docker.sock
```

7.1.2.3 SLM

The SLM requires three specially defined parameters, to inform where it can find the REST API of the repositories and the Monitoring Manager

```
ENV url_nsr_repository
ENV url_vnfr_repository
ENV url_monitoring_server
```

7.1.2.4 FLM

The FLM requires two specially defined parameters, to inform where it can find the REST API of the function repository and the Monitoring Manager

```
ENV url_vnfr_repository
ENV url_monitoring_server
```

7.1.2.5 Placement and Scaling Executives

Besides the general parameters for MANO plugins, the placement and scaling executives do not require any additional parameters to be set.

7.1.2.6 Placement Plugin

Besides the general parameters for MANO plugins, the placement plugin does not require any additional parameters to be set.

7.1.3 Infrastructure Abstraction

This sub-section describes how the **Infrastructure Abstraction** can be configured.

7.1.3.1 Basic configuration

Infrastructure Abstraction is developed in Java and packaged as two Docker containers, one for the **VIM-Adaptor** and one for the **WIM-adaptor**. These two container use the SP's **PostgreSQL** database to store infrastructure related information and the SP's **RabbitMQ** message bus is used to exchange messages with the other SP modules, therefore basic configurations are limited to the connectivity parameters to these two entities. What follows is valid for both the VIM-Adaptor and the WIM-Adaptor container. These configuration parameters are stored in relevant configuration files that are passed to the container in the container building phase.

```
{
    "repo_host": "REPOHOST",
    "repo_port": "REPOPORT",
    "user": "REPOUSER",
    "pass": "REPOPASS",
}

{
    "broker_url": "BROKERURL",
    "exchange": "BROKEREXCHANGE"
}
```

Anyway, those values are more be easily set through the docker environment variables stated below, together with their default values:

```
ENV broker_host broker
ENV broker_port 5672
ENV broker_exchange son-kernel
ENV broker_uri amqp://guest:guest@broker:5672/%2F

ENV repo_host postgres
ENV repo_port 5432
ENV repo_user sonatatest
ENV repo_pass sonata
```

These values are set inside the 'dockerised' configuration files through the following script, also packaged in the container.

```
#!/bin/bash

sed "s#BROKERURL#$broker_uri#" -i /etc/son-mano/broker.config
sed -i "s/BROKEREXCHANGE/$broker_exchange/" /etc/son-mano/broker.config

sed -i "s/REPOHOST/$repo_host/" /etc/son-mano/postgres.config
sed -i "s/REPOPORT/$repo_port/" /etc/son-mano/postgres.config
sed -i "s/REPOUSER/$repo_user/" /etc/son-mano/postgres.config
sed -i "s/REPOPASS/$repo_pass/" /etc/son-mano/postgres.config
```

7.1.3.2 VIMs and WIMs configuration

Once the two containers of the abstraction layer are running and configured, there is still no information stored in the Infrastructure repository on the VIMs/WIMs to be used as NFVI. Since this information is more structured and can change at runtime, the IA provides a set of API through which those can be configured. Anyway, the only endpoint that the IA offers is through the message bus, therefore this configuration is facilitated for the users through the Gatekeeper module Gtk_vim and the Sonata GUI, which provide a more user-friendly way of configuring NFVI. The following Figure 7.1 exemplify the minimum configuration steps required to configure a simple NFVI composed by one NFVI PoP and one WIM.

