Data Centres Optimization for Energy-Efficient and EnvironmentalLy Friendly INternet

Funding scheme: Specific Targeted Research Projects – STREP
Co-funded by the European Commission within the Seventh Framework Programme

Project no. 609140

Strategic objective: FP7-SMARTCITIES-2013 (ICT-2013.6.2)

Start date of project: October 1th, 2013 (36 months duration)

Deliverable D3.4

Energy Efficiency Policy Maker and Actuator component (Implementation)

Due date:	31/01/2016
Submission date:	18/03/2016
Deliverable leader:	NXW
Author list:	Tommaso Zini (NXW), Andrea Gronchi (NXW), Matteo Pardi (NXW), Adrián Roselló (I2CAT), Isart Canyameres (I2CAT), Amaia Legarrea (I2CAT), Artemis Voulkidis (SYN)

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)

List of Contributors

Participant	Contributor
NXW	Tommaso Zini, Andrea Gronchi, Matteo Pardi
i2CAT	Adrián Roselló, Isart Canyameres, Amaia Legarrea
SYN	Artemis Voulkidis

Amendment History

Version	Date	Partners	Description/Comments
0.1	27/01/2016	NXW	ToC Initial draft
0.2	12/02/2016	NXW, I2CAT. SYN	Major contributions
0.3	28/02/2016	NXW	Improvements to Policy Maker and Policy Repository sections and overall editing revision
0.4	10/03/2016	NXW, I2CAT, SYN	Improvements on section VM Priority Classifier and Optimizer. Some minor revisions on the contents and editing fixes
FF	17/03/2016	NXW, I2CAT, SYN	Final review

Table of Contents

Figures Summary

Tables Summary

Abbreviations

API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
CXF	Apache Open-source framework: https://cxf.apache.org/
DB	Data Base
DC	Data Centre
DCO	Data Centre Optimization
DoW	Description of Work
eCOP	energy Consumption and Optimization Platform
HTTP	HyperText Transfer Protocol
HW	Hard Ware
ICT	Information Communication Technologies
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MySQL	https://www.mysql.com/
OS	Operating System
RDBMS	Relational Database Management System
REST	Representational State Transfer
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
SVR	Support Vector Machine for Regression
ToC	Table of Contents
URL	Uniform Resource Locator
VM	Virtual Machine
WP	Work package

Executive Summary

The DOLFIN Energy Efficiency Policy Maker and Actuator represents a second group of elements that, together with the ICT Performance and Energy Supervisor (detailed in the D3.3 [2]), composes the eCOP DOLFIN subsystem, as objective in the context of the WP3.

This document is fed by the deliverables D2.1, D2.2, D3.1 previously submitted by the DOLFIN consortium and provides a detailed description of each modules parts of the Energy Efficiency Policy Maker and Actuator, with description of the interfaces and the interactions with other modules. Moreover, highlights those parts requested some revision since the initial design (described in the deliverable D3.1 [1]). In general, few changes were introduced that in any case were identified aiming to simplify the overall architecture, focusing the works on the parts that where more significant for the objectives in DOLFIN.

The document starts introducing a brief description of overall eCOP DOLFIN subsystem, due to the nature of the Energy Efficiency Policy Maker and Actuator that represents a kind of crossroads between various internal (within the eCOP subsystem) and external (SDC subsystem, see D4.3 [4] and D4.4 [5]) entities.

The four main functionalities are explained in the following subsections where the Energy Efficiency Policy Maker and Actuator is involved:

1. Performance and workload prediction.
2. Control, management and coordination of optimization activities policies based.
3. Build and processing of optimization plan
4. Actuation of optimization plan on the DC

Finally, the document provides the necessary information to build and test every system component separately.

1.Introduction

This document explains the implementation details of the Energy Efficiency Policy Maker & Actuator, a group of elements that form the upper layer of the eCOP DOLFIN system. In this layer reside the major intelligence of the DOLFIN architecture where decisions are taken and performed for the energy optimization, based on pre-defined criteria and the input provided by other DOLFIN modules.

This deliverable focuses on the in-depth explanation of the development activities for the components within the Energy Efficiency Policy Maker & Actuator group and represents a complement part of the DOLFIN eCOP overall architecture, that was addressed by earlier deliverable D3.1 [1] .

For completeness, the description of the high level DOLFIN architecture is reported in Figure 1-1 as starting point for detail description reported in the following subsections.

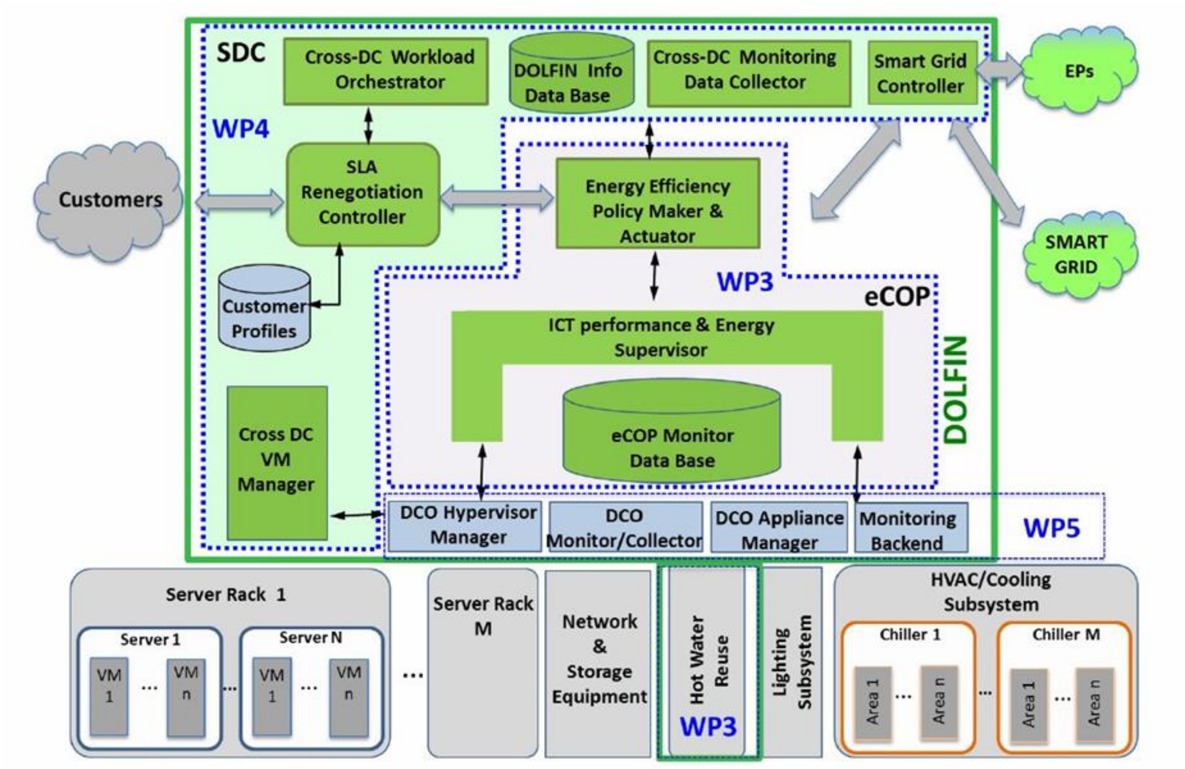


Figure 1-1 - DOLFIN overall architecture

A Git repository have been created to manage all DOLFIN project components, and it can be found at http://stash.i2cat.net/projects/DOL. The access to this repository is secured through username and password and hosted by i2CAT. In other to request access to this repo, one should send an email to the following mailing address: *amaia(.)legarrea(@)i2cat(.)net*

2. Energy Efficiency Policy Maker and Actuator

2.1. Module description

The Energy Efficiency Policy Maker & Actuator consists in specialized components each of which fulfils a specific function:

- *VM Priority Classifier*: assesses the priority of VM streams based on their time criticality, their resource consumption and the SLA of the customers or other entities.
- *Prediction Engine*: provides forecasts regarding the load expected in the near future, based on the status of both the sole DC or as a part of a federated set of DOLFIN-enabled DCs.
- *Policy Repository*: maintains information regarding all possible policies available to conduct the operation efficiency resource management.
- *Policy Maker*: schedules the activation of the policy enforcement, on the basis of the DC status and the information provided by other modules. This module is responsible for the efficient resource management and the acceptance or rejection of incoming requests at local or synergetic DCs level. This is the key component of the chain.
- *Optimizer*: optimizes the allocation of existing and accepted load to the DC physical resources, while adjusting to the operation imposed by the selected policy, also taking into account the forecast of the Prediction Engine on the near future load and/or other information such as the Smart Grid status by the Smart Grid Controller.
- *Policy Actuator*: is responsible for the implementation of the actions identified by the Optimizer and translates the optimizer plan into commands to the DCO Hypervisor Manager and/or the DCO Appliance Manager.

The diagram in Figure 2-1 provide a representation of the above components, presenting also the main interactions between internal eCOP subsystem modules such as the interfaces to and from external entities, which details are defined in the deliverables D4.3 [4] and D4.4 [5] . A reference summary of these interfaces are reported in Table 2-1.

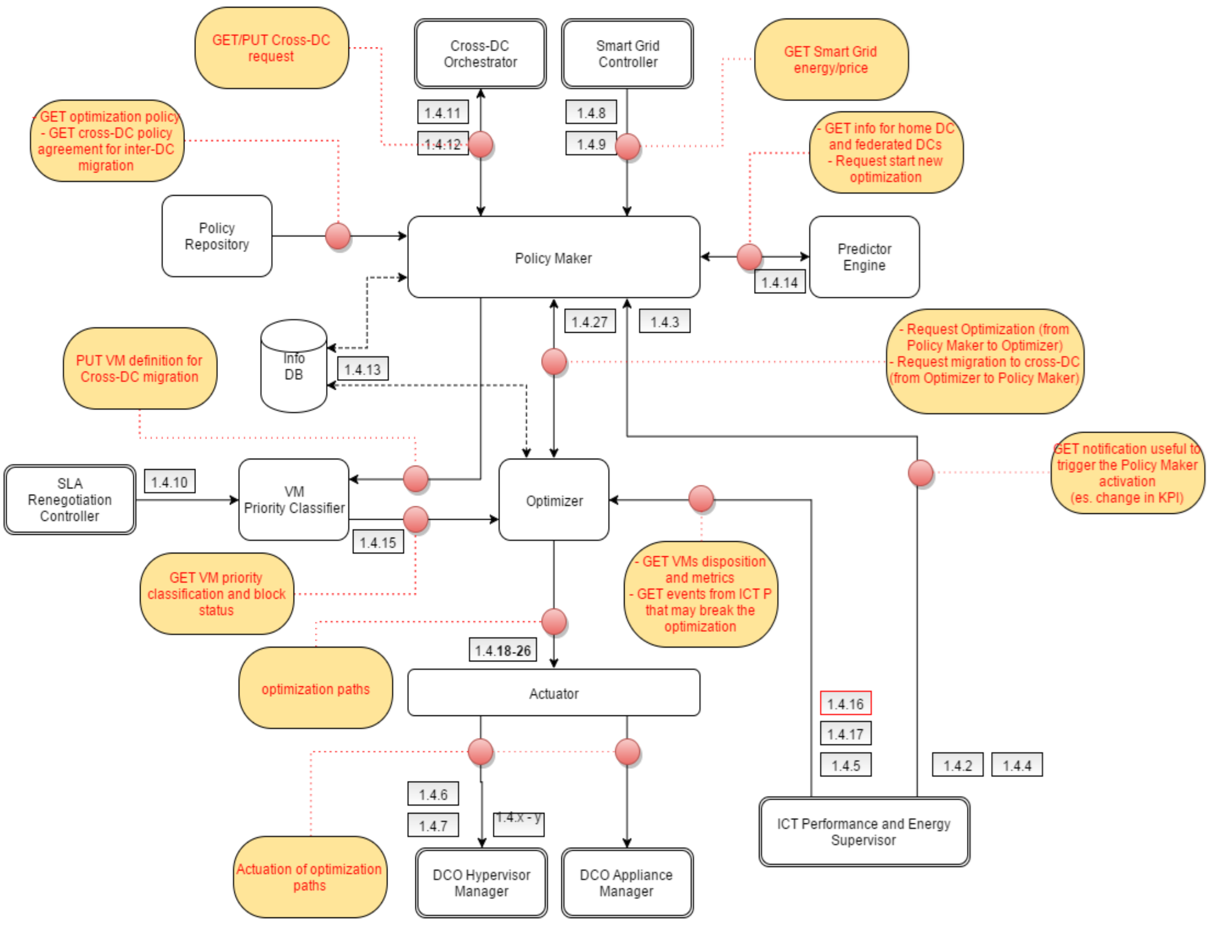

GET/PUT Cross-DC request
Cross-DC Orchestrator
Smart Grid Controller
GET Smart Grid energy/price
- GET optimization policy
- GET cross-DC policy agreement for inter-DC migration
1.4.11
1.4.12
1.4.8
1.4.9
- GET info for home DC and federated DCs
- Request start new optimization
Policy Repository
Policy Maker
Predictor Engine
1.4.14
1.4.27
1.4.3
Info DB
1.4.13
PUT VM definition for Cross-DC migration
- Request Optimization (from Policy Maker to Optimizer)
- Request migration to cross-DC (from Optimizer to Policy Maker)
GET notification useful to trigger the Policy Maker activation (es. change in KPI)
SLA Renegotiation Controller
1.4.10
VM Priority Classifier
Optimizer
1.4.15
GET VM priority classification and block status
- GET VMs disposition and metrics
- GET events from ICT P that may break the optimization
1.4.18-26
optimization paths
Actuator
1.4.16
1.4.17
1.4.5
1.4.2
1.4.4
1.4.6
1.4.7
1.4.x - y
ICT Performance and Energy Supervisor
Actuation of optimization paths
DCO Hypervisor Manager
DCO Appliance Manager

Figure 2-1 - Energy Efficiency Policy Maker and Actuator architecture

Ref. ID	API Name
1.4.1	Policy Plan Submit
1.4.5	Request metric / measure
1.4.6	VM Migration
1.4.7	VM Status Change
1.4.8	Consumption Trend Inquiry
1.4.9	Consumption Energy Price
1.4.10	SLA Downgrade Plan
1.4.11	Cross boundary Inter-DC VM Migration (Inquiry)
1.4.12	Cross boundary Inter-DC VM Migration (Trigger)
1.4.13	Log notification
1.4.14	Load Predictions
1.4.15	VM priorities
1.4.16	Server Energy Classification
1.4.17	Server Energy Consumption
1.4.18	VM Allocation
1.4.19	VM_migrate
1.4.20	VM_shift
1.4.21	DVFS

1.4.22	Serv_term
1.4.23	Serv_oper
1.4.24	Ancil_term
1.4.25	Ancil_oper
1.4.26	Air_cond _temper
1.4.27	VM_relocate

Table 2-1 - Energy Efficiency Policy Maker and Actuator API summary

2.1.1. Policy Maker

The Energy Policy Maker & Actuator represents a central point in the DOLFIN architecture, where decisions for the application of energy policies are taken. The logical functions of this component are distributed over a subsets of specialized entities that implement both the policy decision and enforcement.

Many actions applied by the Energy Policy Maker and Actuator require the interaction with external components, such as:

- Smart Grid Controller; provide information related to energy prices.
- Cross-DC Workload Orchestrator; used to distribute workload on different DCs within the federated DOLFIN environment.
- SLA Renegotiation Controller; used to coordinate and take policy decision based on the user Service Level Agreement (SLA).

Dedicated APIs are implemented at the level of the internal entities of the Energy Policy Maker and Actuator to ensure correctness in data exchange and proper activation control between various components. The Figure 2-2 provides a high level description on the major interactions from the Policy Maker to the surrounded DOLFIN components.

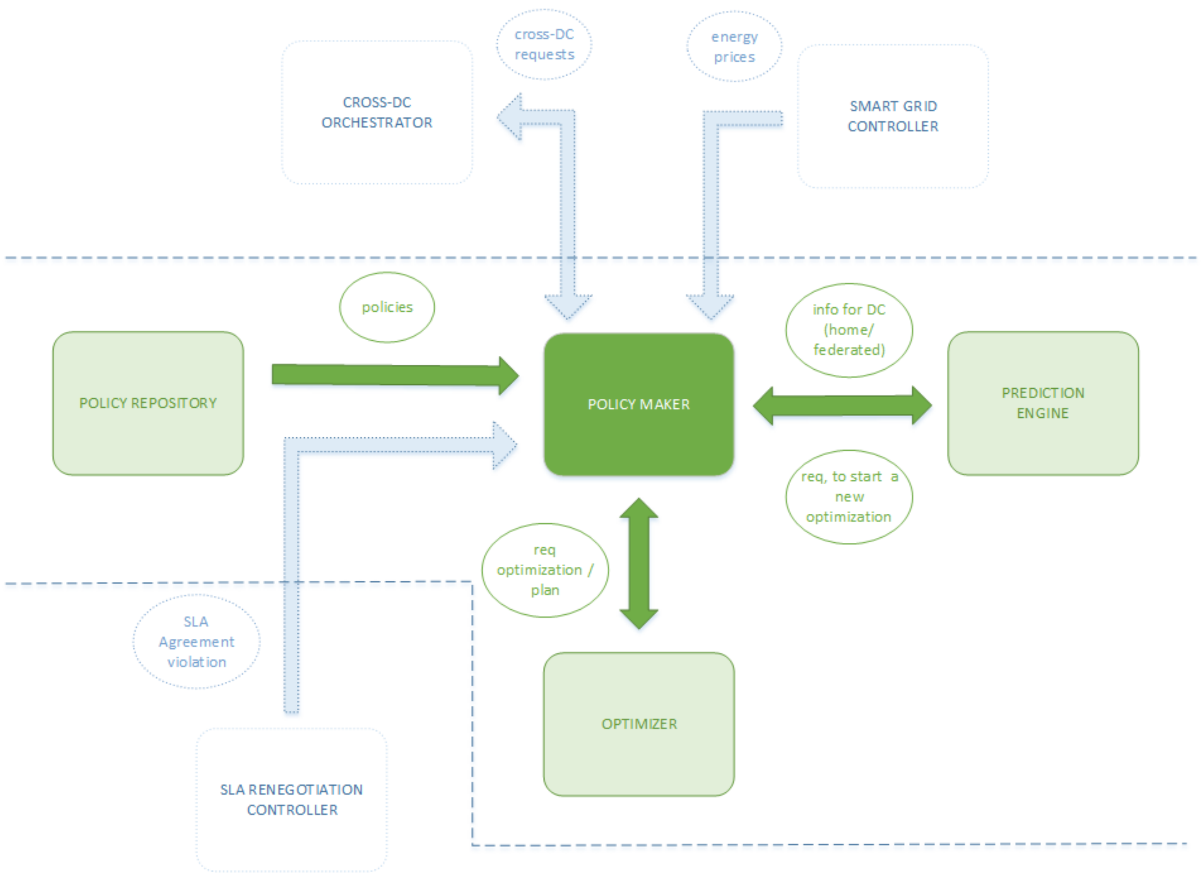


Figure 2-2 - Policy Maker interactions

The communication between the Policy Maker and other modules is performed through REST APIs, except for messages incoming from the SLA Renegotiation Controller, which are put in line in a dedicated queue. This approach is needed to handle violation triggers that can be unexpected, they could arrive in any moment, and they need to be treated with a priority in respect to other messages (an agreement violation must be handled as soon as possible); so the communication model has to be different than with other components.

In a typical behavior, the Policy Maker periodically receives updates from the Smart Grid Controller (about energy prices), and from the Cross-DC Orchestrator about optimizations at Synergetic DC level, or it is triggered by the SLA Renegotiation Controller or the Prediction Engine requesting a new policy to be evaluated. Reacting to these triggers (if needed), the Policy Maker may:

- Reacts depending to the current set policy. The policy could force the Policy Maker to discard specific triggers or define for each of them a different treatment priority.
- Chooses a new policy from the Policy Repository.

Then the Policy Maker sends a proper request to the Optimizer. Once the Optimizer has received data from the Policy Maker, and it has calculated a plan to be actuated, it will send a notification to the Policy Maker itself with details of that plan. It will be the Policy Maker which has to reply, indicating to the Optimizer to apply that plan or to not proceed.

Elaborating on its role, the Energy Policy Maker and Actuator is set to:

- Apply a set of well-known criteria and evaluation patterns to optimize the DC energy consumption (i.e. determining a set of operations to improve the DC energy efficiency).
- Produce a stream of requests which can be translated into actual actions by others DOLFIN servant subsystems within a DC.
- Determine corrective actions taking each action trade-offs and cost into account.
- Reacts to SLA violations messages, coming from Renegotiation Controller module, in order to be in compliance with agreements.

This last bullet point is an add-on for what described in document D3.1 [1] . It has become necessary to make SLA Renegotiation Controller module able to send messages to the Policy Maker; every time a violation of the agreement comes up, the SLA component triggers the Policy Maker to evaluate a new policy, in order to restore conditions into the bounds imposed by the agreement itself.

2.1.2. Policy Repository

The Policy Repository maintains information regarding all possible policies, which can be picked and translated into actions, to achieve different energy efficiency and resource management goals. That is: *what the system does in order to shift specific KPIs toward the value we want to achieve*.

This repository registers a sets of well-defined criteria to respond to different scenarios as follow:

- The energy policy used internally by the eCOP functions. The Policy Maker queries the repository for the available policies in order to extract the best policy. This selection is performed using sets of predefined criteria based on inputs to the Policy Maker and its knowledge of the DC status. It will be the Optimizer Engine module which will reacts selecting proper algorithms based on policy information.
- Agreements for federated DCs workload distribution. VM requests arrive at the Policy Maker through the Cross-DC Orchestrator, which provides basic functions of request pre-processing. These operations, like filtering based on inter-DCs SLA agreement, would reject VM requests from non-federated DCs or even provide a higher priority to load from high priority DCs. This information, pertaining to the DC priority, characterizes also the VM priority, and is passed through the Policy Maker to the VM Priority Classifier, if the VM request is accepted.

2.1.3. Prediction Engine

The Prediction Engine is the DOLFIN component responsible for providing predictions in the course of performing predictive optimization in the single-DC operation context. Particularly, it is capable of providing forecasting services for all supported (and actively measured) measurement types, based on the theory of Support Vector Machines for Regression (SVR).

The Prediction Engine primarily interacts with the eCOP Monitoring DB and the associated Broker, the DOLFIN Optimizer and the Policy Maker. In detail,

- The interaction with the eCOP Monitoring DB is needed for acquiring the supported measurement types and their recent values.

- The interaction with the Optimizer is needed in order to inform the optimizer about the forecasted values of a particular metric (e.g. average CPU utilization of the DC) so that the generated optimization plan also considers these forecasts, implementing predictive optimization. This interaction is synchronous (upon request from the Optimizer).
- The interaction with the Policy Maker is needed to inform it about forecasted changes in the near future that may affect the efficient DC operation. This interaction is asynchronous (the Prediction Engine decides when to inform the Policy Maker).

