COSMOS

Cultivate resilient smart Objects for Sustainable city applicatiOnS

Grant Agreement Nº 609043

D5.2.3 Decentralized and Autonomous Things Management: Software Prototype (Final)

WP5: Decentralized and Autonomous Things Management

Version: 1.0
Due Date: 30 June 2016
Delivery Date: 30 June 2016
Nature: Report
Dissemination Level: Prototype

Lead partner: 4 (ICCS)
Authors: Orfefs Voutyras (ICCS), Panagiotis Bourelos (ICCS), George Kousiouris (ICCS), Juan Rico (ATOS), Juan Sancho (ATOS), Roberto Gonzalez (ATOS)
Internal reviewers: Franz Carrez (UniS), Bogdan Târnaucă (SIEMENS)

www.iot-cosmos.eu

The research leading to these results has received funding from the European Community's Seventh Framework Programme under grant agreement n° 609043

Version Control:

Version	Date	Author	Author's Organization	Changes
0.1	05/06/2016	O.Voutyras	ICCS	Initial version.
0.2	19/6/2016	O.Voutyras	ICCS	Section 3 updated.
0.3	22/6/2016	J. Sancho	ATOS	Section 4 updated.
0.4	23/6/2016	O.Voutyras	ICCS	Version for internal review.
0.5	24/06/2016	F.Carrez	UniS	Internal review.
0.6	27/06/2016	B. Târnaucă	SIEMENS	Internal review.
0.7	28/06/2016	J. Sancho	ATOS	Section 4 updated.
0.8	29/6/2016	P.Bourelos, G.Kousiouris	ICCS	Section 2 updated.
1.0	30/06/2016	O.Voutyras	ICCS	Version for submission.

Table of Contents

List of Figures

List of Tables

Table of Acronyms

Acronym	Meaning
API	Application Programming Interface
APP	Application
CB	Case Base
CBR	Case Based Reasoning
CEP	Complex Event Processing
CPU	Central Processing Unit
D	Deliverable
DoS	Denial of Service
FC	Functional Component
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JDK	Java Development Kit
REST	Representational State Transfer
SSH	Secure Shell
T&R	Trust & Reputation
TTL	Time To Live
URL	Uniform Resource Locator
VE	Virtual Entity
VM	Virtual Machine
WP	Work Package
WSN	Wireless Sensor Network
XP	Experience

1. Introduction

D5.2.3 is the result of the aggregation of the content introduced in D.5.2.1 and D5.2.2 and, as a result, it gives a global and final view of the work done during the whole COSMOS project. In addition to the revision of the pre-existing sections, a **new subsection** was added (subsection 4.3) and **new content** was provided in previous subsections (mainly in subsections 2.1, 2.2, 2.3, 3.1 and 3.2).

This document is the complement to the delivered software regarding the month 34 WP5 prototype deliverable and is based on the concepts and functional components introduced and described in D5.1.3 [1].

The outline of the document is as follows:

- **Section 2** presents the final version of the **Planner**. New capabilities have been added to this component and Stress Tests were undertaken in order to evaluate its performance based on several criteria and across several environments.
- **Section 3** presents the final version of the **COSMOS Trust & Reputation model** (called TRM-SIoT) introduced by our team. The performance of this model in terms of security, quality of service and scalability was retested by running new simulations.
- **Section 4** presents the **Network Runtime Adaptability Module** and details regarding the interfaces it uses are given.
- **Section 5** concludes the deliverable.

Regarding the sections presenting the final results of the various WP5 functional components (Sections 2, 3 and 4) we provide a general description of the components and the corresponding architecture, a short presentation of the various interfaces used and the implementation procedures, as well as test results and usage instructions.

2. The Planner

2.1. Description & Architecture

As it is stated in D5.1.3 [1], the functional component that enables the VEs to use CBR is the **Planner** (a Reasoner). The Planner will become part of the VEs during their registration time and will run locally. The main functionality of the Planner is providing the VEs with the ability to react to problems using a reasoning technique for finding the most appropriate solution to be applied based on similarities to previously encountered or experienced situations. This is a step towards the autonomic behavior of the VEs as depicted by the goals of Task 5.2 (Autonomous and Predictive Reasoning of things). In this deliverable we will further analyze some new functionalities and capabilities of the Planner.

The Planner FC in Y3 has undergone major changes in its internal architecture in order to reflect two things. First is the desired change in inter-component communication as is achieved through the creation of multiple endpoints for filtering incoming requests. This change has changed the way applications handle calls to VE capabilities, because now we have achieved the creation of unique point of Application data convergence to the system.

This happens by hiding in a sense the actual reasoning capabilities of the Planner behind an exposed endpoint which is agnostic and dynamic enough to handle a theoretical multitude of incoming messages. A big part in this abstraction of the original implementation is also due to the standardization of messaging between Apps and VE, imposed by the JSON message format.

Using an internally standardized method of communication, has aided development of a unified external interface, implemented in code by the InterfacesOfPlanner class and specifically the generic message handler. Every call in JSON format received in this API must contain a JSON body conforming to specific prerequisites. First, the message must be contained in a payload attribute, because of integration reasons with the Node-Red platform. Seeing as how Node-Red is a major driving force behind the flow of information and may play an even greater role in joining divergent data sources together, we have chosen to adopt the JSON message format of this platform.

Internally we possess the freedom to structure the rest of the message as desired. Therefore we have approached this structure in a way that is similar to previous year's implementations. The first sub attribute is a message type classifying the message, as arithmetic or a textually represented Case, as far as problem values are concerned. The method of calculating similarities differs and our CBR approach diverges in each case, as textual Cases are being calculated through Sorensen-Dice Coefficient or the Jaccard Index.

2.1.1. Cases creation from historical data

In D5.1.3 [1], the notion of connecting the Planner of an individual VE with the COSMOS Cloud is introduced. More specifically, the Planner should be able to retrieve historical data from the Cloud that is developed by WP4. Such a retrieval will be a blanket operation limited by the scope of a time period for which all relative data should be retrieved. The Application Developer is responsible for the creation of the wrapping SPARK SQL query [2] towards Cloud storage, in accordance with the interface provisions (REST call) determined in D4.2.3, which will perform the filtering of the retrieved data and any needed calculations on them. In this step also integration with the Privacy & Consent Management is performed. The Application should make use of specified Planner interfaces in order to direct a structure of storage for the data as Problem-Solution pairs to be added in the local CB.

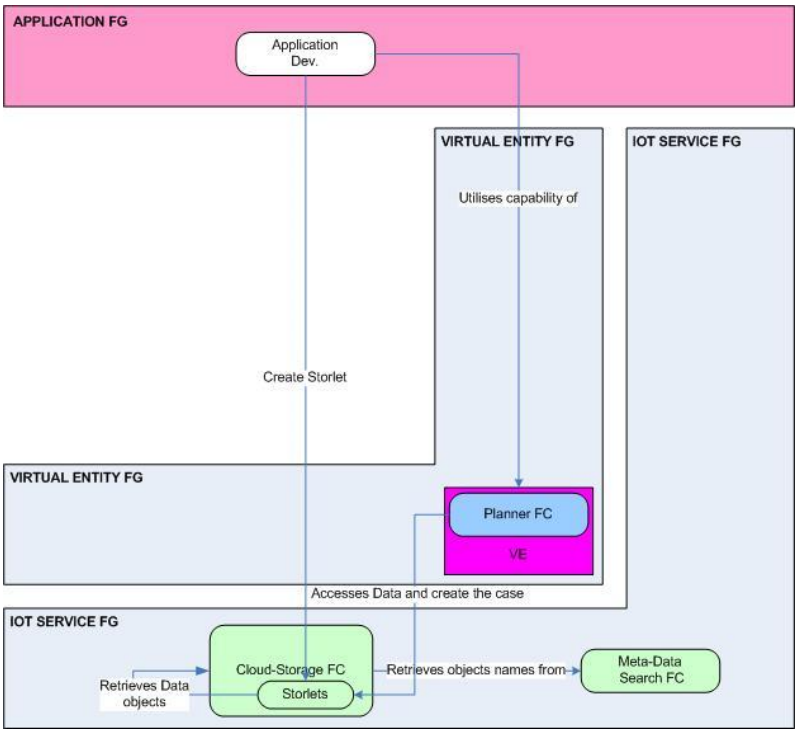

Figure 1: Interaction diagram for Cases creation from historical data.

2.1.2. Topic based detection of events

Another capability of the Planner is the topic based detection of events which pertain to internal state changes of a VE and can be used to initiate correctional actions and reconfigurations which may use other components such as the Experience Sharing component.

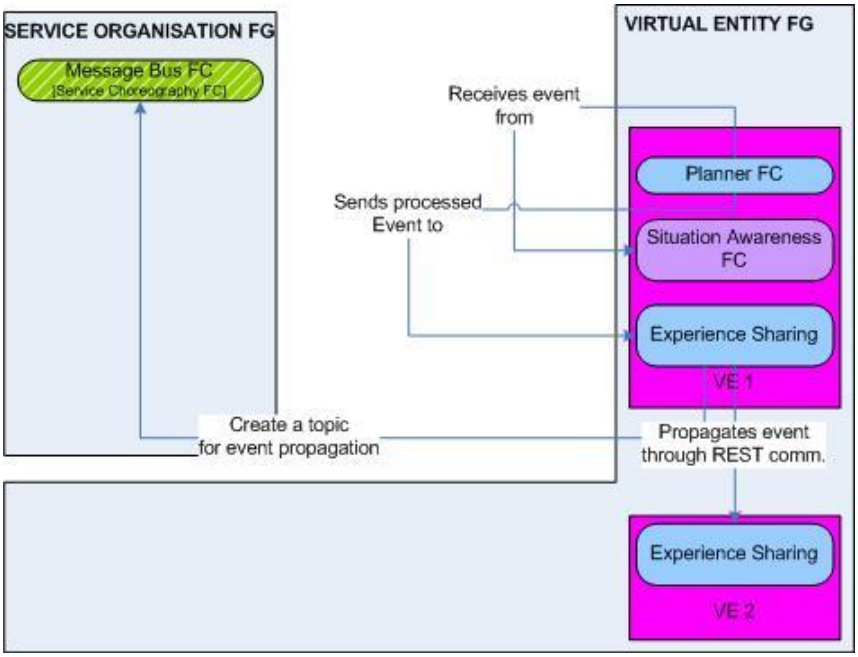

Figure 2: Example of event-activation of the Planner.

Such events can be detected by the internal µCEP of the VEs and can be, for example, changes to the social environment of a VE (thus changes to its Friends lists) may cause the need of reevaluating the TTL variable used by the Experience Sharing component and the rest

Decentralized Discovery mechanisms. The changes to be performed will be dictated by COSMOS specific Cases provided to each VE by the platform or through the Decentralized Case Discovery component.

2.1.3. Cases Retrieval and Revision

The new version of the Planner is able to run SPARQL queries not only for the retrieval of Problems but of Solutions too. In other words, the Planner can retrieve Problem specific values from Cases of the local CB, by providing the Solution values and structure.

Moreover, the first steps towards new ways of Solutions modification have been taken. Now, the Planner can modify numerical Solution attributes (using e.g. linear functions) based on the similarity of the retrieved cases. Currently, this method is used when the Planner faces small divergences of similarities.

Finally, one new type of retrieval technique is being tested, the Adaptive-guided Retrieval (AGR) stated and described in [3]. This retrieval technique focuses on the adaptability of retrieved cases rather than the absolute similarity between them. In the VE CBR case, this will be demonstrated by giving priority to the retrieval of Cases, which have similar similarity ratings per value checked.

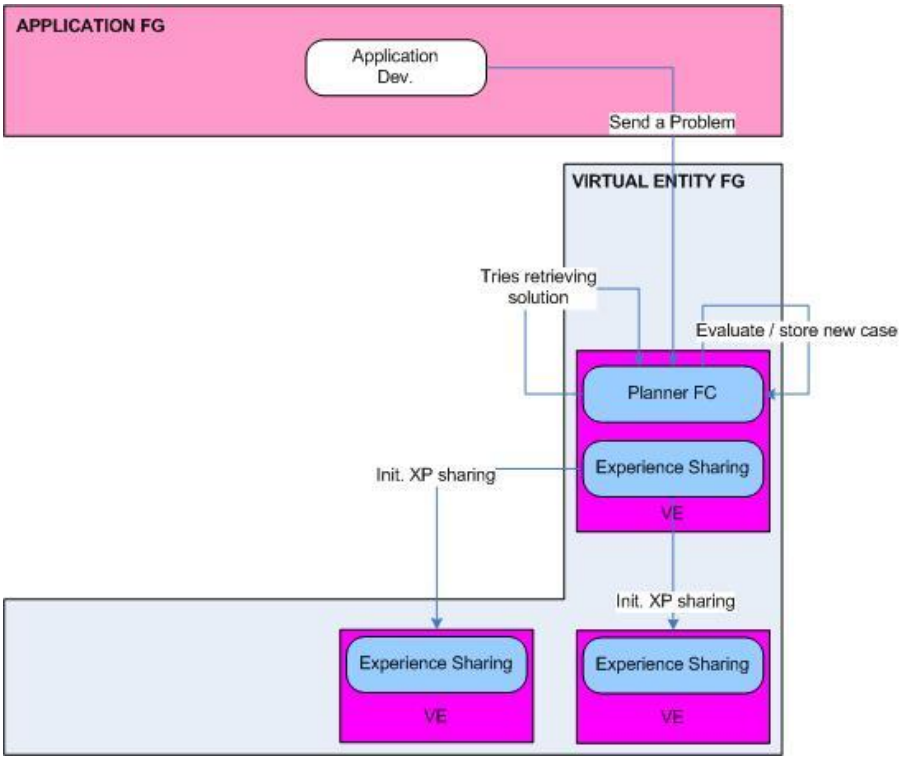

Figure 3: Interaction diagram for Cases creation through the CBR cycle.