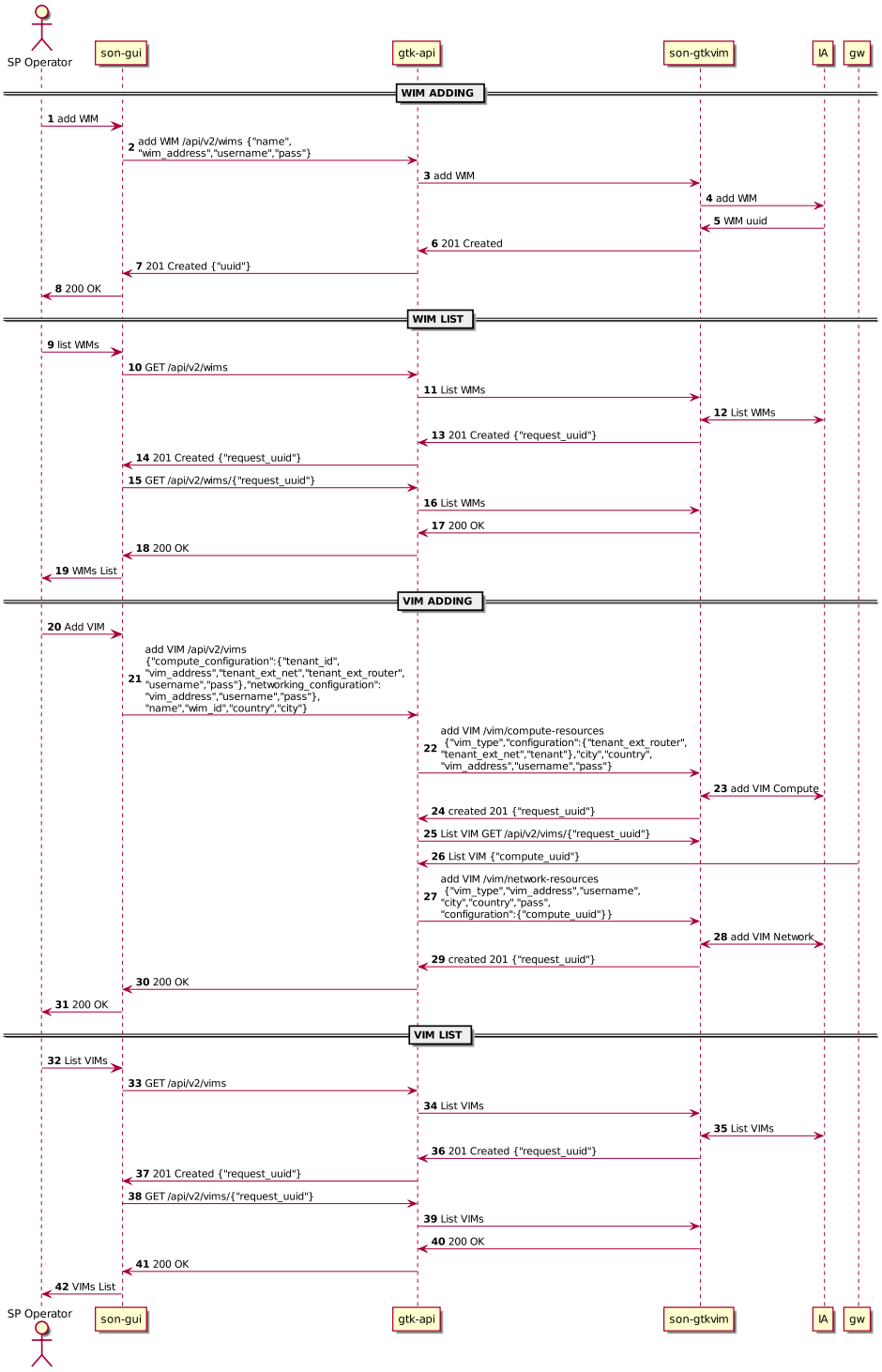

Figure 7.1: IA configuration flow

7.2 Toggling features, modules and infrastructure resources

Software systems naturally grow and evolve, so there has to be a way to switch on or off (to toggle, see [12]) some feature that, for instance, is ready on one module but not yet ready on another one on which it depends on, or shouldn't be available in a certain customer or regions, etc. This can also happen to entire modules or pieces of the supporting (external) infrastructure that is supposed to be there.

This degree of configurability is hard to implement and test, due to the potentially combinatory explosion of every possible combination of features, modules and infrastructure. Therefore, the SONATA SP has addressed the issue in layers

7.2.1 Feature toggles

Because the implementation of the BSS module (see Section 2.5) had a much greater speed than the implementation of the Gatekeeper's API, we have chosen to implement **feature toggling** in it. This was done using is a **grunt** server ([1]) that has three feature toggles implemented like grunt command line options:

- **"protocol"**:
  - possible values: http/https.
  - default value: http.
  - description: with this option, the server is deployed using http or https protocols.
  - code:

```
if ( !grunt.option( 'protocol' ) ) {
    grunt.option( 'protocol', 'http' );
}

if (grunt.option('protocol')=='https') {
   return {
    protocol: 'https',
    key: grunt.file.read('app/certs/sonata.key').toString(),
    cert: grunt.file.read('app/certs/sonata.crt').toString(),
    hostname: grunt.option('hostname'),
    port: 1337,
    base: 'app'
  };
} else {
   return {
    protocol: 'http',
    hostname: grunt.option('hostname'),
    port: 1337,
    base: 'app'
   };
}
```

- **"userManagementEnabled"**:
  - possible values: true/false.

  - default value: true.
  - description: with this option disabled, the Gatekeeper's User Management module is not used to log in. A fake token is sent to the Gatekeeper for authorization and authentication.
  - code:

```
if (ENV.userManagementEnabled == 'false') {
    var fakeToken = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.'
                    +'eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZS...VlfQ.'
                    +'AdgPchW4kBolbrVPn8YlrNIOx8XqcHcO_bCR2gclGyo';
    $localStorage.currentUser = { username: 'sonata',
                                  token: fakeToken ,
                                  user_id:'1234',
                                  user_role:'customer'};
    $http.defaults.headers.common.Authorization = fakeToken;
    defer.resolve(true);
} else {
    $http.post(ENV.apiEndpoint+'/sessions', data)
    .then(function successCallback(response){
    ...
```

- **"licenseManagementEnabled"**:
  - possible values: true/false.
  - default value: true.
