FEDERATED TEST-BEDS FOR LARGE-SCALE INFRASTRUCTURE EXPERIMENTS
FELIX EU-JP

Collaborative joint research project co-funded by the European Commission (EU)and National Institute of Information and Communications Technology (NICT) (Japan)

Grant agreement no:	608638
Project acronym:	FELIX
Project full title:	"Federated Test-beds for Large-scale Infrastructure eXperiments"
Project start date:	01/04/13
Project duration:	36 months

Deliverable D3.1
Resource Planning and Provisioning

Version 1.0

Due date:	31/11/2014
Submission date:	30/01/2015
Deliverable leader:	i2CAT
Author list:	Roberto Monno (NXW), Gino Carrozzo (NXW), Paolo Cruschelli (NXW), Carolina Fernandez (i2CAT), Carlos Bermudo (i2CAT), Kostas Pentikousis (EICT), Umar Toseef (EICT)

Dissemination level

☑	PU:	Public
☐	PP:	Restricted to other programme participants (including the Commission Services)
☐	RE:	Restricted to a group specified by the consortium (including the Commission Services)
☐	CO:	Confidential, only for members of the consortium (including the Commission Services)

<THIS PAGE IS INTENTIONALLY LEFT BLANK>

Table of Contents

List of Figures

List of Tables

Abstract

This document presents a detailed overview of the software modules used for planning and provisioning the assignation of heterogeneous resources in the FELIX Federated Framework, such as setting up the intra- and inter-domain networking or managing computing nodes. An overview of the design and implementation for the Resource Orchestrator, Computing Resource Manager and Software-Defined Networking Resource Manager modules is given; as well as explaining their key internal and inter-communication workflows.

Excecutive Summary

Deliverable D3.1 aims to explain, from a high-level perspective, the most important concepts and choices made for the design, implementation and deployment of the modules involved in the FELIX resource planning and provisioning. The *Resource Orchestrator (RO)* is introduced as the resource planner and request proxy; the *Computing Resource Manager (C-RM)* provisions computing resources and the *Software-Defined Networking Resource Manager (SDN-RM)* manages OpenFlow-based networking resources.

This document covers also the management of computing and intra-domain networking resources, whilst FELIX D3.3 [1] describes in detail the software components that manage the dynamic inter-domain network connectivity and its stitching with the intra-domain connectivity. The information presented in this document contains the latest additions, yet it is subject to future extension or modification derived from the integration tests and the FELIX Use Cases preparation scheduled for Y3.

We introduce first some concepts and definitions common to the Resource Orchestrator and the two Resource Managers explained in this deliverable. We move afterwards to explaining the particularities of the design and the key internal functionalities, as well as the workflows intended for each module and also its communication with related components. We get into higher detail in the deployment section, where we present succinct guides for the configuration and deployment processes, along with the common operation usage.

This document is addressed to software architects, software and network engineers, software developers implementing specific features of the resource orchestration and provision, as well as system administrators.

1 Introduction

One of the key features of the FELIX Framework with respect to the provisioning and monitoring functionalities is the introduction of a new layer on top of the technology-specific Resource Managers (the modules ultimately in charge of provisioning virtual resources from their physical infrastructure) to cover the functionalities of examining and steering the incoming requests to the most fitted **Resource Manager**, as well as aggregating information of such requests to transmit to another modules available in the upper layers, such as the Monitoring System.

This upper layer contains the **Resource Orchestrator** tool, whose main functions are the following:

- Mediating between the user and the Resource Managers.
- Enforcing the correct workflow followed by the user requests.
- Identifying the appropriate destination to proxy requests from users.
- Maintaining a high-level, cross-island topological view.
- Aggregating high-level information for the status of the physical infrastructure and the virtual resources.

The Resource Orchestrator, located on top of the Resource Managers, acts as the entry point for each domain and attends requests from the users. To this matter, the RO provides the GENI [2] API, widely adopted for testbed federation. The user is then able to communicate against the RO by using a proper client -- for instance the GENI command-line client, OMNI [3].

Upon a request arrival, the RO must ensure that the operation requested by the experimenter complies to the proper workflow, that is, that the sequence of actions to be invoked is correct. If that is not achieved, the user is notified of the error condition. After validating the workflow, this module examines the request to identify the different types of resources the experimenter is able to request. The request is consequently divided into a set of requests, each according to the type of the resource identified. For each of these requests, the RO performs an internal search to identify the Resource Manager that is able to fulfill the request. In the initial stages of the development, the request is either fulfilled locally or, when that is not possible, the user is informed back of the error. Future phases of development are expected to deal with the implementation of an intelligent provisioning schema that evaluates a set of metrics (such as internal policies set by the administrator and/or the status of each domain retrieved from aggregated information in upper levels) and steers requests to Resource Orchestrator in other domains, if need be.

As a part of the mediating procedure, the Resource Orchestrator fetches information from the requests received from the user on the top layer and also from the output (or *manifest*) returned by the Resource Managers in the bottom layer. Such information comprises the physical topology (the physical infrastructure, such as servers or switches) and the slice topology (that is, a subset of those resources that were requested by a user for a given slice). This is useful for maintaining and showing a cross-island view, both for the physical topology and for the contents requested per slice and domain. This information is passed from the RO to the the Monitoring Service module in two ways: the physical topology is periodically pushed, whilst the slice topology is sent as response to a small set of events (e.g. when a resource is effectively provisioned or when it is deleted). The Monitoring System, documented in documented in FELIX D3.2 [4], contains a component to graphically show this monitoring data.

The data persisted in the RO for monitoring must be aggregated in order to compile a high-level view. The **Master Resource Orchestrator** (MRO) is another instance of the RO located in the upper layer of the FELIX Framework and acts as an entry point for a given federation (e.g. a continent), effectively following a hierarchical architecture per group of islands. This is depicted in Figure 1.1. The MRO is capable of aggregating the monitoring information, previously compiled by the ROs in the lower layer, so it is able to make decisions regarding different islands or to communicate the inter-domain topology and slice information to the Master Monitoring System (MMS).

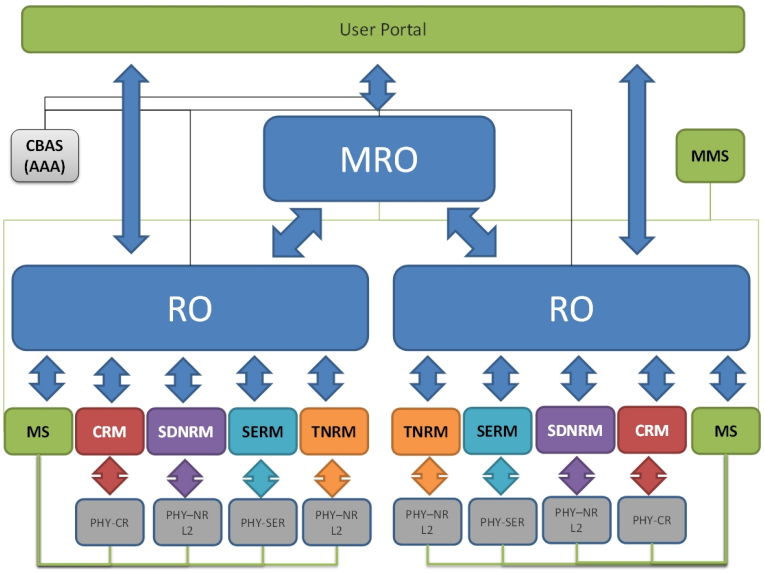

Figure 1.1: Hierarchical Resource Orchestrator in FELIX

Every request is ultimately served by the underlying Resource Managers (RMs), the software modules placed in the lower layers of the architecture and consequently more closely to the hardware that conforms the physical infrastructure. Each RM is able to provision a different kind of resource and this operation is typically performed by offering a virtual instance on top of their physical infrastructure, in a non-exclusive way. In the FELIX Framework, the request performed by the experimenter may consist of computing resources (**Computing Resource Manager**) or Inter-Domain connectivity, either through Software-Defined networks (**SDN Resource Manager**) or through NSI [5] protocol (managed by the Transit Network Resource Manager and helped by the Stitching Entity Resource Manager). In this document we focus on the computing and SDN resources and describe its design, implementation and operational procedures, as well as their communication with the Resource Orchestrator and other modules.

2 Definitions

Throughout this document we use specific notation and acronyms that are explained here. Please refer to this guide to identify the concept or for a more detailed explanation.

2.1 Abbreviations

- **GENI**: Global Environment for Network Innovations.
- **GUI**: Graphical User Interface.
- **MMS**: Master Monitoring System.
- **MRO**: Master Resource Orchestrator.
- **MS**: Monitoring System.
- **OFVER**: OFELIA VERsioning system.
- **PE**: Policy Engine.
- **pyPElib**: python Policy Engine library.
- **RM**: Resource Manager.
- **RO**: Resource Orchestrator.
- **RSpec**: Resource Specification.
- **URN**: Uniform Resource Name.
- **VM**: Virtual Machine.

2.2 Definitions

- **Agent**: Refers to the virtualisation server. This is the software running in each virtualisation server and acting as the entry point to the hypervisor that allows to manage the virtual machines of the users.
- **FlowSpace**: Set of rules to define operations on packets. Contains a variable number of datapath IDs and their selected ports, a filtering condition to match the packets (usually a VLAN or a range of them). This conforms an internal data model of the FlowVisor that is later on inserted on the switches.
- **GENI**: Provides a virtual laboratory for networking and distributed systems research and education, as well as fostering standardization and making the SFA interfaces advance.
- **Island**: Physical domain under particular management. It provides infrastructure and resources to the end user.
- **OFVER**: Versioning system that consists of a number of core scripts to manage the install and update processes, and allows extension through custom scripts.
- **OMNI**: CLI tool which is part of the GENI Control Framework.
- **pyPElib**: Policy Engine library developed in Python. It aims to help programmers using the abstractions provided to apply rule-based policy enforcement.

- **RM**: Software component able to reserve, create, manage and delete resources by communicating with the hardware. It provides interfaces for both administrative and common operations on resources.
- **RSpec**: XML document following agreed schemas to represent resources that are understood by Resource and Aggregate Managers.
- **URN**: Public identifiers given to resources in the network in order to uniquely identify and exhaustively describe the properties of the resource. For that, the *urn* scheme is followed.

3 Implementation Details

Based on the general FELIX architecture defined in deliverable D2.2 [6], the project team has progressed with the more detailed design and implementation of the identified modules. In this section we explain the most relevant implementation aspects of the modules that are related to the provisioning procedure, and particularly the Resource Orchestrator and the Resource Managers **C-RM** and **SDN-RM**, respectively based on the OFELIA VTAM and OFAM [7].

The **Resource Orchestrator** is in charge of the part of the resource planning, as it interprets incoming requests and steers portions of it to the appropriate Resource Manager, from the pool of RMs available on its same island. Upon receiving the corresponding request, the C-RM fulfils the part of the request related to the virtualisation and the SDN-RM manages the part related to SDN connectivity.

We explain in the next sections the design of each of these modules, their workflow and communication with other modules, as well as the actions to take in the future.

3.1 Resource Orchestrator

The Resource Orchestrator (RO) module has been developed from scratch to fulfil the FELIX requirements, which have been documented in the FELIX D2.2 deliverable.

We recall the major key functionalities that the RO needs to provide as follows:

- Manage different kind of resources (computing, networking, etc) and allow data access policies, for instance through the AAA system (user's identity and permissions) or through domain specific restrictions.
- Mediate between the user/experimenter and the technology-specific Resource Managers (RMs) in order to reserve, provision, monitor, release and operate on both resources and slices.
- Maintain a high-level and cross-island topological view (updated by the underlying RMs) for better decision making.
- Manage end-to-end services spawned on the federated testbed and coordinate the correct sequence of actions to instantiate the service.
- Interact with the Monitoring System to provide measurements, statistics and resources provisioned per slice, as well as to provide such System with the overview of the physical topology.

The following sections describe the technical design of the RO, which has been inspired by the previous requirements. In addition, we present some internal workflows that explain how the basic operations are performed inside this module and also document in a simple manner the interaction between the RO and other modules from the FELIX project that are related to it.

3.1.1 Design

The Resource Orchestrator module is structured as a number of components or building blocks. This is so in order to keep a maintainable and extensible structure (i.e. through means of plug-ins). Figure 3.1 shows a brief high-level overview of RO's design model, which has been followed during the development process.