2.2. Interfaces & Implementation

2.2.1. Cases creation from historical data

The Planner will have access to the historical data store for Cases creation through the use of Spark queries. These queries are incorporated in the App code, which afterwards will have to call on a specific API of the Planner to control the aggregation of retrieved data as Cases

(based on a defined Case Structure) as well as the storing of suitably created new Cases in the CB:

> *ArrayList<ArrayList<String>>* ***historicalCaseCreation****(Object cloud_data, ArrayList<String> params, ArrayList<String> solutionat)*

In this interface, the input variables are the cloud data retrieved as Objects, a list of Problem-attribute names and a list of Solution-attribute names. The returned values are the created Cases to be stored:

> *void* ***storeCase****(ArrayList<String> problemparam, ArrayList<String> problemval, ArrayList<String> solutionparam, ArrayList<String> solutionval)*

Additionally problem values and parameters as well as solution parameters are now stored as unified strings separated by special characters, therefore moving the need for processing behind the external interface logic, saving in time, provided the rest of the message format is not up to standards required. After that, the internal calls are being made again through REST APIs into the corresponding Service of the Planner. It is inside those Service logics that calculations and actual processing of data takes place, along with decisions for calls to the also renewed version of the Experience Sharing FC.

2.2.2. Topic based detection of events

The Planner is connected to the local µCEP by an MQTT interface which is configured to "listen to" incoming Complex Events. After the detection of such events, the Planner will perform the necessary corrections/actuations and may further propagate them by calling on the Experience Sharing FC. The interface for this listening endpoint is:

> *void* ***doListen****(String topicFilter)*

The topic filter may limit or increase the scope of the tracked Complex Events.

2.2.3. Cases Retrieval and Revision

The primary API of the Planner FC is the one which retrieves the most appropriate/similar Case from a CB:

> *ArrayList<String>* ***searchSimilarCase****(List<Double> w, ArrayList<String> prob, ArrayList<String> params, ArrayList<String> solutionat, double thr, boolean shareable)*

The input parameters are a vector of Problem-attribute weights for the calculation of the total similarity between Cases, a list of the Problem-attribute names, a list of the Problem-attribute values, a list of the Solution-attribute names, thresholds for the total similarities and a flag on whether the result is intended for Experience Sharing.

Auxiliary APIs have been defined for the retrieval of customized information from the local CB as well as the investigation of specific Case Structures in it. The input parameters are similar to APIs concerning Cases in general. A new API for Case modification is presented which will act as described in Section 2.1. Its description follows:

> *ArrayList<String>* ***getModifiedCase****(ArrayList<String> info, String inclination)*

The inputs are the Case itself (Solution part) and the "direction" of the value modification (ascending, descending).

Finally, there is also an interface for calculating the maximum TTL used by any Decentralized Discovery Requests:

int ***calculateMaxTTL****()*

In Y3, the code of the similarity metrics has been modified in order to handle calculation of Case existence in a more appropriate way, as previous implementations had quirks as far as Cases with interchangeable attributes (same names). Additionally, the similar Case/Solution retrieval method was made more dynamic, by now supporting Case retrieval with no Solution values. This allows us to use only the predefined Solution attributes in providing, if not actuation, then simple messaging through pre-stored solutions. Finally, in correcting the instances of the VE code, we have added a similar standardization behavior to the returned message. Thus both input and output of the Planner FC is now usable by the Node-Red instances running on the VE.

2.3. Demonstration Scenario & Use-Cases

2.3.1. Scenario Synopsis

The main scenario on which WP5 focuses on is the Heating Schedule Management of the Camden Use Case, in which the Planner will use its full CBR capabilities. Through this scenario, the other FCs of WP5 will be demonstrated too.

Each flat will be equipped with a management and monitoring tablet which can act as the Gateway to the entire network sensors and actuators of the VE-flat. The VE code and all the COSMOS applications the End User may choose to install will be located on this tablet. The Heating Scheduling application will provide a GUI for ease of access with a minimal of complexity and required options.

In the beginning, the End User (e.g. the owner of a VE-flat) has to state which is the desired temperature for his/her flat for specific time intervals of the planning period and which is the available budget. The Application will then form the Problems by combining the user's input with predicted external temperatures throughout the time period of interest (provided by COSMOS or third party Apps) and will use VE services and functionalities offered by COSMOS, in order to locate similar Problems and return their Solutions by using the **searchSimilarCase** interface. The Solution is structured as the actuation to be undertaken and the consumption per time interval. This process involves the use of Case Base Reasoning on the local CB of the VE. The initial set of Cases of the CB will be created by historical data through the **historicalCaseCreation** and **storeCase** interfaces of the Planner. Modification of Cases may be performed by using the Planner's **getModifiedCase** interface. Experience Sharing may also be used at this point. Finally, the Application will evaluate the monetary requirements of the returned Solutions and actuate the Schedule or modify the initial input if the End User's budget is overshot.

It should be noted that, in parallel, the Planner will have to react to changes of its social environment. The mechanisms behind these changes are analyzed in Section 3. In case of changes to the Friends Lists of a VE, the Planner uses COSMOS Cases in order to "reprogram" itself by modifying the maximum TTL setting with the **calculateMaxTTL** interface.

The corresponding Sequence and Use-Case diagrams are given in the next subsection.

2.3.2. Sequence and Use-Case diagrams

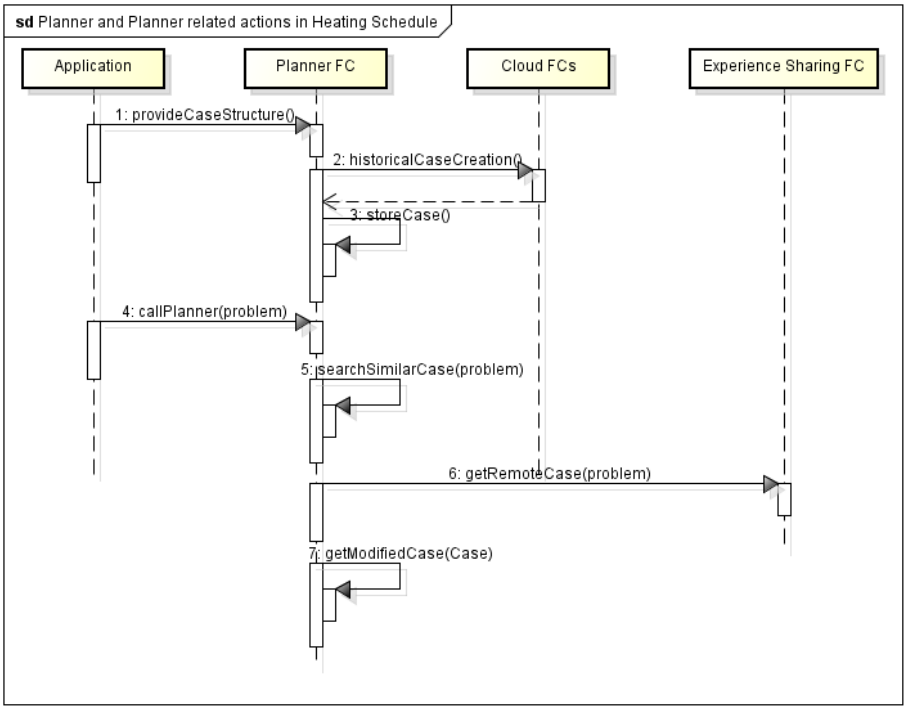

Figure 4: Heating Schedule Sequence Diagram.

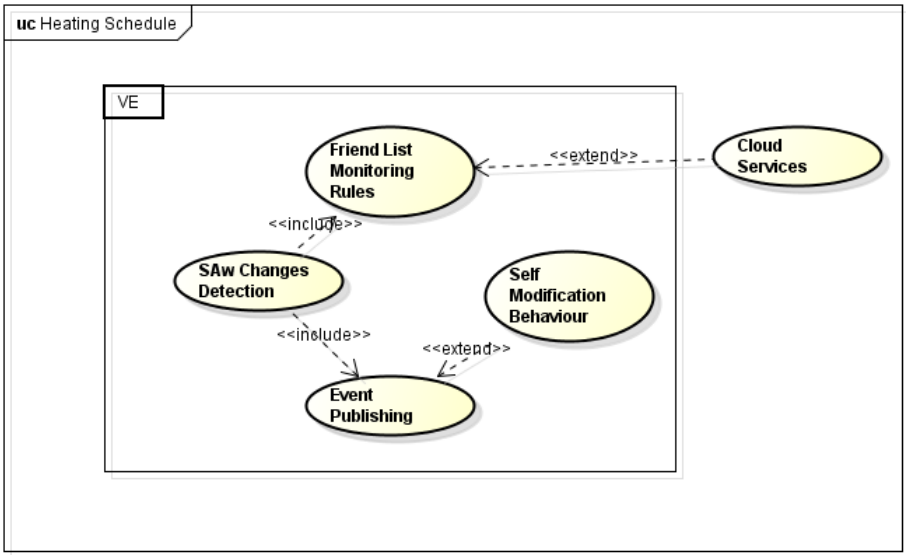

Figure 5: Use-Case diagram for the Heating Schedule Scenario.

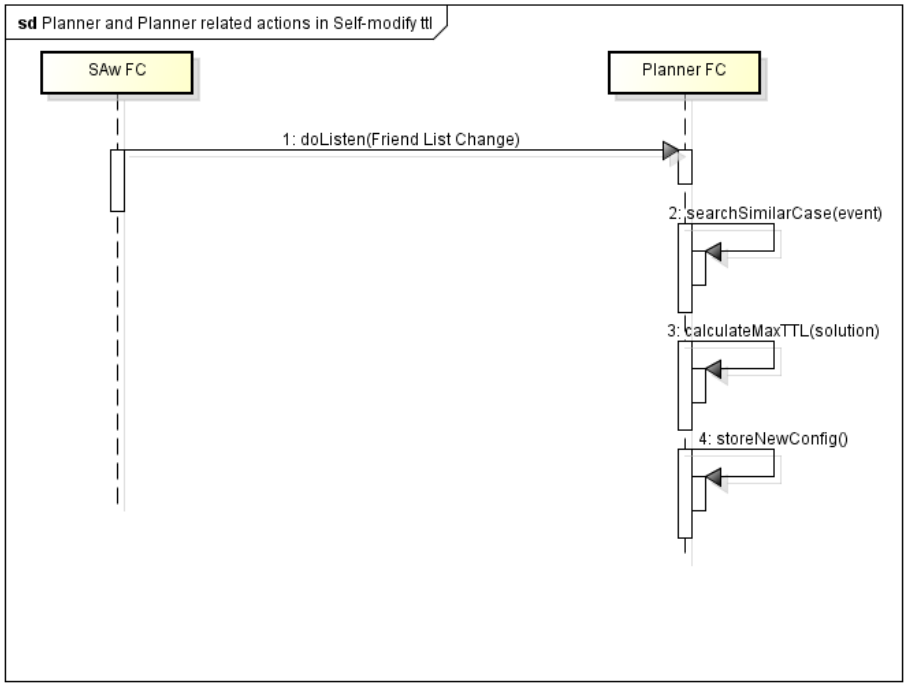

Figure 6: Self-Modification Sequence Diagram.

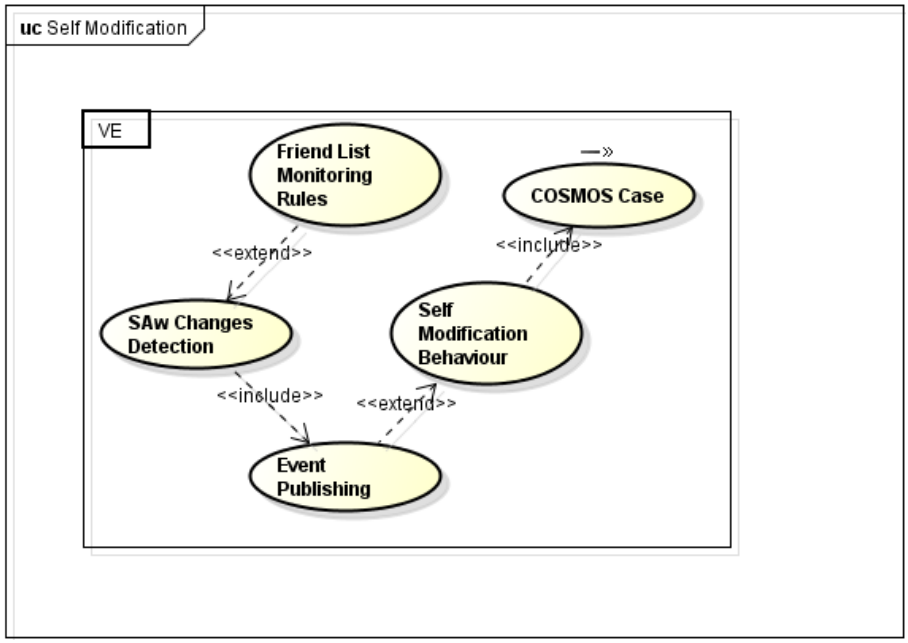

Figure 7: Use-Case diagram for the Self-Modification Scenario.

2.4. Tests of Component's Functions: Stress Tests

The VE specific code is developed having as a basis the Java programming language, in order to ensure that it will be cross platform compatible. Java was also selected since it holds a good balance between the intermediate computational requirements a gateway should possess and the availability of frameworks and technologies for the implementation. **Raspberry Pi 2** [4] was selected as a deployment Testbed since it holds a good balance between cost and gateway capabilities.

