Melodic
Big data cloud

Title:

D3.5 MELODIC Upper Ware

Abstract:

This is the last report of workpackage WP3, accompanying the final software release of all the Melodic Upperware components. These Upperware components refer to Melodic's substantial functionalities that enable the application optimisation recommendation, the initial application placement and the continuous adaptation enactment. Based on Upperware features the Melodic platform offers timely decision-making on optimised cross-cloud data placements and application deployments.

In this deliverable we provide a detailed description concerning the approach, the business logic and the implementation details of all the Upperware components that were integrated as part of the Melodic platform release 3.0. This document concludes with an analysis on the differences with the PaaSage platform, that MELODIC had to be built on and significantly extend as part of the description of work.

Multi-cloud Execution-ware for Large-scale Optimised Data-Intensive Computing

H2020-ICT-2016-2017
Leadership in Enabling and Industrial Technologies; Information and Communication Technologies

Grant Agreement No.:
731664

Duration:
1 December 2016 -
29 February 2020

www.melodic.cloud

Deliverable reference:
D3.5

Date:
29 February 2020

Responsible partner:
ICCS

Editor(s):
Yiannis Verginadis

Author(s):
Yiannis Verginadis, Ioannis Patiniotakis, Vasilis Stefanidis, Fotis Paraskevopoulos, Evagelia Anagnostopoulou, Kyriakos Kritikos, Paweł Skrzypek, Marcin Prusiński, Marta Różańska

Approved by:

ISBN number:
N/A

Document URL:
http://www.melodic.cloud/deliverables/D3.4 Workload optimisation recommendation and adaptation enactment.pdf

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Document	
Period Covered	M26-36
Deliverable No.	D3.5
Deliverable Title	D3.5 MELODIC Upper Ware
Editor(s)	Yiannis Verginadis
Author(s)	Yiannis Verginadis, Ioannis Patiniotakis, Vasilis Stefanidis, Fotis Paraskevopoulos, Evagelia Anagnostopoulou, Kyriakos Kritikos, Paweł Skrzypek, Marcin Prusiński, Marta Różańska
Reviewer(s)	Daniel Baur, Tomasz Przeździęk
Work Package No.	3
Work Package Title	Upper ware
Lead Beneficiary	ICCS
Distribution	PU
Version	2.6
Draft/Final	Final
Total No. of Pages	75

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Table of Contents

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

List of Tables

List of Figures

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

1 Introduction

This document presents the design and implementation details of the final version of Melodic Upperware. Parts of the document require a high-level understanding of the Cloud technologies and the Melodic platform, for which readers are referred to [1]. In addition, for all the Upperware components we provide an overview of the changes (in comparison to their previous version reported in [2]) that led to their final version.

The MELODIC components that are presented in this deliverable are mainly related to the application optimisation recommendation, initial application placement and continuous adaptation enactment. Essentially, we are referring to the Upperware components as defined in [1], [2] and depicted in Figure 1.

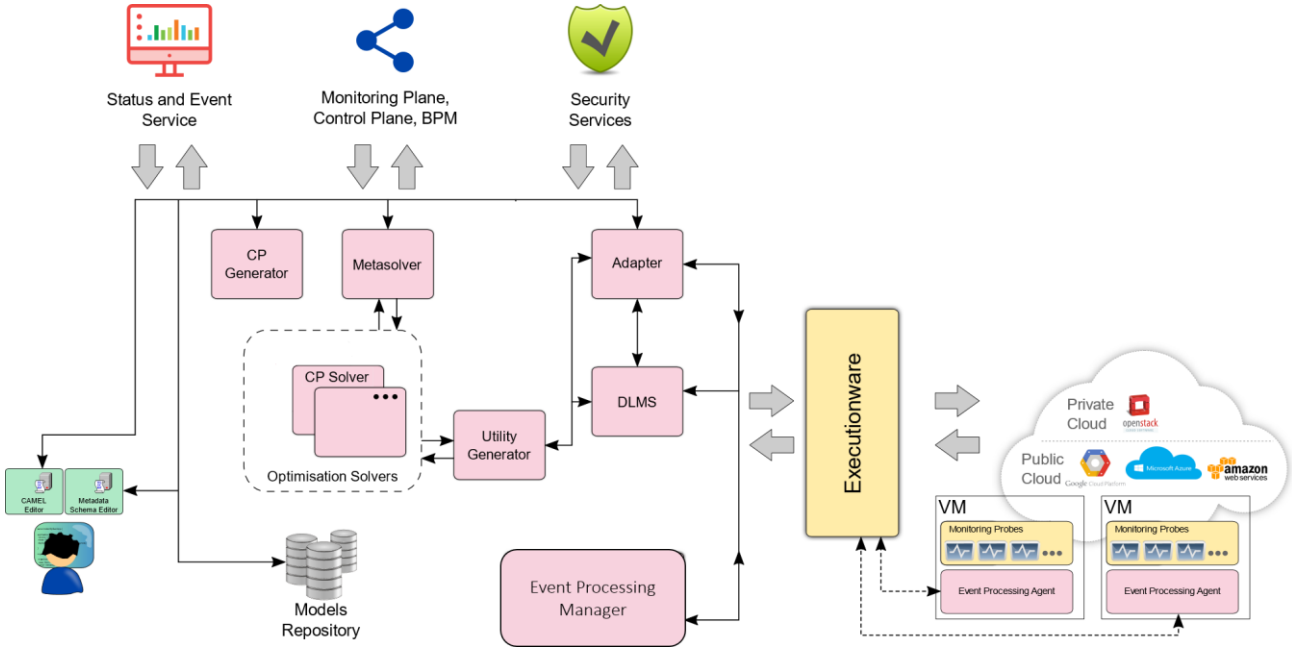

Figure 1. Overview of the Upperware Components

As depicted in Figure 1, the Upperware comprises dedicated software components that introduce all the appropriate functionality for reaching decisions on appropriate cross-cloud data placements and application deployments at the pertinent time. Specifically, in this deliverable, we present the design and implementation details of the following components:

- CP Generator – for generating a formal Constraint Problem (CP) out of a set of a cloud application placement requirements;
- Metasolver – for coordinating and supporting the CP solving process and deciding when reconfigurations are required;
- CP Solver - for finding, in a stateless manner, optimal cross-cloud resources allocation

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

and application placement according to a set of pre-defined requirements captured as a CP. We note that Upperware can integrate any new optimisation solver according to the preferences of the MELODIC adopter;
- Utility Generator – for calculating the utility function value for the deployment solutions proposed by the optimisation solver configuration;
- Adapter – for creating the target application configuration to be deployed into cross-cloud resources and relaying appropriate instructions to the Cloudiator [5];
- Event Management System – for collecting, processing and delivering monitoring information pertaining to a cross-cloud application, deployed and maintained by the Melodic platform;
- DLMS – for enabling the holistic management of the data lifecycle in Cross-Cloud environments;
- Graphical User Interface – for augmenting the user-friendliness of the MELODIC platform.

Each of these Upperware components are discussed in separate chapters of this deliverable by presenting the core aspects of the approach considered for each of them, along with their business logic and technical implementation details. We note that further details on the inter-component communication and API specifications have been provided in the MELODIC deliverable D2.3 [6].

Last, we conclude this deliverable with a section that analses the differences to the PaaSage platform, since according to the description of work it was stated that MELODIC will be built on and significantly extend the PaaSage platform.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

2 CP Generator

Mission: Create Constraint Problem (CP) to be solved by any of the Solvers.

Positioning in Melodic: CP Generator is one of the microservices comprising the Upperware of the Melodic platform.

2.1 Approach

High-level Approach: The CP Generator is responsible for creating the Constraint Problem (CP) Model based on the provided CAMEL Model and a set of Node Candidates[1] fetched from Cloudiator according to the hard requirements described in CAMEL.

Functionalities:

- Creating hard requirements in order to fetch matching Node Candidates from Cloudiator
- Storing fetched Node Candidates in cache for further use
- Creating a Constraint Problem based on requirements from CAMEL Model and border values of corresponding fields of Node Candidates
- Storing Constraint Problem Model to CDO
- Sending Success or Failure Notification in case of any errors

Input:

- CAMEL model
- Node Candidates fetched from Cloudiator

Output:

- Constraint Problem Model stored in CDO
- Node Candidates stored in cache
- Notification sent to the control process

[1] We note that node candidates refer to a list of flavours of virtual machines from all the available cloud service providers that satisfy a certain constraint. This list is fetched from Cloudiator in order the CP generator to be able to create the Constraint Problem

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

2.2 Technical Implementation

2.2.1 Architecture

As shown in Figure 2, the CP Generator comprises the following sub-components:

- Controller - provides the REST API of the CP Generator for receiving process calls from the control layer
- Generator Orchestrator - orchestrates the creation of a new Constraint Problem
- CDO Service - retrieves the CAMEL Model from the Models repository and stores the created CP Model to the Models Repository
- New Constraint Problem Service - creates a new Constraint Problem based on information from CAMEL and the fetched Node Candidates
- Notification Service - notifies the control process
- Cloudiator Service - fetches the Node Candidates fulfilling the requirements
- Cache Service - creates a Melodic cache with a map of fetched Node Candidates for each component

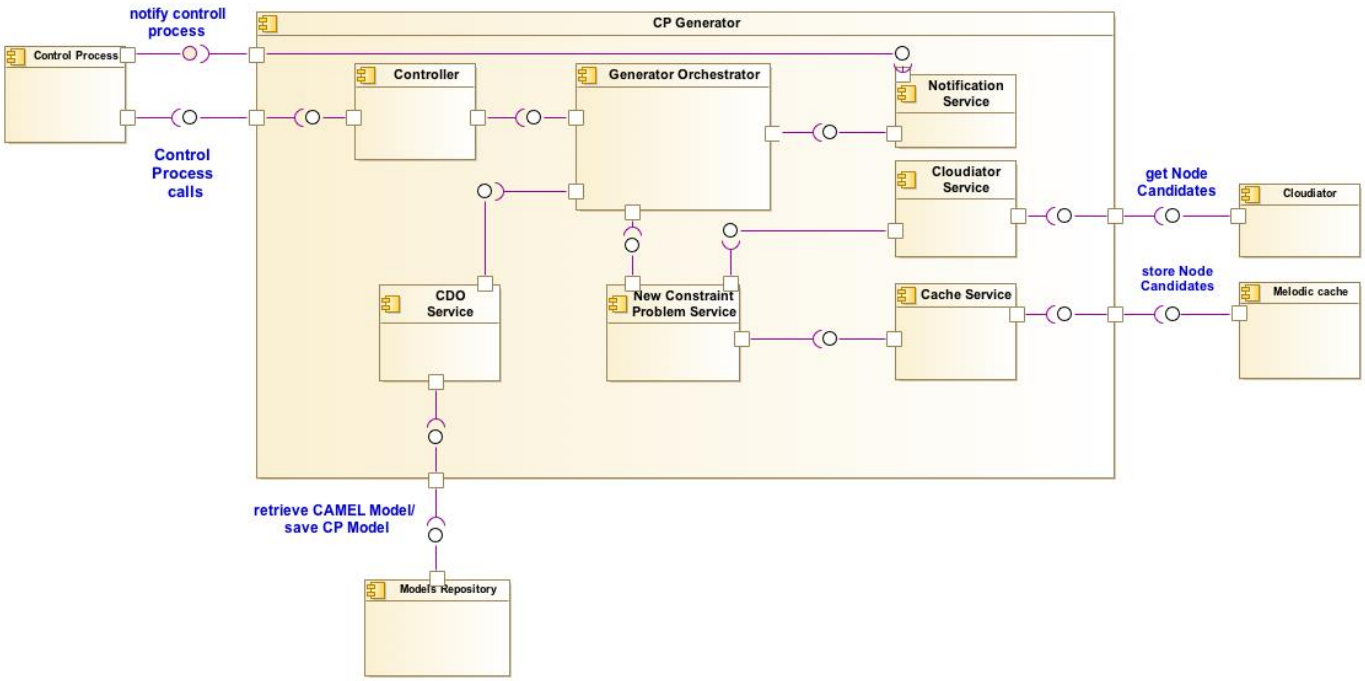

Figure 2. CP Generator in Upperware Component diagram

In the following UML collaboration diagram, we describe the flow of information between the CP Generator and the rest of the Upperware components with which it interacts. Specifically,

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

the following messages are exchanged:

- with the Control service
  - **CP Model generation**: Message that contains information with the id of the CAMEL Model from which the CP Model should be created
  - **Notification**: The Notification that the CP Generator sends to the Control Service to inform about the result of the CP Model creation
- with the Cloudiator
  - **Get Node Candidates:** Message with the requirements from the CAMEL Model to fetch the possible Node Candidates
- With the Melodic cache
  - **Store Node Candidates:** the Message that stores fetched Node Candidates
- with the Models Repository
  - **Retrieve CAMEL Model:** This message asks from the Models Repository to fetch the CAMEL model.
  - **Save CP Model:** The message about storing the new Model to the Models Repository

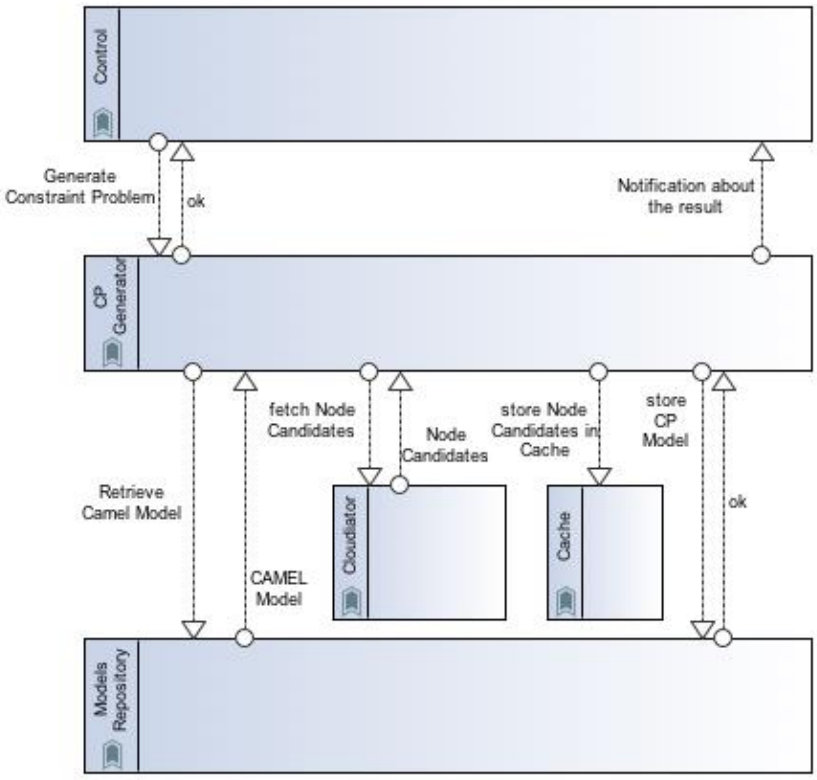

Figure 3. CP Generator Collaboration diagram

2.2.2 Implementation

The CP Generator has been written in Java 8. The project is built using Maven and thanks to the Maven plugin[2], a Docker image is created which allows running this component as a separate microservice.

The CP Generator's source code is available in Bitbucket at:
https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/cp_generator

Dependencies: CDO Client, Melodic cache, Cloudiator Client, MathParser, Spring-boot framework, JWT commons, Melodic commons.

2.2.3 CP-Generator Configuration

The CP-Generator configuration is stored in `eu.melodic.upperware.generator.properties` file, using Java properties format. The most important aspects of the required configuration are the following:

- esb.url - the endpoint of Upperware Control process, which CP-Generator invokes to signal that application reconfiguration is required.
- cloudiatorV2.url - the endpoint of Cloudiator Service
- cloudiatorV2.apiKey - authorization key for REST API of Cloudiator
- cloudiatorV2.httpReadTimeout - HTTP read timeout in ms
- logging.config - path to log configuration file

Below a sample CP-Generator configuration file:

```
#### Communication with ESB ####
esb.url=http://localhost:8088


cloudiatorV2.url=http://localhost:8089/api/adapter/cloudiatorProxy/api/v2
cloudiatorV2.apiKey=xxxxx
cloudiatorV2.httpReadTimeout=60000


### logback configuration ###
logging.config=file:${MELODIC_CONFIG_DIR}/logback-conf/logback-spring.xml
```

[2] `com.spotify:docker-maven-plugin`

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

2.3 Overview of changes of Finalized CP-Generator

The final version of CP Generator is the update of the R2.0 version, reported in D3.4 [2] with few bug fixes and improvements. The CP Generator has not been changed significantly over the last release.

The most important changes are connected with the security and the use of JWT-based authentication in its interactions with other platform components.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

3 Metasolver

Mission: Coordinate and support the Constraint Problem (CP) solving process.

Positioning in Melodic: Metasolver is one of the microservices comprising the Upperware of the Melodic platform.

3.1 Approach

High-level Approach: The Metasolver undertakes the task of selecting an appropriate solver for a given CP problem and subsequently verify that the solution yielded by this solver is significantly better, in terms of utility value, than the currently deployed one. A configurable threshold is used for specifying how much better the new solution must be in order to be deployed.