  - description: this toggle enables or disables the use of the license management. this toggle enables or disables the use of the license management in the BSS. When is disabled, all requests are allowed without checking any license constraint.
  - code:

```
 if (ENV.licenseManagementEnabled == 'false') {
     defer.resolve(true);
 } else {
     $http.get(ENV.apiEndpoint+"/licenses/user/"+user_id)
     .then(function successCallback(result){
        defer.resolve(result)})
     .catch(function errorCallback(error){
        defer.reject(error)});
}
```

7.2.2 Module toggles

While talking about toggling different modules, we have to distinguish two cases, modules that can be :

1. **absent:** without these, some features are lost, but the Service Platform still does what is expected from it;

2. **different** (from the ones provided by default by the SP)**:** within the environment where the SP is to be instantiated, there can be already modules fulfilling the tasks that we have thought for these modules, that need to be integrated.

We address these two distinct cases next.

7.2.2.1 Modules that can be absent

In its current version (v3), we consider the following list of modules that can be absent:

- **User Management**:
  - the SP doesn't know it's users, whether developers or customers;
  - even when marked as **private**, packages, services and functions can be reused and instantiated at will
  - no User Management implies no Licence Management, due the inability to authenticate the user;
  - Impacts:
    - the Gatekeeper API shouldn't validate the **Authorization** header
    - the GUI should not allow for user registration, login or logout;
    - the BSS should be deployed with the flag **userManagementEnabled** to false.
- **Licence Management**:
  - even when marked as **private**, packages, services and functions can be reused and instantiated at will
- **KPIs Management**:
  - no KPIs are collected or shown;

The **impact** of the absence of a given module has to be dealt with...

7.2.2.2 Modules that can be different

- **SP Catalogue**: this module performs the micro-service example of authentication/authorization process for Service Accounts. This process can be enabled/disabled (it is currently disabled (hardcoded)) whether User Management is running or not.
- **SP Repositories**: This module is logically separated from the SP Catalogue but it resides with it. The same case as SP Catalogue can be applied to this module.
- must obey internal APIs, published in [7]
- adaptation effort is expected, since there are no standards

7.2.3 Infrastructure toggles

The absence of a given part of the infrastructure heavily impacts on the workflows of the SONATA environment, since service deployment assumes the presence of a NFVI, composed by a minimal set of resources. Nonetheless, this set can be reduced to the minimum of one NFVI-PoP without any WAN management. In this case the routing of service flows inside this single NFVI-PoP must be dealt with manually by the service operator. The SP GUI allows to configure this basic setting through its VIM configuration facility. More specifically, the WIM registration is the first and mandatory step of this configuration, as shown in Figure 7.1. The SP operator can use the GUI to define a mocked WIM wrapper, that he can use to build the most simple topology, by attaching the actual NFVI-PoP it wants to use. Moreover this configuration can also be passed to the IA module when triggering the container execution, by specifying the environment variable `NOWAN` with a `TRUE`.

7.3 Platform configuration summary

SONATA Service platform is built based on microservices oriented infrastructure. Each component is dynamic configured by environment variables injected to the container context in boot process. By this way, the configuration manager used in SONATA (Ansible) can deploy the Service Platform with specific environment configuration. The Table 7.1 intends to shows the Service Platform components and its relations.

Table 7.1: Platform Configuration Summary

Component	Environment Variable	Upstream	Downstream	Toggles
son-sec-gw			son-gui son-gtkapi son-monitor-manager	Automatic detect absence/presence of certificates and enable https
son-gui	MON_URL: "`https://sp.example.com/monitoring`" GK_URL: "`https://sp.example.com/api/v2`"	son-sec-gw	son-gtkapi	
son-bss	serve: "integration" gkApiUrl: "`http://sonata.example.com/api/v2`" hostname: "sonata.example.com" userManagementEnabled: "false" licenseManagementEnabled: "false" protocol: "http"		son-gtkapi	protocol user Management license Management