3.1.1.1 Building blocks

The different building blocks are logically interconnected to provide the required functionalities introduced previously. From a high level perspective, the RO module consists of three main layers:

North-bound Interfaces

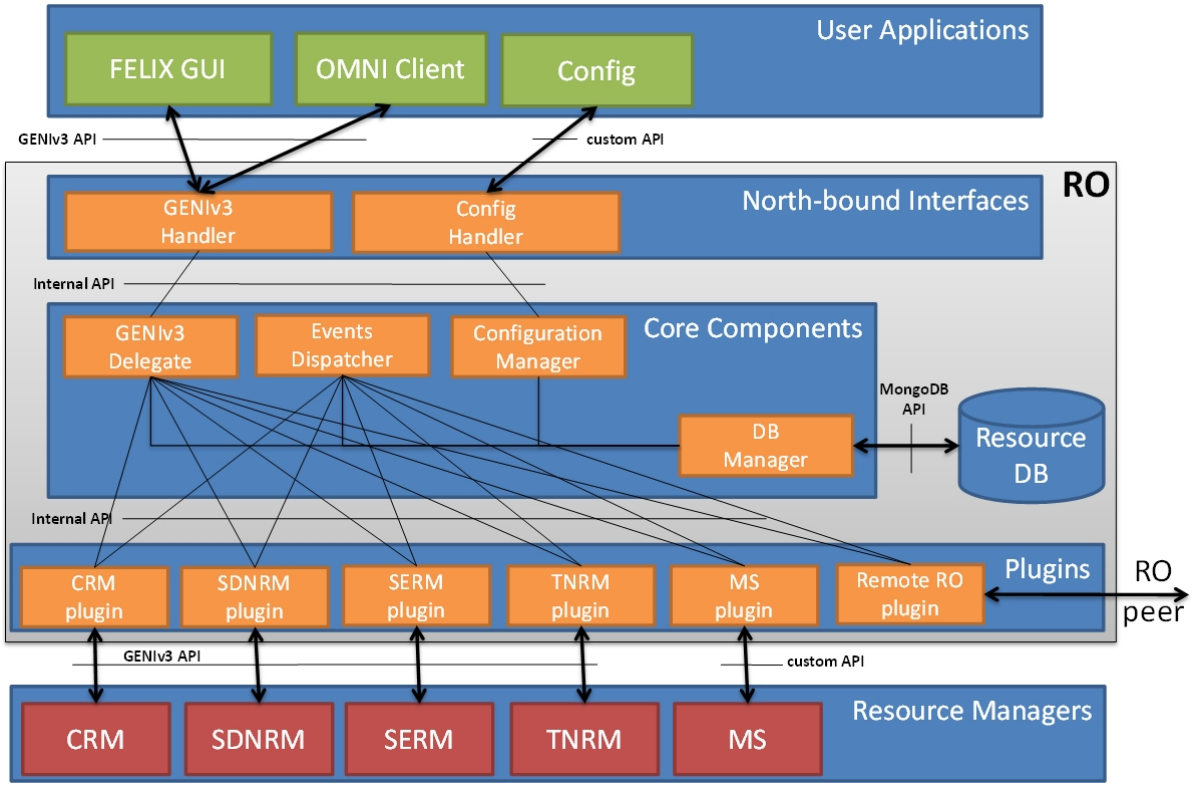

Figure 3.1: RO Design Model

The top layer consists of components that manage the external north-bound interfaces of the RO. That is, the *GENIv3* interface that provides methods for the resources provisioning and its *custom API* that has been developed to allow the configuration of the internal components.

- GENIv3 Handler: receives the (XML) message, translates the message body and builds the proper method signature. Then, it calls the GENIv3 Delegate to execute the command and converts the results into the (XML) message to be send back to the caller.
- Config Handler: manages the configuration messages and invokes the Configuration Manager with the incoming parameters.

Core Components

The middle layer is the main framework which is in charge of e.g. manage the user requests, store the discovered physical topology or generate events for resources realignment, etc.

- GENIv3 Delegate: executes the GENIv3 methods retrieving data from the database, calling the RM plug-ins, aggregating the results, etc. In brief, this is the real core of the RO.
- Events Dispatcher: collects the events generated by the other components and then schedules the execution through the proper plug-in.
- Configuration Manager: configures the internal components and (just in case) stores the data into the database (e.g. intra-island RMs configuration details).
- DB Manager: provides a wrapper to the MongoDB [8] API and introduces dedicated filters for the FELIX resources.

Plug-ins

The bottom layer consists of some plug-ins which take care of the communication with the corresponding Resource Managers.

- C-RM plug-in: interacts with the Computing Resource Manager.
- SDN-RM plug-in: interacts with the SDN Resource Manager.
- SE-RM plug-in: interacts with the Stitching Entity Resource Manager.
- TN-RM plug-in: interacts with the Transport Network Resource Manager.
- MS plug-in: interacts with the Monitoring System.
- Remote RO plug-in: interacts with the Remote Resource Orchestrator to implement the top-down hierarchical (management) approach.

3.1.1.2 Exposed interfaces

The RO module exposes a *GENIv3* [9] compliant interface which can be used by the user applications (e.g. the FELIX-GUI) to allocate, configure, describe or release resources into the federated test-bed. It also provides a *custom API* used for configuration purposes.

GENIv3 API

Each incoming GENIv3 request is received by the **GENIv3 Handler** and then forwarded to the **GENIv3 Delegate**. This component performs all the authentication and authorization checks and then validates the received RSpec with a proper schema. Depending on the input parameters, the delegate chooses the correct plug-in (or plug-ins) that should serve the request and translates the context to be executed into the underlying layer. In addition, the delegate can store data into the **Resource DB** simply calling a dedicated **DB Manager** object which can be considered as a DB abstraction layer.

We have several plug-ins to interact with different Resource Managers. Basically, there is a 1:1 mapping between the RM and the corresponding plug-in. This mechanism allows us to isolate the specific RM management *logic* inside the plug-in and to offer a common and generalized interface to the upper layer. The methods offered by this API are as follows:

- **GetVersion**: return basic information of the RO, such as the format of the supported RSpecs.
- **ListResources**: retrieve description of the available resources that are managed by the RO.
- **Allocate**: perform a reservation request of a set of any kind of resources managed by the RO. Each request follows the specification expected for the RO attending the request (see each RM for details on the data structure).
- **Provision**: steer the provisioning request to each RM attending the resources that were previously reserved.
- **Status**: check status of the current reservation.
- **PerformOperationalAction**: given the identifier of a resource, this method performs a *start*, *stop* or *restart* operation on it by contacting the appropriate RM.
- **Renew**: extend expiration time for the given resource(s). This can also be used to extend the reservation time for resources previously provisioned.
- **Delete**: deletes the resource(s) identified by the URN [10]. Both allocated and provisioned resources are affected by this.

RSpecs

Here below we provide an example of a *request RSpec* that can be used to perform an Allocation operation on specified SDN and TN resources. A detailed description on the contents expected for each of the aforementioned resources can be found in the SDN-RM section of this document and in the TN-RM section in D3.3 [1], respectively. The proposed RSpec can be considered as an *aggregate* of the computing, SDN and transport network resources that the user can choose to create a virtual *slice* into the provided large-scale environment. It is worth noting that the experimenter's request will induce the RO to automatically select the Stitching Entity resources to create a virtual link between the SDN datapaths (intra-island) and TN nodes (inter-island).

```
<?xml version="1.0" encoding="UTF-8"?>
<rspec  xmlns="http://www.geni.net/resources/rspec/3"
        xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:openflow="http://www.geni.net/resources/rspec/ext/openflow/3"
        xmlns:sharedvlan="http://www.geni.net/resources/rspec/ext/shared-vlan/1"
        xs:schemaLocation="http://www.geni.net/resources/rspec/3/request.xsd
                           http://www.geni.net/resources/rspec/ext/openflow/3/of-resv.xsd"
        type="request">
   <openflow:sliver email="a@b.com" description="OF request example">
       <openflow:controller url="tcp:10.216.12.134:6633" type="primary"/>
       <openflow:group name="fs1">
           <openflow:datapath
                  component_id="urn:publicid:IDN+openflow:i2cat+00:10:00:00:00:00:00:01"
                  component_manager_id="urn:publicid:IDN+openflow:i2cat.ofam+cm"
                  dpid="00:10:00:00:00:00:00:01">
               <openflow:port name="GBE0/3" num="3"/>
               <openflow:port name="GBE0/12" num="12"/>
           </openflow:datapath>
           <openflow:datapath
                  component_id="urn:publicid:IDN+openflow:i2cat+00:10:00:00:00:00:00:03"
                  component_manager_id="urn:publicid:IDN+openflow:i2cat.ofam+cm"
                  dpid="00:10:00:00:00:00:00:03">
               <openflow:port name="GBE0/1" num="1"/>
               <openflow:port name="GBE0/12" num="12"/>
           </openflow:datapath>
       </openflow:group>
       <openflow:match>
           <openflow:use-group name="fs1" />
           <openflow:packet>
               <openflow:dl_vlan value="18" />
           </openflow:packet>
       </openflow:match>
       <openflow:match>
           <openflow:datapath
                  component_id="urn:publicid:IDN+openflow:i2cat+00:10:00:00:00:00:00:05"
                  component_manager_id="urn:publicid:IDN+openflow:i2cat.ofam+cm"
                  dpid="00:10:00:00:00:00:00:05">
               <openflow:port name="GBE0/7" num="7"/>
               <openflow:port name="GBE0/8" num="8"/>
           </openflow:datapath>
           <openflow:datapath
                  component_id="urn:publicid:IDN+openflow:i2cat+00:10:00:00:00:00:00:02"
                  component_manager_id="urn:publicid:IDN+openflow:i2cat.ofam+cm"
                  dpid="00:10:00:00:00:00:00:02">
               <openflow:port name="GBE0/4" num="4"/>
               <openflow:port name="GBE0/15" num="15"/>
           </openflow:datapath>
           <openflow:packet>
               <openflow:dl_vlan value="1234" />
           </openflow:packet>
       </openflow:match>
   </openflow:sliver>
   <node client_id="urn:publicid:tn-network1"
         component_manager_id="urn:publicid:IDN+NSI+authority+TN-RM">
       <interface client_id="urn:publicid:tn:aist:network1+urn:ogf:network:aist:network1:stp1">
           <sharedvlan:link_shared_vlan
               name="urn:publicid:tn:aist:network1+urn:ogf:network:aist:network1:stp1+vlan"
               vlantag="1980-1989"/>
       </interface>
       <interface client_id="urn:ogf:network:xxx:stp1"/>
       <interface client_id="urn:ogf:network:yyy:stp2"/>
```

```
        <interface client_id="urn:publicid:tn-network1+urn:felix:i2cat-stp2">
            <sharedvlan:link_shared_vlan
                name="urn:publicid:tn-network1+urn:felix:i2cat-stp2+vlan"
                vlantag="1980-1989"/>
        </interface>
    </node>
    <link client_id="urn:publicid:tn-network1:link">
        <component_manager name="urn:publicid:IDN+NSI+authority+TN-RM"/>
        <interface_ref
          client_id="urn:publicid:tn:aist:network1+urn:ogf:network:aist:network1:stp1"/>
        <interface_ref client_id="urn:ogf:network:xxx:stp1"/>
        <interface_ref client_id="urn:ogf:network:yyy:stp2"/>
        <interface_ref
          client_id="urn:publicid:tn-network1+urn:felix:i2cat-stp2"/>
        <property source_id="urn:publicid:tn:aist:network1+urn:ogf:network:aist:network1:stp1"
                  dest_id="urn:publicid:tn-network1+urn:felix:i2cat-stp2"
                  capacity="1000">
        </property>
        <property source_id="urn:publicid:tn-network1+urn:felix:i2cat-stp2"
                  dest_id="urn:publicid:tn-network1+urn:ogf:network:aist:network1:stp1"
                  capacity="500">
        </property>
    </link>
</rspec>
```

3.1.1.3 Custom API

The RO also offers a dedicated *custom API* that is used to configure (at start-up or during the life-cycle of the process) the internal RO details, e.g. the list of the Resource Managers that belong to the same island, their IP addresses and port numbers, the supported protocol, the provided endpoints of the resources, etc.

3.1.1.4 Synchronisation

This module periodically persists a high-level view of the physical topology managed by it. This data is used both internally (to provide an up-to-date list of the available resources to the experimenter) and also transmitted to different modules, such as the Expedient [expedient] and Monitoring System to enable it to graphically show this data.

The RO manages this recurrent tasks through both *one-shot* and *periodical tasks*, for example to realign the discovered topology with all the underlying Resource Managers, to create a channel for the communication with the Monitoring System, to release pending resources, etc. The first tasks are run once the RO module is started, whereas the periodical jobs are triggered by an underlying daemon devoted to this specific matter.

3.1.2 Workflows

This section describes the workflows that the RO follows through its GENIv3 API in order to perform basic operations, such as listing, allocating, managing or deleting resources. The workflow expected by this API is defined in the GENIv3 API Common Concepts site [11].

3.1.2.1 Listing the resources

The RO is configured to use an *event scheduler* for executing actions at specified intervals or times of the day. Basically, when a timeout expires, the RO asks each Resource Manager the list of their physical resources and then inserts this data into a non-relational database. The GUI (or any other GENIv3 compliant client, for that matter) can retrieve the list of resources using the **ListResources** method. Upon receiving this request, the RO retrieves the resources from its database and translates the entries into a proper RSpec, called *advertisement Rspec*, that is defined as a well-known and standardized schema. The response can be considered just an *aggregation* of all the discovered resources at the RM layer. Figure 3.2 shows the ListResources workflow (this sequence diagram and any other appearing in this document are generated using www.websequencediagrams.com).

