D5.2: Intelligent Management and Orchestration

Deliverable

Document ID	NECOS-D5.2
Status	Final
Version	0.31
Editors(s)	Fabio Luciano Verdi (UFSCar)
Due	31/10/2019
Submitted	30/10/2019

Abstract

This document is presenting the final design and implementation specifications of NECOS system and components. The previous deliverable, D5.1, presented an initial explanation about how the architecture was designed and implemented without significant number of artefacts and details. Also, the D5.1 included only the main components of the architecture components implemented at the time of deliverable (M12). This follow up deliverable D5.2 presents all the components implemented, it also provides additional relevant design artefacts and details. This document is revisiting the components described in D5.1, namely, Slice Builder, Slice Broker, Slice Agent and the DC/WAN Slice Controllers giving all the updates on their internal operation. The remaining components of the architecture, namely, the Slice Specification Processor, Service Orchestrator Adaptor, Slice Resource Orchestrator (SRO) and Infrastructure & Monitoring Abstraction (IMA), are introduced and detailed in this document. Finally, this document provides updates on some workflows presented in D5.1 due to the technical insights gained after the implementation of the additional components.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF CONTRIBUTORS

Contributor	Institution
Fábio Luciano Verdi	Federal University of São Carlos (UFSCar)
André Beltrami	Federal University of São Carlos (UFSCar)
Sand Luz Corrêa	Federal University of Goiás (UFG)
Leandro Alexandre Freitas	Federal University of Goiás (UFG)
Javier Baliosian	Universitat Politècnica de Catalunya (UPC)
Francesco Tusa	University College London (UCL)
Ilias Sakellariou	University of Macedonia (UOM)
Chronis Valsamas	University of Macedonia (UOM)
Sotiris Skaperas	University of Macedonia (UOM)
George Violettas	University of Macedonia (UOM)
Tryfon Theodorou	University of Macedonia (UOM)
Lucian Beraldo	Federal University of São Carlos (UFSCar)
Silvio Sampaio	Federal University of Rio Grande do Norte (UFRN)
Rafael Pasquini	Federal University of Uberlândia (UFU)
Joan Serrat	Universitat Politècnica de Catalunya (UPC)
Augusto Neto	Federal University of Rio Grande do Norte (UFRN)
Felipe Dantas Silva	Federal University of Rio Grande do Norte (UFRN)
Marcilio Lemos	Federal University of Rio Grande do Norte (UFRN)
Celso Cesila	University of Campinas (UNICAMP)
Billy Pinheiro	Federal University of Pará (UFPA)
Raquel Fialho Lafetá	Federal University of Uberlândia (UFU)

REVIEWERS

Reviewer	Institution
Alex Galis	University College London (UCL)
Rafael Pasquini	Federal University of Uberlândia (UFU)
Lefteris Mamatas	University of Macedonia (UOM)

Acronyms

Acronym	Description
NECOS	Novel Enablers for Cloud Slicing
IMA	Infrastructure & Monitoring Abstraction
SRO	Slice Resource Orchestrator
LSDC	Lightweight Software Defined Cloud
VIM	Virtual Infrastructure Manager
WIM	Wide-area network Infrastructure Manager
KPI	Key Performance Indicator
VM	Virtual Machine
PDT	Partially Defined Template
SRA	Slice Resource Alternatives
DC	Data Center
VNF	Virtual Network Function
CPU	Central Processing Unit
API	Application Programming Interface
SLA	Service Level Agreement
DQN	Deep Q-Network
REST	Representational State Transfer

VLSP	Very Lightweight Network & Service Platform
RAN	Radio Access Network
YAML	Yet Another Markup Language
VDU	Virtualization Deployment Unit

Executive Summary

One of the main goals of the NECOS project is to define and implement an architecture that provides Slice-as-a-Service capabilities. To accomplish this in two years, the initial step was the identification of the main components in support of such a functionality, which was followed by the design and implementation of the architecture with a set of elements and methods to perform effectively the main actions to create, terminate, manage and monitor a slice in a cloud network environment. This document includes the internal details of all implemented components and the enablers employed on their implementation. New components (i.e. Slice Specification Processor, Service Orchestrator Adaptor, Slice Resource Orchestrator (SRO) and Infrastructure & Monitoring Abstraction (IMA) which are not described in D5.1, are introduced in this document, with an emphasis on the Slice Resource Orchestrator (SRO) and Infrastructure & Monitoring Abstraction (IMA). The document also provides design details about Elasticity, Machine Learning algorithms embedded to the SRO, and the Marketplace components.

1. Introduction

This deliverable presents the final design and implementation specification of the NECOS architecture, showing internal details of each component. In Deliverable D5.1, we presented the design and implementation of the following components: Slice Builder, Slice Broker, Slice Agent and the DC/WAN Slice Controllers. In this deliverable, D5.2, we revisit those components to show other internal details, mainly the components related to the Marketplace, i.e., Slice Broker and Slice Agent. New components - not elaborated on the previous deliverable D5.1 - are detailed in this document, namely, the Slice Specification Processor, Service Orchestrator Adaptor, Slice Resource Orchestrator (SRO) and Infrastructure & Monitoring Abstraction (IMA).

Regarding SRO, we show its internals, while giving attention to the elasticity actions that are necessary when a slice is scaled up or down. As mentioned in the previous deliverables describing the architecture - D3.1 and D3.2 -, the SRO is a crucial component since it is responsible for receiving a data model from the Slice Builder related to the new slice. The SRO will then make the stitching among the slice parts and notify the IMA to start monitoring the recently created slice.

Since the SRO is the module responsible for slice elasticity, machine learning algorithms were embedded in it to provide intelligent elasticity control. For this purpose, we implemented four options of algorithms to deal with elasticity: one threshold-based algorithm, two machine learning-based algorithms and one statistic-based algorithm.

The IMA is designed and implemented in two main parts: (i) Resource and VM Management, and (ii) Resource and VM Monitoring. The former is used to control the life-cycle management of the different service elements that are part of the slice while the latter is dedicated to collect monitoring metrics from the slice infrastructure. An example of a management operation implemented in this project is the deployment of the service, which is used by the SRO to trigger the deployment of services based on the slice resource specification received by the tenant. Also, other internal components of IMA will be described, such as Adaptors for VIM/WIM and a Monitoring Interface.

Although the previous deliverable D5.1 introduced the Marketplace components, in this current document, we present more details about its components, mainly regarding the Slice Agent and the way it selects resources from the provider. We also show details about scalability evaluations that were done to verify how the number of tenant requisitions may affect the performance.

Lastly, this document presents some refinements on the workflows presented in D5.1, which were necessary to reflect the interactions of the implemented components. For example, the slice creation workflow was rewritten to accommodate the deployment of the service. Similarly, elasticity workflows were updated to be in line with new findings of year two.

It is worth to mention the second year of the project was not only devoted to design and implement the components. The partners have put significant effort and time on their integration. The result was a unique platform that allows the entire slice-as-a-service with full control loops, starting from the creation of a slice, deployment and usage of service as well as the monitoring. In this sense, unit tests were initially done locally on local labs by each partner responsible for implementing specific components, and then, a fully step-by-step integration was performed to put all the components to run together. During this integration, several partners working in different geographical places made the final adjustments to have the first multi-domain transoceanic slice up and running between Brazil and Europe. All the calls done among the modules are compliant with the APIs designed in the Deliverables D4.1 and D4.2.

All the code implemented in NECOS is made open source and is available in https://gitlab.com/necos.

1.1. Deliverable Structure

This deliverable presents the final implementation of the NECOS components. We describe in detail the implementation of each component, while highlighting the NECOS capabilities that relate to it. In section 2, we present the design and the implementation of the NECOS components, describing internal details and the algorithms used. Section 3 presents the updates on the workflows, namely, the slice creation, slice decommission and elasticity. Finally, Section 4 concludes this report.

1.2 Contribution of this Deliverable to the project and relation with other deliverables

This deliverable describes the design and implementation of the NECOS system. Such implementation followed the API specifications done in D4.1 and D4.2 and the architecture definitions presented in D3.1 and D3.2. The API specifications define how the client interacts with NECOS as well as how NECOS components interact in a multi-cloud environment. However, internal calls among NECOS components not specified in D4.1 and D4.2 were designed and implemented in the context of WP5 and defined in this document. An example of such a call is the one from Slice Builder to SRO. In what regards the architecture, we followed the update done in D3.2 which are in line with our work carried out in the second year of the project. Finally, the implementation done in WP5 and described here is in line with the workflows defined in D5.1 and updated in this document. Such workflows highlight the main NECOS capabilities, namely, the slice creation and decommission, slice elasticity and service deployment.

2. Design and Implementation of the NECOS Architecture

As described in Deliverable 3.2, in NECOS, we have designed a specific slicing model that has as its foundation a mechanism to partition the underlying infrastructure into constituent slice parts and then combine these slice parts into a full end-to-end cloud network slice instance. This architectural design allows NECOS to create a powerful and flexible Slice-as-a-Service mechanism. To support such a mechanism, the architecture comprises of three main high-level sub-systems: the NECOS (LSDC) Slice Provider, the Resource Providers, and the Resource Marketplace.

2.1. Full Design of the NECOS Architecture

Before we present the details of every NECOS component, in Figure 1 we depict a general view of the NECOS Architecture design.

In Figure 1, it is possible to observe the three subsystems of our architecture: in blue, the Slice Provider, in orange, the Marketplace and in green, the Resource Providers. We can also see the interactions among the NECOS components. The red arrows represent the Client-to-Cloud API (specified in D4.1 and D4.2), the green arrows represent the Cloud-to-Cloud API (specified in D4.1 and D4.2), the purple arrows show the internal calls among the components, the black arrows depict the calls inside each component (internal sub-components), and finally, yellow arrows show the interactions between Marketplace and the Slice Agents. All those interfaces work together to support the Slice-as-a-Service key feature of the system.

Also, the figure shows an overall view of the full design of the NECOS Architecture in terms of technologies used for implementing the components. Most of the components were implemented in Python, however, Java was used to implement the IMA Resource & Monitoring and Prolog was adopted to implement an internal component of the Slice Agent. All the communications are done using HTTP/RESTful. Finally, Figure 1 depicts the technologies used for deploying the components. As can be seen, we adopted virtual machines (XEN) and microservices (Docker) for hosting the NECOS components.

Although this project does not intend to deliver a final ready-to-production system, we followed some best-practices approaches during the integration of the components. First of all, every partner responsible for the implementation did local/on premise unit tests. After being locally tested, the team worked together to make the full integration test. However, such a full integrated test was done individually for every workflow defined in D5.1 and D5.2 and implemented in this project. So, the sequence of integration tests was: (1) slice creation, (2) slice decommission, (3) service deployment, (4) vertical elasticity and (5) horizontal elasticity. In between the tests, we also integrated the monitoring (IMA) as well as the policies for vertical and horizontal elasticity.

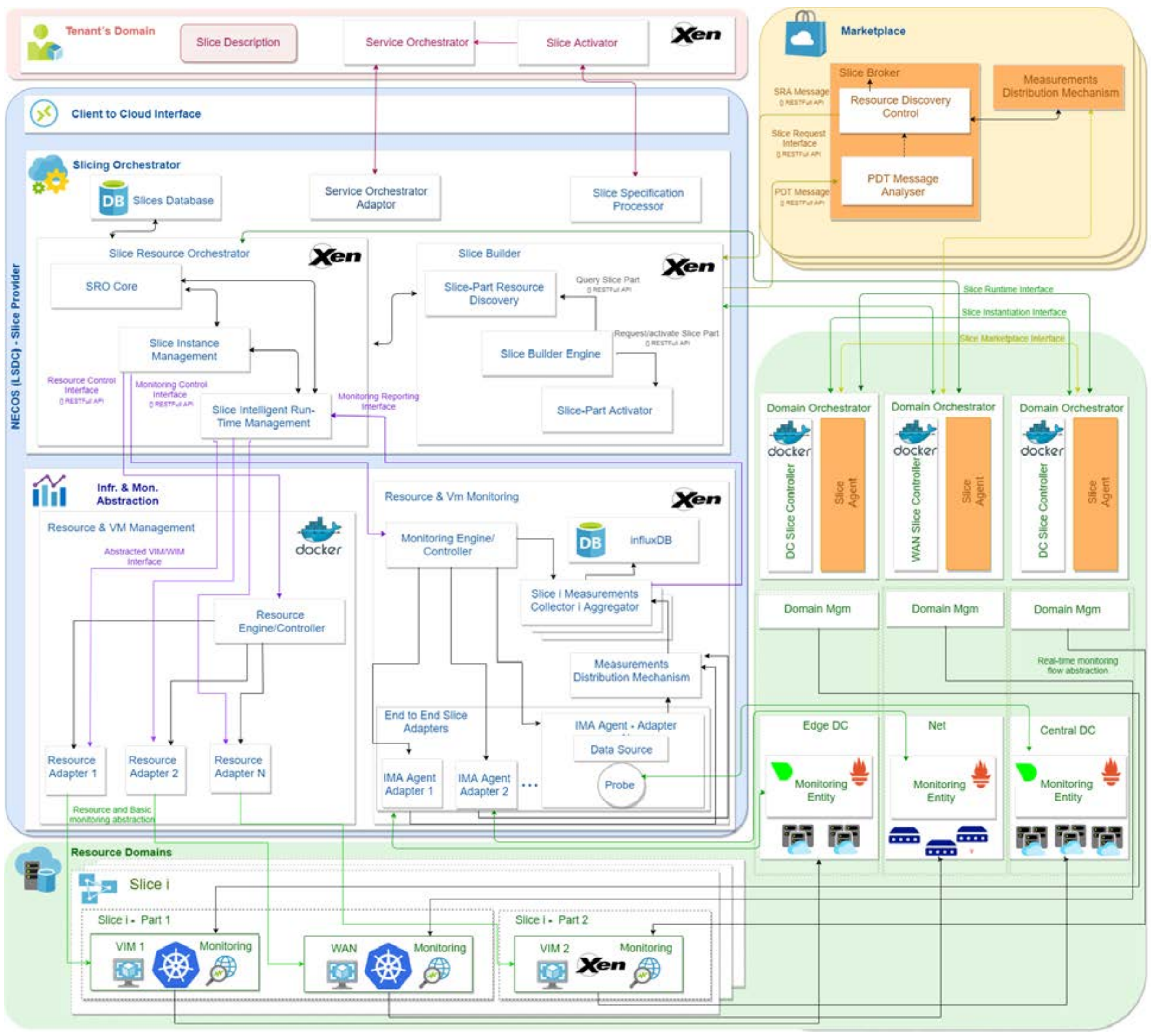


Figure 1: Full Design of the NECOS Architecture.

2.1.1. Components designed and implemented in D5.1

The following components were described in D5.1, however, they are present in D5.2 to show more internal details of each one.

- Slice Builder: Slice Builder is the component for creating and end-to-end slice by asking the Slice Broker to find candidate slice parts. On finishing the slice parts selection, the Slice Builder proceeds to prepare the slice, which is done by contacting the corresponding DC Slice Controllers and/or WAN Slice Controllers;
- Slice Broker: This component is responsible for contacting a set of resource providers (computing, networking and storage) to attend the tenant requirements to create an end-to-end slice. After finding candidate slice parts, the Slice Broker returns to the Slice Builder the alternatives;

- Slice Agent: Running inside each provider, it is responsible to interact with local controllers to gather available resources which may be used to create an end-to-end slice;
- DC/WAN Slice Controllers: These two components are running inside providers and are responsible for allocating the required computing, storage and networking resources for a given slice part, and returning a set of entry points to SRO.

2.1.2. New Components designed and implemented in D5.2

The components below were not defined in D5.1 and then, are fully described in this document.

- Slice Spec Processor: It takes as input a Slice Specification from the Slice Activator in the Tenant's Domain, and performs the processing tasks required to build a Slice creation request for the Slice Builder;
- Service Orchestrator Adaptor: It is the component that interacts with the tenant's Service Orchestrator. Its northbound interface talks with the southbound interface of the tenant's Service Orchestrator;
- Slice Resource Orchestrator (SRO): SRO can be considered as the core of NECOS system. It is, as the name suggests, responsible for orchestrating the main functions of the LSDC and coordinating the external and internal calls among the NECOS components. It is the component responsible for combining the Slice Parts that make up a slice into a single aggregated slice and also responsible for deploying the service as well as implementing the control loop for triggering the elasticity;
- Infrastructure & Monitoring Abstraction (IMA): The IMA is designed and implemented as a set of sub-components responsible for monitoring the end-to-end slice among different domains and technologies. The IMA is capable of providing to SRO an abstracted view of the virtual and physical resources of every slice in an isolated way so that the SRO is agnostic about the underlying VIM/WIM technologies. Also, IMA is responsible for the deployment and redeployment (update) of services running in each slice.

During the two years-project, we designed and implemented the key features of NECOS architecture. As such, the LSDC implementation supports the following features:

Table 1: Features implemented in the LSDC

● Slice-as-a-service	● Slice creation/Slice decommission	● Service deployment
● Service orchestration	● Vertical elasticity upgrade	● Horizontal elasticity upgrade
● Slice monitoring	● Resource discovery	● Slice parts selection
● Isolation	● Cost efficiency	● VIM-independence

The features shown in Table 1 represent most of the features necessary to have a Slice-as-a-Service system. The missing features are: service decommission and vertical and horizontal elasticity downgrade.

In the following subsections, we describe the design and implementation of the main components of each sub-system.

2.2. The NECOS (LSDC) Slice Provider

The NECOS (LSDC) Slice Provider is the sub-system that allows the creation of full end-to-end slices from a set of constituent slice parts. This sub-system comprises two main functional blocks:

- the ***Slicing Orchestrator,*** which performs operations that deal with slice creation, orchestration and decommissioning. This functional block is formed by the following components: Slice Specification Processor, Slice Builder, Slice Resource Orchestrator, Service Orchestrator Adaptor, and Slice Database;
- the **Infrastructure & Monitoring Abstraction (IMA)**, which interacts with the heterogeneous VIM / WIM and monitoring subsystems that are part of an end-to-end slice, being formed by the following components: Resource and VM Monitoring, Resource and VM Management, and Adaptors for VIM/WIM Control.

Below we detail the implementation of each functional component of the NECOS Slice Provider.

2.2.1. Slice Specification Processor

The **Slice Specification Processor** is the component that handles the request for slice creation coming from the **Slice Activator** (in the tenant's domain). The detailed design of the Slice Specification Processor is needed because it is an essential part in the slice creation workflow as defined in D5.1. It includes the following functionality which is not part of any other component: to receive the slice request and to parse it to generate a slice creation request to Slice Builder.

As described in Deliverable 4.2, there are three different ways to request the creation of the slice. In each approach, the tenant provides different specifications, reflecting different levels of abstraction: (i) the Slice Specification, the lowest abstraction level focusing on resource aspects, (ii) the Slice Requirements, specifying the general slice requirements and leaving to the **Slice Builder** and the **Slice Specification Processor** to determine the slice details, and (iii) the Service Specification that describes only the type of service to be deployed and delegates to the NECOS system the identification of the characteristics of the best suitable slice to be allocated.

Considering the three different approaches for slice creation specified in the NECOS architecture, we implemented both, the one where the Tenant provides the Slice Specification (option (i)), and the one where the Tenant provides the Service Specification (option (iii)).

Figure 2 illustrates a Slice Specification to create an end-to-end slice (**IoTService_sliced)** according to the information model specified in Deliverable 4.2. The slice contains three slice parts: two data center (DC) slice parts and one network (WAN) slice part to connect the two others. One of the DC slice parts (**dc-slice1)** must be located in Brazil and should host three nodes (**vdus**) managed by a Kubernetes Virtual Infrastructure Manager (VIM). The other DC slice part (**dc-slice2)** must be located in Europe and must host a Kubernetes cluster of two nodes. In both DC slice parts, the VIM must be instantiated on demand, i.e., the Mode-0 or VIM-independent slicing approach (**vim-shared=false)**. The WAN slice part connecting the two DC parts follows the Mode-1 slicing approach (**wim-shared=true)**, being based on the VXLAN technology. The WAN slice part has a bandwidth requirement of 1 GB.

```
      slices:
         sliced:
            id: IoTService_sliced
            slice-constraints:
               geographic: [BRAZIL, EUROPE]
               dc-slice-parts: 1
               edge-slice-parts: 1
               net-slice-parts: 1
            cost:
            ...
            slice-parts:
               - dc-slice-part:
                   name: dc-slice1
                   location: BRAZIL
                   monitoring-parameters:
                      tool: netdata
                      ...
                   VIM:
                       name: KUBERNETES
                       vim-shared: false
                       vdus:
                           - vdu:
                               description:   Master   of   kubernetes
cluster
                               instance-count: 1
                               hosting: SHARED
                               vdu-image: 'k8s-dojot-template'
                               epa-attributes:
                               ...
                           - vdu:
                               description:   Compute   of   kubernetes
cluster
                               instance-count: 2
                               hosting: SHARED
                               vdu-image: 'k8s-dojot-template
                               epa-attributes:
                               ...
              - dc-slice-part:
                 name: dc-slice2
                 location: SPAIN
                 monitoring-parameters:
                   tool: prometheus
                 VIM:
                    name: KUBERNETES
                    vim-shared: false
                    vdus:
                       - vdu:
                         description: Master of kubernetes cluster
                         instance-count: 1
                         hosting: SHARED
                         vdu-image: 'k8s-dojot-master-template'
                         epa-attributes:
                         ...
                       - vdu:
                         description: Compute of kubernetes cluster
```

```
                          instance-count: 1
                          hosting: SHARED
                          vdu-image: 'k8s-dojot-worker-template'
                          epa-attributes:
                          ...
               - net-slice-part:
                   name: pop-dc-slice1-to-pop-dc-slice2
                   WIM:
                     name: VXLAN
                     wim-shared: true
                   links:
                     - dc-part1: dc-slice1
                     - dc-part2: dc-slice2
                     - requirements:
                       bandwidth-GB: 1
```

Figure 2: Slice specification according to D4.1

When requesting a slice creation using a Slice Specification, the **Slice Specification Processor** takes as input a Slice Specification from the Slice Activator in the Tenant's Domain, and performs the processing tasks required to build a Slice creation request for the Slice Builder.

Below, we present a solution that was implemented in NECOS to infer slice requirements from non-structured service description, option (iii) explained above.

2.2.1.1 Inferring Cloud-Network Slice's Requirements from Non-Structured Service Description

To support future computing and communication scenarios, cloud-network management tools should deploy cloud-network services adopting uncomplicated ways, reducing not only the time to market but also broadening the community capable of deploying new services. In this research initiative, we investigate alternative techniques to support NECOS Platform towards slice-as-a-service creation from non-structured service description. In this section we present the initial efforts we took for structuring such a mechanism at the Slice Specification Processor module using machine learning.