gtk-api	RACK_ENV: "integration" USER_MANAGEMENT_URL: "`http://son-gtkusr:5600`" LICENSE_MANAGEMENT_URL: "`http://son-gtklic:5900`" METRICS_URL: "`http://son-monitor-manager:8000/api/v1`" CATALOGUES_URL: "`http://son-catalogue-repos:4011/catalogues/api/v2`" PACKAGE_MANAGEMENT_URL: "`http://son-gtkpkg:5100`" SERVICE_MANAGEMENT_URL: "`http://son-gtksrv:5300`" FUNCTION_MANAGEMENT_URL: "`http://son-gtkfnct:5500`" VIM_MANAGEMENT_URL: "`http://son-gtkvim:5700`" RECORD_MANAGEMENT_URL: "`http://son-gtkrec:5800`" KPI_MANAGEMENT_URL: "`http://son-gtkkpi:5400`"	son-sec-gw	son-gtkusr son-gtklic son-gtkpkg son-gtksrv son-gtkfnct son-gtkvim son-gtkrec son-gtkkpi son-monitor-manager son-catalogue-repos	

Component	Environment Variable	Upstream	Downstream	Toggles
son-gtkusr	KEYCLOAK_ADDRESS: "son-keycloak" KEYCLOAK_PORT: "5601" KEYCLOAK_PATH: "auth" SONATA_REALM: "sonata" CLIENT_NAME: "adapter"	son-gtkapi	son-keycloak	
son-gtklic	DATABASE_HOST: "son-postgres" DATABASE_PORT: "5432" POSTGRES_PASSWORD: "sonata" POSTGRES_USER: "sonatatest" POSTGRES_DB: "gatekeeper"	son-gtkapi	son-postgres	
son-gtkpkg	RACK_ENV: integration CATALOGUES_URL: "`http://son-catalogue-repos:4011/catalogues/api/v2`"	son-gtkapi	son-catalogue-repos	
son-gtksrv	CATALOGUES_URL: "`http://son-catalogue-repos:4011/catalogues/api/v2`" RACK_ENV: "integration" MQSERVER: "amqp://guest:guest@son-broker:5672" DATABASE_HOST: "son-postgres" DATABASE_PORT: "5432" POSTGRES_PASSWORD: "sonata" POSTGRES_USER: "sonatatest"	son-gtkapi	son-catalogue-repos son-broker son-postgres service lifecycle management	
son-gtkfnct	CATALOGUES_URL: "`http://son-catalogue-repos:4011/catalogues/api/v2`" RACK_ENV: "integration"	son-gtkapi	son-catalogue-repos	
son-gtkvim	RACK_ENV: "integration" MQSERVER: "amqp://guest:guest@son-broker:5672" DATABASE_HOST: "son-postgres" DATABASE_PORT: "5432" POSTGRES_PASSWORD: "sonata" POSTGRES_USER: "sonatatest"	son-gtkapi	son-broker son-postgres	
son-gtkrec	REPOSITORIES_URL: "`http://son-catalogue-repos:4011/records`" RACK_ENV: "integration"	son-gtkapi	son-catalogue-repos	
son-gtkkpi	RACK_ENV: "integration" PUSHGATEWAY_HOST: "pushgateway" PUSHGATEWAY_PORT: "9091" PROMETHEUS_PORT: "9090"	son-gtkapi	Pushgateway	
son-keycloak	KEYCLOAK_USER: "admin" KEYCLOAK_PASSWORD: "admin"	son-keycloak	son-mongo	
son-catalogue-repos	PORT 4002	son-gtkapi son-gtkpkg son-gtksrv son-gtkfnct son-gtkrec service lifecycle management	son-mongo	
service lifecycle management	url_nsr_repository: "`http://son-catalogue-repos:4011/records/nsr/`" url_vnfr_repository: "`http://son-catalogue-repos:4011/records/vnfr/`" url_monitoring_server: "`http://son-monitor-manager:8000/api/v1/`" broker_host: "amqp://guest:guest@son-broker:5672/%2F"	son-gtksrv	son-catalogue-repos son-monitor-manager son-broker	
pluginmanager	mongo_host: "son-mongo" broker_host: "amqp://guest:guest@son-broker:5672/%2F"		son-broker son-mongo	
placement executive	broker_host: "amqp://guest:guest@son-broker:5672/%2F"		son-broker	
specific manager registry	broker_host: "amqp://guest:guest@son-broker:5672/%2F" broker_name: "son-broker,broker"		son-broker	

Component	Environment Variable	Upstream	Downstream	Toggles
son-sp-infrabstract	broker_host: “son-broker” broker_uri: “amqp://guest:guest@son-broker:5672/%2F” repo_host: “son-postgres” repo_port: “5432” repo_user: “sonatatest” repo_pass: “sonata”	service lifecycle management son-gtkvim	son-postgres son-broker	
wim-adaptor	broker_host: “son-broker” broker_uri: “amqp://guest:guest@son-broker:5672/%2F” repo_host: “son-postgres” repo_port: “5432” repo_user: “sonatatest” repo_pass: “sonata”	service lifecycle management son-gtkvim	son-broker son-postgres	
son-monitor-manager		service lifecycle management son-gtkapi son-sec-gw	son-monitor-postgres son-monitor-prometheus	
son-monitor-prometheus	RABBIT_URL: “son-broker:5672” EMAIL_PASS: “SHA1”	son-monitor-manager	son-broker son-monitor-pushgateway son-monitor-influxdb	
son-monitor-pushgateway son-monitor-influxdb		son-monitor-prometheus son-monitor-prometheus		
son-monitor-postgres	POSTGRES_DB: “monitoring” POSTGRES_USER: “monitoringuser” POSTGRES_PASSWORD: “sonata”	son-monitor-manager		
son-broker		son-gtkvim son-gtksrv pluginmanager service lifecycle management placement executive specific manager registry son-sp-infrabstract wim-adaptor son-monitor-prometheus		
son-postgres	POSTGRES_DB: “gatekeeper” POSTGRES_USER: “sonatatest” POSTGRES_PASSWORD: “sonata”	son-gtksrv son-gtklic son-gtkvim son-sp-infrabstract wim-adaptor		

8 Conclusions and future work

Version 3.0 of the SONATA Service Platform (SP) has been enhanced with a set of features, improvements and corrections that makes it worth to be one of the players to be considered in the vast NFV/SDN Service Platforms/Orchestrators eco-system.