2.1.4. Optimizer

The Optimizer is the DOLFIN component responsible for devising the actions that need to be taken in order to achieve policy-based efficiency under a changing environment. The optimizer acts on the basis of relevant Policy Maker calls, finally notifying the Policy Actuator about the actions that should be implemented. In such a framework, the Optimizer interacts with the Policy Maker, the Policy Actuator and, when needed, with the Prediction Engine. In detail:

- The interaction with the Policy Maker is needed in order to acquire the active policies that outline the targets of the DC efficiency directives (e.g. absolute energy reduction, maximization of RES usage or cost-optimization) and to acquire requests for devising new optimization plans. This interaction is synchronous (upon request from the Policy Maker).
- The interaction with the Prediction Engine is needed in order to acquire predictions of various DC/VMs metrics. These predictions are considered in the course of generating the optimization plans.
- The interaction with the Policy Actuator is needed in order to inform the latter about the actions that need to be taken in order to implement the devised optimization plans.

2.1.5. VM Priority Classifier

The VM Priority Classifier is responsible for aggregating SLA and measurements information of the VMs in a consolidated manner, facilitating the optimization process with the provision of a ranked resources (VMs) list, where the VM rank indicates the extent according to which the state of a VM can be altered without compromising its SLA restrictions. The VM Priority Classifier primarily interacts with the SLA Renegotiation controller in order to acquire the SLA states of the various VMs of the DC.

As this component has been integrated with the Optimizer (constituting a functional element of the Optimizer rather than of the overall DOLFIN platform), its interactions with the various DOLFIN components are not presented separately.

2.1.6. Policy Actuator

The Policy Actuator main objective is to translate the optimizer plan into commands that will be executed by the DCO Hypervisor Manager and the DCO Appliance Manager as defined in D3.1 [1] . The responsibilities of the Policy actuator remain untouched after the changes to the ICT Performance and Energy Supervisor defined in deliverable D3.3 [2] .

In order to achieve such goal, the Policy Actuator has been designed as a single standalone component, composed by different sub-modules and also different APIs for communications with others DOLFIN elements:

- The Northbound REST API exposing methods for requesting modifications in the DOLFIN system in terms of VMs location, VM state, air conditioning state and air conditioning temperature. This API aims to be consumed by the Optimizer in order to address plans.
- The main module containing the logic used to translate the Optimizer plans into commands that will be addressed to the DCO Hypervisor Manager and the DCO Appliance Manager. Additionally, all executed actions are logged in eCOP DB.
- The Southbound API consuming remote services exposed by DCP Hypervisor Manager and DCO Appliance Manager to perform actions, such as VM migration, VM state modification, air conditioning state modification and air conditioning temperature configuration.
- A client for the eCOP DB, in order to log in it all actions performed by the Policy Actuator.

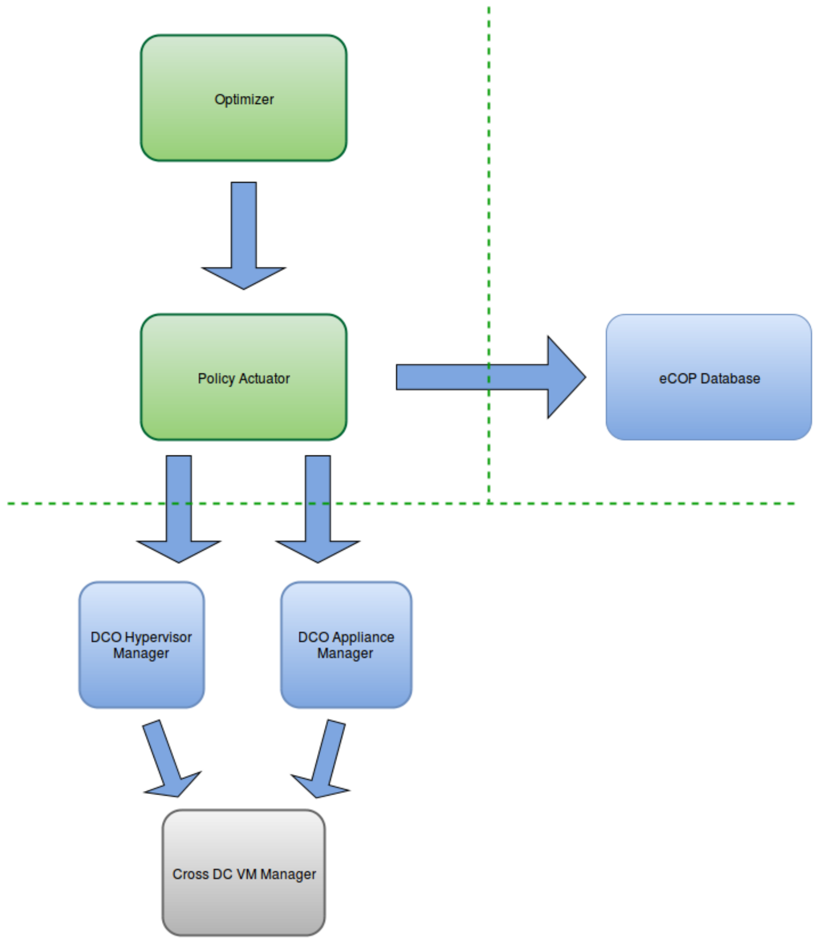


Figure 2-3 - Policy Actuator interaction diagram

3. Implementation description

3.1. Policy Maker

The Energy Policy Maker and Actuator module is a key component in the DOLFIN architecture. It is the one responsible for the schedule activation of policy enforcement, in order to evaluate event triggers coming from other modules (e.g. SmartGrid Controller, Prediction Engine), and translate these events into a change of scenario.

It coordinates requests for the initiation of an optimization process (e.g. decides the acceptance or rejection of incoming requests), based on evaluation patterns to optimize the DC energy consumption.

3.1.1. Basic concepts

The Policy Maker mode of operation could be broken down into two main levels:

- Intra-DC level: the module reacts to the changes being operated on the VMs, by assessing the status of DC topology and metrics and provides a coordination point to activate internal procedures based on policy criteria. In this context, the Policy Maker mainly reacts to external trigger provided by the Predictor Engine module as in
- Figure 3-1 (see 2.1.3 and 3.3).
- Synergetic DC level: Policy Maker interacts with Cross-DC Orchestrator in order to start and request load migration to federated DCs (Figure 3-2), or accept migration requests from other DCs; it implements closed control loop to control and validate the optimization workflow. See deliverable D4.4 [5] for more details on Cross-DC Orchestrator implementation model.

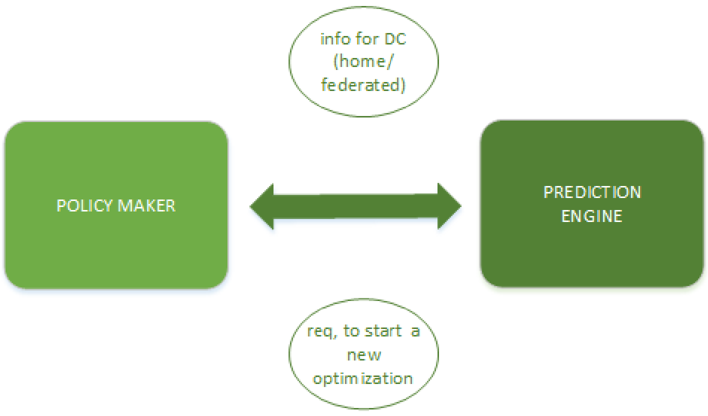

info for DC (home/ federated)
POLICY MAKER
PREDICTION ENGINE
req, to start a new optimization

Figure 3-1 - Policy Maker / Predictor Engine interaction

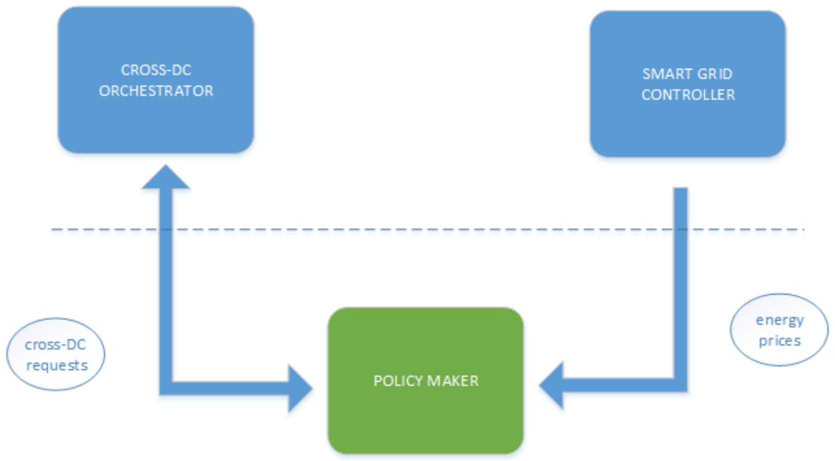


Figure 3-2 - Policy Maker SDC interactions

Through the integration with the Smart Grid Controller (Figure 3-2), the Policy Maker is pushed to adapt the operation of the DC to the energy policy, using the energy price provided by the Smart Grid environment. In this context the Smart Grid module uses RESTFul API to send price changes in a form of “absolute” energy costs. It is the responsibility of the Policy Maker to evaluate if this information is used or not to reschedule internal policy actions.

Regardless of the origin of the external requests and triggers, the processes activated during the optimization can influence, in a more or less accentuated manner, the quality of services offered to the end customers. Through the concept of "Green SLA", DOLFIN proposes a mechanism of flexible SLA, able to "tolerate" a service downgrade in favor of a saving in energy at the DC level. In this scenario, the actions defined by the optimization algorithms can exploit this flexibility of SLA, going to consciously reduce the customer's performance.

The SLA Renegotiation Control (see [5]) is the system component that deals with the treatment and monitoring of SLA. The Policy Maker communicates directly with this module through a priority events queue (based on framework RabbitMQ) in order to react quickly in case of critical events when an agreement violation is underway that may cause excessive downgrade of user SLAs (Figure 3-3).

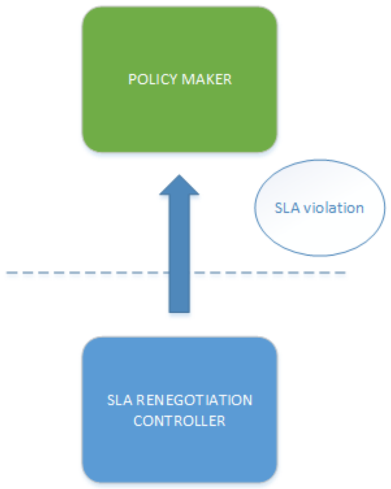


Figure 3-3 - Policy Maker / SLA Controller interaction

When the Policy Maker receives a new request from one of its triggers, it evaluates if it has to take action or not. In the latter case, it returns in attendance of new signals; in the former one it looks for a new possible optimization scenario, and retrieve new policy criteria from the Policy Repository component so to proper instruct the components that are directly involved in the building of optimization actions.

The Policy Maker has the task to react to such events adaptively. Ideally, if an event notification is bound to alter the DC working environment and performances in a suboptimal way, the Policy Maker should assemble and issue a new optimization run, and invoke the optimization process again.

In order to achieve this, an established set of possible event types is internally handled:

Event type	Issued by
Request of inbound Cross-DC workload	Cross-DC Orchestrator
Request to restrict grid power absorption	Smart Grid Controller
Lifting of a previous grid power restriction	Smart Grid Controller
Request to increase smart grid power contribution	Smart Grid Controller
Lifting of a previous smart grid power contribution request	Smart Grid Controller
Notification of predicted increasing change of the PUE KPI	Prediction Engine
Notification of predicted decreasing change of the PUE KPI	Prediction Engine
Notification of predicted increasing change of the ERE KPI	Prediction Engine
Notification of predicted decreasing change of the ERE KPI	Prediction Engine
Notification of predicted increasing change of the CER KPI	Prediction Engine

Table 3-1 - Event types by issuer

The Policy Maker will change its internal status by factoring the effect of the notified events, and may set certain reaction criteria to counter the effect of a specific event, depending on its type (see Table 3-2).

Event type	Reaction criteria (target)
Request of inbound Cross-DC workload	consolidate_local_servers: yes push_for_xdc_outbound_migrations: no
Request to restrict grid power absorption	consolidate_local_servers: yes reduce_vm_performances: yes push_for_xdc_outbound_migrations: yes boost_green_energy_reuse: yes
Request to increase smart grid power contribution	consolidate_local_servers: yes reduce_vm_performances: yes push_for_xdc_outbound_migrations: yes boost_green_energy_reuse: no
Notification of predicted increasing change of the PUE KPI	improve_kpi_pue: yes accept_xdc_migrations: yes reduce_vm_performances: no
Notification of predicted increasing change of the ERE KPI	boost_green_energy_reuse: yes improve_kpi_ere: yes accept_xdc_migrations: no

Notification of predicted increasing change of the CER KPI	accept_xdc_migrations: no consolidate_local_servers: yes

Table 3-2 - Event types reaction criteria

The compound effect of the reaction criteria set in a certain moment, concur to the make-up of the request made from the Policy Maker to the Policy Repository (see section 3.2 – Policy Repository).

These interactions make the Policy Maker to be able to adapt to different optimization scenarios, that are linked to a sets of policies. At this point it is required to converge toward a new scenario, so the selected policy must be transmitted to the Optimizer module.

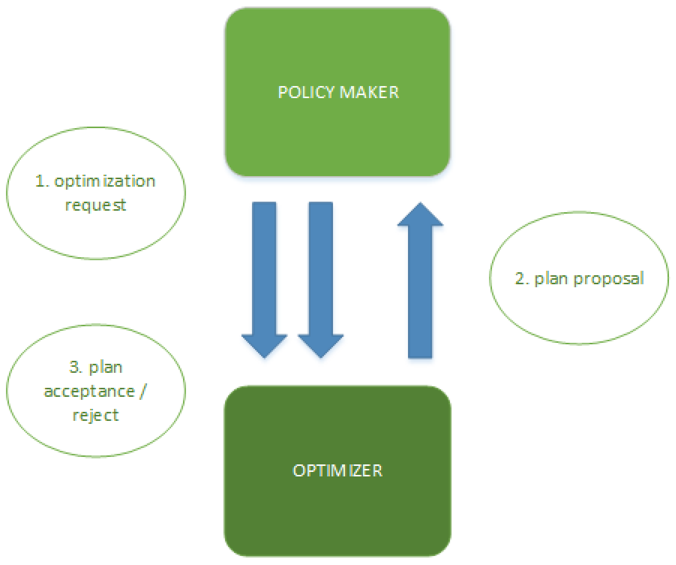


Figure 3-4 - Policy Maker / Optimizer interaction

When the Policy Maker sends the request to perform a new optimization, the Optimizer calculates an optimization plan, but it does not communicate it to the Policy Actuator. Instead, the Optimizer returns (though a specific REST API) the plan to the Policy Maker, which, in turn, decides whether the plan should be applied or not. In a synchronous way, the Policy Maker replies: if the response is positive, it sends a message telling to proceed with the application of the plan, if it is negative, to not proceed. In the latter case, the optimizer rejects the plan. In the former case, now the Optimizer forwards the plan to the Policy Actuator.

Table 3-3 provides a summary description of information passed from Policy Maker to the Optimizer and relation between optimization target and related constraints:

Target	**Description**	**Valid constraints**
energy	Find the optimal in energy consumption	• stop_vms | preserve_performance • no_xdc
cost	Optimize the use of energy in relation of energy price	• stop_vms | preserve_performance • no_xdc
performance	Makes priority to the DC computation performance	• free_cooling

sla	Optimize and preserve the customer SLAs	• <uuid>

Table 3-3 - Policy plan to the Optimizer

Where the constraints are described below:

- **stop_vms**; put VMs in stop/standby status
- **preserve_performance**; not downgrade the performance (is alternative to *stop_vms*)
- **no_xdc**; avoid cross-DC migration
- **<uuid>**; used only when target is “*sla”* and used for restore the SLA of specific VM as requested by SLA Renegotiation Controller
- **free_cooling**; used only when target is “*performance”* and tells to maximize the usage of free cooling

The JSON format of the message exchanged between the Policy Maker and the Optimizer, to indicate the new targets and constraints for optimization is the subsequent:

```
{
  "target": "energy", // energy, cost, performance, sla
  "constraints"1:
  [{
    "stop_vms","preserve_performance","no_xdc","free_cooling","<uuid>"
  }]
}
```

Other JSON message formats with WP4 components (e.g. the Smart Grid Controller) will be fully described in deliverables D4.3 and D4.4 [5] , relatives to that components.

3.1.2. Policy Maker process loop

The diagram in Figure 3-5 shows the Policy Maker process status loop, starting from the arrival of an event trigger, describing the decisions it has to make based on input data, to choose a candidate scenario regarding the supported policies.

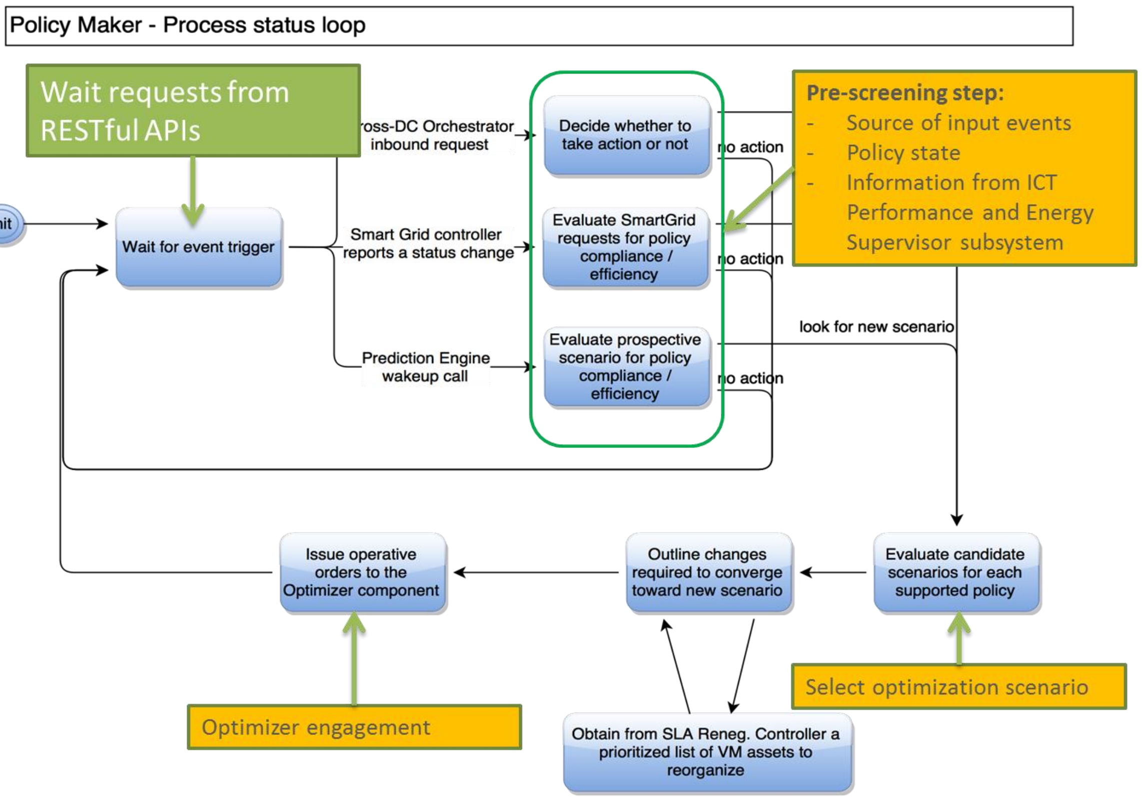

Policy Maker - Process status loop
Wait requests from RESTful APIs
Init
Wait for event trigger
ross-DC Orchestrator inbound request
Smart Grid controller reports a status change
Prediction Engine wakeup call
Decide whether to take action or not
Evaluate SmartGrid requests for policy compliance / efficiency
Evaluate prospective scenario for policy compliance / efficiency
no action
no action
no action
look for new scenario
Pre-screening step:
- Source of input events
- Policy state
- Information from ICT Performance and Energy Supervisor subsystem
Evaluate candidate scenarios for each supported policy
Select optimization scenario
Outline changes required to converge toward new scenario
Obtain from SLA Reneg. Controller a prioritized list of VM assets to reorganize
Issue operative orders to the Optimizer component
Optimizer engagement

Figure 3-5 - Process status loop

The process is structured in a *decision chain loop* where each stage is responsible to perform and apply a subsequence of actions: analysis, validation, filtering, evaluation and finally starts a communication session with southbound modules involved in the optimization process (such as the Optimizer). Following the process step outlined:

1. Startup initialization

Load Policy Maker configuration and activate the required interfaces to get inbound request from other modules:

- Cross-DC Orchestrator
- Prediction Engine wakeup call
- Smart Grid Controller

2. Wait for event block

In general, the Policy Maker is supposed to expose callable servant JSON interfaces using well-defined service endpoints. The wait block collects the requests arriving from these.

Moreover, the wait block can internally implement the mechanisms to integrate polling events. In this case, the block implements a time schedule to periodically query updates from external resources.

3. Pre-screening

In this stage the Policy Maker takes actions on the basis of three main factors:

1. the latest, relevant input events
2. the internal state of the policy processing loop (for example the Policy Maker could block new request if there in another processing ongoing)
3. the information taken from the ICT Performance and Energy Supervisor subsystem

The screening of the events may result in dropping the inbound request and ignoring it, or schedule a new action.

Example of processes in the Pre-screening step:

Cross DC

Input:

- *inbound VM transfer request; should be contain information of the VM to be host*

Process:

- *Query the InfoDB to determine if inbound cross-DC transfers are currently acceptable*
- *Verify if the Cross-DC transfer would result in a conflict with some other active constraint/limitation (such as limitations imposed by the Smart Grid Controller)*

Smart Grid

Input:

- *change energy price of +- X%*

Process:

- *Verify the availability of Cross-DC workload shifting*
- *Verify local energy production*
- *Verify prospective elision of other current constraints*

4. Policy selection

The action produced in the pre-screening step is translated in a procedure to query the list of possible policies. For each action, presumably, there is a one-to-many relationship with the policies, in this case the policy selection could returns a prioritized ordered list of available policies that are evaluated in the next step in the policy evaluator block.

5. Policy evaluator

In this step evaluates the prioritized list of policies, to try to make a screening of which policy is best applied, considering the current status of the DC and other factors.

This stage is the final building block for the policy rule, where all needed parameters and constraints are valorised.

6. Optimizer engagement

In this step the Policy Maker pass the control to the Optimizer module and stop to processing further requests until the conclusion of optimization. The requests to the Optimizer will have the format of JSON message as defined in subsection 3.1.4.