Figure 3 illustrates the overall scenario investigated in this section. As can be seen, the process starts with a tenant providing a service description to our learning mechanism, using abstract (non-infrastructure-specific) information. The learning mechanism implements a function f(x), capable of predicting structured descriptions of infrastructure requirements for new slices.

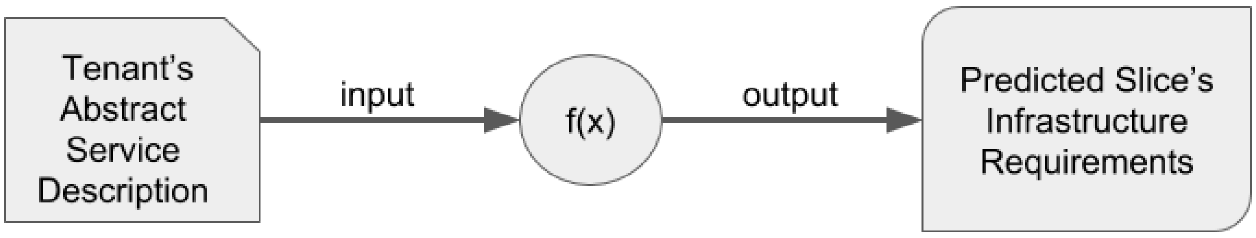

Tenant's Abstract Service Description
input
f(x)
output
Predicted Slice's Infrastructure Requirements

Figure 3: Learning infrastructure requirements for new slices based upon abstract service descriptions provided by tenants.

In order to tackle this challenge, we leverage the concepts of Structured Output Learning (SOL) [BAKLR 2007], a supervised learning umbrella useful for inferring functional dependencies between arbitrary input and output domains. Rather than predicting discrete elements or real numbers, SOL predicts structured objects that must satisfy rules/constraints specified in its domain. The basic definition of the problem addressed by this work is stated in the following equation:

$$\hat{y} = \underset{y \in Y}{argmax} f(x, y)$$

Where x is the input information, Y is the set of all possible outputs and f(x,y) is a compatibility function to evaluate how accurate is the fit of y to x. The prediction ŷ is the element of set Y, output of the trained model that better fits x. Therefore, the objective is to find a model M: $x_i \rightarrow y_i$, such that $\hat{y}_i$ closely approximates y_i for a given x_i request.

Although Y is finite, it can be very large, making unfeasible the adoption of classifiers and other techniques that exhaustively try different values for y. Common techniques adopted in the literature, and also currently available in tools like PyStruct [JMLR 2014], include energy functions or conditional random fields (CRFs) for building f(x,y) [LAFFERTY 2001].

As an example of such a mapping, suppose that a logistics industry needs to perform asset tracking. With asset tracking an enterprise should be able to easily locate and monitor key assets such as materials, products, or containers, in order to optimize logistics, maintain inventory levels, monitor quality and detect theft.

Although it may be a large tenant, a maritime shipping company, for example, may not want to get into the technological implementation details of a cloud-network slice that supports its assets IoT-based tracking systems. Sensors may help to track the location of a ship at sea, and they can provide the status and temperature of cargo containers. So the shipping company may want to describe its slicing needs as follows:

> *"Provide a cloud-network slice to serve an asset-tracking service. The service must run on the ports of Barcelona, Thessaloniki and Santos. The service must automatically adapt to the demand which will change depending on the time of the year. It is expected to track around 50,000 containers at the same time, with peaks of 80,000 at a frequency of 12 samples per minute. The collected data must be stored inside European Union. Information system's availability should be 99.998%, and data collection availability must be 99.9% as a minimum. Queries to the data, from any city, should have a response time of 100ms or less and will occur at a peak rate of 20 per minute."*

To translate those requirements into a structured slice-requirement, i.e., a PDT message, the mapping process has to perform several tasks. They will be described in more detail later in this section, but in a sketchy way they are: i) to identify the service (or services) to be provided, ii) to determine the expected workload, and iii) to identify the constraints.

Each of these tasks may have different sub-tasks, some of them regarding quite different domains. The identification of the service is not a complicated task. Once the service is recognized from the description, it should be matched against known services setups and its particular requirements on CPU, memory, storage, and network must be identified.

Then, the service's workload has to be understood. This case might be more complicated. It is not so easy to verify if the workload characterizing the service relates to the number of shipping containers to track, or the query rate that the database has to manage. In this case, we can say that the workload is mainly given by the number of tracked shipping containers per time unit, but it is not so clear from the point of view of an automatic text processing algorithm.

Finally, service constraints have to be identified: geographical restrictions --in this case the mentioned cities and ports--, service-performance constraints --response time for the queries--, and availability for the data collection and information subsystem. NECOS will, then, create a high-level specification like the one in Figure 4. There, we just depict a fragment of the specification (the complete one is too long for this section), but it provides an idea of how natural language expressions such as *"Queries to the data, from any city, should have a response time of 100ms or less".*

```
...
slice-constraints:
   geographic:
         continent:
         country:
         city: [Barcelona, Thessaloniki, Santos]
         dc-slice-parts: 3
         net-slice-parts: 2
slice-requirements:
   elasticity: true
         target:
         service_metric: query_response_time
         value: {lower_than_equal: 100ms}
   reliability:
         enabled: true
         value: logical-backup
service:
   service-function:
         service-type: dojot
...
```

Figure 4: Fragment of the high-level YAML-based slice specification.

In order to allow tenants to provide service descriptions using natural language, our proposal is to adopt Recurrent Neural Networks (RNN). RNNs are a family of neural networks for processing sequential data. Its connections between nodes form a directed graph along a chain. This allows to display a temporary dynamic behavior during a time sequence. Unlike neural feedback networks, RNNs use their internal state to process input sequences. This has made them successful on tasks such as handwriting or voice recognition. RNNs have been successful, for example, in the learning sequence and tree structures in natural language processing. This success makes them an obvious candidate to pursue our goal.

An exemplification on how to adopt RNNs to instantiate our proposal of Figure 3 is depicted in Figure 5. Basically, tenants can provide information using, for example, input forms. In Figure 5, tenants provide information structured in three fields. The first field receives information, using natural language, that describes the service requested by the tenants. The tenant can also select a service among options in a combo list. The combo selection goes directly to a service description, while textual information passes through an RNN to identify the service requested by the tenant.

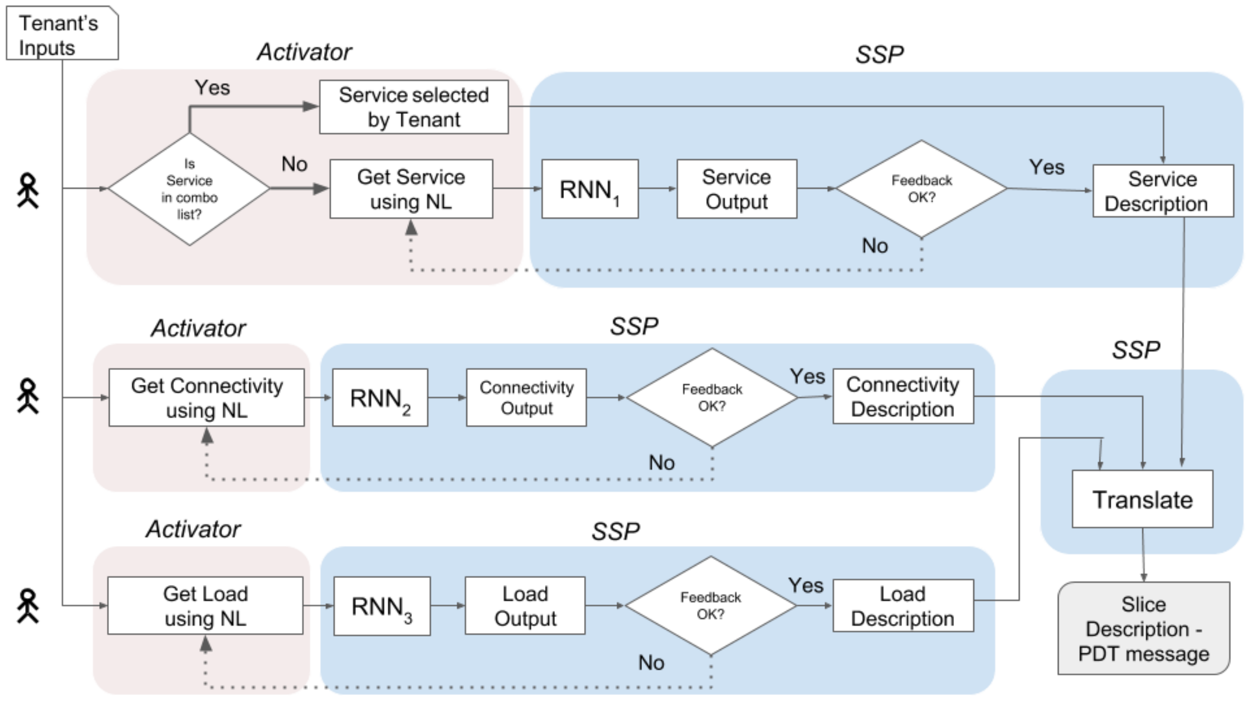


Figure 5: Example for mapping tenant's inputs into slice descriptions using information forms structured in three fields.

The second field receives information that describes all connectivity/geographic aspects of the requested slice. Such information passes through a second RNN, trained to translate such aspects of the request into connectivity description.

The third field receives information that describes load aspects of the requested slice. Such information passes through another RNN, trained, in this case, to translate such aspects into load description.

The treatment of information by RNNs can be seen as a first phase within SSP. The outputs of this phase is then used as a controlled input towards a second phase, in which, for example Structured Output Learning mechanism described earlier is instantiated. SOL translates such controlled input of first SSP phase into PDT messages of NECOS. The first phase can be seen as a way of taming the complexity of such translation.

2.2.2. Slice Builder

The **Slice Builder** is responsible for building a full end-to-end multi-domain slice from the relevant constituent slice parts. When the PDT message has been specified, the **Slice Specification Processor** sends such a message to the **Slice Builder** invoking the **initiate_slice_creation()** method**.** The **Slice Builder** consists of the following functions, as depicted in Figure 6:

Slice Builder Engine: This is the main **Slice Builder** function responsible for initiating the slice resource discovery process by forwarding the PDT message, received from the **Slice Specification Processor**, to the

Slice-Part Resource Discovery function. The **Builder Engine** is also responsible for processing the Slice Resource Alternatives (SRA) message received via the **Slice-Part Resource Discovery** from the **Slice Broker**, and selecting slice resources that best fit the Tenant's requirements. In our implementation, from the set of alternatives presented for each slice part in the SRA message, the **Builder Engine** selects the one with lower cost. However, other strategies based on optimization models could be formulated to find the optimal resource instantiation as shortly explained in Section 2.2.2.1. Upon deciding on a resource instantiation, the **Builder Engine** forwards the instantiation description to the **Slice-Part-Activator** function. Finally, when the Slice Discovery and Instantiation phases are complete, it returns the complete slice details to the **Slice Resource Orchestrator**;

- **Slice-Part Resource Discovery:** This function is responsible for sending the PDT to the **Slice Broker** and receiving the SRA message. This is done by invoking the **locate_slice_resources()** method implemented by the **Slice Broker**. Thus, **the Slice-Part Resource Discovery** function implements the message interface between the **Slice Builder** and the **Slice Broker**. Its role is to check the integrity of the PDT message and the SRA message received. The reply (SRA) is then forwarded to the **Slice Builder Engine**;
- **Slice-Part Activator**: The role of this function is to receive an almost complete slice parts' description from the **Slice Builder Engine** and contact each selected DC and WAN providers to allocate the required resources. The **Slice-Part Activator** contacts each selected **DC/WAN Slice Controller** invoking the **request_slice_part()** method. Upon successful resource allocation, it receives back the identification of the allocated slice part (Slice Part ID). As soon as all the resources are successfully allocated in all slice parts, the **Slice-Part Activator** contacts each **DC/WAN Slice Controller** to actually instantiates the slice part. This is done invoking the **activate_slice_part()** method implemented by the **Slice Controllers**. After activating the slice part, the **Slice-Part Activator** receives back the details regarding each particular slice part (e.g., entry points for: ssh, monitoring system, the VIM API). When all slice parts have been activated, the **Slice-Part Activator** sends back to the **Builder Engine** the complete slice information.

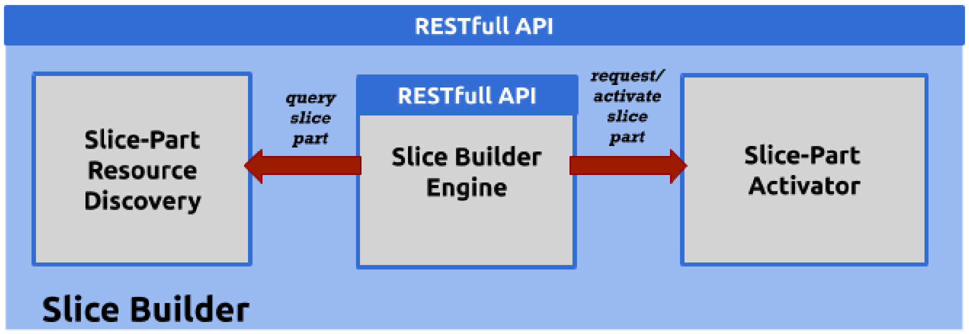


Figure 6: Slice Builder internal functions.

Figure 7 illustrates part of a possible SRA message returned to the **Slice Builder** as a response to the Slice Specification of Figure 2. In this case, the **Slice Builder** selects **UNICAMP** to host the slice part **dc-slice1,** as it provides the lowest data center cost.

Upon selecting **UNICAMP** as the provider, the **Slice Builder** will contact the DC Slice Controller corresponding to the **UNICAMP** domain using the IP address and port number provided in the SRA message. Figure 23 illustrates the YAML file that the **Slice Builder** generates to request the slice part **dc-**

slice1 in the **UNICAMP** domain. The **Slice Builder** completes the information by adding the IP address of each **vdu** in the slice data plane.

For each DC slice part returned in the SRA message, the **Slice Builder** selects the alternative with lower cost and contacts the **DC Slice controller** in charge of creating slice parts in that data center to reserve the required resources. Similar, for each net slice part returned in the SRA message, the **Slice Builder** contacts the **WAN Slice Controller** of that site to reserve the required bandwidth. When all the required resources are successfully allocated in each domain, the **Slice Builder** contacts the respective **DC/WAN Slice Controllers** to actually instantiate the slice-parts. When all the slice-parts are successfully created, the **Slice Builder** calls the **SRO** passing a YAML file describing the end-to-end slice just created. The **SRO** then stores this description in the **Slices Database** (a Neo4j database) and the **Slice Builder** returns the YAML file to the **Slice Specification Processor**.

2.2.2.1 Slice parts selection algorithms

The efficient selection of the resources for the cloud network slice is not a trivial task, it is usually done in an autonomic fashion, with optimization techniques in charge of the network and data center resources selection. Optimization techniques are commonly used to select resources in the service chain context, although this could also be used in the context of cloud network slices in NECOS, with the actual physical resources specified in the Slice Parts. These techniques can be separated in heuristic methods and Mixed Integer Linear Programming (MILP), which are described next.

Mixed Integer Linear Programming (MILP)

Within the MILP method, the goal is to find the global maximum or minimum solution in a linear function, avoiding the local peaks and valleys solutions which are not guaranteed to be as much optimized as the global one. MILP has a linear function representing the objective equation, this mathematically aggregates the relevant metrics for the choosing of resources, e.g. the financial cost, amount of RAM memory, storage, number of CPU cores and energy consumption in the data center Slice Parts and the financial cost and the bandwidth of the Network Slice Part. To specify the range of the variables in the objective equation, constraints specify the value limits for parameters, e.g., being positive and integer values for resources. The constraint could also contain the performance requirements specified by the tenant.

Some algorithms are commonly used in MILP applications like Criss-Cross, branch-and-bound and Simplex. The Simplex algorithm is the most used for resource selection between the MILP ones. Simplex represents the objective and the restrictions equations through a matrix where the variables are the columns and the equations themselves are the lines in the matrix. After this representation, it creates a canonical matrix to manipulate the equations and find the maximum and minimum values for each parameter. Some linear optimizers could be cited for the possible use in NECOS, such as IBM CPLEX [CPLEX 2019], Matlab [MATLAB 2019], R [RPROJECT 2019], Python SciPy [SCIPY 2019], Python PuLP [PULP 2019], Microsoft Excel Solver [MICROSOFT 2019], among others.

Heuristic methods

A heuristic formulation is a set of directives (heuristics) which specifies the objective and constraints of a problem, e.g., find the cheapest Slice Part residing in Europe. This approach often finds the solution in a faster manner, without the need for trying all the possibilities within the scope of feasible answers. In contrast, due to these characteristics, the first solution attending the constraints is often final, leading to local solutions in the peaks and valleys of the function and avoiding the most optimal solution. Some algorithms that could be used in the context of NECOS is the Greedy Algorithm and Local Search.

```
slice-parts-options:
         - dc-slice-choices:
             name: dc-slice1
             alternatives:
                 - dc-slice-part:
                     location: BRAZIL
                     cost:
                         dc-model:
                            model: COST_PER_PHYSICAL_MACHINE_PER_DAY
                            value-euros: 5
                     dc-slice-controller:
                         dc-slice-provider: UNICAMP
                         ip: 192.168.111.1
                         port: 5000
                     ...
                 - dc-slice-part:
                     Location: BRAZIL
                     cost:
                         dc-model:
                            model: COST_PER_PHYSICAL_MACHINE_PER_DAY
                            value-euros: 9
                     dc-slice-controller:
                         dc-slice-provider: UFU
                         ip: 164.172.256.2
                         port: 5000
                      ...
```

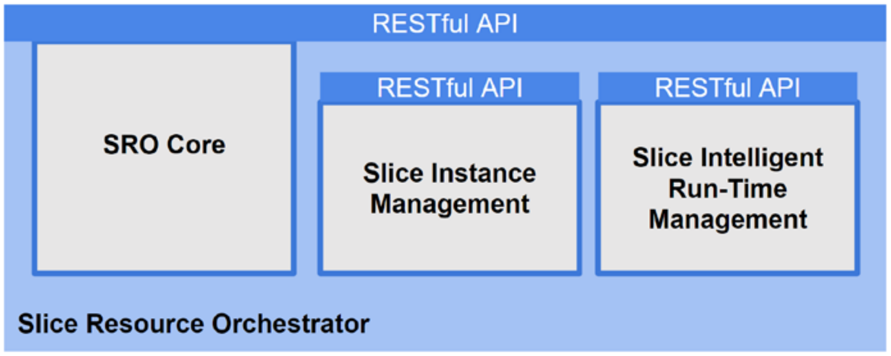
Figure 7: SRA message for the Slice Specification presented in Figure 2.

2.2.3. Slice Resource Orchestrator (SRO)

As described in Deliverable 3.1, the Slice Resource Orchestrator (SRO) is the component responsible for combining the Slice Parts that make up a slice into a single aggregated slice. It is also responsible for orchestrating the service elements across the Slice parts that make up the full end-to-end slice, and as such, it is the component that handles the actual placement and embedding of VMs into the resource domains. It is involved in four main tasks:

- End-to-End Slice Activation and Registration;
- End-to-End Slice Elasticity;
- Decommissioning of the End-to-End Slice;
- Instantiation of Virtual Resources for the Services.

Following those tasks, we decided to organize the SRO as depicted in Figure 8. The component itself is a distributed system implemented as a set of RESTful services. The *SRO Core* is in charge of exposing the component external interfaces and of performing all generic and auxiliary tasks; the *Slice Instance Management* service, being in charge of all the tasks related to the end-to-end slice activation and registration, the instantiation of virtual resources for the services, and the slice decommission; finally, the *Slice Intelligent Run-time Management* is the component responsible for taking more complex decisions such as to when triggering an elasticity process, and if it has to be vertical or horizontal.


RESTful API
SRO Core
RESTful API
Slice Instance Management
RESTful API
Slice Intelligent Run-Time Management
Slice Resource Orchestrator
Figure 8: Internal SRO architecture.

Among the three components, the functionality described in the workflows presented in Deliverable 5.1 were considered as input and the SRO functionalities described in them were distributed among the internal components. The three internal components implement the following functionalities:

a) SRO Core, which is the subcomponent exposing all external interfaces and also responsible for:

    1. Forwarding the VIM/WIM Pointers;
    2. Updating the Slice Pointers to VIM/WIM;
    3. Updating the Slice Topology.

b) Slice Instance Management (SIM), which is responsible for the following aspects:

    1. Confirms the Slice Decommission;
    2. Inactivates the Slice Topology;
    3. Receives the Full Slice Details;
    4. Recovers the Slice Topology;
    5. Removes the VIM/WIM Slice Pointers;
    6. Requests the E2E Binding of Parts;
    7. Confirms the Binding;
    8. Requests the Slice Parts Decommission;
    9. Stores Full Slice Details.

c) Slice Intelligent Run-Time Management (SIRTM), which is responsible for:

    1. Evaluating the Monitoring Data of Slice;
    2. Defining the Requirements for Elasticity;
    3. Requesting Horizontal Slice Elasticity;
    4. Requesting Vertical Slice Elasticity;
    5. Confirmation of Slice Elasticity.

2.2.3.1 SRO Elasticity Algorithms

The main idea behind the modular design of the SRO is to isolate one of its functionalities, i.e., the elasticity management. Having all the elasticity features implemented as independent RESTful services permits us to develop and try different approaches, techniques, and complexities for a task that is a complicated closed-loop control process.

As defined in D5.1, in the NECOS Project, elasticity is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources (computing, networking and storage) in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible. More than that, in the NECOS Project, this elasticity considers that the slice is provisioned in a multi-domain and multi-technological environment and the run-time change of resources as such requires a sophisticated mechanism for orchestration.

We have implemented three different elasticity strategies: (i) a classic strategy based on thresholds, (ii) two algorithms based on machine learning techniques, and (iii) one statistic-based algorithm.

Thresholds-based elasticity

This strategy is intended to drive elasticity by means of threshold crossings of specific parameters being monitored by the SRO component. For this purpose, the SRO implements the threshold-based programmable mechanism described below.

Regarding the monitoring data

The SRO gathers monitoring data from the IMA component where the resource-usage metrics are monitored and stored. The monitoring frequency and type of data are being defined by the SRO on a per-slice basis.