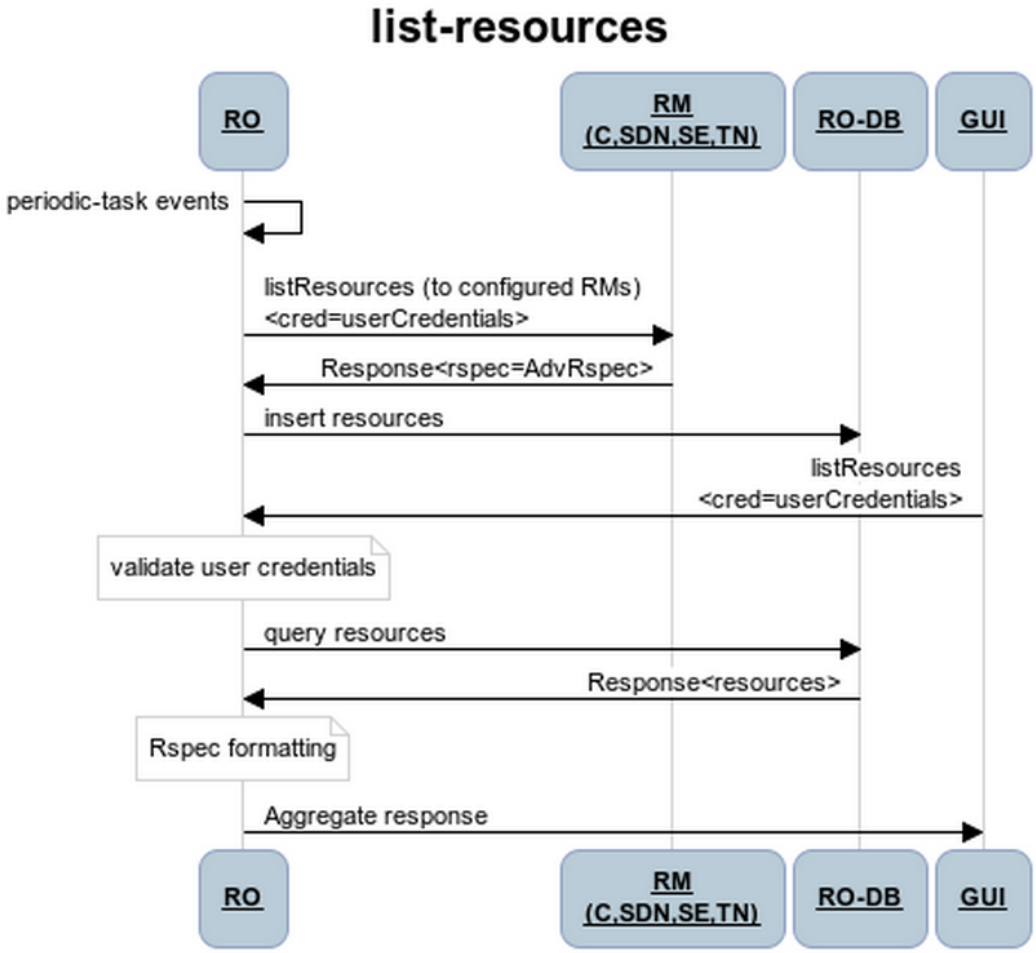

Figure 3.2: RO list-resources workflow

3.1.2.2 Reserving the resources

When the **Allocate** request is received, the RO validates the credentials of the experimenter and then parses the XML request RSpec. The resources are then retrieved from the databases in order to fetch the Resource Manager(s) to be contacted. Indeed, several allocate requests can be generated to fulfil the incoming request. At the end, the RO replies with the list of reserved resources identified by their URNs, in the form of an aggregated response. See workflow in Figure 3.3.

3.1.2.3 Provisioning the resources

When the RO receives a **Provision** request, the user credentials are validated first and then, using the slice URN input parameter, this module retrieves all the resources involved into the reservation that was previously performed (through the Allocate method) for the specified slice. Once the affected RMs are identified, the provision request is forwarded to each of them with the proper resource identifiers. The workflow is finalised once each aggregate returns their response to this method, in the form of a *manifest RSpec*. This method triggers an event so that RO informs the Monitoring System to start collecting meters for all the provisioned slice resources. Refer to Figure 3.4 for further details.

3.1.2.4 Deleting the resources

The workflow for the **Delete** method (Figure 3.5) is very similar to the previous one, as it only needs to send the proper resource identifier. The only difference is in the trigger event: on deletion, the RO informs the Monitoring

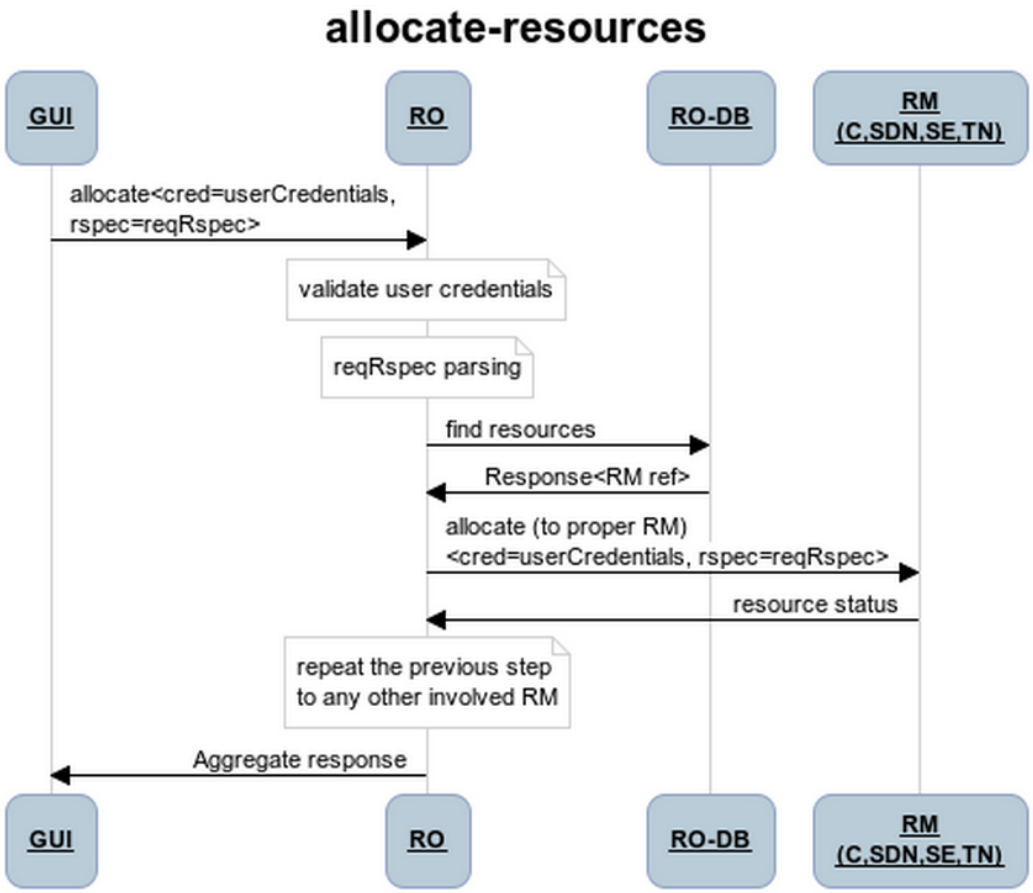

Figure 3.3: RO allocate workflow

System to stop the monitoring procedure and to clean all the data associated to the corresponding slice (URNs).

It is important to note that the workflows for the other operations (i.e. **Status**, **Describe**, **PerformOperationAction** and others) follow similar steps as the ones here exposed. This allows parts of the implementation to be reused in different contexts, sticking to software reuse precepts for aiming to save time and resources and reducing the redundancy.

3.1.3 Future Work

The RO software module is one of the new building blocks devised in the FELIX Architecture and has been developed from scratch during the Y2 of the project. This implies that the remaining effort will be spent to finish the integration with other software components and to refine and rearrange the code during the integration tests to be performed before running the FELIX Use Cases defined in the D2.1 [12] deliverable.

It is noteworthy mentioning that a fruitful bug fixing stage will take place whilst adopting the large-scale FELIX facility as a real, heterogeneous and distributed testing environment. Furthermore, future releases will cover more advanced features, the following among then:

- Aggregate data regarding physical topology and slice information, previously retrieved from other Resource Orchestrators, in order to communicate it to the Master Monitoring System.
- Perform validation of the credentials of the user request prior to performing the requested operations.

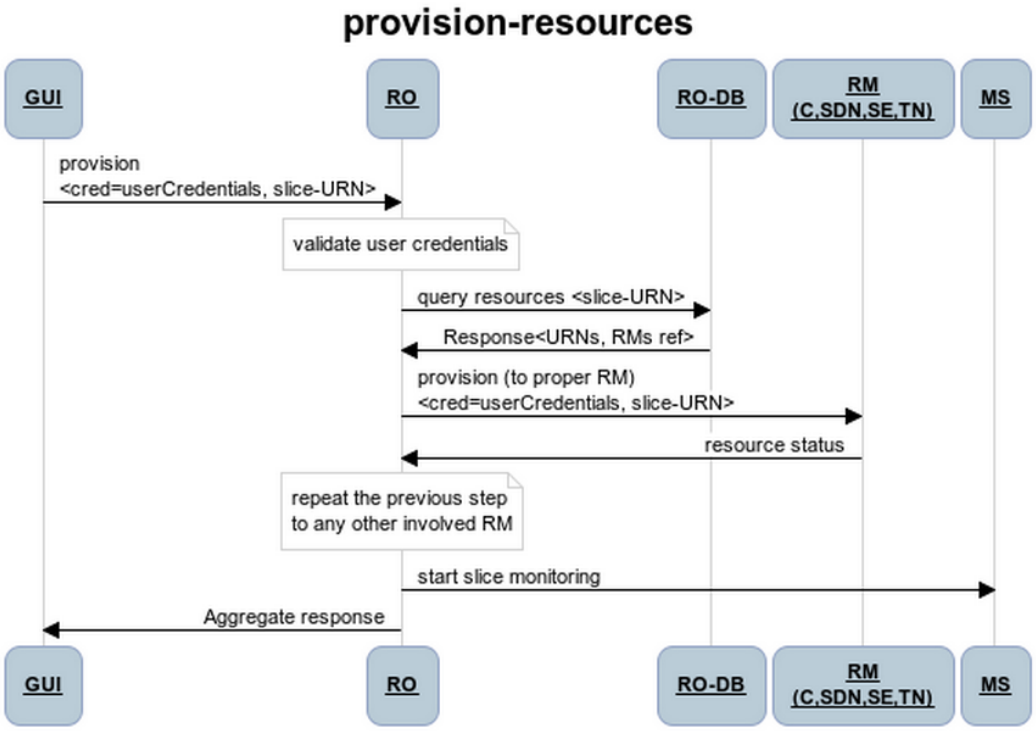

Figure 3.4: RO provisioning workflow

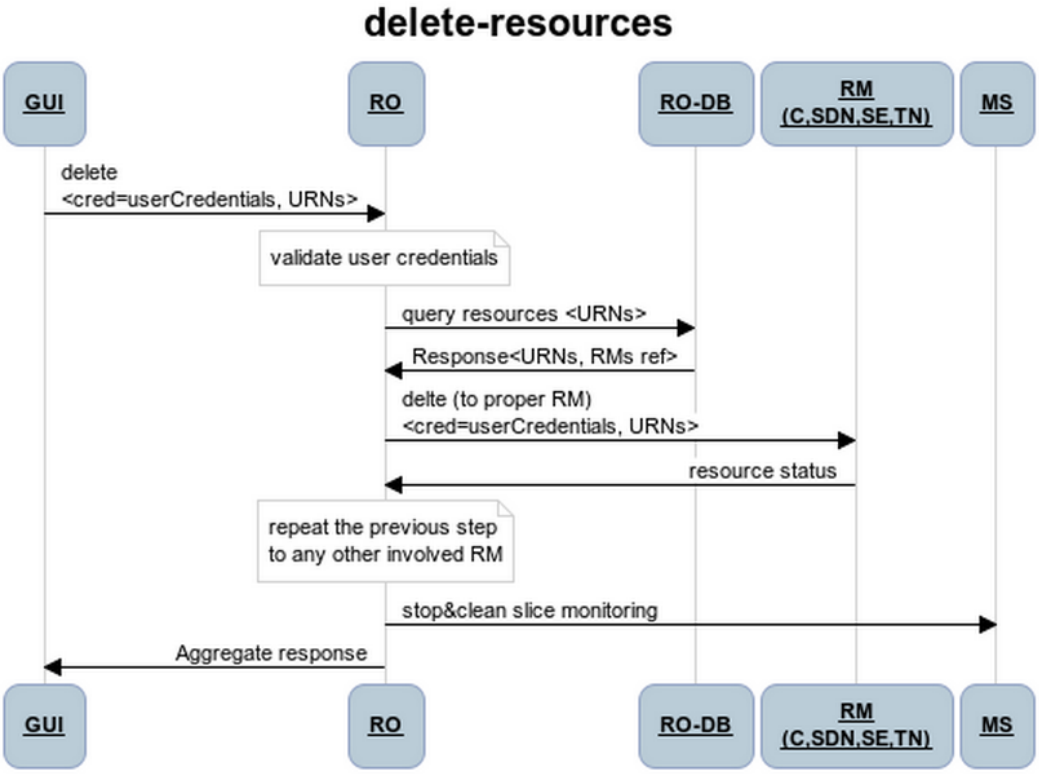

Figure 3.5: RO delete workflow

3.2 Computing Resource Manager

The Computing Resource Manager module (C-RM) is based on the OFELIA Virtualisation Technology Aggregate Manager. The C-RM allows an experimenter to provision and manage VMs on different physical servers.