Running on Raspberry Pi 2 is the Raspbian OS [5] which is a Linux version for Raspberry based on the ARM hard-float Debian 7 'Wheezy' architecture port. The latest version of the Raspbian OS comes bundled with Java version 1.8. This part of the Testbed will act as the Hardware Gateway of the Flat on which the Application is running and is being operated on by the End User. We are simulating two additional interconnected VEs by using two remote Virtual Machines which have been retrofitted with the required code (application and VE code).

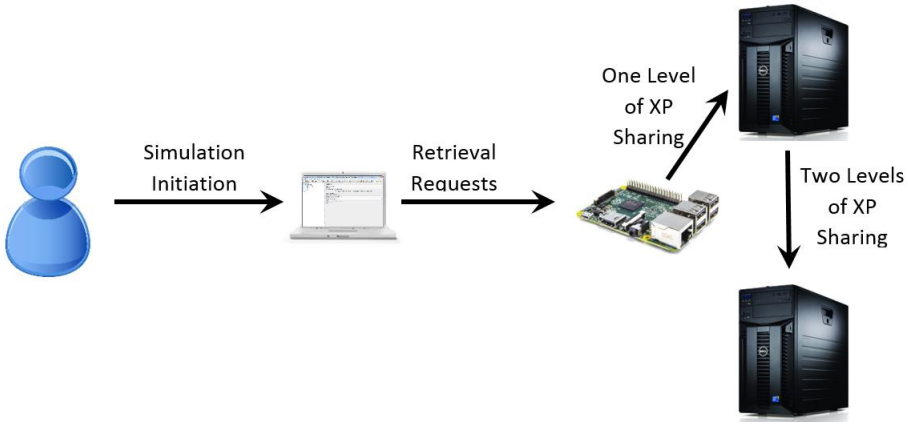

Figure 8: Our Testbed.

Our simulation approach focuses on the ability of the Testbed at the Raspberry side to provide a consistently stable environment for remote VE components being used by an Application to communicate and share their Experiences as well as reason on their stored Knowledge in order to provide answers to each other's queries. Our tests are demonstrating how differentiated workloads of queries may affect VE performance in resource constraint environments. The Raspberry running VE Service is contacted and attempts to locate a Solution to the incoming Problem are made. In the first attempt, the VE searches in its own CB (local retrieval) and, if nothing is found, it contacts the first VM (first level of XP-sharing). The first VM then searches in its own CB and, if nothing is found, it contacts the second VM (recursive Experience Sharing). Of course, the response time to the query is considered to be the overall delay.

The tool used for the simulation is **Apache JMeter®** [6]. It is located in an external workstation dedicated to the metering process and simulates requests from End Users. Our tests are divided into four categories, each one corresponding to a different possible traffic condition of the Network of Things. In all categories, differentiations of query values which lead to XP-Sharing were made in order to test the effect that the depth of the queries has on the retrieval and response times.

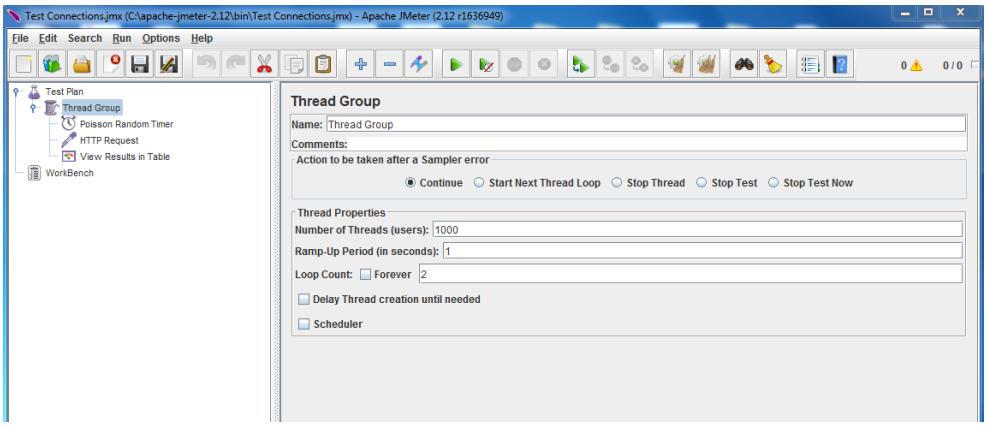

Figure 9: Apache JMeter®

The first category is the **Low Volume** category which is characterized by a single Query Thread running infinite loops, with a constant 10 second delay between calls. Results of the three versions of this category of simulations are demonstrated in Figure 10. In the Low Volume category results in all three versions demonstrate that the lightweight design of our approach can produce response times of less than a second from the initiation of the query to the eventual return of a valid answer.

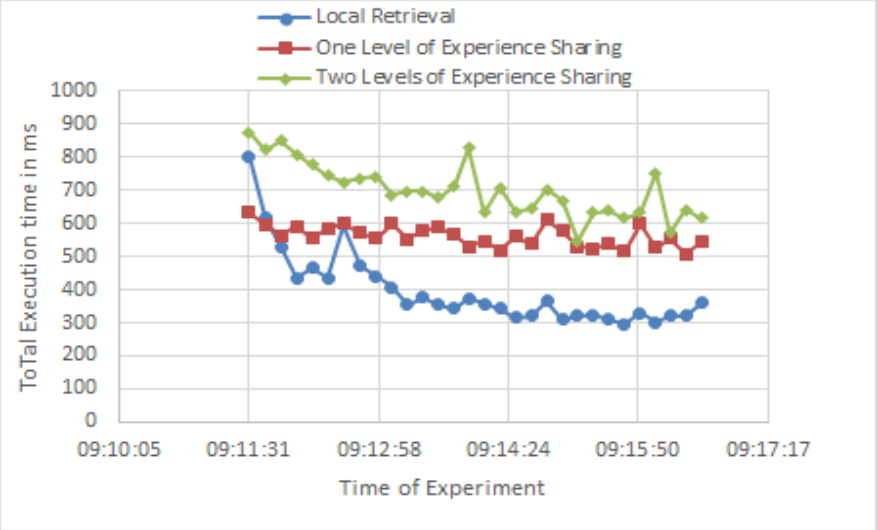

Figure 10: Low Volume Simulations.

The second category is the **Medium Volume** category, which we deem to be the normal/common volume of communications expected by the COSMOS system. In this category of simulations, ten Query Threads are used, starting their operation in intervals of two seconds, with infinite loops for each one and a Poisson timer of query delay with a lambda value of 10,000 ms. These settings lead to an increased load of queries which, due to the nature of Java parallelization in handling incoming HTTP requests, are serviced in less time than the serial arrivals of the Low Volume category. This is the case for all three versions of the simulation. Results are demonstrated in Figure 11.

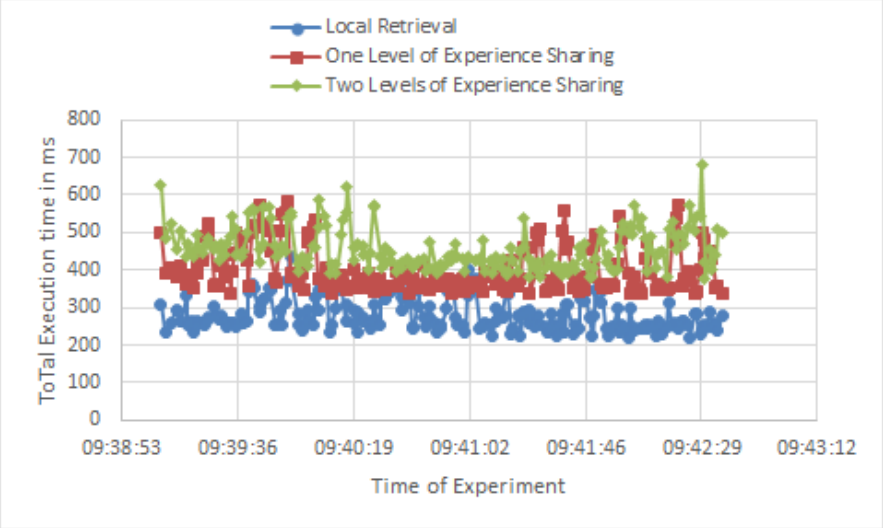

Figure 11: Medium Volume Simulations.

The third category is the **High Volume** category. In this category we used one hundred Query Threads, operational in two second intervals, running infinite loops with a Poisson timed query delay of 5,000 ms. This category initially presents similar response times to the previous two categories, but as more Query Threads come into operation, response times increase (especially after 50 concurrently running Threads) and eventually reach averages of 12 s, 16 s and 17 s. In this category deviation is also increased with results in the third version reaching as high as 28 s and as low as 8 s. Results are demonstrated in Figure 12.

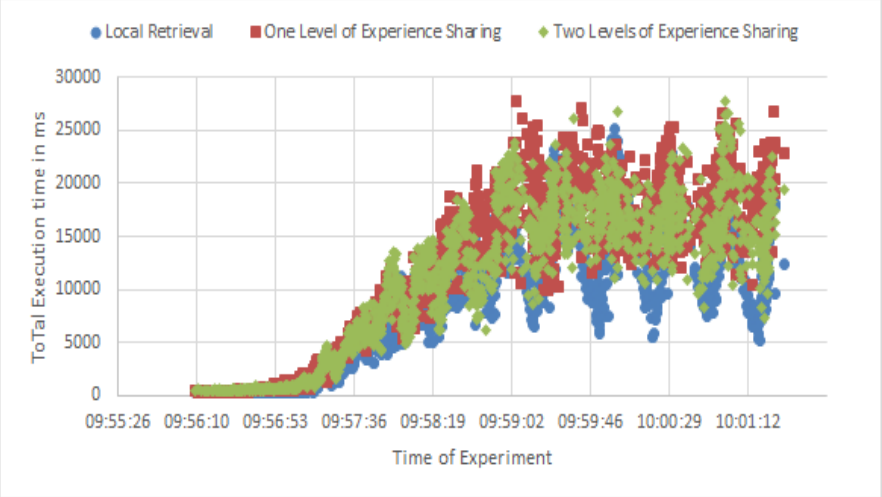

Figure 12: High Volume Simulations.

Finally, we attempted tests in the **DoS Volume** category, which includes the use of 1,000 Query Threads running infinite query loops, starting simultaneously and with no delay between queries. As was expected, response times were greatly increased with the VE responding to queries at an average of 200 s. However, it was proven that the VEs are robust and reliable enough to sustain such an increased loads of operations without denying service. Comparative levels of throughput between categories as measured by JMeter indicate that the request service rate was similar to that of the High volume category. Results are demonstrated in Figure 13. Specifically in this category, the query versions implementing XP-sharing were not used because of permission issues pertaining to the use of the VMs under heavy incoming loads. Therefore only the use of the Raspberry running VE and Application code is demonstrated.

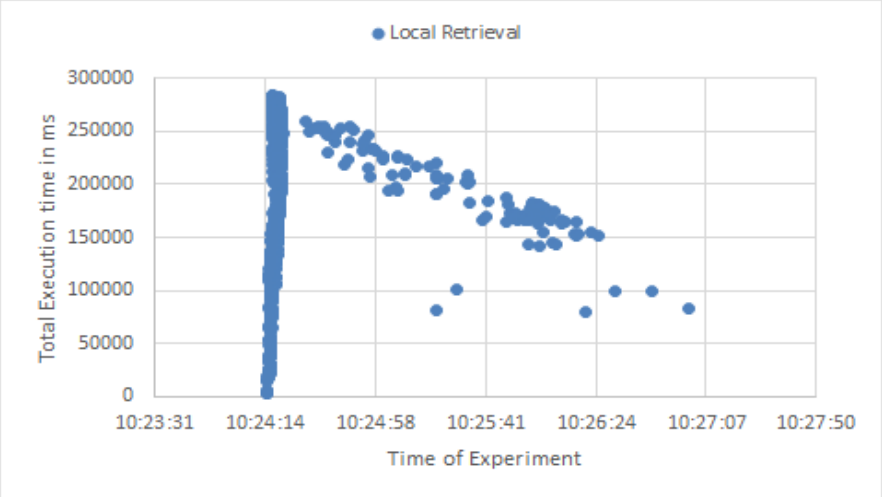

Figure 13: DoS Volume Simulation.

It should be noted that the measurements for XP-sharing of one or two levels are biased, considering the fact that the auxiliary VMs used are not Raspberry Pi 2 themselves. Therefore, further simulations are performed in order to clarify the amount of time differences from the previous measurements. The approach followed is to simulate local search of a Case inside the VM, in order to compare it with the first simulation version of the Low and Medium Volume categories (with no XP-sharing taking place). Also measured is the actual time the VE spends in searching and retrieving the Case. Results of the simulation are demonstrated in Table 1.

Table 1: Raspberry Pi 2 and VM Time Comparisons.

	RASP2	VM
Request Low (ms)	392	201
Search and Retrieval Low (ms)	378	139
Request Medium(ms)	360	212
Search and Retrieval Medium (ms)	346	132

These results demonstrate that while the efficiency of the VM is increased by as much as 68% in internal calculations in comparison to the Raspberry Pi 2, by adding the amount of time the request takes to travel to and from the request originator, the actual simulation gain is a little less than 50% for each VM used. Therefore the results have to be considered under this light.

While these findings, may adversely affect the Application performance under heavier load, the delays in normal and low volume are not greatly affected, considering their low order of magnitude.

2.5. Delivery & Usage

The basic pre-requisite for the Planner FC is JDK 1.8 along with the Jetty Server libraries for RESTful communication between VEs. More specifically, the Jetty version used is jetty-servlet-9.1.5.v20140505.