Regarding the threshold settings

The monitoring data is used by the Slice Intelligent Run-Time Management (SIRTM) to drive elasticity. Thresholds over the monitored data are being set by the SRO to define when to trigger elasticity. In this regard, thresholds are defined in terms of percentage of a given resource or in terms of given value of a resource metric, e.g. MBytes, bits per second, msecs, etc.

In ecosystems controlled by the NECOS approach, resource usage is very dynamic and prone to change drastically so that it is necessary to implement a mechanism to avoid unnecessary elasticity triggering due to short statistical fluctuations of resource usage. This mechanism is a time window, which is implemented to make sure that statistical fluctuations of resource usage do not render to unnecessary elasticity triggers. This way, when a given threshold is crossed, the crossing should hold for a given time window to actually trigger the elasticity process.

To test the functionality of this strategy, an experiment for vertical elasticity was set. The purpose of this experiment was to maintain an acceptable performance of memory in one virtual machine between 100 and 400 MBytes. When memory usage was above 400MBytes it indicates that the current resources are insufficient to handle the workload, so the SRO triggers elasticity upgrade to increment the number of virtual machines. When memory usage was under 100MBytes it indicates that the current resources can be reduced and keep an acceptable memory usage under the above threshold and save money from the tenant’s point of view. For this experiment, there was a generator of simulated metrics that started generating data within an acceptable range, then it was forced to generate data over/under the threshold, so the SRO was capable of triggering elasticity (upgrade or downgrade), and after the elasticity was completed the generator generates data within the acceptable range again.

For example, Figure 9 shows the behavior of the above mechanisms for a threshold crossing upwards of memory usage defined in 400MBytes. In this example, there are crossings upwards the threshold, however elasticity upgrade is triggered only when the threshold is crossed during a value of time higher than the time window defined by the SRO. This way, the SRO avoids triggering unnecessary elasticity.

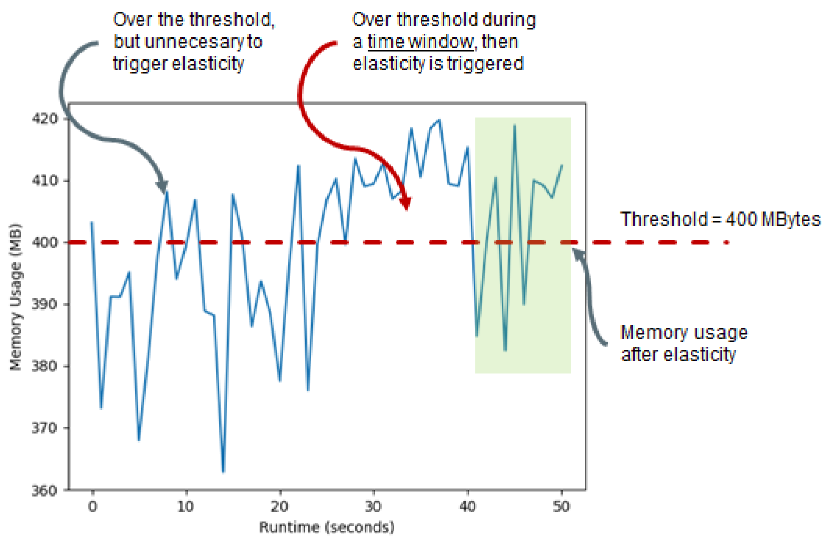


Figure 9: Memory usage threshold crossing upwards.

Figure 10 shows another example of the threshold-based elasticity approach and illustrates a threshold crossing downwards of memory usage defined in 100MBytes. Similarly, elasticity downgrade (in this case a hardware scaling-down) is triggered only when the threshold is crossed downwards during a value of time higher than the time window defined by the SRO. This way, the SRO avoids triggering unnecessary elasticity.

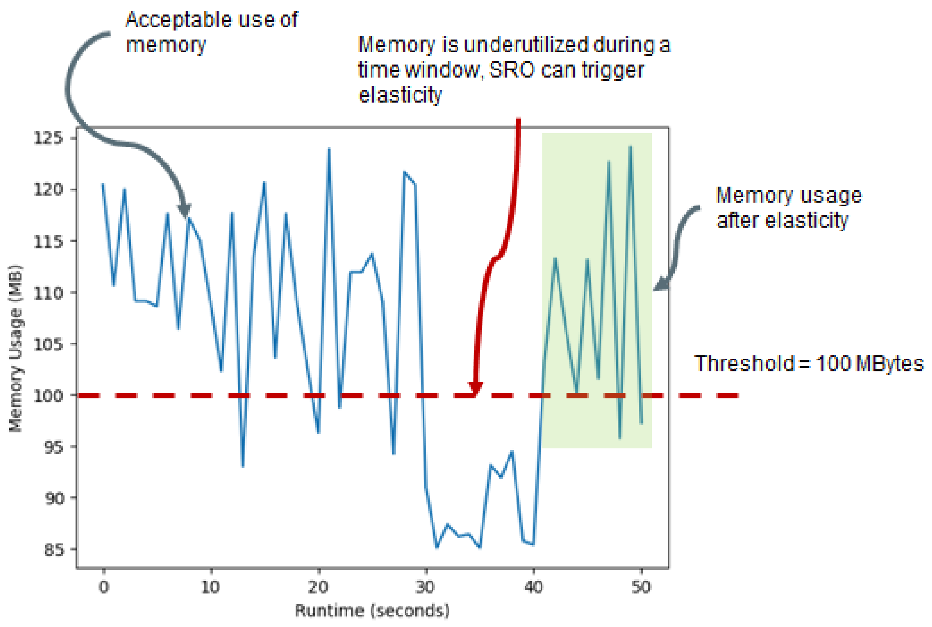


Figure 10: Memory usage threshold crossing downwards.

Regarding the elasticity strategy
Making use of the threshold-based elasticity approach described above, the SRO implements elasticity strategies of how soon (or how late) to trigger elasticity based on the specific data monitored, and the kind of elasticity (horizontal or vertical). All in all, on a per slice basis.

Learning-based elasticity

This implementation exercises the adoption of machine learning to perform elasticity. The proposed mechanism is composed of four steps: (i) KPI Estimation; (ii) SLA Prediction; (iii) Slice Resources Optimization; and (iv) Enforcement of Slice Modifications. These four steps are defined below.

- ***KPI Estimation***

Slices are associated to SLAs in which KPIs are listed as a means of controlling the fulfilment of quality parameters. Examples of common KPIs include CPU, memory, network traffic, storage, and others. As an alternative, aiming to better integrate the orchestration of slice's infrastructure and the actual service running inside it, this implementation considers that SLAs are specified in terms of Service KPIs. For example, a Service KPI could be the response times while executing a read/write operation, the number of frames per second delivered in a video distribution service, and others.

By adopting this Service KPIs approach, this step builds upon supervised learning, more specifically as a regression and/or classification problem. We are interested in how the slice's infrastructure statistics, referred to in this section as X, relate to the Service KPIs on the end-user side, referred to in this section as Y. The infrastructure statistics X include measurements from the slice, spread over the multiple infrastructure providers. The performance indicators Y on the end-user side refer to service-level metrics, for example, frame rate and response time, as mentioned before.

The metrics X and Y evolve over time, influenced, e.g., by the load on the infrastructure, operating system dynamics, network traffic, number of active clients. We model the evolution of the metrics X and Y as time series $\{X_t\}$, $\{Y_t\}$, and $\{(X_t, Y_t)\}$. Our objective is to estimate the Service KPI Y_t at time t, based on knowing the slice's infrastructure statistics X_t. Using the framework of statistical learning, the problem is finding a model $M: X_t \rightarrow \hat{Y}_t$, such that $\hat{Y}_t$ closely approximates Y_t for a given X_t.

- ***SLA Prediction***

The objective of this step is to predict the SLA in a given point in the future, enabling the SRO to proactively adapt the slice so the SLA can be properly maintained. In order to do it, this step receives the time series $\{\hat{Y}_t\}$, estimated in the previous step, and performs trend analysis. There are several techniques available in the literature to perform such trend analysis, especially in the stock market prediction. We perform it using deep learning with the support of Tensorflow [TF 2019].

In essence, such a prediction grants a □t to SRO operation. Also, using the learning framework, the problem being tackled in this step is to train a deep neural network DNN: $\{\hat{Y}_t\} \rightarrow \hat{Y}_{t+□t}$, such that $\hat{Y}_{t+□t}$ closely approximates the future $Y_{t+□t}$.

- ***Slice Resources Optimization***

Adjusting the resources of slices is a complex problem to be solved. Slices involve many different resources, located in different providers, that can receive different types of loads along the slice's lifetime.

As a way of reducing the complexity of this problem and, at the same time, enabling autonomic adaptations to the arrangement of resources serving slices, this step builds upon the concept of slice flavors. When requesting a slice, tenants provide the flavor to its slice creation process, together with a set of flavors which the tenants allow NECOS to use while performing elasticity (both, downgrade and upgrade).

Similarly, to the previous steps, our approach to performance slice's resource optimization is based upon statistical learning whereby the behavior of the target system is learned from observations. Fundamental to this step is the concept of the Capacity Region [PASQUINI 2019]. Given a service and an SLA, the capacity region characterizes the load that the underlying slice infrastructure can carry while conforming to the SLA.

Figure 11 shows the capacity region for read and write operations performed on a key-value store running on our testbed. In this case, the load space has two dimensions, comprising the rate of read operations (horizontal axis) and the rate of write operations (vertical axis), respectively. The (carried) load of the system at any time is thus a vector in this space. The capacity region is shown in green and describes all possible load vectors the system can support while conforming to the SLA.

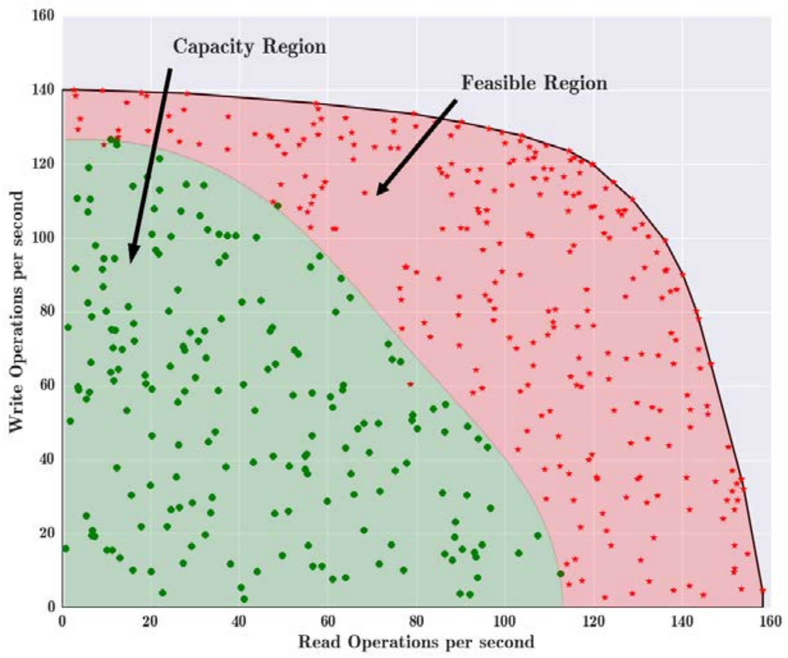


Figure 11: The capacity region and the feasible region for a key-value store service.

The red area in Figure 12 describes all possible load vectors that violate the SLA. The green and red areas combined make up the feasible region for the service on the given infrastructure. The boundary of the capacity region is the line (manifold) that separates the green and the red parts of the load space. It has been learned through measurements, using a classifier. Green dots in the figure show load vectors that conform to the SLA, red dots show vectors that violate the SLA.

Note that by learning the capacity and feasible regions, this step three can use the estimated time series of step one and the predictions of step number two to: 1) infer the movements performed by the service running inside the slice and 2) evaluate where the service tends to be located in the near future, as depicted in Figure 12.

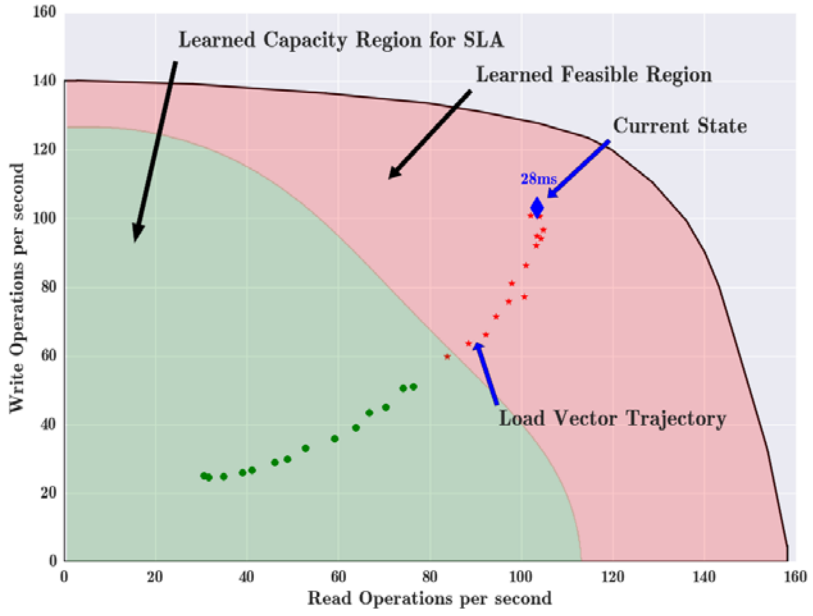


Figure 12: Trajectory of the load vector in the load space. The current response time is displayed for illustration purpose.

In order to optimize the resources for a given slice, step number three relies on the learned capacity and feasible regions for the different flavors specified by the tenant during its slice request. Basically, step number three evaluates the current load vector against the set of feasible regions and selects the best arrangement for the slice, i.e., the arrangement in which the predicted load vector is not only located in the capacity region, but also the one which saves the most resources actually located for the slice.

- ***Enforcement of Slice Modifications***

This final step enforces the required modifications to the slices infrastructure by using the Cloud-to-Cloud API of NECOS. However, besides the infrastructural modifications, this final step also triggers the internal updates to the intelligent mechanisms described from steps one to three, allocating the model *M* properly trained for the current slice arrangement at step one, allocating the deep neural network DNN for step number two, and allocating the current capacity and feasible regions for step number three.
An alternative approach applying reinforcement learning follows.

Reinforcement Learning-based elasticity

This implementation exercises the case in which the elasticity is addressed, at the same time, separately in two different realms: service elasticity and platform elasticity. The first, responds to service-related metrics, e.g., a web server response-time, and acts on the virtual substrate adding or removing service virtual resources, and the second observes how the platform (normally physical) resources are being used and modify the amount or location of those resources assigned to a particular slice. In this case, we consider these two control loops are decoupled, as depicted in Figure 13, but potentially interfering with each other.

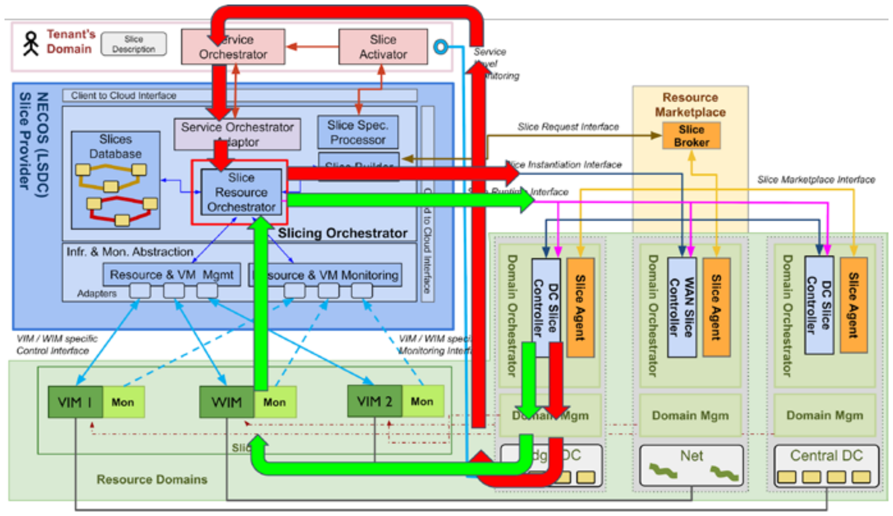


Figure 13: Two control loops for elasticity in the PaaS-like provision model.

The control loop dealing with the services, red in Figure 13, might be of two kinds:

- One pure service-elasticity, adding and removing virtual resources, or;
- One for platform-elasticity, asking NECOS to add resources (it may be horizontal elasticity).

The metrics considered as reference for the control process are typically service metrics such as Response Time, or Quality of Experience. The actions taken by the "controller" are Vertical Elasticity actions, i.e., to add more service instances within the slice as it is.

The control loops related to the platform part of the slice might be of two kinds too:

- One that adds or removes physical resources to keep a reference infrastructure metric, or;
- One that adds or removes slice parts with the purpose of keeping different slices isolated.

The metrics to be considered the reference for the control process, are typically infrastructure metrics such as CPU usage, or slicing-related metrics such as the Isolation Index. The actions taken by the "controller" are Horizontal Elasticity actions, i.e., to add, remove or "move" slice parts.

To implement these two loops in this particular instance of the SIRTM, we have used a machine learning strategy consisting of:

- Modelling the system in Simulink as close to reality as possible, paying special attention to the different latencies induced by each process, mainly the time it takes to deploy new slice parts or service instances. These large delays are an important challenge for any control process and some reinforcement learning techniques are specially fitted to work with them;
- to train two Deep Q-Network (DQN) agents on the simulation platform, one for each control loop;
- to export the trained agents to the real prototype, and;
- to let the DQN agents to control the elasticity process while they keep learning to adapt the simulation-based trained networks to the real system. This methodology used in works such as [DKMMRT 2011] permits to use a reinforcement learning agent in a production system avoiding most of the random experimentation made by the learning process.

The deep Q-network (DQN) algorithm [MNIH 2013] is a model-free, online, off-policy reinforcement learning method. A DQN agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards. DQN is a variant of Q-learning. DQN agents can be trained in environments with continuous or discrete observation spaces and with only discrete action spaces, which are in line with our necessities.

During training, the agent:

- Updates the critic properties at each time step during learning;
- Explores the action space using epsilon-greedy exploration. During each control interval the agent selects a random action with probability ϵ, otherwise it selects an action greedily with respect to the value function with probability $1-\epsilon$. This greedy action is the action for which the value function is greatest;
- Stores past experience using a circular experience buffer. The agent updates the critic based on a mini-batch of experiences randomly sampled from the buffer;

To estimate the value function, a DQN agent maintains two function approximations:

- Critic Q(S,A) — The critic takes observation S and action A as inputs and outputs the corresponding expectation of the long-term reward;
- Target critic Q'(S,A) — To improve the stability of the optimization, the agent periodically updates the target critic based on the latest critic parameter values;
- Both Q(S,A) and Q'(S,A) have the same structure and parameterization.

When training is complete, the trained value function approximation is stored in critic Q(S,A). DQN agents use the following training algorithm, in which they update their critic model at each time step:

- Initialize the critic Q(s,a) with random parameter values θ_Q, and initialize the target critic with the same values: $\theta_{Q'}=\theta_Q$.
- For each training time step:
    - For the current observation S, select a random action A with probability ϵ. Otherwise, select the action for which the critic value function is greatest.

$$A = \arg\max_{A} Q(S, A|\theta_Q)$$

    - Execute action A. Observe the reward R and next observation S'.
    - Store the experience (S,A,R,S') in the experience buffer.
    - Sample a random mini-batch of M experiences (S_i,A_i,R_i,S'_i) from the experience buffer
    - If S'_i is a terminal state, set the value function target y_i to R_i. Otherwise set it to:

$$\begin{aligned} A_{\max} &= \arg\max_{A'} Q(S_i', A'|\theta_Q) \\ y_i &= R_i + \gamma Q'(S_i', A_{\max}|\theta_{Q'}) \end{aligned} \quad \text{(double DQN)}$$

$$y_i = R_i + \gamma \max_{A'} Q'(S_i', A'|\theta_{Q'}) \quad \text{(DQN)}$$

    Update the critic parameters by one-step minimization of the loss L across all sampled experiences.

$$L = \frac{1}{M}\sum_{i=1}^{M}(y_i - Q(S_i, A_i|\theta_Q))^2$$

    - Update the target critic depending on the target update method (smoothing or periodic).

$$\begin{aligned} \theta_{Q'} &= \tau\theta_Q + (1-\tau)\theta_{Q'} \quad &\text{(smoothing)} \\ \theta_{Q'} &= \theta_Q \quad &\text{(periodic)} \end{aligned}$$

We now elaborate our statistics-based approach to elasticity.

Solidarity Elasticity Mechanism

In addition to the classical scheme for vertical elasticity functions, we introduce a new strategy denoted Solidarity Elasticity [MEDEIROS, 2019], which employs a fine-grained statistic-based resource-computing algorithm. The Solidarity Elasticity follows an algorithm capable of indicating resource amounts to drive scaling up functions on elasticity-demanding slices before horizontal elasticity happens (i.e., on deriving resource-depleting condition). The Solidarity Elasticity aims to overcome the gap of mainstream elasticity solutions (e.g., Kubernetes), whereby leverage stochastic-based solutions (far inefficient in computing adding resources by resulting in undesired over-provisioning), and either trigger time-consuming vertical elasticity or reject elasticity when detecting resource-depleting conditions.

The Solidarity Elasticity solution departs from the concepts of Recipients and Donors slices. Slices facing bottlenecks which might have an impact on SLAs and, as such, are eligible to receive more resources (scale-up) from the infrastructure providers, are denoted to as Recipients. Such resources designed to scale up slices might be obtained from either, a part of the additional resources from the infrastructure providers, or resources already in use by other activated slices. The latter are denoted to as Donors, meaning a set of slices in operation that for any reason have more resources committed to them than their current occupation rate. An efficient elasticity mechanism should always provide over-provisioned resources for slices, although these resources cannot be either high or low. In situations where, over-provisioned resources are high, there might be an undesirable degree of expenditure. Conversely, in circumstances where over-provisioned resources are low, the elasticity mechanism may be triggered several times. Therefore, assuming that, under normal conditions there are always over-provisioned resources for slices in resource saturation situations, the Solidarity Elasticity is capable of calculating new resources patterns for allocating from Donors to the Recipients slices.