The components and their functionality are explained in the *Design* section. There is some extra implementation that extends the functionality of the C-RM in order the requirements from FELIX:

- Adopting the latest GENI testbed federation API (GENIv3) to allow allocation prior to provisioning.
- Allowing user SSH keys contextualisation in the Virtual Machines to enable direct access through public keys.

The latest additions collaborates in standardizing the access to the Resource Managers of the FELIX Framework and to the resources provided by them; not only internally but also looking to possible inter-testbed federations.

This module contains a number of components and subcomponents that work together in order to reserve, create, manage and delete the requested Virtual Machine resources.

3.2.1 Design

Building blocks

The components of this module could be grouped into 1) a core in charge of background operations, also exposed through a series of APIs, 2) the Agent that is present in the virtualisation server and interacts with the core of the C-RM, 3) the Policy Engine used by the core to evaluate requests prior to provisioning and 4) a web-based GUI to allow easier management for experimenters and administrators. These components and some other subcomponents are depicted in Figure 3.6.

The core implements the basic functionality through a number of components (e.g. the Dispatcher of the different actions and responses, parsers and crafters, etc). Some of the functionalities provided by them are indirectly exposed through a series of northbound APIs, namely GENIv3, GENIv2 and OFELIA custom APIs. The first two are widely adopted in federated testbeds; being the first the latest version and allowing both allocation and provisioning of resources. Internally, the requests may traverse different components and behave differently, depending on the entry point (i.e. whether the user request was generated via the GUI or using some CLI to access the exposed APIs). After persisting the information in the database and provisioning some associated resources (e.g. such as MAC and IP), the core communicates with the agent in the desired server.

The Agent, on the other hand, acts as an interface for managing Virtual Machines on the desired physical server or host. This means the Agent is able to communicate with the hypervisor that is running on the physical virtualization server, thereby being able of managing the creation, deletion or management of the guest machine. After a VM is instantiated, a user may be able to authenticate on the machines either through the LDAP module or by contextualizing the user's SSH keys into the Virtual Machine. The description on how a user may provide his public SSH key for this purpose follows in deliverable D3.4 [13]

The Policy Engine evaluates every incoming request against a list of rules or restrictions determined by the administrator of each domain. Such evaluation is performed sequentially in order to validate against the first matching rule or, alternatively, exit if none of them matched. Before the VM can be effectively provisioned and the request sent to the Agent, the request must comply with the established restrictions.

The last component, the web GUI of the C-RM, allows the administrator to manage virtualized resources in their domain in a clear way; so that the manager can see the status of every VM per server, manage the ranges of IPs and MACs, define restrictions and so on.

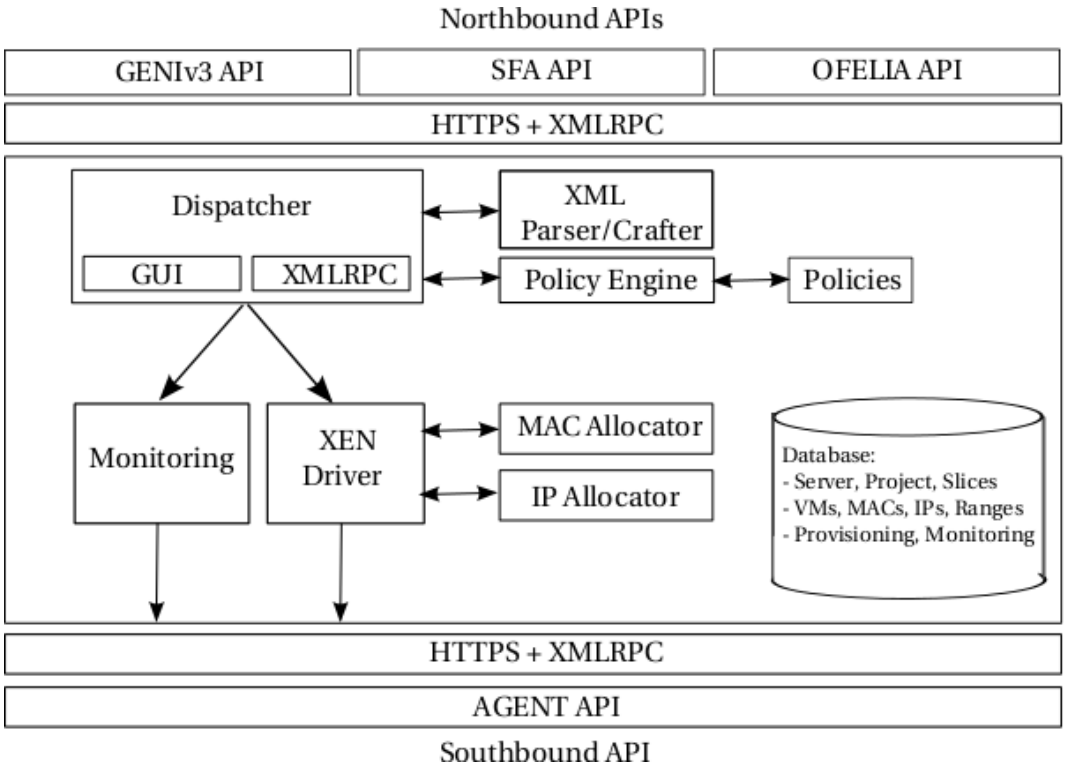

Figure 3.6: Components of the FELIX C-RM

3.2.1.1 Communication between blocks

Once a user submits a request, it is passed down; depending on the user being an experimenter or an administrator. For the typical case of the creation of a VM, the experimenter access through the GUI to ask for a VM. This request is intercepted by the Dispatcher component. The data of the request is then marshalled and the specific conditions set by the requester (e.g. quantity of memory, template name) are evaluated by the Policy Engine. The IP and MAC associated to the VM are calculated and inserted into the request. After this, the request is sent over the network to the Virtualisation Agent, through its southbound, custom API. Upon receiving the request, the Agent interprets the request and communicates with the Virtualisation Hypervisor in order to create the VM. When the process is finished and the VM is created, the Agent sends a notification back to other subcomponents involved on this (the GUI and the administrative panel of the C-RM).

The generated VM will be eventually accessed by an experimenter. In the original version of the C-RM, the access to the VM is supported through basic authentication (user and password). This has been extended for FELIX so that the VMs generated through the GENIv3 can be accessed via public keys, in compliance with SFA [14].

3.2.1.2 Exposed interfaces

As one of the FELIX Resource Manager, this module offers two well-known APIs (see GENI AM API [15]) that enable a programmatic, abstracted and standardized use of any resource in the testbed. These standard interfaces were designed initially by SFA and have been since revisited, standardized and widely adopted by other infrastructures.

The main benefit of using such interfaces is to allow easier federation between testbeds; that is, the sharing of resources of different kinds and offered by different providers; but also serve for exposing a standard interface that can help automation and abstraction when it comes to reserve, provision or scheduler resources.

GENIv2 API

The GENIv2 API is the previously accepted interface and is still supported nowadays by several infrastructures.

Through a simple workflow, the more important methods and their functionality can be observed:

- **GetVersion**: learn basic information about the C-RM, such as the format of the supported RSpecs.
- **ListResources**: retrieve description of available servers and their links to the switches in the same island.
- **CreateSliver**: provision and initialize a Virtual Machine, according to an RSpec.
- **SliverStatus**: check status of the VM (sliver).
- **RenewSliver**: extend expiration time for the VM.
- **DeleteSliver**: when done, delete the VM.

GENIv3 API

This is the latest interface adopted by the GENI and other testbeds community. Compared to the previous interface, this allows new operations such as the Allocation (a reservation, prior to the effective provisioning of the resource), or the PerformOperationalAction that allows extending operations on a resource to include any extra functionality. The main methods are as follow:

- **GetVersion**: learn basic information about the C-RM, such as the format of the supported RSpecs.
- **ListResources**: retrieve description of available servers and their links to the switches in the same island.
- **Allocate**: request an (incremental) reservation of a subset of resources (VMs), according to an RSpec.
- **Provision**: effectively provision and get ownership of a subset of previously allocated VMs.
- **Status**: check the status of the VM(s) previously reserved or provisioned.
- **PerformOperationalAction**: a resource (or a number of resources) can be started, stopped or restarted. Furthermore, this module implements the optional operation methods to update the SSH keys of the users on a VM and to retrieve the access information on the given Virtual Machine.
- **Renew**: extend expiration time for the given VM or VMs. This can also be used to extend the reservation time for a VM.
- **Delete**: when done, delete one or more VMs. A reservation can also be deleted with this command.