The **Gatekeeper** now fully supports **User Management**, which enables the implementation of authentication and authorization mechanisms. This, together with a simple **License Management** flow, shows how a SP owner can monetize the usage of the API, by controlling which developers can download packages (e.g., for reusing services or functions) and which customers can subscribe services. Most of the endpoints of the API now also generate **KPIs**, allowing for the demonstration of the times each process takes. These KPIs will serve as the basis for the **SLA Management**, to be described in [9]. In this version, the **GUI** supports all the (developer) user related processes (user creation, login, etc.), as well as the KPIs visualization. The **BSS** also supports (customers) user management processes and the new flow related with service licensing. The **Micro-services Management** module has been implemented, with a proof-of-concept in one of the micro-services, **Catalogues** (see below).

In the **MANO Framework**, we have implemented a clear separation between the SLM and the FLM, thus more strictly following ETSI's proposed two-level design (service-level and function-level). We now have default **scaling** and **placement** plugins, which get called if no SSM/FSM with that feature is given. SSMs/FSMs (i.e., code provided by developers) run in a 'sandbox' in terms of topics they can subscribe/post to.

In the **Infrastructure Abstraction** (IA), we have now the possibility to configure multiple VIMs, connected through multiple WIMs, which is the realistic scenario for a carrier-grade solution that SONATA is willing to cover. When a service instantiation is requested (through the **BSS**), the customer provides additional data that is then used in the optimization of placement of the different function instances and the connections between them. The IA also provides in this version the possibility to use OpenStack's Mistral as a script-based alternative (to the code-based FLM) tool to scale functions. We have spent some effort designing and prototyping the abstraction of a container (docker)-based VIM, but we have stopped when it became obvious that the implementation of the demanding needs of the scenarios we were trying to implement (namely at the networking level, to use Service Function Chaining to connect the different VNF instances of a given service) were not possible.

In **Monitoring**, we have now supports a distributed architecture that scales with the different VIMs/PoPs we want the Service Platform to interact with. Only configured **alerts** are centralized, thus maintaining the view of a network service that is realized by several VNFs deployed in more than one PoPs. Such functionality is also needed in this kind of carrier grade solution. Monitoring data can be requested to the platform by a given function developer, which may help debugging or in general know more about the performance of that function, and thus contributing to future improvements. This data may not be accessed by unauthorized users, but only by the function 'owner' or some other developer that has a license to reuse it, and is limited in time for overall platform performance reasons.

Catalogues now support an 'intelligent delete' feature that denies deletes of still in use packages, services and functions. Being one of the key components in a realistic deployment scenario, Catalogues is most probably one of the micro-services that would have to be changed. We have therefore

chosen this micro-service to implement the concept of Micro-service Management, mentioned above. **Repositories** did not need any upgrade from the previous versions.

The whole SP **configuration** has also been enhanced in all modules, allowing the inclusion or exclusion, at deployment time, of some of the modules of the platform. This is crucial for making the experimentation easier, by having to deploy and configure fewer components.

8.1 Future work

Nevertheless, some features had to be left out of this version. The following list summarises them:

- when the current implementation of containers evolve towards supporting better networking, we would like to use the implemented (for sure with some adaptions) Infrastructure Abstraction on top of that **container-based VIM**;
- while the **Resolver** component itself has been already fully developed, it still has to be integrated in the SONATA Service Platform to allow an end-to-end workflow;
- **pilots**, which definition is still a work in progress, will have and impact on the current set of features of the Service Platform. Some of these features have already been identified, and some of them even implemented, but for sure some more will come;
- increasing the Service Platform's **flexibility** in terms of features provided, is also something this team would like to do. With the micro-services architecture we have implemented and their APIs documented, the key features are there to allow some developer to, e.g., just deploy the MANO Framework and attach another Infrastructure Adaptor, avoid the Gatekeeper all together, etc.
- we would like to increase the level of control we have on the **monitoring data provided to the developer**, who can, in the current implementation, maintain any number of Web Sockets for any length of time sending back monitoring data.

A Licence Management

This section describes some Licence Management concepts related with the need of managing the widest range of licensing terms, conditions and models associated with the millions of instances of network functions and services supplied through the Service Platform. For more info: [16].