Operations of the Solidarity Elasticity Solution

While being committed to SLA-guaranteed, a Maximum Threshold (the maximum amount of resources that can be assigned for every active slice -- *Mrth*) and a Committed Threshold (the minimum amount of resources ensured overtime for every active slice -- *Cth*) are set to every slice, in the sense to avoid starvation. In what concerns the reasoning of action sets, the Solidarity Elasticity solution statistically calculates fractions of residual resources (meaning current resource usage rates between the Maximum and Committed thresholds per slice) to shrink the size of the selected Donor slices, so that it is possible to scale up a given recipient slice with the addition of the donated resource fractions. The primary purpose of the statistical resource-compute scheme featuring the Solidarity Elasticity solution is to enable the residual resources of all the listed Donor slices to be shrunk in proportion to their resource use ratio (instead of being based on stochastic-based models, that apply predefined static rates to shrink resources until they match the required amount). Thus, the Solidarity Elasticity solution is able to ensure an SLA commitment in a per Donor slice level, as well as allowing a scheme that is far more efficient than stochastic-based models.

```
1:  while True do
2:      monitoring(slices) #monitoring action
3:      for slice in slices do
4:          if resource_utilization(slice) is critical then
5:              select_resource(slice)
6:              if check_available_resources = True then
7:                  scale_up(slice)
8:              else
9:                  #reasoning action
10:                 resource_discovery(donor_slices_list)
11:                 computes_discovery(donor_slices_list)
12:                 obtains_percentage(donor_slices)    #calculate new amount for each donor slice
13:                 scale_down(donor_slices) #enforcing
14:                 aggregate_resources(donor_slices)    #calculate new amount for recipient slice
15:                 scale_up(recipient_slice) #enforcing
16:             end if
17:         end if
18:     end for
19: end while
```

Algorithm 1: Vertical elasticity mechanism.

The Solidarity Elasticity is invoked whenever the NECOS SRO component notices a slice part is facing critical condition in its residual resource, i.e., when the current usage rate matches a predefined threshold (e.g., 80%) of the assigned amount. The Solidarity Elasticity approach encompasses monitoring, reasoning and enforcement action sets, as shown in Algorithm 1. The Monitoring action set incorporates functions for collecting metrics from the infrastructure providers by composing a slice and delivering it to the elasticity module. The Enforcement action set, in turn, entails operations that involves enforcement of new resource definitions for the Recipient and the Donor slices, at the end of the elasticity control algorithm. The Reasoning action set takes place in between the Monitoring and Enforcement actions and plays a crucial role in the elasticity control solidarity solution by calculating the new resource definitions for the Recipient and the Donor slices based on a statistical model. As a means of providing knowledge support, a state table structured by potential donor slices is maintained (as background information from a performance optimization perspective), in which entries are ordered according to their levels of resource usage.

On the basis of the definitions enclosing the Algorithm 1, a non-exhaustive description of the vertical elasticity functions driven by the Solidarity Elasticity solution is provided in the following. From the beginning, the algorithm keeps monitoring every activated cloud network slice instances (line 2). On deriving that a given cloud network slice instance achieves critical resource utilization condition (line 3), SRO invokes the vertical elasticity function to deal with this Recipient slice demanding elasticity. Firstly, the algorithm searches potential Donor slices, and then it selects a set (among all searched) that will donate resources (lines 5-6). After achieving this, it calculates the percentage donating ratio for each selected Donor slice (line 7), multiplies the respective amounts with each of the selected Donor slice residual resources and scales down all them accordingly (lines 8-9-10). In the end, the algorithm sums all the donating rates (line 14) and scales up the Recipient slice (lines 11-12).

The mathematical model of the Solidarity Elasticity solution encompasses a set of equations. The Equation 1 is idealized to calculate the maximum number of donating portions of residual resources (in terms of CPU, RAM, Bandwidth, according with elasticity demands) associated to the selected Donor slices.

$$Trr = \sum_{k=1}^{n} MRth(k) - Cth(k)$$ Eq. 1.

Where:

- *Trr* (Total Residual Resources) – The sum of residual donating resources that SLOTS calculate on all selected Donor slices;
- *MRth(k)* – Maximum Reservation Threshold for Donor slice(k);
- *Cth(k)* – Committed Threshold for Donor slice(k);
- *n* – Number of Donor slices.

On the basis of the Equation 1, Equation 2 addresses to calculate the percentage of residual resources (*Ru*) for each selected Donor slice, being a composition between resources available and the committed rates in function to the current resource amount.

$$Ru = \sum_{k=1}^{n} \frac{(MRth(k) - Cth(k)) * 100}{Trr}$$ Eq. 2

Where:

- *Ru* (Resource in use) – represents the amount of resources in use in each donor slice.

Lastly, the Equation 3 calculates the amount of donating resources that each selected Donor slice will effectively donate. Equation 3 is performed for each slice in a range between *k* and *n*, where n is the total number of selected Donor slices.

$$Mrth(k) = (MRth(k) - Cth(k)) - \frac{((Trr - Nr) * Ru(k))}{100}$$ Eq. 3

Where:

- *Mrth(r)* – the amount that will be donated to the Recipient slice;
- *Nr* (Necessary Resources): (Necessary Resources) – resource required by a given recipient slice.

Next section presents another NECOS subsystem, namely, the IMA subsystem.

2.2.4. Infrastructure & Monitoring Abstraction

The Infrastructure & Monitoring Abstraction (IMA) component of the NECOS architecture was designed to provide a uniform abstraction layer above the heterogeneous VIM / WIM and monitoring subsystems that are part of an end-to-end Slice. As different Slice Parts constitute an end-to-end Slice, and each part can potentially rely on a different technology (e.g., specific VIM / WIM and monitoring subsystem implementations), IMA plays the important role of providing a common abstract interface at its northbound, which allows the SRO to perform its functions without taking care of the implementation details of each Slice Part.

This section describes the implementation of the two main parts of IMA, namely the *Resource and VM Monitoring* and the *Resource and VM Management.* The former is required to collect information about the status of the resources substrate on which the end-to-end slices are deployed. This is then provided as feedback to the Slicing Orchestrator's components (e.g., the Slice Resource Orchestrator) in order to perform the life-cycle of the different slices, enforcing elasticity rules, etc. The *Resource and VM Management* is instead responsible for providing the abstract API to be used by the SRO to interact with the VIM / WIM inside the different Slice parts to perform both resource operations and VM management operations. This is mainly required to perform the life-cycle management of the different service elements that are part of the Slice.

2.2.4.1. Implementation of IMA Resource and VM Monitoring Components

This section presents a detailed description of the implementation of the monitoring system that was designed within the NECOS project to support the concept of Slice-as-a-Service. The main scope of the developed solution was to allow the collection of multiple metrics (in the form of monitoring measurements) from different end-to-end Slices deployed across a federated, multi-cloud environment scenario. This is of paramount importance for the SRO to perform end-to-end slice orchestration duties, which include slices reconfiguration, elasticity, etc.

The monitoring solution described in this section was implemented according to the architectural design foreseen for the *IMA Resource & VM Monitoring component* as highlighted by Figure 14.

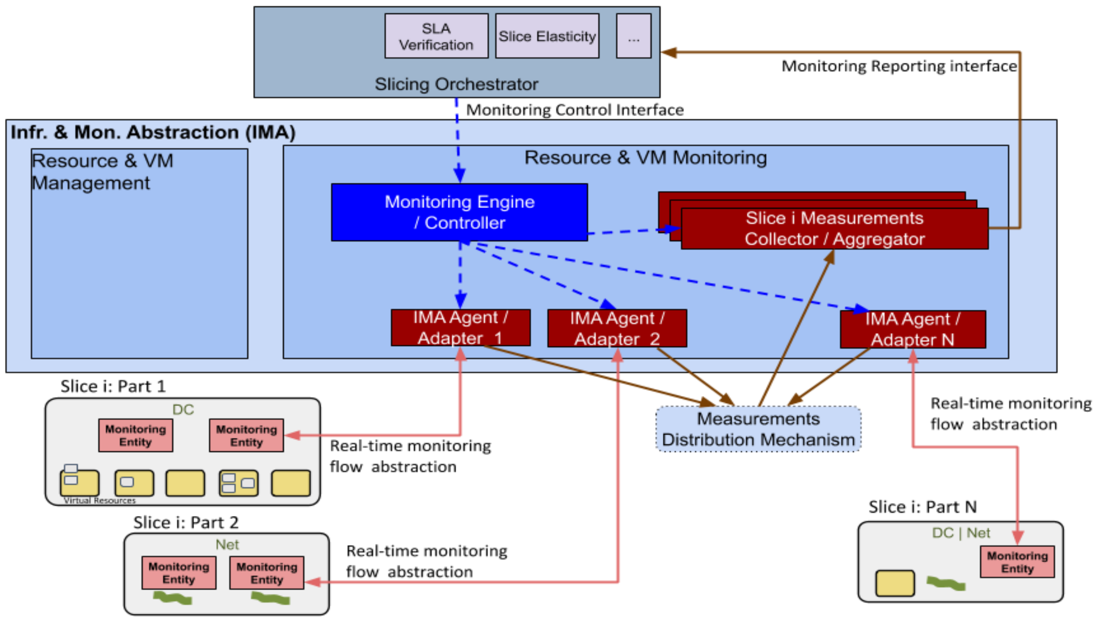


Figure 14: IMA Resource & VM Monitoring Components.

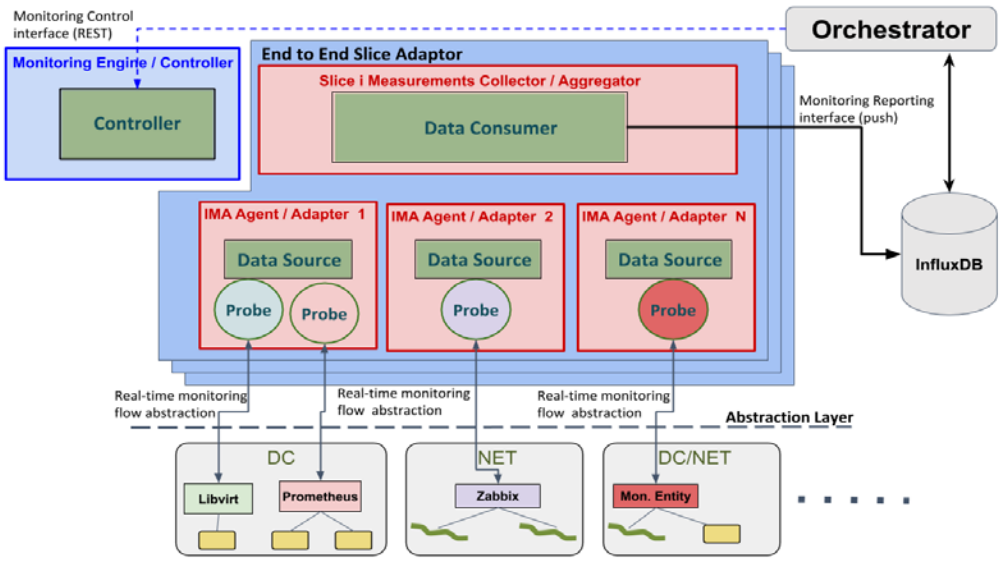

Monitoring Control interface (REST)
Orchestrator
Monitoring Engine / Controller
Controller
End to End Slice Adaptor
Slice i Measurements Collector / Aggregator
Data Consumer
Monitoring Reporting interface (push)
InfluxDB
IMA Agent / Adapter 1
IMA Agent / Adapter 2
IMA Agent / Adapter N
Data Source
Data Source
Data Source
Probe
Probe
Probe
Probe
Real-time monitoring flow abstraction
Real-time monitoring flow abstraction
Real-time monitoring flow abstraction
Abstraction Layer
DC
Libvirt
Prometheus
NET
Zabbix
DC/NET
Mon. Entity

Figure 15: Component mapping to Architecture.

Figure 15 shows how each component of the proposed monitoring solution is mapped to a corresponding component of the IMA architecture shown in Figure 14. The proposed software modules are based on an extended version of the ones provided by the Lattice Monitoring Framework [TUSA 2019]. More specifically, the *Controller* implements the *IMA Monitoring Engine / Controller*, the *Data Consumer* corresponds to the *IMA Measurements Collector / Aggregator,* and the *Data Sources* support the functionalities of the IMA Agents / Adapters. Together with the expected IMA functional components, the proposed monitoring approach provides an implementation of the IMA interfaces represented in the same figure, i.e., *Monitoring Control* and *Monitoring Reporting* to allow the full interaction of the monitoring system with the other NECOS components, such as the SRO. In particular, the *Monitoring Control* interface is based on a RESTful implementation, whereas the *Monitoring Reporting* interface relies on an intermediate time-series DB (InfluxDB) that enables the push / pull of measurements between the monitoring system and the SRO.

Figure 16 depicts a close-up view of the implemented monitoring solution that was already shown in Figure 14. From an implementation perspective, these two figures represent the same concept. However, for the reader's convenience, Figure 16 highlights the implemented software blocks instead of their mapping on the IMA architecture. All the software entities depicted on Figure 16 will further be detailed in the remainder of this section. In Figure 16 (and also in Figure 15) the reader will see that a Data Consumer plus the set of Data Sources and Probes associated to different Slice Parts of an end-to-end Slice have been grouped under an umbrella entity named *End-to-End Slice Adapter*. Please note that this should only be considered as an additional aggregated logical view of those components rather than an actual new software module to be mapped on the IMA architecture of Figure 14.

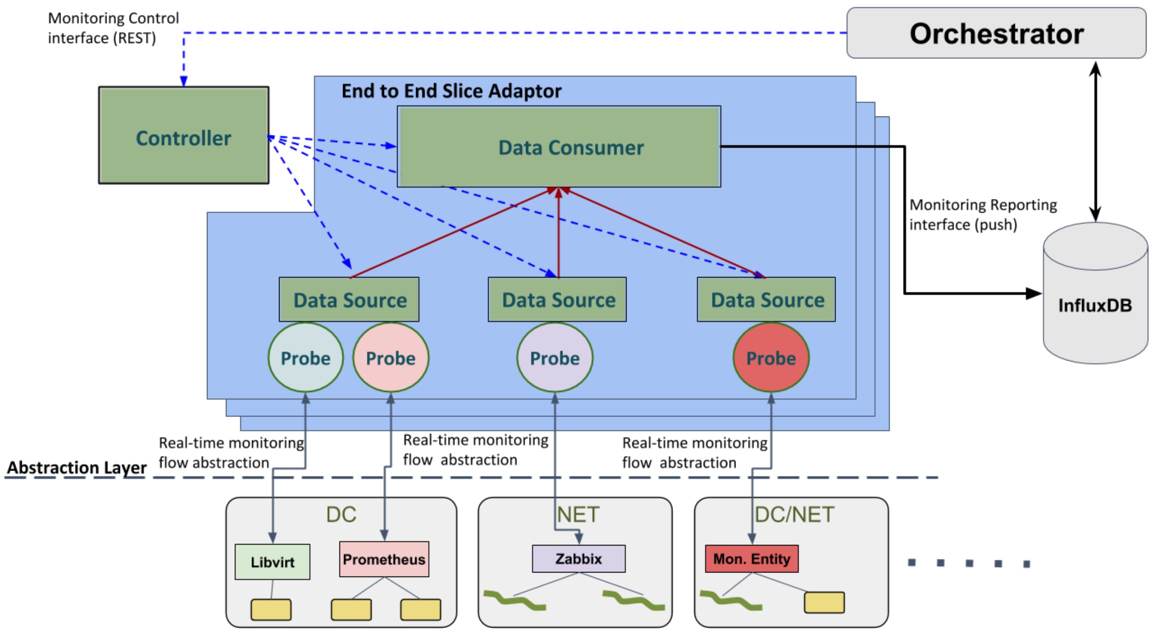

Monitoring Control interface (REST)
Orchestrator
End to End Slice Adaptor
Controller
Data Consumer
Monitoring Reporting interface (push)
InfluxDB
Data Source
Data Source
Data Source
Probe
Probe
Probe
Probe
Real-time monitoring flow abstraction
Real-time monitoring flow abstraction
Real-time monitoring flow abstraction
Abstraction Layer
DC
NET
DC/NET
Libvirt
Prometheus
Zabbix
Mon. Entity

Figure 16: IMA Resource & VM Monitoring Components.

Controller

This is the component that implements the functionalities of the IMA Monitoring Engine / Controller, i.e., it takes care of starting, configuring and controlling relevant IMA Agents / Adaptors in the Monitoring Adaptation Layer, in order to hide the implementation details of a specific monitoring subsystem deployed in a Slice Part. A given IMA Agent / Adapter is in fact implemented by a Data Source instance plus a technology-specific implementation of a Probe. The latter gathers measurements from that Slice Part and forwards them to a Data Consumer (also instantiated on-demand by the Controller) that acts as measurements aggregator for a whole end-to-end slice.

The interaction between the Orchestrator and the Controller is based on a REST API invocation that receives a YAML descriptor as parameter, e.g. (Figure 17 below).

EUB-01-2017

Request:
POST --data-binary *start_file.yaml* -H "Content-type: text/x-yaml"
[http://*address*:*port*/necos/ima/start_monitoring](http://address:port/necos/ima/start_monitoring)

Response:

```
slice:
  id: IoTService_sliced
  name: IoTService_sliced
  short-name: IoTService_sliced
  description: Slicing for Elastic IoT Service
  vendor: dojot
  version: '1.0'
  slice-timeframe:
    service-start-time: {100919: 10 pm}
    service-stop-time: {101019: 10 pm}
  slice-parts:
    - dc-slice-part:
      name: dc-slice1
      dc-slice-part-id:
        slice-controller-id: 2
        slice-part-uuid: 1
      type: DC
      location: Brazil
      monitoring-parameters:
        tool: netdata
        measurements-db-ip: <IP>
        measurements-db-port: <PORT>
        granularity-secs: 10
        type: host
        metrics:
          - metric:
            name: PERCENT_CPU_UTILIZATION
          - metric:
            name: MEGABYTES_MEMORY_UTILIZATION
          - metric:
            name: TOTAL_BYTES_DISK_IN
          - metric:
            name: TOTAL_BYTES_DISK_OUT
          - metric:
            name: TOTAL_BYTES_NET_RX
          - metric:
            name: TOTAL_BYTES_NET_TX
    - dc-slice-part:
      name: dc-slice2
      dc-slice-part-id:
        slice-controller-id: 5
        slice-part-uuid: 1
      type: EDGE
      location: Europe
      monitoring-parameters:
        tool: prometheus
        measurements-db-ip: <IP>
```

```
measurements-db-port: <PORT>
granularity-secs: 10
type: all
metrics:
  - metric:
    name: PERCENT_CPU_UTILIZATION
  - metric:
    name: MEGABYTES_MEMORY_UTILIZATION
  - metric:
    name: TOTAL_BYTES_DISK_IN
  - metric:
    name: TOTAL_BYTES_DISK_OUT
  - metric:
    name: TOTAL_BYTES_NET_RX
  - metric:
    name: TOTAL_BYTES_NET_TX
```

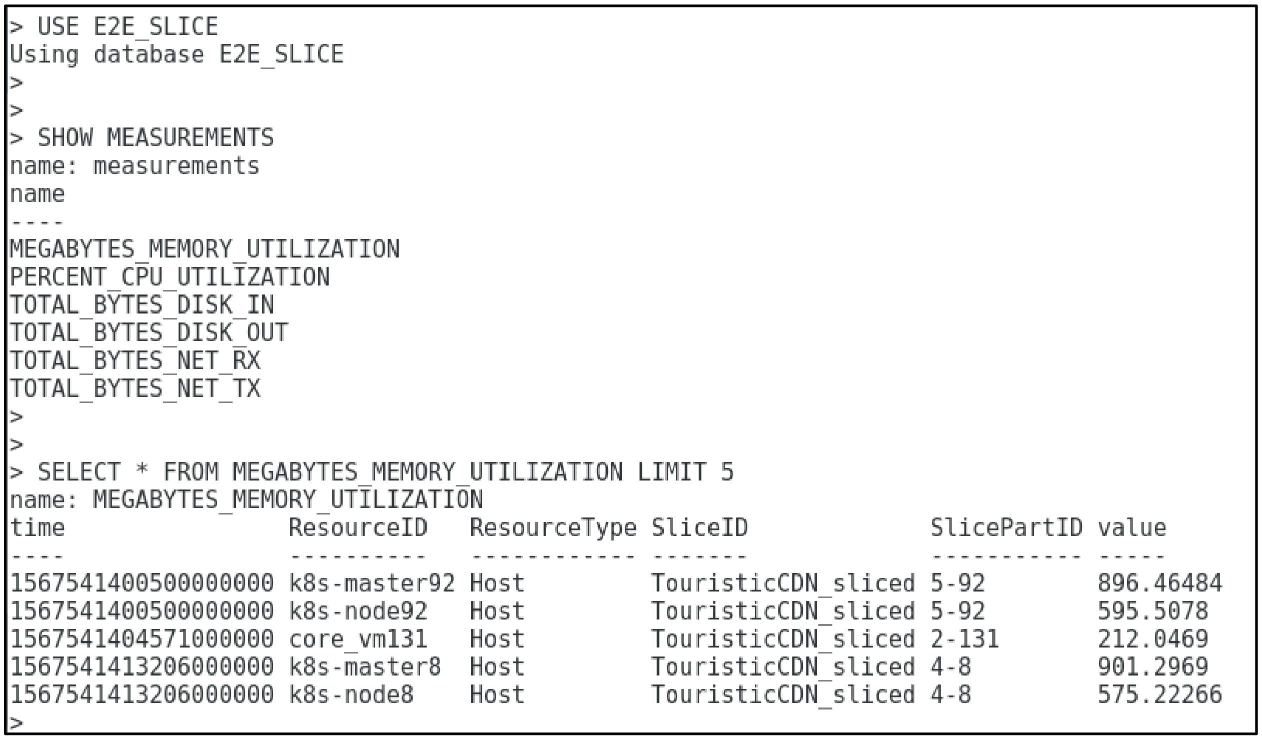
Figure 17: YAML descriptor - Orchestrator REST API invocation.

When a new end-to-end slice is allocated, the SRO can use the above REST call to trigger the instantiation of both a measurements aggregator (i.e., a Data Consumer) and of the monitoring adapters that will be required for each Slice Part (i.e., a bespoke combination of different Data Sources and Probes). The particular type of adapters to be instantiated will be determined by parsing the elements of the YAML descriptor that specify the type of monitoring technology to be abstracted and the monitoring endpoint associated to each Slice Part.