In all cases, the Policy Maker will keep listening for inbound requests to handle during the optimization process. It may opt to immediately react to high priority event (such as critical event form SLA Renegotiation Controller), and this may result in a stop command sent to the Optimizer about a previously ongoing optimization.

Request made by the Policy Maker can happen to be ineffective, not applicable for a number of reasons, or plain wrong. It is a perfectly legal action for the Optimizer, to report back errors and conditions of refusal about a specific run optimization run.

3.1.3. Module Dependencies

The Policy Maker are few strictly dependencies that provide a minimal set of functionalities needed for running the module:

- Policy Repository; needed to initialize the process loop
- eCOP DB; to query useful information status used during the event evaluation

Regardless these dependencies, the work performed by the Policy Maker cannot be discerned without the integration with the external modules which provide inbound requests:

- Cross-DC Orchestrator (1.4.12 in Fig. 2-1)
- Prediction Engine wakeup call (1.4.14 in Fig. 2-1)
- Smart Grid Controller interface (1.4.8, 1.4.9 in Fig. 2-1)
- SLA Renegotiation controller interface

The communication between the Policy Maker with external (WP4) components listed have been achieved with RESTful APIs (described in section 3.1.4) except for the one with the SLA Renegotiation Controller, which is handled by a RabbitMQ.

3.1.4. Service endpoints and data model

A list of main Policy Maker REST APIs follows, with a brief description.

	Prediction Engine interface
Description	Receives a wakeup call from the Prediction Engine about a suggested change in metrics
Endpoint Name	/policy/steer/{measure}
Allowed Methods	POST
Body	Notifies old, predicted and deadline values of the kpi specified in the endpoint
Provides response	Yes
Response Parameters	Status (“accepted” / “refused”), eta
Response code	200, 400
Requester	Prediction Engine
Example	{

	"old_value": 0, "expected_value": 0, "deadline": 0 }

Table 3-4 - Policy Maker: Predictor Engine API

	Cross-DC Orchestrator Interface
Description	Handles a cross-DC preauthorization request for an inbound workload migration
Endpoint Name	/policy/xdc/inbound
Allowed Methods	POST
Body	Specifies origin and destination DCOs, and describes assests
Provides response	Yes
Response Parameters	Status (“go_ahead”, “denied”), xdc_endpoint, xdc_preauth_key
Response code	200, 400
Requester	Cross-DC Orchestrator

Example	{
	"origin_dco": "dco0",
	"dest_dco": "dco1",
	"assets": [{
	"type": "vm",
	"uuid": "0123",
	"specs": {
	"vcpus": 1,
	"ram_mbs": 1024,
	"compressed_size_mbs": 1024,
	"uncompressed_size_mbs": 2048,
	"logical_size_mbs": 2048
	}
	}]
	}

Table 3-5 - Policy Maker: Cross-DC POST API

	Cross-DC Orchestrator Interface
Description	Reports the status of the Cross-DC endpoint and current availability for inbound requests
Endpoint Name	/policy/xdc/inbound
Allowed Methods	GET
Body	Provides status of the endpoint
Provides response	Yes
Response Parameters	Status (“available”, “unavailable”)
Response code	200
Requester	Cross-DC Orchestrator
Example	{

```
                                    "status": "available",
                                    "dc": "dc1",
                                   }
```

Table 3-6 - Policy Maker: Cross-DC GET API

3.1.5. Deployment and installation

Below you may find the installation instructions for Ubuntu systems. Please note that this installation procedure has been verified on Ubuntu 14.04.x x86_64 systems.

Sources are available on a Git repository at:

http://stash.i2cat.net/scm/dol/ecop_policy_maker.git

Preparing system:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install git python3.4 python-pip python-dev build-essential
rabbitmq-server
```

Cloning Git repository for DOLFIN Utils modules:

```
$ git clone http://stash.i2cat.net/scm/dol/dolfin_utils.git
```

Installing module using PIP tool:

```
pip install .
```

Launching Policy Maker application:

```
$ python -m dolfin.pmaker.server
```

When launching the Policy Maker server some useful options are available:

- **--debug** -> enables better error reporting
- **--logging=debug**
- **--server_port=<PORT_NUMBER>**
- **--server_addr=<SERVER_ADDRESS>** -> use 0.0.0.0 to bind on all interfaces

3.1.6. Testing the installation

The server component should be running (by default) at port 8080 of your machine. You may test it by issuing the following command to see if the documentation page is properly fetched:

```
$ curl locahost:8080/docs
```

Remember to satisfy prerequisites listed in previous paragraph:

- Git
- Python 3.4+
- RabbitMQ
- dolfin-utils python package (will be explicitly required during the install phase)

Dependencies are included in *requirements.txt* file.

3.2. Policy Repository

The Policy Repository maintains information regarding all possible policies available to conduct the operation efficiency resource management.

This repository stores the energy policy used internally by the eCOP functions. The Policy Maker queries this repository, looking for the available policies in order to extract the best policy subset (that will be used later to determine the optimization logic to apply), according to the situation encountered.

The selection is performed using sets of criteria based on the inputs of the Policy Maker and its knowledge of the DC status. An energy policy is then "translated" in algorithms by the Optimizer Engine.

3.2.1. Basic concepts

There has to be a one-to-one relationship between the contents of the Policy Repository and the strategies (algorithms) known by the Optimizer module. Policies have to be expressed in a form of inputs which will clearly specify:

1. the objective (*target*) of the policy (its name / identifier).
2. A set of parametric constraints, affecting the behaviour of how the specific policy is planned for implementation by the Optimizer. The constraints can, for example, state the total deviation from the current status, which the system seeks to achieve, impose or deny cross-DB cooperation.

All the information that characterize a policy are then re-processed and adapted by the Policy Maker in a form suitable for the Optimizer. This second step of translation is needed to maintain a separation between the policy definition and the policy actuation as represented, understood and used internally by the Optimizer.

At the level of Policy Repository, the policy will refer to specific parameters which are globally available for querying in the ECOP DB (es. DCPWR, HVACPWR, HVACflow, etc), stating the new value the system wants to reach. These parameters will be direct measures or calculated KPIs. Moreover, the Policy Repository is able to estimate the effect of the activation of each policy, in order to compare it with the targets currently set (and required) by the Policy Maker.

Each policy in the Policy Repository is identified by a label, and has a number of constraints and intended effects (Table 3-8). The Table 3-7 provides a representation of defined constraints used for the policy definition:

Constraint name	**Intended effect**	**Range of possible values**
avoid_standbys	Avoid noticeable (SLA infringing) VM standbys	Yes / no / not set
no_xdc_inbound_migrations	Reject any and all inbound cross-DC workload migration requests	Yes / no / not set
dnm_free_air_cooling_boost	Prevent initiation of intra-DC VM migration to achieve better free air cooling exploitation	Yes / no / not set
dnm_for_green_server_boost	Prevent initiation of intra-DC VM migration to achieve better green server exploitation	Yes / no / not set
dnm_for_server_consolidation	Prevent initiation of intra-DC VM migration to achieve better server consolidation	Yes / no / not set
do_not_stop_vms	Prevent VM shutdowns or suspensions	Yes / no / not set
preserve_vm_performances	Prevent engaging in actions which may degrade actual VM performances (observe SLA performance limits)	
push_for_kpi_pue_target	Take actions in order change the PUE metric value towards the wanted target	-100→+100%/not set
push_for_kpi_ee_target	Take actions in order change the EE metric value towards the wanted target	-100→+100%/not set
push_for_kpi_cet_target	Take actions in order change the CET metric value towards the wanted target	-100→+100%/not set
push_for_kpi_apc_target	Take actions in order change the APC metric value towards the wanted target	-100→+100%/not set
push_for_kpi_apc_ren_target	Take actions in order change the APC REN metric value towards the wanted target	-100→+100%/not set
push_for_kpi_dci_target	Take actions in order change the DCI metric value towards the wanted target	-100→+100%/not set
push_for_kpi_ere_target	Take actions in order change the ERE metric value towards the wanted target	-100→+100%/not set

push_for_kpi_gridii_target	Take actions in order change the Grid II metric value towards the wanted target	-100→+100%/not set
push_for_xdc_outbound_migrations	Try to offload work to federated DCs	Yes / no / not set

Table 3-7 - Policies constraints

The internal setup of the Policy Repository allows any defined policy to have a number of consequent effects associated to it. An example of the make-up of defined policies and correlated effects is:

Policy name	**Effects**	**Constraints**
Air cooling reuse	Perform relocation of internal VMs to better optimize the utilization of free air cooling.	push_for_kpi_ere_target: <<0% preserve_vm_performances: yes do_not_stop_vms: yes
Local only	apply the optimization only to local VMs, and within the DC boundary	no_xdc_inbound_migrations: yes push_for_xdc_outbound_migrations: no push_for_kpi_pue_target: <<0%
Receive XDC workload	local DC will accept requests form federated DCs inconditionally	preserve_vm_performances: yes do_not_stop_vms: yes avoid_standbys: yes push_for_kpi_pue_target: <<0%

Table 3-8 - Policy definition

Upon request, the Policy Repository evaluates the set of targets passed from the Policy Maker with the intended effects of every defined policy. Each target will yield a score (which can be negative if the effect of a policy is counter-productive) and will concur in assigning a fitting value to each policy.

By iterating the evaluation algorithm upon all targets, for all constraints for all the defined policies, the Policy Repository is then able to rank its defined policies and select the best fit for the present set of targets. An answer is sent to the Policy Maker, with the resulting selected policy and its associated constraints. In the section 3.2.2 an overview to the workflow is provided, describing the procedure used internally for policy selection.

Then the Policy Maker is in charge of the right policy translation to produce a proper formatted message used to communicate with the Optimizer (see section 3.1.4 for API description).

3.2.2. Policy selection workflow

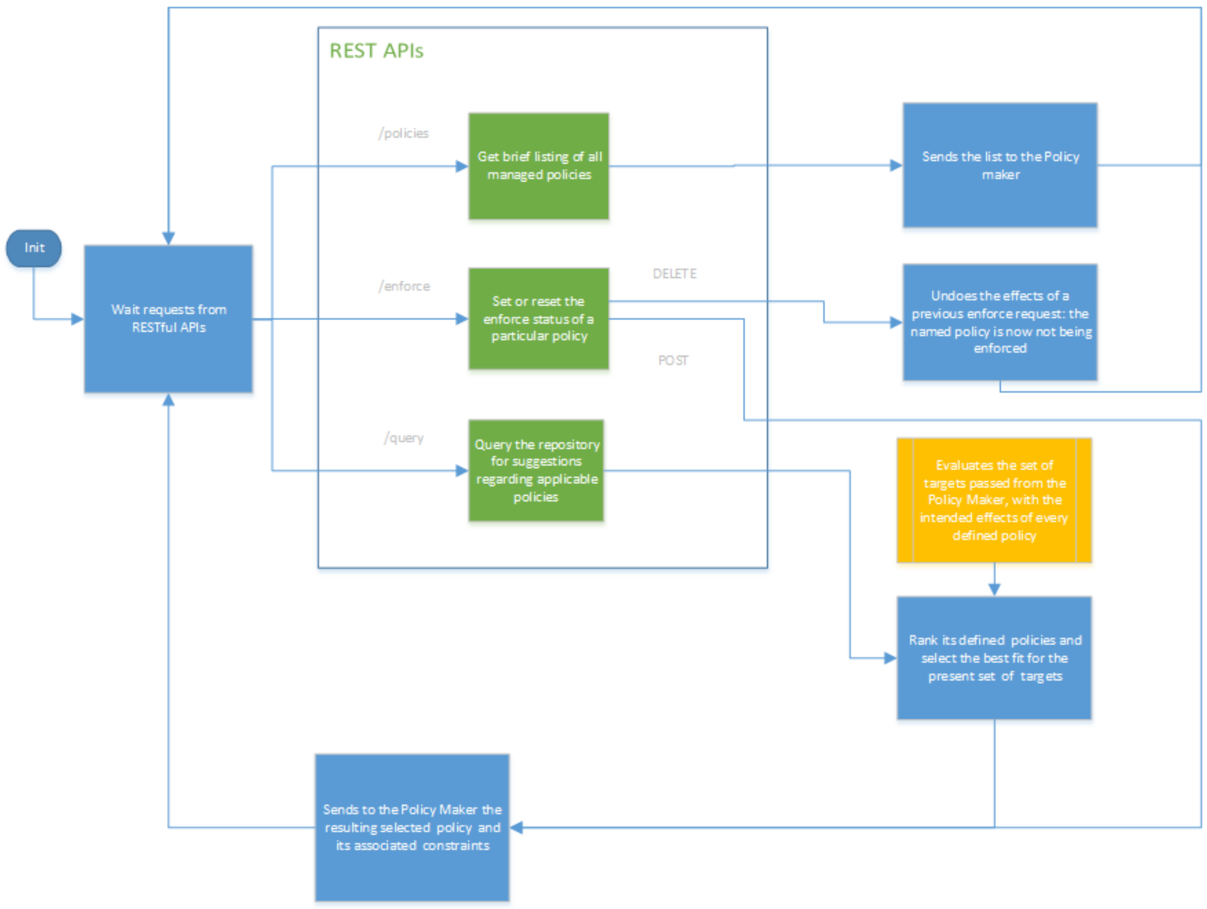

REST APIs
Init
Wait requests from RESTful APIs
/policies
Get brief listing of all managed policies
/enforce
Set or reset the enforce status of a particular policy
/query
Query the repository for suggestions regarding applicable policies
DELETE
POST
Sends the list to the Policy maker
Undoes the effects of a previous enforce request: the named policy is now not being enforced
Evaluates the set of targets passed from the Policy Maker, with the intended effects of every defined policy
Rank its defined policies and select the best fit for the present set of targets
Sends to the Policy Maker the resulting selected policy and its associated constraints

Figure 3-6 - policy selction

The process structure is a *decision chain loop*, where each stage is responsible to perform and apply a subsequence of actions: analysis and evaluation. Finally, the module replies to the caller (the Policy Maker) the results of the requested operation. Following the process step outlined:

1. **Startup initialization**

Load Policy Repository configuration and activate the required interfaces to get inbound request from Policy Maker

2. **Wait for event block**

The Policy Repository is supposed to expose callable servant JSON interfaces using well-defined service endpoints. The wait block collects the requests arriving from these.

3. **Pre-screening**

In this stage the Policy Repository takes actions on the basis of the endpoint called from the Policy Maker:

a. *policies*; the Policy Maker is requesting the list of the policies currently active
b. *enforce*; the Policy Maker is forcing the Policy Repository to get a specific policy, no matter what, or is undoing a previous policy enforcement
c. *query*; the Policy Maker instructs the Repository about a specific scenario requirement, so that the repository engine can measure the effectiveness of the active policies and suggest the best one to be applied

So, the screening of the events may result in different actions the Policy Repository have to take during its chain loop.

4. Event Processor

The action produced in the pre-screening step is translated in a different procedure to for each action; with simpler or more complex actions based on the flow to follow.

a. Policies

The Policy Repository is polled to return a listing of all managed policies; so it returns a message to the Policy maker with a plain list of these elements. Then, the process loop returns to the wait for event block.

b. Enforce

The Policy Repository exposes two endpoints: to make caller (so the Policy Maker) able to force a specified policy or to undo a previous enforce request.

In the latter case the Repository makes the named policy not to be enforced anymore, undoing a previous command sent to the module. So the Policy Repository could restore its behaviour to a standard one, not affected from an outside already made decision. Then it returns to the wait for event block.

In the former case, instead, it replies to the Policy Maker the constraints of the forced policy, like it was the result of the policy selection. Then it could return to the wait for event block.

c. ***Query***

The Policy Repository receives a request from the Policy Maker with the intended effects of every defined policy, in order to evaluate the set of targets. This could be considered as the standard behaviour of the Policy Repository inside the Policy Maker and Actuator module. So, each target evaluated from the process, will yield a score (which can be negative if the effect of a policy is counter-productive) and will concur in assigning a fitting value to each policy. By iterating the evaluation algorithm upon all targets, for all constraints for all the defined policies, the Policy Repository is then able to rank its defined policies and select the best fit for the present set of targets. At this point it replies to the Policy Maker, with the resulting selected policy and its associated constraints; then the process loop returns to the wait for event block.

3.2.3. Module Dependencies

The Policy Repository module interacts directly with the Policy Maker component, which have been described in previous chapter. No other directly dependencies are defined and needed to run the

Policy Repository. In any case the Policy Repository is to be consider required components of the DOLFIN architecture, due to this "leads" the decision taken at the level of the Policy Maker.

3.2.1. Service endpoints and data model

A list of main Policy Repository REST APIs follows.

	Inspect
Description	Obtain the list of currently active policies
Endpoint Name	/policies
Allowed Methods	GET
Body	None
Provides response	Yes
Response Parameters	Name, Description, IsActive, IsEnforced, ConflictsWith, Constraints
Response code	200
Requester	Policy Maker

Example	`{`
	`  "name": "policy_01",`
	`  "descr": "description",`
	`  "is_active": true,`
	`  "is_enforced": true,`
	`  "conflicts_with": [],`
	`  "constraints": [ {`
	`    "name": "constraint_01",`
	`    "descr": "constraint_description",`
	`    "value_type": "true",`
	`    "supported": true`
	`   } ]`
	`}`

Table 3-9 - Policy Repository: Inspect policies API

	Inspect
Description	Obtain the list of currently enforceable constraints
Endpoint Name	/contraints
Allowed Methods	GET
Body	None
Provides response	Yes
Response Parameters	Name, Description, ValueType, Supported
Response code	200,
Requester	Policy Maker

Example	`{`
	`  "name": "constraint_01",`
	`  "descr": "description",`
	`  "value_type": "true",`

	"supported": true }

Table 3-10 - Policy Repository: Inspect constraints API

	Enforce
Description	Force the application of a specific policy, no matter what
Endpoint Name	/enforce/{policy_name}
Allowed Methods	POST
Body	None
Provides response	Yes
Response Parameters	Name, Description, IsActive, IsEnforced, ConflictsWith, Constraints
Response code	200, 404
Requester	Policy Maker

Example	{
	"name": "policy_01",
	"descr": "description",
	"is_active": true,
	"is_enforced": true,
	"conflicts_with": [],
	"constraints": [{
	"name": "constraint_01",
	"descr": "constraint_description",
	"value_type": "true",
	"supported": true
	}]
	}

Table 3-11 - Policy Repository: enforce policy API

	Enforce
Description	Undoes the effects of a previous enforce request (the named policy is now not being enforced)
Endpoint Name	/enforce/{policy_name}
Allowed Methods	DELETE
Body	None
Provides response	Yes
Response Parameters	Name, Description, IsActive, IsEnforced, ConflictsWith, Constraints
Response code	200
Requester	Policy Maker
Example	{ "name": "policy_01", "descr": "description", "is_active": true, "is_enforced": true,

	"conflicts_with": [],
	"constraints": [{
	  "name": "constraint_01",
	  "descr": "constraint_description",
	  "value_type": "true",
	  "supported": true
	 }]
	}

Table 3-12 - Policy Repository: enforce policy DELETE API

	Query
Description	Instruct the repository about a specific scenario requirement, so that the repository engine can measure the effectiveness of the active policies and suggest the best one to be applied
Endpoint Name	/query
Allowed Methods	POST

Body	{
	 "push_for_xdc_outbound_migrations": true,
	 "reduce_vm_performances": true,
	 "reduce_vm_performances_sla_infringing": true,
	 "consolidate_local_servers": true,
	 "boost_green_energy_reuse": true,
	 "improve_kpi_ere": true,
	 "improve_kpi_pue": true,
	 "accept_xdc_migrations": true
	}

Provides response	Yes
Response Parameters	Name, Description, IsActive, IsEnforced, ConflictsWith, Constraints
Response code	200
Requester	Policy Maker

Example	{
	  "name": "policy_01",
	  "descr": "description",
	  "is_active": true,
	  "is_enforced": true,
	  "conflicts_with": [],
	  "constraints": [{
	    "name": "constraint_01",
	    "descr": "constraint_description",
	    "value_type": "true",
	    "supported": true
	   }]
	}

Table 3-13 - Policy Repository: query for policy scenario API

3.2.2. Deployment and installation

Below you may find the installation instructions for Ubuntu systems. Please note that this installation procedure has been verified on Ubuntu 14.04.x x86_64 systems.

Sources are available on a Git repository at:

http://stash.i2cat.net/scm/dol/ecop_policy_repository.git

Preparing system:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install git python3.4 python-pip python-dev build-essential
```

Cloning Git repository for DOLFIN Utils modules:

```
$ git clone http://stash.i2cat.net/scm/dol/dolfin_utils.git
```

Installing module using PIP tool:

```
pip install .
```

Launching Policy Repository server:

```
$ python -m dolfin.policies.server
```

When launching the Policy Repository server some useful options are available:

- **--debug** -> enables better error reporting
- **--logging=debug**
- **--server_port=<PORT_NUMBER>**
- **--server_addr=<SERVER_ADDRESS>** -> use 0.0.0.0 to bind on all interfaces

3.2.3. Testing the installation

The server component should be running (by default) at port 8082 of your machine. You may test it by issuing the following command to see if the documentation page is properly fetched:

```
$ curl localhost:8082/docs
```

Remember to satisfy prerequisites listed in previous paragraph:

- Git
- Python 3.4+
- dolfin-utils python package (will be explicitly required during the install phase)

Dependencies are included in *requirements.txt* file.

3.3. Prediction Engine

The Prediction Engine is responsible for offering prediction services over the sets of data monitored by eCOP, either raw (e.g. average CPU load, RAM utilization or energy consumption) or calculated (e.g. metrics calculated from the eCOP KPI Engine).

To implement the predictions, the Prediction Engine bases its operation on the well-established Support Vector Machines theory, particularly applied for Regression, namely SVR. The prediction processes may run either synchronously (upon request from another entity or component) or asynchronously (e.g. every 5 minutes). When asynchronously invoked, the Prediction Engine calculates the underlying model error and, if higher than a configurable lower threshold, it notifies the Policy Maker to inform it about urgent changes in the expected DC behaviour.