A.1 Licence Procurement

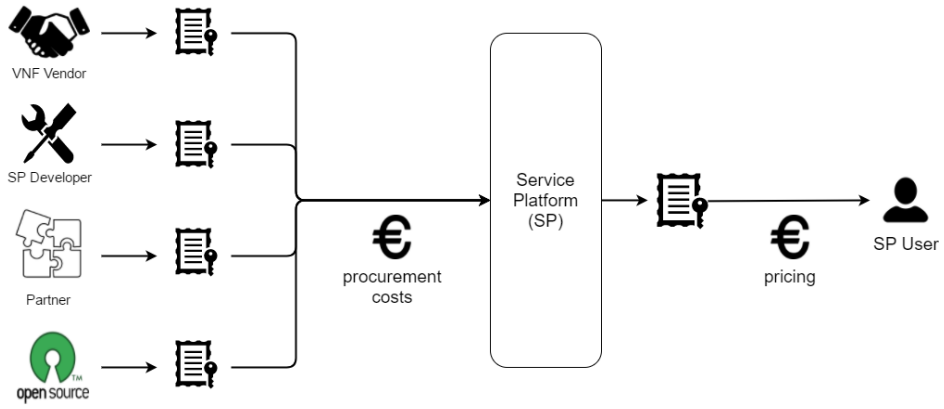

Figure A.1: Licence Procurement

The Licence Procurement process (Figure A.1) has two main parts:

- Licence addition in the Service Platform: the process where the licences are included in the platform, being available to the platform users.
  The platform's licences procurement costs might be different for
  - Licensing model for VNF vendors: the Service Platform (SP) purchases/rents VNF licences from VNF vendors.
  - Licensing model for co-development of VNFs: the SP co-develops VNFs with a partner.
  - Licensing model for utilising the Open Source VNFs
  - Licensing model for homemade VNFs: the SP develops VNF (homemade VNFs)
- Licence acquisition by platform user.
  The pricing of VNFs will vary depending on factors like:
  - location constraints
  - usage constraints
  - the licensing model:
    * Pay per use
    * Time-based usage

    * SLA (gold/silver/bronze)
    * Subscription based

A.2 Licence Management Requirements

- The Licence entity model should be business or technology agnostic.
- A single on-boarded VNF/Service can be associated with more than one licences agreement (licence type)
- The lifecycle stages of licences will be dependent on on-boarded VNFs/Services but not directly with the versions. While the version itself can change but agreed licence will continue.
- Licence entity should be linked to the instance, this information will be required to track the licence usage.
- The types of licences and related policies (like a restriction at various level of use) should be standardised. This will be very important for policy enforcement during orchestration.
- It is desirable to keep the supplier/vendor information linked with VNF/Service and not directly linked to VNF licences.

MANO will interface with the licence management functional block to query, reserve, release the licences associated with the instances. Licences information will be used across OSS/BSS and MANO, therefore, it is desirable to place the functional block in a centralised way.

Figure A.2 shows the Licence Management related architecture and the relationship between components.

A.3 Entity Model

Figure A.3 shows the first approach to the licence entity model.