Data Consumer

This is the component instantiated in order to aggregate / multiplex the monitoring information coming from different Parts of an end-to-end Slice. It receives the measurements collected by the Probes that are attached to each Data Source, aggregates them to build an end-to-end Slice monitoring view, and finally pushes that aggregated monitoring information to the SRO via the Monitoring Reporting Interface. The latter has specifically been implemented via a time-series DB that ensures a loosely-coupled interaction between the different Data Consumers and the SRO (respectively generating and consuming those measurements).

An example of aggregated measurement associated to an end-to-end Slice is reported below in Figure 18:



```
> USE E2E_SLICE
Using database E2E_SLICE
>
>
> SHOW MEASUREMENTS
name: measurements
name
----
MEGABYTES_MEMORY_UTILIZATION
PERCENT_CPU_UTILIZATION
TOTAL_BYTES_DISK_IN
TOTAL_BYTES_DISK_OUT
TOTAL_BYTES_NET_RX
TOTAL_BYTES_NET_TX
>
>
> SELECT * FROM MEGABYTES_MEMORY_UTILIZATION LIMIT 5
name: MEGABYTES_MEMORY_UTILIZATION
time                ResourceID   ResourceType SliceID             SlicePartID value
----                ----------   ------------ -------             ----------- -----
1567541400500000000 k8s-master92 Host         TouristicCDN_sliced 5-92        896.46484
1567541400500000000 k8s-node92   Host         TouristicCDN_sliced 5-92        595.5078
1567541404571000000 core_vm131   Host         TouristicCDN_sliced 2-131       212.0469
1567541413206000000 k8s-master8  Host         TouristicCDN_sliced 4-8         901.2969
1567541413206000000 k8s-node8    Host         TouristicCDN_sliced 4-8         575.22266
>
```

Figure 18: Aggregated measurement example of an end-to-end slice monitoring.

Data Source

Together with the Probes (described later on in this section), the Data Sources are the components that implement the functionalities of the IMA Monitoring Adaptation Layer.

A Data Source is the entity able to manage the instantiation of one or more Probes and to also exert a specific level of control on them. Moreover, each Data Source multiplexes the measurements collected by the Probes and sends them to the Data Consumers via a specific distribution mechanism. The current implementation of the distribution mechanism is based on measurements encoded using the XDR protocol and transmitted to the Data Consumers via ZeroMQ sockets. Each Data Source is configured at instantiation time to report measurements to a specific Data Consumer (i.e., the one that aggregates the monitoring information associated to that specific end-to-end Slice).

Probe

A Probe is dynamically instantiated on a Data Source by the Controller to provide the monitoring adaptation / abstraction mechanisms required for a Slice Part. This is achieved by configuring the probe to interact with the particular VIM / WIM and / or the monitoring entry points that were allocated for that Slice Part. The interaction with the above entry points will in fact depend on the particular technology used for a Slice Part (see Real-time Monitoring Flow Abstraction Interface in Figure 16).

The collected measurements will include resource facing KPIs (e.g., CPU, Memory, Storage, delay, bandwidth, network I/O) of both the physical resource and the virtual service elements running on that VIM / WIM. Regardless of the particular monitoring technology of a Slice Part, the measurements will be provided to the Data Source (to which the Probe is attached) in a common format and will then be sent to the relevant Data Consumers for further aggregation / processing. Please note that the collection of service specific KPIs is not in the scope of the IMA and is purposely left to the Tenant.

Interfaces with the SRO

The monitoring system described earlier in this section allows the SRO to trigger the dynamic instantiation of a set of monitoring components (i.e., Data Consumers, Data Sources and Probes) that can be utilised to deploy the abstraction mechanisms required for collecting measurements from an end-to-end Slice. This is performed on-the-fly when a new Slice is successfully created via the REST API that implements the functionalities of the *Monitoring Control Interface.* Currently, this interface supports the following API endpoints:

```
POST --data-binary start_file.yaml -H "Content-type: text/x-yaml"
http://localhost:4567/necos/ima/start_monitoring

POST --data-binary update_file.yaml -H "Content-type: text/x-yaml"
http://localhost:4567/necos/ima/update_monitoring

POST --data-binary delete_file.yaml -H "Content-type: text/x-yaml"
http://localhost:4567/necos/ima/delete_monitoring
```

The above endpoints are exposed by a wrapper component that was created on top of the existing Lattice Controller in order to simplify the interaction between this latter and the SRO. The Start Monitoring call in fact includes a set of finer grained operations that are performed by the Controller in order to instantiate the required Data Consumers, Data Sources and Probes.

As soon as the end-to-end Slice monitoring abstraction is in place, each Data Consumer will start sending the aggregated measurements to the SRO. As we anticipated earlier this happens via the *Monitoring Reporting Interface,* which was implemented via using a time-series DB as intermediate component for the exchange of monitoring metrics between the monitoring system and the SRO. More specifically, *Lattice Reporters* attached to the Lattice Data Consumer are created and configured to push time-series measurements to an InfluxDB instance from which the SRO can pull via REST. For each end-to-end Slice, a bespoke, isolated adaptation layer is created to collect a push measurements to a per-tenant time-series DB. This choice was made to ensure the proper degree of isolation between different slices and tenants.

Figure 19 shows some timings related to the *Start Monitoring* and *Delete Monitoring* operations when end-to-end Slices consisting of a variable number of resources were considered. More specifically, multiple tests were conducted while creating end-to-end Slices consisting of *1* Slice Part (1 resource), *2* Slice Parts (2 resources) and *3* Slice Parts (3, 6, 8, and 12 resources). The graph shows how both the overall number of Slice Parts and the number of Slice Resources (physical servers in this specific case) do not affect the deployment and the decommission of the required monitoring abstractions. This is due to the intrinsic parallel nature of the system: as soon as a monitoring instantiation request for multiple Slice Parts is received, the required monitoring abstraction entities can be allocated in parallel in order to speed-up the process. Furthermore, since the monitoring abstraction interacts with an already existing asset (either a VIM or a monitoring system), the time required for its instantiation / deletion is not a function of the number of resource elements of the end-to-end Slice.

Values on the y-axis represent average measurements with 90% confidence interval (the standard deviation for each experiment turned out to be very small, hence the very small intervals on the graph).

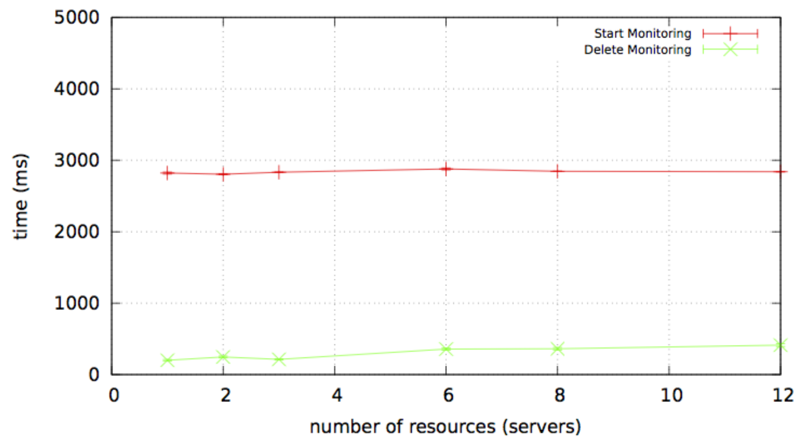


Figure 19: Evaluation of the IMA Resource and VM Monitoring Performance.

2.2.4.2. Implementation of IMA Resource and VM Management Components

This section presents a detailed description of the implementation of the management system that was designed within the NECOS project to support the concept of Slice-as-a-Service. The main scope of the developed solution was to allow the management of multiple VIM/WIMs from different end-to-end Slices deployed across a federated, multi-cloud environment scenario which include mainly the deployment and reconfiguration of the services.

The management solution described in this section was implemented according to the architectural design foreseen for the *IMA Resource & VM Management component* as highlighted by Figure 20.

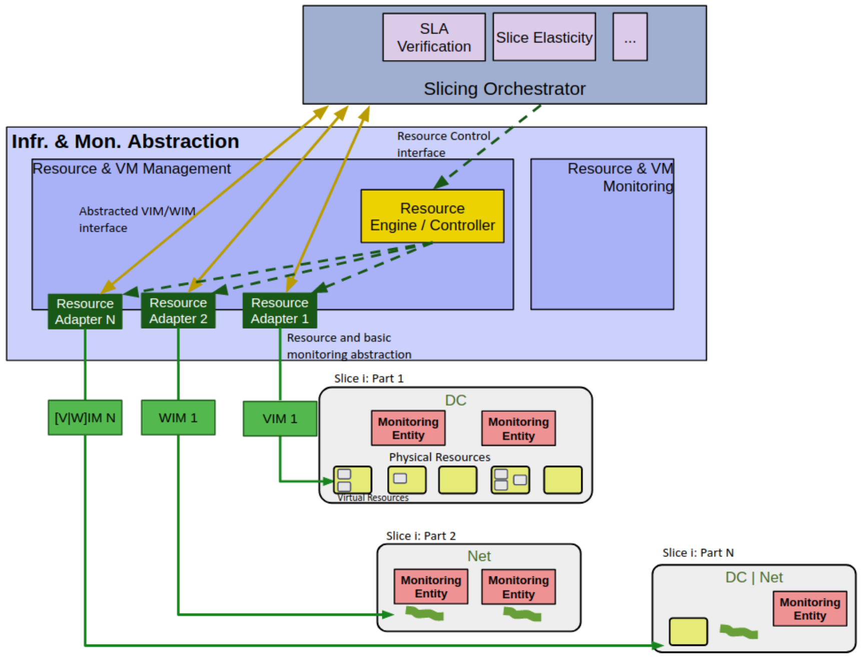


Figure 20: IMA Resource & VM Management Components.

EUB-01-2017

Resource Engine Controller

Management Controller functionalities are similar to those for Monitoring Controller, it takes care of starting, configuring and controlling relevant IMA Agents / Adaptors in the Management Adaptation Layer, in order to hide the implementation details of a specific VIM / WIMs deployed in a Slice Part. The SRO interacts with the Resource Engine Controller via the *Resource Control Interface* and provides in addition to the aforementioned methods, methods referring to each VIM / WIMs present in the slice via the *Abstracted VIM/WIM Interface*.

One example of SRO to Management Controller HTTP Request is:

```
POST --data-binary start_file.yaml -H "Content-type: text/x-yaml"
http://localhost:5001/necos/ima/start_management
```

Actually, the Management Controller has four main methods that enable management operations, namely:

start_management():
This method is triggered by the SRO component in the Slice Creation Workflow and is responsible for instantiating the specific Adapters for each VIM / WIM present in the Slice. An example of YAML file can be seen in https://gitlab.com/necos/demos/integrated-demo/blob/master/yamls/startManagement.yaml.

stop_management():
This method is triggered by the SRO component in the Slice Decommission Workflow and is responsible for removing all the Resource Adapters instantiated for a specific Slice. The parameter of this method is the <Slice ID>.

update_management():
This method is triggered by the SRO component in the Slice Elasticity Workflow Upgrade / Downgrade and is responsible for instantiating/removing Resource Adapters on-demand for the new / removed Slice Part (s). An example of YAML file can be seen in https://gitlab.com/necos/demos/integrated-demo/blob/master/yamls/updateManagement.yaml.

deploy_service():
This method is triggered by the SRO component in the Deploy Service Workflow and is responsible for instantiating the service requested by the Tenant. The parameter for this method changes according to the VIM / WIMs utilized.

Resource Adapters

Specific adaptors for particular VIMs and WIMs are required to support multiple technologies usually providing different levels of abstraction and a variety of supported features. The VIM and WIM Adaptors are specific wrappers for different VIM/WIM implementations, providing basic service platform agnostic wrapping to common NECOS functions. In practice, the adaptors should translate generic calls into specific commands or methods in the context of specific VIMs and WIMs.

Typical VIM and WIM adaptors include:

• Cloud adaptors: OpenStack, Heat, Kubernetes, VLSP;
• VNF adaptors: OpenMano, Open Baton, OPNFV;
• Transport adaptors: SDN controllers such as Opendaylight, Floodlight;
• RAN adaptors;
• Edge adaptors.

Currently, two resource adaptors have been developed, one for Kubernetes and one capable of abstracting SSH access to the physical machines instantiated in the slice. The Kubernetes Adaptor supports most of the functionalities provides by the Kubernetes REST API, such as deploy service, get pods, list services, get services, stop service, etc. The SSH Adaptor abstracts the execution of commands remotely on physical machines that belongs to a specific slice.

The principal method of the *Resource Adaptor* is the *deploy_service()*. Therefore, as mentioned earlier the parameter for the *deploy_service()* method changes according to the VIM / WIMs used by each slice part. Examples of deploy_service() parameters for the Kubernetes Adapter and SSH Adapter can be seen in https://gitlab.com/necos/demos/integrated-demo/blob/master/yamls/deployServiceK8s.yaml and https://gitlab.com/necos/demos/integrated-demo/blob/master/yamls/deployServiceSSH.yaml, respectively.

2.2.5. Service Orchestrator Adaptor

The **Service Orchestrator Adaptor** is the component that interacts with the tenant's Service Orchestrator in order to deploy the service over the created slices. As discussed in Section 2.1.1, in our prototype implementation, the Tenant requests a slice creation passing the Slice Specification or the Service Specification. After the slice is created, the tenant receives a YAML file detailing the slice created and containing some entry points to access the slice parts. Later, the tenant will use this information to make decisions regarding the service embedding. More specifically, based on the information received, the tenant can specify in which slice part and which commands to use to deploy the service components. This is done by having the tenant's Service Orchestrator sending a YAML file to the **Service Orchestrator Adaptor**, using the method **deploy_service()**. Upon receiving such a call, the Service Orchestrator Adaptor forwards the call to the SRO, so that it can handle the service deployment.

As defined in D3.1, the Service Orchestrator Adaptor provides the features to adapt the calls from the tenant's Service Orchestrator into calls internal to the Slicing Orchestrator. The Service Orchestrator is part of the service deployment workflow defined in D5.2 as part of the slice creation process. In our implementation, the Service Orchestrator is a simple component and does not have any internal details or specific engine. It is basically a forwarding module which receives the call from the tenant and dispatch it to the SRO.

2.3. Resource Providers

2.3.1. DC Slice Controller

In NECOS architecture, for each data center, a **DC Slice Controller** is in charge of creating Data Center (DC) slices within the data center, allocating the required compute and storage resources for a given slice part, and returning a handle to a VIM running on it. The VIM can either be a separate instance deployed on demand based on the tenant specification or an existing VIM running in that resource domain on which an isolated interaction point (such as a shim) was created for the tenant.

The **DC Slice Controller** is responsible for managing a pool of DC resources, such as compute and storage resources that must be allocated to participate in the slicing process. It also handles requests for DC slices and determines if it is possible to create a new DC slice in the data center based on local resource availability. If the DC slice creation is possible, it will select resources from the pool that should be allocated to the slice.

As described in Deliverable 3.2, in NECOS there are two slicing approaches: Mode-0 and Mode-1. In Mode-0, for each DC slice there is an on-demand VIM and the **DC Slice Controller** is responsible for allocating and deploying the VIM (of a particular type) to the DC slice, as well as configuring the VIM to use the resources, which have been picked for the slice. In Mode 1, for each data center, there is an existing VIM running in that resource domain. In this case, the **DC Slice controller** is responsible for creating an isolated interaction point (such as a shim) for the tenant.

In our prototype, the **DC Slice Controller** is focused on the Mode-0 approach and on virtualized resources, so that, a slice part can be seen as a set of virtual hosts (VMs) connected through a virtual switch. This set of virtual hosts forms a cluster over which a VIM can be deployed on demand to manage the entire cluster, as shown in Figure 21. In addition, clusters of virtual resources representing different slice parts (in the same data center) are completely isolated from each other.

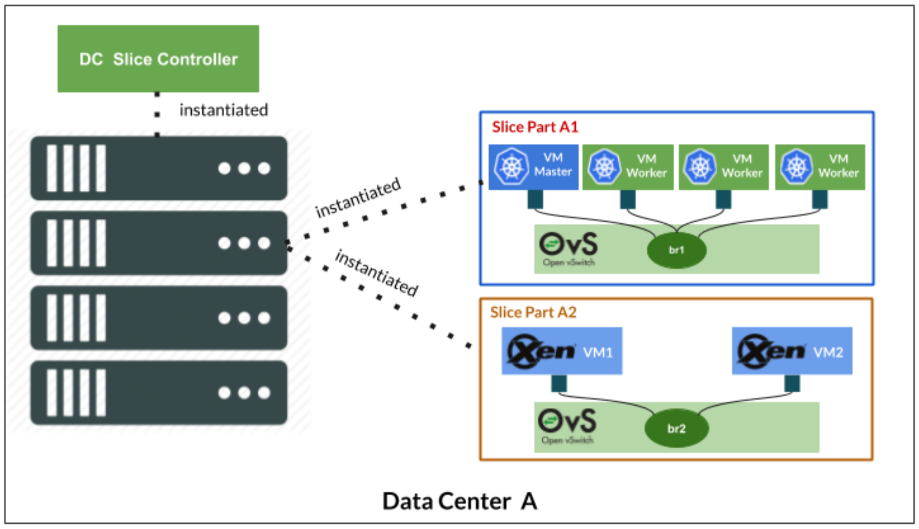


Figure 21: Implementation of Slice Parts in a data center.

In order to deliver such slice parts, the DC Slice controller consists of the following functions, as depicted in Figure 22:

- **Slice Part Manager**: This is the main **DC Slice Controller** function responsible for initiating the slice part creation processes requested by the **Slice Builder**. Upon receiving the **request_slice_part()** call along with the Slice Part Specification (see Figure 23), the **Slice Part Manager** checks the resource and VM template availability forwarding the request to the **Host Manager** and **VM Factory** functions, respectively. Both the resources **(epa-attributes)** and the VM templates (**vdu-image)** are specified in the Slice Part Specification received from the **Slice Builder**. If the request for the slice part creation can be satisfied, the **Slice Part Manager** reserves the required resources and creates the slice part entry in its local database (**Slice part, and Host Database**), returning the Slice Part ID of the allocated slice part to the **Slice Builder**. Later, when the **activate_slice_part()** method is invoked by the **Slice Builder**, **The Slice Part Manager** interacts with the **VM Factory** and the **Network Manager** functions to actually instantiate the cluster of virtual hosts, setup the virtual switch and data plane for the slice part and deploy and configure the VIM specified in the Slice Part Specification (**VIM** section) over the created cluster. When the cluster is completed setup, the **Slice Part Manager** deploys the monitoring system (specified in the **monitoring-parameters** section of the Slice Part Specification) and creates the entry points to access the cluster and the monitoring system. This entry points are returned back to the **Slice Builder** (see Figure 24)**.**

- **Host Manager**: This function is responsible for keeping an inventory of all of the resources in the data center as well as their states.
- **VM Factory**: The role of this function is to instantiate the cluster of virtual hosts specified in the Slice Part Specification. Given the **vdus** listed in the Slice Part Specification, the **VM Factory** checks if the VM templates for the vdus are available in the **VM Database**. If the templates can be found, the **VM Factory** instantiates the required virtual hosts (VMs) from the templates using as physical hosts the resources allocated by the **Slice Part Manager** for the slice part. A VM template is a VM image where a VIM of a particular type (e.g., XenServer or Kubernetes) is partially installed. After instantiating the cluster of virtual hosts from the templates, the **VM Factory** configures the VIM according to its type. There is a set of configurations to be performed for each type of VIM supported by the **DC Slice Controller**.
- **Network Manager:** This function is responsible for setting up and configuring the virtual network that connects the virtual hosts of the slice part and that isolates them from other virtual hosts in other slice parts.

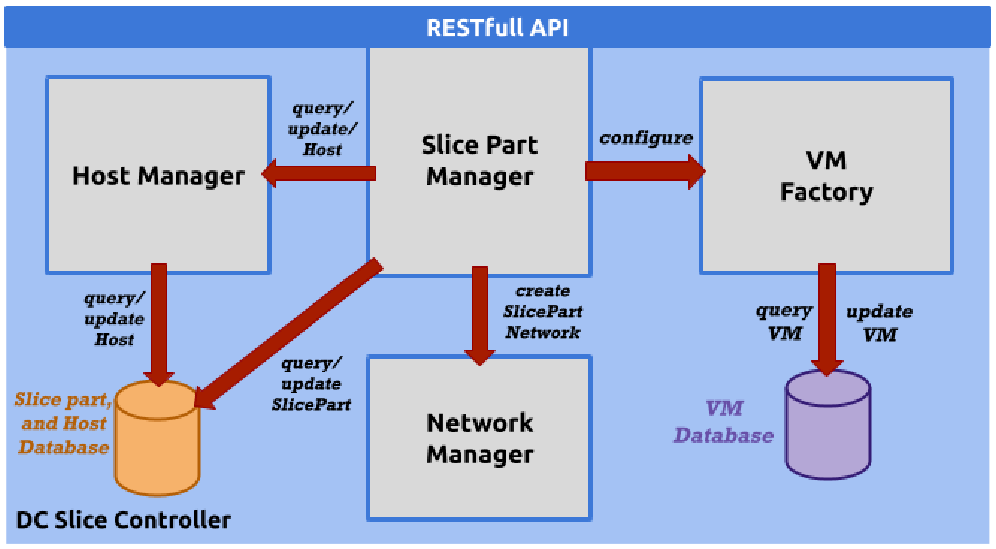


Figure 22: DC Slice Controller internal functions.

Figure 23 illustrates the YAML file that the **Slice Builder** sends to the DC Slice Controller (in the **UNICAMP** domain) when requesting the creation of the slice part **dc-slice1**. When the slice part is activated, the **DC Slice Controller** returns back to the Slice Builder the YAML file described in Figure 24.

```
slices:
  sliced:
    id: IoTService_sliced
    ...
    slice-requirements:
    ...
    slice-part:
      - dc-slice-part:
        name: dc-slice1
        monitoring-parameters:
          tool: netdata
          ...
        VIM:
           name: KUBERNETES
           vim-shared: false
           vdus:
              - vdu:
                 hosting: SHARED
                 vdu-image: 'k8s-dojot-master-template'
                 epa-attributes:
                 ...
                 ip: 10.10.2.1
              - vdu:
                 hosting: SHARED
                 vdu-image: 'k8s-dojot-worker-template'
                 epa-attributes:
                 ...
                 ip: 10.10.2.2
              - vdu:
                 hosting: SHARED
                 vdu-image: 'k8s-dojot-worker-template'
                 epa-attributes:
                 ...
                 ip: 10.10.2.3
```

Figure 23: Slice Part Specification sent from Slice Builder to a DC Slice Controller.

```
slice-part-id:
      slice-controller-id: 2
      uuid: 1
monitoring-handle:
      ip: 143.106.11.131
      port: 19266
vim-handle:
      ip: 143.106.11.131
      port: 21266
ssh-handle:
      ip: 143.106.11.131
      port: 22266
vim-credential:
      ….
vswitch:
      bridge-name: br_2_1
      type: openvswitch
```

Figure 24: Entry points returned by the DC Slice Controller.