3.3.1. Basic concepts

The prediction engine has been developed to provide forecasts for all measurement types and DC infrastructure resources that are supported by the eCOP Monitor DB. Measurement types supported by the eCOP Monitor DB and, hence, by the Prediction Engine, are shown in the following table.

calculated	disk.read.requests	energy
compute.node.cpu.frequency	disk.read.requests.rate	hvac.air_volume
compute.node.cpu.percent	disk.write.bytes	hvac.heat_transfer
cpu_util	disk.write.bytes.rate	hvac.temperature_drop
disk.read.bytes	disk.write.requests	memory
disk.read.bytes.rate	disk.write.requests.rate	memory.usage
network.incoming.bytes	network.incoming.bytes.rate	network.outgoing.bytes
network.outgoing.bytes.rate	power	temperature

Table 3-14 - Predictor Engine measurement types

The Prediction Engine features an auto-discovery mechanism to create models of measured data, without necessitating explicit configuration. Every night, the Prediction Engine polls the eCOP DB Broker for all measurement types and resources (e.g. servers, VMs, racks etc.) and gets the relative measurements, spanning the last 14 days[1]. Then, a training model based on Support Vector Machines for Regression (SVR) is applied over the complete set of acquired data in order to get an accurate representation of their trends, after combining them to meaningful tuples of type [*resource, measurement_type*]; as an example, supposing that for a specific physical server there exist measurements related to energy consumption and CPU utilization only, the Prediction Engine will generate two models, one pertaining to the CPU utilization of the server and one related to its energy consumption. Following this approach, the Prediction Engine is able to provide information for a vast number of target [*resource, measurement_type*] forecasting tuples. The resulted models are, next, stored in appropriate directory and file hierarchies so that predictions can be acquired almost instantly, without having to re-train the Prediction Engine. The parameters of the SVR machines are selected after employing parameter space exploration and cross-validation, to extract the parameter values that minimize the prediction error, i.e. present the best fit for the provided input values. In any case, the kernel is statically set to Radius Basis Function (RBF). The directory structure used for storing the models is, by default, following the following convention:

`/opt/ecop_prediction_engine/api/oracle/<measurement_type>/<resource>/<resource_id>`, e.g.:

```
ubuntu@dolfin-dev:~$ tree -L 3 /opt/ecop_prediction_engine/api/oracle/
. . .
└── temperature
    ├── rack
    │   ├── 1
    │   ├── 2
    │   ├── 3
    │   └── 4
    ├── room
    │   ├── 1
    │   ├── 2
    │   └── 3
    ├── server
    │   ├── SN-1
    │   ├── SN-2
    │   ├── SN-3
    │   ├── SN-4
    │   ├── SN-5
    │   └── SN-6
    └── vm
        ├── test_uuid
        ├── test_uuid_1
        ├── test_uuid_2
        ├── test_uuid_3
        ├── test_uuid_4
        ├── test_uuid_5
        ├── uuid_test_6
        ├── uuid_test_7
        ├── uuid_test_8
        └── uuid_test_9
. . .
```

[1] The limited span is intended because the actual data already acquired do not span more than a year, hence capturing seasonality at yearly levels is not possible. When the acquired data span more than 12 months, the learning procedure of the Prediction Engine will be slightly altered to consider all available data, over-weighting most recent ones.

Evidently, as not all measurement types are applicable to all resources (e.g. a VM cannot be measured in terms of temperature), the relevant directories are empty; the Prediction Engine only creates models for the tuples `[measurement_type, resource, resource_id]` that are metered.

Apart from SVR, we also experimented with Neural Networks and various time series analysis models such as the ARMA, ARMAX and ARIMA models. The choice of SVR in favour of the rest of evaluated techniques was made because SVR exhibited satisfactory performance in terms of both accuracy using only limited sets of data and speed of forecasting procedures. Indicatively, the prediction of the expected power demand of a single rack for four hours in the future (starting from the time of the request) and quantized in timeframes of five (5) minutes (producing 48 prediction values in total) takes approximately 55ms (including network transfer time).

The Prediction Engine features two modes of operation, an interactive and an automated one, depending on whether the prediction request was performed by another component (e.g. the Policy Maker) or the Prediction Engine itself, respectively.

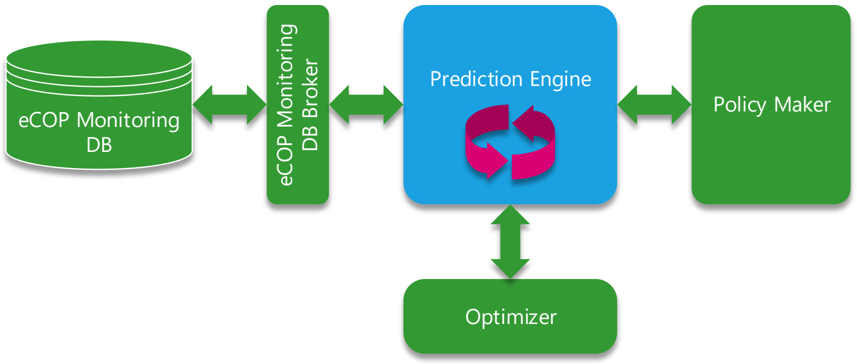
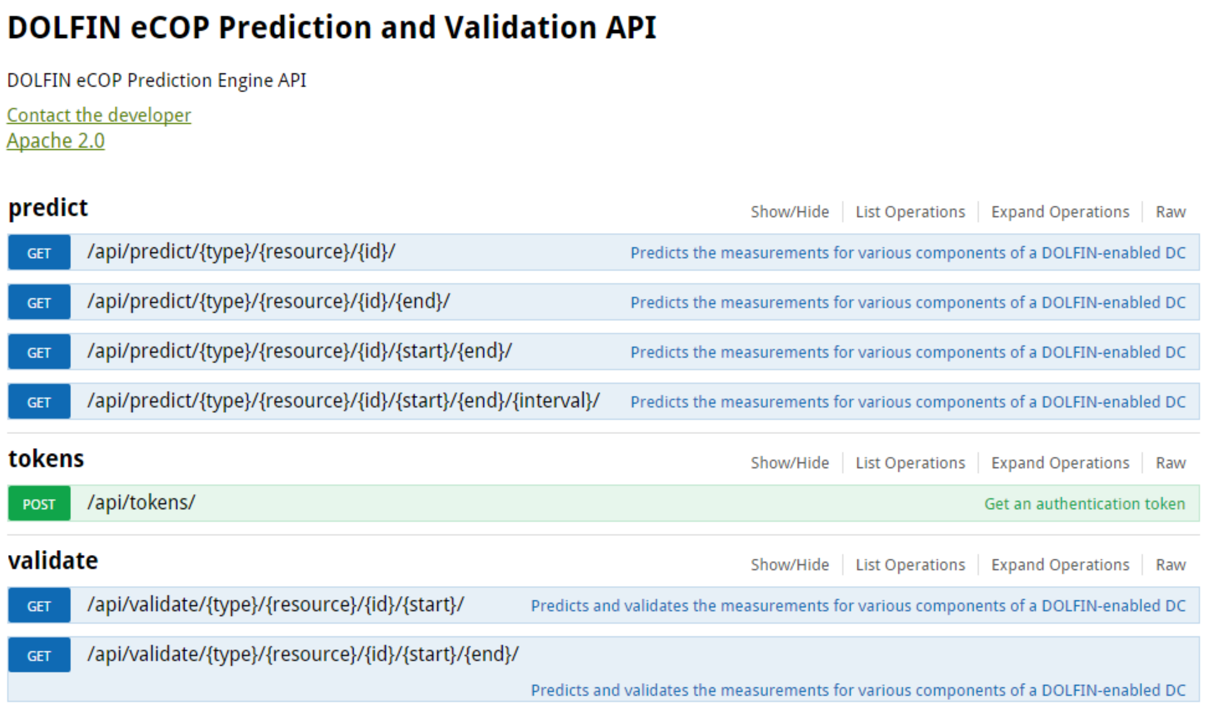


Figure 3-7 – Interactions of the Prediction Engine with the rest of the DOLFIN eCOP components.

As far as the automated operation mode of the Prediction Engine is concerned, a periodical task is activated every five (5) minutes that checks the hourly forecasts of the prediction engine and correlates them with the actual relevant measurements acquired over the last hour (prior each request, in a windowed mode) by the eCOP Monitor DB. If the relative difference between the two time-series is larger than 15%, then the Prediction Engine notifies the Policy Maker of the event.

Regarding the interactive Prediction Engine mode, following the paradigm of the eCOP Monitor DB, the core functionality is exposed via a RESTful web service exposing 7 application-oriented endpoints, supported by 26 administration-oriented ones. Further, all APIs are secured against unauthorized access through the employment of time-expiring authentication and authorization tokens. Figure 3-8 presents the application-oriented API endpoints exposed by the RESTful interface of the Prediction Engine. The software used for generating this online, dynamic documentation is Swagger[2] together with its associated web-interface, Swagger-UI[3].

[2] http://swagger.io/
[3] http://swagger.io/swagger-ui/


DOLFIN eCOP Prediction and Validation API

DOLFIN eCOP Prediction Engine API

Contact the developer
Apache 2.0

predict — Show/Hide | List Operations | Expand Operations | Raw

GET /api/predict/{type}/{resource}/{id}/ — Predicts the measurements for various components of a DOLFIN-enabled DC

GET /api/predict/{type}/{resource}/{id}/{end}/ — Predicts the measurements for various components of a DOLFIN-enabled DC

GET /api/predict/{type}/{resource}/{id}/{start}/{end}/ — Predicts the measurements for various components of a DOLFIN-enabled DC

GET /api/predict/{type}/{resource}/{id}/{start}/{end}/{interval}/ — Predicts the measurements for various components of a DOLFIN-enabled DC

tokens — Show/Hide | List Operations | Expand Operations | Raw

POST /api/tokens/ — Get an authentication token

validate — Show/Hide | List Operations | Expand Operations | Raw

GET /api/validate/{type}/{resource}/{id}/{start}/ — Predicts and validates the measurements for various components of a DOLFIN-enabled DC

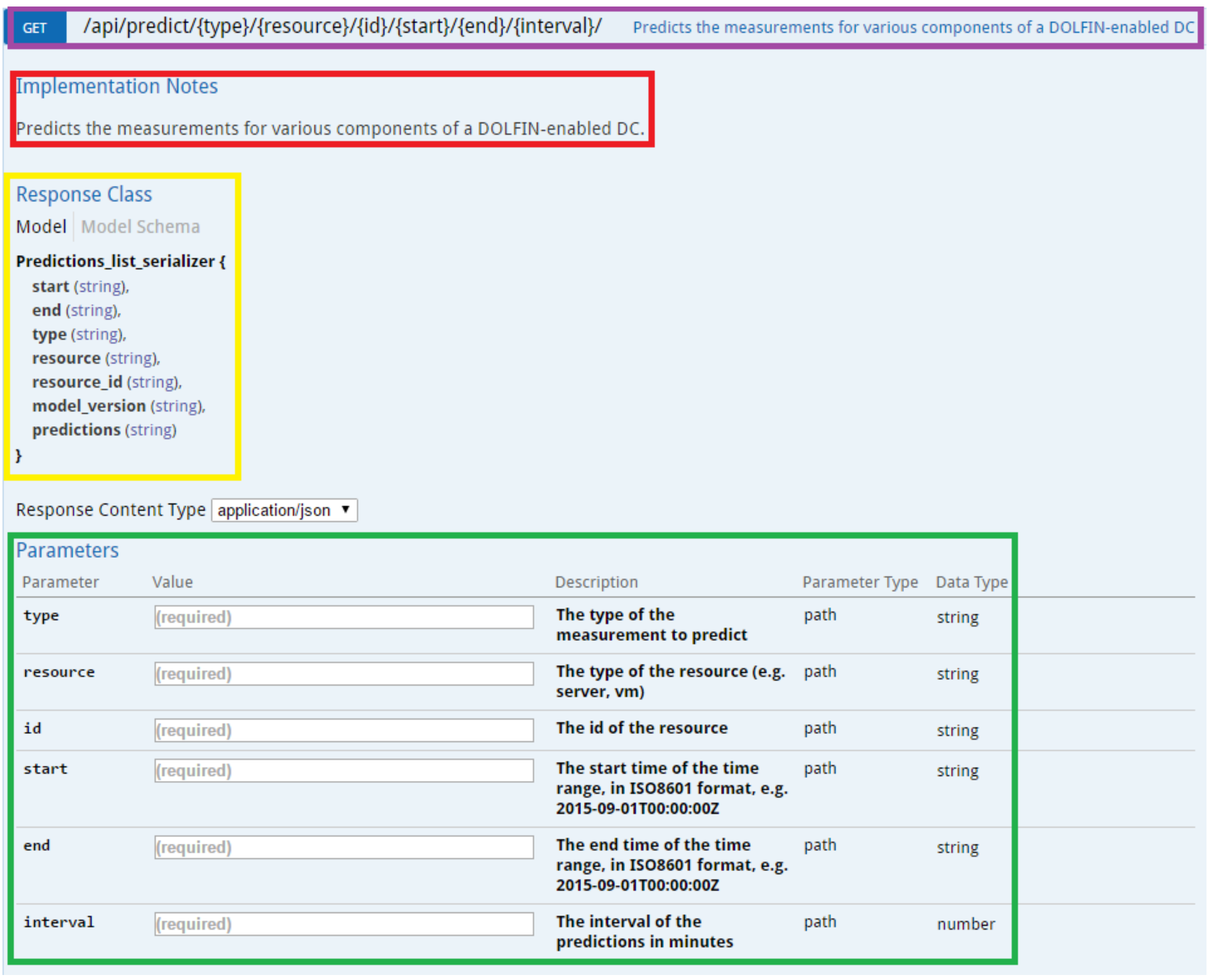
GET /api/validate/{type}/{resource}/{id}/{start}/{end}/ — Predicts and validates the measurements for various components of a DOLFIN-enabled DC

Figure 3-8 - RESTful API exposed by the Prediction Engine

As depicted in Figure 3-8 and with respect to the endpoints exposed by the Prediction Engine, two main categories of API services are supported, accompanied by a service for Authentication and Authorization services (exposed under the relative path /api/tokens/). The two main service classes refer to predictions provisioning and predictions validation, detailed as follows:

- The on-demand predictions provisioning services (grouped under the relative path /api/predict/) offer predictions for the various supported measurement types, over various time frames featuring (optionally) various forecasting points intervals.
- The on-demand predictions validation services (grouped under the relative path /api/validate/) offer validity services with respect to the derived predictions. In other words, by using these services one can get a model-based prediction of a measurement type for a given resource for a past period of time, compare it with the actual measurements and get some insightful statistical information regarding the quality of the model-based prediction. The validation is subjective as model updates happen regularly but not in real time; hence, performing a prediction in the past (after the model has been created) is as valid as generating a prediction for a time period in the future. The statistical measures supported by the validation endpoints are the following: R^2, Mean Square Error, Relative Mean Square Error and Relative Standard Deviation.

Figure 3-9 presents an overview of the documentation of a single API service exposed by the Prediction Engine. As can be easily observed, information about the purpose of the endpoint, the supported HTTP method(s) and the various path parameters required for properly invoking the service are provided, boxed with magenta colour; the type of the parameters and their description are also provided, as depicted framed in a green box. Further implementation notes and guidelines of API Service usage are provided through a dedicated section framed with a red box. Moreover, the details of the response class provided by the service are given in the yellow box.



GET /api/predict/{type}/{resource}/{id}/{start}/{end}/{interval}/ Predicts the measurements for various components of a DOLFIN-enabled DC

Implementation Notes

Predicts the measurements for various components of a DOLFIN-enabled DC.

Response Class

Model | Model Schema

```
Predictions_list_serializer {
  start (string),
  end (string),
  type (string),
  resource (string),
  resource_id (string),
  model_version (string),
  predictions (string)
}
```

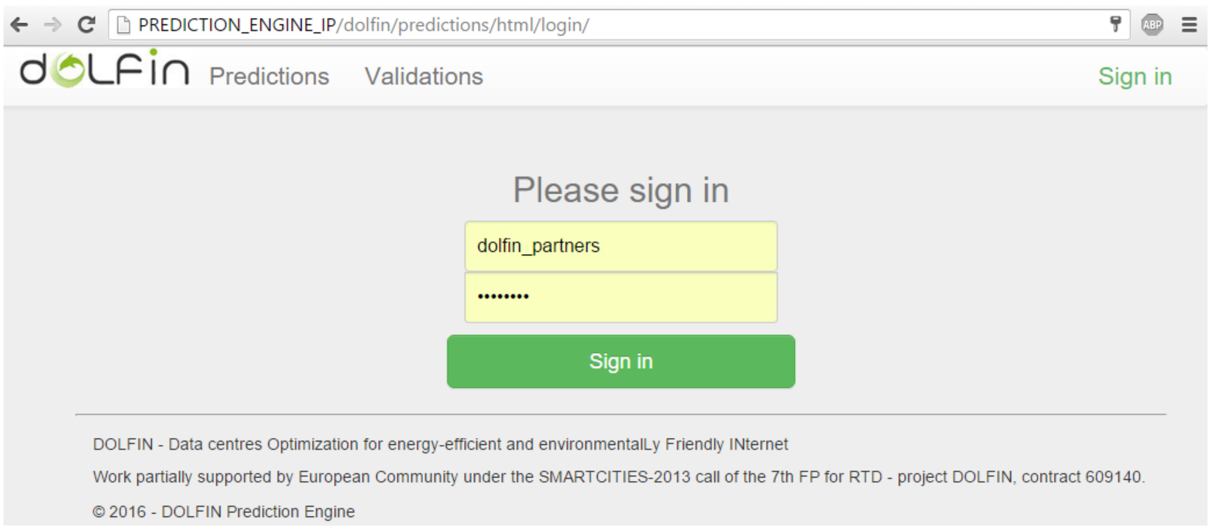
Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
type	(required)	The type of the measurement to predict	path	string
resource	(required)	The type of the resource (e.g. server, vm)	path	string
id	(required)	The id of the resource	path	string
start	(required)	The start time of the time range, in ISO8601 format, e.g. 2015-09-01T00:00:00Z	path	string
end	(required)	The end time of the time range, in ISO8601 format, e.g. 2015-09-01T00:00:00Z	path	string
interval	(required)	The interval of the predictions in minutes	path	number

Figure 3-9 - Overview of the Prediction Engine online documentation

Apart from the online documentation system detailing the exposed Prediction Engine API structure and usage, the Prediction Engine also features a dashboard to offer a visualized reference of the generated predictions and the validations of them. To visit the dashboard, one should navigate to the URL http://<PREDICTION_ENGINE_IP>/dolfin/predictions/html/login/, unless otherwise configured (see 3.3.3.2 for details on how to properly configure the login redirection URL).



DOLFIN - Data centres Optimization for energy-efficient and environmentalLy Friendly INternet

Work partially supported by European Community under the SMARTCITIES-2013 call of the 7th FP for RTD - project DOLFIN, contract 609140.

© 2016 - DOLFIN Prediction Engine

Figure 3-10 - The prediction engine default login page

When logged in, the user is able to either instruct the prediction engine to generate a new prediction for a certain (valid) [`measurement_type`, `resource_type`, `resource_id`] tuple by clicking on the Predictions button of the navigation bar, or generate a validation of the existing models, by pressing the Validations button. To ease the determination of a new resource identification tuple, the dashboard features relevant, auto-updating dropdown menus, as depicted in Figure 3-10. A similar view is available for the forecasts validation services, omitted for reasons of brevity.

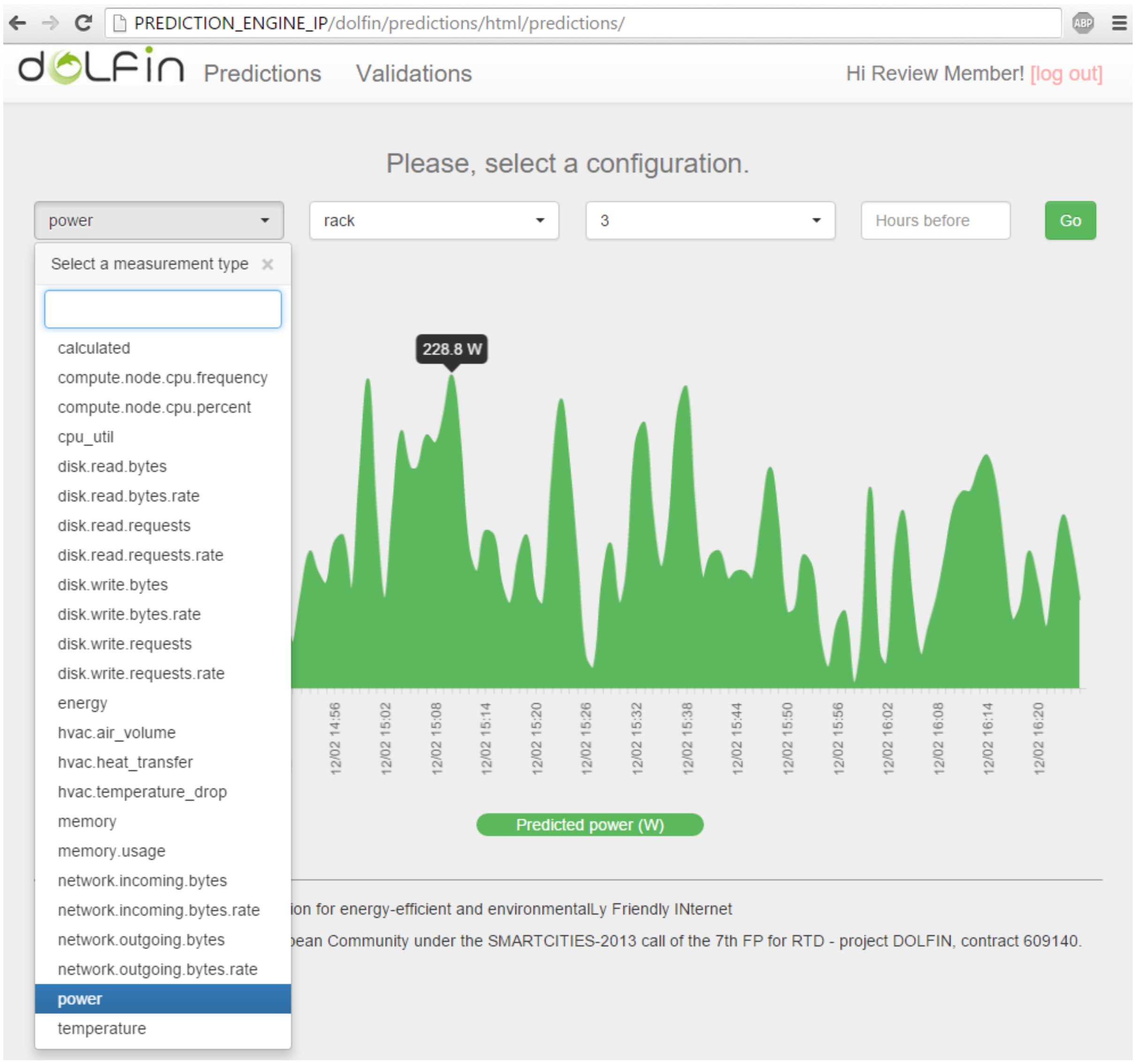


Figure 3-11 - Determination of a particular prediction configuration

3.3.2. Module Dependencies

As the Prediction Engine bases its operation on data retrieved through the eCOP Monitoring DB, the latter constitutes the only functional dependency for its proper operation.

3.3.3. Deployment and installation

Next, the deployment and configuration steps of the Prediction Engine are detailed, assuming that it is deployed on top of a Debian-based system. In particular, the following software dependencies steps have been tested and validated using an Ubuntu 14.04.3 LTS OS. For installation in other OS', the installation steps might be different.