The functionalities belonging to the Planner are coded as a specific class inside the VE code. Any Applications added to the VE gateway device must be able to integrate themselves in the VE running code. Therefore, any calls from an Application perspective to the Planner will use the internal APIs mentioned in Section 2.2. As for the automatic or Planner internal functionalities, these will be used as needed by the Planner itself, either on demand or continuously. Prerequisites for implementing these functionalities are presented in the following list of libraries that are required:

- Apache Jena version 2.10.0
- Pellet Reasoner
- JSON Simple
- Eclipse Paho MQTT library
- Internal Java libraries (Java version 1.8)

3. The COSMOS T&R model

3.1. Description & Architecture

In Section 8 of D5.1.3 [1] we present the social approach that the COSMOS project introduces in order to achieve enhanced services like discovery, recommendation and sharing between Things enriched with social properties.

The interaction metrics (Shares, Assists and Applauses) monitored by the Social Monitoring component of a VE are stored locally in the corresponding **Followees Lists**. These metrics are calculated in a distributed manner by the VEs on a per-VE basis and are the main input for the services provided by the **Social Analysis** and **Friends Management** components. From these metrics, the **social indexes** of the several kinds of Friends are extracted. These are the **Trust Index**, the **Reputation Index**, the **Reliability Index** and the **Dependability Index**. It is evident that since the social indexes will constantly change, it is quite important to take under consideration their **evolution**. Although a VE may have a low Reputation Index when we study a wide time-window, it may have a much greater Reputation Index when we study a smaller and recent time-window. This means that the specific VE is improving and this improvement should be evaluated fairly by the system and the community. For this reason, for each Followee in each Followees List the **timestamps** (Unix time) and the evaluation of the last e.g. 3-10 interactions may be kept, so that, when applying simple rules, the evolution of the indexes can be studied. All these are illustrated in Figure 14.

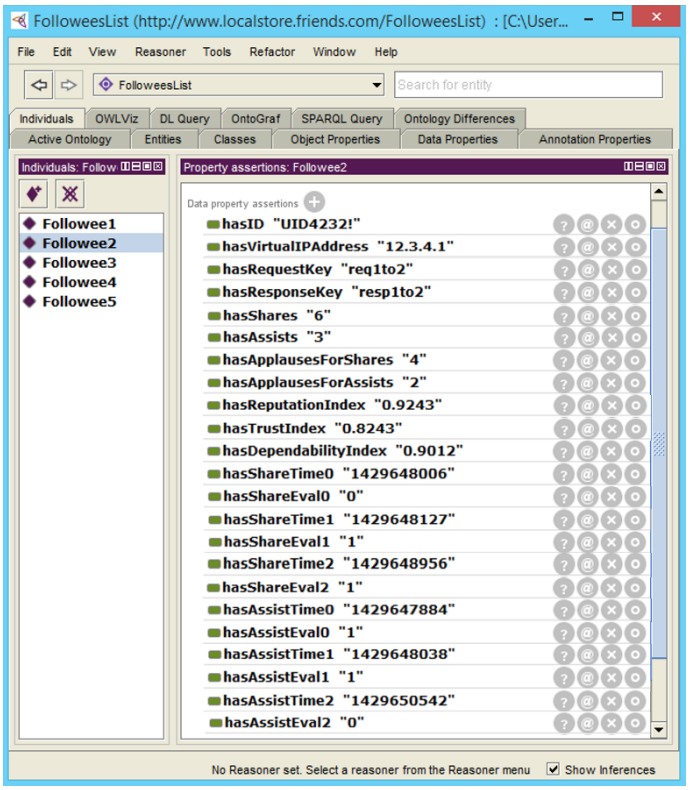

Figure 14: Example of a Followees List.

The basic functionality assisting the Friends Management component is the social characterization of Things. For that reason, COSMOS has introduced a new **Trust & Reputation**

model. The updated version of the specific functions and mathematical expressions used for this model are extensively presented in [1].

3.2. Interfaces & Implementation

The Trust & Reputation model is a functionality working behind the scenes and has already been integrated with the Planner and Social Monitoring code. Having a populated Registry though, could give us the opportunity to visualize the results produced by this model. These results can be presented by our Social App, a java application that clarifies the role of a VE's Social Indexes and enables their calculation. It collaborates with the Followees List of the VE and includes a number of methods that will add on the Social Analysis functionalities, such as the calculation of Trust, Reputation and Dependability. The GUI of the application is shown in Figure 15.

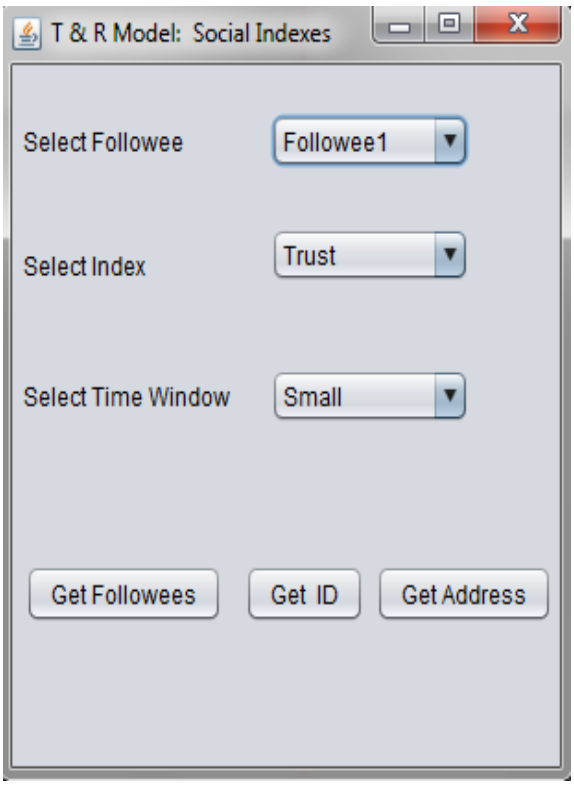

Figure 15: The GUI of our Social App.

When we press the **Get Followees** button, the Followees List is read and **Select Followee** combo box with all the names of the Followees in the list is updated. After selecting a particular Followee we can ask for the calculation of its Trust, Reputation and Dependability through the options of the **Select Index** combo box. All these are depicted in the next figures.

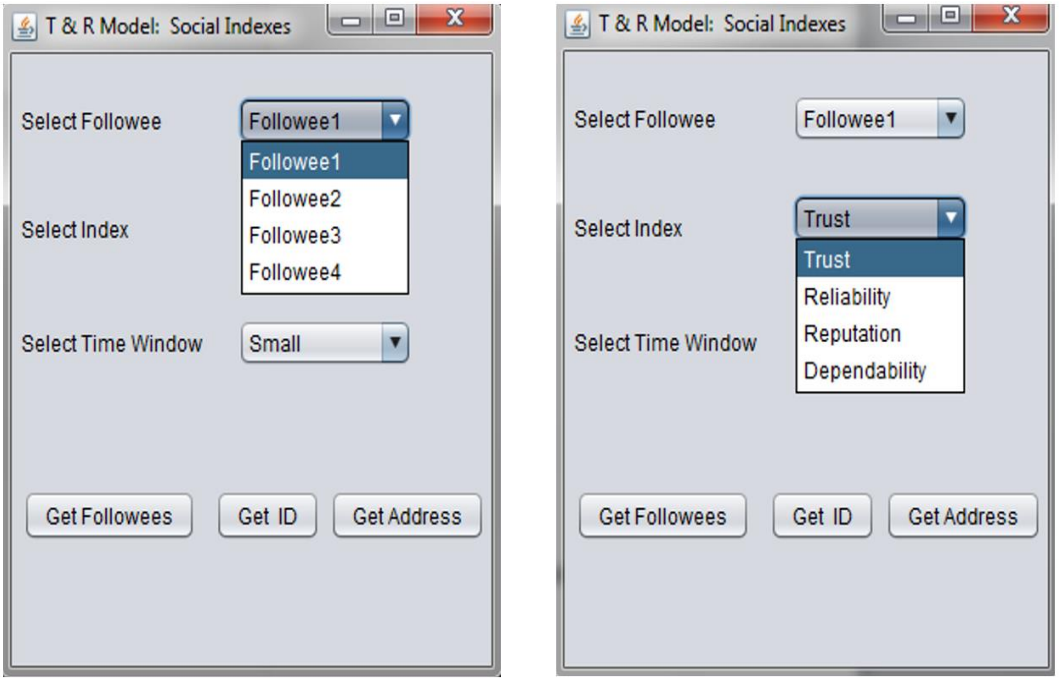

Figure 16: The "Select Followee" and "Select Index" combo boxes.

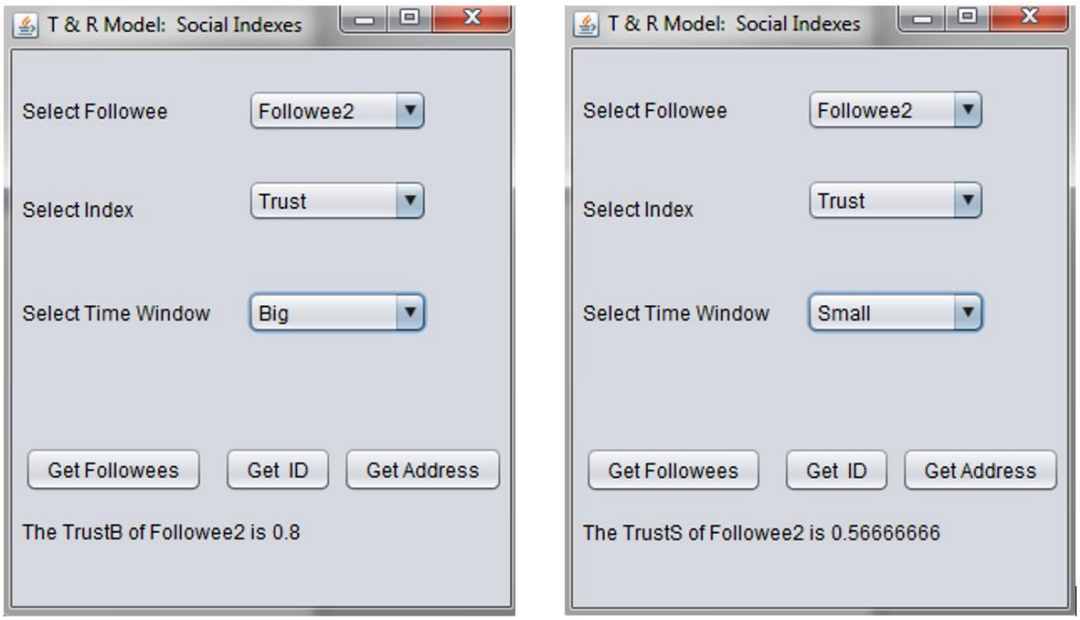

Figure 17: Calculation of the Trust Index of a Followee.

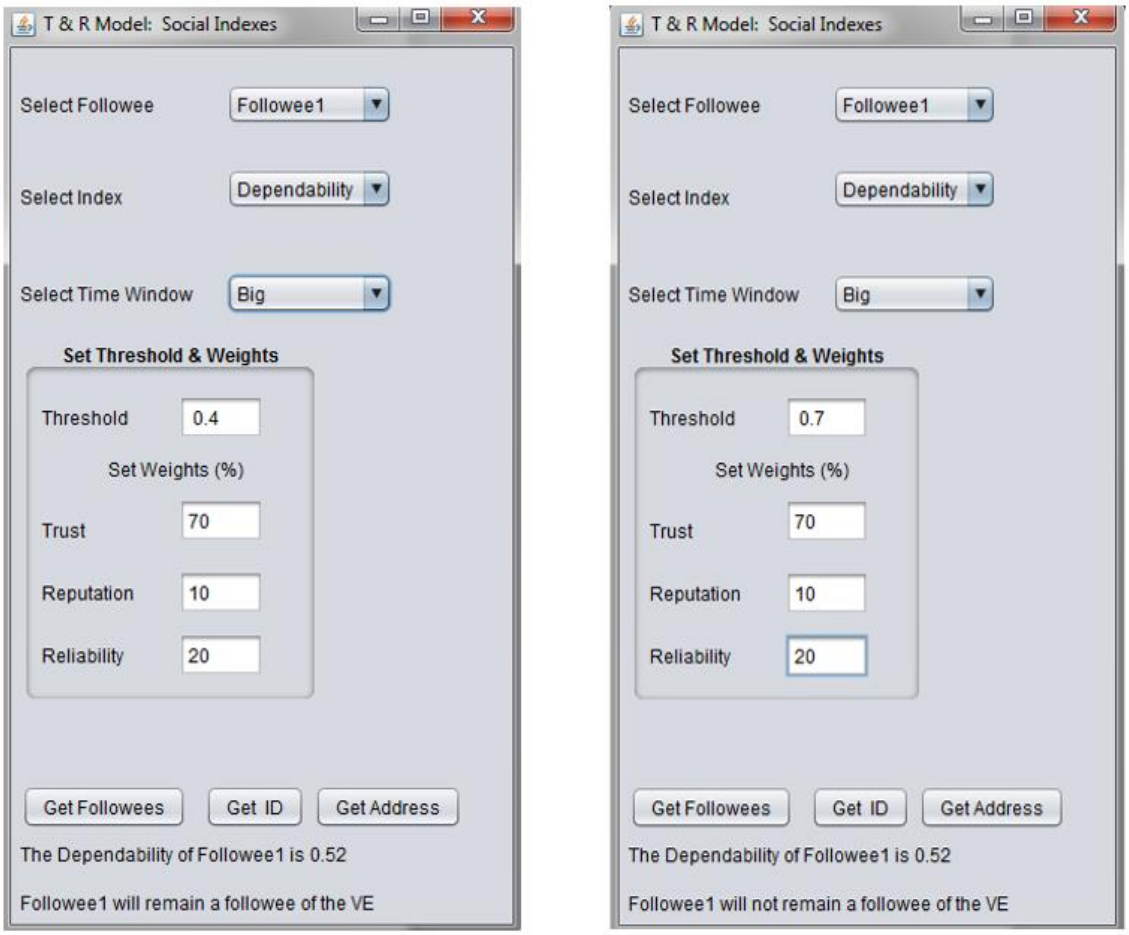

Figure 18: Calculation of the Dependability Index of a Followee.