Functionalities:

- Select a solver given a CP model. Currently the solver selected needs to be specified in configuration file.
- In case an initial application placement has already been realized, update the CP model with the most current metrics values from the application monitoring infrastructure (EMS – see chapter 7), as well as with the most recently deployed solution (i.e. description of application deployment topology) from the Adapter (see chapter 6).
- Evaluate a given CP solution, provided by the selected solver and *accept* or *reject* it, by comparing it to the currently implemented solution (if any). This comparison is based on the utility values of the two solutions.
- Receive events signalling SLO violations from the application monitoring infrastructure (EMS) and trigger a new reconfiguration process.
- Receive events conveying metric values and update CP model with them.

Input:

- CP model
- Metric and SLO violation event subscription information (from EMS)
- Monitoring information in the form of metric values from events relayed through EMS
- Reconfiguration Events, signalling SLO violations

Output:

- Solver selected for solving a given CP problem

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

- Acceptance or Rejection of a new solution (yielded by a solver)
- Signal the start of a new reconfiguration process in response to an SLO violation event.

3.2 Technical Implementation

In this subsection, all the technical details on the Metasolver implementation are discussed (e.g. language, frameworks, 3rd party libraries, code structure etc.).

3.2.1 Architecture

A high-level depiction of the Metasolver architecture is given through the following UML component diagram (Figure 4).

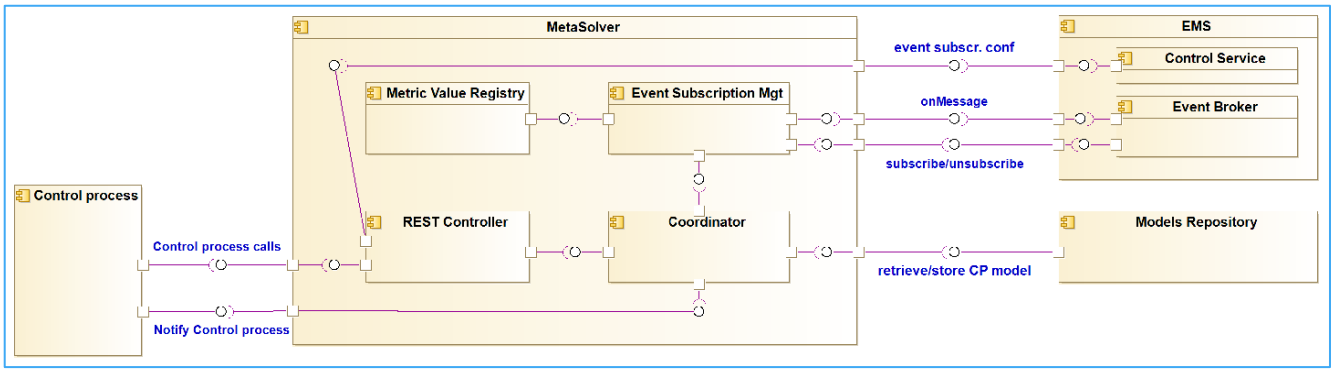

Figure 4. Metasolver in Upperware Component diagram

As shown in Figure 4, the Metasolver comprises the following sub-components:

- **REST Controller:** provides the REST API of the Metasolver for receiving process calls from the control layer (see [6] for more details) and configuration information from EMS at runtime
- **Coordinator:** orchestrates the functioning of the whole component and provides the business logic of MetaSolver.
- **Event Subscription Management:** subscribes to the configured event topics registered in the Event Broker of EMS. EMS provides this configuration, which includes event topic names & URLs and an indication whether the corresponding events must be used for updating CP model or they signal a SLO violations (hence a reconfiguration process must start). Event Subscription Management is also responsible for unsubscribing when the configuration changes or when Metasolver shuts down.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

- **Metric Value Registry:** maintains an in-memory catalogue of metric values. The values are extracted from the corresponding events, received from the EMS Event Broker. When the Metasolver is requested to select a solver (for application reconfiguration), it first updates the corresponding CP model with the most recent metric values.

The Metasolver internally uses an instance of the CDO client in order to communicate with the Upperware Models Repository and retrieve or modify the application CP model. Moreover, it uses an instance of the Broker Client in order to communicate with Event Broker of EMS and subscribe to event topics for receiving events. Both CDO and Broker clients are imported as dependencies from other Melodic-platform modules, namely *CDO client* and *Broker Client* of EMS.

In the following UML collaboration diagram, we describe the flow of information between Metasolver and the rest of the Upperware components with which it interacts. Specifically, the following messages are exchanged:

- with the Control service
  - **Constraint enhancement & Solver Selection**: Message that contains information for selecting an appropriate solver for a given CP problem
  - **Solution evaluation**: Message that includes information for the possible solvers that can be used.
  - **Solution result**: Message that includes the appropriate solver that will be used.
  - **Deployment result:** Upon request by the Upperware control process, Metasolver evaluates a given CP solution, captured in a CP model, and indicates whether it should be realized as the new application VM deployment topology or not (i.e. accept or reject it).
  - **Application Reconfiguration:** Message that in case the new solution is acceptable signals the start of a new reconfiguration process.
  - **Process id:** Message that includes the process id of the new reconfiguration process that is started.

- with the EMS service
  - **Event Subscription Configuration:** Upon request by EMS, Metasolver receives a new event subscription configuration and applies it. This configuration also indicates the event broker where each topic has been registered to, as well as whether each event topic provides *Metric Value* events or *SLO violation* events.
  - **Topic Subscription:** Shows the event topic of the new *Metric Value or SLO violation* event.
  - **New event:** any raw or processed event propagated from the EMS to Metasolver
  - **CP Model Updated:** It shows to EMS if the CP model is updated or not after the evaluation of the solution (old versus the current).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

- with the Models Repository
  - **Retrieve CP Model:** This message asks from the Models Repository to fetch the current CP model.
  - **CP Model:** The message includes the description of CP model currently used.
  - **Update Model:** If the solution from Metasolver showed a new CP model then there is a notification about transferring and storing of this new Model to Models Repository.

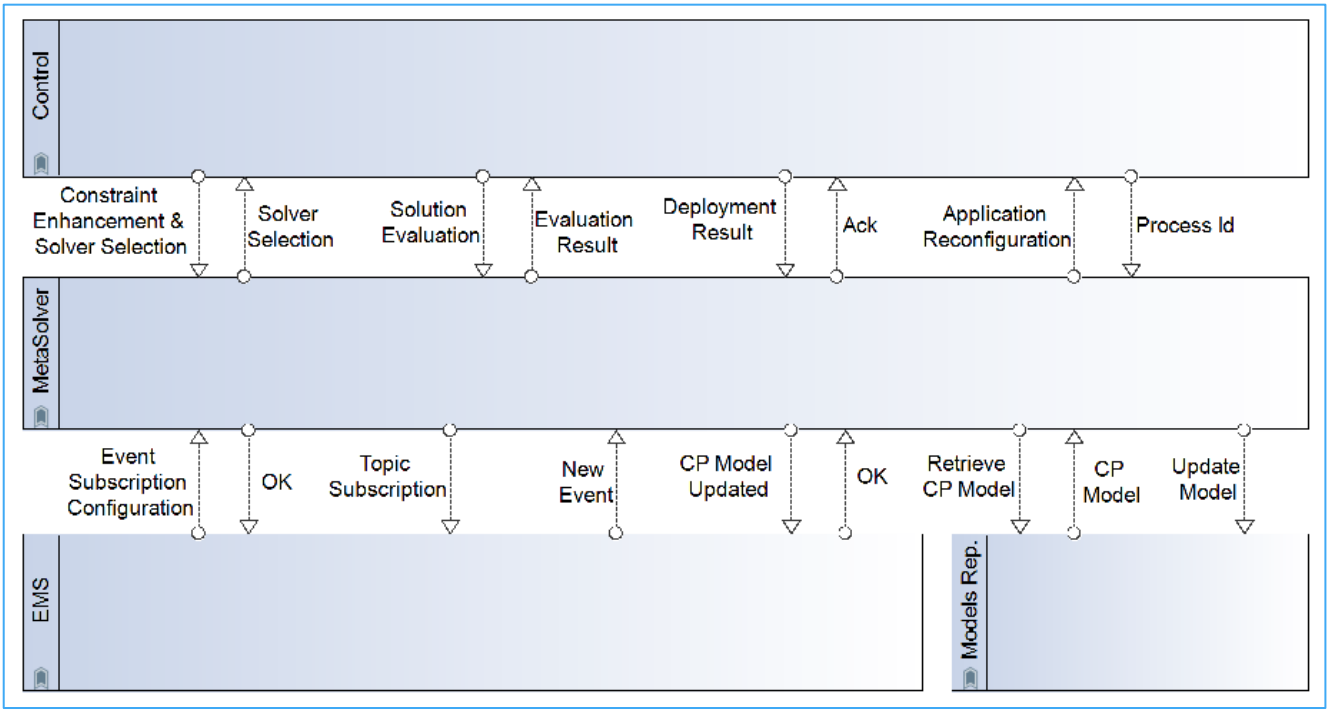

Figure 5. Metasolver Collaboration diagram

3.2.2 Implementation

Metasolver has been implemented using the Java programming language, version 8. It has been developed as a Spring-boot application for easier dependency management and customization of component properties. It is built and bundled, using the well-known Maven system, into a single fat JAR, containing Metasolver classes and dependencies. It is also bundled (during its building with Maven) as a Docker container and subsequently been added in the Melodic-platform component swarm.

Metasolver's source code is available in Bitbucket at:

https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/meta_solver

Dependencies: Broker-Client library, CDO client, Spring-boot framework

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

3.2.3 Metasolver Configuration

The Metasolver configuration is stored in `eu.melodic.upperware.metasolver.properties` file, using Java properties format. The most important aspects of the required configuration are the following:

- **ESB URL:** the endpoint of Upperware Control process, which Metasolver invokes to signal that application reconfiguration is required.
- **EMS URL:** the endpoint of EMS, which Metasolver invokes to notify EMS about a new CP model.
- **EMS event broker credentials:** the credentials used to connect to EMS event broker, if authentication is enabled
- **PubSub:** event subscription configuration loaded at boot-time. It includes:
  - an Enable flag (to turn-off event subscriptions)
  - a Topics list (see next)
- **Utility Threshold Factor:** is the percentage by which the utility value of the new solution must exceed the utility value of the current solution. Utility values are used to express the suitability of solutions to a certain Constraint Problem, and they are also used to compare solutions (of the same CP). In order to avoid deploying solutions that will yield just marginal benefit to the application, it is required the utility value of a new solution to be clearly better than the utility value of the currently deployed solution. This is expressed as the percentage by which the new solution utility value exceeds current solution's utility value.
- **Topic configuration (elements of PubSub Topics list):**
  - **Name:** of the event topic (e.g. AverageResponseTime)
  - **URL:** The endpoint of the corresponding Event Broker (e.g. tcp://172.24.29.139:61616)
  - **Client Id:** an optional name for the event broker client used internally
  - **Event Type:** (i) Metric Value event, or (ii) SLO violation event

Next, we provide a sample Metasolver configuration file:

```
#### Communication with ESB ####
esb.url = http://mule:8088/api/metaSolver/deploymentProcess

#### Communication with EMS ####
ems-url = http://ems:8111/cpModelJson
ems-broker-username=...
ems-broker-password=...
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

```
#### REST interface port ####
server.port = 8092

#### Pub/Sub configuration for Metrice Values and Scale events ####
pubsub.on = true

#### Settings for boot-time event subscription configuration
pubsub.topics[0].name = AverageResponseTime
pubsub.topics[0].url = tcp://ems:61616
pubsub.topics[0].clientId = Metasolver-Metric-Value-Monitor-Bean-client
pubsub.topics[0].type = MVV

#### New solutions must have utilities at least '1.1' times higher than
#### the utility of the deployed solution
utility-threshold-factor = 1.1
#### Default solver
default-solver = CPSOLVER

### logback configuration ###
logging.config = file:${MELODIC_CONFIG_DIR}/logback-conf/logback-
spring.xml
```

We note that the Metasolver internally uses a CDO client to communicate with the Upperware Models Repository, in order to retrieve and modify the application CP model. The CDO client reads its configuration from `eu.paasage.mddb.cdo.client.properties` file.

3.3 Overview of changes towards Metasolver Finalization

The final version of MetaSolver is essentially an update of the RC2.0 version, reported in D3.4, in order to include the new features introduced in Melodic platform since then and provide a few bug fixes and code improvements. The most notable differences are the:

- use of JWT-based authentication in its interactions with other platform components;
- support of encrypted (HTTPS-based) communication with other platform components;
- support of encrypted communication with EMS event broker (ActiveMQ-SSL protocol); and use of event broker credentials (provided by EMS during MetaSolver

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

configuration).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

4 CP Solver

Mission: Solve a CP model after it has been generated by the CP Generator or updated by the Metasolver, in cooperation with the Utility Generator.

Positioning in Melodic: The CP Solver, part of the Upperware module, is available in a micro-service form.

4.1 Approach

High-Level approach: The CP Solver is one of the solvers available in the Melodic platform utilised for solving a CP model. Such a solver can be used both for performing initial application deployment reasoning as well as application redeployment reasoning. Once the CP model is received, it is transformed into an internal representation which is fed into the CP solving engine. During CP model solving, the CP Solver cooperates with the Utility Generator in order to compute the utility of the currently examined candidate solution. Once the CP model solution is produced, it is incarnated inside the CP model in a certain specialised part.

Functionalities:

- Solves a CP model
- Cooperates with the Utility Generator during the CP model solving
- Registers the discovered solution within the CP model manipulated

Input:

- CP Model (path to that model within the CDO Model Repository)

Output:

- CP Model enhanced with the computed solution (i.e., the CP model is extended by incorporating the solution that has been computed by the CP Solver and a path to that model within the CDO Model Repository is returned)

4.2 Technical Implementation

In this subsection, the technical details of the CP Solver's implementation are discussed.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

4.2.1 Architecture

The internal architecture of the CP Solver is shown in the component diagram of Figure 6. As it can be seen, this component comprises the following three sub-components:

- CP Solver: this sub-component is responsible for computing the optimal solution to a CP model, which already resides in the (CDO) Models Repository, and incorporating it (if the solution exists, i.e., the problem is feasible) in this CP model. The whole functionality is wrapped by a single method called *solve* that takes no input parameters (as all relative parameters are given as input to this sub-component/class constructor) and returns as a result a boolean parameter indicating whether an optimal solution has been produced or not (i.e. if the problem is infeasible).
- CP Solver Executor: this sub-component is responsible for manipulating the CP Solver component (i.e., utilise it to compute the optimal solution) and notify back the (successful or unsuccessful) result produced. This whole functionality is encapsulated in the form of one method called *generateCPSolution*. This method does not return any result and takes as input the parameters relevant for the CP model solving (such as the application ID, the path to the CDO Models Repository where the CP model is situated and the callback URI (part) for the notification).
- REST Controller: this sub-component encapsulates the solving process interface (comprising one core method called *applySolution*) in the form of a REST API. The sole method realised does not return any result (apart from the status of the REST call) and takes as input a ConstraintProblemSolutionRequestImpl object. This object encapsulates contextual information from the call that is then applied over the sole CP Solver Executor sub-component's method in the form of its input parameters (see description in the previous bullet).

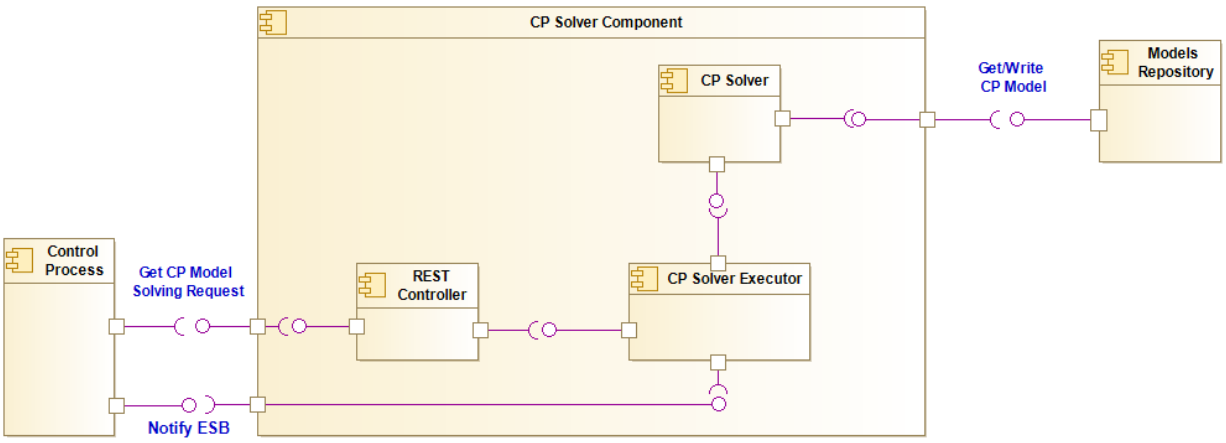

Figure 6. CP Solver in Upperware Component diagram

In Figure 7 we present the way the CP Solver interacts with other Upperware components and the respective interfaces involved. In particular, this component exchanges the following messages which are grouped based on the respective Upperware component the corresponding interaction involves:

- with the Control Process:
  - **CP Solution Request:** this message conveys the CP model solving request that (indirectly) comes from the Metasolver. In this request, the actual path to the (CDO) Models Repository where the CP model to be solved resides is given.
  - **CP Solution Notification:** this is an output message that is sent back to the Control Process in order to notify it about the completion of the CP model solving process. This message incorporates the actual result of the solving (i.e., where it has been successful or not).
- with the (CDO) Models Repository:
  - **Get CP Model:** this actually reflects the interaction between the CP Solver and the Models Repository for the fetching of the CP model to be solved. In the context of this interaction, a transaction is opened and then the actual CP model is retrieved based on its path in the Models Repository. This interaction is facilitated through the use of the CDO Client component.
  - **Write CP Model:** once the CP model is solved, the respective solution is written in it and the CP model is written back to the Models Repository. Again, this model writing is facilitated through the use of the CDO Client.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

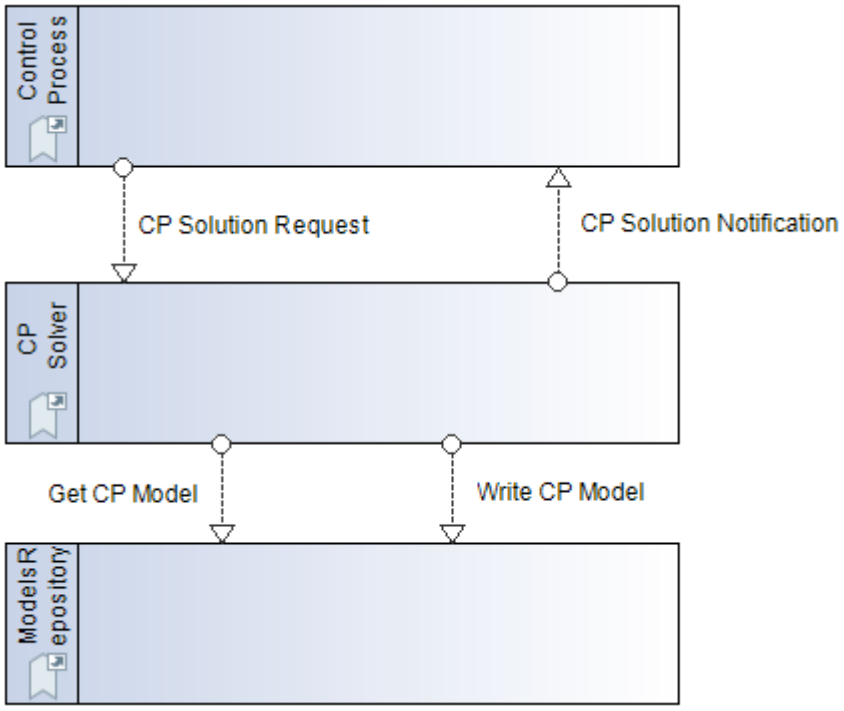

Figure 7. CP Solver collaboration diagram

4.2.2 Implementation

The CP Solver has been implemented in Java as a Spring-boot application. It can be built and bundled via Maven in a form of either a fat JAR file or a Docker image. The latter form facilitates its integration into the Melodic's platform swarm. Internally, the CP Solver exploits the Choco Constraint Programming solving engine integrated with the Ibex CP solver as well as the CDOClient in order to retrieve a CP model for solving, plus writing back to it, the respective solution found. In addition, the Utility Generator is utilised for computing the utility of candidate solutions (in particular, its encompassed standalone client to facilitate a fast interaction between the CP Solver and the Utility Generator).

The CP Solver's source code is available in Melodic's bitbucket at:
https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/cp_solver?at=RC2.5

Dependencies: CDOClient, Utility Generator, Spring-boot framework, Choco solver

4.2.3 Configuration

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

The configuration of the CP Solver component can be specified via a properties file (called as eu.melodic.upperware.cpSolver.properties) which comprises the following information pieces:

- ESB URL: the callback URL to the upperware control process that is called when the CP Solver has finished its execution and produced a respective optimal (CP model) solution.
- LOGGING CONFIG: path in the file system where the logging configuration file is situated

A sample properties file for this component is given below:

```
#
# Copyright (C) 2017 7bulls.com
#
# This Source Code Form is subject to the terms of the
# Mozilla Public License, v. 2.0. If a copy of the MPL
# was not distributed with this file, You can obtain one at
# http://mozilla.org/MPL/2.0/.
#

#### Communication with ESB ####
esb.url=http://localhost:8088/api/cpSolver

#### REST interface port ####
server.port = 8093
### logback configuration ###
logging.config=file:${MELODIC_CONFIG_DIR}/logback-conf/logback-spring.xml
```

4.3 Overview of changes towards CP Solver Finalization

Due to code instability with respect to previous version of Choco Solver[3] and especially its cooperation with the Ibex CP solver in the context of solving CP models that contain also real-valued variables, the CP Solver has been refactored in order to conform to the latest version of Choco Solver (4.10.0) and its new API.

[3] http://www.choco-solver.org/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

5 Utility Generator

Mission: Calculating the utility function value for a configuration proposed by an Upperware solver.

Positioning in Melodic: The Utility Generator is a library responsible for evaluating each solution found by Upperware solvers of the Melodic platform.

5.1 Approach

High-level Approach: The Utility Generator class is instantiated for each reasoning separately. The Utility Generator exposes the method to evaluate the proposed solution. This is performed through the notion of Utility Function that expresses the suitability of alternative solutions to a certain Constraint Problem.

Functionalities:

Calculating the utility function value for each solution/configuration proposed by a solver

Input:

- Constraint Problem model
- CAMEL model
- Current measurements (metric values)
- Cache with Node Candidates

Output:

- Utility function value

5.2 Technical Implementation

5.2.1 Architecture

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

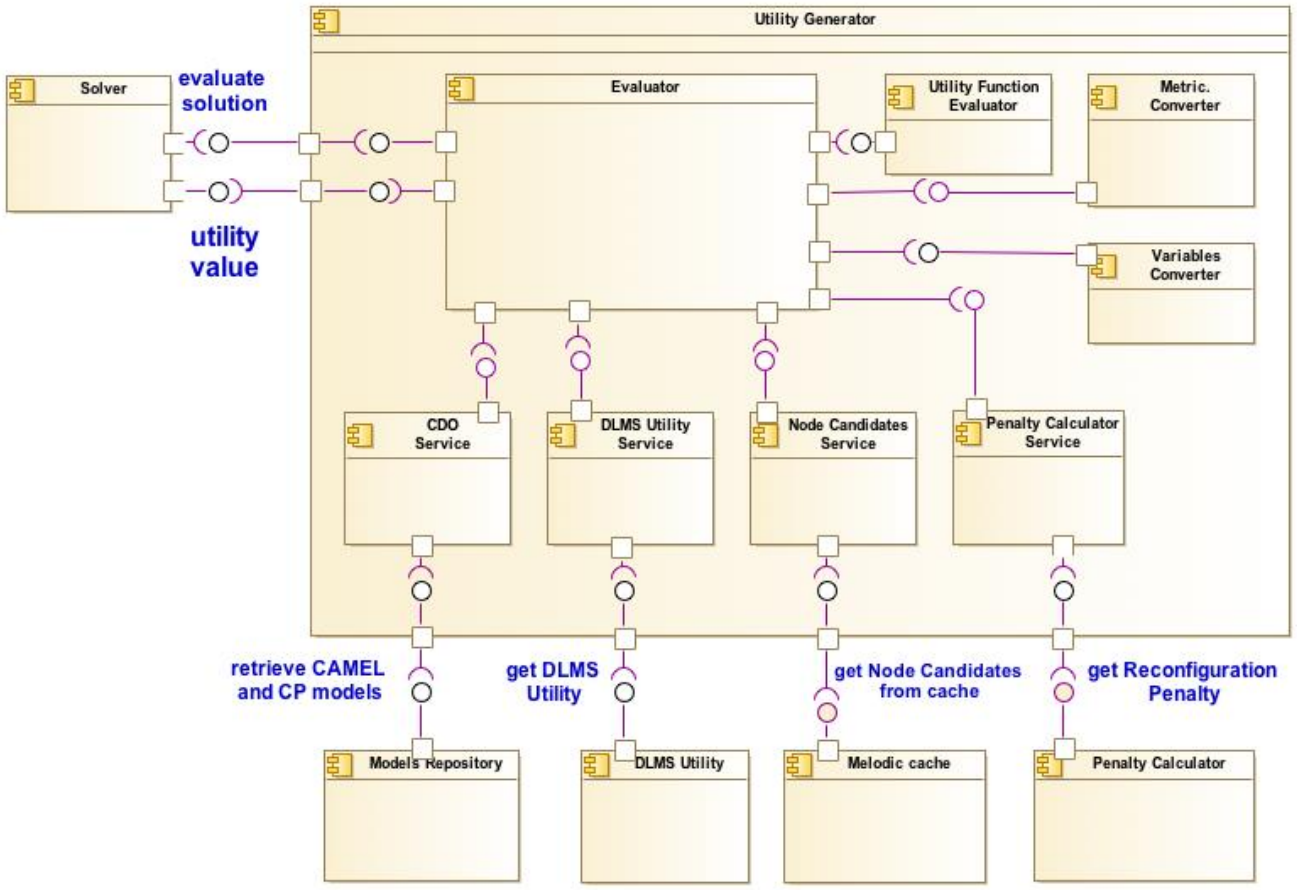

Figure 8. Utility Generator in Upperware Component diagram

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

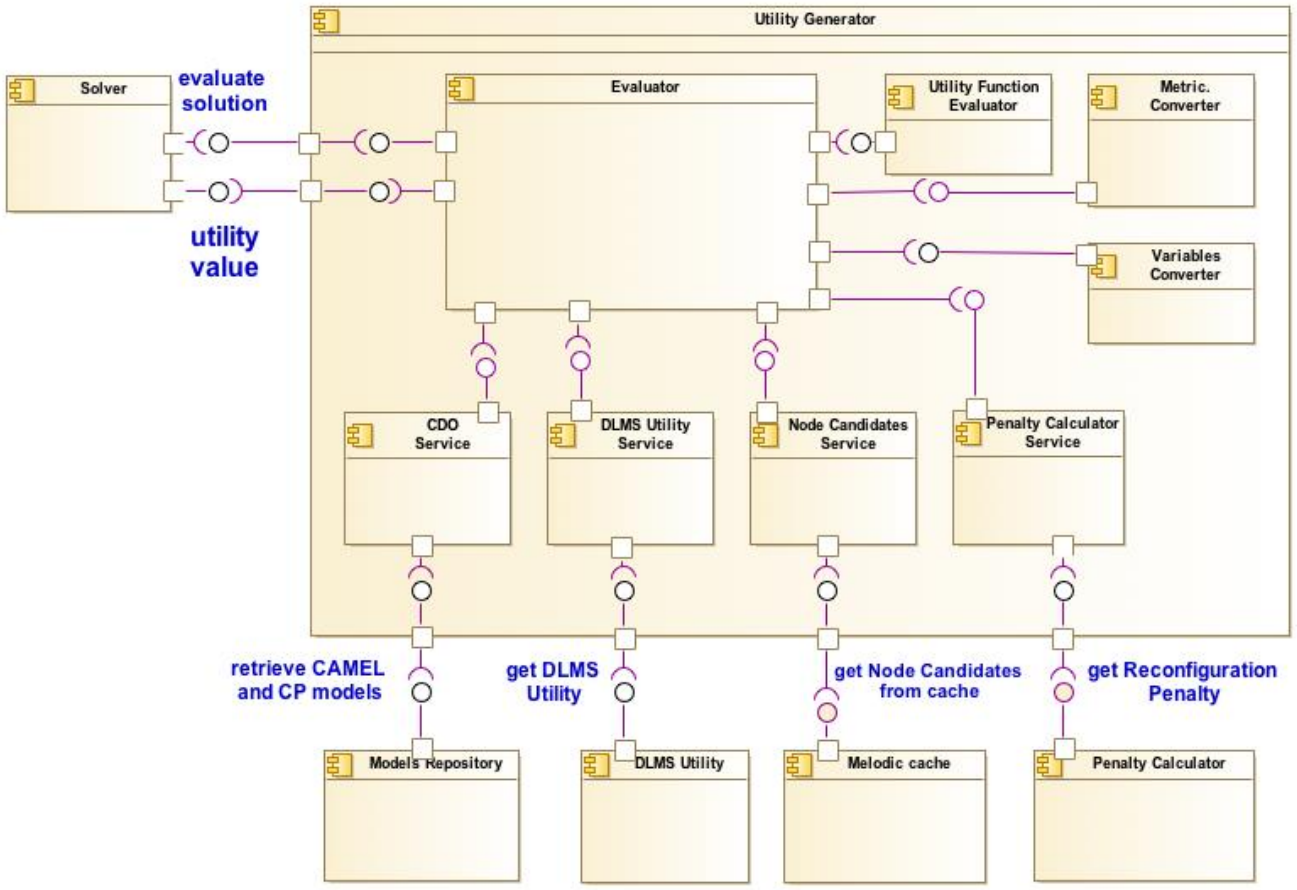

Figure 8, the architecture of the Utility Generator is depicted which comprise the following main components:

- Evaluator: the main module of the Utility Generator library, responsible for getting Node Candidates, collecting arguments and invoking the Utility Function Evaluator to get the utility function value
- Utility Function Evaluator: the module, which uses the MathParser[4] library, responsible for calculating the utility function formula
- CDO Service: the module responsible for retrieving CAMEL and CP models from the Models repository (CDO)
- DLMS Utility Service: the module responsible for calling the DLMSUtility library to get the DLMS utility and converting the DLMS utility object to the arguments needed by the Utility Function Evaluator module
- Penalty Calculator Service: the module responsible for calling the Penalty Calculator library to get the time-based penalty of the reconfiguration. Specifically, it sends XMI files describing the collections of configuration elements for the current and the new

[4] http://mathparser.org/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

proposed configuration and receives a normalized penalty value between 0 and 1 as a result.

- **Metrics Converter:** the module responsible for converting metrics to the arguments needed by the Utility Function Evaluator
- **Variables Converter:** the module responsible for converting solution variables values to the arguments needed by the Utility Function Evaluator
- **Node Candidates Service:** the module responsible for converting from the configuration with Node Candidates attributes used in the utility function formula to the arguments needed by the Utility Function Evaluator

In the following UML collaboration diagram, we describe the flow of information between the Utility Generator and the rest of the Upperware components with which it interacts. Specifically, the following messages are exchanged:

- with the Solver
  - **Create:** Message that contains information with the id of the CAMEL Model, the CP Model
  - **Evaluate solution**: Message with the solution that should be evaluated
  - **Utility value**: The message with the utility value of the evaluated solution
- with the Melodic cache:
  - **Get Node Candidates**: Message with the requirements from the solution to fetch the possible Node Candidates
- with the Penalty Calculator
  - **Get the Reconfiguration Penalty**: the message that contains the current configuration and the proposed configuration to get the penalty of the reconfiguration
- with the DLMS Utility
  - **Get DLMS Utility**: the message that contains the current configuration and the proposed configuration to get the values of the DLMS utility
- with the Models Repository
  - **Retrieve CAMEL Model:** This message asks from the Models Repository to fetch the CAMEL model.
  - **Retrieve CP Model:** This message asks from the Models Repository to fetch the CP model.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

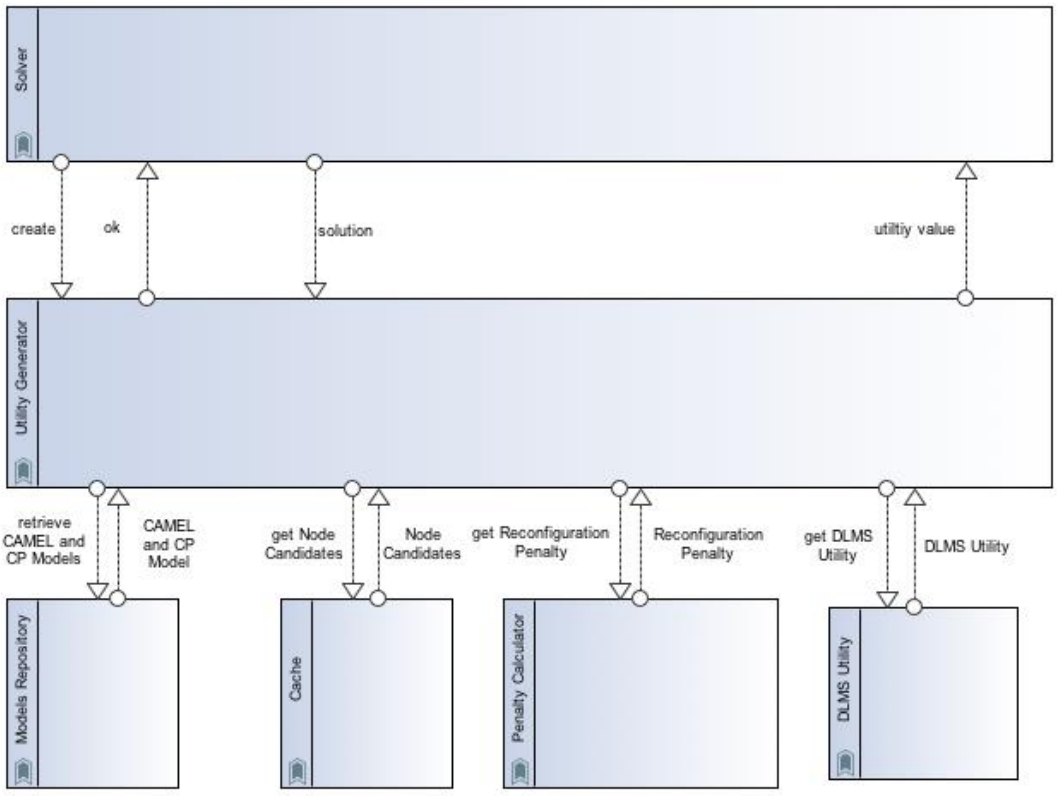

Figure 9. Utility Generator Collaboration diagram

The Penalty Calculator (Figure 9) is a part of the Upperware of the Melodic platform and is used as a library by the Utility Generator. Since this library is important for the Utility Generator, we provide here further details. Penalty Calculator's objective is to calculate a normalized reconfiguration penalty value by comparing the current and the new candidate configuration, coming from a Solver component of the Upperware. A constraint programming solver is used to generate a sequence of candidate configurations under specific constraints and optimization goals (e.g. reduce cost and increase service response time) that will serve according to the desired QoS of the incoming workload. The Penalty Calculator affects the decision on accepting and deploying a new candidate cross-cloud application topology based on its' function value. The smaller the output value of the penalty function is, the more preferable the candidate solution is as it implies a less time for implementing the proposed reconfiguration. Specifically, the Penalty Calculator receives from the Utility Generator, the collections of configuration elements for the current and the new proposed configuration (including OS, hardware and location related information of the virtualised resources to host certain application components). Based on the feed from the Utility Generator in Melodic, the Penalty Calculator Algorithm is applied for comparing the old and the new proposed (candidate) solution, issuing a penalty value, thus affecting the decision on whether or not a specific new solution should be deployed. The Penalty Calculator Algorithm uses measured VM startup times and measured component deployment times (their average values) for

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

calculating the time-related cost for changing from the current to a new application topology. It is important to note that the component deployment times are constantly measured per VM type and their latest values are considered during the time penalty calculation. If there are no component deployment times from past measurements (i.e. initial deployment and no historical records available), the algorithm takes into consideration only the VM startup times. In case of new custom VMs that are to be provisioned, the Ordinary Least Squares Regression Algorithm is used by the Penalty Calculator to estimate the expected startup time by exploiting the measurements of the available predefined cloud providers' flavours.

As a final outcome result, the Penalty Calculator provides normalized output values between 0 and 1 (by using the min-max normalization method), where 0 indicates the lowest possible penalty (i.e. the most desired solution) and 1 indicates the highest possible penalty which is the less desired solution from the point of view of the time penalty calculation.

5.2.2 Implementation

The Utility Generator has been implemented using the Java programming language, version 8. It has been developed as a Java library.

The Utility Generator's source code is available in Bitbucket at:
https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/utility-generator

Dependencies: CDO Client, Melodic cache, Cloudiator Client, DLMS Utility, MathParser, Melodic commons, Penalty Calculator.

We note that the Penalty Calculator, which is a dependency for the Utility Generator, it has been implemented using the Java programming language, version 8. It has been developed as a Java library. The **Penalty Calculator code** is available at Bitbucket at:

https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/penalty-calculator

5.2.3 Utility Generator configuration

The Utility Generator configuration is stored in `eu.melodic.upperware.utilityGenerator.properties` file, using Java properties format. It contains only one property: the url of the DLMS Controller.

Next, we provide a sample Utility Generator configuration file:

```
#
# This Source Code Form is subject to the terms of the
# Mozilla Public License, v. 2.0. If a copy of the MPL
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

```
# was not distributed with this file, You can obtain one at
# http://mozilla.org/MPL/2.0/.
#
UtililityGenerator.dlmsControllerUrl = DLMS_CONTROLER_IP
```

We note that the Utility Generator internally uses a CDO client to communicate with the Upperware Models Repository, in order to retrieve and modify the application CP and CAMEL model. The CDO client reads its configuration from `eu.paasage.mddb.cdo.client.properties` file. It also passes the Security Properties to the DLMS Client. Therefore, it needs `eu.melodic.upperware.security.properties`.

5.3 Overview of changes towards Utility Generator Finalization

The Utility Generator has not been changed significantly over the last period, with the exception of the integration with the Penalty Calculator. This Penalty Calculator is used as a library by the Utility Generator for providing a normalized penalty value belonging to the range [0..1], concerning a reconfiguration. The VM predefined start-up times from various Cloud Operators are considered along with the possibility to cope with custom VMs for which their startup times have not been measure before. Last, the Component Deployment Times can also be considered when calculating a time-based penalty value that affects the decision of the reconfiguration.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

6 Adapter

Mission: Create a target application configuration to be deployed into cross-cloud resources.

Positioning in Melodic: Adapter is one of the microservices comprising the Upperware of the Melodic platform.

6.1 Approach

High-level Approach: Adapter is responsible for preparing a complete plan of application reconfiguration for an accepted (by the Metasolver) new deployment. The plan includes a series of tasks to be sent for execution to the Cloudiator, following a specific order. To fulfil this requirement, a graph structure is maintained internally by the Adapter to reflect the application structure along with the dependencies among tasks.

Functionalities:

- Create Deployment Instance Model from the Constraint Problem solution for deployment process
- Analyse and verify the new CAMEL Deployment Instance Model
- Compute the difference between a currently running application (topology) and the new proposed solution given by the solver
- Producing the reconfiguration plan
- Validate the plan
- Apply the plan to the running system by calling the Cloudiator REST API

Input:

- CAMEL Deployment Model
- Constraint Problem solution

Output:

- Series of action tasks instructions for execution by the Cloudiator in correct and efficient order
- Notification of successful/unsuccessful deployment for the control layer

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

6.2 Technical Implementation

6.2.1 Architecture

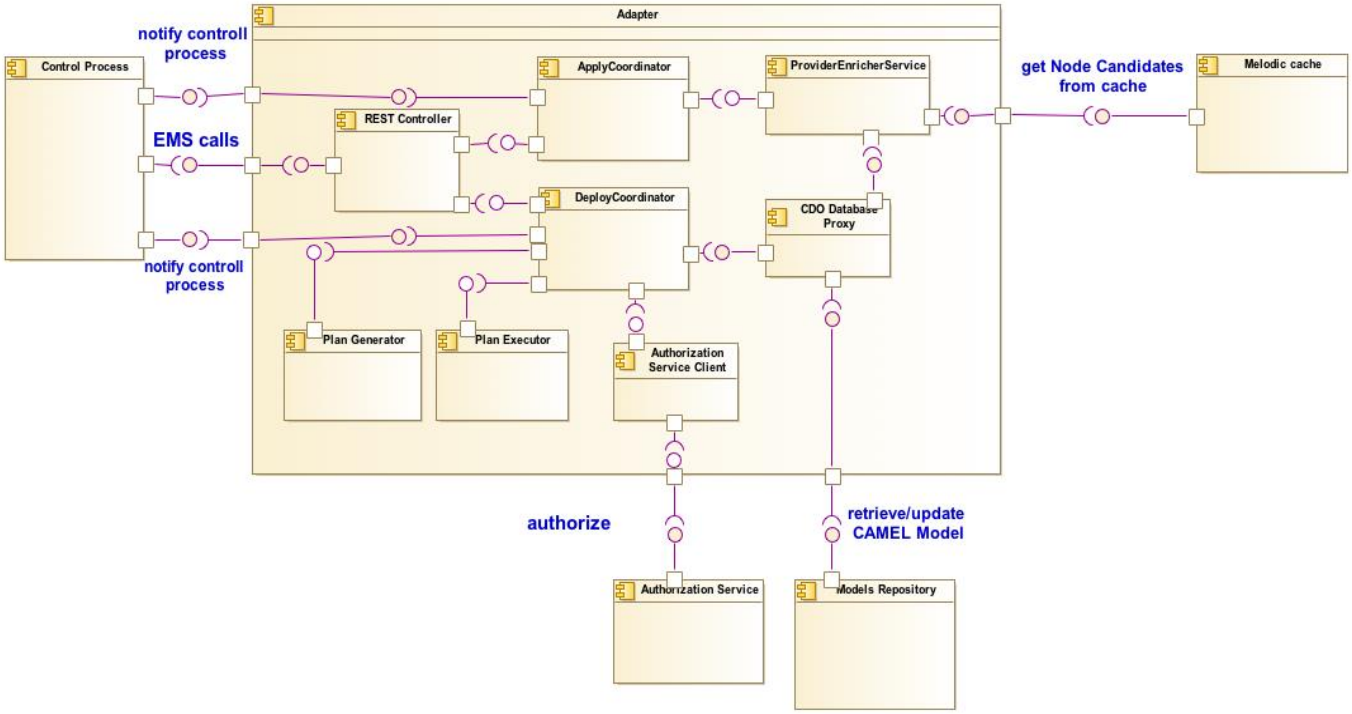

Figure 10. Adapter in the Upperware Component diagram

In Figure 10, the architecture of the Adapter is depicted which comprise the following main components:

- **REST Controller:** part of the application responsible for exposing REST endpoints
- **DeployCoordinator:** main sub-component responsible for coordinating all the actions during the (re)configuration process
- **Apply Coordinator**: sub-component responsible for coordinating creating the Deployment Instance Model (the functionality of the old Solver to Deployment). It invokes the Provider Enricher Service to get the information about Node Candidates. Then for each application's component, it computes the data concerning the Deployment Instance Model such as information about: communications, software components, and geographical regions
- **Plan Generator:** part of the application responsible for generating the (re)configuration deployment plan, where deployment plan is a set of tasks which must be executed by Cloudiator to finish a (re)configuration. The deployment plan has the form of a graph. It is directed graph, where nodes represent tasks and edges – dependencies between

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

these tasks (e.g. one task must be finished before the start of another one). Thanks to the usage of directed graph, the right execution order of tasks is known. In a deployment plan the following types of tasks exist:

- Schedule Task – create Schedule (type used by Cloudiator for management of group of processes),
- Process Task – create or delete Process (type used by Cloudiator - it represents the description of an operation to be performed on a particular node, e.g. installation of a required tool),
- Check Finish Task – check status of task,
- Scale Task – create or delete Scale (type used by Cloudiator – it refers to the triggering action of adding new or additional nodes or even deleting some of them),
- Node Task – create or delete Node (type used by Cloudiator - it corresponds to the created instance.),
- Monitor Task – create or delete Monitor (type used by Cloudiator - monitors are used for collecting metrics),
- Job Task – create Job (type used by Cloudiator - it corresponds to the element that should be deployed),
- Wait Task – create or delete waiting process.

The Plan Generator can generate a deployment plan in two ways, which depend on the process type: i) generate configuration plan (in case of configuration process) or ii) generate reconfiguration plan (for reconfiguration process). In both cases this is created based on the Deployment Instance Model, built previously by the *Apply Coordinator*. Plan Generator compares the two Deployment Instance Models – the current process with the latest already deployed one (in case of configuration the second one is empty, i.e. initial deployment). Based on this comparison, it decides which elements should be created, which should remain unchanged and which should be deleted.

- Plan Executor: part of the application responsible for invoking the generated deployment plan. It goes through the deployment plan (directed graph) and submits tasks to be executed (by Cloudiator). Each type of tasks from the deployment graph (described above) has its own executor (e.g. Schedule Task is performed by Schedule Task Executor, Process Task by Process Task Executor etc.). Each executor sends requests to Cloudiator and monitors any responses from it. For communicating with Cloudiator a dedicated Cloudiator Client is used. Some of the requests sent to Cloudiator are asynchronous (e.g. VM creates that takes some time), so the response is a returned queue item with the task status that must be monitored. For this reason, these executors exploit a mechanism implemented in the Adapter for queue monitoring purposes.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

- **Provider Enricher Service:** module, which enriches information about application's components (i.e. Software Components) by adding data from Node Candidates
- **CDO Database Proxy:** part of application responsible for the communication with CDO Database
- **Authorization Service Client:** client for external Authorization Server which should be invoked to check privileges to (re)configure deployable application according to defined security policies.

In the following UML collaboration diagram, we describe the flow of information between the Adapter and the rest of the Upperware components with which it interacts. Specifically, the following messages are exchanged:

- with the Control Process:
  - **Apply Solution**: Message that contains information with the id of the CAMEL Model and CP Model that should be transformed into the Deployment Instance Model
  - **Deploy Solution**: Message that contains information with the id of the CAMEL Model that should be deployed
  - **Notification Message**: Message that contains information about the status of the task that the Adapter should done
- with the Melodic cache:
  - **Get Node Candidates**: Message with the requirements from the solution to fetch the possible Node Candidates
- with the Models Repository
  - **Retrieve CAMEL Model:** This message asks the Models Repository to fetch the CAMEL model.
  - **Update CAMEL Model:** This message asks the Models Repository to update the CAMEL Model
- with the Authorization Service
  - **Authorize**: This message contains the information about the configuration that will be deployed to get the authorization

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

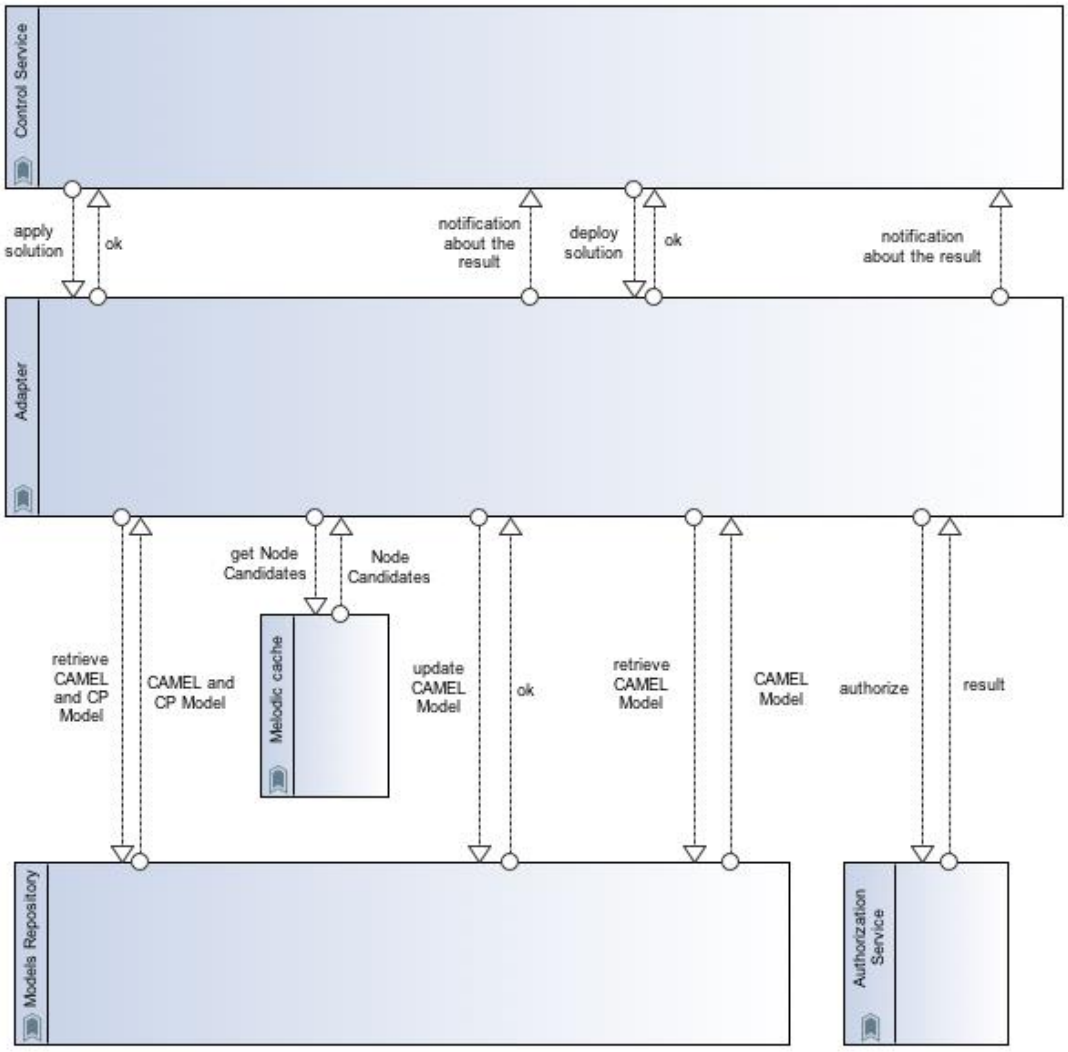

Figure 11. Adapter Component collaboration diagram

6.2.2 Implementation

The Adapter component has been developed in Java, version 8. The project is built by Maven and thanks to the Maven plugin[5], a Docker image is created which allows to run this component as a separate microservice.