First, one should install MySQL server, in order to enable the authentication and authorization services of the component:

```
# Update the system software
$ sudo apt-get update
$ sudo apt-get upgrade
# Actually install MySQL
$ sudo apt-get install mysql-server
```

After having installed MySQL server, one needs to secure the DB installation:

```
$ sudo mysql_secure_installation
```

Next, the DB schema creation should take place by issuing the following commands:

```
$ mysql -uroot -p
  Enter password:
mysql> create schema predictions;
mysql> exit;
```

Next, the core software dependencies and the actual code of the Prediction Engine should be installed as follows:

```
# Update the system software
$ sudo apt-get update
$ sudo apt-get upgrade
# Install necessary software dependencies
$ sudo apt-get install python-pip python-dev build-essential python-numpy
python-scipy python-matplotlib python-pandas python-sympy python-nose
python-django
$ sudo pip install arrow django-extensions PyYAML django-cors-headers
django-rest-swagger djangorestframework djangorestframework-xml
djangorestframework-yaml joblib nose requests
# Get the eCOP Prediction Engine code
$ git clone
http://artemis_voulkidis@stash.i2cat.net/scm/dol/ecop_prediction_engine.git
# Move the code to /opt
$ sudo mv ecop_prediction_engine/ /opt/ecop_prediction_engine
```

After the successful installation of all the software dependencies, the Prediction Engine should be configured in order to connect to the eCOP Monitoring DB. All configuration options are located in the file `/opt/ecop_prediction_engine/predictions/settings.py`. The list of options that should be configured in order to get a valid instance of the Prediction Engine is documented in the following paragraphs.

3.3.3.1. *Configuration of the DB connection*

```
DATABASES = {
        'default': {
                'ENGINE': 'django.db.backends.mysql',
                'NAME': 'predictions',
                'USER': 'root',
                'PASSWORD': '<password>',
                'HOST': 'localhost',
                'PORT': '3306'
        }
}
```

The `DATABASES` section of the Broker settings should be configured to use the DB installation described in paragraph 3.3.3. If a username other than 'root' is used[4], this username should be placed instead of 'root'. Evidently, the password of the DB user should be entered instead of '<password>'. If the Prediction Engine and the MySQL server are not running in the same server, then the HOST and PORT options should be appropriately configured to match the configuration of the DB.

3.3.3.2. *Configuration of the login redirect URL*

```
LOGIN_REDIRECT_URL = '/dolfin/predictions/html/login/'
```

The `LOGIN_REDIRECT_URL` setting should be changed if the Prediction Engine is running under a BASE URL path other than `http://<PREDICTION_ENGINE_HOSTNAME_OR_IP>/dolfin/predictions/`. When using the default configuration presented in this document, this setting should not be changed.

Indicatively, if the Prediction Engine is running under the base URL path `http://<PREDICTION_ENGINE_HOSTNAME_OR_IP>/`, then the `LOGIN_REDIRECT_URL` setting should be set to `'/html/login/'`.

3.3.3.3. *Configuration of the static files URL*

```
STATIC_URL = '/dolfin/predictions/static/'
```

The `STATIC_URL` setting should be changed if the Prediction Engine is running under a BASE URL path other than `http://<PREDICTION_ENGINE_HOSTNAME_OR_IP>/dolfin/predictions/`. When using the default configuration presented in this document, this setting should not be changed.

[4] This is actually also the recommended way for accessing the DB contents.

Indicatively, if the Prediction Engine is running under the base URL path `http://<PREDICTION_ENGINE_HOSTNAME_OR_IP>/`, then the `STATIC_URL` setting should be set to `'/static/'`.

3.3.3.4. *Configuration of the SWAGGER online documentation module*

```
'api_path': '/dolfin/predictions/',
'base_path': '/<PREDICTION_ENGINE_HOSTNAME_OR_IP>/dolfin/predictions/docs/',
```

These two settings should be changed if the Broker is running under a BASE URL path other than `http://<PREDICTION_ENGINE_HOSTNAME_OR_IP>/dolfin/predictions/`. When using the default configuration presented in this document, the two settings should not be changed.

Indicatively, if the Broker is running under the base URL path `http://<PREDICTION_ENGINE_HOSTNAME_OR_IP>/`, then the two settings should be configured as follows:

```
'api_path': '/',
'base_path': '<PREDICTION_ENGINE_HOSTNAME_OR_IP>/docs/',
```

3.3.3.5. *Configure CORS headers*

```
CORS_ORIGIN_ALLOW_ALL = True
CORS_ORIGIN_REGEX_WHITELIST = ('^(http?://)?192\.168\.1\.\d+\w*$',)
```

If one needs to configure the Prediction Engine to allow CORS, then the relative CORS headers should be included in the responses of the Prediction Engine. In order to configure the respective Broker behaviour, one should appropriately configure the above two settings[5].

3.3.3.6. *Integration with Apache server and configuration [optional, recommended]*

In order to deploy the Prediction Engine with Apache and mod_wsgi, the following commands should be issued:

```
# Update the system software
sudo apt-get update
sudo apt-get upgrade
# Actually install Apache and mod_wsgi
sudo apt-get install apache2 libapache2-mod-wsgi
```

[5] For more information on how to configure CORS in Django, the interested reader is request to refer to https://github.com/ottoyiu/django-cors-headers.

After having installed the necessary software, the file /etc/apache2/sites-enabled/predictions.conf should be created and, then, edited in order to let Apache know of the Prediction Engine installation. The contents of the file should be as follows.

```
WSGIPassAuthorization On
WSGIScriptAlias /dolfin/predictions
/opt/ecop_prediction_engine/predictions/wsgi.py

Alias /dolfin/predictions/static/ /opt/ecop_prediction_engine/static/

<Directory /opt/ecop_prediction_engine/static>
        Require all granted
</Directory>

<Directory /opt/ecop_prediction_engine/predictions>
        <Files wsgi.py>
                Require all granted
        </Files>
</Directory>
```

Evidently, if the Prediction Engine should be configured to run in a base URL path, the WSGIScriptAlias and the static Alias settings should change accordingly[6].

3.3.3.7. *Model Synchronization with the DB*

After the successful configuration of the Prediction Engine software, the following commands should be issued in order to synchronize the Prediction Engine authorization and authentication Models with the DB.

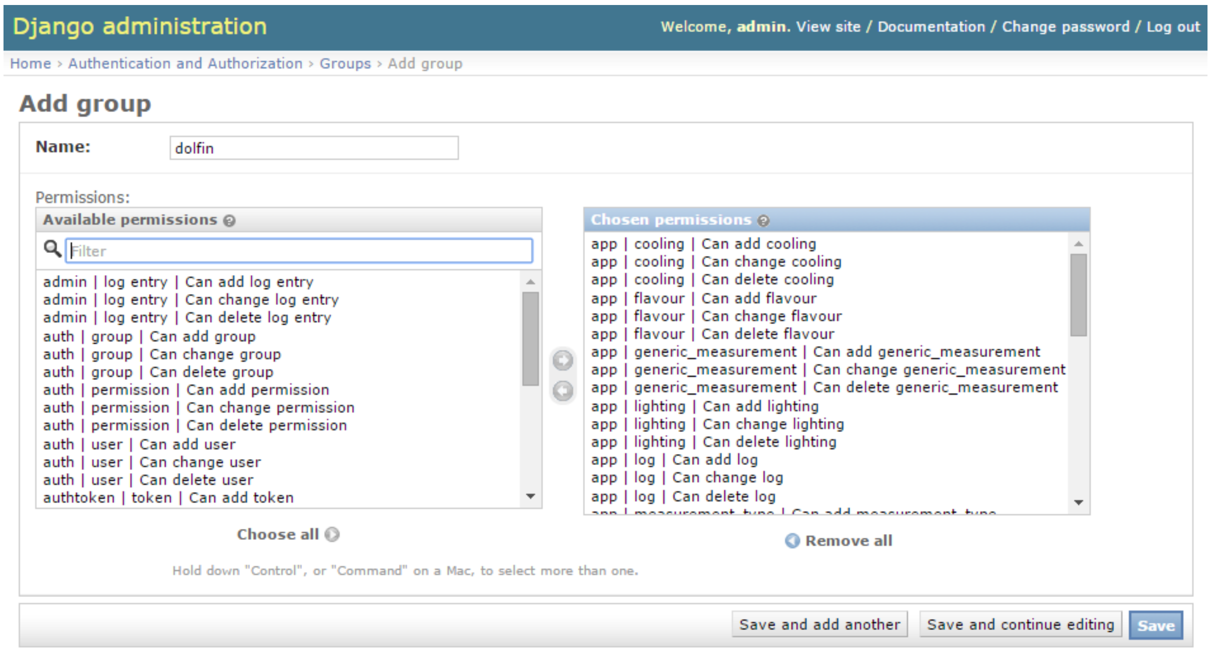
```
#  Synchronize  the  DB  with  the  Prediction  Engine  models  for  the
authentication and authorization services
# Remember to create a superuser
$ sudo python manage.py syncdb
$ sudo python manage.py makemigrations
$ sudo python manage.py migrate
# [Optional] Restart the http service if configured to run with Apache
$ sudo service apache2 restart
```

The Prediction Engine should now be ready for use.

[6] For information on how to configure Apache and mod_wsgi to work with Django, the interested reader is requested to refer to https://docs.djangoproject.com/en/1.9/howto/deployment/wsgi/modwsgi/.

3.3.3.8. *Adding users and groups for using the Prediction Engine*

With regard to the configuration part, it is assumed that a superuser has been created during the synchronization of the Prediction Engine authentication and authorization models with the DB (see previous paragraph). Then, one should visit from a browser the Prediction Engine administration page at `http://<BROKER_HOSTNAME_OR_IP>/dolfin/predictions/admin/`, click the "Groups" link and create a new group with the name 'dolfin' and choose all permissions starting with "app", as depicted in Figure 3-12.


Django administration
Welcome, admin. View site / Documentation / Change password / Log out
Home › Authentication and Authorization › Groups › Add group
Add group
Name:
dolfin
Permissions:
Available permissions
Filter
Chosen permissions
Choose all
Remove all
Hold down "Control", or "Command" on a Mac, to select more than one.
Save and add another
Save and continue editing
Save

Available permissions:

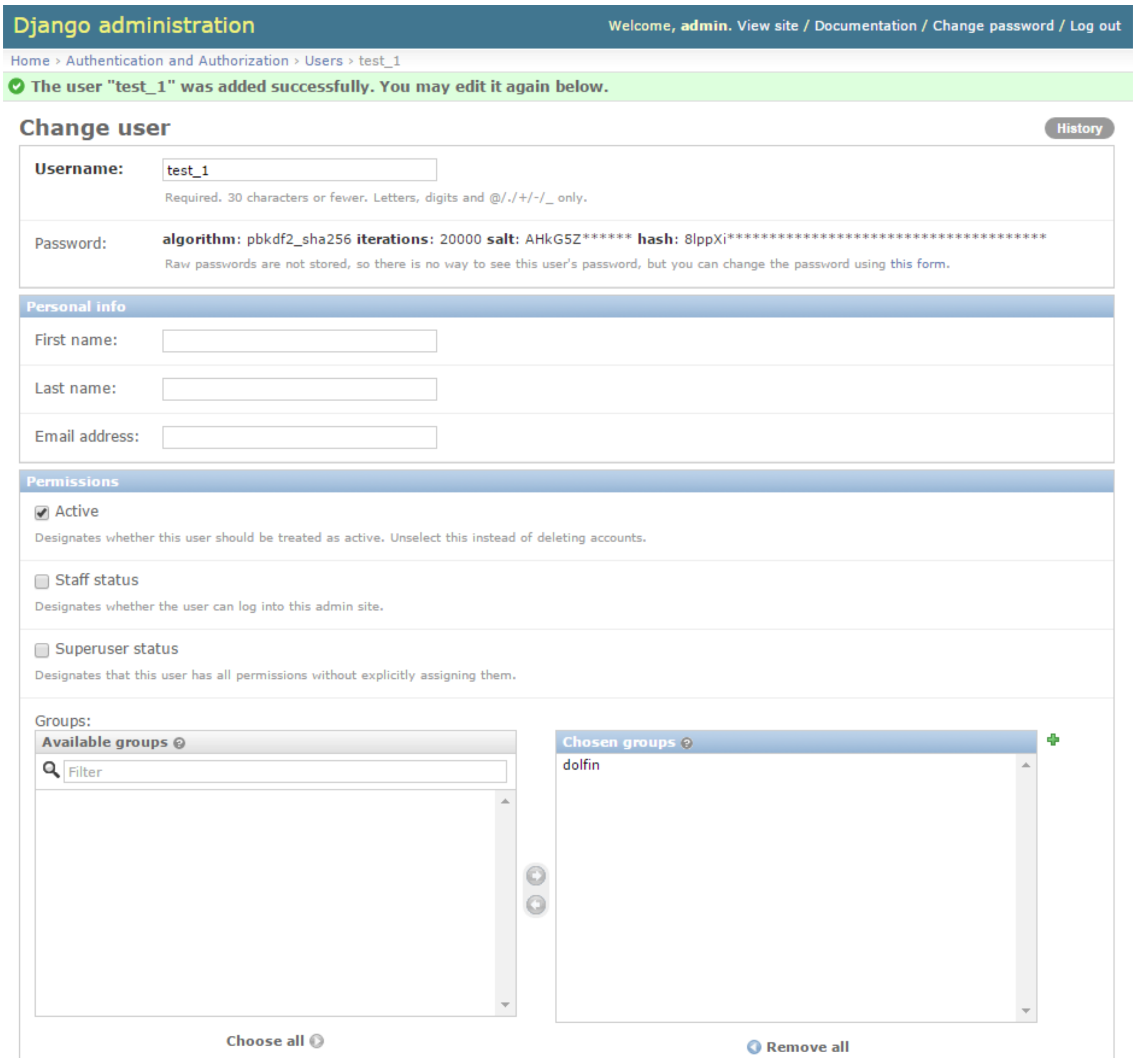
```
admin | log entry | Can add log entry
admin | log entry | Can change log entry
admin | log entry | Can delete log entry
auth | group | Can add group
auth | group | Can change group
auth | group | Can delete group
auth | permission | Can add permission
auth | permission | Can change permission
auth | permission | Can delete permission
auth | user | Can add user
auth | user | Can change user
auth | user | Can delete user
authtoken | token | Can add token
```

Chosen permissions:

```
app | cooling | Can add cooling
app | cooling | Can change cooling
app | cooling | Can delete cooling
app | flavour | Can add flavour
app | flavour | Can change flavour
app | flavour | Can delete flavour
app | generic_measurement | Can add generic_measurement
app | generic_measurement | Can change generic_measurement
app | generic_measurement | Can delete generic_measurement
app | lighting | Can add lighting
app | lighting | Can change lighting
app | lighting | Can delete lighting
app | log | Can add log
app | log | Can change log
app | log | Can delete log
app | measurement_type | Can add measurement_type
```

Figure 3-12 - Dolfin group creation through the administration page

Any user willing to use the Prediction Engine should belong to this group. To create a new Prediction Engine user, navigate to the administration page, click on 'Users' and then 'Add User'. Upon creation of the user, edit the user to add her into the 'dolfin' group as depicted in Figure 3-13.


Django administration

Welcome, **admin**. View site / Documentation / Change password / Log out

Home › Authentication and Authorization › Users › test_1

The user "test_1" was added successfully. You may edit it again below.

Change user

Username: test_1

Required. 30 characters or fewer. Letters, digits and @/./+/-/_ only.

Password: **algorithm**: pbkdf2_sha256 **iterations**: 20000 **salt**: AHkG5Z****** **hash**: 8lppXi**************************************

Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

Personal info

First name:

Last name:

Email address:

Permissions

[✓] Active

Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

[ ] Staff status

Designates whether the user can log into this admin site.

[ ] Superuser status

Designates that this user has all permissions without explicitly assigning them.

Groups:

Available groups | Chosen groups: dolfin

Figure 3-13 - Prediction Engine user creation through the administration page

3.3.3.9. *Configuring the default Predictions interval and duration*

```
DEFAULT_INTERVAL_IN_MIN = 1
DEFAULT_DURATION_IN_HOURS = 1
```

By changing these variables, one can specify the granularity and default timeframe of the predictions generated by the Prediction Engine, unless otherwise specified by using the appropriate URL path parameters when invoking the relevant API services. By default, the Prediction Engine produces predictions with a duration of 1 hour, predicting the respective values for a time interval of 1 minute (thus resulting in the generation of 60 predicted values).

3.3.3.10. *Configuring the connection to the eCOP DB*

```
BASE_ECOP_DB_API_URL = 'http://<ecop_base_url>/dolfin/db/api/'
```

The configuration of the eCOP DB URL is possible through properly adjusting the `BASE_ECOP_DB_API_URL` setting.

3.3.3.11. *Configuring the automated model updating process*

In order to configure the automated model updating process, one needs to edit the relevant file located in `/opt/ecop_prediction_engine/model_updater.py`. The relevant configuration options are the following:

```
TRAINING_DATA_START_DATE = before(24*2*7)
TRAINING_DATA_END_DATE   = now()
ORACLE_DIR = '/opt/predictions/api/oracle/'
token = '_a_token_'
```

First, the default duration of the time period to examine updating the models should be configured, by properly setting the `TRAINING_DATA_START_DATE` and `TRAINING_DATA_END_DATE` settings. The `TRAINING_DATA_START_DATE` directive should be set on an hours-base; by default, a time period of 14 days is considered. If one wants to have another directory for storing the models, the `ORACLE_DIR` setting should be appropriately set. Last, the token setting should be set to match a unique token provided by the eCOP Monitoring DB authentication service (see Deliverable D3.3).

3.3.3.12. *Configuring the Interaction with the Policy Maker*

In order to configure the interaction of the Prediction Engine with the Policy Maker, one needs to edit the relevant file located in `/opt/ecop_prediction_engine/policy_maker_updater.py`. The configuration options are the following:

```
BASE_ECOP_DB_API_URL= 'http://<ecop_base_url>/dolfin/db/api/'
BASE_PREDICTION_ENGINE_URL='http://localhost/dolfin/predictions/api/'
BASE_POLICY_MAKER_URL='http://<policy_maker_url>/v1/policy/steer/kpi_pue'
MAXIMUM_ACCEPTABLE_DEVIATION = 0.1
```

First, the eCOP Monitoring DB URL should be determined, by setting the `BASE_ECOP_DB_API_URL` directive. The same holds for setting the respective URLs for the Prediction Engine and the Policy Maker, through the settings `BASE_PREDICTION_ENGINE_URL` and `BASE_POLICY_MAKER_URL`. Last, the maximum acceptable deviation between the predictions of the Prediction Engine and the actual measurements during the last hour can be set through the MAXIMUM_ACCEPTABLE_DEVIATION setting; when a relative deviation larger than MAXIMUM_ACCEPTABLE_DEVIATION is detected under the automated, asynchronous Prediction Engine mode, the Policy Maker is notified of this event.

3.3.4. Testing the installation

There are three ways of testing the Prediction Engine operation:

1. Via command line,
2. Through the Prediction Engine online documentation system service
3. Through the Prediction Engine Dashboard

3.3.4.1. *Testing via command line*

For testing the Prediction Engine via command line, the following commands can be issued:

```
# Install curl command
$ sudo apt install curl
$ curl -X POST -H "Content-Type: application/json" -d '{"username": "test","password":
"test"}' http://<PREDICTION_ENGINE_HOSTNAME>/dolfin/predictions/api/tokens/
# Here comes the response with the token
{"token":"581e76b175a6b48c78bb39ee598284e7183affdb","created_at":"2016-02-
12T11:16:54.293112+00:00","expires_at":"2016-02-13T11:16:54.293112+00:00"}
# Request a prediction of the power consumption of rack 1 of the DC, using
the token already acquired
$ curl -H "Authorization: Token 581e76b175a6b48c78bb39ee598284e7183affdb"
http://<PREDICTION_ENGINE_HOSTNAME>/dolfin/predictions/api/predict/power/rack/1/
# No predictions have been made yet in a vanilla installation
{
  "start": "2016-02-12T11:18:06.094162Z",
  "end": "2016-02-12T12:18:06.094187Z",
  "type": "power",
  "resource": "rack",
  "resource_id": "3",
  "model_version": "20160212002249",
  "predictions": []
}
```

3.3.4.2. *Testing via the online documentation system*

In order to test the Prediction Engine installation and configuration assuming that the default configuration presented in the previous paragraphs has been followed, one should visit the page `http://<prediction_engine_url_or_ip>:<port>/dolfin/predictions/docs` which constitutes the landing page of the online documentation module supporting the operation of the Prediction Engine operation already depicted in Figure 3-9. From there, one can get an authentication token by invoking the tokens API service and, then, query the Prediction Engine for generating a prediction. Instructions on how to use the online documentation module can be found in the page of

Swagger-UI[7] and in the deliverable D3.3, paragraph 3.3.4.1, where the online documentation system is presented for the eCOP Monitoring DB.

3.3.4.3. *Testing via the Prediction Engine Dashboard*

To check the installation of the dashboard, one is required to log into the Prediction Engine Dashboard login page defined in the main settings file as documented in 3.3.3.2. After successful login, the dashboard should expose a view similar to the one presented in 3.3.1.

3.3.5. Service endpoints and data model

The Prediction Engine has been configured to offer services categorized into two main groups of services together with the authentication service as follows:

1. Predictions
2. Validations
3. Tokens (Authentication and Authorization)

3.3.5.1. *Predictions [/api/predict/]*

The Predictions service provides predictions over defined resources and timeframes. The following endpoints have been defined for this group:

HTTP Method	API Service Endpoint URL	Description
GET	`/api/predict/<type>/<resource>/<id>/`	Predicts the measurements of type `<type>` of a resource of type `<resource>` with id `<id>` over the next hour with interval of 1 minute.
GET	`/api/predict/<type>/<resource>/<id>/<end>/`	Predicts the measurements of type `<type>` of a resource of type `<resource>` with id `<id>` over the `<end>` hours with interval of 1 minute.
GET	`/api/predict/<type>/<resource>/<id>/<start>/<end>/`	Predicts the measurements of type `<type>` of a resource of type `<resource>` with id `<id>` starting from `<start>` and ending in `<end>` with interval of 1 minute.
GET	`/api/predict/<type>/<resource>/<id>/<start>/<end>/<interval>/`	Predicts the measurements of type `<type>` of a resource of type `<resource>` with id `<id>` starting from `<start>` and ending in

[7] Swagger UI webpage, http://swagger.io/swagger-ui/.

		`<end>` with interval of `<interval>` minutes.