3.3. Testing our T&R model

3.3.1. TRMSim-WSN

In order to test the updated version of our T&R model, like in Year 2, we used TRMSim-WSN [7], a simulator for T&R models. The TRMSim-WSN is a Java-based T&R models simulator aiming to provide an easy way to test a trust and/or reputation model over WSNs and to compare it against other models.

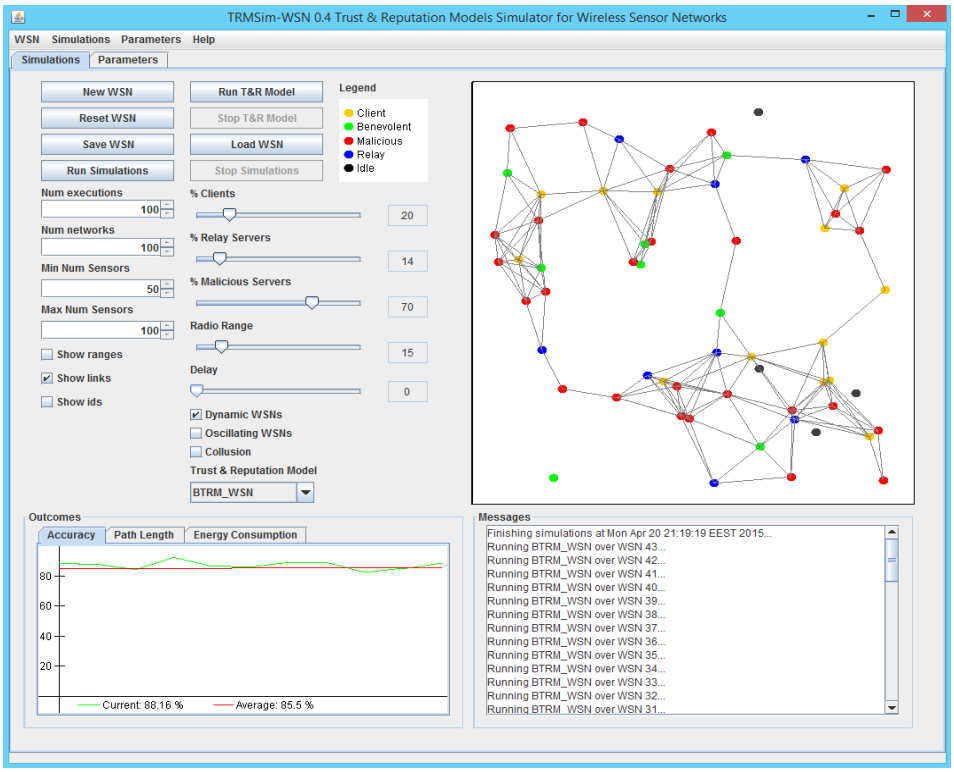

Figure 19: TRMSim-WSN.

The TRMSim-WSN is, as far as we know, the state-of-the-art simulation platform for confidence-renowned systems. It is aimed at simulating algorithms for reputation and trust management in WSN systems, but the same principles can apply to IoT systems in general. The simulation can be run over a single randomly generated WSN or over a set of networks. The user is able to define parameters of the network, such as the percentage of clients and that of malicious nodes. Network topologies may also be loaded from and saved to XML files. Sample trust and reputation models have been included and an API is offered which provides a template for the users to help them easily load new T&R models to the simulator [8].

As shown in Figure 20 the system has a configuration panel. The parameters are:

- **Num executions**: How many times the customers make requests to a particular network.
- **Num networks:** To how many different random networks the simulation will run in order to check the system's behavior. In each network there are Num executions requests for the service from each VE.
- **Min Num Sensors:** The minimum number of VEs per simulated network.
- **Max Num Sensors:** The maximum number of VEs per simulated network.
- **% Clients:** The percentage of VEs that are not providing services.
- **% Relay Servers**: The percentage of VEs that do not provide the particular service (we keep this parameter to 0%).
- **% Malicious Servers:** The percentage of malicious VEs providing services.
- **Radio Range**: The radio range of WSN systems. There is no point of using it in our case.
- **Delay:** Delay time between two successive requests (used for better supervision).
- **Collusion:** When selected, malicious collusions are formed in the network.
- **Oscillating:** When selected, some servers periodically change their behavior from benevolent to malicious and vice versa. However, the rest of the rates (%malicious, %relay, %clients) remain constant.

- **Dynamic:** When selected, some VEs are periodically disabled for 0.5 secs, simulating this way the dynamic activation and deactivation of things / sensors /actuators.

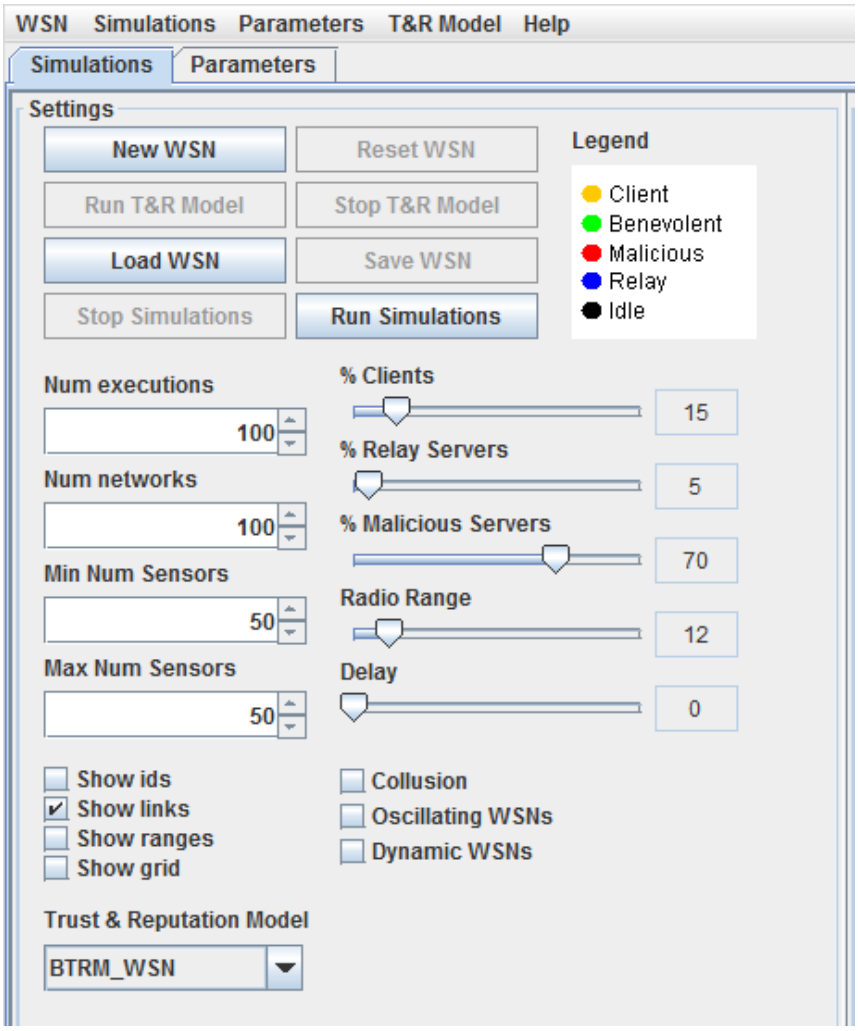

Figure 20: The configuration panel of TRMSim-WSN

As shown in Figure 21, the system has also a panel to represent the VEs and relationships between them. Since the Follower-Followee relationship is not bidirectional arrows (instead of plain lines) are used.

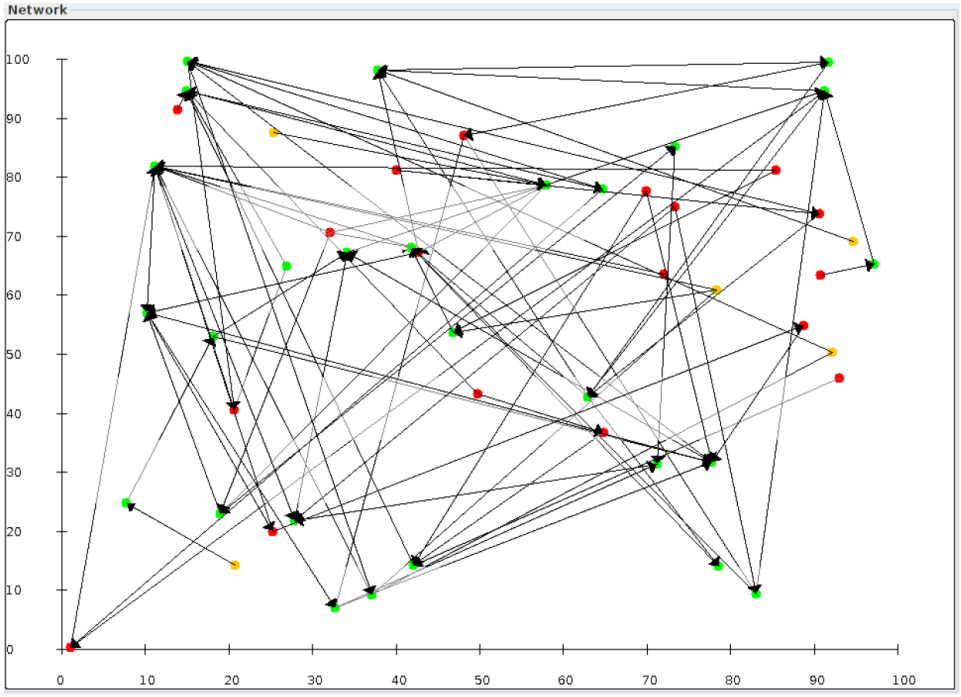

Figure 21: Visualization of the network and VEs' relationships.

Moreover, as shown in Figure 22, the system has a panel to depict the average **satisfaction** of the VEs. When a simulation runs for a given network (*RUN T&R Model button*) then the green line shows the total satisfaction of VEs for each request cycle (e.g. if Num execution = 30 then it has 30 points) while the red line shows the average value. If the simulation runs for several networks (*Run Simulations button*) then the green line shows the total satisfaction in each individual network while the red line shows once more the average value. For example, if Num networks = 100 and Num executions = 30 the green line will have 100 points. Each point will show the current total satisfaction after 30 executions in a random network. As expected, in a network without oscillating and dynamic selection, the first panel will produce the diagram of an increasing function since the networks are isolating malicious VEs.

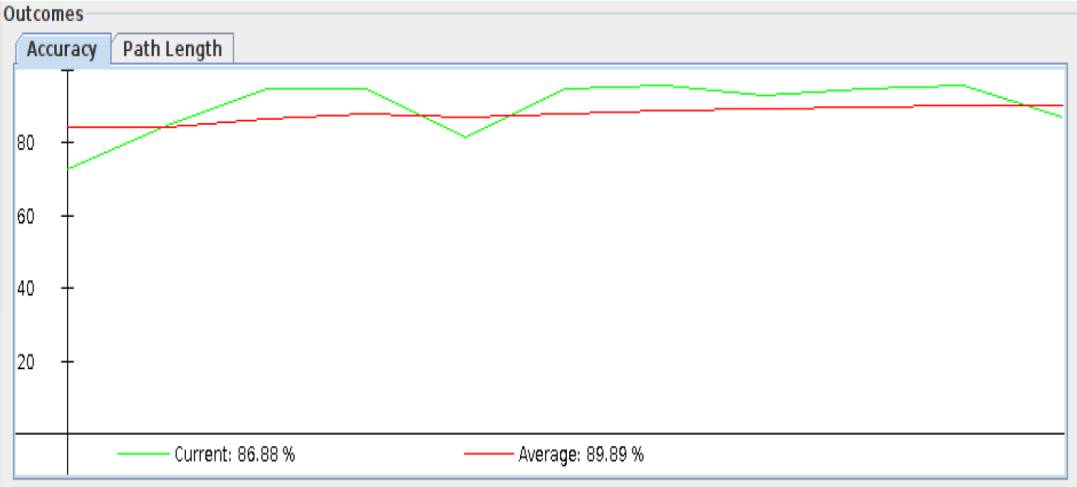

Figure 22: Panel depicting the total current and average satisfaction.

3.3.2. Social Exclusion and Reintegration of VEs

Figure 23 presents in a visual way the social exclusion and reintegration of VEs with oscillating behavior that take place in a network because of TRM-SIoT. The first image presents the initial state of the network where each VE has random friends. In the second image we can see that the network reaches a balance after a few executions. In the next two pictures we can see the social exclusion of ex-benevolent VEs while the last two pictures we can see the social reintegration of VEs when they stop being malicious again.