RSpecs

The request RSpec passed when performing an Allocation operation must comply with a format similar to the one in the sample below:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <rspec type="request"
      xsi:schemaLocation="http://www.geni.net/resources/rspec/3
                          http://www.geni.net/resources/rspec/3/request.xsd"
      xmlns="http://www.geni.net/resources/rspec/3"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:emulab="http://www.protogeni.net/resources/rspec/ext/emulab/1">
    <node client_id="Verdaguer"
        component_id="urn:publicid:IDN+ocf:i2cat:vtam+node+Verdaguer"
        component_manager_id="urn:publicid:IDN+ocf:i2cat:vtam+authority+cm"
        exclusive="true">
      <sliver_type name="emulab-xen">
        <emulab:xen cores="10" ram="8192" disk="50"/>
        <disk_image name="urn:publicid:IDN+emulab-ops//DEB60_64-VLAN"/>
     </sliver_type>
    </node>
</rspec
```

As observed in the sample RSpec, the data being transmitted with this method must contain at least the following information to identify the server containing the requested machine: 1) the *client ID* or name of the requested virtual machine, 2) the *component ID* or URN of the virtualisation server, and 3) the *component manager* or URN of the authority of such server, that is, the RM itself.

To better specify the requirements of the VM being requested, the *sliver type* (type of virtualisation), *disk image name* (URN of the selected template or flavour) and other parameters such as the RAM or disk size can be passed.

3.2.2 Workflows

In this section we identify the most common operations for the C-RM and describe their workflow through sequence diagrams and flow charts.

3.2.2.1 Requesting a new VM

The experimenter may ask for one or multiple VMs to be generated by interacting with the Expedient GUI or by directly contacting the CLI. Depending on the entry point for the generation of the virtual machine through the C-RM, it may be necessary to request one by one and start each VM afterwards (when requesting VMs through Expedient) or it could be possible to provision multiple VMs at once and access them right after, when using the GENIv3 CLI.

The differences between the workflows that are them is explained by the fact that GENIv3 establishes a set of standard workflows that are different to previous implementations, more focused on usability and in minimising resource usage.

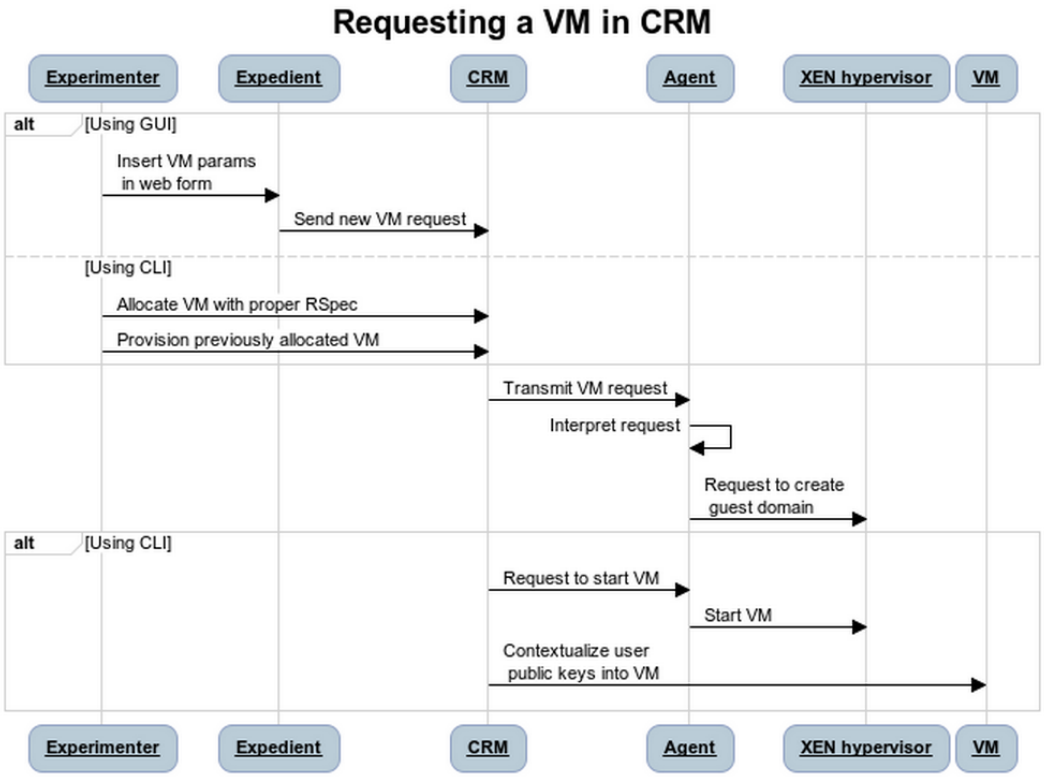

Figure 3.7: C-RM - Requesting a Virtual Machine

3.2.2.2 Accessing a VM

After the VM is provisioned, it is automatically started and the experimenter is able to access through the SSH protocol. The authentication and authorisation procedures may vary depending on the way the machine was generated. Summing up, the following workflow takes place:

- **Accessing a VM created through Expedient**
  1. SSH to the machine with the Expedient username.
  2. The LDAP PAM modules present in the VM contrast the user credentials against the ones stored in the LDAP.
  3. On match, the user is authorised and able to enter the VM.
- **Accessing a VM provisioned through CLI**
  1. SSH to the machine with the username associated to the credentials passed to C-RM.
  2. A matching between the local private key and the remote public key is performed.
  3. On match, the user is authorised and is able to enter the VM.

It is worth noting here that, once the Expedient, C-RM and AAA are fully integrated; the public key associated to the certificates of the experimenter will be passed to the VM on creation time. When accessing the VM, the experimenter shall be able to follow the same steps as for a VM provisioned through CLI.

3.2.2.3 Deleting a VM

When the experimenter does not need to use a particular Virtual Machine any more, it is time to delete it. Machines generated through the GUI must be deleted one by one, while the VMs created using the GENIv3 CLI can be deleted at once.

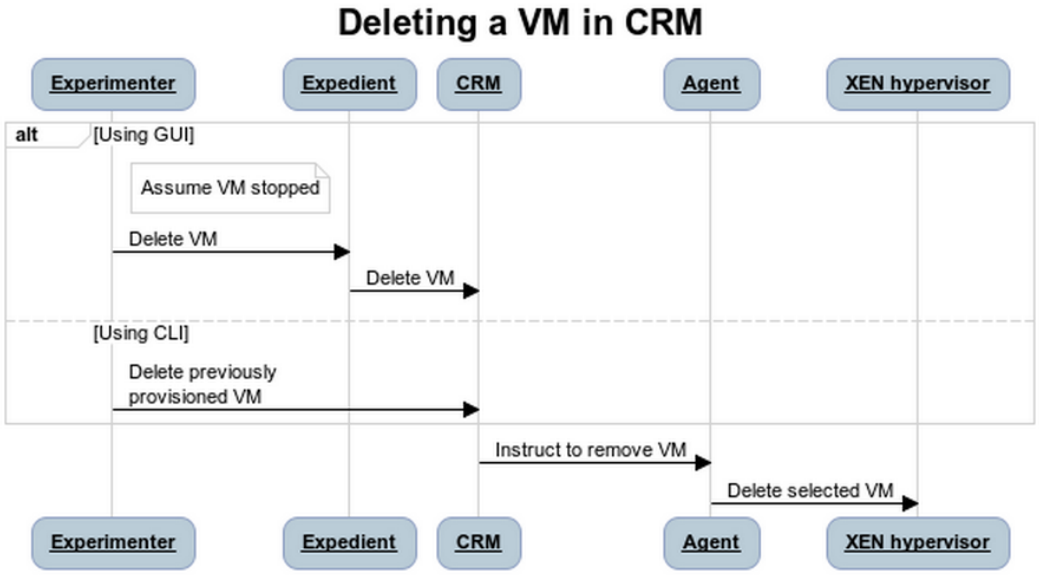

Figure 3.8: C-RM - Deleting a Virtual Machine

3.2.3 Future Work

While this module already fulfills its requirements, the following features could be contemplated for future releases:

- Advertise every available kind of templates present at a given server, when the GENI's "ListResources" method is called.
- Enable the instantiation of a VM with any kind of available template through the GENI interfaces.
- Contact the Monitoring Service to retrieve metrics on physical servers and show these on the administration panel.

3.3 Software-Defined Networking Resource Manager

The Software-Defined Networking Resource Manager (SDN-RM) module is based on the OFELIA Software-Defined Aggregate Manager, which is in turn based on Stanford's Opt-in Manager. The SDN-RM allows an experimenter to request, update and delete OpenFlow resources.

The SDN-RM module offers similar northbound APIs as the C-RM, that is, the GENIv3, GENIv2 and OFELIA custom APIs. As introduced before, the GENI APIs are widely adopted in testbeds. The request here may also traverse different components and behave differently, depending on the entry point. This module has the particularity of acting as a proxy between the experimenter and the domain controller (here, FlowVisor); so the latter can operate on the request and communicate it to the controller, in charge of inserting the pertinent OpenFlow rules into the SDN-enabled switches.

The SDN-RM has also been further extended to meet requirements from FELIX. This extension comprises:

- Adopting the latest GENI testbed federation API (GENIv3) to allow allocation prior to provisioning.
- Implement an SDN-RM plug-in for the GUI that eases the process of selection of a VLAN for the slicing of experiments.

3.3.1 Design

3.3.1.1 Building blocks

The SDN-RM can be roughly divided into 1) its core functionality, 2) the web-based GUI and 3) the proxy class to communicate with the FlowVisor module.

The core receives requests for FlowSpaces from the Expedient GUI or through the GENI APIs. Internally, it parses those requests, translates them to its internal information model, fills any missing information required by the FlowVisor module and sends a properly formatted request to the aforementioned module by instantiating the proxy class that is able to talk with the FlowVisor API.

The web GUI features the main function of approval/denial of FlowSpaces (either after manual inspection of the administrator or automatic; when possible) and also provides configuration options and basic monitoring of the currently approved FlowSpaces per slice and the rules contained on them.

3.3.1.2 Communication between blocks

An incoming request for a FlowSpace is passed down to the SDN-RM. This validates the type of the data against the expected format as a first step, then checks that there are no similar slices or rules already provisioned. The data is initially persisted in its database as an *Experiment Flowspace* and identified as an *Opt-in FlowSpace* once it has been granted. After the input data is validated and persisted, the request is ready to be sent to the FlowVisor controller.

This controller is in charge of interacting with the switches to input the OpenFlow rules selected by each user for their own experiment, as well as being able to steer traffic according to the user controller available per slice. This is, in the end, the client that speaks the OpenFlow protocol to the switches and that enables proper transmission of packets along the SDN network.

3.3.1.3 Exposed interfaces

The SDN-RM inherits its base from the OFELIA OFAM, which exposes the GENIv2 interface. Now, as any other FELIX Resource Managers, this module also offers the GENIv3 API; which enables a programmatic, abstracted and standardized use of any resource in the testbed. This -as the previous standard interfaces- were designed initially by SFA and have been since revisited, standardized and widely adopted by other infrastructures.

The main benefit of using such interfaces is to allow easier federation between testbeds; that is, the sharing of resources of different kinds and offered by different providers; but also serve for exposing a standard interface that can help automation and abstraction when it comes to reserve, provision or scheduler resources.

GENIv2 API

The GENIv2 API is the previously accepted interface and is still supported nowadays by several infrastructures. Through a simple workflow, the more important methods and their functionality can be observed:

- **GetVersion**: learn basic information about the SDN-RM, such as the format of the supported RSpecs.
- **ListResources**: retrieve description of the topology (switches and their interconnections).
- **CreateSliver**: provision and initialize a set of datapath IDs and conditions (called FlowSpace), according to an RSpec.
- **SliverStatus**: check status of the FlowSpace (sliver).
- **RenewSliver**: extend expiration time for the FlowSpace.
- **DeleteSliver**: when done, delete the FlowSpace.

GENIv3 API

This is the latest interface adopted by the GENI and other testbeds community. Compared to the previous interface, this allows new operations such as the Allocation (a reservation, prior to the effective provisioning of the resource), or the PerformOperationalAction that allows extending operations on a resource to include any extra functionality. The main methods are as follow:

- **GetVersion**: learn basic information about the SDN-RM, such as the format of the supported RSpecs.
- **ListResources**: retrieve description of the topology (switches and their interconnections).
- **Allocate**: request reservation of a subset of resources (VMs), according to an RSpec. In contrast with C-RM, this is not incremental: a new allocation does not extend previous reservations.
- **Provision**: effectively provision and get ownership of a subset of previously allocated FlowSpaces.
- **Status**: check status of the reservation or the FlowSpace.
- **PerformOperationalAction**: the resource can be started, stopped or restarted. When stopped, the FlowSpace is removed from the FlowVisor controller (i.e. connectivity if effectively disabled). A FlowSpace that has been previously granted is ensured to be automatically granted on subsequent starts or restarts.
- **Renew**: extend expiration time for the FlowSpace. This can also be used to extend the reservation time for a FlowSpace.

- **Delete**: when done, delete the FlowSpace. A reservation can also be deleted with this command.

RSpecs

The request RSpec passed when performing an Allocation operation must comply with a format similar to the one of the following sample:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <rspec type="request"
      xs:schemaLocation="http://www.geni.net/resources/rspec/3
                         http://www.geni.net/resources/rspec/3/ad.xsd
                         http://www.geni.net/resources/rspec/ext/openflow/3
                         http://www.geni.net/resources/rspec/ext/openflow/3/of-ad.xsd"
      xmlns="http://www.geni.net/resources/rspec/3"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:openflow="http://www.geni.net/resources/rspec/ext/openflow/3">
    <openflow:sliver email="user@geni.net"
                     description="My GENI experiment"
                     ref="http://www.geni.net">
      <openflow:controller url="tcp:myctrl.example.net:9933" type="primary" />
      <openflow:group name="mygrp">
        <openflow:datapath
            component_id="urn:publicid:IDN+openflow:ocf:i2cat:00:10:00:00:00:00:00:02"
            component_manager_id="urn:publicid:IDN+ocf:i2cat:ofam+authority+cm"
            dpid="00:10:00:00:00:00:00:02">
          <openflow:port name="GBE0/1" num="1"/>
          <openflow:port name="GBE0/2" num="2"/>
        </openflow:datapath>
      </openflow:group>
      <openflow:match>
        <openflow:use-group name="mygrp" />
        <openflow:packet>
          <openflow:dl_type value="0x801" />
          <openflow:nw_dst value="10.1.1.0/24" />
          <openflow:nw_proto value="6, 17" />
          <openflow:tp_dst value="80, 81" />
          <openflow:dl_vlan value="890,900" />
        </openflow:packet>
      </openflow:match>
    </openflow:sliver>
</rspec>
```

As it can be seen in this sample RSpec, the data transmitted along with this method must contain enough information to describe the slice and its owner (by using the *e-mail of the experimenter*, the *description of the slice* and a *reference to the project associated to it*); as well as defining the most important part: the *group*.

The *group* identifies exhaustively the matching conditions required to filter the desired packets and which actions are taken. This combination defines the behaviour of the experiment. For defining this, it is possible to use the *datapath ID* to identify a network element, choose a subset of its *ports* and define a *packet* XML structure where the matching conditions are defined, namely the source and/or destination *VLANs*, *IP addresses*, *ports* and many others.

3.3.2 Workflows

In this section we identify the most common operations for the SDN-RM and describe their workflow through sequence diagrams and flow charts for better comprehension.

3.3.2.1 Requesting a FlowSpace

The experimenter can define a FlowSpace either through the Expedient GUI or through the CLI. After this request is effectively provisioned, the user can start the experiment and send packets properly through the SDN network. In case the FlowSpace needs to be later on updated, the experimenter will proceed in a similar fashion.