Adapter's source code is available in Bitbucket at:
https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/adapter

[5] `com.spotify:docker-maven-plugin`

Dependencies: Spring-boot framework, CDO Client, Memcache, Cloudiator Client, Melodic commons, JWT commons, Authorization Service Client

6.2.3 Adapter Configuration

The Adapter configuration is stored in `eu.melodic.upperware.adapter.properties` file, using Java properties format. The most important aspects of the required configuration are the following:

- ESB URL: the endpoint of Upperware Control process, which Adapter invokes to signal that application reconfiguration is required.
- EMS URL: the endpoint of EMS, which Adapter invokes to collect information of sensors to be deployed.
- EMS ENABLED: flag that indicates if calls to the EMS will be made
- Task Executor core pool size - initial thread pool size for concurrent tasks execution
- Task Executor max pool size - maximum thread pool size for concurrent tasks execution
- Task Executor queue capacity - max number of tasks in queue
- Logging config: path to log configuration file
- Cloudiator URL: the endpoint of Cloudiator Service
- Cloudiator API key: authorization key for REST API of Cloudiator
- Cloudiator HTTP Read Timeout: HTTP read timeout in ms
- Cloudiator delay between queue check: time to pass before getting the status of executed task in ms

Next, we provide a sample Adapter configuration file:

```
#### Communication with ESB ####


esb.url=http://MELODIC_IP:8088

ems.enabled=true

ems.url=http://MELODIC_IP:8099/monitors


 #### Spring thread pool task executor config (used for concurrent
calling tasks generated by Plan Generator) ####


taskExecutor.corePoolSize=16

taskExecutor.maxPoolSize=16
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

```
taskExecutor.queueCapacity=

#### Logback-config
logging.config=file:${MELODIC_CONFIG_DIR}/logback-conf/logback-spring.xml

cloudiatorV2.url=http://MELODIC_IP:8089/api/adapter/cloudiatorProxy/api/v2
cloudiatorV2.apiKey=XXX

### timeout in milliseconds
cloudiatorV2.httpReadTimeout=300000
cloudiatorV2.delayBetweenQueueCheck=5000
```

Configuration of used components

Adapter uses `eu.melodic.upperware.cache.properties` to connect with Memcache in order to load Node Candidates from that. The most significant elements are the following:

- Number of load attempts: on the off-chance that some errors occur during this connection, we provide more than one attempt of connection and loading data;
- Time between loading attempts: time in seconds between attempts of connection and loading data

Sample Memcache configuration file:

```
cache.host=melodic-memcache
cache.port=11211
cache.ttl=86400
cache.timeBetweenLoadAttempts=2
cache.numberOfLoadAttempts=3
```

We note that the Adapter internally uses a CDO client to communicate with the Upperware Models Repository, in order to retrieve the Camel Model and create Deployment Instance Model. The CDO client reads its configuration from eu.paasage.mddb.cdo.client.properties file.

6.3 Overview of changes towards Adapter Finalization

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

The most important change is the fact that the Adapter has been merged with the Solver to Deployment. Now the module of the Adapter that is responsible for the Solver to Deployment job is called Apply Coordinator. The main reasons for the merge were:

a) reducing the resource usage of the Melodic platform;
b) to have single point of adapting responsibility

The first point is very straightforward - having one microservice instead of two separate ones simply reduces the usage of the memory and CPU for the Melodic instance. The second reason means that after the change, all features required to properly transform the Constraint Problem Solution (Reasoning Domain) to a Deployment Model (Execution Domain) are being encapsulated within a single component. This allows for more efficient development of new software features to the platform.

Another change is connected with the support of scale in feature in the Cloudiator. It allows for easy scaling of Spark components. The Adapter module can now support the BYON deployment. It also uses a JWT-based authentication in its interactions with other Upperware components.

.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

7 Event Management System

Event Management System (EMS) is a distributed application monitoring system, which is used by Melodic Upperware for monitoring the operation of the cross-cloud application it deploys, in order to take appropriate actions when certain constraints are violated; i.e. to reconfigure the application.

Mission: Collect, process and deliver to interested parties, monitoring information pertaining to a distributed, cross-cloud application, according to CAMEL model specifications.

Positioning in Melodic: The EMS server, called Event Processing Manager (EPM), is one of the microservices comprising the Upperware of the Melodic platform. EMS clients, called Event Processing Agents (EPAs), reside inside each VM that hosts a cross-cloud application component. The EMS server exposes REST APIs and endpoints, used while interacting with EMS clients or other Upperware components.

7.1 Approach

High-level Approach: Deploy a network of agents for collecting monitoring information from the sensors (i.e. monitoring probes) as events, process them using distributed event processing techniques, and forward results to the interested parties (e.g. Metasolver). A CAMEL model specifies the needed monitoring information and the kind of processing required.

Functionalities:

- Acquire and analyse the application CAMEL model. It yields an abstract form of CAMEL model parts, relating to monitoring, capturing and processing information, as well as other auxiliary structures. This abstract form is a multi-root Directed Acyclic Graph (DAG).
- Generate complex event processing rules for EPAs and EPM. Rule generation is based on the traversal of the DAG resulted from the CAMEL model analysis.
- Deploy and manage the network of EPAs. Deployment is actually carried out by Cloudiator, which runs an EPA-specific installation process. Upon activation, EPAs connect to the EPM node (specifically to the network orchestration module of EPM) and receive their configurations that encompass the event types to be collected, the event processing rules to be enforced and the event types (raw or generated) to be propagated to another EPA or to the EPM node.
- Event brokering. Each EMS node (EPM node or EPAs) encompasses an event broker. Locally captured or generated events, as well as events forwarded from other nodes,

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

are published there. Events are organized in topics according to their source or type. Some of these topics can be configured to forward their events to other EMS nodes. Typically, EPA brokers are private (internal to EPA). However, the EPM node broker can be accessed (for consuming events) by any interested Melodic platform component.

- Complex Event Processing (CEP). Each EMS node (EPM or EPA) encompasses a CEP engine, which receives events as they arrive to the local event broker, applies the configured event processing rules and publishes the generated events back to the local event broker. Typically, event processing that occurs at cross-cloud application nodes, refers to filtering and aggregating local events (i.e. those collected by local sensors) and pertain solely to that application node. Event processing at the EPM node typically aggregates and filters events from all application nodes. Some EPAs can be designated as intermediary event processors (filters or aggregators), hence resulting in a multitier distributed event processing hierarchy.

Input:

- Raw monitoring information from sensors
- Distributed application topology (i.e. deployment model)
- CAMEL model of a cross-cloud application.

Output:

- Monitoring information as simple or complex events
- Configurations for other Upperware components (i.e. Metasolver).

7.1.1 Event Processing Network

As already mentioned, EMS is a distributed application monitoring system that comprises of a server integrated in Melodic Upperware, named *Event Processing Manager* (EPM), and several clients, named *Event Processing Agents* (EPAs). EPM and EPAs formulate a network of nodes for distributed event processing, called Event Processing Network (EPN). This network is orchestrated and controlled by EPM.

There are several operations involved in event processing. Some of the most common are:

- Event collection (from monitoring probes)
- Event filtering (selecting events that meet certain conditions)
- Complex event generation (creating new events based on the values of other events)
- Event aggregation (creating complex events by aggregating the values of other events)
- Event pattern detection (finding predefined motives of event occurrences)
- Event propagation/delivery (sending events to other processing nodes or

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

destinations).

Furthermore, these operations might be applied in different fashions and scopes. For instance, they might be applied per application node/VM, per cloud provider or for the whole application. In Melodic we have devised a hierarchical model of scopes, where event processing can occur locally (i.e. in each application VM), across the application nodes deployed in a single cloud provider, or across the whole application (cross-clouds). The event processing results of one scope are propagated to the higher-level scope.

For example, RAM usage events can be collected per application VM and averaged per minute. Average value events are then propagated to the cloud level. In cloud scope, all average RAM usage events from the VMs deployed in the same cloud provider are collected and filtered. Events might be checked, if they exceed a specified threshold. Such events might then be propagated to the Application level (i.e. global) in order to signal that a certain condition occurred (i.e. a node has been overloaded).

In Melodic, we have defined the following scopes, which we call *groupings*:

- Per Instance: processing involves events from a single application instance executed in a VM.
- Per Host: processing involves events originating from the same local VM. It may also aggregate events generated and propagated from the Per Instance grouping.
- Per Zone: processing involves events originating from VMs in the same cloud availability zone.
- Per Region; processing involves events originating from the same cloud region.
- Per Cloud; processing events originating from all application nodes deployed in the same cloud provider.
- Global; processing events originating from any application nodes. Events might be received from any subordinate grouping. This is the terminal grouping.

For each grouping, different event processing operations might be required. These can be expressed as sets of event processing rules and event propagation flows between groupings.

An important aspect of the hierarchical grouping model of Melodic, is that it requires the event processing to occur as close to the event generation point as possible. This statement means that the scope processing must occur in the same VM/location where input events are created. For example, in per host grouping, event processing must take place locally, in each VM. In per cloud grouping, event processing must take place in application VMs designated for this purpose, per cloud provider. The global grouping event processing occurs in the EPM node (usually hosted with the rest Melodic components).

To implement this approach, an EPA must be deployed in each application VM. EPM configures the EPAs into a cooperating network of event processing nodes, providing all necessary event processing rules and event propagation routes.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

This approach is based on the assumption that large numbers of events can be exchanged very fast and much cheaper (network-wise) between VMs running on the same availability zones, regions or cloud infrastructures. Processing events inside the same zone/region/cloud can reduce the number of events that need to be propagated to a central processing node thus reduce the overall network throughput for application management purposes. Moreover, the computational load of the event processing is distributed to all application VMs, hence the central processing node does not require increased computational capabilities or network resources. Of course, if an application requires centralized processing it is possible to reduce the number of groupings into per host and global, thus resulting in events flowing from VMs to EPM.

In the context of EMS, the EPM acts as the global event processing node, whereas EPAs are the per cloud/region/zone/host nodes. EPAs are also responsible for collecting events from local sensors (i.e. installed in the same VM), perform the operations required for the grouping they have been designated to and propagate the results to another EPA or the EPM, which acts as the higher grouping processing node.

Between EPN nodes, two types of connections are established: (i) Control connections, created between each EPA and the EPM, and (ii) event propagation connections, created either among EPAs or between EPAs and EPM. The former connection type is used by EPM to control and configure EPAs, as well as by EPAs to announce their presence to EPM (during VM launching). The latter connection type is used to convey events between the grouping processing nodes.

7.2 Technical Implementation

In this subsection, all the technical details of the EMS implementation are discussed (like programming language, frameworks, third party libraries, code structure etc.). It comprises the following modules which are explained in the next sub-sections:

- **Event Processing Manager (EPM)** node or EMS server, responsible to analyse the CAMEL model, deploy and manage the whole monitoring network of EPAs, and interact with the rest of the Melodic platform (by providing interfaces and invoking interfaces of others).
- **Event Processing Agents (EPAs)**, deployed to each distributed application node. They are responsible to contact EPM node for taking their configurations (i.e. events to collect, event processing rules and where to propagate events (raw or processed)).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

7.2.1 Architecture of EPM (EMS server)

A high-level depiction of the EPM architecture is given through the UML component diagram depicted in Figure 12.

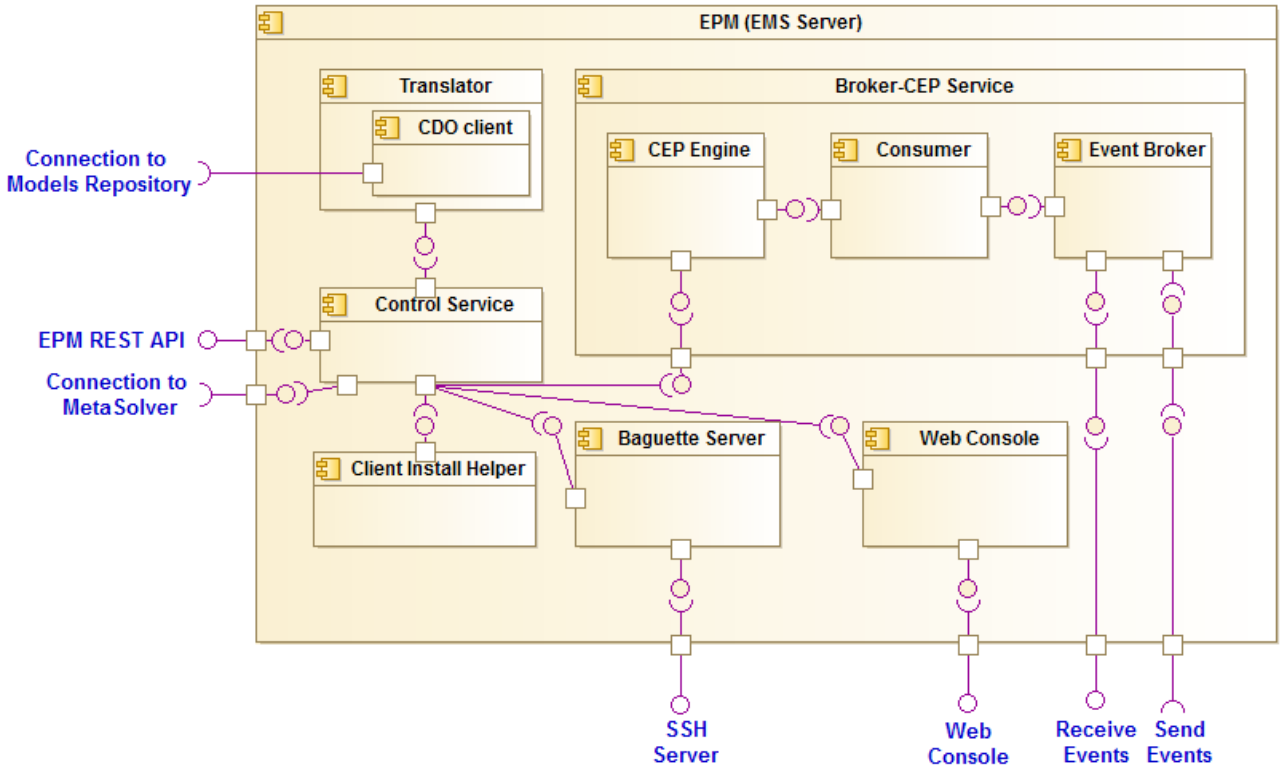

Figure 12. EPM (EMS server) Component diagram

Based on Figure 12, the EPM comprises the following sub-components:

- Translator: provides a two-step process involving the analysis of the CAMEL model to produce a multi-root Directed Acyclic Graph and also the Generation of EPL rules and other related information.
- Client install component: provides the necessary instructions on how to install an EPA to the application VM defined in the application deployment model.
- Baguette Server: is responsible for the deployment and management of the Event Processing Network. Specifically, it designates EPAs installed in each application VM, to the appropriate grouping, by sending the corresponding configuration. It also collects VM identification information sent from EPAs. The Baguette server encapsulates an SSH server used to accept incoming connections from EPAs. These connections are used to send configurations or other commands to EPAs.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

- **Control Service**: Coordinates and oversees the functioning of EPM. It also interacts with the Upperware control process through the REST API and furthermore offers a few EPM management and debugging functions (as REST endpoints as well).
- **Web console component:** Provides a dashboard for monitoring the functioning of the local event broker.
- **Broker-CEP Service:** encapsulates an event broker instance and a CEP engine instance, appropriately wired to implement the process presented in Figure 12, hence Broker-CEP provides event brokerage and complex event processing capabilities. The Consumer sub-component depicted inside Broker-CEP is used to forward the event broker messages into the CEP engine in order to be processed.

7.2.2 Architecture of EPA

A high-level depiction of the EPA architecture is given through the following UML component diagram, shown in Figure 13.

Based on Figure 13, the EPA comprises the following sub-components:

- **SSH Client:** provides secure communication with the Baguette server (of the EPM) through an EPM connection interface.
- **Executor component:** Configures and starts the Broker-CEP Service based on the instructions by the Baguette Server.
- **Broker-CEP service:** provides the local Complex Event Processing engine (i.e. Esper) along with the local Event Broker (i.e. ActiveMQ) that collects and propagates data messages to other Virtual Machines components in the distributed hierarchy of Virtual Machines in our cloud environment. This client can undertake the role of per instance, per host, per zone, per region or per cloud grouping as explained in section 7.1.1.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

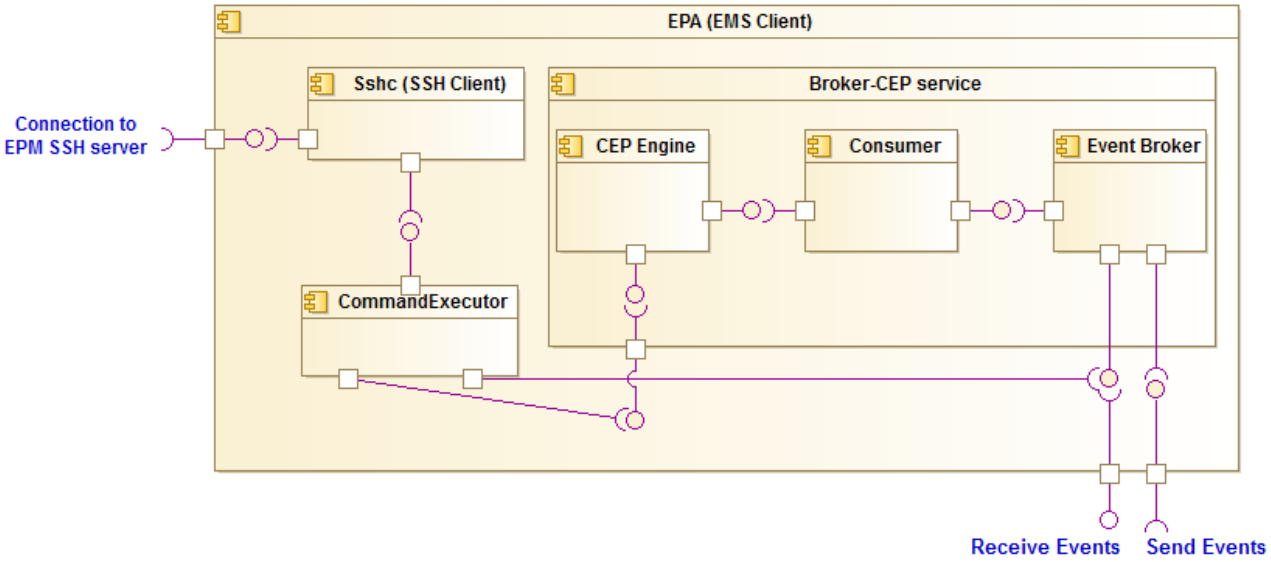

Figure 13. EPA Component diagram

7.2.3 Operation of Event Management System

A graphical representation of the EMS functioning is given in Figure 14 using BPMN 2 notation.

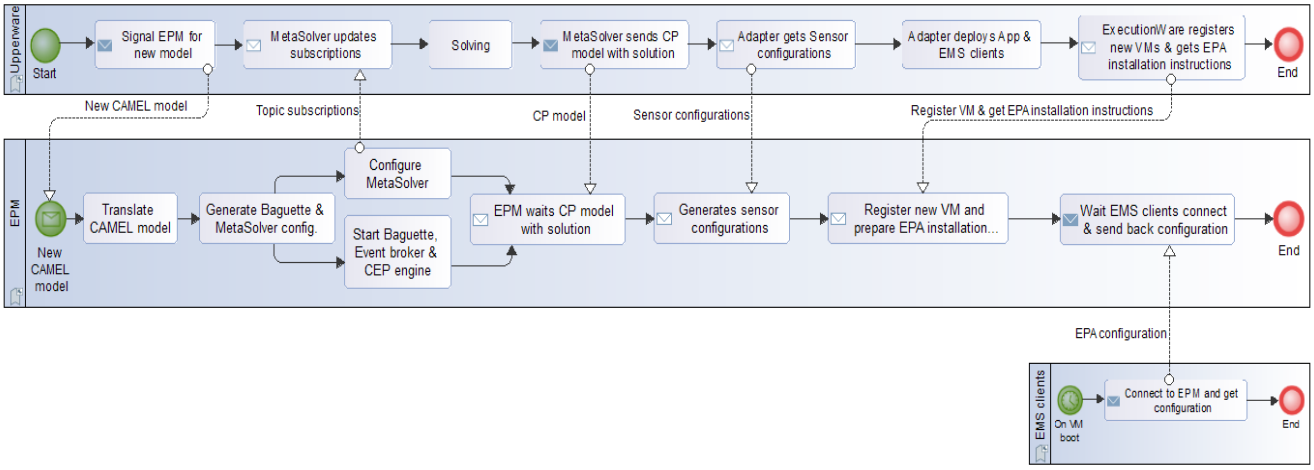

Figure 14. EMS Collaboration diagram

Based on Figure 14, the EPM (EMS server) interacts with several Upperware components, as well as with EPAs (EMS clients). Specifically:

- Process: EPM is notified by Upperware control process (Process microservice) about a new application CAMEL model. EPM will subsequently read it from Models repository and initialize itself for monitoring the application.
- MetaSolver: After initializing itself, EPM contacts MetaSolver and passes it the event

topics that it must monitor in order to receive information for updating CP model or triggering a new application reconfiguration process.

- **MetaSolver:** When a new application deployment solution is generated and successfully evaluated, MetaSolver notifies EPM about the solution parameters. EPM registers this information in order to make them available in event processing computations.
- **Adapter:** During application deployment, Adapter will query EPM for the sensors, which must be installed in application VMs. EPM extracts the sensor information from CAMEL model.
- **Cloudiator:** After creating a new application VM, Cloudiator installs application components as well as a number of Melodic VM-side services, including the EMS client (EPA). For this reason, Cloudiator contacts EPM to declare the new application node and also get relevant installation instructions.
- **EPAs:** After being installed and started, EPAs connect to EPM and receive their individualized configurations and initialize. Afterwards, they are ready to receive events for CEP processing and propagation to the next EPA.

7.2.4 Implementation

All EMS modules have been implemented using the Java programming language, version 8, and almost all of them are Spring-boot applications, components or configurations, thus making their maintenance quite predictable. EMS is delivered as a set of software packages; one for the EPM node, one for EPAs, and one for the Broker client library.

All EMS parts (EPM node, EPAs and Broker client library) are built and bundled using the well-known Maven system. Due to license incompatibilities, the third-party libraries used are not bundled with the EMS code but are kept separately. Therefore, an EMS package is a collection of JAR files (containing the EMS code and the compiled third-party libraries), configuration files and launch scripts.

The final version of EMS is available in Bitbucket at:

https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/event-management

Dependencies: Spring-boot framework, ActiveMQ library, Esper CE library, CDO client, Melodic commons, Apache MINA SSH library, Jasypt library, JGraphT library, Apache Commons libraries (lang3 and text) and Bouncy-Castle cryptography provider.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

7.2.5 EMS Configuration

EMS configuration is stored in a number of files, most of them using the Java properties format. Specifically, EMS server (the Upperware part of EMS) configuration files reside in the default MELODIC configuration location. EMS client configuration files reside in a separate directory in the default MELODIC configuration location. In the majority of installations, we expect that no changes will be required in order to have a working instance of EMS server and EMS clients.

EMS server configuration files:

- ***eu.melodic.event.baguette-client-install.properties***
  Contains settings related to the installation instructions of EMS clients, like download URL of EMS client installation package, installation directory in target VMs etc.
- ***eu.melodic.event.baguette-server.properties***
  Contains the settings of the Baguette server, i.e. the EMS subsystem responsible for sending event processing information to EMS client at runtime.
- ***eu.melodic.event.brokercep.properties***
  Contains the settings related to the Broker-CEP EMS subsystem. This information is pertaining to the event broker configuration (i.e. ActiveMQ broker) as well as the event processing engine (i.e. Esper). This is a critical file and must be altered with great care. Settings that might be meaningful to modify include:
  - *brokercep.ssl.keystore-password* and *brokercep.ssl.truststore-password* for changing the key store and trust store file default passwords
  - *brokercep.ssl.key-entry-dname* for changing the owner information of the certificates dynamically generated for EMS clients
  - *brokercep.additional-broker-credentials* for adding a priori known username / password pairs for accessing EMS server event broker (useful for integration with third-party tools)
  - *brokercep.usage.memory.jvm-heap-percentage* for altering the amount of memory reserved by event broker at start up.
- ***eu.melodic.event.brokerclient.properties***
  Contains default settings used by *brokerclient* library of EMS. This library offers an easy way to send and receive event to/from EMS event brokers and is intended for use by third-party tools.
- ***eu.melodic.event.control.properties***
  Contains information pertaining to various aspects of EMS server, including the REST API exposed for consumption to other Upperware components as well as various debug features. This is a critical file and must be altered with great care. Settings that might be meaningful to modify include:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

- `control.ssl.keystore-password`, `control.ssl.truststore-password` for changing the REST HTTPS server's key store and trust store file passwords
- `control.ssl.key-entry-dname` for changing the owner information of the HTTPS server certificate, in case `control.ssl.key-entry-generate` property is set to a value different than `NO` or `NEVER` (meaning that HTTPS certificate might be generated if needed)

- ***eu.melodic.event.translator.properties***
  Contains information pertaining to various aspects of the CAMEL model translator subsystem of EMS. Settings that might be meaningful to modify include:
  - `dag.export-to-dot.enabled` and `dag.export-to-file.enabled` that control whether the DAG created during CAMEL model translation into event processing rules, will be exported to DOT format or to an number of images
  - `dag.export.formats` and `dag.export.image-width` for specifying the image format and size of the generated DAG
- ***hostkey.ser***
  stores the Baguette server SSH private key.

EMS server also uses the following common Upperware configuration files:

- ***logback-conf/logback-spring.xml***
  Specifies the log levels (i.e. degree of detail of log records) of all Upperware components (including EMS server) and their subsystems. The settings related to EMS server are those starting with `eu.melodic.event`. All Logback log levels are applicable here.
- ***application.yml***
  Contains templates for the generation of event processing rules. This is a critical file and must not be altered.
- ***authorization-client.properties***
  Contains settings for the authorization client library used in some Upperware components (including EMS server). Settings that might be meaningful to modify include:
  - `pdp.access-key` for changing the authorization server access key (requires a respective change in `authorization-server.properties` too)
  - `pdp.http-client.keystore-password` for changing the authorization server's trust store file password
- ***authorization-truststore.p12***
  It is the trust store file used by the authorization client library.
- ***eu.melodic.upperware.security.properties***
  Contains settings pertaining to the authentication of Upperware components and validation of the JWT token.
- ***eu.paasage.mddb.cdo.client.properties***

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Contains information needed for connecting to the Upperware Models repositorty.

EMS client configuration files:

- ***baguette-client/conf/baguette-client.properties***
  Contains a template for generating EMS client configuration files. These configuration files are created dynamically when a new EMS client is installed on a new application VM and include information and credentials required by EMS client in order to connect to the EMS server.
- ***baguette-client/conf/eu.melodic.event.brokercep.properties***
  Contains the settings related to the Broker-CEP EMS subsystem of EMS client. This file is analogous to the one of EMS server.
- ***baguette-client/conf/eu.melodic.event.brokerclient.properties***
  Contains default settings used by *brokerclient* library of EMS, which is also used by EMS clients.
- ***baguette-client/conf/logback-spring.xml***
  Specifies the log levels (i.e. degree of detail of log records) of EMS client subsystems. All Logback log levels are applicable here.

7.3 Overview of changes towards EMS Finalization

The final version of EMS is essentially an update of the RC2.0 version reported in D3.4. It includes new features introduced in Melodic platform since then and provides several bug fixes and improvements. The most notable differences are:

- use of JWT-based authentication in its interactions with other platform components;
- support of encrypted (HTTPS-based) communication with other platform components;
- support of encrypted (TLS-based) communication authentication for propagating events;
- endpoint for providing installation instructions and configuration for new EMS clients (needed by Cloudiator);
- generation of EMS installation package at EMS server boot time;
- improved initialization and launch scripts (both for Docker images and standalone use);
- support of password encryption in configuration files;
- improved configuration files and new settings for more control on EMS operations;
- enhancements in CAMEL to Event Processing Rules translator (added support for Logical, Metric Variable and If-Then constraints, added support for metric variable calculation);
- generation of private key and certificate at boot time of EMS server and clients (if

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

configured); certificate exchange mechanism between EMS server and EMS clients;
- logging of incoming events and REST API requests
- Topic Beacon for publishing informational events for application monitoring UI (include statistics and limits of various constraints);
- event interceptor to set the source IP address for application monitoring UI

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

8 DLMS

In multi-cloud services, such as Melodic, intelligent strategies on data management and placement play an important role in cost-efficient and data-intensive computing. Data Life-Cycle Management System (DLMS) is a component acting as a handler for registering user input data sets into Melodic eco-system and providing a convenient way to retrieve the data. It is important to emphasize that DLMS is not designed to physically store the data.

Mission: Handle user input data and enable easy retrieval of the data from the Melodic platform.

Positioning in Melodic: DLMS provides an interface for Utility Generator and enables other application components to access the data managed through the Melodic platform.

8.1 Approach

High-Level approach: The DLMS contains methods enabling other components to read and store data to or from Melodic internals. These include reading underlaying application models, information about providers, managing (reading, writing, modifying data sources). Generally, it provides an interface to interact with data stored that can be conveniently utilized by other components. DLMS consists of DLMSController and DLMSUtility, providing various methods for managing the data and DLMSWebService which exposes endpoints for interaction with data from outside.

Functionalities:

- Management of data sources on behalf of Melodic user;
- Optimised data placement in the cloud based on user-defined data placement requirements, constraints and associated costs;
- Keeping user-defined data requirements satisfied throughout the data lifecycle;
- Assignment of data transfer and access costs associated with data sources given an application topology and its data access requirements;
- Providing an interface for the Utility Generator;
- Retrieve logical and physical configuration of underlaying nodes (e.g. IP addresses and machines names).

Because of its design and purpose, DLMS requires various inputs and produces appropriate outputs. Below, examples of input and entities are listed.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Input:

- Datasource
- Name of the datasource
- Cloud provider name
- Component IP (component of which one wants to get configuration info)

Output:

- Appropriate name
- Datasource or list of datasources
- Various underlying information, like provider parameters
- Relative configuration of components and machines they run on

8.2 Technical Implementation

8.2.1 Architecture

A high-level depiction of the DLMS architecture is presented in the UML component diagram (Figure 15).

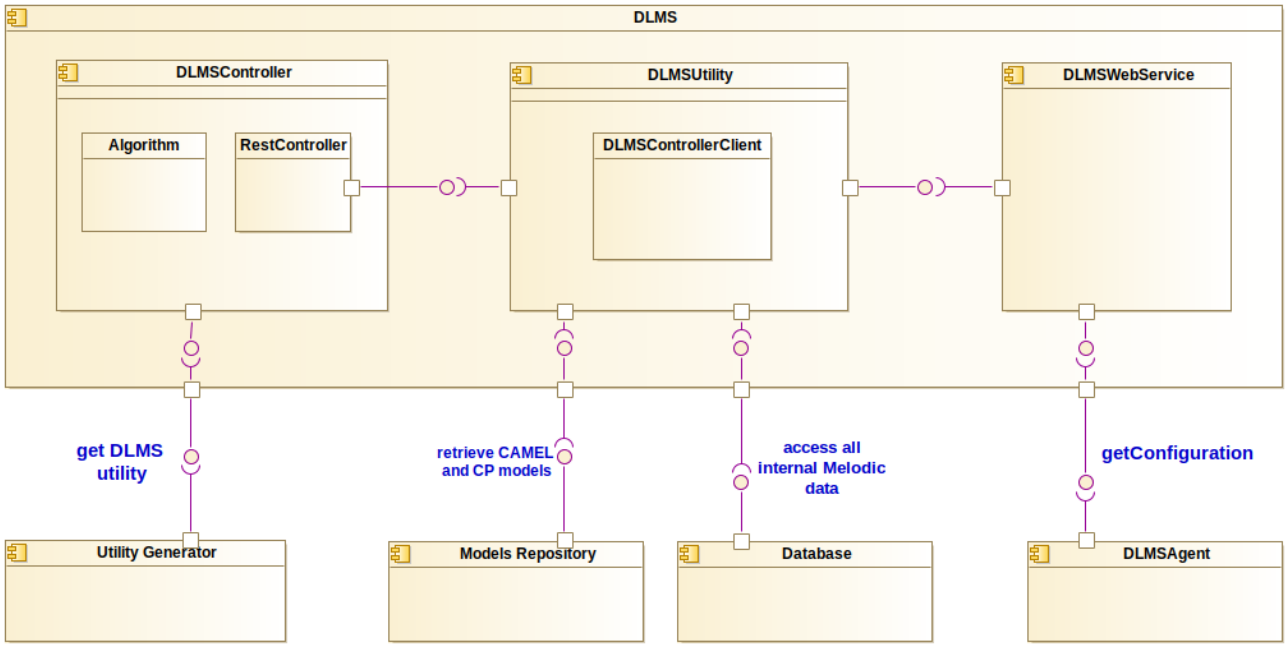

Figure 15: DLMS Component diagram

As shown in Figure 15, the DLMS contains the following sub-components:

- **DLMS Controller**: provides the essential functionality for the Utility Generator,

implements metric algorithms and manages all the information flow between them.
    - Rest Controller: provides the REST API of the DLMS Controller for receiving calls from the Utility Generator and algorithms. Enables to get all the information related to cloud providers, data canters, regions, application components and data sources which is required to calculate appropriate utility value with particular metrics.
    - Algorithm: A scheme and the implementations of the algorithms designed for calculating specific metrics. The details of these algorithms employed by DLMS have been detailed in previous MELODIC deliverable [3], [4].
- DLMS Utility: comprises such functionalities as interface for Utility Generator and implementing DLMSConfiguration employed for comparing current internal configuration to a proposed one.
    - DLMSControllerClient: exposes an API for Utility Generator to access methods implemented in DLMS Controller.
- DLMS WebService: exposes a REST API for complete management of data sources and getting a physical deployed configuration which is employed by DLMS agent.

Moreover, a separate component – DLMS Agent – was designed and implemented as an independent agent installed by Cloudiator's InstallAgent on the machine with a running application. It was designed to gather metric data and periodically (with a period specified by user) send it to JMS server. The global metric which takes into account all running components can be calculated then.

The following UML Collaboration (Figure 16) diagram describes the flow of information between DLMS sub-components and the rest of the Upperware components it interacts with.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

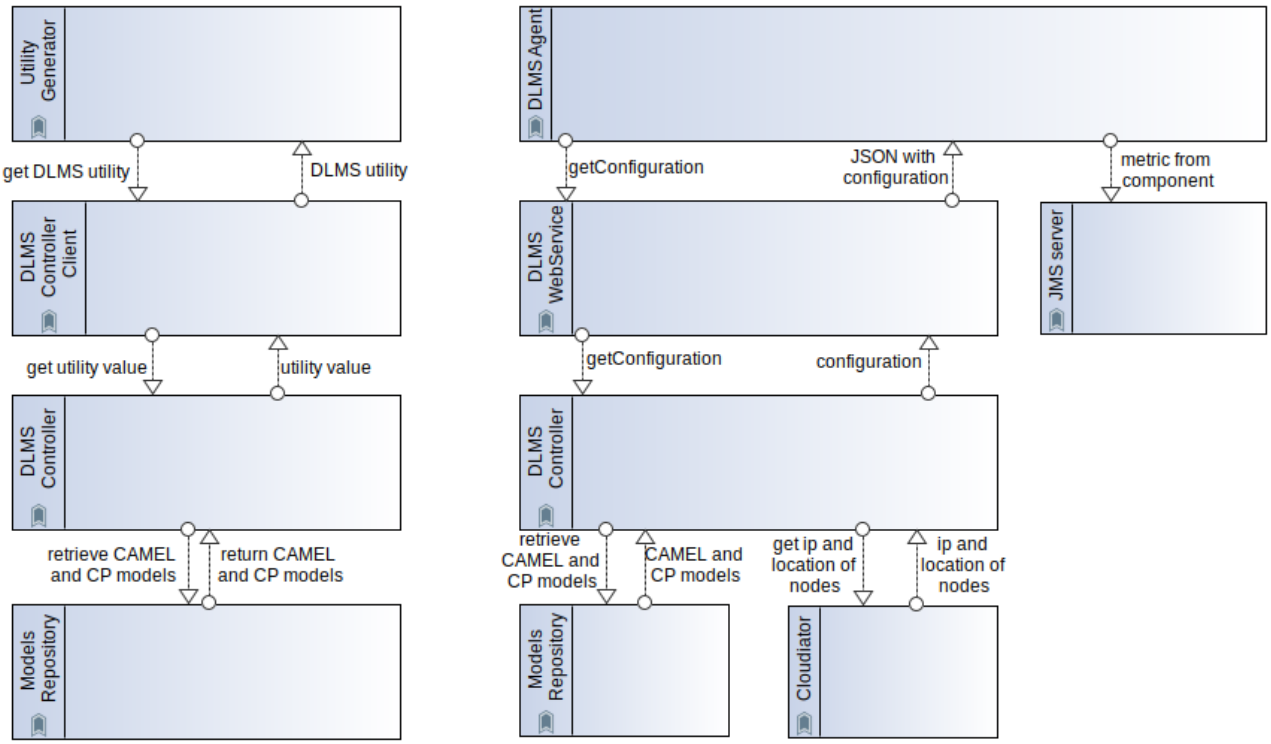

Figure 16. DLMS Collaboration diagram

The explanation of all communication details from the perspective of the DLMS Controller, is listed below. It should be noted that the *messages* described below can be method calls, http methods or elements of message queue and are denoted like this to give a high-level description of the communication.

- Utility Generator:
  - **get DLMS utility**: it sends the message to a DLMS ControllerClient (element of DLMS Utility) with a request for a DLMS utility calculation
  - **DLMS utility:** After the whole process, a utility value is returned to Utility Generator
- DLMS ControllerClient:
  - **get utility value**: internally calls DLMS Utility to run a calculation of the utility value
  - **utility value**: returns value with calculated utility value
- Models Repository:
  - **retrieve CAMEL and CP Models:** The message from DLMS Controller to Models Repository fetching the Camel Model
  - **return CAMEL and CP Models**: The message includes the description of CP model.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

The description of messages of the flow for getting the physical configuration, calculating and propagating the partial metric value from the perspective of DLMS agent is listed below.

- DLMS WebService:
  - **getConfiguration**: the message sent by DLMS Agent to DLMS Web service triggers the process of fetching all information required for the Agent to calculate the metric
  - **JSON with configuration**: return message with physical configuration (IP, location etc.) of the nodes used
- DLMS Controller
  - **getConfiguration**: the call is made by an interface exposed by DLMS WebService to an actual implementation of the required functionality
  - **configuration**: returned physical information about the nodes
- Models Repository:
  - **retrieve CAMEL and CP models**: The message from DLMS Controller to Models Repository fetching the Camel Model
  - **return CAMEL and CP Models**: The message includes the description of CP model.
- Cloudiator:
  - **get IP and location of nodes**: calls the Cloudiator API to fetch physical configuration of nodes
  - **ip and location of nodes**: a return message from the Cloudiator
- JMS server:
  - Metric from component: Message with a calculated metric periodically sent by DLMS Agent to a message queue on JMS sever

8.2.2 Implementation

DLMS has been implemented in Java as a Spring-boot application. It can be built and bundled via Maven in a form of either a fat JAR file or a Docker image. The latter form facilitates its integration into the Melodic's platform swarm. It uses the Lombok library for easier defining classes consistent with a JavaBeans standard. It exploits CDOClient in order to retrieve a CP model and logical configuration of application components. DLMS consists of DLMS Controller, DLMS Utility and DLMS Webservice sub-components, while DLMS Agent is implemented as a separate independent module.

DLMS's source code is available in Bitbucket at: https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/dlms

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

DLMS Agent's source code is available in Bitbucket at: https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/dlmsAgent

Dependencies: Spring-boot framework, CDO Client, Cloudiator Client, Melodic commons, Lombok library.

8.2.3 Configuration

The DLMS configuration is stored in several files:

- eu.melodic.upperware.dlms.input.properties
- eu.melodic.upperware.dlms.properties
- eu.melodic.upperware.dlmsws.access
- eu.melodic.upperware.dlmsws.properties

The most important properties defined in each of these files are:

- esb.url - the endpoint of Upperware Control process
- logging.config - path to log configuration file
- server.port - port employed by a component

Moreover, *eu.melodic.upperware.dlms.properties* defines fields that have to be filled in order to generate an algorithm. Example fragment of this file is listed below:

```
#spring datasource
#needs to be changed
#spring.datasource.url=jdbc:mysql://localhost:3306/dlms?useSSL=false
# below line to prevent Time Zone Issue in mysql
spring.datasource.url=jdbc:mysql://${DB_HOST}:${DB_PORT}/melodic_db?useSS
L=false&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&se
rverTimezone=UTC

spring.datasource.username=melodic
spring.datasource.password=melodic
server.port=8094

## Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen
database
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.MySQL5InnoDBDialect

# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

```
# Calculate latency and bandwidth between two datacenters for
the records based on different weight assignment strategy
dlms.algorithms[0].name=Algo_DataCenterAwarenessRunner

dlms.algorithms[0].className=eu.melodic.dlms.algorithm_runners.Algo_
DataCenterAwarenessRunner
dlms.algorithms[0].interval=1000
#argument order is: update all before the specified time interval
(integer),  update number of records (integer), update based on
time or records (string) => "time" or "numRecords", weight of data
calculated          based on (string) => "averageWeight" or
"latestHigher"
#e.g., 600, 1000, numRecords, latestHigher => implies update all the
records 600 seconds from the current time, 1000 records, and the
update is done based on the number of records in this instance and
latest data are given higher weights
dlms.algorithms[0].arguments=800000,100,numRecords,averageWeight
dlms.algorithms[0].weight=0.1
dlms.algorithms[0].camelId=DataCentreAwareness

#Cluster data centers to different zones, which will be used for
computing graph similarity
dlms.algorithms[1].name=Algo_ClusterDataCentersRunner
dlms.algorithms[1].className=eu.melodic.dlms.algorithm_runners.Algo_
Cluster DataCentersRunner
dlms.algorithms[1].interval=1000
# in the form of clustering_method (AffinityPropagation/PAMClustering),
# cluster. # cluster is used by PAMClustering
dlms.algorithms[1].arguments=AffinityPropagation,4
dlms.algorithms[1].weight=0.1
dlms.algorithms[1].camelId=algo1_CAMEL_ID

# For dlms web service
# communication with esb
esb.url=https://mule:8088/

#### Logback-config
logging.config=file:${MELODIC_CONFIG_DIR}/logback-conf/logback-spring.xml
```

Example configuration file *eu.melodic.upperware.dlmsws.properties* is as follows:

```
server.port: 14000
management.server.address: 127.0.0.1
alluxio.master.hostname: alluxio-master
alluxio.master.address: alluxio-master
alluxio.server.conf.file: /opt/alluxio/conf/alluxio-site.properties
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

```
spring.datasource.url=jdbc:mysql://${DB_HOST}:${DB_PORT}/melodic_db?useSSL=false
spring.datasource.username=melodic
spring.datasource.password=melodic
spring.jpa.properties.hibernate.id.new_generator_mappings = false
spring.jpa.properties.hibernate.format_sql = true
spring.jpa.hibernate.ddl-auto = create-drop
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.MySQL5InnoDBDialect
#debug=true

### logback configuration ###
logging.config=file:${MELODIC_CONFIG_DIR}/logback-conf/logback-spring.xml

#### Communication with ESB ####mpi
esb.url=https://mule:8088
```

Example configuration file *eu.melodic.upperware.dlmsws.access* is as follows:

```
# access information in the form:

# key1.access key1, secret key1;key2,access key2,secret key2...

DataSource,access=dipesh_s3_userid.AKIAA,Tsasat0VyJruadax/;key2,username2
,password2
```

Example confliguration file *eu.melodic.upperware.dlms.input.properties* is as follows:

```
#### Communication with ESB ####
esb.url=https://mule:8088/api/metaSolver/deploymentProcess

#### REST interface port ####
server.port = 8092
pubsub.on = true

 pubsub.topics[0].name = dataCenterConnection
 pubsub.topics[0].url = tcp://ems:61616
 pubsub.topics[0].type = LATENCY_BANDWIDTH

 pubsub.topics[1].name = dataRead
 pubsub.topics[1].url = tcp://ems:61616
 pubsub.topics[1].type = BYTES_READ

 pubsub.topics[2].name = dataWrite
 pubsub.topics[2].url = tcp://ems:61616
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

```
pubsub.topics[2].type = BYTES_WRITTEN
```

8.3 Overview of changes of finalized DLMS

DLMS component and its implementation details were introduced in this deliverable and not in the previous one (D3.4) since DLMS was integrated to Upperware as part of the 3rd MELODIC release. Therefore, no changes can be reported with respect to the previous period.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

9 Graphical User Interface

Mission: The convenient usage of MELODIC platform.

Positioning in Melodic: GUI contains two microservices: GUI frontend and GUI backend comprising the Upperware of the Melodic platform.

9.1 Approach

High-level Approach: GUI Adapter is responsible for managing and usage of MELODIC using the graphical interface.

Functionalities:

- Management of users: creating, adding roles, changing passwords, blocking the accounts
- JWT-based authentication
- Upload the CAMEL Models into the CDO repository
- Start and monitor the deployment of an application
- Display the (re)configuration process to the user
- Management of Cloud provider's credentials
- Showing the deployed artefacts of the application
- Redirection to Grafana[6], Camunda[7] and Webssh[8]

[6] https://grafana.com/
[7] https://camunda.com/
[8] https://pypi.org/project/webssh/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

9.2 Technical implementation

9.2.1 Architecture

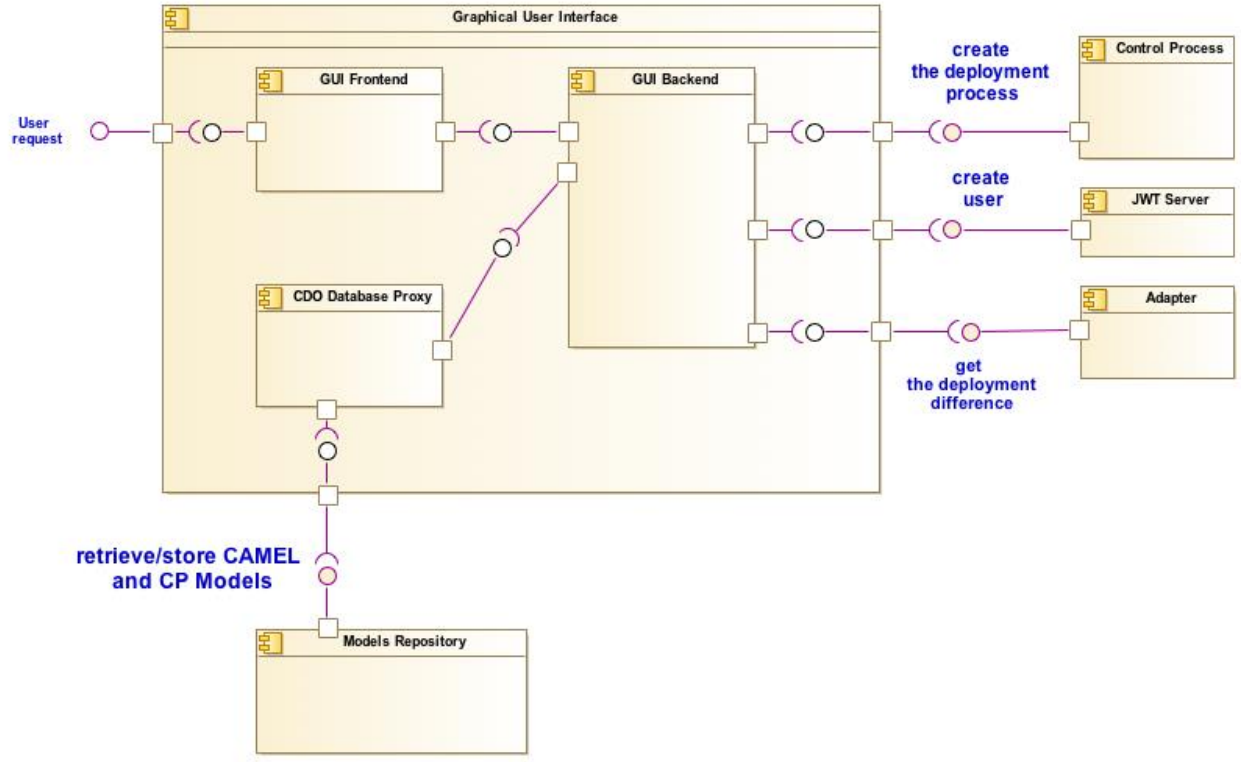

Figure 17. GUI Component in Upperware diagram

In Figure 17, the architecture of the GUI is depicted which comprise the following main components:

- **GUI frontend:** part of the GUI responsible for displaying the data to the user
- **GUI backend:** part of the GUI responsible for processing and retrieving the data from the Upperware and the Cloudiator

In the following UML collaboration diagram, we describe the flow of information between the Adapter and the rest of the Upperware components with which it interacts. Specifically, the following messages are exchanged:

- with GUI Frontend:
    - **Get the data:** with the information which data should be passed
- with Models Repository

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

- **Store CAMEL Model:** message that contains the CAMEL Model that should be stored
- **Get CAMEL and CP Model:** message with id of the CAMEL or CP Model that should be retrieved

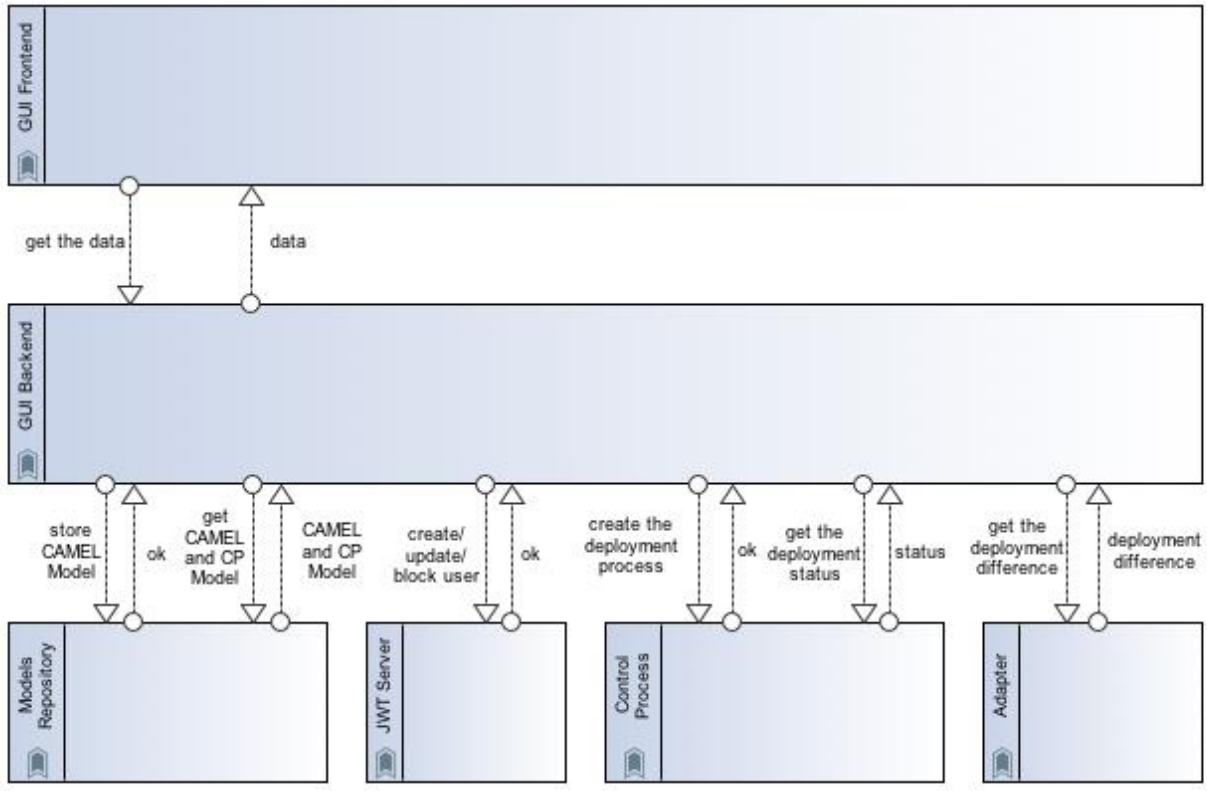

Figure 18. GUI Collaboration diagram

- with JWT Server:
  - **Create user:** message that contains the username and password of a new user
  - **Update user:** message with user id and the password that should be updated
  - **Block user:** message with the id of the user that should be blocked
- With Control Process:
  - **Create the deployment process:** message that contains the information about the user and the application that should be deployed
  - **Get the status:** message that contains the id of the process
- With Adapter
  - **Get the deployment difference:** message that contains the difference of the

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

currently deployed and new solution

9.2.2 Implementation

The GUI frontend component has been developed in Typescript 3 using Angular 7. The graphical elements have been developed using Angular Material 7.