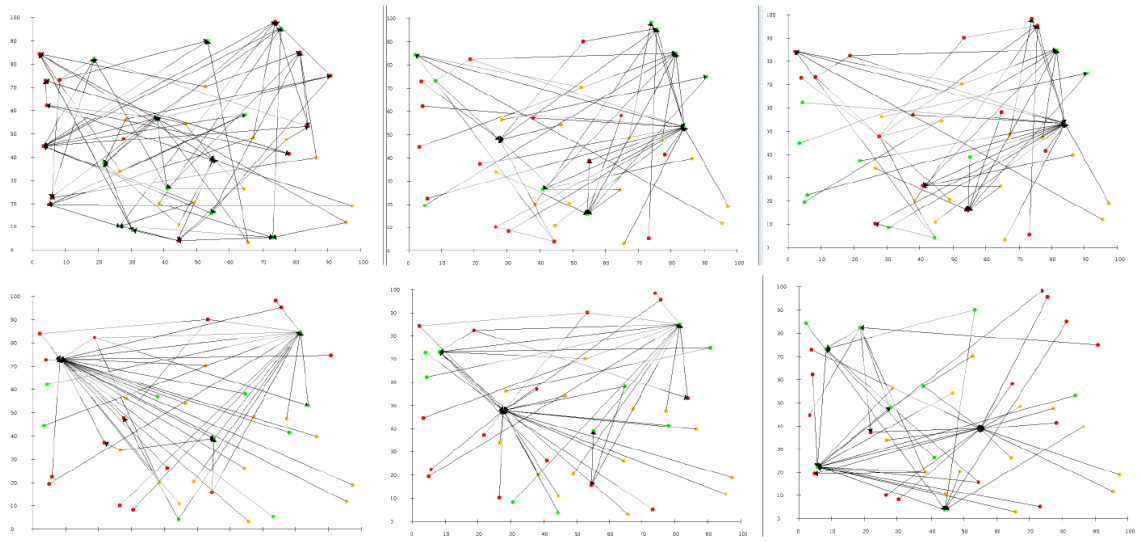

Figure 23: Social Exclusion and Reintegration of VEs.

In order to demonstrate the level of success of the TRM-SIoT and be able to draw the appropriate conclusions, various types of simulations were conducted. We studied and combined three different and independent situations to explore a total of 8 possible states of random networks. The three situations are:

1. Collusion of malicious VEs (col),
2. Oscillating behavior of VEs (osc) and
3. Dynamic networks (dyn).

For all these states, the average total satisfaction of the VEs across 30 to 50 different networks with the same parameters was monitored. The results are depicted in Figure 24.

The conclusions drawn are:

- In a stable system, the TRM-SIoT can provide good levels of security. Even in a network with 90% malicious VEs the average satisfaction is more than 96%.
- In a network with a normal percentage of malicious entities (30% and below) the system manages to provide an average satisfaction greater than 90% even in dynamic networks with malicious collusions and oscillating behavior of VEs (col + osc + dyn).
- In networks with a percentage of malicious entities over 50%, where the platform can make wrong suggestions, the distributed recommendation system manages to keep the network at adequate satisfaction levels (e.g. in a network with 70% of malicious VEs the satisfaction is over 70%).
- Generally, our model provides a satisfaction level greater than that a network with 30% fewer malicious VEs without a T&R model. That means that a dynamic network with 60% malicious entities and the TRM-SIoT to support it is similar to a dynamic network with 30% malicious VEs without a T&R model.

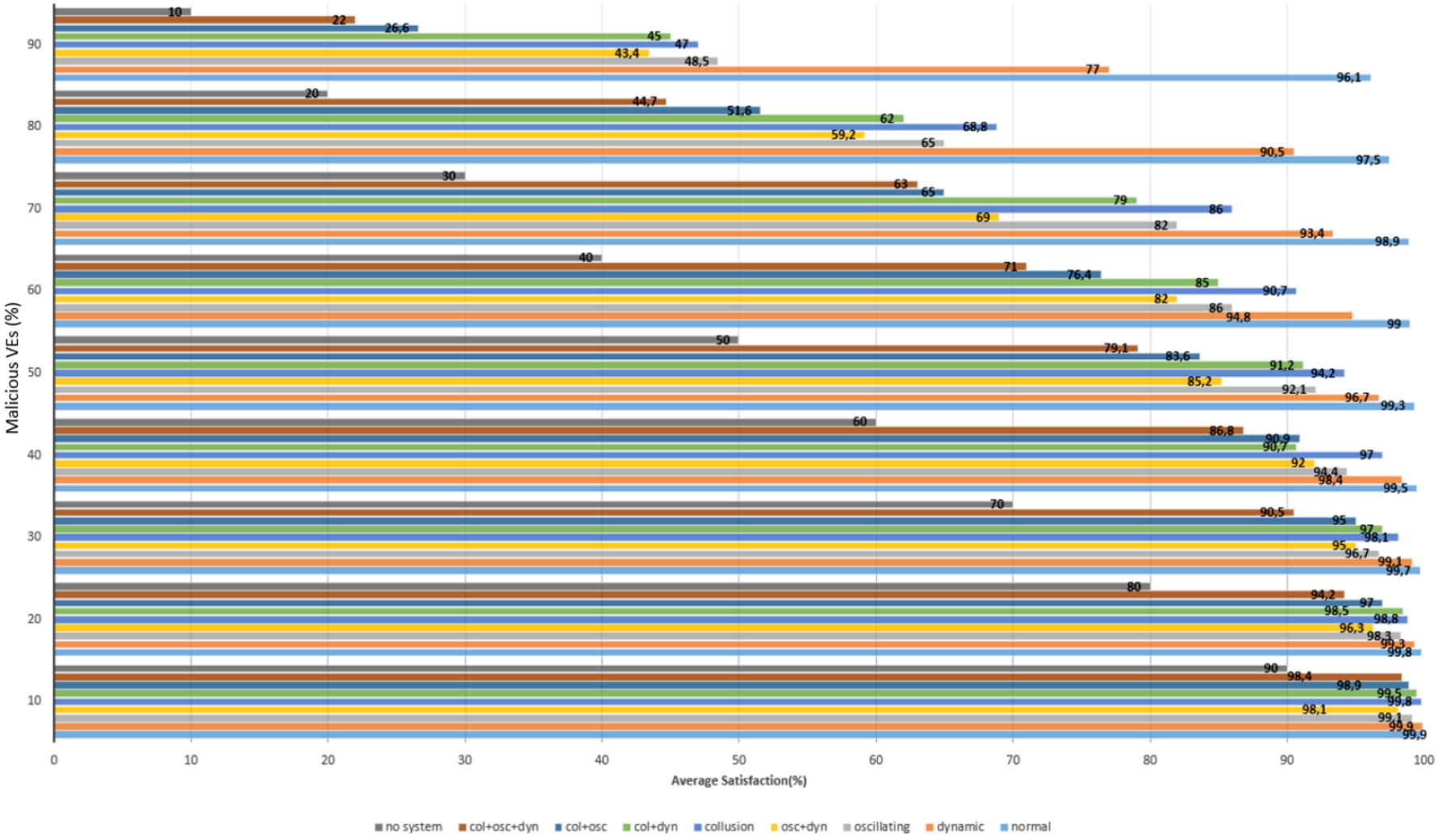

Figure 24: Average Satisfaction for different percentages of malicious VEs in networks with different characteristics.

3.3.3. Comparison of TRM-SIoT with other T&R models

To evaluate our T&R model we compared it with three predominant (as of today) T&R models (Eigentrust, PeerTrust and PowerTrust) as well as with a relatively new system known as BTRM (Bio -Inspired Trust and Reputation Model) that applies a biological algorithm known as Ant-Colony System.

3.3.3.1. Average Satisfaction level

We run simulations both in simple networks and in networks with dynamic entry or oscillating behavior of nodes. Measurements of the **average satisfaction** were made at various percentages of malicious VEs (10%, 50% and 90%). The results are given in Figure 25 and Figure 26.

Although TRM-SIoT has design "constraints" since the VEs do not have a full insight into the behavior of rest of the network, it is evident that its performance is comparable to that of the over models (and in some cases better). So, the combination of a central authority with distributed methods of calculating trust and reputation leads to a model capable of maintaining satisfaction at very good levels.

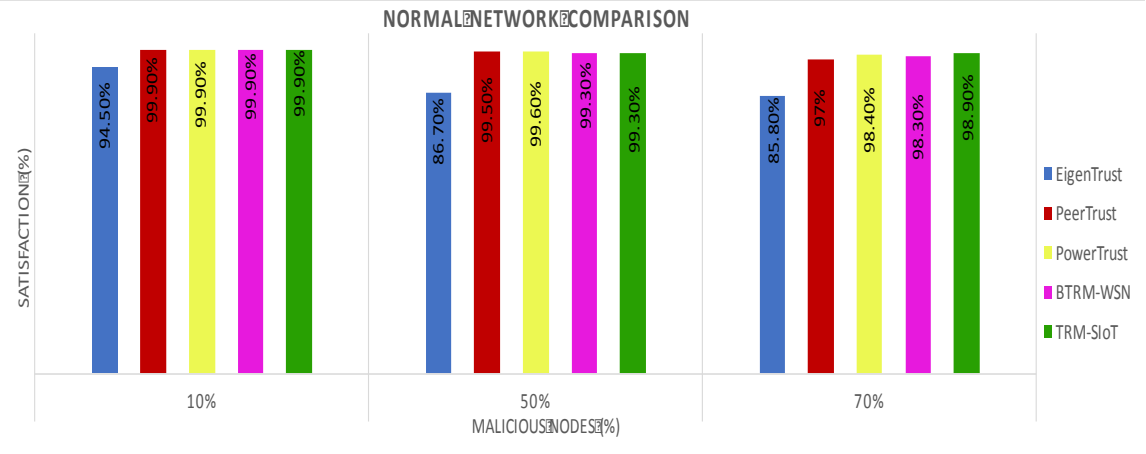

Figure 25: Normal Network Comparison

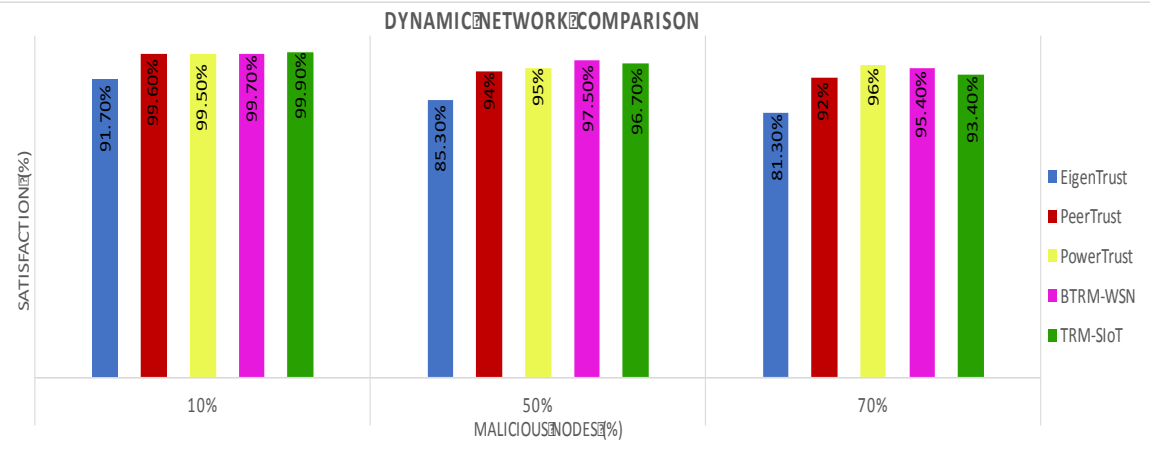

Figure 26: Oscillating Network Comparison

3.3.3.2. Scalability

Regarding the average satisfaction levels, TRM-SIoT is comparable to over models. However, it is completely different from the other models since, due to its design, neither the individual VEs nor the platform have a complete picture of the whole network. This characteristic gives our model a huge advantage regarding **scalability**. Figure 27 shows the scalability of the various models by depicting how many nodes can be used by the simulator while running these models. All measurements were made on a personal computer with Intel Core i7 3630QM and 8GB RAM. A 30 s time limit per transaction cycle (step) was given. We should note that the TRM-SIoT did not reach the 30s/step limit but only the 1s/step. The problem was that it was not possible to create other threads in the system. As it is evident, the TRM-SIoT managed to run simulations with tenfold nodes, while we believe that -with enough memory- it would be possible to run simulations with 250 times more nodes.

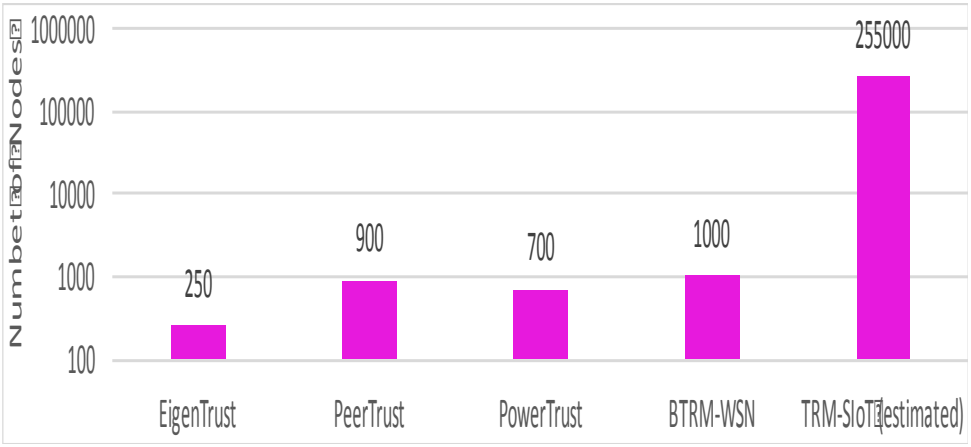

Figure 27: Number of nodes on simulation.

Generally, as it is shown in Figure 28, the time needed during one cycle for each node almost constant. This means that our model in the simulation presents linear scalability.