3.3.2.2 VLAN assignment

The VLAN manager is a component devoted to identifying free VLAN(s) between SDN-RMs in a given slice that

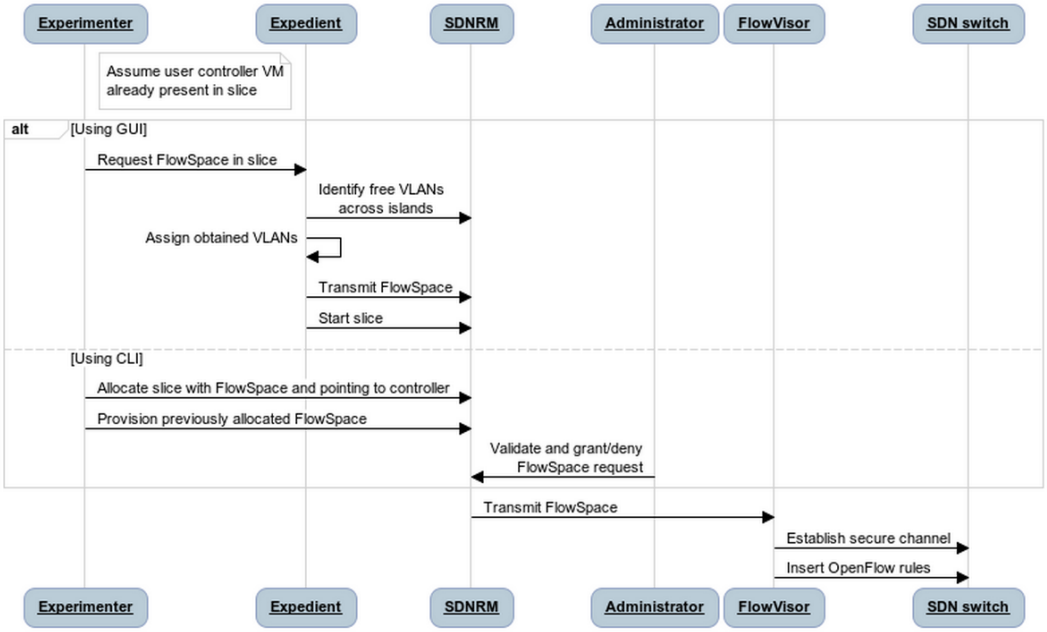

Figure 3.9: SDN-RM - Requesting a FlowSpace

spans across different domains (see Figure 3.10). It is able to retrieve a single VLAN tag or a VLAN range and it is enabled or disabled by configuring a setting on the SDN-RM. The VLANs are treated as a list of integers, ranging from 0 to 4096. However, due to the island configuration performed by each Island Manager, there are VLANs that may be tagged as reserved and cannot be used for slicing.

When a experimenter requests a FlowSpace, one or more VLANs may be requested. To ease the assignation of that value or set of values, and provided that the user only selected one SDN-RM, the VLAN manager will return directly an available VLAN. If the experimenter uses two or more SDN-RMs, the VLAN manager contacts with every SDN-RM, gets all the used VLANs and finds VLAN or VLAN range available in all the SDN-RMs. Specifically, its workflow is as follows:

1. It analyses the number of SDN-RMs used in the slice.
    - If only one SDN-RM is used, it directly asks for a random available VLAN.
    - Retrieve the used VLANs of every domain. It was decided to retrieve the used VLAN because the list of used VLANs is shorter than the list of available VLANs, a list of almost 4096 elements (the total amount of VLANs subtracting the reserved VLANs and the ones used on slices).
2. The list of VLANs can be treated as a set of elements, where all the used VLANs are the union of all the used VLAN sets.
3. To get the used VLANs set, it is only required to make the difference between the all used VLANs set and the all available VLANs (that is, 4096 VLAN tags)
4. Then is evaluated whether the user requested a single VLAN or a VLAN range.
    - If the user selected a single VLAN, then the VLAN manager assigns one random VLAN from the final VLAN set.
    - If the user requested a VLAN range, VLAN manager takes a random element from the list X, then compares if the value placed on the position equal to the length of the VLAN range is X+(VLAN Range

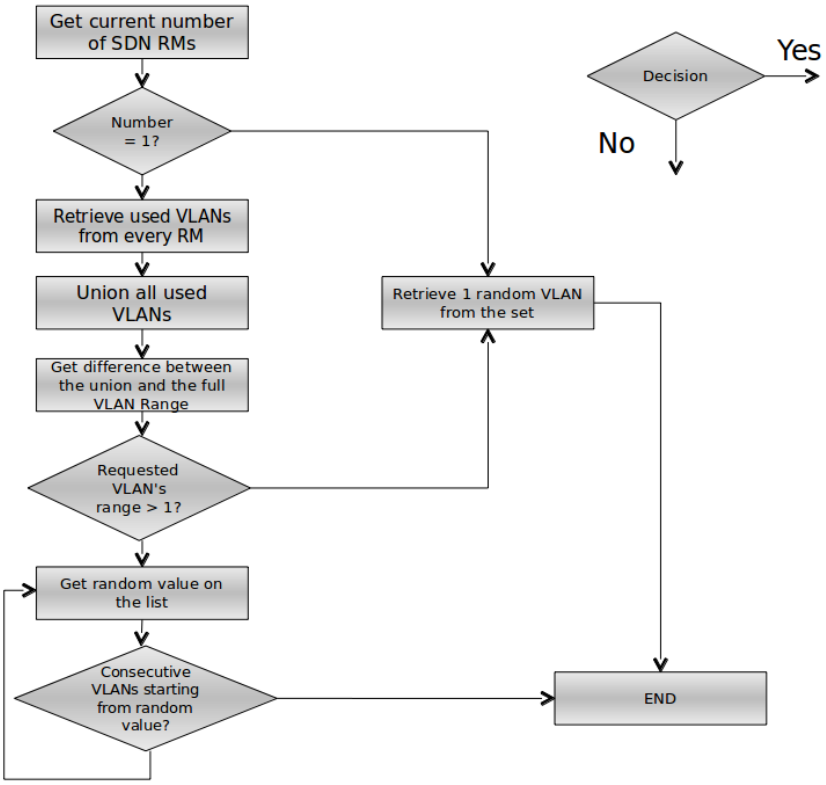

Figure 3.10: VLAN manager workflow

Length), if true then returns the range. If not true, then tries to get another random value of the VLAN set and repeats the process.

3.3.2.3 Deleting a FlowSpace

Once the FlowSpace has served its purpose, the experimenter may want to delete it. This can also be done through any of the UIs in a simple manner.

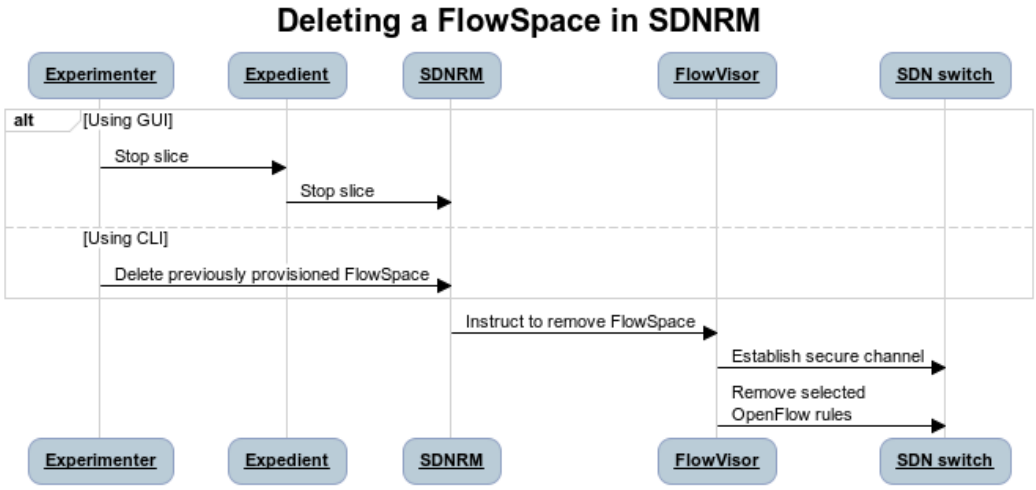

Figure 3.11: SDN-RM - Deleting a FlowSpace

3.3.3 Future Work

As with the C-RM module, the SDN-RM module is essentially complete for the current requirements, though it will probably be subject to short-term modifications as needed by the set up of he inter-communication through NSI and also to increase automation for the current FlowSpace approval process.

Besides that, the following features could be contemplated for future releases:

- Extend information presented through the GUI to allow better management.
- Improve internal data structure (e.g. representation of FlowSpace, ExperimentFlowSpace…) to allow fine-grained management and easier management.

4 Deployment

The modules presented in this deliverable provide a series of scripts along with the source that allow the domain administrator to deploy the components separately according to each domain's preferences. In the future, the deployment process is expected to be improved by using any of the open source process automation frameworks that are currently available. This would allow periodic redeployments or an easy way to recover from a failure in the deployment servers of any domain.

This section provides some details on how to deploy the Resource Orchestrator (**RO**) and the Resource Managers for computing (**C-RM**) and SDN (**SDN-RM**) resources. That is, we briefly explain how to:

- Get the code from the GIT-based FELIX repository [16].
- Install the required dependencies for each component.
- Install and configure the component itself.

At the end we also provide useful information on how the users can perform different operations on the modules that are correctly up and running.

4.1 Resource Orchestrator

The RO is a pure Python module based on the SFA architecture [14] and (partially) the eiSoil [17] framework. Basically it is composed of 2 main threads that follow an *event-driven* approach.

The first one, based on the **flask-rpc** server, is responsible to manage the GENIv3 interface and provide the provisioning for the different kind of resources. The (GENI) XML messages that are exchanged among processes or daemons are parsed and formatted according to proposed schemas (using the **lxml** library). Moreover, the operations are performed only in case of a success of the authentication and authorization mechanisms (based on the GENI ClearingHouse directives).

The second one, based on **apscheduler** python module, manages auto-generated events (periodic or one-shot) to obtain and update the discovered physical topology. The resources are stored into a non-relational database, based on the **mongo-db** server.

The following sections want to clarify the aforementioned dependencies trying to give further software details.

4.1.1 Requirements and Dependencies

Currently there is a small set of dependencies needed by the RO to work. As it has been partially based on the GENI and eiSoil frameworks, it shares some of their requirements and also requires some others.

4.1.1.1 Requirements

The working environment must be a Debian-based distribution. Specifically, the RO is being developed under Debian 7 (Wheezy).

Because the RO provides GENI APIs -which use credentials signed by a ClearingHouse-, the AAA module [13] must be running along with the RO. The RO also uses the MongoDB database for resource synchronisation; therefore this service must be running as well.

4.1.1.2 Dependencies

The packages required by RO are retrieved either by Debian's Advanced Packaging Tool (*apt-get*) and the Python version system (*easy_install*, *pip*).

Advanced packaging tool

The following Debian packages must be installed on the system for RO to properly work:

```
sudo apt-get install python-pip mongodb-server python-lxml python-m2crypto
sudo apt-get install python-openssl python-dateutil xmlsec1
```

Pip

The following python packages must be installed on the system for RO to properly work:

```
sudo pip install argparse pymongo Flask-PyMongo python-dateutil
sudo pip install lxml blinker flup Flask-XML-RPC unittest2
sudo pip install networkx apscheduler requests
```

4.1.2 Configuration and Installation

4.1.2.1 Installation

Save the *resource-orchestrator* branch under */opt/felix*:

```
mkdir -p /opt/felix/resource-orchestrator
git clone https://github.com/dana-i2cat/felix.git /opt/felix/resource-orchestrator
git checkout resource-orchestrator
```

In order to install the required dependencies, the following script can be run:

```
cd /opt/felix/resource-orchestrator/modules/resource/orchestrator/deploy
./install.sh
```

4.1.2.2 Configuration

The *modules/resource/orchestrator/conf* directory contains a set of configuration files that are used by the RO. These are explained as follows:

RO parameters

The file *ro.conf* contains the core parameters of the RO module.

Section	Parameter	Type	Description
scheduler	frequency	Integer	Seconds between runs of the RO daemons to synchronise resources between RO and other available modules
monitoring	protocol	String	Type of protocol used by the Monitoring System
monitoring	address	String	IP address where the MS is running
monitoring	port	Integer	Port where the exposed server of the MS is listening
monitoring	endpoint	String	Endpoint to reach the corresponding URL where the RO pushes the monitoring data.

Table 4.1: RO General Parameters

Server parameters

The file *flask.conf* gathers parameters for the configuration of the server. Some of the most important parameters are explained below.

Section	Parameter	Type	Description
general	host	String	IP where the server is running (0.0.0.0 to make publicly accessible)
general	port	Integer	Port of the server where the server will be listening
general	debug	Boolean	When true, performs some actions that are not called in production mode
fcgi	enabled	Boolean	Determine if FastCGI is used within Flask
fcgi	port	Integer	Port of the server where the FastCGI server will be listening
certificates	force_client_certificate	Boolean	Enable or disable verification of the certificates of the client

Table 4.2: RO Server Parameters

GENIv3 parameters

The file *geniv3.conf* gathers parameters for the configuration of the GENIv3 interface exposed by the RO.