The GUI backend component has been developed in Java, version 8. The project is built by Maven and thanks to the Maven plugin[9], a Docker image is created which allows to run this component as a separate microservice.

GUI backend's source code is available in Bitbucket at:
https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/gui-backend

Dependencies: Spring-boot framework, CDO Client, Cloudiator Client, Melodic commons, JWT commons

9.2.3 GUI Configuration

The GUI configuration is stored in `eu.melodic.upperware.guiBackend.properties` file, using Java properties format. The most important aspects of the required configuration are the following:

- ESB URL: the endpoint of Upperware Control process, which Adapter invokes to signal that application reconfiguration is required.
- ESB ssl Verification Enabled: flag indicated if the ssl verification should be enabled
- CAMUNDA URL: the endpoint of Camunda, which GUI invokes to get the status of current deployment process
- CDO UPLOADER validation enabled: flag indicated if the validation during uploading the CAMEL Model into CDO repository should be done
- JWT Server URL: the endpoint of the JWT Server which GUI invokes to authorize the user
- ADAPTER URL: the endpoint of the Adapter which GUI invokes to get the deployment difference
- Server ssl key store: the path to the key store
- Server ssl key store password: the password to the key store
- Server ssl trust store: the path to the trust store

[9] `com.spotify:docker-maven-plugin`

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

- Server ssl trust store password: password to the trust store
- Logging config: path to log configuration file

Next, we provide a sample GUI backend configuration file:

```
#### Communication with ESB ####
esb.url = https://mule:8088
esb.sslVerificationEnabled = true
#### Communication with Camunda
camunda.url = http://camunda:8095
#### CDO uploader settings
cdoUploader.validationEnabled = true
#### JWT-server settings
jwtServer.url = http://jwtserver:8080
#### Adapter settings
adapter.url = http://adapter:8080
### HTTPS configuration
server.ssl.key-store = /certs/keystore.p12
server.ssl.key-store-password = XXX
server.ssl.trust-store = /config/common/truststore.p12
server.ssl.trust-store-password = XXXX

#### Logback-config
logging.config = file:${MELODIC_CONFIG_DIR}/logback-conf/logback-spring.xml
```

We note that the GUI internally uses a CDO client to communicate with the Upperware Models Repository, in order to retrieve the Camel Model and create Deployment Instance Model. The CDO client reads its configuration from `eu.paasage.mddb.cdo.client.properties` file.

9.3 Overview of changes of finalized GUI

The Graphical User Interface has not been considered as an obligatory component of the MELODIC project. The component is completely new as it has been developed as the

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

additional feature of the MELODIC platform.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

10 Differences with respect to underpinning frameworks

According to the description of work, MELODIC was designed in a way that re-used as much as possible relevant concepts and implementations from previous open-source frameworks. Specifically, PaaSage was one of these frameworks that made sense to be re-used and be significantly extended for achieving the objectives of this project. Therefore, in this section, we go through the core Upperware components and discuss the main similarities and dissimilarities among Melodic and the PaaSage framework. The following Table 1 summarizes these differences. Specifically, for each of the Upperware components, we note if it is a completely new component, an adapted or a re-engineered one, with respect to an underpinning framework. In the remaining subsections further details are provided.

Table 1: Upperware components and relation to underpinning frameworks

Upperware Components	New component built in terms of Melodic project	Adapted from underpinning frameworks	Re-engineered from underpinning frameworks
CP Generator			√
Metasolver			√
CP Solver		√	
Utility Generator	√		
Solver to Deployment		√	
Adapter	√		
DLMS	√		
EMS	√		
GUI	√		
Metadata Schema Editor	√		
CAMEL Editor			√

10.1 CP Generator

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

The CP Generator has been re-engineered due to the changes connected with the new CAMEL and the new concept of the Constraint Problem. The main differences between the CP Generator in PaaSage and the CP Generator in Melodic are:

Input: In PaaSage the CP Generator gets the Application and Resource CAMEL Model while now it gets only the Application CAMEL Model.

Output: In PaaSage the CP Generator produces the Constraint Problem (CP) Model and PaaSage Application Model with PaasageConfiguration. Now the PaasageConfiguration is completely removed and not needed anymore. CP Generator produces the CP Model and caches with Node Candidates offers.

Information about available offers: In PaaSage the CP Generator gets information about available virtual machine offers from Provider Model in CAMEL Model. Now the CP Generator fetches the available Node Candidates from Cloudiator 2.0 and filters the fetched Node Candidates based on annotation in resource requirements and location requirements.

Variables: Variables in the Constraint Problem Model represent the virtual machines offers in PaaSage. The result of this is the big domain and the solution space. In MELODIC variables are attributes of Node Candidates like RAM, CPU, storage or location. The domain of variables is based on the fetched and filtered Node Candidates offers. Melodic also has a new way of mapping from CAMEL objects to Constraint Problem, through annotations.

Metrics: Currently, the CP Generator supports raw and composite metrics. New types of CP Metrics have been introduced: representing variables for a currently deployed solution.

Constraints: CP Generator supports variable constraints with a formula which may contain both variables and metrics. Also, automatic constraints for reducing the domain of variables are created.

Optimization: In PaaSage the CP Generator creates a basic objective function for solvers. Currently, the function is written in CAMEL Model and the CP Generator does not need to create a function.

Security: In Melodic the CP Generator is integrated with JWT Server and it secures the API.

10.2 Metasolver

In the context of Melodic project, MetaSolver has been rewritten and repurposed. Specifically, its functionality has been extended, from merely selecting the solver to use for solving the CP problem, to also include:

- Evaluation of the generated (by the selected solver) solution, based on the utility value

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

of the currently deployed and the newly produced solution. The new solution must have a significantly better utility value than the current one in order to be realized.

- Monitoring of the most recent values of various important metrics, by subscribing to the corresponding event topics, and updating of CP model with them. EMS provides the relevant information regarding which metric to monitor and how to subscribe for them.
- Monitoring for SLO violation events and triggering a new application reconfiguration process iteration. EMS provides the relevant information regarding which SLO violation events to monitor and how to subscribe for them.

Metasolver has been rewritten in order to comply with the architectural approach of Melodic platform and coding standards. For instance, Spring boot framework has been used for its implementation and an ESB is used for its interactions with other platform components. Moreover, it has been developed following the versioning standards and code styles selected by Melodic consortium.

10.3 CP Solver

With respect to its previous version in PaaSage, the CP Solver has evolved in terms of both functionality and implementation. Concerning functionality, the main advancements performed include the following:

- Interaction with the Utility Generator for the calculation of the utility of candidate solutions. In PaaSage, the utility function was already fixed in the CP model and was part of the optimisation objective in the constraint (optimisation) problem that this solver had to solve. Now, the utility function is encapsulated by the Utility Generator which is called as a black-box by passing to it the appropriate parameters for calculating the candidate solution's utility
- The running mode was altered from the form of a daemon to that of a micro-service. This was a result due to the change of integration paradigm in Melodic. This has resulted in the production of additional classes in the implementation of this component as well as the removal of previous ones dedicated to realising that demonized form of the component. Further, this obviously resulted in the change of the component's internal architecture and class diagram.
- Due to the previous change as well as to the unification of the logging mechanism of all components in the platform, the component now reads its configuration from a properties file.
- The aforementioned change also has led to the need to actually notify the deployment process of the platform when the respective solution is produced (optimal or infeasibility result).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

On the other hand, in terms of implementation, apart from the respective changes in the component classes and class diagram, the corresponding artifact produced from the compilation of the component has changed. Initially, the artifact took the form of a flat jar. Now, it can also take the form of a certain image which can enable the component to be containerised within the deployment and execution of the Melodic platform instances. Further, due to the need to configure the component, apart from the requirement for the proper configuration of its enclosing CDO Client that comes through a certain properties file, the component configuration relies also on another property file which needs to be appropriately pointed to through different ways (e.g., system/environment variables). Finally, we should highlight the re-engineering of CP Solver to become conformant to the latest version of Choco Solver which enabled to produce a more robust and reliable CP model solving code.

10.4 Utility Generator

This is a completely new component responsible for calculating the utility value for each generated by solvers solution. The general approach has been described in the PaaSage deliverables: observation, fuzzyfication, evaluation and defuzzification. During the MELODIC project, the concept has been changed. Currently, the Utility Generator extracts the utility function formula from the CAMEL Model, converts all needed attributes and calculates the value.

10.5 Solver to Deployment

The Solver to Deployment has been rewritten during the MELODIC project and then merged with the Adapter. The main reasons for the merge were:

a. the reduction of the resource usage of the Melodic platform
b. the introduction of a single point of adapting responsibility

The first point is very straightforward - having one microservice, instead of two separate ones, simply reduces the usage of the memory and CPU for the Melodic instance. The second reason means that after the change, all features required to properly transform the Constraint Problem Solution (Reasoning Domain) to a Deployment Model (Execution Domain) are being encapsulated within a single component. This allows for more efficient development of new software features to the platform.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

10.6 Adapter

The Adapter has been completely rewritten during the MELODIC project. The main differences are connected with the architecture and communication. The PaaSage Adapter is a simple Java program which is checking for 30 seconds if a new Deployment Camel Model arrived. The main differences are listed:

Input: in PaaSage the Adapter gets the Application Model, the Organisational Model and the Provider Camel Model. In Melodic the Adapter gets only the Application Camel Model.

Deployment configuration plan: In PaaSage, the Adapter has 10 types of tasks while in MELODIC the number of tasks has been reduced to 6.

Validation of the plan: In PaaSage, the validation of the plan does not exist. In MELODIC, the validation is done through the Authorization Service.

Deployment: The communication with the Cloudiator API is multi-threaded.

10.7 DLMS

The DLMS is a completely new component of the MELODIC project. Its purpose is to provide a middle layer for holistic management of user interaction with the data stored internally in MELODIC internals. From the highest-level perspective, the DLMS is divided into two components: DLMS being the central management system and DLMS Agent which is installed on deployed entities with a task of gathering and sending metrics. The main DLMS consists of three sub-components: DLMS Utils, DLMS Controller and DLMS WebService.

The general design was described in previous deliverables of MELODIC [3]], [[4]. DLMS' specific architecture and implementation details was described in this document.

10.8 Event Management System

Event Management System (EMS) corresponds to SRL Adapter and Axe components of PaaSage, since they both aim at deploying sensors at application VMs and then gathering measurements, in order to assist application scaling. However, they differ in a number of ways.

First, the goal of SRL[10] Adapter was (a) to instantiate the MetricInstances of MetricContexts in application's CAMEL model, based on Scalability and Metric models, and (b) to install probes and aggregation in Visor (part of Cloudiator installed in application VM in order to collect and

[10] https://gitlab.ow2.org/paasage/srl-adapter

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

propagate measurements). After model instantiation and probe deployment SRL Adapter stays idle and Axe takes responsibility to propagate sensor measurements to Upperware, following a centralised approach. Contrary, EMS takes a holistic approach, since it instantiates the monitoring system and moreover performs measurements processing and propagation in several layers. It also provides the processing and propagation mechanisms. Hence EMS as a single component undertakes the whole monitoring task.

EMS scope has been extended beyond application scaling, in order to address the additional monitoring requirements of Melodic project (for instance SLO violations and computation of metric variable values). It also provides the means for setting up a multi-level measurement processing topology; for instance, having various processing tasks taking place at application VMs and/or at cloud provider level or at Upperware level. One of the advantages of this approach is that it allows nodes to join after application deployment (for example due to scaling up or due to node restart). But one of its strong points is the fact that EMS avoid the need to propagate all the monitoring information to a centralised server for processing, due to its multi-layer and hierarchical solution.

In technical terms, EMS has been developed from scratch using Spring-boot framework. It has a modular structure, thus making it easy to maintain and extend it. It also encompasses two well-known tools for measurement propagation and processing, namely Apache ActiveMQ and EsperTech's Esper event processing engine. Much attention has been put to secure propagation of measurements as well as communication to EMS. Eventually, EMS provides an extensive set of configuration options for controlling behaviour although the default values should allow using it out-of-the-box.

10.9 Graphical User Interface

The Graphical User Interface is completely new as it has been developed as an additional feature of the MELODIC platform that provides a real-time overview of the application life-cycle management and simplifies the initial configuration steps of the process. What is more, it allows for convenient management of users, uploading and deleting models into the Models Repository, the management of Cloud provider's credentials and the deployed artifacts.

10.10 Metadata Schema Editor

The Metadata Schema Editor is also a completely new component that has been developed in order to create and manage the Melodic Metadata Schema, as it has been analysed in [9]. The Melodic Metadata Schema provides a comprehensive, modular and extensible vocabulary for modelling cloud application aspects. In terms of Melodic it is used for formally capturing any

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

required extensions of the CAMEL model.

10.11 CAMEL Editor

There were two editors for CAMEL in PaaSage. Both of them were re-engineered for the purposes of the MELODIC project as well as for conformance reasons with respect to the changes that have been performed in CAMEL. In a nutshell, the changes that have been conducted to both components can be summarised as follows:

- CAMEL textual editor:
  - Conformance to latest CAMEL version
  - Change of textual syntax to become even more uniform as well as cover the specification only of those elements that must be modelled by the devops (and not the system itself) in the context of their multi-cloud applications
  - Introduction of extra logic for the parsing of mathematical expressions in the context of metric and (metric) variable derivation formulas
  - Development of CAMEL documentation and its visualisation via the editor during the CAMEL element modelling
  - Production of self-sustained XMI models with no external CAMEL model references to be exploited for their correct uploading in the MELODIC platform
- CAMEL form-based editor:
  - Conformance to CAMEL version 2.0
  - Editor modularisation and automatic compilation via Maven
  - Proper visualisation of metadata models
  - Coverage of additional aspects/domains via the incorporation of extra modelling perspectives
  - Integration with MELODIC platform version 2.0

Finally, we should mention that for both editors, template models (e.g., for metrics) have been developed which can or are incorporated in the modelling space to allow the re-use of CAMEL elements and thus the more rapid production of new CAMEL models. In our opinion, all these changes enhanced the experience of the modeller, reduced his/her modelling effort as well as enabled a better integration of the editors with the MELODIC platform.

11 Conclusions

This last deliverable of WP3 encapsulated a detailed description of the following core components of Upperware: *CP Generator, Metasolver, CP Solver, Utility Generator, Adapter, Event Management System, DLMS and GUI.* All these Upperware components undertake the

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

critical roles of the application optimisation recommendation, the initial application placement and the continuous adaptation enactment. Their operation brings the necessary functionality to the Melodic platform, for making timely decisions on appropriate cross-cloud data placements and application deployments, starting with the generation of a formal Constraint Problem model and resulting to a target application configuration the captures the optimised deployment into cross-cloud resources.

All these Upperware components have been integrated with the rest of the platform components, consolidating a major part of the Melodic Release 3.0.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

References

[1] Y. Verginadis, G. Horn, K. Kritikos, F. Zahid, D. Baur, P. Skrzypek, D. Seybold, M. Prusiński, S. Mazumdar, "D2.2 Report on Architecture and Initial Feature Definitions", Melodic Deliverable, 2018.

[2] Y. Verginadis, C. Chalaris, I. Patiniotakis, V. Stefanidis, F. Paraskevopoulos, E. Psarra, B. Magoutas, E. Bothos, E. Anagnostopoulou, K. Kritikos, P. Skrzypek, M. Prusiński, M. Różańska "D3.4 Report on Workload optimisation recommendation and adaptation enactment", 2019

[3] F. Zahid, D. Pradhan, "D3.2 Report on Business logic for supporting the complete data and data-intensive application life-cycle management", Melodic Deliverable, 2019.

[4] F. Zahid, K. Kritikos, S. Mazumdar, D. Seybold, Y. Verginadis, "D2.5 Report on Data Placement and Migration Methodologies", Melodic Deliverable, 2018.

[5] D. Baur, D. Seybold, "D4.3 Report on Resource Management Framework Prototype", Melodic Deliverable, 2018.

[6] F. Zahid, Y. Verginadis, G. Horn, K. Kritikos, D. and E. G. Gran, "D2.3 Report on Final framework and external APIs", Melodic Deliverable, 2019.

[7] P. Skrzypek, I. Patiniotakis, Y. Verginadis and C. Chalaris, "D5.3 Report on Security requirements & design", Melodic Deliverable, 2018.

[8] Kritikos, K., Magoutis, K., & Plexousakis, D. (2016). Towards Knowledge-Based Assisted IaaS Selection. In Proceedings of the 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Luxembourg, 12-15 December (pp. 431-439).

[9] Y. Verginadis, I. Patiniotakis, C. Chalaris, G. Mentzas, "D3.1 Metadata Schema Management", 2018.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664