Table 3-15 - API services exposed by the Predictions group

The data model of a prediction response is as follows:

Attribute	Type	Description
start	String	The start time of the prediction in ISO8601 format
end	String	The end time of the prediction in ISO8601 format
type	String	The type of the measurement (must be a valid measurement_type name, see D3.3)
resource	String	The resource type (can be a vm, server, rack etc.)
resource_id	String	The id of the resource (can the uuid of a vm, the serial_number of a server, the id of a lighting element etc.)
model_version	String	The version of the models used, namely when was the model used last updated, in a YYYmmDDHHMMSS format.
predictions	List<Prediction>	The list of predicted values

Table 3-16: Data model of a predictions response

The data model of the Prediction class is detailed in the table below:

Attribute	Type	Description
time	String	The time of the predicted value in ISO8601 format
value	Decimal	The predicted value

Table 3-17: Data model of a prediction object

3.3.5.2. *Validations [/api/validate/]*

The Validations service provides validations of predictions over defined resources and timeframes. The following endpoints have been defined for this group:

HTTP Method	API Service Endpoint URL	Description
GET	`/api/validate/<type>/<resource>/<id>/<start>/`	Validates the prediction of the measurements of type `<type>` of a resource of type `<resource>` with id `<id>` over the `<start>` last hours with interval of 1 minute.
GET	`/api/validate/<type>/<resource>/<id>/<start>/<end>/`	Predicts the measurements of type `<type>` of a resource of type `<resource>` with id

	`<id>` starting from `<start>` and ending in `<end>` with interval of 1 minute.

The data model of a validation response is as follows:

Attribute	Type	Description
start	String	The start time of the prediction validation in ISO8601 format
end	String	The end time of the prediction validation in ISO8601 format
type	String	The type of the measurement (must be a valid measurement_type name, see D3.3)
resource	String	The resource type (can be a vm, server, rack etc.)
resource_id	String	The id of the resource (can the uuid of a vm, the serial_number of a server, the id of a lighting element etc.)
r_squared	Decimal	The R^2 value of the validated predictions
mse	Decimal	The Mean Square Error between the predictions and the actual measurements
rmse	Decimal	The Relative Mean Square Error between the predictions and the actual measurements
rsd	Decimal	The Relative Standard Deviation between the predictions and the actual measurements
model_version	String	The version of the models used, namely when was the model used last updated, in a YYYmmDDHHMMSS format.
data	List<Validation>	The list of validated predictions.

The data model of the Validation class is detailed in the table below:

Attribute	Type	Description
time	String	The time of the referenced validation in ISO8601 format
actual	Decimal	The actually measured value
predicted	Decimal	The predicted value

3.3.5.3. *Authentication and Authorization [/api/tokens/]*

The tokens endpoint is used for users to acquire authorization tokens to use for accessing the Prediction Engine. The following API service endpoint has been configured for this endpoint:

HTTP Method	API Service Endpoint URL	Description
POST	`/api/tokens/`	Get a new authorization token

The data model of entity to be posted in order to acquire a new token is structured as follows:

Attribute	Type	Description
username	String	The username of the user
password	String	The password of the user

The data model of a token object retrieved by the Prediction Engine is as follows:

Attribute	Type	Description
token	String	The authorization token
created_at	String	The date when the token was generated in ISO8601 format
expires_at	String	The date when the token seizes to be valid in ISO8601 format

The token should be used in every HTTP request performed against the Prediction Engine API service endpoints as a header in the form:

```
Authorization: Token <token>
```

where <token> is the acquired token.

3.4. Optimizer

As its name suggest, the Optimizer is the DOLFIN eCOP component responsible for optimizing the intra-DC state, based on the policy that dictates, each time, the proper DC operation. Based on the active policy, the Optimizer is able to devise action plans either to minimize the energy consumption of the DC, or increase its cost-efficiency. The devised action plans may refer to IT and non-IT infrastructures management and include VM management (e.g. migration or relocation), Server management (hibernate inactive servers when no significant demand for computing services is predicted by the Prediction Engine), Lighting management (e.g. turn off lights in inactive DC rooms) and HVAC management.

3.4.1. Basic concepts

Tightly coupled with and implementing the energy optimization part of the intra-DC optimization processes of a DOLFIN-enabled DC, the Optimizer component aims at providing optimal plans for re-allocating the DC resources (at both IT and non-IT levels) to reach the objectives (currently active policy) set by the Policy Maker.

The Optimizer is a mostly passive component, waiting for optimization requests from the Policy Maker. Upon receipt of a relevant request, the Optimizer acknowledges receipt and updates its status in order to let the interested components know that an optimization plan is currently on the works. Next, based on the currently active policy governing proper DC operation, the Optimizer asynchronously serves the optimization request and produces a relevant optimization plan. The latter is forwarded to the Policy Actuator in order to be executed. Simultaneously, the Optimizer changes its status to Idle and notifies the Policy Maker that the particular optimization request has been served and a relevant optimization plan has been sent to the Policy Actuator.

When requested to absolutely minimize the energy consumption of the DC without any restriction, the Optimizer tries to find the optimal VM allocation to the physical servers of the DC so that the number of idle server that can be finally switched off is maximized, simultaneously maximizing the possible energy consumption merits acquired by the avoidance of their operation. The problem has been modelled as a one dimension (1D) Bin-Packing one, considering RAM of the VMs and the physical servers as the sizable dimension and the physical servers as bins.

As time is critical for proper and timely DC adaptation and it is not possible to acquire an analytical solution to such problems in reasonable (polynomial) time (the generalized Bin-Packing problem is known to be NP-Hard), we have developed two heuristics that are able to achieve near-optimal allocations, without being too time consuming. Particularly, when the DC is under high load and an optimization request for minimizing the energy consumption is received, the Optimizer performs a simple Best Fit Decreasing (BFD) algorithm that is able to arrange the VMs in the operating servers in a near-optimal way. Particularly, in the direction of employing the BFD the DC servers are indexed based on their energy-efficiency, with energy-efficient servers being assigned a lower index. Subsequently, the VMs are placed into physical servers in order of increasing index. As a result, energy-efficient servers are assigned a higher priority and for instance servers of a Green Room are reserved first, or servers of the same DC segment are reserved prior to remote DC servers in order to allow remote DC servers to hibernate, providing substantial energy savings. Next, the VMs are sorted by (RAM) size and are then placed in order of increasing index, first into the occupied physical servers of lower available capacity and then, in case they do not fit into the occupied servers, or in case of a tie, VMs are placed in order of increasing index into the lower indexed physical server they fit.

In the case a more drastic DC load re-organization is required (i.e. in the case an extreme DC energy minimization is needed), a complementary solution employing Grouping Genetic Algorithms (GGAs) has been implemented. In the context of the Optimizer, GGAs are seen as simple GAs where each gene of a GAs' chromosome corresponds to a tuple of elements corresponding to the VMs of each physical server and the latter are the building blocks evolved by the employment of the GAs. This approach alters all GA operators significantly; however, our approach outperforms the standalone GA substantially when applied to grouping problems.

The use of GGA, initialized by BFD, for the optimal VM allocation allows for the consolidated allocation of VMs at an intra-DC level as well as an inter-DC level, whenever a VM consolidation is imposed by the Smart Grid operation. Thus, the distributed application of the above optimization algorithm on DOLFIN DCs, when that is deemed necessary based on the SVR load predictions, could yield significant energy savings as well as reliable Smart Grid operation.

In case an optimization request is performed when a non-trivial optimization policy is active (e.g. optimize the DC performance in terms of cost, given some arbitrary constraints), the optimizer attempts to map the active policy and the accompanying data to a linear convex optimization problem, formulating the objective function as a minimization of either the energy cost or the absolute energy consumption and appropriately handling the restrictions based on the policy ones, to conclude on a simple, linear, convex optimization problem as in [6] and [7] . The problem is then solved by using well known, open convex optimization libraries [8] . The VM Priority Classifier and the abstractions it performs (to be briefly discussed in the following paragraphs) assists the optimization process by abstracting part of the restrictions, also limiting the set of resources upon which the optimization process will be applied; in this sense, the VM Priority Classifier acts as a catalyst, accelerating the optimization process.

The generic, high level architecture of the Optimizer is presented in Figure 3-14.

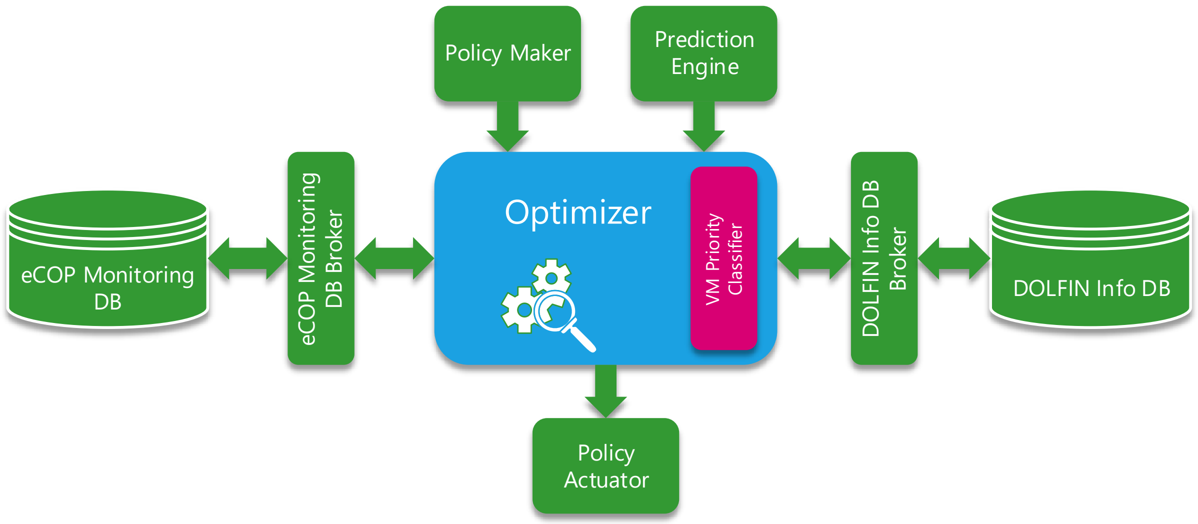


Figure 3-14 – The Optimizer interfacing and interconnection with other components

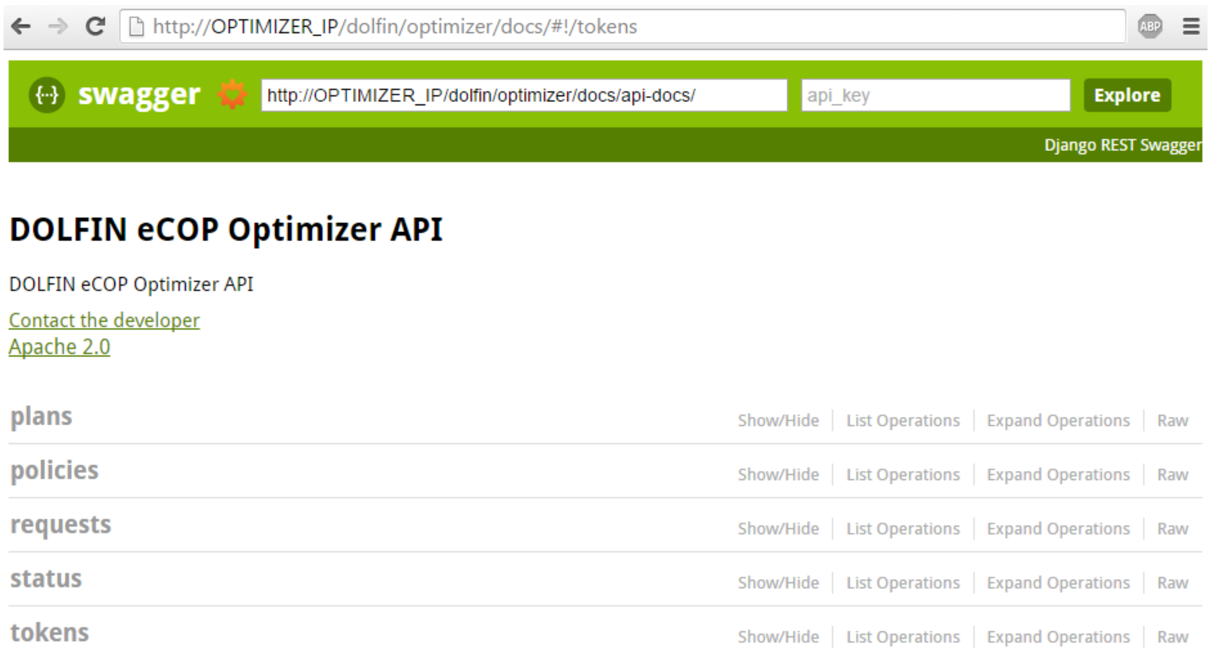
Evidently, in order to achieve its optimization cause, the Optimizer needs access to certain sets of data originating from other components and stored in the eCOP Monitoring DB and the DOLFIN Info DB. Upon receipt of an optimization request and on the basis of the active policy governing the optimization process, the Optimizer identifies the datasets that are required in order to devise a new optimization plan. These datasets reside in the eCOP Monitoring DB and are acquired via querying the eCOP Monitoring DB Broker via the VM Priority Classifier. Further, the needed datasets may pertain to monitoring (raw or calculated), or to ICT and non-ICT assets of the DC in hand (e.g. VMs, Servers, HVAC equipment etc.). Apart from these data, the VM Priority Classifier queries the DOLFIN Info DB in order to get information related to the SLAs that relate to VMs and Servers that will participate in the optimization process[8]. Upon receipt of all this information, the VM Priority Classifier fuses and aggregates it, limiting its dimensionality and transforming the heterogeneous types of data into homogeneous, so that the optimization problem can be properly formulated as a linear convex function.

It should be underlined that as the optimization processes can be time consuming, particularly in the case of large DCs, the Optimizer software has been configured to perform the optimization processes in a multi-threaded manner. This grants the Optimizer the ability to either simultaneously handle multiple optimization requests, e.g. each one targeting at different sets of VMs, or complete an optimization request faster compared to a single-threaded case (by default, a -configurable- number of 4 threads have been considered for use by the Optimizer at any time).

As in the case of the eCOP Monitor DB and the Prediction Engine, the Optimizer functionality is exposed via a RESTful web service exposing 10 application-oriented endpoints, supported by 26 administration-oriented ones. Again, all APIs are secured against unauthorized access through the employment of time-expiring authentication and authorization tokens. The following Figure presents the application-oriented group of API endpoints exposed by the RESTful interface of the Optimizer

[8] Depending on the various constraints set by the active Optimization Policy, the Optimizer may exclude certain ICT infrastructure from the optimization process, or only include a small subset of it.

with the help of a relevant self-documentation web service. More details on the full range of service API endpoints exposed by the Optimizer may be found at paragraph 3.4.5.



http://OPTIMIZER_IP/dolfin/optimizer/docs/#!/tokens

swagger http://OPTIMIZER_IP/dolfin/optimizer/docs/api-docs/ api_key Explore

Django REST Swagger

DOLFIN eCOP Optimizer API

DOLFIN eCOP Optimizer API

Contact the developer
Apache 2.0

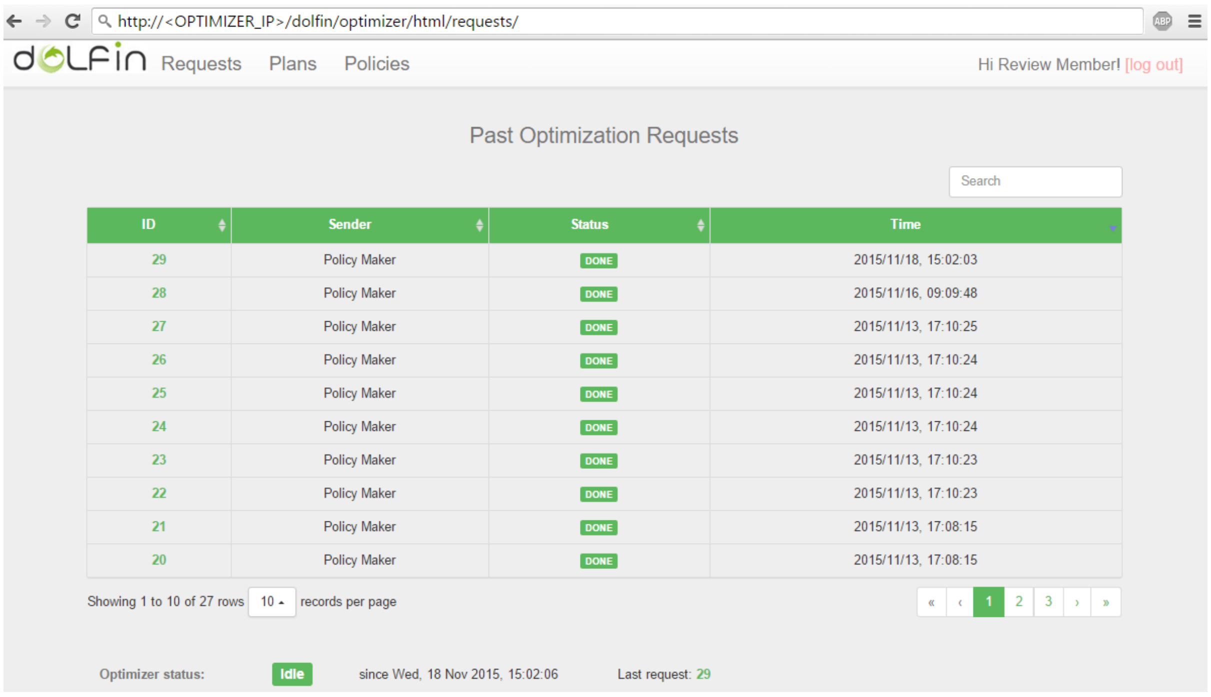
plans	Show/Hide	List Operations	Expand Operations	Raw
policies	Show/Hide	List Operations	Expand Operations	Raw
requests	Show/Hide	List Operations	Expand Operations	Raw
status	Show/Hide	List Operations	Expand Operations	Raw
tokens	Show/Hide	List Operations	Expand Operations	Raw

Figure 3-15 - The groups of service API endpoint exposed by the Optimizer

Details on how to use the online documentation system are omitted as they are documented at [9] and at D3.3 and in the current document in paragraph 3.3.1.

Apart from the online documentation system and similar to the eCOP Monitoring DB and the Prediction Engine, the Optimizer exposes a dashboard interface that allows DC owners and operators to monitor the various optimization processes, the plans generated, set the policies that govern efficient DC operation etc. The following figure depicts the web page where one can check the optimization requests that have been received by the Optimizer.

[9] http://swagger.io/swagger-ui/


http://<OPTIMIZER_IP>/dolfin/optimizer/html/requests/

Past Optimization Requests

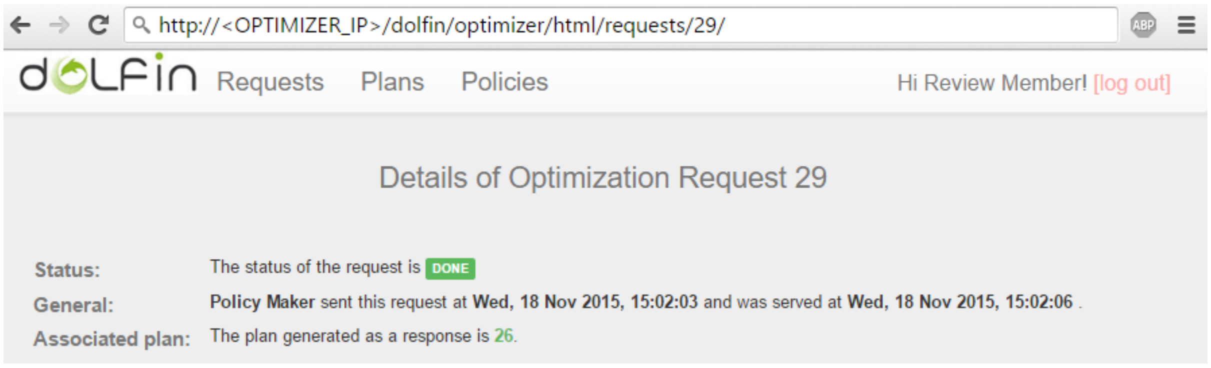
ID	Sender	Status	Time
29	Policy Maker	DONE	2015/11/18, 15:02:03
28	Policy Maker	DONE	2015/11/16, 09:09:48
27	Policy Maker	DONE	2015/11/13, 17:10:25
26	Policy Maker	DONE	2015/11/13, 17:10:24
25	Policy Maker	DONE	2015/11/13, 17:10:24
24	Policy Maker	DONE	2015/11/13, 17:10:24
23	Policy Maker	DONE	2015/11/13, 17:10:23
22	Policy Maker	DONE	2015/11/13, 17:10:23
21	Policy Maker	DONE	2015/11/13, 17:08:15
20	Policy Maker	DONE	2015/11/13, 17:08:15

Showing 1 to 10 of 27 rows 10 records per page

Optimizer status: Idle since Wed, 18 Nov 2015, 15:02:06 Last request: 29

Figure 3-16 - Overview of the Requests tab of the Optimizer Dashboard

As can be readily seen in Figure 3-16, through this page, one can overview the past optimization requests, grouped by the sender of the Request the optimization and request status. Moreover, at the end of the page, one is able to check the status of the Optimizer (namely whether it is working on an optimization plan or not) and what was the id of the last request served. Clicking on the id of a request will bring the user to a new page where details of the request are presented, including the time when the request was received, the time that it was served (i.e. the time when a relevant optimization plan was generated) and the id of the associated plan generated, as presented in the figure below with a reference optimization request of id 29. Again, when clicking on the generated plan id (in this case 26), the user is redirected to the details page for this particular optimization plan, presented later.