The DC Slice Controller uses the libvirt toolkit to manage the XEN virtualization platform and the LVM system to create the virtual hosts. The Slice Part and Host Database and the VM Databases are implemented using MySQL. The connectivity inside the slice part is handled by the Open vSwitch (OVS) tool, which instantiates an OVS bridge for each slice part. The entry points are created using port forwarding with iptables.

2.3.2. WAN Slice Controller

The **WAN Slice Controller** is the component that resides inside each Network Provider and that dynamically creates a Network slice, as a part of a full cloud network slice. A Network slice is a set of virtual links that connects two DC slices. In order to create a Network slice, the **WAN Slice Controller** manages all of the network resources in the network provider domain that are allocated to participate in slicing and keeps track of which network resources have already been allocated to which slice.

The **WAN Slice Controller** handles requests for Network slices and determines if it is possible to create a new network slice in the network domain based on local resource availability (e.g., bandwidth). If the Network slice creation is possible, the **WAN Slice Controller** provides the set of virtual links required to connect the given DC slice parts.

In Mode-0, for each Network slice there is an on-demand WIM allocation. In this case, the **WAN Slice Controller** deploys a WIM to the Network slice and configures it to use the network resources, which have been assigned for the slice. In Mode-1, for each network domain, there is an existing common WIM running in that domain that is shared by multiple slices. Thus, the **WAN Slice Controller** is responsible for creating an isolated interaction point for each tenant.
Figure 25 illustrates the **WAN Slice Controller** we have implemented in our prototype. It is composed by two main functions:

- **WAN Master**: This function resides in the Network Provider and it is the main **WAN Slice Controller** function, responsible for initiating the Network slice part creation requested by the **Slice Builder**. Upon receiving the **request_slice_part()** call along with the Slice Part Specification (see Figure 26), the **WAN Master** creates the slice part entry in its local database (**Slice Part Database**), returning the Slice Part ID of the allocated slice part to the **Slice Builder**. Later, when the **activate_slice_part()** method is invoked by the **Slice Builder**, the **WAN Master** interacts with the **WAN Agents** residing in the Data Centers that hosts the two DC slice parts to be connected. In practice, the **WAN Master** coordinates the two **WAN agents** to setup a VXLAN tunnel connecting the OVS bridges of both slice parts. This connection is done by implementing a L2 network overlay over the existing L3 network layer, which in most cases are the Internet.
- **WAN Agent**: this function resides in each Data Center where the **WAN Slice Controller** is able to provide connection to. This function receives instructions from the **WAN Master** to configure a VXLAN tunnel ending at one of the OVS bridges under its responsibility.

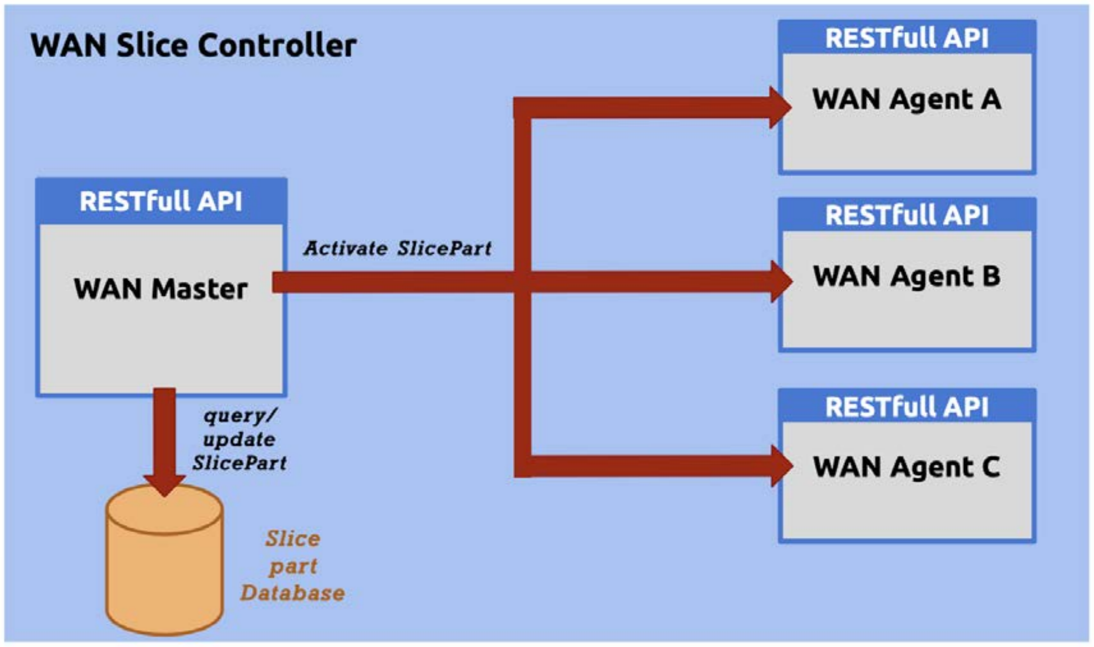


Figure 25: WAN Slice Controller internal functions.

```
slices:
  sliced:
    id: IoTService_sliced
    ...
    slice-requirements:
    ...
    slice-part:
      - net-slice-part:
          name: pop-dc-slice1-to-pop-dc-slice2
          type: NET
          WIM:
            name: VXLAN
            version: 1.0
            wim-shared: true
          links:
            - dc-part1:
                #slice part id
                dc-slice-controller-id: 2
                slice-part-uuid: 1
            - dc-part2:
                #slice part id
                dc-slice-controller-id: 5
                slice-part-uuid: 1
            - requirements:
                bandwidth-GB: 1
```

Figure 26: Slice Part Specification sent from Slice Builder to a WAN Slice Controller.

2.4. Marketplace

As mentioned before, the resource discovery framework proposed in the NECOS architecture is responsible for locating the appropriate resources that compose a slice, i.e., slice components that correspond to service functions and service links, according to the information model discussed previously.

In a nutshell, a **Partially Defined Template (PDT)** message defines the general slice requirements and acts as the input to the resource discovery framework. This message is passed from the **Slice Builder** to the **Slice Broker**, which in turn interacts with a set of **Slice Agents** in order to allocate resources that fulfil the slice requirements. A corresponding response in the form of a **Slice Resource Alternatives (SRA)** message is sent back to the Slice Builder once the Slice Broker has annotated the PDT message with alternative slice component options. In more detail:

- The **Slice Builder** is responsible for initiating the slice resource discovery process, by forwarding the PDT message to the **Broker**, and selecting the most appropriate slice components, among alternatives listed in a SRA message;
- The **Slice Broker** decomposes the PDT message it receives and creates a different query for each slice part. Each query contains all the constraints preferences, resources needed for the slice part, i.e., it presents a self-contained description of the slice part that will be addressed to the agents;

- The **Slice Agents** reside on the providers' domains and are responsible to answer requests for resources (queries) originating from the **Slice Broker.** The request message received is translated in a form that it can be *matched* with the resource descriptions that the provider maintains. In case of a successful matching the answer is returned back to the **Slice Broker**, annotated with all the missing information.

In essence, the marketplace can be viewed as a multiagent system, with the aim to achieve the task of discovering resources in a dynamic, distributed manner. As such, the marketplace needs an efficient, stable messaging scheme. For the needs of the marketplace the current implementation adopted the RabbitMQ, an open source message-broker middleware, that will be detailed in the corresponding section.

2.4.1. The Slice Broker

The Slice Broker and its relations to the rest of the NECOS system components are depicted in Figure 27. Its two main functions, as those were described in Deliverable 5.1, are the *PDT Message Analyser* and the *Resource Discovery Control*.

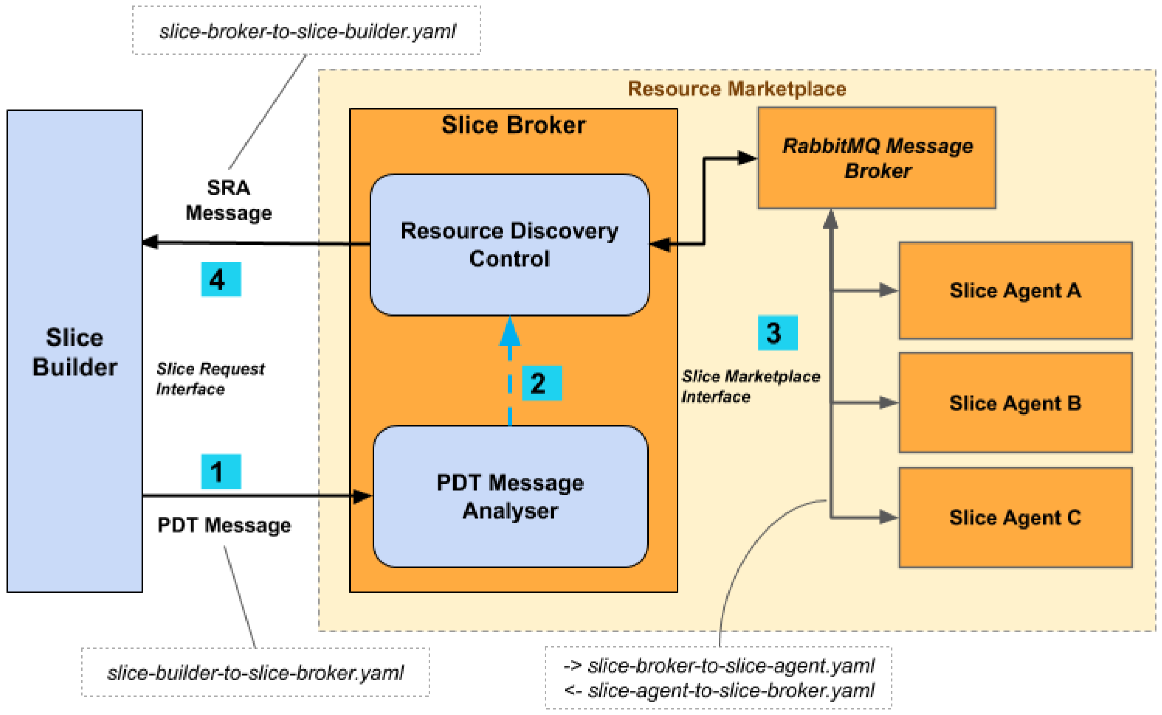

slice-broker-to-slice-builder.yaml
Resource Marketplace
Slice Broker
RabbitMQ Message Broker
SRA Message
Resource Discovery Control
4
Slice Builder
Slice Agent A
3
Slice Agent B
2
Slice Request Interface
Slice Marketplace Interface
1
PDT Message Analyser
PDT Message
Slice Agent C
slice-builder-to-slice-broker.yaml
-> slice-broker-to-slice-agent.yaml
<- slice-agent-to-slice-broker.yaml

Figure 27: The Slice Broker Internal Components and Interactions.

The role of the **PDT Message Analyser** is break down the PDT message to requests, each involving a single slice part. Given the structure of the PDT message, such decomposition is performed with ease, however the analyser has to ensure that a request for a slice part is self-contained, i.e., it has all the necessary information for the agent to provide a reply. However, the latter is not fully defined in the same section of the PDT message. This occurs since different components refine the original tenant's request for a slice: for instance, the *Slice Activator* is responsible for specifying characteristics such as EPA attributes, whereas the *Slice Specification Processor* decides on the slice graph, i.e., the number of slice parts and *vdus* residing in each part. Thus, the PDT Message Analyser aggregates all that information, forming a complete request message, in terms of necessary demand information.

Figure 28 shows an excerpt of a slice part request for a particular *dc-vdu* (i.e., a web_server_VM).

```
"dc-slice-part": { "name": "dc-slice1",
  "vdus":{[
    {"dc-vdu": {
      "id":"web_server_VM",
      "description":"web-servers
         for elastic CDN deployment",
      "host-count-in-dc": { "equal": 5 },
         ... }
      },...]}
```

Figure 28: DC-VDU specification example.

The requested EPA attributes are defined in the service function section of the PDT message, as shown in Figure 29 below:

```
"service-function": {
  "service-element-type": "vdu",
    "vdu":{"id": "web_server_VM",
         "epa-attributes":
           {"host-epa":
             {...,
              "storage-gb": 2,
              "memory-mb": 4096,...}
              } ...
```

Figure 29: EPA attributes specification example.

The analyser collects all the necessary information found in the request message for the resource discovery, in an aggregated request message for a particular slice part as shown in Figure 30:

The **Resource Discovery Control** component is responsible to query all agents in the marketplace for each slice part and collect the corresponding responses. To achieve this, it forwards each formulated request from the PDT message analyser to the **Message Broker**. Regarding the implementation features, each such request is annotated with geographical constraints as a means of reducing the message exchange overhead. Thus, in practice, the Message Broker directs the incoming requests to a suitable (geographically-wise) subset of agents according to a relative mechanism described in the following section.

```
"dc-slice-part":
  {"name": "dc-slice1",
        ...
   "vdus":{[
    {"dc-vdu": {
      "id":"web_server_VM",
      "host-count-in-dc": 5,
      "storage-gb": 2,
      "memory-mb": 4096 , ...},
    {"dc-vdu": ...}
      ]}
```

Figure 30: Request message for a particular slice part.

2.4.2. The Message Broker

The marketplace employs the RabbitMQ message broker to implement all communication among the Slice Broker and the Slice Agents. It should be noted that having a reliable, scalable messaging component is crucial for the implementation of any multi-agent system and thus of great importance to the NECOS Marketplace. RabbitMQ [RABBITMQ 2019] is a widely deployed open source message broker and the platform of choice for the Marketplace implementation, since it offers the following advantages:

- it is lightweight and easy to deploy;
- supports multiple messaging protocols and asynchronous messaging;
- can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements;
- it is officially supported on a number of operating systems and several programming languages.

In the current implementation, all the components involved in the marketplace are connected via the RabbitMQ server, each associated with a unique queue. A queue is the name for a “post box” which lives inside RabbitMQ. Although messages flow through RabbitMQ, they can only be stored inside a queue, in order to be consumed by the appropriate components. An important feature of RabbitMQ is that the producer never sends any messages directly to a queue, but instead message delivery relies on the notion of an *exchange*: a mechanism that receives messages from producers and pushes them to *specific queues*, according to the exchange type employed (direct, fan-out, topic and header) in the messaging model in RabbitMQ. The exchange must know exactly what to do with a message it receives or otherwise it will discard it.

One important message reduction feature needed to be implemented in NECOS, is that of selectively addressing requests to the providers upon specific geographic criteria. For instance, it is pointless to direct a message to Brazil-located providers, when the incoming request specifies that the slice part to be located in Europe. The implementation of this feature takes advantage of “topic” exchanges in RabbitMQ, that provide a mechanism for pattern matching between the *routing key* property of a message and the *routing patterns* of each queue belonging to a provider agent. The routing key is simply a list of arbitrary words delimited by dots, that usually specify some features related to the message.

In the Marketplace case, the *routing key* expresses the geographic constraints of the specific request. Thus, if the slice part is constrained to be located in Europe, then the *routing key* will be "**Europe.***", with the star token ("*") standing for any location in the continent. In a similar manner, if the slice is to be located in Greece, then the corresponding key will be "**Europe.Greece**". Obviously, the pattern can accommodate constraints of increasing specificity, i.e., we can define "**Europe.Greece.Thessaloniki**" to inquire providers located to the area of Thessaloniki.

Similarly, each provider binds its queue with a pattern that defines its own location. For instance, assuming that there are three providers, located in Spain, Greece and Brazil, their *routing patterns* will be "Europe.Spain", "Europe.Greece" and "America.Brazil", respectively. If the Broker annotates the request message with the key "Europe.*" then the first two providers will receive the message, whereas if the request is annotated with "Europe.Greece", then only the second one will be contacted.

Thus, this simple mechanism allows for the implementation of a "filtering" mechanism that significantly alleviates the message exchange overhead.

2.4.3. The Slice Agent

The Slice Agent, as depicted in Figure 31, consists of four main modules: the agent Request Handling, the Translator to Provider Module, the Matching Mechanism and the Resource Information DB.

The **Agent Request Handling** is responsible for accepting and forwarding messages originating from the Slice Broker via the Message Broker. The component is also responsible for orchestrating all the other components.
The **Translator to Provider** module communicates directly with each resource provider, and obtains the list of available resources. This allows to retrieve in real-time up-to-date information. It is also responsible for translating the provider's response message in a more coherent format, in compliance to the NECOS information model. It then forwards the message to the resource Resource Information DB component.

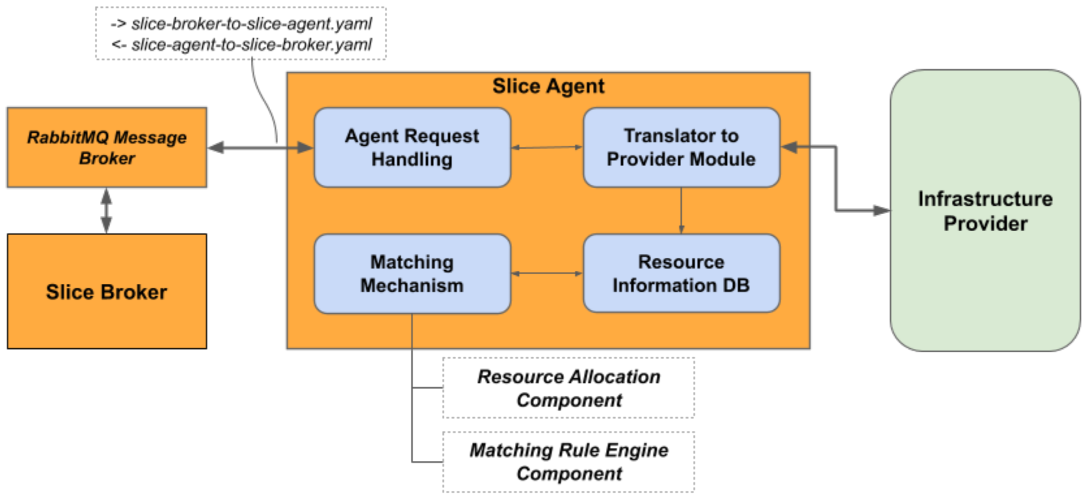

-> slice-broker-to-slice-agent.yaml
<- slice-agent-to-slice-broker.yaml
Slice Agent
RabbitMQ Message Broker
Agent Request Handling
Translator to Provider Module
Infrastructure Provider
Matching Mechanism
Resource Information DB
Slice Broker
Resource Allocation Component
Matching Rule Engine Component

Figure 31: The Slice Agent internal components and interactions.

The **Resource Information DB** stores information related to resources in a form that can be processed by the Matching Mechanism.
Finally, in order to discover whether a provider can accommodate a slice part, the **Matching Mechanism** component translates the request to a set of resource availability constraints the part should satisfy. Details regarding this component are to be presented in the section that follows.

2.4.3.1. The Matching Mechanism Implementation

As depicted in Figure 31, the **matching mechanism** consists of two parts:

- the *matching rule engine*, and
- the *resource allocation component*

The **matching rule engine** component is responsible to infer *coverage* of a given request by a subset of the resources of the resource provider, i.e., finding those resources that meet or exceed (cover) the requirements stated in the request message (*resource coverage problem*). One important assumption that we make in the following is that providers organise their offered resources in *clusters,* i.e., sets of machines with identical configuration and price. We also assume that a dc-vdu should be allocated to a single such cluster, meaning that the selected cluster must have enough host capacity to cover the number of hosts the vdu requires.

As mentioned in the corresponding section of deliverable D4.2 this mechanism relies on a set of *matching rules*, i.e., rules of the form:

matching_rule (RuleId, YamlPath, ProviderResourcePath, Operator)

where *RuleID*, is self-explanatory, *YamlPath* is the attribute's path in the YAML resource request message to be matched, *ProviderResourcePath* is the corresponding Provider's infrastructure representation path to the matched characteristic and, finally, *Operator* is the binary relational operator used to compute coverage, e.g., ">", "<", etc. In our Prolog implementation, matching rules are represented as Prolog facts. For instance, the rule illustrated in Figure 32 below checks if hosts in a provider have RAM memory capacity above a specific threshold to cover the dc-vdu (virtual deployment unit assigned to a dc-slice part) constraints, as the latter were defined in the corresponding epa-attributes.

```
matching_rule(r1,
          ['dc-vdu','epa-attributes','host-epa','memory-mb'],
          ['node_cluster','memory_mb'], >=)
```

Figure 32: Matching rule representation.

This matching mechanism also translates textual constraints to a more appropriate form, overriding possibly the relation stated in the corresponding rule. This allows easy implementation of a rich language to state constraints: any textual description such as {**"less_than": 10**} can be translated to the corresponding relational expression, in this case "**<=**".

The approach taken currently by the matching rule engine is rather straightforward: the engine iterates over every vdu in the request, and for every *set of resources (clusters)* in the provider it checks the satisfiability of every rule (*rule-test*). Each such rule-test essentially consists of:

1. fetching the request attribute value (rule argument 2),
2. fetching the provider resource value (rule argument 3),

3. dynamically creating the relation given the operator and the two values above,
4. evaluating the result of the created relation.

This rule-test implementation relies heavily on higher order Prolog predicates, in order to be able to create dynamically the computed relations.

Each rule-test outcome is recorded and returned by the matching rule engine. This “exhaustive” approach the engine follows (instead of actually failing on the first rule-test failure), allows the agent to provide explanations *why a request has failed*: the outcome of the rule-test can be returned as an answer which assists the user (tenant) to understand the reason behind the slice request failure. For instance, as it was stated in deliverable D4.2, an example of an indicative answer is illustrated in Figure 33.

```
dc-slice-part:
        name: core_vm,
        allocated_to: DSS/MOBILE
        nodes:5
        outcome:
                - succeeded:
                …
                - failed:
                        path: [dc-vdu,epa-attributes,host-epa,cpu-
                number]
                        offered: 1
                        desired: 2
```

Figure 33: Example of an indicative answer.

The fields in this example indicate not only that the request has failed, but also *why* it failed. In the specific case, hosts in the cluster had only a single core (*cpu_number,* value of the *offered* field) instead of two cores required (value of the *desired* field).