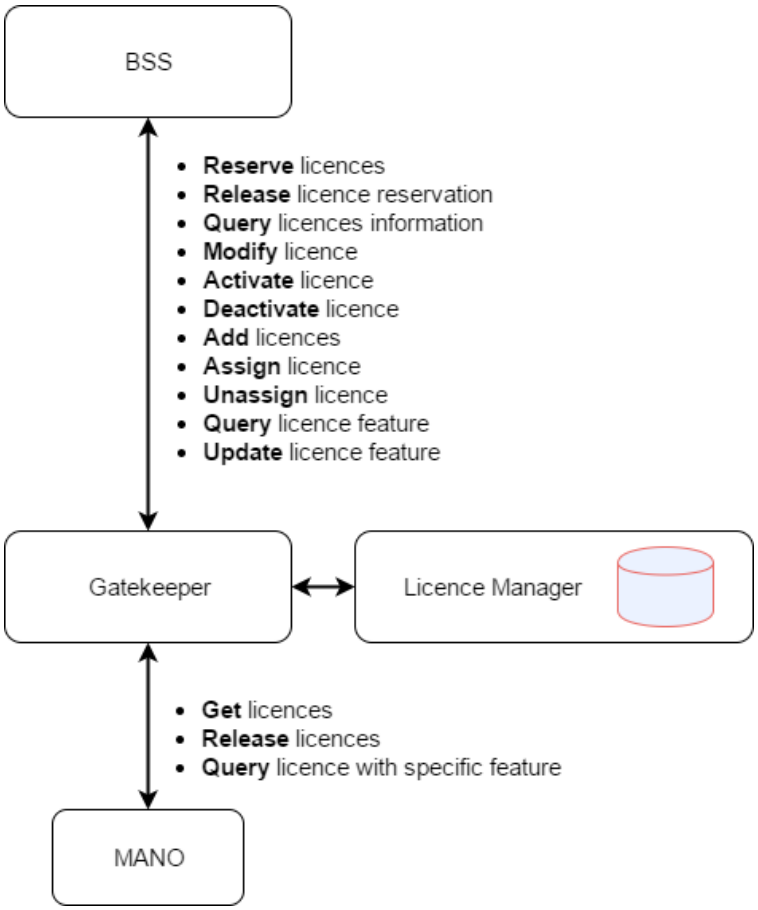

Figure A.2: Licence Management Architecture

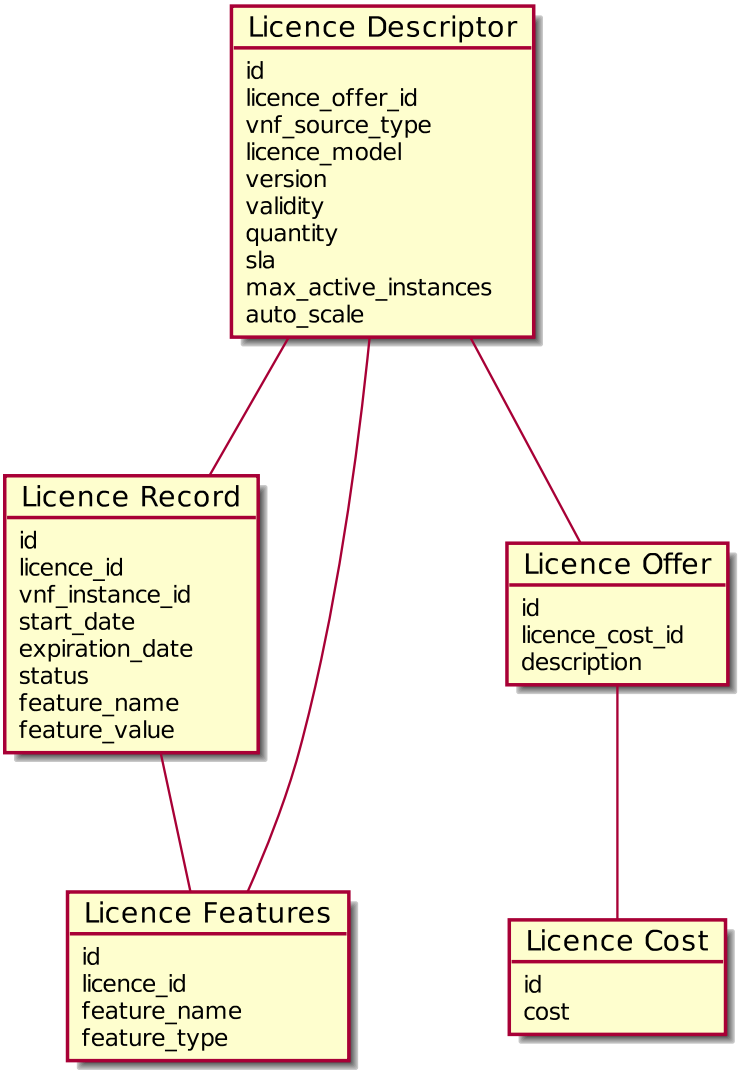

Figure A.3: Entity Model

B Abbreviations

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

CM Configuration Management

CRUD Create, Read, Update, Delete

DSL Domain-Specific Language

ETSI European Telecommunications Standards Institute

FLM Function Lifecycle Manager

FSM Function-Specific Manager

FSMD Function-Specific Manager Descriptor

GitHub Git repository hosting service

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

IA Infrastructure Adaptor

IaaS Infrastructure as a Service

IDE Integrated Development Environment

IoT Internet of Things

JMS Java Messaging System

JWT JSON Web Token

KPI Key Performance Indicator

MANO Management and Orchestration

NF Network Function

NFV Network Function Virtualization

NFVI-PoP Network Function Virtualisation Points of Presence

NFVO Network Function Virtualization Orchestrator

NFVRG Network Function Virtualization Research Group

NS Network Service

NSD Network Service Descriptor

NSO Network Service Orchestrator

OASIS Organization for the Advancement of Structured Information Standards

OIDC OpenID Connect

OSS Operations Support System

PD Package Descriptor

PSA Personal Security Applications

RBAC Role-based Access Control

REST Representational State Transfer

RPC Remote Procedure Call

SDK Software Development Kit

SDN Software-Defined Networking or Software-Defined Network

SLA Service Level Agreement

SLM Service Lifecycle Manager

SNMP Simple Network Management Protocol

SP Service Platform

SSM Service-Specific Manager

SSMD Service-Specific Manager Descriptor

UBAC User-based Access Control

VDU Virtual Deployment Unit

VIM Virtual Infrastructure Manager

VLD Virtual Link Descriptor

VM Virtual Machine

VN Virtual Network

VNF Virtual Network Function

VNFD Virtual Network Function Descriptor

VNFFGD VNF Forwarding Graph Descriptor

VNFM Virtual Network Function Manager

WAN Wide Area Network

WIM Wide area network Infrastructure Manager

C Glossary

DevOps A term popularized since a series of conferences emphasizing a higher degree of communication between **Dev**elopers and **Op**eration**s**, those who deploy the developed applications.