Section	Parameter	Type	Description
general	rspec_validation	Boolean	Enable or disable validation of RSpecs passed in requests
certificates	cert_root	String	Relative path to the folder where the certificates or trusted clients are kept

Table 4.3: RO GENIv3 Parameters

Logging parameters

The file *log.conf* contains arguments to further customise the logs.

Section	Parameter	Type	Description
general	name	String	Name of the log
general	level	String	Level of the log in a Pythonic format (e.g. logging.DEBUG)
general	format	String	Formatting string that customises the structure of the logged messages
general	file	String	Name of the log file (extension included)

Table 4.4: RO Logging Parameters

4.1.2.3 Operation

A typical set of operations performed on the RO comprises starting/stopping it and accessing through exposed (GENIv3) API, which is implemented on top of a XMLRPC server.

Managing the server

The server can be started or stopped as explained below.

Starting the server

In order to start the RO server, it can be run as a daemon by calling its *initscript*:

```
/etc/init.d/felix-ro start
```

Or, alternatively, running its main class directly:

```
cd /opt/felix/resource-orchestrator/modules/resource/orchestrator/src
python main.py
```

Stopping the server

The way to stop the RO's flask server is, respectively, using the same *initscript* with the *stop* argument or sending a SIGQUIT signal (for instance, Ctrl + D).

Operating through the GENI interfaces

The Resource Orchestrator operates in the background and exposes a GENIv3 API for interaction, in a similar way as other modules in FELIX. There are different ways to connect to the RO, such as accessing through XMLRPC clients with the proper certificates or using a command-line interface. For example, this module can be accessed using the OMNI client in the following manner:

```
python omni.py -V3 -a <url> <method> <arguments>
```

Where the arguments may be optional or required and refer to an specific method (e.g. *--no-compress* for *listresources*) or to the manner in which OMNI is generated (e.g. *--debug* or *-c omni_config*).

We include some working examples with the available methods and some arguments:

```
common_args="-V3 -a https://127.0.0.1:8440/xmlrpc/geni/3/ -o"

python omni.py $common_args getversion
python omni.py $common_args listresources --no-compress --available
python omni.py $common_args describe <slice-name>
python omni.py $common_args allocate <slice-name> <req-rspec>
                            --end-time=<epoch-time>
python omni.py $common_args renew <slice-name> <epoch-time>
python omni.py $common_args provision <slice-name>
python omni.py $common_args status <slice-name or URNs>
python omni.py $common_args performoperationalaction <slice-name>
                            {geni_start, geni_stop, geni_restart}
python omni.py $common_args delete <slice-name or URN>
python omni.py $common_args shutdown <slice-name or URN>
```

A noteworthy point on security: in order for the RO to proxy requests to other modules using these interfaces, every module must trust the RO; that is, for the RO to send a request to any Resource Manager, the latter must trust the former. The trust process is currently performed by placing the RO server certificate on the *trusted* or *trusted_roots* of each module accessed by the Resource Orchestrator.

4.2 Computing Resource Manager

The C-RM is a module developed using the Python language and the Django web framework, and it includes the components mentioned in the *Design* section: 1) the core, 2) the Agent, 3) the Policy Engine and 4) the web-based GUI.

The core, the Agent and the web-based GUI are released together in the form of source code. On the other hand, the Policy Engine is designed as an external library which is released on its own and in the form a Debian package.

Once the C-RM sources are downloaded and the installation or upgrade script is run, these scripts automate the installation and help with its configuration. This is performed this way because of the PE integration within C-RM. As for the Agent installation, it has different requirements (e.g. structure of folders, installation requirements and workflow). This installation is to be performed in a different machine, where the hypervisor is running.

This is the list of requirements and dependencies to install C-RM. It is the basically the same as the needed for other OFELIA Control Framework components (e.g. Expedient, OpenFlow AM) and thus almost the same for the FELIX RMs.

4.2.1 Requirements and Dependencies

4.2.1.1 Requirements

The working environment must be a Debian-based distribution. Specifically, the framework is ensured to work under Debian 6 (Squeeze), but has been gradually being ported to Debian 7 (Wheezy). The only exception to this occurs in the Virtualisation Agent environment, which has not been migrated yet due to hard environment constraints. On the other hand, the set up of the FELIX islands provided feedback useful for further steps in this aspect.

Besides, the C-RM module requires downloading and configuring the PyPElib module during its installation procedure. The PyPElib module allows defining and evaluating against a set of rules with specific filters and data, as chosen by the user.

4.2.1.2 Dependencies

There is a number of packages required for C-RM to work. These are retrieved either by Debian's Advanced Packaging Tool (*apt-get*) and the Python version system (*easy_install*, *pip*).

Advanced packaging tool

The following Debian packages must be installed on the system for C-RM to properly work:

```
sudo apt-get install apache2 openssl ssl-cert libapache2-mod-wsgi libapache2-mod-macro
python-setuptools python-django python-mysqldb python-django-auth-ldap python-openssl
python-m2crypto python-dateutil python-decorator python-paramiko build-essential
python-imaging python-django-registration python-configobj python-pyparsing python-lxml
python-argparse python-pexpect
```

Pip

The following python packages must be installed on the system for C-RM to properly work:

```
django-evolution (<=0.6.9), django-autoslug, django-extensions (<=1.2.5)
```

4.2.2 Configuration and Installation

4.2.2.1 Installation

Save the *ocf* branch under */opt/felix*:

```
mkdir -p /opt/felix/ocf
git clone https://github.com/dana-i2cat/felix.git /opt/felix/ocf
git checkout ocf
```

Now, the root chooses which modules to install through a screen with a menu:

```
cd /opt/felix/ocf/deploy
python install.py
```

The module to be selected in the menu is named *vt_manager*.

Once the installation starts, the OFVER installation scripts under the C-RM (or *vt_manager*) folder are triggered and will ask whether the current installation is run within the OFELIA project or not. Select No (N) for non-OFELIA testbeds.

There are similar procedures for upgrading and removing the modules as well.

4.2.2.2 Configuration

General parameters

Table 4.5 shows general C-RM parameters.

FLAG	Values	Comments
ISLAND_NAME	String	Island/testbed name. Shown on the upper right corner of the Web frontend
VTAM_IP	String	C-RM host domain name/IP
VTAM_PORT	String	Web UI and XMLRPC TCP port (default:8445)
XMLRPC_USER	String	XMLRPC interface username
XMLRPC_PASS	String	XMLRPC interface username's password

Table 4.5: C-RM General Parameters

Root (Island Manager) account information

Root account parameters are listed in Table 4.6.

FLAG	Values	Comments
ROOT_USERNAME	String	C-RM's root username
ROOT_PASSWORD	String	C-RM's root password
ROOT_EMAIL	String	C-RM's root email

Table 4.6: C-RM Root Account Parameters

Database parameters

Important database parameters can found in following table.

FLAG	Values	Comments
DATABASE_USER	String	MySQL username
DATABASE_PASSWORD	String	MySQL password
DATABASE_HOST	String	MySQL host (e.g. 127.0.0.1)
DATABASE_NAME	String	C-RM database name

Table 4.7: C-RM Database Parameters

Configure Agent on a virtualisation server

The configuration of the Agent is done during the installation process. The most important parameter to take into account is the *XMLRPC_PASSWORD*. This parameter is required later on, when adding servers to the C-RM.

VM AM GUI procedures

Various VM AM parameters to be configured are described below.

Adding an Ethernet/IP range

The virtualisation servers that are able to generate the VMs of the users must have one available range (pool) of MAC addresses and IPs. This, in a common configuration, means that at least one global IP and MAC range shall be defined. This is further explained in the next sections.

Global vs. non-global ranges

By default, if a server is not subscribed to any range in particular, it will use all the global ranges to obtain MACs and IPs, in a random order. However, if the servers subscribe to one or more ranges (whether they are global or not), VMs will be assigned an address contained only in the pool of subscribed ranges.

The global flag, accordingly, should be disabled by those ranges that are particular to one server and that do not apply to the rest of the servers (for instance a local testing server out of the addressing scheme used for the testbed). In general, a server should use global ranges for simplicity.

Creating a range

The C-RM UI dashboard allows to define starting and ending values for the IPv4 and Ethernet ranges. Such ranges (along with other parameters such as the network mask, gateway and DNS) will be used to assist the VM generation process with the needed networking information. It is possible to generate a range by accessing the C-RM UI, then selecting the appropriate option under *Network Settings* and then click on *Create range*.

For IPv4 ranges, the following fields are available:

- **Range name**: Name used to identify the range.
- **Global range**: Flag that indicates whether the range is global or not.
- **Range start IP**: Starting IP address defined for the range.
- **Range end IP**: Ending IP address defined for the range.
- **Network mask**: Network mask to apply over the IPs.
- **Gw**: IP address of the default gateway for the VMs.
- **Dns1**: Main DNS used to generate IPs for VMs.
- **Dns2**: Secondary DNS used to generate IPs for VMs.

For Ethernet ranges, the following fields are available:

- **Range name**: Name used to identify the range.
- **Global range**: Flag that indicates whether the range is global or not.
- **Start Mac Address**: Starting physical address assigned to VMs.
- **End Mac Address**: Ending physical address assigned to VMs.

After both IPv4 and Ethernet ranges have been set, it is possible to subscribe a given server to a specific range; if on need of a particular addressing schema per server.

Creating a server

At least one server must be configured in order to provide VMs to the users. This is accessible either from the main page or by the *Administrate Server* section.

- **UUID**: Automatically generated.
- **Enabled**: The server may be disabled by non checking this field.

- **Name**: Name of the server.
- **OS Type**: Server's OS type.
- **OS Distribution**: Server's OS type.
- **OS Version**: Server's OS version.
- **Virtualization Technology**: hypervisor that is running in the server. Currently restricted to XEN.
- **URL of the Server Agent**: URL where the Agent daemon in the server is listening. It should be `https://DOMAIN_NAME:PORT/`, where default PORT is *9229*.

4.2.3 Operation

The C-RM may be used by an experimenter through the GUI or using its exposed interfaces. The basic operations are explained here.

4.2.3.1 Operating through the GUI

Different functionalities of the C-RM can be accessed through the GUI: the experimenter is able to perform the following operations through the Expedient GUI (via the corresponding *C-RM* plug-in), whilst the administrator is allowed to enter the specific C-RM GUI to configure part of the physical infrastructure and have direct control on the resources. We document the approach of the experimenter in the following sections.

Creating a VM

In order to create a virtual machine, the user shall access the Expedient GUI and access a previously existing slice. Provided that the slice contains at least one Virtualisation RM (and thus is able to request this kind of resources), the experimenter must choose an specific island and select a server on it. After that, another page is loaded where the user must define the following fields:

- **Disc Image**: The type of template used to generate the VM (e.g. Debian 6, 7…).
- **Name**: Name of the VM using alfanumeric characters.
- **Memory**: Size of virtual memory, in Mb.
- **HD Setup Type**: Type of virtualised hard disk; for instance a file image with or without partitions.
- **Virtualization Setup Type**: Type of virtualisation used; for instance through Paravirtualisation or HVM.

After the VM is requested, the petition is sent to the Virtualisation Agent and the process of generating the virtual machine starts in the physical substrate. Once finished, the GUI shows a change of status indicating the availability to start operating on the machine.

Managing a VM

Once the VM is provisioned by the previous set of steps, it can be started, stopped, rebooted or deleted through the same page with the details of the slice. A similar process is started on background so the Agent is contacted and the command is executed remotely.

Accessing the VM

When the VM is up and running, the experimenter can access it through ssh. The authentication and authorisation processes are taken using the credentials of the user, either being explicitly passed (basic auth). With the full integration of the AAA module, the machines will be accessible through the user's public key.

4.2.3.2 Operating through the GENI interfaces

Creating a VM

When using the public interfaces through a command line (or indirectly through a client), the process is a bit more complicated; both in terms of compliance to the workflow as well as handling the credentials that must be passed along with the commands.

Here, the VM is initially requested by calling to the *Allocate* method with a file called RSpec. This file must follow a specific format (GENIv3 compliant) and contains a specific set of information, such as the server to host it and the name. This request acts as a reservation, as it keeps the requested resources on hold during a specific period of time. Before this time is exhausted, a *Provision* request must be made to effectively provision the VM and start operating on it. Alternatively, a *Renew* command may be issued to extend the reservation.

Managing a VM

When the machine is provisioned and ready to use, the experimenter is able to invoke a *PerformOperationalAction* command to start, stop and restart the VM. The experimenter may also call *Renew* once again on the provisioned resource to extend its or use the *Delete* operation on it.

Accessing a VM

Once the VM is up, the experimenter can access it the same way, through ssh. The authentication and authorisation processes are performed now by matching against the user public key that is placed on the VM at the time of its generation.