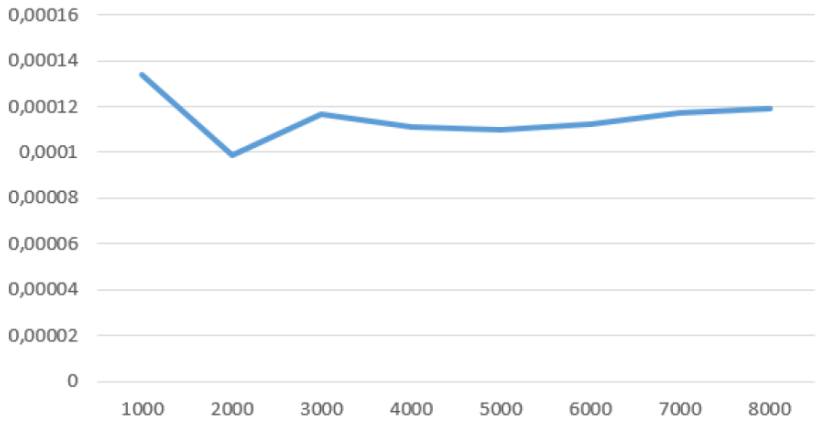

Figure 28: Time/node for one step.

4. The Network Runtime Adaptability Module

4.1. Description & Architecture

The Network Runtime Adaptability module is able to dynamically assign resources performing different activities within COSMOS architecture. In this sense, the key objective of this module is to control the resources usage of every single component. The monitoring of resources usage enables the optimization and prioritization of processes inside a VE. Besides of this, the same functionality can be used in a multi-CPU environment for distributing the computational load thus minimizing the risk of blocking processes.

The enablers for network runtime adaptability are mainly two (but can be extended in the future to more):

- **PM2** – This is an open source framework that offers process launching and monitoring in machines. It is capable of reporting real time resources consumption and provides functionalities for correcting misbehaviors in real time. However, its logic is quite simple and additional intelligence is required in order to take fully advantage of its benefits.
- **µCEP** – This additional intelligence is added by the µCEP. The results from monitoring are used as a data-source for the µCEP and can be combined with many other sources, leading to an optimal decision making module.

The flowchart is shown in the next figure.

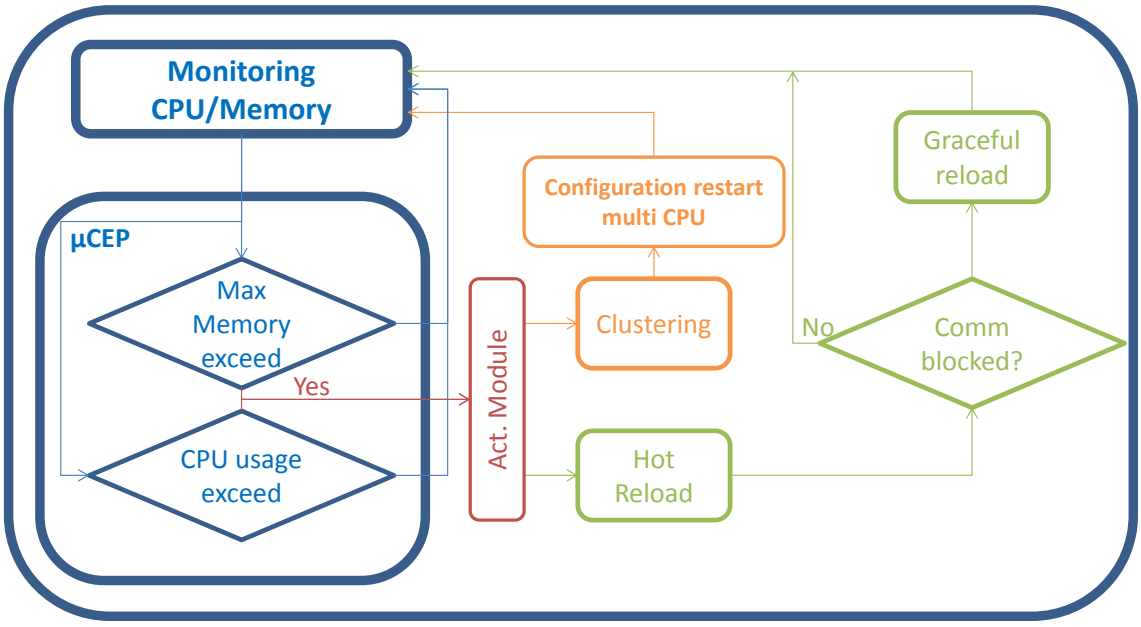

Figure 29: Network runtime adaptability functional architecture

The Network Runtime Adaptability module uses many different functional components of the COSMOS architecture. It contains an actuation module that can be represented by IoT-services and services of VEs or cloud based components. Its functionality is to “transform” Complex Events into actuations. These actions may be based on PM2 or µCEP or be service oriented actuations.

In Figure 30 below the different COSMOS functional components that can participate in the Network Runtime Adaptability are depicted (the components that are inside red squares).

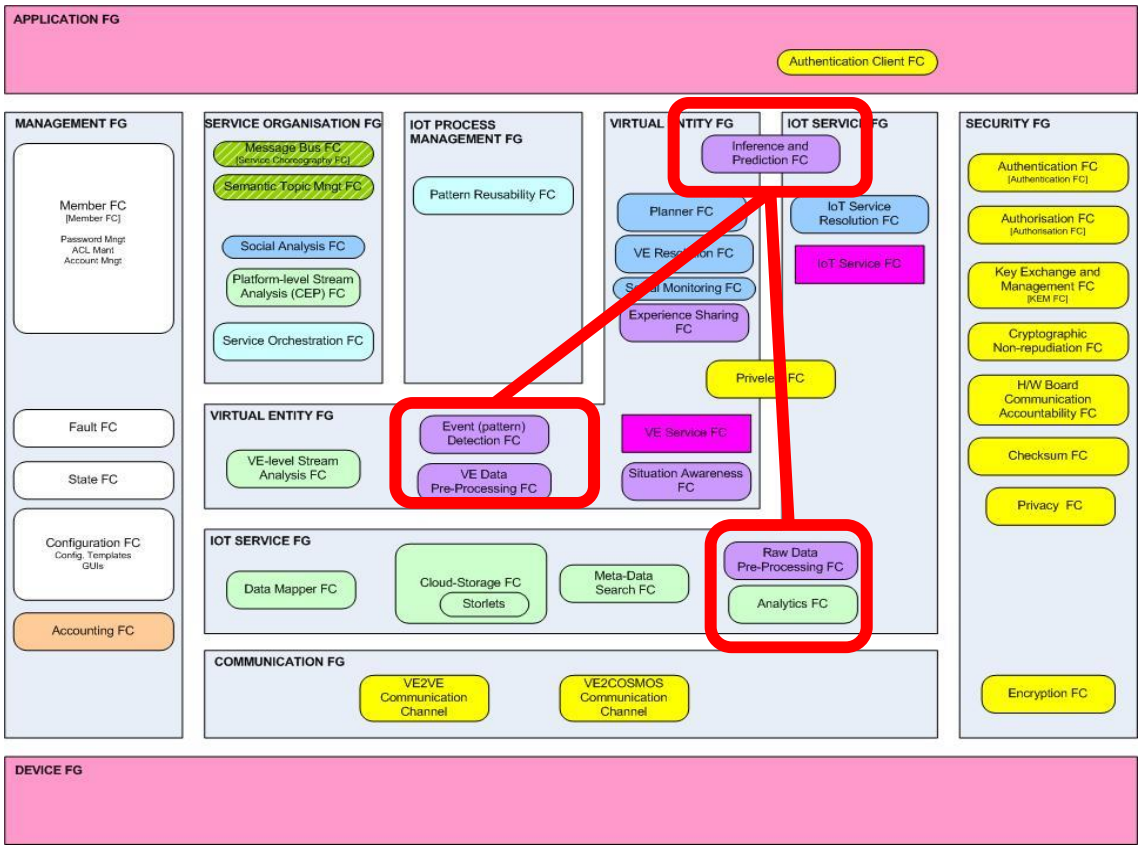

Figure 30: COSMOS FCs participation in Network Runtime Adaptability.

The roles of the FCs that participate in the process are:

- Inference and prediction FC – This module completes the overall picture of system needs by providing relevant input for making decisions related to the resources allocation for each process.
- Raw-data pre-processing FC and Analytics FC – The preprocessing of raw data could also be an input for the decision making processes and offer capabilities for identifying the business logic that has to be implemented.
- VE Data pre-processing FC – It works in a similar way with the two previous FCs and injects data that are considered in the execution of business logic in the µCEP.
- Event pattern detection FC – This is where logic is applied for the identification of the actions that should be taken. In this sense, the module will link complex events to specific adaptive actions.

4.2. Interfaces & Implementation

As it is presented above, the implementation of the Network Runtime Adaptability is based on two main components, the µCEP and PM2. The following subsections provide some implementation details regarding them.

4.2.1. µCEP REST interface Methods

µCEP implements a REST interface enabling the access to the functionalities provided by the CEP. Some methods that can be used to enable the interaction and modification of parameters, business logic and behavior are:

Get DOLCE Rules files

	Get DOLCE Rules files
Description	Gets the list of DOLCE Rules files being processed in a CEP instance
URL	**BASE_URL**/uCEP/dolceRuleSpecification
Method	GET
Input	-
Output	{ "dolceSpecifications": { "dolceSpecification": [{ "-id": "detect" }] } }

Get a DOLCE Rules file details by the name

	Get a DOLCE Rules file
Description	Gets the DOLCE Rules file details by the given name
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}
Method	GET
Input	-
Output	{ "dolceSpecification": { "complex": [{ "definition": "payload { int SensorID = sensor_id, int Valor = value };\t\tdetect temperature;", "-id": "SiempreActivo" }, { "definition": "payload { int SENSORID = sensor_id, int Valor = value }; detect temperature where (count(temperature) > 1) in [T_WINDOW];", "-id": "hot" }], "event": [{ "definition": "use { int sensor_id, int value };\t\taccept { value > 39 };", "-id": "temperature" }], "external": [{ "name": "TEMP_ALERT", "value": "39", "type": "int" }, { "name": "T_WINDOW", "value": "1 seconds", "type": "duration" }], "-id": "detect" } }

Add or Modify a DOLCE Rules file

	Add or Modify a DOLCE Rules file
Description	Add a new DOLCE Rules file or modigy an existing one by the given name
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}
Method	PUT
Input	{ "dolceSpecification": { "complex": [{ "definition": "payload { int SENSORID = sensor_id, int Valor = value }; detect temperature where (count(temperature) > 1) in [T_WINDOW];", "-id": "hot" }], "event": [{ "definition": "use { int sensor_id, int value };\t\taccept { value > 39 };", "-id": "temperature" }], "external": [{ "name": "TEMP_ALERT", "value": "39", "type": "int" }, { "name": "T_WINDOW", "value": "1 seconds", "type": "duration" }], "-id": "detect" } }
Output	{ "data": { “success": "true" } }

Delete a DOLCE Rules file

	Delete a DOLCE Rules file
Description	Delete the DOLCE Rules file by the given name
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}
Method	DELETE
Input	-
Output	{ "data": { “success": "true" } }

Get the External declarations

	Get external declaration
Description	Gets all external declarations in a DOLCE prgram
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}/external
Method	GET
Input	-
Output	{ "dolceSpecification": { "external": [{ "name": "TEMP_ALERT", "value": "39", "type": "int" }, { "name": "T_WINDOW", "value": "1 seconds", "type": "duration" }], "-id": "detect" } }

Add/Modify an external declaration into a DOLCE Rules file

	Add/Modify an external declaration into a DOLCE Rules file
Description	Add a new external declaration or modify an existing one into a DOLCE Rules file
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}/external
Method	PUT
Input	{ "name": "threshold", "value": "50", "type": "int" }
Output	{ "data": { "success": "true" } }

Delete an external declaration into a DOLCE Rules file

	Delete an external declaration into a DOLCE Rules file
Description	Delete an external declaration into a DOLCE Rules file by the given name
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}/external/**{external-id}**
Method	DELETE
Input	-
Output	{ "data": { "success": "true" } }

Get the Event declarations

	Get Event declaration
Description	Gets all event declarations in a DOLCE prgram
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}/event
Method	GET
Input	-
Output	{ "dolceSpecification": { "event": [{ "definition": "use { int sensor_id, int value }; accept { value > 39 };", "-id": "temperature" }], "-id": "detect" } }

Add/Modify an event declaration into a DOLCE Rules file

	Add/Modify an event declaration into a DOLCE Rules file
Description	Add a new event declaration or modify an existing one into a DOLCE Rules file
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}/event
Method	PUT
Input	{ "definition": "use { int sensor_id, int value }; accept { value > 66 };", "-id": "myEvent" }
Output	{ "data": { “success": "true" } }

Delete an event declaration into a DOLCE Rules file

	Delete an event declaration into a DOLCE Rules file
Description	Delete an event declaration into a DOLCE Rules file by the given name
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}/event/**{event-id}**
Method	DELETE
Input	-
Output	{ "data": { “success": "true" } }

Get the Complex declarations

	Get Event declaration
Description	Gets all complex declarations in a DOLCE program
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}/complex
Method	GET
Input	-
Output	{ "dolceSpecification": { "complex": [{ "definition": "payload { int SensorID = sensor_id, int Valor = value }; detect temperature;", "-id": "SiempreActivo" }, { "definition": "payload { int SENSORID = sensor_id, int Valor = value }; detect temperature where (count(temperature) > 1) in [T_WINDOW];", "-id": "Calooooooor" }], "-id": "detect" } }

Add/Modify a Complex declaration into a DOLCE Rules file

	Add/Modify an event declaration into a DOLCE Rules file
Description	Add a new complex declaration or modify an existing one into a DOLCE Rules file
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}/complex
Method	PUT
Input	{ "definition": "payload { int SENSORID = sensor_id, int Valor = value }; detect temperature where (count(temperature) > 20);", "-id": "opc" }
Output	{ "data": { "success": "true" } }

Delete a Complex declaration into a DOLCE Rules file

	Delete a Complex declaration into a DOLCE Rules file
Description	Delete a Complex declaration into a DOLCE Rules file by the given name
URL	**BASE_URL**/uCEP/dolceRuleSpecification/{**dolceProgram-id**}/complex/**{event-id}**
Method	DELETE
Input	-
Output	{ "data": { "success": "true" } }

4.2.2. PM2 deployment

PM2 is an open source component that can be found in GitHub [9] and which is deployed in many different machines. In this case, the implementation has been done over a Raspberry Pi board. The module also provides an API that facilitates the integration of the multiple components required for Network Runtime Adaptability and also some additional functionalities that can be further analyzed in the rest of the section.