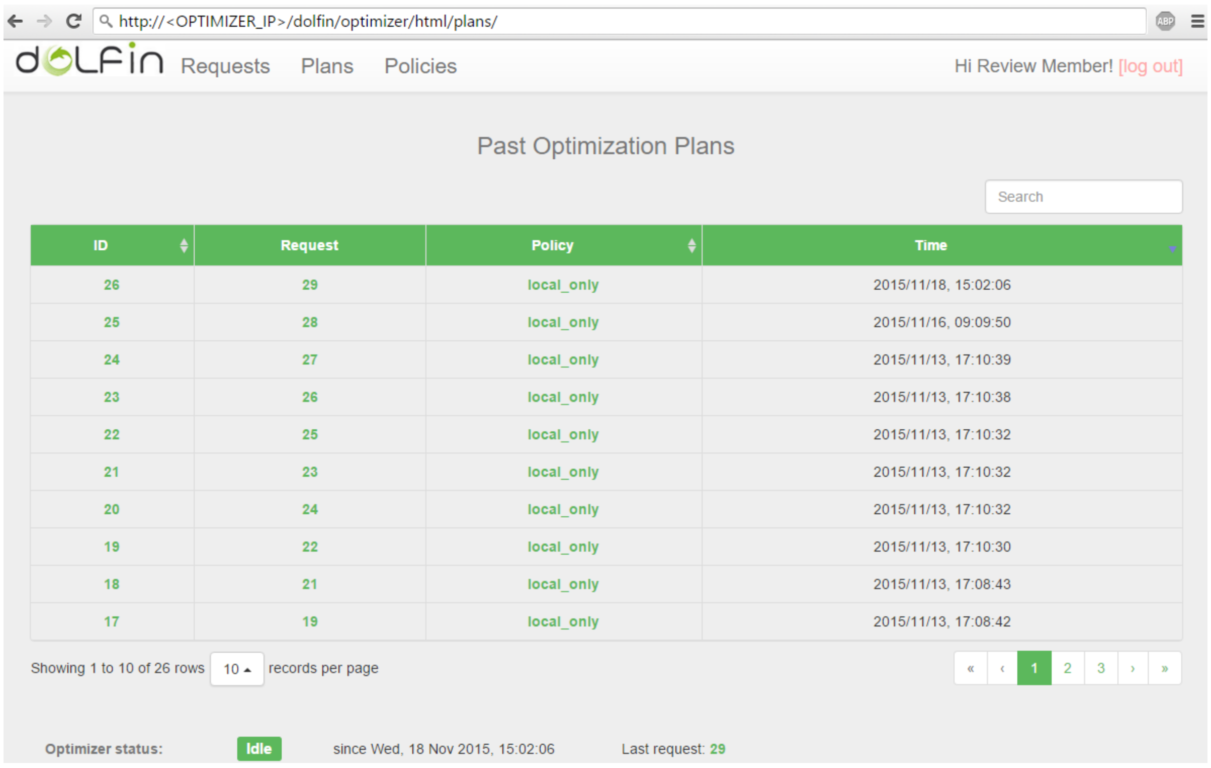

http://<OPTIMIZER_IP>/dolfin/optimizer/html/requests/29/

Details of Optimization Request 29

Status:	The status of the request is DONE
General:	**Policy Maker** sent this request at **Wed, 18 Nov 2015, 15:02:03** and was served at **Wed, 18 Nov 2015, 15:02:06** .
Associated plan:	The plan generated as a response is 26.

Figure 3-17 - Overview of the optimization request details

If the user clicks on the Plans tab, an overview of the generated plans is presented as depicted in Figure 3-18.


http://<OPTIMIZER_IP>/dolfin/optimizer/html/plans/
Requests
Plans
Policies
Hi Review Member! [log out]
Search

Past Optimization Plans

ID	Request	Policy	Time
26	29	local_only	2015/11/18, 15:02:06
25	28	local_only	2015/11/16, 09:09:50
24	27	local_only	2015/11/13, 17:10:39
23	26	local_only	2015/11/13, 17:10:38
22	25	local_only	2015/11/13, 17:10:32
21	23	local_only	2015/11/13, 17:10:32
20	24	local_only	2015/11/13, 17:10:32
19	22	local_only	2015/11/13, 17:10:30
18	21	local_only	2015/11/13, 17:08:43
17	19	local_only	2015/11/13, 17:08:42

Showing 1 to 10 of 26 rows 10 records per page

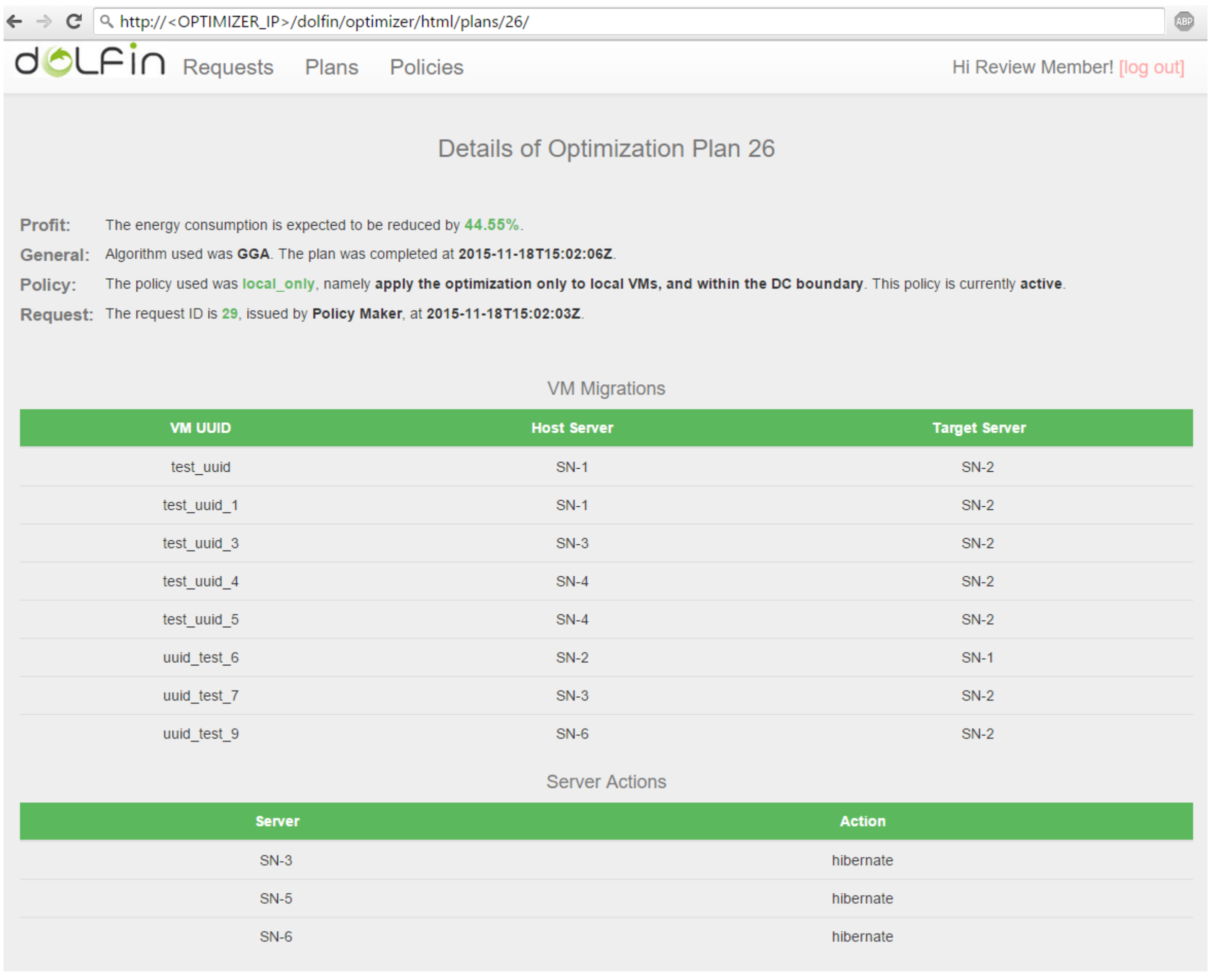
« ‹ 1 2 3 › »

Optimizer status: Idle since Wed, 18 Nov 2015, 15:02:06 Last request: 29

Figure 3-18 - Overview of the Plans tab of the Optimizer dashboard

Through this tab, one can check the IDs of the generated plans together with the respective requests received, the policies that were used for each request and the plan generation time. If one clicks on the id of a specific plan, he gets redirected to the plan details page, depicted in Figure 3-19. In this view, the DC operators are able to check

1. the profit expected from the application of the specific optimization plan,
2. the algorithm that was used in order to acquire the plan,
3. the policy that was governing the DC operation at the time of the generation of the plan
4. the request that was issued that led to the generation of the particular optimization plan, and
5. The list of actions that are needed in order to implement the optimization plan. These actions depend on the equipment registered in the eCOP Monitoring DB such as VMs, Servers, Lighting systems and HVAC.


http://<OPTIMIZER_IP>/dolfin/optimizer/html/plans/26/
Requests
Plans
Policies
Hi Review Member! [log out]

Details of Optimization Plan 26

Profit: The energy consumption is expected to be reduced by **44.55%**.
General: Algorithm used was **GGA**. The plan was completed at **2015-11-18T15:02:06Z**.
Policy: The policy used was **local_only**, namely **apply the optimization only to local VMs, and within the DC boundary**. This policy is currently **active**.
Request: The request ID is **29**, issued by **Policy Maker**, at **2015-11-18T15:02:03Z**.

VM Migrations

VM UUID	Host Server	Target Server
test_uuid	SN-1	SN-2
test_uuid_1	SN-1	SN-2
test_uuid_3	SN-3	SN-2
test_uuid_4	SN-4	SN-2
test_uuid_5	SN-4	SN-2
uuid_test_6	SN-2	SN-1
uuid_test_7	SN-3	SN-2
uuid_test_9	SN-6	SN-2

Server Actions

Server	Action
SN-3	hibernate
SN-5	hibernate
SN-6	hibernate

Figure 3-19 - Details of an optimization plan

3.4.2. Module Dependencies

In the framework of getting the necessary data for devising an optimization plan, the Optimizer needs to have access to a running instance of the eCOP Monitoring DB and the associated Broker. In a similar context, the Optimizer needs to have access to predictions from the Prediction Engine. Moreover, in order to get the SLA status of the various users and VMs, the Optimizer depends on a running instance of the SLA Renegotiation Controller. Last, in order to materialize the devised optimization plans, the Optimizer needs to have access to a running installation of the Policy Actuator.

As an optional (though practically hard) dependency, the Optimizer needs to have access to a Policy Maker instance in order to acquire DC Management Policies and optimization requests.

3.4.3. Deployment and installation

Next, the deployment and configuration steps of the Optimizer are detailed, assuming that it is deployed on top of a Debian-based system. In particular, the following software dependencies steps have been tested and validated using an Ubuntu 14.04.3 LTS OS. For installation in other OS', the installation steps might be different.

First, one should install MySQL server, in order to enable the authentication and authorization services of the component:

```
# Update the system software
$ sudo apt-get update
$ sudo apt-get upgrade
# Actually install MySQL
$ sudo apt-get install mysql-server
```

After having installed MySQL server, one needs to secure the DB installation:

```
$ sudo mysql_secure_installation
```

Next, the DB schema creation should take place by issuing the following commands:

```
$ mysql -uroot -p
  Enter password:
mysql> create schema optimizer;
mysql> exit;
```

Next, the core software dependencies and the actual code of the Optimizer should be installed as follows:

```
# Update the system software
$ sudo apt-get update
$ sudo apt-get upgrade
# Install necessary software dependencies
$ sudo apt-get install python-pip python-dev build-essential redis-server
$ sudo pip install -r requirements.txt
# Get the Optimizer code
$ git clone
http://artemis_voulkidis@stash.i2cat.net/scm/dol/ecop_optimizer.git
# Move the code to /opt
$ sudo mv ecop_optimizer/ /opt/ecop_optimizer
```

As a next step, the CELERY workers should be configured to run in a daemon mode, as documented in [10]. After the successful installation of all the software dependencies, the Optimizer should be configured in order to connect to the eCOP Monitoring DB. All configuration options are located in the file `/opt/ecop_optimizer/optimizer/settings.py`.

The basic configuration of the Optimizer follows the initial 8 steps presented for the case of the Prediction Engine (paragraphs 3.3.3.1 - 3.3.3.8), with the necessary paths and filenames adaptations, adjusting the local filesystem paths from `/opt/ecop_prediction_engine/predictions/` to `/opt/ecop_optimizer/optimizer/` and the web URLs from `/api/predictions/` to

[10] http://celery.readthedocs.org/en/latest/tutorials/daemonizing.html#daemonizing

/api/optimizer/. Also, the DB settings should change to point to the optimizer schema, rather than the predictions one.

The list of extra options that should be configured in order to get a valid instance of the Optimizer is documented in the following paragraphs.

3.4.3.1. *Configuring the CELERY workers*

In order to enable asynchronous operation and decouple the Policy Maker requests from the core optimization processes, the Optimizer dispatches its optimization tasks to different CELERY workers. To configure the workers, one should create or edit the file /etc/default/celeryd as follows:

```
ubuntu@dolfin:~$ cat /etc/default/celeryd | grep ^[^#]
ENABLED="true"
CELERYD_NODES="w1 w2"
CELERYD_CHDIR="/opt/ecop_optimizer"
CELERYD_OPTS="--concurrency=2"
CELERY_CONFIG_MODULE="celeryconfig"
CELERYD_LOG_FILE="/var/log/celery/%n.log"
CELERYD_USER="celery"
CELERYD_GROUP="celery"
BROKER_URL="redis://localhost:6379/0"
CELERY_APP="optimizer"
```

This will instruct celery to use the REDIS instance on localhost at port 6379 (default), use two concurrent workers for parallel execution and consider the optimizer base directory (/opt/ecop_optimizer/) as CELERY home directory as well.

3.4.3.2. *Configuring the interaction with the REDIS server and CELERY workers*

Next, the core Optimizer infrastructure should be configured to use the REDIS server to dispatch the optimization execution tasks to the various CELERY instances. This can be done by editing the /opt/ecop_optimizer/optimizer/settings.py file as follows:

```
BROKER_URL = 'redis://localhost:6379'
CELERY_RESULT_BACKEND = 'redis://localhost:6379'
CELERY_ACCEPT_CONTENT = ['application/json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
CELERY_TIMEZONE = 'UTC'
```

Note that the base configuration options again refer to appropriately connecting to REDIS.

3.4.3.3. *Configuring the interaction with the Policy Actuator*

To enable proper communication with the Policy Actuator, the following setting should be configured in the `/opt/ecop_optimizer/optimizer/settings.py` file:

```
POLICY_ACTUATOR_BASE_URL = 'http://84.88.40.69:9091/policyactuator'
```

3.4.3.4. *Configuring the interaction with the Prediction Engine*

To enable proper communication with the Prediction Engine, the following setting should be configured in the `/opt/ecop_optimizer/optimizer/settings.py` file:

```
PREDICTION_ENGINE_BASE_URL =
'http://<PREDICTION_ENGINE_HOSTNAME_OR_IP>/dolfin/predictions/api/
```

3.4.3.5. *Configuring the interaction with the Dolfin Info DB*

To enable proper communication with the Prediction Engine, the following setting should be configured in the `/opt/ecop_optimizer/optimizer/settings.py` file:

```
DOLFIN_INFO_DB_BASE_URL = 'http://<DOLFIN_INFO_DB_API_SERVICE_URL>/
```

3.4.4. Testing the installation

There are three ways of testing the Optimizer operation:

4. Via command line,
5. Through the Optimizer online documentation system service
6. Through the Optimizer Dashboard

3.4.4.1. *Testing via command line*

For testing the Optimizer via command line, the following commands can be issued:

```
# Install curl command
$ sudo apt install curl
$ curl -X POST -H "Content-Type: application/json" -d '{"username": "test","password":
"test"}' http://<OPTIMIZER_HOSTNAME_OR_IP>/dolfin/optimizer/api/tokens/
# Here comes the response with the token
{"token":"581e76b175a6b48c78bb39ee598284e7183affdb","created_at":"2016-02-
12T15:35:44.253258+00:00","expires_at":"2016-02-13T15:35:44.253258+00:00"}
# Request all generated optimization plans, using the token acquired
$ curl -H "Authorization: Token 581e76b175a6b48c78bb39ee598284e7183affdb"
http://<OPTIMIZER_HOSTNAME_OR_IP>/dolfin/optimizer/api/plans/
```

```
# No optimization plans have been produced yet in a vanilla installation
{
  "count": 0,
  "next": null,
  "previous": null,
  "results": []
}
```

3.4.4.2. *Testing via the online documentation system*

In order to test the Optimizer installation and configuration assuming that the default configuration presented in the previous paragraphs has been followed, one should visit the page `http://<OPTIMIZER_HOSTNAME_OR_IP>/dolfin/optimizer/docs` which constitutes the landing page of the online documentation module supporting the operation of the Optimizer, already depicted in Figure 3-15. From there, one can get an authentication token by invoking the tokens API service and, then, query the Optimizer for generating a new optimization plan etc. Instructions on how to use the online documentation module can be found in the page of Swagger-UI[11] and in the deliverable D3.3, paragraph 3.3.4.1, where the online documentation system is presented for the eCOP Monitoring DB.

3.4.4.3. *Testing via the Optimizer Dashboard*

To check the installation of the dashboard, one is required to log into the Optimizer Dashboard login page. After successful login, the dashboard should expose a view similar to the one presented in 3.4.1.

3.4.5. Service endpoints and data model

The Optimizer has been configured to offer services categorized into two main groups of services together with the authentication service as follows:

- Plans
- Policies
- Requests
- Status
- Tokens (Authentication and Authorization)

3.4.5.1. *Plans (/api/plans/)*

The Plans service provides information over the generated optimization plans devised by the Optimizer. The following endpoints have been defined for this group:

[11] Swagger UI webpage, http://swagger.io/swagger-ui/.

HTTP Method	API Service Endpoint URL	Description
GET	`/api/plans/`	Retrieves all optimization plans devised by the Optimizer.
GET	`/api/plans/<id>`	Retrieves the optimization plan characterised by the id `<id>`
PUT	`/api/plans/<id>`	Updates the optimization plan characterised by the id `<id>`
PATCH	`/api/plans/<id>`	Patches the optimization plan characterised by the id `<id>`
DELETE	`/api/plans/<id>`	Deletes the optimization plan characterised by the id `<id>`
GET	`/api/plans/by-request-id/<id>`	Retrieves the optimization plan that was generated as a response to the optimization request with id `<id>`

Table 3-18 - API services exposed by the Plans group.

The data model of a plan is as follows:

Attribute	Type	Description
id	Integer	The id of the generated plan
policy	Policy	The policy according to which the plan was devised
request	Optimization Request	The optimization request that drove the process of generating the plan
algorithm	String	The algorithm used to generate the plan
algorithm_version	String	The version of the algorithm used
time	String	The ISO8601 representation of the time when the plan was generated
expected_benefit	Decimal	The expected benefit from applying the optimization plan
vm_migrations	List<VM Migration>	The list of VM Migrations expected to happen for implementing the particular optimization plan
vm_actions	List<VM Action>	The list of VM Actions expected to happen for implementing the particular optimization plan
server_actions	List<Server Action>	The list of server actions expected to happen for implementing the particular optimization plan
hvac_actions	List<HVAC Action>	The list of HVAC actions expected to happen for implementing the particular optimization plan
lighting_actions	List<Lighting Action>	The list of lighting systems actions expected to happen for implementing the particular optimization plan

Table 3-19 – Data model of an optimization plan

The data models of the entities included in the definition of the optimization plan data model are as follows:

Attribute	Type	Description
id	Integer	The id of the optimization request
time	String	The ISO8601 representation of the time when the request was issued
target	String	The target of the optimization request (optional)
prediction	String	A prediction to guide the optimization procedure (optional, deprecated)
status	String	The status of the request. Can be either 'RECV' to determine that the request was successfully received, 'WIP' to determine that an optimization plan is being generated as a response to this request, or 'DONE' to indicate that an optimization plan has been generated by the optimizer as a response to this request. (optional)

Table 3-20 – Data model of an optimization request

Attribute	Type	Description
id	Integer	The id of the VM Migration
uuid	String	The UUID of the VM to migrate
host	String	The server (the serial number of it) hosting the VM.
target	String	The target server (serial number)

Table 3-21 – Data model of a VM Migration object in the context of an optimization plan

Attribute	Type	Description
id	Integer	The id of the VM Action
uuid	String	The UUID of the VM to act upon
action	String	The action to perform on the VM (e.g. SHUT-OFF)

Table 3-22 – Data model of a VM Action object in the context of an optimization plan

Attribute	Type	Description
id	Integer	The id of the Server Action
serial_number	String	The serial number of the server to act upon
action	String	The action to perform on the server (e.g. hibernate)

Table 3-23 – Data model of a Server Action object in the context of an optimization plan

Attribute	Type	Description
id	Integer	The id of the Server Action
cooling	String	The HVAC system to act upon
action	String	The action to perform on the cooling system (e.g. hibernate)

Table 3-24 – Data model of an HVAC Action object in the context of an optimization plan

Attribute	Type	Description
id	Integer	The id of the Server Action
lighting_id	String	The id of the lighting system to act upon
state	String	The state of the lighting system to achieve

Table 3-25 – Data model of a Lighting Acton object in the context of an optimization plan

The Policy data model is described in §3.2.1

3.4.5.2. *Policies [/api/policies]*

The Policies service provides information over the supported DC policies governing the behaviour of the Optimizer. The following endpoints have been defined for this group:

HTTP Method	API Service Endpoint URL	Description
GET	`/api/policies/`	Retrieves all policies supported by the Optimizer.
GET	`/api/policies/<name>/`	Retrieves a supported policy based on its `name`.
DELETE	`/api/policies/<name>/`	Deletes a supported policy based on its `name`.
POST	`/api/policies/active/`	Sets a certain policy as active
GET	`/api/policies/active/`	Gets the active policy of the optimizer

Table 3-26 - API services exposed by the Policies group.

All APIs support the Policy data model described in §3.2.1.

3.4.5.3. *Requests [/api/requests/]*

The Requests service provides information about the optimization requests received and processed by the Optimizer.

HTTP Method	API Service Endpoint URL	Description
GET	`/api/requests/`	Retrieves all received optimization requests
POST	`/api/requests/`	Creates a new optimization request
DELETE	`/api/requests/<id>`	Deletes an optimization request
GET	`/api/requests/by-status/<status>`	Retrieves the optimization plans that share a common status identified by `<status>`.

Table 3-27 – API services exposed by the Requests group.

All APIs support the Optimization Request data model documented in Table 3-20.

3.4.5.4. *Status [/api/status/]*

The Status service may be used to retrieve the status of the optimizer. A single API service endpoint has been defined as documented in the following table:

HTTP Method	API Service Endpoint URL	Description
GET	`/api/status/`	Gets the status of the Optimizer

Table 3-28 – API service exposed by the Status group.

The data model of a Status object is as follows:

Attribute	Type	Description
id	Integer	The id of the status (can be ignored)
status	String	The status of the Optimizer
since	String	The ISO8601 representation of the time until when the Optimizer has this status
last_request	Integer	The id of the request that was last processed by the Optimizer

Table 3-29 –Data model of a Status object

3.4.5.1. *Authentication and Authorization [/api/tokens/]*

This service endpoint follows the same service endpoint URLs and data model as the Prediction Engine ones documented in §3.3.5.3.

3.5. VM Priority Classifier

The functionality of this component has been integrated into the Optimizer and is, hence, omitted in this section.

3.6. Policy Actuator

3.6.1. Basic concepts

The Policy Actuator has been implemented as a single module containing four different components.

a. Northbound API

The Northbound API is the entry point for all requests coming from the Optimizer. As explained in 2.1.6 of the present document, the Policy Actuator is exposed as a REST API, allowing an easy, light communication with remote components.

The REST API has been built and published by a third-party library called Apache CXF, which is an open-source-services framework providing mechanisms to easily build and develop web services using frontend programming APIs, such as JAX-RS in this case. By adding annotations to the classes and attributes to our model, the CXF automatically builds a REST API with the specifications described in section 2.1.6. More specifically, publishing the web resources in the specified URL, validating the format and content of the messages, and linking the REST resources to the classes of our module that will manage and receive those requests. Hence, the REST API is also responsible for translating the JSON messages to JAVA instances containing such information and vice-versa.