Obviously, the engine’s performance can be increased by turning off the explanation facilities and simply failing on the first unsatisfiable rule-test. Other methods, such as prioritizing rule-tests of the most likely to fail attributes can be also used to improve the performance.

The mechanism described above is the implementation of a simple matching process that discovers all dc-vdus a provider can host. The current implementation relies on the advanced pattern matching and symbolic manipulation characteristics of Prolog to perform this task.

However, in many cases multiple clusters match a single vdu. Thus, *a set of allocation pairs* (*<vdu>*,*<cluster>*) is generated by the matching mechanism, representing all possible allocations. For instance, if a slice part request has three (dc-)vdus, named *load_balancer*, *web_server_vm* and *orchestrator*, and the clusters in the Data Center wilab2 (Fed4FIRE testbed) are *ZOTAC*, *APU 1d4*, *SERVER1P*, *SERVER5P*, then an example result of the matching rule engine is shown in Figure 34.

***** (VDU, Cluster) *****
load_balancer,ZOTAC
load_balancer,APU 1d4
load_balancer,SERVER1P
load_balancer,SERVER5P
***** (VDU, Cluster) *****
web_server_VM,ZOTAC
web_server_VM,APU 1d4
***** (VDU, Cluster) *****
orchestrator,ZOTAC
orchestrator,APU 1d4
orchestrator,SERVER1P
orchestrator,SERVER5P

Figure 34: Alternative Matching Pairs.

Clearly, the matching mechanism should select one of the above alternatives for each (dc-)vdu. The issue that arises is that the rule-tests check the cluster coverage for one vdu at a time, and thus might not have the host capacity to host all vdus indicated by the resulting pairs. For instance, in the example above assume that the web_server_VM requires five (5) hosts and the load_balancer demands one (1) host. If the ZOTAC cluster has a total of 5 hosts, it is obvious that, although the resulting pairs are valid, allocating both the load_balancer and the web_server_VM to ZOTAC is not possible.

The aforementioned problem is a classic example of a resource allocation problem, where vdus are tasks and clusters are the resources. Out of the plethora of techniques to tackle the problem (linear programming, dynamic programming, etc.), we have selected to use Constraint Logic Programming (CLP) and formulate the problem as a Constraint Satisfaction Problem (CSP). Informally, a CSP problem consists of:

- a set of Decision Variables;
- a domain (possible values) associated with each Variable; and
- a set of Constraints on the Decision Variables.

In this specific case, we consider the following formulation. Given the set of $VDUS = \{vdu_{(1,)} vdu_{2,} \dots vdu_v\}$ to allocate, with $VDUS \vee v$, and the set of provider $CLUSTERS = \{cluster_{1,} cluster_{2,}, \dots, cluster_c\}$, with $CLUSTERS \vee c$, and each $vdu_i \in VDUS$ can be allocated to a subset of the clusters $clusters_i \subseteq CLUSTERS$. In the following V stands for the set of integers $V = \{1..v\}$ and similarly C is the set $C = \{1..c\}$. Each cluster $cluster_k \in CLUSTERS$ has an associated host cost $hostCost_k$, and a maximum available host capacity $maxHosts_k$ and each $vdu_i \in VDUS$ has a demand for hosts $vduHosts_i$. The problem's variables are:

- One variable *Vdu*$_i$ per vdu ranging over its potential clusters, i.e.$\forall i \in \{V\}, Vdu_i \in clusters_i$ where $clusters_i \subseteq CLUSTERS$
- One variable $Cost_i$ per vdu that represents the cost of the final allocation of the vdu to a cluster.
- V Variables for each $cluster_k$, i.e. $\forall k \in C, \forall l \in V \{Cluster_{kl} \in \{0..nodes_k\}$. Each variable $Cluster_{kl}$ represents the allocation of vdu_l to the cluster $cluster_k$, with 0 representing that the vdu was not allocated to the cluster.

The problem's constraints are the following. The allocation of a vdu $vdu_i$to a cluster $cluster_k$implies that the ith variable of the cluster will be equal to the demand for hosts of the vdu $vduHosts_i$ and the cost of the allocation will be that of the cluster's host cost ($hostCost_k$times the number of allocated hosts:

$$\forall i \in V, Vdu_i = cluster_k \rightarrow Cluster_{ki} = vduHosts_i \wedge Cost_i = hostCost_k * vduHosts_i$$

Since each vdu has to be allocated to a single cluster, once an allocation is decided, it implies that all other clusters "remove" this allocation from their respective variables, i.e. impose the constraint that the respective variable should be equal to 0.

$$\forall i \in V, Vdu_i = cluster_k \rightarrow \wedge_{(\forall m \in C, m \neq k)} Cluster_{mi} = 0$$

Since each cluster *k* has a limited number of hosts, the following capacity constraint must hold:

$$\forall k \in C, \sum_{i \in V} Cluster_{ki} \leq maxHosts_k$$

Given the CSP formulation provided above, the problem is to find the allocation, that *minimizes the total cost* of the allocations:

$$TotalCost = \sum_{i \in V} Cost_i$$

The *Resource Allocation Component* uses Constraint Logic Programming (CLP) and the Prolog solver to automatically formulate the problem (i.e., build the respective constraints) based on the results of the Resource Matching Engine. The constraints used to model the problem are the classic constraints used in problems of this kind, i.e., the *element/3, global_cardinality/2* and *transpose/2* constraints. It should be noted that the solver returns the guaranteed minimum cost allocation, using the branch and bound algorithm, with standard CLP heuristics employed in search, i.e., the fail-first heuristic.

One advantage of this approach is that any constraints concerning a specific provider are included relatively easily. For instance, the provider might wish to allocate each vdus to different clusters in order not to overtax resources on a single cluster, or allocate as many vdus to a single cluster in order to minimize communication between clusters.

The agent returns its minimum cost solution, since the latter maximizes its probability that its offer will be selected, by the NECOS *Builder.*

2.4.4. Experiments

In order to test the resource allocation component implementation we have generated a set of benchmarks with varying characteristics. Each problem generated has the following characteristics:

- *number of VDUs*, that is the number of VDUs in the generated allocation pairs;
- *vdu size*, that is the maximum number of hosts in each vdu;
- *max matchings*, that is the maximum number of matched clusters, i.e. number of allocation pairs for a single vdu;
- *number of clusters* that is the number of clusters in the provider; and

EUB-01-2017

- *maximum availability*, that is the maximum available hosts in each cluster.

Figure 35 presents our initial experimental results. For this experimental set, all the problems generated have a *vdu size* equal to 5 hosts and the number of *maximum matchings* is 5 clusters for each vdu. Each line represents the average execution time for each allocation when there are different numbers of clusters (50 and 20 clusters) with a host availability ranging from 7 to 15 hosts in each cluster. Each set of experiments depicted in a line concerns 1, 2, 5, 10 and 15 vdus respectively. It should be noted that the time reported is in *millisecond (ms)* and is the average value out of 10 problem instances. Experiments were conducted on an INTEL Core i5 machine with 8GB of RAM.

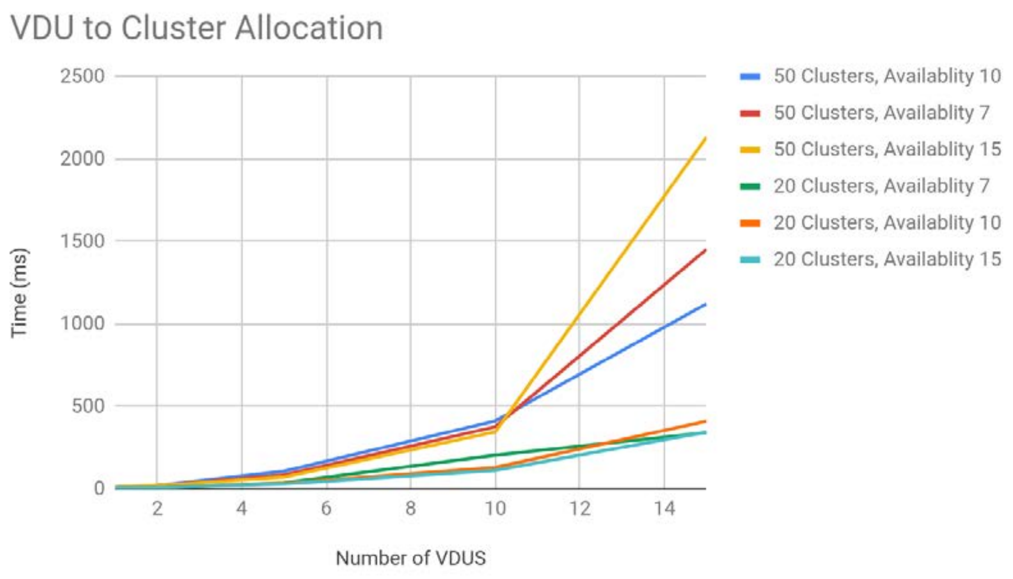


Figure 35: Initial experimental results of the Resource Allocation Component.

Not surprisingly, as the number of *vdus* increases the execution time increases. Given that this is a classic combinatorial problem and we are searching for the best solution using branch and bound, time increases exponentially. The diagram also shows that the number of provider *clusters* affects the execution time, and this is attributed to the fact that the problem has significantly more decision variables.

The reader should consider that each such allocation problem is solved by a single provider for each slice part that it receives a request for. If we consider that slice parts will usually host a few services (thus vdus) then the approach can be employed without introducing significant delays in the resource discovery process. Future work includes experimentation with far more efficient CP platforms than the one found in the SWI-Prolog used in the current implementation, such as ECLiPSe [ECLIPSE 2019], Prolog, MiniZinc [MINI 2019], Google OR-Tools [ORTOOLS 2019] and the related solvers.

2.4.5. Initial Testing of the Marketplace Implementation

In order to test the implementation and have an initial view of the number of alternative slice parts, we have conducted a series of initial test runs. In order to be able to test the implementation we have created a problem generator that consists of two components:

- the **agent resource file generator**, that is responsible for generating resource files, with resource characteristics (memory, storage, etc.) ranging over random values;

- the **request generator**, that given a set of desirable characteristics of the request, i.e. the number of slices and hosted services, number of network slice parts and the slice part connectivity, is able to create requests, with specific demands populated by random values.

Given the above generator, we have created 10 provider resource files, initiating an agent for each one and a small set of requests of different sizes, i.e. different number of DC/Net slice parts in each request. In order to test the marketplace approach to resource discovery, we tried three different slice requests, with a varying number of DC and Net Slices parts. For each such request, we recorded the total number of alternative DC parts received by the marketplace, i.e. offers by different providers that can ”cover” the DC-Slice requirements and the total number of alternative Net slice parts, i.e. offers by WAN providers to connect the DC-parts. These offers are combined in order to generate the alternative slice instantiations, i.e. the options in the selection of the final slice providers. Results from these initial experiments are summarized in Table 2.

Table 2: Number of Alternative Slice Instantiations on request of different sizes.

Slice Name	No DC Slice parts	No Net Slice Parts	Total number of Alternative DC Parts	Total number of Alternative Net Parts	Alternative Slice Instantiations
Slice 1	2	1	7	30	30
Slice 2	3	2	7	30	108
Slice 3	4	5	13	120	4374

As expected, this is a combinatorial problem: the number of alternative slice parts has a significant impact on the number of alternative slice instantiations that the builder has to consider in order to select the one with the minimum cost (best one). When the number of alternatives is relatively small, brute-force techniques can solve the problem, however, a larger number of slice part alternatives will generate a significant number of slice instantiations to consider (i.e. a combinatorial explosion phenomenon) and thus, demand more advanced techniques. A more in depth experimental evaluation with a larger number of providers and requests will be presented in deliverable D6.2.

2.5. Tenant

Although the Tenant is not strictly part of the main scope of NECOS, it still plays an important role as it might be considered the user of the NECOS ecosystem, i.e., the entity requesting the creation of end-to-end slices that will eventually be utilised for the deployment of services delivered to its customers (end-users). As such, the following section will provide a brief overview of some of the software elements existing in the Tenant’s domain and how they interact with the NECOS components, specifically those that are part of the LSDC Slice Provider.

2.5.1. Service Orchestrator

The functionalities of a Service Orchestrator are complex. Many initiatives have worked / have been working on the development of software systems that implement the features defined in the MANO specification that are associated to the service orchestration. Among them, the 5GEx project [5GEx 2019] aimed at delivering a fully working MANO prototype that was compliant with the ETSI specifications. In the context of NECOS we wanted to highlight that the creation of end-to-end Slices is a nearly transparent process from the Tenant perspective: more specifically, as soon as an end-to-end Slice is requested and returned back to the Tenant, it can be utilised by existing components of the Tenant's domain without any particular additional efforts.

As such, we demonstrate in D6.2 that it is possible for a Tenant to attach its existing software components dealing with the embedding of virtual service elements on a newly created end-to-end Slice. This has been achieved by re-utilising the component of the 5GEx MANO prototype dealing with the virtual service embedding in order to automate the deployment of services on the allocated slice(s). In 5GEx this was implemented via a multi-layered hierarchical embedding approach which is reflected in NECOS by the existence of the Service Orchestrator (in the Tenant's domain) and the Service Orchestrator Adapter in the LSDC Provider.

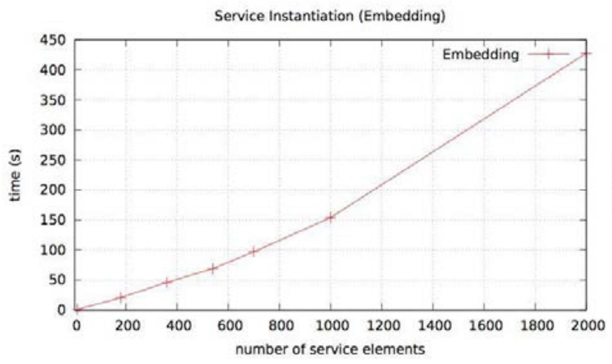
2.5.2. Slice Activator

In order to properly interact with a NECOS LSDC Slice Provider and request the instantiation of Slices to be used for the deployment of services, a Tenant will also need to run their own Slice Activator component. As a proof of concept demonstration, in D6.2 we described the usage of such component in a large-scale service deployment workflow. More specifically, a newly developed software artefact was responsible for the execution of the following main tasks:

- Allowing a Slice Description to be sent to the NECOS platform entry-point, i.e., the Slice Preprocessor (or Slice Builder);
- Handling the response coming back from the NECOS Slice Provider that informs a Tenant that a new end-to-end Slice is live and ready to be utilised;
- Run (if it does not already exist) and / or (re-)configure an instance of a Service Orchestrator to utilise the allocated end-to-end Slice as resource substrate for the embedding of the service elements;
- Finally triggering the instantiation of the service on the end-to-end Slice via the above mentioned Service Orchestrator

For the sake of demonstrating this concept, a basic implementation of the Slice Activator was developed and used in the large-scale service deployment demonstration (see D6.2). The component was developed in Python and provides a RESTFul API that can be utilised by the Tenant to trigger the Slice creation workflow (via POSTing a YAML Slice descriptor). Once the Slice is allocated, the Slice Activator will create an on-demand instance of the ESCAPE Embedding Engine (from the 5GEx MANO) as a Docker Container configured on-the-fly to utilise the above new end-to-end Slice instance. A separate API endpoint will finally allow the Tenant to submit a new service instantiation request to the Service Orchestrator (via POSTing a descriptor in the specific accepted format).

Figure 36 shows some timings collected during the execution of service instantiation operations (i.e., embedding and deployment) when the scenario described above was considered. As such, the component utilised to perform the embedding was ESCAPE (from the 5GEx MANO) and the resource substrate consisted of an end-to-end Slice with Parts based on the lightweight VLSP VIM (see D6.2, Instantiation of large-scale lightweight services on a sliced infrastructure for further details).



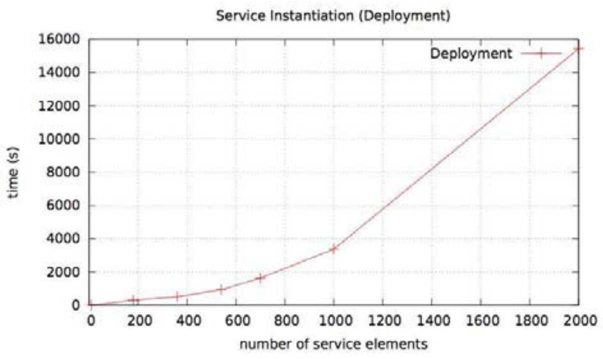


Figure 36: Execution times for service instantiation operations.

The results are specifically related to the scenario that was considered for the demonstration and shows how the orchestrator was able to utilise the end-to-end Slice transparently as resource substrate to embed and deploy virtual service elements in a variable number (i.e., from 10 to 2000 VLSP virtual routers). The deployment timings also include the time contributions required for the instantiation of the application logic on each VLSP virtual router.

3. Provisioning and Orchestration Workflows

As in Deliverable 5.1, in this section, we use the BPMN (*Business Process Model and Notation*) methodology to document the current flows developed in NECOS. As the project matured, some concepts changed and with them some of the workflows. Notably, the main change made during the project development was the inclusion of the "service", this is the higher-level application directly managed by the tenant, as a part of the slice. This impacted directly in the responsibilities of the NECOS system, having to take responsibility over some service management tasks. For example, to scale up a slice now includes redeploying the services running on its parts to newly created slice parts and modifying the configuration of infrastructure services such as load balancers or DNSs. **In this section, we include only those workflows that were modified since Deliverable 5.1.**

Regarding the notation adopted for building diagrams, a green circle denotes the start of a workflow and an orange circle denotes the end of a workflow. A rounded gray rectangle represents an action taken during the workflow. A set of actions (gray rectangles) grouped in dashed squared boxes represents a NECOS Architectural component, i.e., they represent a set of actions assigned to a specific module proposed in NECOS architecture.

There are different types of gateways (diamond shapes) in the workflows. The diamond with a circle inside represents an **Inclusive gateway**, i.e., when the flow reaches such a gateway, it is necessary to evaluate the set of next steps to be taken. It is possible to select a mix of actions composed by one or more steps linked from this gateway. The diamond with an X inside represents an **Exclusive gateway**, i.e., when the flow reaches such a gateway, only one next step is taken from the options linked in this gateway. The diamond with an "+" inside represents a **Parallel gateway**, i.e., when the flow reaches such a gateway, all the steps linked from this gateway are performed in parallel.

3.1. Slice Creation Workflow

This section documents the slice creation workflow according to architectural design aspects available on Deliverable D3.1 and APIs defined on Deliverable D4.1 and it was updated following the design changes presented in D3.2 and D4.2. Figure 37 shows all the architectural modules involved in the Slice Creation workflow, and the arrows represent the interaction in between modules required to create a new slice. For this workflow, we assume that a tenant seating at the Service Provider realm uses the Slice Activator module to issue a slice request to a NECOS Slice Provider. The Slice Provider interacts with the Marketplace and Infrastructure Providers in order to deliver the slice to the tenant. The process goes on as the Slice Activator delivers all the details of the new slice to the Service Orchestrator module. This later, in its turn, starts the Service Deployment Workflow on top of the new infrastructure parts of the slice, by communicating with the Service Orchestrator Adaptor using the dotted line present in Figure 37, which, in turn, ask the SRO to do that through the IMA. These two steps are the main difference with the corresponding workflow in D5.1.

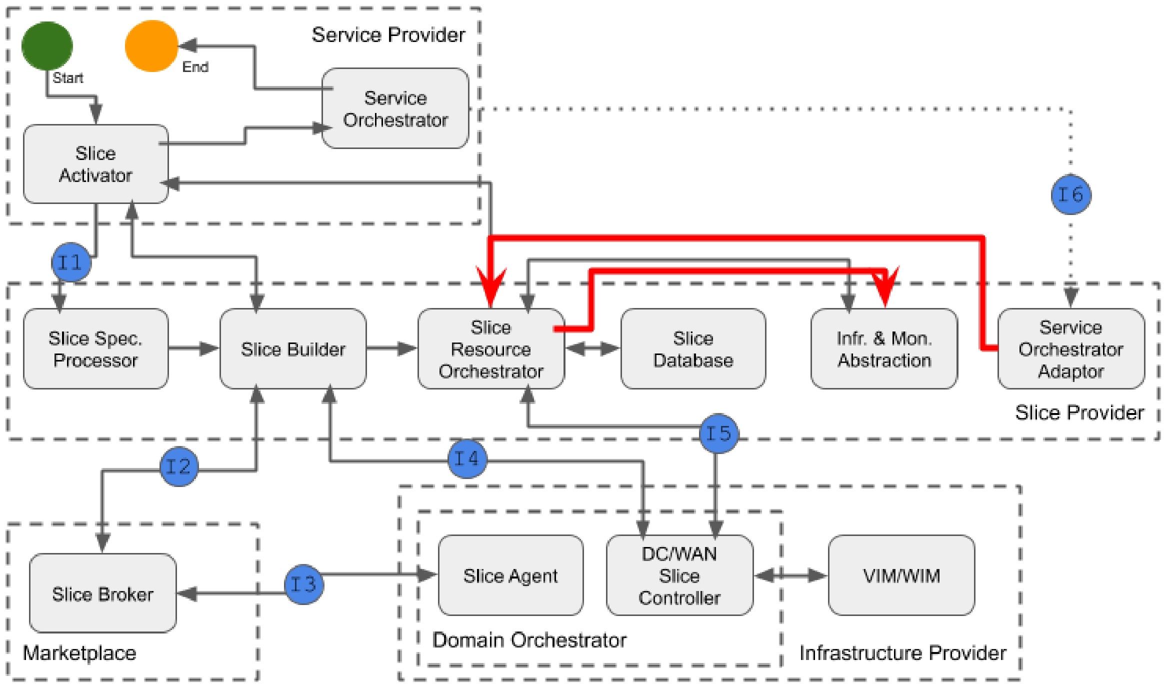


Figure 37: Updated overall slice creation workflow.

Figure 38 presents the Slice Resource Orchestrator, responsible for the end-to-end integration of the new slice and for initial management aspects of it. Once all slice parts are ready to use, the Slice Builder delivers to the Slice Resource Orchestrator the Full Slice Details, so it can define the required actions in order to realize the end-to-end binding of the separated slice parts. This is a task performed in parallel by all the DC/WAN Slice Controllers involved in the slice construction, and after all of them finalize their duties, the Slice Resource Orchestrator runs the management actions. One of the tasks is responsible for keeping the detailed slice description in the Slice Database module, and another is responsible for delivering the management pointers (VIM/WIM information) to the Infrastructure & Monitoring Abstraction (IMA) module, in order to set up monitoring and infrastructure adaptors that will be necessary during runtime. After all slice details are returned to the Slice Activator, it contacts the Service Orchestrator which, in turn, triggers the service deployment with the help of the SRO. In red, we depict the updated part of the workflow.