Function-Specific Manager A function-specific manager is a small management program implemented by a service developer with the help of SONATA's SDK. It is executed by the SONATA service platform to manage individual network functions based on inputs, say monitoring data, specific to the network function it belongs to.

Gatekeeper In general, gatekeeping is the process through which information is filtered for dissemination, whether for publication, broadcasting, the Internet, or some other mode of communication. In SONATA, the gatekeeper is the central point of authentication and authorization of users and (external) Services.

Management and Orchestration (MANO) In the ETSI NFV framework ETSI-NFV-MANO, MANO is the global entity responsible for management and orchestration of NFV lifecycle.

Message Broker A message broker, or message bus, is an intermediary program module that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. Message brokers are elements in telecommunication networks where software applications communicate by exchanging formally-defined messages. Message brokers are a building block of Message oriented middleware.

Network Function The atomic entity of execution anything in the context of a service. Cannot be further subdivided. Runs as a single executing entity, such as a single process and a single virtual machine. Treated as atomic from the point of view of the orchestration framework.

Network Function Virtualization (NFV) The principle of separating network functions from the hardware they run on by using virtual hardware abstraction.

Network Function Virtualization Infrastructure Point of Presence (NFVI PoP) Any combination of virtualized compute, storage and network resources.

Network Function Virtualization Infrastructure (NFVI) Collection of NFVI PoPs under one orchestrator.

Network Service A network service is a composition of network functions.

Network Service Descriptor A manifest file that describes a network service. Usually, it consists of the description of the network functions in the server, the links between the functions, a service graph, and service specifications, like SLAs.

Resource Orchestrator (RO) Entity responsible for domain wide global orchestration of network services and software resource reservations in terms of network functions over the physical or virtual resources the RO owns. The domain an RO oversees may consist of slices of other domains.

Service-Specific Manager (SSM) A service-specific manager is a small management program implemented by a service developer with the help of SONATA's SDK. It is executed by the SONATA service platform to manage individual services based on inputs, say monitoring data, specific to the service it belongs to.

Service Level Agreement (SLA) A service-level agreement is a part of a standardized service contract where a service is formally defined.

Service Platform One of the key contributions of SONATA. Realizes management functionality to deploy, provision, manage, scale, and place service on the infrastructure. a service developer/operator can use SONATA's SDK to deploy a service on a selected service platform.

Slice A provider-created subset of virtual networking and compute resources, created from physical or virtual resources available to the (slice) provider.

Software Development Kit (SDK) A set of tools and utilities which help developers to create, monitor, manage, optimize network services. A key component of the SONATA system.

Virtualised Infrastructure Manager (VIM) provides computing and networking capabilities and deploys virtual machines.

Virtual Network Function (VNF) One or more virtual machines running different software and processes on top of industry-standard high-volume servers, switches and storage, or cloud computing infrastructure, and capable of implementing network functions traditionally implemented via custom hardware appliances and middleboxes (e.g. router, NAT, firewall, load balancer, etc.).

Virtualized Network Function Forwarding Graph (VNF FG) An ordered list of VNFs creating a service chain.

D Bibliography

[1] Grunt Community. Grunt. Website, June 2016. Online at (http://gruntjs.com).

[2] Kubernetes Community. Kubernetes. Website, November 2016. Online at http://kubernetes.io/.

[3] OpenSource community. Java docker api library. Website, June 2017. Online at https://mvnrepository.com/artifact/com.github.docker-java/docker-java.

[4] The OpenStack Community. Openstack mistral project. Website, Dec 2016. Online at https://wiki.openstack.org/wiki/Mistral.

[5] The OpenStack Community. Fluent openstack sdk for java. Website, June 2017. Online at https://FluentOpenStackSDKforJava.

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

[7] SONATA consortium. D4.2: Service platform operational release and documentation. Website, December 2016. Online at http://sonata-nfv.eu/content/d42-service-platform-first-operational-release-and-documentation.

[8] SONATA consortium. D3.3 sdk operational release and documentation. Website, June 2017. Online at http://www.sonata-nfv.eu/.

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

[10] Ddos - distributed denial of service attack. Online at http://www.digitalattackmap.com/understanding-ddos/.

[11] Embedded ruby. Website. Online at https://ruby-doc.org/stdlib-2.4.1/libdoc/erb/rdoc/ERB.html.

[12] Pete Hodgson. Feature toggles. February 2016. Online at https://martinfowler.com/articles/feature-toggles.html.

[13] Leaky bucket algorithm. Website. Online at https://en.wikipedia.org/wiki/Leaky_bucket.

[14] Robert Martin. *Clean Code*. 2009.

[15] mozilla. Rabbitmq management http api. Website. Online at https://pulse.mozilla.org/api/.

[16] Abinash Vishwakarma and Alicja Kawecki. Ig1143 - license management v.1.0.2, March 2017. Online at https://www.tmforum.org/resources/standard/ig1143-license-management-r16-5-1/.