Startup script

One of the most important aspects of PM2 is that it simplifies the monitoring activities. For initializing a module or restarting in a safe way, it is mandatory to follow its guidelines. The available actions that can be taken are the following:

- Declaring environment variables that depend on the deployment.
- Embedding applications.
- Using JSON as default.
- Deploying code via ssh.
- Connecting to external repositories.

Monitoring CPU/Memory

PM2 offers a simple way to monitor the resources usage of an application. For example, memory and CPU usage can be monitored very easily straight from the terminal by using:

```
$ pm2 monit
```

Clustering

The **cluster mode** allows the application to be scaled across all available CPUs and to be updated without any downtime. This characteristic is really important for cloud based approaches and distributed environments. Some remarkable aspects of this feature are:

- Code does not need to be modified.
- It is possible to use multiple listening ports for assuring zero downtime while clustering reconfiguration takes place.
- It is possible to share states among sessions by using proper technologies such as using Redis [10] or MongoDB [11].

The implementation looks is presented below:

```
$ pm2 start app.js -i 1
```

Multiple values to the instances (-i) option can be passed:

```
# Start the maximum processes depending on available CPUs
$ pm2 start app.js -i 0

# Start the maximum processes -1 depending on available CPUs
$ pm2 start app.js -i -1

# Start 3 processes
$ pm2 start app.js -i 3
```

Max Memory Restart

PM2 allows restarting an application based on a memory limit, this is a feature that does not allow further processing, so this can be further combined with µCEP rules and have this as safe restart. The way to implement the restart is the following.

Table 2: Max Memory Restart implementation

Technology	Implementation
CLI	`$ pm2 start big-array.js --max-memory-restart 20M`
JSON	`{ "name" : "max_mem", "script" : "big-array.js", "max_memory_restart" : "20M" }`
Programmatic	`pm2.start({ name : "max_mem", script : "big-array.js", max_memory_restart : "20M" }, function(err, proc) { // Processing });`

Hot reload or 0 sec reload

In contrast with **restart** which kills and restarts a process, reload achieves a 0-second-downtime reload. This feature only works for apps in **cluster mode** that use HTTP/HTTPS/Socket connections. The command for reloading an app is:

```
$ pm2 reload api
```

In case of unsuccessful reload, a standard restart process will take place.

Graceful reload

The external control of the system can sometimes suffer a very long reload or a reload failure meaning that the app still has open connections to exit. In this case, Graceful Reload handles the problem. This feature is a mechanism that sends a **shutdown** message to processes before reloading them. It is possible to control the time that the app requires to shutdown via the PM2_GRACEFUL_TIMEOUT environment variable.

For example:

```
process.on('message', function(msg) {
  if (msg == 'shutdown') {
    // Your process is going to be reloaded
    // You have to close all database/socket.io/* connections
    console.log('Closing all connections...');

    // You will have 4000ms to close all connections before
    // the reload mechanism will try to do its job
    setTimeout(function() {
      console.log('Finished closing connections');
      // This timeout means that all connections have been closed
      // Now we can exit to let the reload mechanism do its job
      process.exit(0);
    }, 1500);
  }
});
```

Then the following command has to be used:

```
$ pm2 gracefulReload [all|name]
```

When PM2 starts a new process to replace an old one, it will wait for the new process to begin listening to a connection or a timeout before sending the shutdown message to the old one.

Programmatic API

To sum up, the key methods that are used are:

Method name	API
Connect/Launch	pm2.connect(fn(err){})
Disconnect	pm2.disconnect(fn(err, proc){})
Options	nodeArgs(arr), scriptArgs(arr), name(str), instances(int), error(str), output(str), pid(str), cron(str), mergeLogs(bool), watch(bool), runAsUser(int), runAsGroup(int), executeCommand(bool), interpreter(str), write(bool)
Restart	pm2.restart(proc_name|proc_id|all, fn(err, proc){})
Stop	pm2.stop(proc_name|proc_id|all, fn(err, proc){})
Delete	pm2.delete(proc_name|proc_id|all, fn(err, proc){})
Reload	pm2.reload(proc_name|all, fn(err, proc){})
Graceful Reload	pm2.gracefulReload(proc_name|all, fn(err, proc){})
List	pm2.list(fn(err, list){})
Describe process	pm2.describe(proc_name|proc_id, fn(err, list){})
Dump (save)	pm2.dump(fn(err, ret){})
Flush logs	pm2.flush(fn(err, ret){})
Reload logs	pm2.reloadLogs(fn(err, ret){})
Send signal	pm2.sendSignalToProcessName(signal,proc,fn(err, ret){})
Send signal	pm2.sendSignalToProcessName(signal,proc,fn(err, ret){})
Generate start script	pm2.startup(platform, fn(err, ret){})
Kill PM2	pm2.killDaemon(fn(err, ret){})

4.3. Packaging

In an attempt to ease the deployment of the presented functionalities, we use container technology to package and deliver the µCEP along with PM2. Moreover, by using containers we are able to perform rollout deployments in different hardware architectures, so in this sense we can reach a wider range of IoT devices. The following sections describe the process to prepare a container for a Raspberry Pi device.

Docker Installation in Raspberry Pi

Docker containers have been chosen in order to get a high grade of versatility and easy deployment on IoT devices. These containers contain the necessary yet sufficient processes, libraries and dependencies to run properly a task.

Docker is coded in GO language, so the first step consists in installing it in the Raspberry Pi. Given that some Raspbian packages –the Operating System running in the device– may not be up to date, we are going to compile GO from scratch. In fact, this is the best option in order to have the latest version and to avoid problems with the particularity of ARM microprocessors, which are the ones empowered in the *Pi*.

```
$ mkdir go ; cd go
$ curl http://dave.cheney.net/paste/go-linux-arm-bootstrap-c788a8e.tbz | tar xj
$ curl https://storage.googleapis.com/golang/go1.5.src.tar.gz | tar xz
$ ulimit -s 1024
$ cd src
$ env GO_TEST_TIMEOUT_SCALE=10 GOROOT_BOOTSTRAP=$HOME/go-linux-arm-bootstrap ./all.bash
```

Then, download and compile the source code of Docker.

```
$ mkdir docker_for_rpi ; cd docker_for_rpi
$ git clone https://github.com/hypriot/rpi-docker-builder.git
$ cd rpi-docker-builder
$ sudo sh run-builder.sh  #(This step takes several minutes)
```

Now the Raspberry Pi is allowed to run ARM-images on Docker Containers.

Docker Containers

Docker Containers are a novel approach to Continuous Integration and Continuous Deployment practices. Whenever a new application version is released, the only step needed to provision that new version in a device is to update the Docker Image; all the libraries and dependencies are enclosed in that image, thus highly increasing the reliability of the deployment and minimizing the impact of misbehaviour.

In this new Docker Container we include both the µCEP and PM2. This way, we are providing application developers with a powerful mechanism to monitor what is going on inside de container in a seamless manner through a web service, for instance to check the status of the µCEP, to display CPU and memory usage, or to stop, start, restart, reload the µCEP if required.

In the following next steps we are going to download and setting up the µCEP service. Currently two versions are available, one uses raw UDP connections and the other features an MQTT client to interact with an MQTT Message Broker.

- ***Setting-up µCEP UDP***

First, we get the latest source code from the ATOS GitLab repository.

```
$ mkdir rpi-ucep-udp ; cd rpi-ucep-udp
```

```
$ git clone -b docker_raspberry_UDP_#61
https://gitlab.atosresearch.eu/ari/bcep.git
```

```
$ cd bcep
```

Now we can edit the Dolce Rules file in the /source folder to apply a certain behaviour. Once done, the next commands build and start the service.

```
$ docker build -t iotcosmos/rpi-ucep-udp ./
```

```
$ docker run -d -p 8088:8088 -p 29654:29654/udp -p 50000:50000/udp --
name rpi-ucep-udp iotcosmos/rpi-ucep-udp
```

The last command interfaces three ports from within the container and the host machine. Ports 29654/UDP and 50000/UDP will be used by the µCEP while port 8088/TCP will be used for PM2-WEB administration.

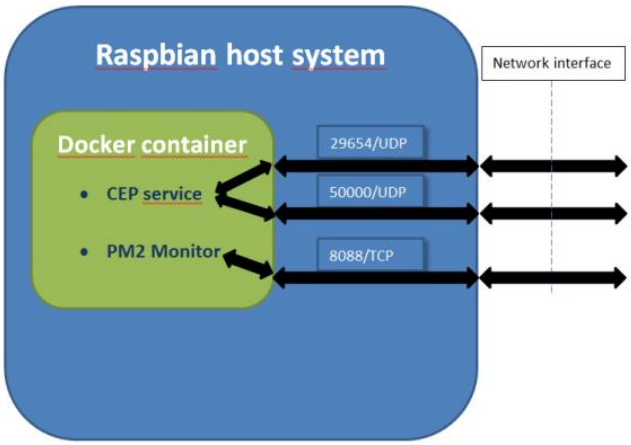

Figure 31: Ports used in µCEP UDP

- ***Setting-up µCEP MQTT***

First, we get the latest source code from the ATOS GitLab repository.

```
$ mkdir rpi-ucep-mqtt ; cd rpi-ucep-mqtt
```

```
$ git clone -b docker_raspberry_MQTT_#61
https://gitlab.atosresearch.eu/ari/bcep.git
```

Now we can edit the Dolce Rules file in the /source folder to apply a certain behaviour. This configuration has the particularity of involving two Docker Containers rather than just one. On one of them we will run a Mosquitto service, which works as a bridge between µCEP and third-party systems through MQTT pub/sub messaging.

```
$ cd bcep/mosquitto
```

```
$ docker build -t  iotcosmos/rpi-mosquitto ./
```

```
$ docker run -p 1883:1883 --name rpi-mosquitto -d iotcosmos/rpi-
mosquitto
```

The other container includes the MQTT version of the µCEP.

```
$ cd ..
$ docker build -t  iotcosmos/rpi-ucep-mqtt ./
$ docker run -d -p 8088:8088 --name rpi-ucep-mqtt iotcosmos/rpi-ucep-mqtt
```

This time only two ports will be used: port 1883/TCP is used for MQTT messaging service, so to send and receive events and complex events, respectively, to/from the µCEP; port 8088/TCP will be used for PM2-WEB administration.

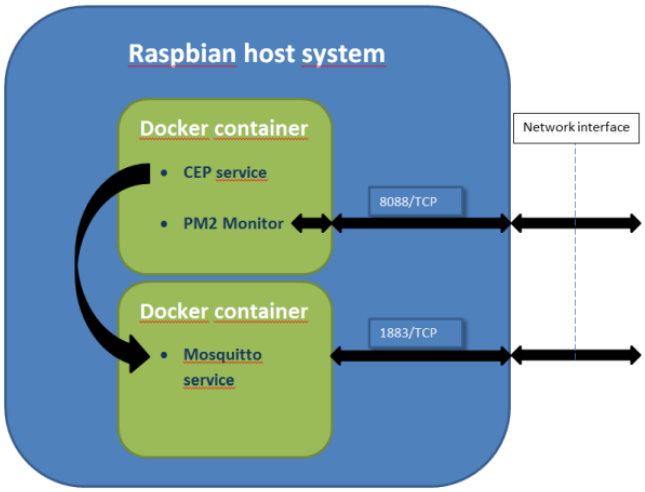

Figure 32: Ports used in µCEP MQTT

5. Conclusions

In this document, we described different components which were developed and implemented in order to fulfil the objectives of the work package. The functionalities offered by the different components are generic in nature and can be applied to different use case scenarios.

Each component has been implemented independently, and most components have also been integrated within WP5 (e.g. the Planner and Social Monitoring and Social Analysis components). This work will form the basis of our demonstrations for the end of Year 3.

Special attention was given during this year in testing our components by using various tools and simulators. Of course, the next step is to evaluate our functional components and architecture in real conditions.

6. References

[1] D5.1.3 Decentralized and Autonomous Things Management: Design and Open Specification (Final)

[2] D4.1.3 Information and Data Lifecycle Management: Design and Open Specification (Initial)

[3] R. L. Mantaras, D. McSherry, D. Bridge, D. Leake, et al; "Retrieval, reuse, revision, and retention in case based reasoning"; The Knowledge Engineering Review, Vol. 00:0, 1–2, 2005

[4] Raspberry Pi: https://www.raspberrypi.org/

[5] Raspbian: https://www.raspbian.org/

[6] Apache Jmeter®: http://jmeter.apache.org/

[7] F. G. Marmol and G. M. Perez, "TRMSim-WSN, Trust and Reputation Models Simulator for Wireless Sensor Networks".

[8] F. G. Marmol, "Implementing and Integrating a new Trust and/or Reputation Model in TRMSim-WSN".

[9] PM2: https://github.com/Unitech/pm2

[10] Redis: http://redis.io/

[11] MongoDB: https://www.mongodb.org/