4.3 Software-Defined Networking Resource Manager

The SDN-RM is another module developed using the Python language and the Django web framework. It consists of the components mentioned in the Design section: 1) its core functionality, 2) the web-based GUI and 3) the proxy class to communicate with the FlowVisor module.

The FlowVisor module is a third-party library developed in Java that is able to communicate with the switches and proxy packets to their corresponding experiment controller. The latter is required by the SDN-RM.

All of these modules are released together in the form of source code and are installed using the scripts provided by OFVER, released with this module. Another script automates the installation and configuration of the FlowVisor package right before the SDN-RM is installed or upgraded.

4.3.1 Requirements and Dependencies

This is the list of requirements and dependencies to install SDN-RM. It is the basically the same as the needed for other OFELIA Control Framework components (e.g. Expedient, Virtualisation AM) and thus almost the same for the FELIX RMs.

4.3.1.1 Requirements

The working environment must be a Debian-based distribution. Specifically, the framework is ensured to work under Debian 6 (Squeeze), but has been gradually being ported to Debian 7 (Wheezy).

Besides, the SDN-RM module requires downloading and configuring the FlowVisor package, as it is able to communicate with the OpenFlow-enabled switches.

4.3.1.2 Dependencies

There is a number of packages required for SDN-RM to work. These are retrieved either by Debian's Advanced Packaging Tool (*apt-get*) and the Python version system (*easy_install*, *pip*).

Advanced packaging tool

The following Debian packages must be installed on the system for SDN-RM to properly work:

```
sudo apt-get install apache2 openssl ssl-cert libapache2-mod-wsgi libapache2-mod-macro
sudo apt-get install python-setuptools python-django python-mysqldb
sudo apt-get install python-django-auth-ldap python-openssl python-m2crypto
sudo apt-get install python-dateutil python-decorator python-paramiko build-essential
sudo apt-get install python-imaging python-django-registration python-configobj
sudo apt-get install python-pyparsing python-lxml
```

Pip

The following python packages must be installed on the system for SDN-RM to properly work:

```
django-evolution (<=0.6.9), django-autoslug, django-extensions (<=1.2.5)
```

4.3.2 Configuration and Installation

4.3.2.1 Installation

Save the *ocf* branch under */opt/felix*:

```
mkdir -p /opt/felix/ocf
git clone https://github.com/dana-i2cat/felix.git /opt/felix/ocf
git checkout ocf
```

Now, the root chooses which modules to install through a screen with a menu:

```
cd /opt/felix/ocf/deploy
python install.py
```

The module to be selected in the menu is named *optin_manager*.

Once the installation starts, the OFVER installation scripts under the SDN-RM (or *optin_manager*) folder are triggered and will ask whether the current installation is run within the OFELIA project or not. Select No (N) for non-OFELIA testbeds.

There are similar procedures for upgrading and removing the modules as well.

4.3.2.2 Configuration

FLAG	Values	Comments
SITE_DOMAIN	String	SDN-RM host domain name

Table 4.8: SDN-RM General Parameters

Root (Island Manager) account information

FLAG	Values	Comments
ROOT_USERNAME	String	SDN-RM's root username
ROOT_PASSWORD	String	SDN-RM's root password
ROOT_EMAIL	String	SDN-RM's root email. This is used to send notifications

Table 4.9: SDN-RM Root Account Parameters

Database parameters

FLAG	Values	Comments
DATABASE_USER	String	MySQL username
DATABASE_PASSWORD	String	MySQL password
DATABASE_HOST	String	MySQL host (e.g. 127.0.0.1)
DATABASE_NAME	String	SDN-RM database name

Table 4.10: SDN-RM Database Parameters

GUI procedures

Configuring the connection with Expedient

From the *Manage Website* button two actions should be performed:

1. Set Clearinghouse: This sets the username/password which the Clearinghouse (in Expedient) will use to authenticate against SDN-RM when using the XMLRPC interface. Just set:
   - *Username*
   - *Password*
2. Set FlowVisor: This sets the parameters required to communicate with the FlowVisor:
   - *FV Name*: Name to identify this FlowVisor instance.
   - *Username*: Username to use to access the FlowVisor (set during FlowVisor installation)
   - *Password*: Password to use to access the FlowVisor (set during FlowVisor installation)
   - *Server URL*: URL of the FlowVisor's XMLRPC interface. It should be `https://DOMAIN_NAME:PORT/xmlrpc/`, where default port is *8080*. *DOMAIN_NAME* can be an IP address as long as it matches the certificate's Common Name of the server where FlowVisor is running.

Handling FlowSpace requests

If properly configured, the SDN-RM module will send an email to the root e-mail address each time a new request comes from the Expedient. The FlowSpace approval can be set either to automatic or manual modes:

- *Automatic approval*: automatically approve incoming FlowSpace requests without the administrator inspecting them beforehand.
- *Manual approval*: administrator reviews each FlowSpace request prior to approving or denying it.

a) Automatic approval

In the automatic approval mode, SDN-RM can be configured to:

1. Automatically negotiate a VLAN for slices that span several domains
2. Automatically approve FlowSpace requests

The selection screen for those options is available under *Manage Website* > *Auto-Approve Settings*. When the *Approve all requests* option is selected, one or both may be selected:

- *Grant VLANs automatically*: automatically establishes a connection against other SDN-RMs involved in the experiment.
- *Approve Flowspace automatically*: in conjunction with the previous option, it allows to automatically approve the incoming FlowSpace request

b) Manual approval

If the automatic approval has not been enabled on the island, the FlowSpace requests must be handled manually by the island administrator. Requests coming from Expedient do not appear on the *Request list* section, but instead must be accessed through the *Administrate Flowspace* > *Add rule* area.

4.3.3 Operation

The SDN-RM, as the C-RM, may be used by an experimenter through the GUI or using its exposed interfaces. The basic operations are explained in the following sections.

4.3.3.1 Operating through the GUI

As before, the different approaches (*experimenter* and *administrator*) are distributed in two different GUIs: the experimenter may perform a number of requests and managing operations through the Expedient GUI (via the corresponding *SDN-RM* plug-in), whilst the administrator has permission to grant, deny and monitor currently available FlowSpaces. The following sections detail the operations an experimenter may perform.

Creating a FlowSpace

The experimenter must choose a path for the packets to flow across the network. This is done by choosing a series of ports for each network devices (through its datapath ID) as well as a slicing condition. That is, a condition that makes it possible to differentiate traffic from this experiment and others. Slicing is typically performed using one or more VLANs, which are assigned exclusively and under demand to a given experiment.

In order to define the FlowSpace, the user accesses a previously existing slice through the Expedient GUI. Providing that an instance of the SDN-RM is already available for the slice, the experimenter must click the *Define flowspace (virtual topology)* button. The following screen allows the user to select the desired SDN path by manually selecting each port of the network devices. The switches and ports that conform this path can be selected through an interactive graph or through a simple list of check boxes.

After selecting the ports (and thus the SDN path), the user must define at least one filtering condition to match the packets from this experiment. This can be chosen either trough the *Simple* or the *Advanced* mode.

Simple mode

In the simple mode, the user only selects how many VLANs are needed for the experiment. The OpenFlow plug-in available in the Expedient does the rest of the work by agreeing on a shared set of VLANs across the islands spanned by the slice.

Advanced mode

In the advanced mode, the user is provided with a web form that shows every possible matching header in an OpenFlow packet. This form can be filled to match any of these header with a specific value, although the VLANs are the main mechanism we use for slicing and therefore one (at least) must be chosen.

Managing a FlowSpace

The FlowSpace is inherently started or activated when the slice is started, which is possible by clicking the *Start* button on top of the slice detail page. Before the slice can be started, the FlowSpace chosen by the user must point to a VM appointed to use as the experiment controller.

When started, the FlowSpace is sent from Expedient to the SDN-RM, where it is automatically granted and sent to the FlowVisor, which is ultimately in contact with the SDN-enabled switches. When the slice is stopped, the FlowSpace is accordingly deactivated.

4.3.3.2 Operating through the GENI interfaces

Creating a FlowSpace

When using the public interfaces of the SDN-RM, the experimenter must follow a similar set of steps as when requesting any other resource and also meet the appropriate syntax.

The FlowSpace is requested through the *Allocate* method in the first place. The RSpec passed to it must be GENIv3 compliant and follow the standard OpenFlow schema. This request acts as a reservation, as it keeps the requested resources on hold during a specific period of time. Before this time is exhausted, a *Provision* request must be made to effectively activate the FlowSpace and start operating on it. Alternatively, a *Renew* command may be issued to extend the reservation.

Managing a FlowSpace

Once the FlowSpace is active, the user is able to experiment using the requested SND path. When invoking the *PerformOperationalAction* option with the *start* or *restart* command, the FlowSpace of the experiment is activated, automatically granted (if this exact request was granted before) and the rules are placed on the switchs.

5 Conclusions and Summary

In order to effectively provide resource planning capabilities, a system needs the ability to retrieve enough information of the underlying infrastructures. The Resource Orchestrator (RO) is a new software module in the FELIX Framework whose objective is to intercept and examine each experimenter request, then evaluate the feasibility of the requested operation and finally steer to the appropriate destination. The process of determining its feasibility is carried out by comparing against monitored data (either from its internal database or from the Monitoring System) and taking advantage of the overall view it keeps on the infrastructure(s) below.

This document has presented this module and two of the Resouce Managers, namely the Computing Resource Manager (C-RM) and the Software-Defined Networking Resource Manager (SDN-RM).

After a brief introduction of the general architecture of the FELIX Framework, in which we highlighted the key functionalities of each component, we have described the RO, the C-RM and the SDN-RM and provided high-level implementation details on their design, internal structure and documented the exposed interfaces. We have also introduced a set of sequence diagrams and flow charts to explain the internal workflow of the most relevant functions and the inter-module communication required by each module to realize its basic functions.

We pointed out that the key module presented in this deliverable, the RO, is developed from scratch during the Y2 of the project and so it must undergo through iterative stages of refinement during the integration stage with other components and the evaluation of the FELIX Use Cases. The Computing and Software-Defined Networking Resource Managers are, in turn, devoted to allocating and provisioning their specific resources. In conformance with the FELIX commitment to reuse existing modules, the latter two are based on the ones provided by the OFELIA project and are subsequently extended as needed to meet the FELIX requirements and Use Cases.

In the final sections of the document we have presented the deployment of every software component into the physical infrastructure along with their requirements and dependencies. The installation and configuration steps and examples on the key operations are also provided.

Future work is addressed towards the integration of the different software modules into the large-scale and distributed FELIX infrastructure. We aim to present the six Use Cases defined into the D2.1 deliverable by using the FELIX Framework as their basic foundation.

References

[1] B. Belter, et al., ``FELIX Deliverable D3.3, Inter-Domain Networking Between SDN Slices,'' FELIX Deliverable D3.3, Jan. 2015.

[2] ``GENI: Global Environment for Network Innovations.'' http://www.geni.net.

[3] ``The Omni client version 2.5.'' http://trac.gpolab.bbn.com/gcf/wiki/Omni.

[4] T. Ikeda, et al., ``FELIX Deliverable D3.2 - Slice Monitoring,'' FELIX Deliverable D3.2, Jan. 2015.

[5] T. Kudoh, et al., ``Network services interface: An interface for requesting dynamic inter-datacenter networks,'' *Optical Fiber Communication Conference (OFC)*, Mar. 2013.

[6] R. Krzywania, et al., ``FELIX Deliverable D2.2, General Architecture and Functional Blocks,'' tech. rep.

[7] M. Sune, et al., ``Design and implementation of the OFELIA FP7 facility: The European OpenFlow testbed,'' *Computer Networks*, 2014.

[8] ``MongoDB database Version 2.6.'' http://www.mongodb.org/.

[9] ``GENI Aggregate Manager API v3.'' http://groups.geni.net/geni/wiki/GAPI_AM_API_V3.

[10] ``GENI API Identifiers.'' http://groups.geni.net/geni/wiki/GeniApiIdentifiers.

[11] ``GENI APIv3 Common Concepts.'' http://groups.geni.net/geni/wiki/GAPI_AM_API_V3/CommonConcepts.

[12] R. Krzywania, et al., ``FELIX Deliverable D2.1, Experiment Use Cases and Requirements,'' FELIX Deliverable D2.1, Sept. 2013.

[13] C. Bermudo, et al., ``FELIX Deliverable D3.4, End User Tools and API,'' FELIX Deliverable D3.4, Jan. 2015.

[14] ``Slice Facility Architecture 2.0.'' http://groups.geni.net/geni/attachment/wiki/SliceFedArch/SFA2.0.pdf, July 2010.

[15] ``GENI Aggregate Manager API.'' http://groups.geni.net/geni/wiki/GAPI_AM_API.

[16] ``Felix repository website - private to consortium for the time being.'' https://github.com/dana-i2cat/felix.

[17] ``eiSoil framework.'' https://github.com/EICT/eiSoil.