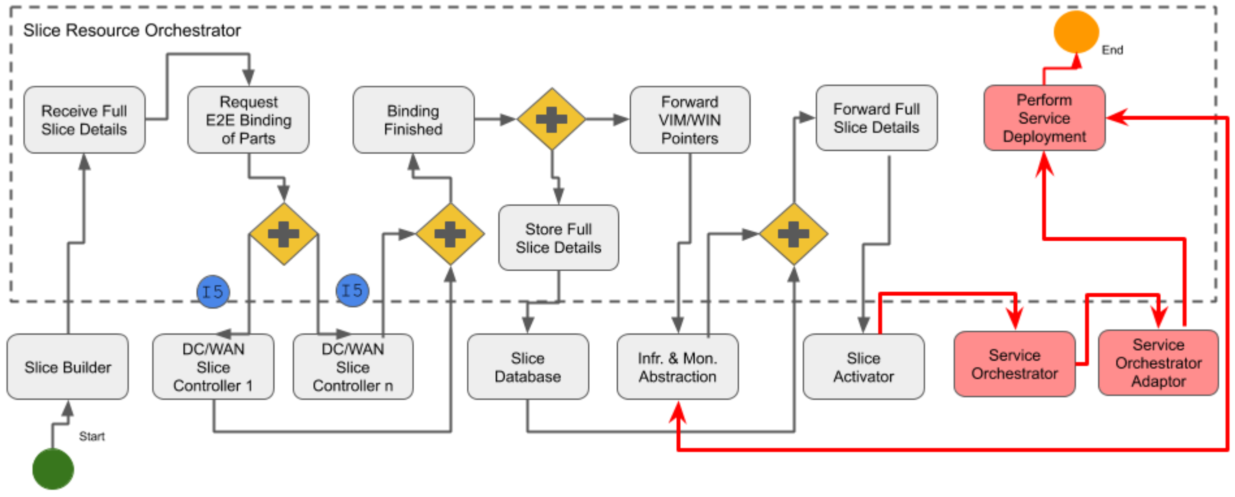

Slice Resource Orchestrator
Receive Full Slice Details
Request E2E Binding of Parts
Binding Finished
Forward VIM/WIN Pointers
Forward Full Slice Details
Perform Service Deployment
End
Store Full Slice Details
I5
I5
Slice Builder
DC/WAN Slice Controller 1
DC/WAN Slice Controller n
Slice Database
Infr. & Mon. Abstraction
Slice Activator
Service Orchestrator
Service Orchestrator Adaptor
Start

Figure 38: Updated version of the detailed Slice Resource Orchestrator participation in the Slice Creation Workflow.

3.2. Slice Decommission Workflow

This subsection documents the updated slice decommission workflow. An overview of the interactions between the modules is presented in Figure 39.

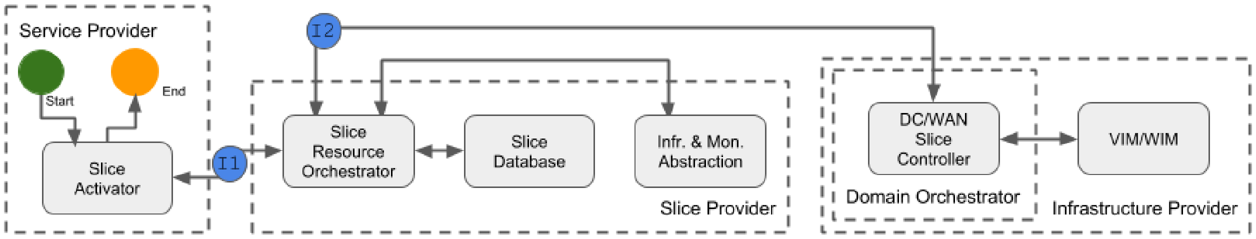


Figure 39: Overall interaction of modules for slice decommission.

As illustrated in Figure 40, the Client-to-Cloud API, depicted as I1, is used by the Slice Activator (tenant) to start the decommission process. The main difference with the version in D5.1 is that before decommissioning the infrastructure parts of the slice, its service parts must be decommissioned successfully too; this is made with the help of the IMA. In D5.1, this task was left to the tenant. After this, the workflows continue as before. The Slice Resource Orchestrator is the component responsible for receiving this request and retrieving information about the Slice Topology and its parts located in different Infrastructure Providers, by interacting with the Slice Database. With that information, the Slice Resource Orchestrator sends the decommission request to the DC/WAN Slice Controllers in order to shut down the slice parts. Such request uses the Cloud-to-Cloud API depicted as I2 in Figure 40. Each DC/WAN Slice Controller, after receiving the request from the Slice Resource Orchestrator, requests the VIM/WINs to decommission the respective infrastructure inside each provider. After this, the VIM/WINs send an Ack Message to the respective DC/WAN Slice Controllers, informing the status of the request. Once all involved DC/WAN Slice Controller finish the local decommission, the Slice Resource Orchestrator starts a process to update the Infrastructure and Monitoring Abstraction (IMA) and the Slice Database components. These actions include removing the VIM/WINs pointers inside the IMA and deactivating the Slice information from the Slice Database. Finally, after finishing all steps, the Slice Resource Orchestrator informs the Slice Activator about the status of the decommission process.

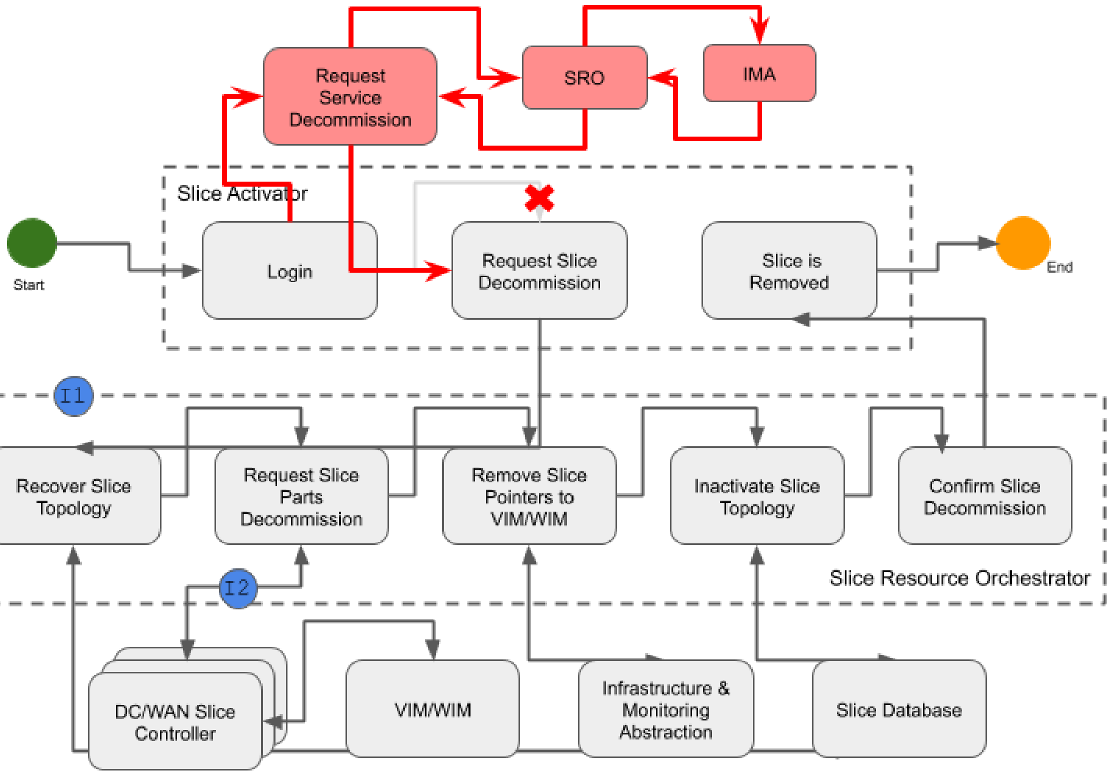

Request Service Decommission
SRO
IMA
Slice Activator
Start
Login
Request Slice Decommission
Slice is Removed
End
I1
Recover Slice Topology
Request Slice Parts Decommission
Remove Slice Pointers to VIM/WIM
Inactivate Slice Topology
Confirm Slice Decommission
Slice Resource Orchestrator
I2
DC/WAN Slice Controller
VIM/WIM
Infrastructure & Monitoring Abstraction
Slice Database

Figure 40: Updated decommission workflow.

3.3. Elasticity - Downgrade of Resources Workflow

The modification suffered by the workflow for the Elasticity Downgrade is similar to that for the Slice Decommission. An overview of the interactions between the modules is presented in Figure 41. As illustrated in Figure 42, the process starts from the IMA component collecting monitoring metrics. The SRO evaluates these metrics and detects whether it is necessary to downgrade an idle resource or not. After identifying the need of slice downgrade, the SRO retrieves information related to the respective Slice Part from the Slice Database in order to define which slice parts should be adapted. Unlike in the previous version of this workflow, now the SRO asks the IMA to modify the deployed service to adapt to the new, smaller, topology. Next, the SRO follows as before. It requests reduction and/or removal to the DC/WAN Slice Controller via the Slice Runtime Interface as depicted in I1. The DC/WAN Slice Controller communicates with the VIM/WIM in order to decrease or shutdown the resources. The SRO receives the confirmation from the DC/WAN Slice Controller and updates the information into the IMA and the Slice Database. Finally, the Slice Resource Orchestrator informs the Service Provider (tenant) about the elasticity performed.

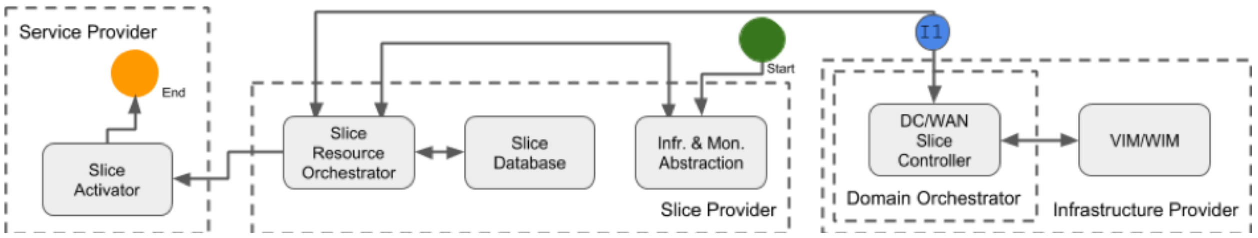


Figure 41: Overall interaction of modules for elasticity downgrade.

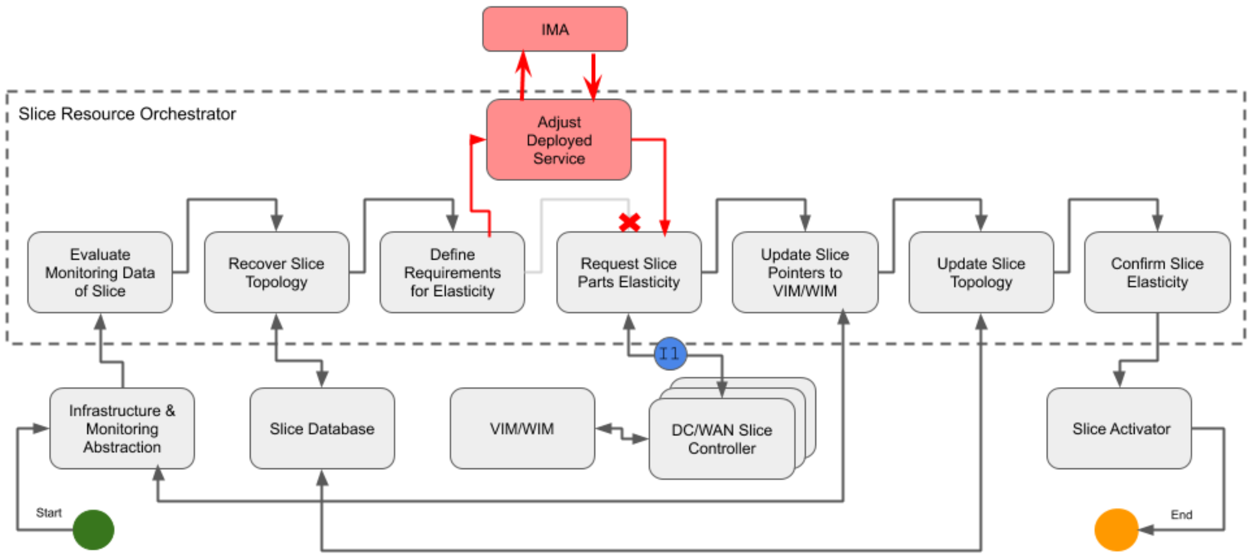

IMA
Slice Resource Orchestrator
Adjust Deployed Service
Evaluate Monitoring Data of Slice
Recover Slice Topology
Define Requirements for Elasticity
Request Slice Parts Elasticity
Update Slice Pointers to VIM/WIM
Update Slice Topology
Confirm Slice Elasticity
I1
Infrastructure & Monitoring Abstraction
Slice Database
VIM/WIM
DC/WAN Slice Controller
Slice Activator
Start
End

Figure 42: Updated elasticity downgrade workflow.

4. Conclusions and Outlook

As stated in the Description of Action (DoA) of this project, WP5 is meant to materialize a set of software artefacts, which constitute the system architecture specified in WP3 and the APIs designed in WP4. In addition, this platform and their constituting software components have to fulfil the set of functional and non-functional requirements elicited in WP2. In that way, WP5 would deliver the appropriate platform to WP6, which is aimed to design the NECOS testing environments. More specifically, the DoA establishes that the software to be created has to implement the monitoring framework of virtual and physical resources (objective O5.2), the implementation of a knowledge plane to ground the processes of decision taking (objective O5.4) and finally, the implementation of the NECOS cloud management and orchestration system underpinning the slice as a service paradigm developed in this project (objectives O5.1 and O5.3). At the moment of ending the edition of this deliverable, we can attest that all these goals have been successfully achieved, as argued in the following paragraphs.

One of the results of this project is the NECOS Lightweight Software Defined Cloud (LSDC), which constitutes the core of the design and development tasks carried out within WP5 and WP4 and the specification tasks conducted within WP3 and WP2. The LSDC is a platform allowing slice-as-a-service offerings by means of the integration of two main components, one of them being the Integrated Monitoring Abstraction (IMA), which confers in great part the design with the lightweightness property. In fact, the IMA is leveraging on top of the Very Lightweight Network & Service Platform (VLSP) formerly developed by UCL and that constitutes an abstraction layer allowing the interaction with any type of virtual infrastructure managers (i.e. VIMs and WIMs) as well as any type of monitoring systems. In addition, the other component of the LSDC, is the Slice Resource Orchestrator (SRO), which materializes the above mentioned knowledge plane. The SRO is in charge of slices activation and registration, both vertical and horizontal elasticity processes, the instantiation of virtual resources to deploy the services on the slices and the slice decommissioning. The elasticity processes are based on scalable optimization and machine learning algorithms meant to estimate KPIs and predict potential violations of SLAs. The LSDC platform is thus the proof of achievement of the part of Objective 1 of the NECOS project that was stated in the DoA as to develop and build a Lightweight Slice Defined Cloud (LSDC) platform enabling computing, network, and storage elements in the cloud through the Slice as a Service across federated clouds. LSDC would be the first implementation of the Network-Cloud Slices-as-a-Service. In addition, we can claim that the platform developed within WP5 is the proof of achieving the Objective 3 of NECOS that was stated as to develop the service and resource orchestration and management methods for the LSDC infrastructure of resources within the network and at the edge of the network. This service orchestration and management approach includes the automatic re-allocation of resources and services across distributed and geographically separated computing, data storage, and network infrastructures within separate slices.

The LSDC characteristics reside not only in its internal subsystems but also in its APIs, by means of which interacts with the outside world. These APIs were designed in WP4 but their implementation and testing have been done in the context of WP5. The LSCD APIs are of two categories, namely the tenant-to-cloud and the cloud-to-cloud. Whilst the former is meant to facilitate tenants to specify slices, the latter is to facilitate the LSDC to interact with resource providers in a multidomain environment, thus allowing for the creation of slices constituted by resources of different resource providers. In that direction, NECOS brings another important contribution that has been fully designed and developed within WP5, which consists in its capability to discover resource parts and to select the resources that best fit the tenant specifications. This is achieved not like in the conventional federation agreements but through a dynamic Marketplace where the resource providers present their offers and the LSCD selects the most appropriate one (e.g. the cheapest one). The Marketplace is constituted by a Broker that interacts with data center and network providers through on purpose designed Slice Agents that are to be integrated inside the local domain orchestrator of each resource provider. The design and development of the components of the Marketplace constitute an important step to confer NECOS its unique distinguishing characteristics. These components close the loop that makes the LSDC to be operational and in that sense we can claim that WP5 has also contributed to part of Objective 1 of NECOS that is stated in the DoA as this LSDC platform will have open APIs offering different levels of service and resource abstractions. Also, the LSDC will enable the integration of applications with slice resources for a faster and automated service provisioning in the networked cloud. At the same time, LSDC contributes to Objective 2 of NECOS that was stated in the DoA as to develop the artefacts needed to make the LSDC a service provisioning approach characterized by the integration of resources within the collection of independent slices.

As part of research on integrating Network computation with computing clouds the following further development and integration challenges are envisaged:

- A Uniform Reference Functional & Non-Functional Model for Large Scale Network Cloud Slicing;
- Uniform Slice Templates: Providing the design of slices to different scenarios;
- Efficient Slicing Service Mapping – creating an efficient service mapping model binding across network cloud slicing;
- Recursion, namely methods for slicing segmentation allowing a slicing hierarchy with parent–child relationships;
- Customized security mechanisms per slice - In any shared infrastructure;
- Network Slices Reliability - Maintaining the reliability of a network cloud slice instance, which is being terminated, or after resource changes;
- Capability exposure for network cloud slicing (allowing openness); with APIs for slice specification and interaction;
- Native Programmability and monitoring control of Network Cloud Slices;
- Optimum and Light Weight Slice lifecycle management including creation, activation, deactivation, protection, elasticity, extensibility, safety;
- Optimum Slice dimensioning;
- Autonomic slice management and operation, namely self-X for slices that will be supported as part of the slice protocols;
- Very large scale slicing with efficient elasticity; and, finally,
- Efficient and large scale slice stitching / composition.

5. References

[BAKLR 2007] BakIr, Gökhan, et al., eds. *Predicting structured data*. MIT press, 2007.

[JMLR 2014] Müller, Andreas C., and Sven Behnke. "Pystruct: learning structured prediction in python." *The Journal of Machine Learning Research* 15.1 (2014): 2055-2060.

[LAFFERTY 2001] Lafferty, John, Andrew McCallum, and Fernando CN Pereira. "Conditional random fields: Probabilistic models for segmenting and labeling sequence data." (2001).

[CPLEX 2019] "**CPLEX Optimizer | IBM**." https://www.ibm.com/analytics/cplex-optimizer. Accessed 9 Sep. 2019.

[DKMMRT 2011] Dutreilh, X., Kirgizov, S., Melekhova, O., Malenfant, J., Rivierre, N., & Truck, I. (2011, May). Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems (pp. 67-74).

[FS 2019] "**Frontline Systems**." https://www.solver.com/. Accessed 9 Sep. 2019.

[HOMMA 2019] S. Homma, H. Nishihara, T. Miyasaka, A. Galis, V. Ram OV, D. Lopez, L. Contreras-Murillo, J. Ordonez-Lucena, P. Martinez-Julia, L. Qiang, R. Rokui, L. Ciavaglia X. de Foy. **Network Slice Provision Models - draft-homma-slice-provision-models-00**, IETF104, 2019. https://tools.ietf.org/html/draft-homma-slice-provision-models-00

[IPSOLVE 2019] "**lpsolve**." http://lpsolve.r-forge.r-project.org/. Accessed 9 Sep. 2019.

[TUSA 2019] F. Tusa, S. Clayman, A. Galis., "Dynamic Monitoring of Data Center Slices," 2019 IEEE Conference on Network Softwarization, Paris, 2019.

[5GEx 2019] 5GEx Project. https://5g-ppp.eu/5gex/.

[MATLAB 2019] "**Optimization Toolbox - MATLAB - MathWorks**." https://www.mathworks.com/products/optimization.html. Accessed 9 Sep. 2019.

[MEDEIROS 2019] Medeiros, A., Neto, A., Sampaio, S., Pasquini, R., & Baliosian, J. (2019). **End-to-end elasticity control of cloud-network slices**. *Internet Technology Letters*, *2*(4), e106.

[MICROSOFT 2019]. Microsoft Excel Solver.https://www.microsoft.com/microsoft-365. Accessed 9 Sep. 2019.

[MNIH 2013] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari With Deep Reinforcement Learning," NIPS Deep Learning Workshop, 2013.

[PASQUINI 2017] Pasquini, R., Moradi, F., Ahmed, J., Johnsson, A., Flinta, C., & Stadler, R. (2017, June). **Predicting SLA conformance for cluster-based services**. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops* (pp. 1-2). IEEE.

[PULP 2019] "**Optimization with PuLP - PythonHosted.org.**" https://pythonhosted.org/PuLP/. Accessed 9 Sep. 2019.

[RABBITMQ 2019] "**RabbitMQ.**". https://www.rabbitmq.com. Accessed 9 Sep. 2019.

[RPROJECT 2019] "**The R Project for Statistical Computing**". https://www.r-project.org. Accessed 9 Sep. 2019.

[ECLIPSE 2019] https://eclipseclp.org. Accessed 10 Set. 2019.

[MINI 2019] https://www.minizinc.org. Accessed 10 Set. 2019.

[ORTOOLS 2019] https://developers.google.com/optimization/. Accessed 10 Sep. 2019.

[TF 2019]. TensorFlow. https://www.tensorflow.org/, Accessed 10 Jul. 2019.

[SCIPY 2019] "**scipy.optimize.linprog — SciPy v1.3.0 Reference - SciPy.org**." https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html. Accessed 9 Sep. 2019.

Version History

Version	Date	Author	Change record
0.1	09.09.2019	Fábio L. Verdi	Creation
0.2	07.10.2019	Fábio L. Verdi	First integrated draft after first internal review
0.3	28.10.2019	Fábio L. Verdi	Final version
0.31	30.10.2019	Joan Serrat	Final check

EUB-01-2017