The base address of the northbound API can be easily and dynamically configured using the policy actuator configuration file, provided in its code and, hence, in its deployment package.

b. eCOP DB client

The eCOP DB client is a REST client used to log all received requests into the eCOP DB component. According to the eCOP DB API description defined in deliverable D3.1, the Policy Actuator implements a complete client allowing the communication with this component of the DOLFIN System.

As the northbound API, client is built using open source library CXF, which also provides methods to build Web Services clients. The Policy Actuator implements the JAVA representations of the messages to be sent to or received from the eCOP DB. The classes and attributes taking part of it are annotated with specific JAX-RS annotations, allowing CXF framework to automatically translate from JSON to JAVA and vice versa. It also provides ways to inspect the server response, allowing the Policy Actuator to handle errors in the communication and the message format. If the client launches any exception, this would be treated by the Policy Actuator and logged into local log file.

As described in D3.2 [3] , the eCOP DB API is secured by a token-based authentication. Therefore, all logging requests coming from Policy Actuator have to include a token for authentication and authorization. In order to add such token to the headers of all messages addressed to the eCOP DB, a CXF Interceptor has been implemented as part of the Policy Actuator. This Interceptor registers itself as a handler of outgoing messages before they are sent to the eCOP DB server adding the token to the message headers.

The remote address of the eCOP DB used to build the client can be easily and dynamically configured using the Policy Actuator configuration file, provided as part of the deployment package. Given that the eCOP DB requires an authentication token for all communications, this should be included in such file as well.

c. Southbound API

The southbound API is used by the Policy Actuator to communicate with the DCO Hypervisor Manager and the DCO Appliance Manager. The communication between those components is done via JAVA SPI (Service provider Interface). The SPI allows the developer to make extensible applications without modifying the original base code of the Policy Actuator. The base code declares the specification of the extensible service and other libraries can implement the interface without requiring modifications to the original application.

In this case, the Policy Actuator defines the Service Interface, which contains a single method to execute the actions that the Policy Actuator has translated from the Optimizer plan. It contains an internal registry to access to all available implementations of the Service Interface. This is carried out through the ServiceLoader, which is an SPI component responsible for searching and loading implementations of a specific service. The only restriction is that they should be in Policy Actuators classpath, so it forces the implementations to be in the same host and, of course, developed in same language.

The DCO Hypervisor Manager and the DCO Appliance Manager contain a so-called Adaptor. The Policy Actuator and DCO Broker communication is done through the adapters. Hence, these modules should contain an implementation for the Service Interface and notify to the SPI that they contain implementations for that interface. This is done by adding a specific file in JAR specifying which interface they implement and which is the file implementing it inside the local JAR file.

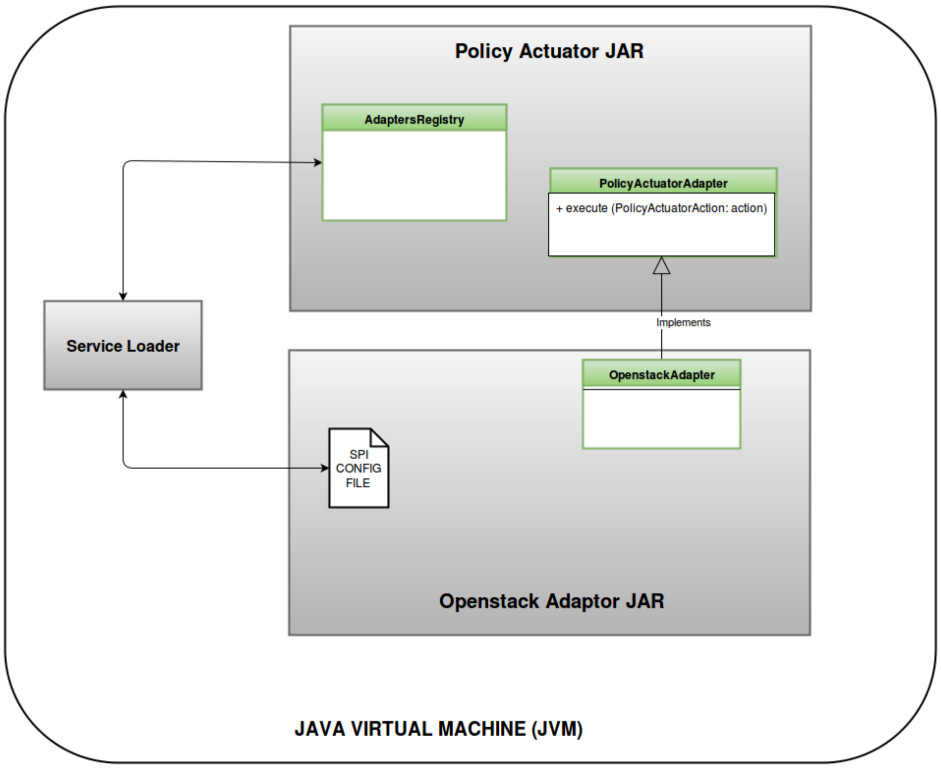


Figure 3-20 - Communication between Policy Actuator and DCO Brokers

d. Policy Actuator core

The core module of the Policy Actuator acts as an orchestrator, making usage of the three components described before. Each request addressed to Policy Actuator through the northbound API fulfils a specific workflow:

1) The request is handled by CXF library and translated into JAVA classes with information stored in their attributes. Each published REST resource is mapped to a method of a class instance, which are the methods of the Policy Actuator core.
2) The Policy Actuator translates received request into actions (AKA. Commands) to be performed by the DCO Hypervisor Manager and the DCO Appliance Manager.
3) The Policy Actuator core makes usage of the eCOP DB client to log the actions to be performed.
    a. If server or protocol fails, error is logged into local log file.
4) Actions are sent to the DCO Hypervisor Manager and the DCO Appliance Manager through the Southbound API.
    a. If a communication or the remote component fails, an error is logged into local log file.

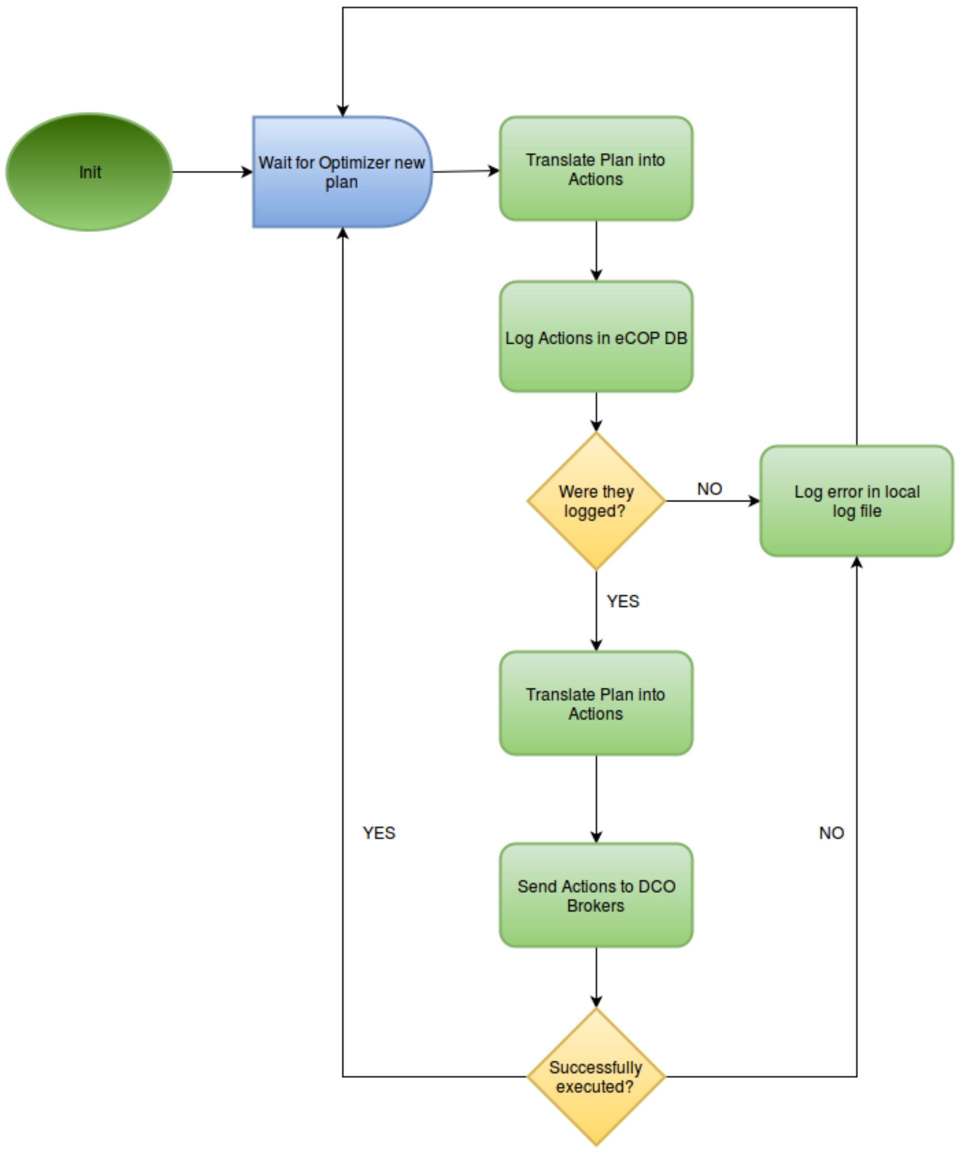

Init
Wait for Optimizer new plan
Translate Plan into Actions
Log Actions in eCOP DB
Were they logged?
NO
Log error in local log file
YES
Translate Plan into Actions
YES
NO
Send Actions to DCO Brokers
Successfully executed?

Figure 3-21 - Policy Actuator Workflow

3.6.2. Module Dependencies

The Policy Actuator module interacts with several components of the DOLFIN system in order to receive plans, translate them into commands and send them to the proper modules.

Module Name	Direction	Motivation	Protocol
Optimizer	To Policy Actuator	Optimizer defines a plan to be executed and it's passed to Policy Actuator, which is responsible for the plan execution.	REST

eCOP DB	From Policy Actuator	All actions to be applied in order to execute a plan are stored into the eCOP Database.	REST
DCO Hypervisor Manager	From Policy Actuator	Actions to execute a plan are translated to each specific DCO Hypervisor Manager so this can apply them.	Protocol determined by DCO Hypervisor Manager. REST in most cases (e.g. OpenStack)
DCP Appliance Manager	From Policy Actuator	Actions to execute a plan are translated to each specific DCO Appliance Manager so this can apply them.	Protocol determined by DCO Appliance Manager. REST in most cases.

Table 3-30 - Policy Actuator interfaces

No further direct dependencies exist between the Policy Actuator and the rest of DOLFIN's components.

3.6.1. Service endpoints and data model

As mentioned in previous paragraphs, the communication between the Optimizer and the Policy Actuator is performed through REST APIs. Once the Optimizer has calculated a plan to be executed in a specific DC managed by the DOLFIN system, it will send a notification to the Policy Actuator including the actions that should be executed in order to apply that plan. This API is the entry point to the Policy Actuator, and it allows to request:

- VM migration among servers.
- Delay VM instantiations to a specific time period.
- Modify operational state of a VM.
- Modify operational state of ancillary equipment.
- Modify operational temperature of air conditioning equipment.

In order to implement those, the API has been designed as described in following tables, this information is new as compared to D3.1 [1] .

	VM migration
Description	Used by Optimizer to request VM migration among servers of same or different DCs.
Endpoint Name	/vm/migrate
Allowed Methods	PUT
Body	List containing the ids of the VMs to be migrated together with the id of the destination server.

Provides response	No
Response Parameters	None
Response code	200
Requester	Optimizer

Example	{
	  "vms":[
	    {
	      "vm_uuid":"vm1",
	      "server_sn":"server2"
	    },
	    {
	      "vm_uuid":"vm2",
	      "server_sn":"server6"
	    }
	  ]
	}

Table 3-31 - Policy Actuator: VM migration API

	VM shift
Description	Used by Optimizer to request that a VM instantiation gets postponed until a specific time period.
Endpoint Name	/vm/shift
Allowed Methods	POST
Body	List containing ids of the VMs which instantiation should be postponed, together with the specific time when the VM should be instantiated. The expected format of the date is “yyyy-MM-ddThh:mm:ssZ”
Provides response	No
Response Parameters	None
Response code	200
Requester	Optimizer

Example	{
	  "vms":[
	    {
	      "vm_uuid":"vm1",
	      "start_time":"2015-12-10T20:00:02Z"
	    },
	    {
	      "vm_uuid":"vm2",
	      "start_time":"2015-12-10T20:00:02Z"
	    }
	  ]
	}

Table 3-32 - Policy Actuator: VM shift API

	DVFS
Description	Used by Optimizer to request to scale down the voltage and frequency of a specific server by a given percentage.
Endpoint Name	/server/dvfs
Allowed Methods	PUT
Body	List containing the serial numbers of the servers which voltage and frequency should be scaled down, together with the percentage to apply.
Provides response	No
Response Parameters	None
Response code	200
Requester	Optimizer
Example	{ "dvfs":[{ "server_sn":"server1", "scale":"40" }, { "server_sn":"server2", "scale":"5" }] }

Table 3-33 - Policy Actuator: DVFS set API

	Server hibernation
Description	Used by Optimizer to request that a specific server enters into hibernation mode.
Endpoint Name	/server/hibernate
Allowed Methods	PUT
Body	List containing the ids of the servers to hibernate.
Provides response	No
Response Parameters	None
Response code	200
Requester	Optimizer
Example	{ "servers":["server1", "server2"] }

Table 3-34 - Policy Actuator: server hibernation API

	Server wakeup
Description	Used by Optimizer to request that a specific server goes back into operational mode.
Endpoint Name	/server/wakeup
Allowed Methods	PUT
Body	List containing the ids of the servers to wake up.
Provides response	No
Response Parameters	None
Response code	200
Requester	Optimizer

Example	{
	  "servers":[
	    "server1",
	    "server2"
	  ]
	}

Table 3-35 - Policy Actuator: server wakeup API

	Ancillary equipment hibernation
Description	Used by Optimizer to request that a specific ancillary equipment enters into hibernation mode.
Endpoint Name	/airconditioning/hibernate
Allowed Methods	PUT
Body	List with the ids of the ancillary equipment which should enter into hibernation mode.
Provides response	No
Response Parameters	None
Response code	200
Requester	Optimizer

Example	{
	  "air_conditioning":[
	    "air_conditioning_1",
	    "air_conditioning_2"
	  ]
	}

Table 3-36 - Policy Actuator: ancillary equipment hibernation API

	Ancillary equipment wakeup
Description	Used by Optimizer to request that a specific ancillary equipment goes back into operational mode.
Endpoint Name	/airconditioning/wakeup
Allowed Methods	PUT
Body	List with the ids of the ancillary equipment which should enter into operational mode.

Provides response	No
Response Parameters	None
Response code	200
Requester	Optimizer

Example	{
	"air_conditioning":[
	"air_conditioning_1",
	"air_conditioning_2"
	]
	}

Table 3-37 - Policy Actuator: ancillary equipment wakeup API

	Air conditioning temperature
Description	Used by Optimizer to request that a specific air conditioning equipment works in a specific temperature.
Endpoint Name	/airconditioning/temperature
Allowed Methods	PUT
Body	List with the ids of the air conditioning equipments and the temperature to set in each of them. Temperature should be defined in Celsius.
Provides response	No
Response Parameters	None
Response code	200
Requester	Optimizer

Example	{
	"air_conditioning":[
	{
	"air_cond_sn":"air_conditioning_1",
	"temperature":"20.0"
	},
	{
	"air_cond_sn":"air_conditioning_2",
	"temperature":"17.0"
	}
	]
	}

Table 3-38 - Policy Actuator: air conditioning API

3.6.2. Deployment and installation

The outcome of the compilation of the core module is a Java Archive (JAR). This file format is OS independent so it can run on Windows, Linux or MacOS. In this section the instructions that describe how to deploy Policy Actuator in a Linux machine are provided.

Java Runtime Environment (JRE) version 8 is the only mandatory library the user should install in its computer in order to execute the module.

The program requires a configuration file to load some properties from. The implementation contains a sample configuration file with all required properties called "*policyactuator.conf*". Basically, administrator should configure the base URL of the northbound API, the URL to access the eCOP DB and the token to be used for authentication and authorization. Once installed, the Policy Actuator module could be executed by running following command:

```
$ java -jar policy-actuator-0.0.1.jar |
$PATH_TO_CONFIG_FILE
```

where PATH_TO_CONFIG_FILE contains either absolute or relative path to configuration file. Second parameter may differ according to the configuration file location. The execution launches a server process. To stop the server, simply interrupt the program.

Only in Linux based OS, the process can be simplified making use of the available scripts located in "target" directory after a fresh build.

```
$ ./start.sh $PATH_TO_CONFIG_FILE
…
$ ./stop.sh
```

In order to launch it together with an adapter, the adapter must be in the classpath of the Actuator. After building both the Actuator and the adapter, the following command must be used to launch them:

```
$ java -cp policy-actuator-0.0.1.jar:$PATH_TO_BROKER |
net.i2cat.seg.dolfin.ictpes.actuator.entrypoint.EntryPoint
$PATH_TO_CONFIG_FILE
```

where PATH_TO_CONFIG_FILE contains either absolute or relative path to configuration file, or PATH_TO_BROKER also absolute or relative path to adapter library file.

3.6.3. Testing the installation

Once Policy Actuator has been deployed, the administrator could guarantee that the module has been successfully initialized. In order to do that, it is important to know the base address where the server has been deployed, which is configured in configuration file explained in section 3.6 of this document.

Assuming that the file has been configured to publish the server in *localhost* on port *9000*. Administrator (and also regular developers) can access using any web browser to the Northbound API WADL document, which should be published in http://*BASE_ADDRESS/policyactuator?_wadl* URL. (in our example, http://localhost:9090/policyactuator?_wadl.

WADL stands for Web Application Description Language and it is a machine-readable XML description of a REST Web Service. By accessing this URL, administrator should be able to see the Policy Actuator Northbound API WADL document. Otherwise, it would mean that there was an unexpected exception while instantiating the Policy Actuator, so the administrator should take care of the error messages either displayed in the shell while starting the service or stored in the application log file (by default /var/log/policyactuator.log).

3.6.4. Other module-specific subsections

3.6.4.1. *Online API documentation*

As described in chapter 3.6.1, the Northbound API has been built and deployed using Apache CXF Library. When creating REST Web Services, CXF automatically generates and publish a WADL document, which is a machine-readable XML description of the API. It is divided in two main sections:

- Grammar: Container for definitions of the format of the exchanged data between client and server.
- Resources: Container for resources provided by the API. Each resource consists of a relative path, an HTTP method, a content-type specification and a reference to the exchanged data, described in grammar section.

This document is very useful to create Web Services clients. There are tools, such as CXF that can generate a complete Web Service client by translating the WADL document to Java classes (but also other alternatives such as python). Taking "*wadl2java*" as reference, provided by CXF, implementing a JAVA client for a WADL-documented API is as simple as executing following command (in a Linux-based OS);

```
$ ./wadl2java -p $PACKAGE_NAME -impl -compile |
http://$SERVER_ADDRESS/policyactuator_wadl?
```

where PACKAGE_NAME describes the name of the package where the classes will be created, and $SERVER_ADDRESS contains base address of the server where Policy Actuator is deployed, including address, port and context, if any.

3.6.4.2. *Log files*

The Policy Actuator module logs all its activities in a local file, including errors in the communication or in the interactions with other DOLFIN components using Log4j library. Log4j is an open source logging library for printing log output in local and remote destinations. It is fully configurable at runtime using external configuration files.

The Policy Actuator provides its own configuration file for log4j library, performing customized configurations on the log file location, the maximum size and file appenders of the logging, the logging level and the logging messages format.

A set of actions to the log file, such as:

- Policy Actuator is started or stopped.
- Northbound API receives a request to a non-existing resource.

- Northbound API detects a wrong message format or an unexpected content-type when Optimizer sends the new plan.
- Policy Actuator successfully logs the actions to be performed by the DCO Brokers into the eCOP DB.
- eCOP DB is neither available nor reachable.
- eCOP DB launches an unexpected exception when the Policy Actuator sends the actions to be executed by the DCO Brokers.
- There is no available DCO Broker that can execute the actions.
- The DCO Broker launches an unexpected exception when trying to execute the received actions.
- The DCO Broker successfully executes the actions, so plan has been execute.

These actions are logged into different levels according to their value or the impact they have in the system.

4. Conclusions

This document provided a complete description of the Energy Efficiency Policy Maker and Actuator implementation and the external interfaces that are exposed by each modules.

For each module the main functions have been defined in relation to the specifications that were identified during the initial design. Compared to what were already defined during the design phase, this document contains a number of improvements and revisions of the functionality in addition to other details regarding the implementation as well as the description of the different logics that are supported by each module.

In particular, the document describes the operation of the four phases principal taken by the Energy Efficiency Policy Maker and Actuator:

- Phase 1: DC workload assessment and prediction, performed by the Predictor Engine module.
- Phase 2: Coordination of the intra-DC optimization and management activities, as well as of handling of various interfaces to internal and external modules for the reception of trigger and for the activation of new optimization procedures. This coordination role is attributed to the Policy Maker module which is attached to the Policy Repository to manage operations conforming to defined energy policies.
- Phase 3: creation of optimization path at the DC level, taking into account DC status information, SLA conditions, policy constraints, etc. These phase represents the actual optimization process that is sustained by the Optimizer Engine module.
- Phase 4: interfacing with various brokers DCO (VMs Hypervisor, Appliances, etc.) for the effective implementation of the optimization path.

Finally, the document also shows the functionalities and interfaces required for inter-DC integration to support (in cooperation with the developments performed in the WP4 context) the energy optimization procedures at the level of DOLFIN synergetic-DCs environment.

References

[1] DOLFIN_D3.1_NXW_FF-20150430, PU deliverable, http://www.dolfin-fp7.eu/wp-content/uploads/2015/07/DOLFIN_D3.1_NXW_FF-20150430.pdf

[2] DOLFIN D3.3 - ICT Performance and Energy Supervisor component (Implementation) – CO deliverable

[3] DOLFIN D3.2 – Water cooling server module – CO deliverable

[4] DOLFIN D4.3 – SLA Renegotiation Controller component (Implementation) - CO deliverable

[5] DOLFIN D4.4 - Workload and VM Manager component (Implementation) - CO deliverable

[6] https://www.usenix.org/legacy/event/hotpar11/tech/final_files/Bird.pdf

[7] R. Urgaonkar, U. C. Kozat, K. Igarashi and M. J. Neely, "Dynamic resource allocation and power management in virtualized data centers," Network Operations and Management Symposium (NOMS), 2010 IEEE, Osaka, 2010, pp. 479-486.

[8] CVXOPT homepage, http://cvxopt.org/