



COSMOS

Cultivate resilient smart Objects for Sustainable city applicatiOnS

Grant Agreement N° 609043

D5.2.1 Decentralized and Autonomous Things Management: Software prototype (Initial)

WP5: Decentralized and Autonomous Things Management

Version: 1.0

Due Date: 30 June 2014

Delivery Date: 22 July 2014

Nature: Prototype

Dissemination Level: Public

Lead partner: 4 (ICCS)

Authors: Orfefs Voutyras (ICCS),
Panagiotis Bourellos (ICCS)

Internal reviewers: Adnan Akbar (UniS)



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	11/7/2014	Orfefs Voutyras, Panagiotis Bourelos	ICCS	Sections 5,6,7,8 introduced.
0.2	14/7/2014	Orfefs Voutyras, Panagiotis Bourelos	ICCS	Providing more content for sections 2,3,4.
0.3	18/7/2014	Orfefs Voutyras, Panagiotis Bourelos	ICCS	Figures added.
0.4	21/7/2014	Orfefs Voutyras, Panagiotis Bourelos	ICCS	First version for internal review.
0.5	22/7/2014	Orfefs Voutyras, Panagiotis Bourelos	ICCS	Final version after UniS review.

Annexes: /

Nº	File Name	Title

Table of Contents

Table of Contents	3
Table of Figures.....	4
1. Introduction	7
2. Planner: Case Based Reasoning	8
2.1. Description.....	8
2.2. Functionalities.....	8
2.2.1. Case Base.....	8
2.2.2. Reasoner	9
3. Decentralized Discovery Mechanisms	10
3.1. Description.....	10
3.2. Functionalities.....	10
3.2.1. Cases discovery	10
3.2.2. IoT-services discovery.....	11
3.2.3. Friends discovery.....	11
3.2.4. Calculation of max ttl	11
4. Social Analysis.....	12
4.1. Description.....	12
4.2. Functionalities.....	12
4.2.1. Friends list and friend recommendation service.....	12
4.2.2. Dependability Index calculation.....	13
5. DEMO: Efficient Heating Service	14
5.1. London Scenario:.....	14
5.2. Madrid Scenario.....	14
6. Used ontology files.....	15
7. Delivery and usage	16
7.1. Package information	16
7.2. Download and Installation instructions	16
7.3. User Manual.....	17
7.4. Licensing information.....	20
8. Architecture.....	21

Table of Figures

Figure 1: Main interface of the GUI.	17
Figure 2: Message returned after a successful case retrieval from the local case base.	18
Figure 3: Case returned from Flat2.	18
Figure 4: Case returned from Flat3 with Flat2 acting as a broker.	18
Figure 5: The result of a friend recommendation service provided by a friend of the initial VE.	19
Figure 6: Sequence diagram of the CBR and cases discovery mechanism.	21
Figure 7: Sequence diagram of the IoT-service discovery mechanism.	22
Figure 8: Sequence diagram of the Friend List Renewal mechanism.	22
Figure 9: The components used from the CBR and cases discovery mechanisms.	23
Figure 10: The components used for the social links establishment.	23

Table of Acronyms

Acronym	Meaning
API	Application Programming Interface
CB	Case Base
CBR	Case-Based Reasoning
D	Deliverable
FM	Friends Management
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ID	Identifier
IDE	Integrated Development Environment
IoT	Internet of Things
JDK	Java Development Kit
KB	Knowledge-Base
K-NN	K-Nearest Neighbor
OS	Operating System
OWL	Web Ontology Language
PPM	Profiling and Policy Management
RDF	Resource Description Framework
REST	Representational State Transfer
SA	Social Analysis
SM	Social Monitoring
SNA	Social Network Analysis
SPARQL	SPARQL Protocol And RDF Query Language
TTL	Time To Live



URI	Uniform Resource Identifier
VE	Virtual Entity
VM	Virtual Machine
WP	Work-package
XP	Experience

1. Introduction

This document is the complement to the delivered software regarding the month 10 WP5 prototype deliverable. More specifically, the structure of the document is as follows:

- **Sections 2-4:** Discusses the services that have been developed till now under the several components of WP5. The schema of the analysis followed in each one of these sections is:
 - **Description:** Provides the main description of the component, the rationale behind deciding its development and general advantages that it can offer to end users.
 - **Functionalities:** Describes briefly the main services that are currently provided by the component while using references to the source code.
- **Section 5:** Describes the use case that has been chosen in order to demonstrate the use of the main components and services of WP5.
- **Section 6:** Contains a brief description of the ontology files that have been produced in order to provide input to the several services described in Sections 2-4.
- **Section 7:** It provides delivery and usage information regarding the prototype source code and is organized as follows:
 - **Package information:**
 - **Download and installation instructions:**
 - **User manual:** The most important part of this document. It describes how the results of the several services implemented by the source code can be showcased even in a small degree.
 - **Licensing information:**
- **Section 8:** Summarizes the various relations between the different components and services of WP5 that have been presented on this document.

2. Planner: Case Based Reasoning

2.1. Description

The main functionality of the Planner is providing the VEs the ability to run applications by using a reasoning technique for “solving” problems and facing various situations. This is a step towards the autonomy of the VEs as depicted by the goals of Task 5.2 (Autonomous and predictive reasoning of things).

The reasoning technique that was chosen for the development of this component is the Case-based reasoning (CBR). CBR is the process of solving problems based on past experience. In more detail, it tries to solve a case (a formatted instance of a problem) by looking for similar cases from the past and reusing the solutions of these cases to solve the current one. Cases encompass knowledge accumulated from specific (specialized) situations. The advantages and disadvantages of case-based reasoning and the reasons we chose this reasoning technique instead of others were analysed in COSMOS WP5 D5.1.1 (pp.17-19).

The CBR reflects human reasoning and is the easiest approach for the developers and the one that can be used in a huge variety of domains. Application developers will have the opportunity to create new applications based on CBR, by defining how new cases should be created (what is going to be the pattern of the problems and the solutions). COSMOS from its side will provide the VEs, which will download these applications, with the necessary reasoner and all the mechanisms needed for the CBR cycle (Section 4.1.5. in COSMOS WP5 D5.1.1).

2.2. Functionalities

2.2.1. Case Base

The application developers will be able to define how new cases should be created, by describing them through an ontology. Generally, a case will consist of:

- i. **a problem** which is going to be a series of events that have to be identified to trigger the solution. This description of events has to be linked with the corresponding topics on the COSMOS message bus. The problem can be simple (event) or complex (series of events)
- ii. **a solution** which (in its simplest form) can be the URI of an IoT-service. A solution can be primitive (1 task- IoT-service) or complex (series of IoT-services).

One of the main disadvantages of CBR (store/compute trade-offs because of large case bases) is tackled, as most of the VEs are going to have their own light-weight case base stored locally. As it mentioned in COSMOS WP5 D5.1.1 (Section 4.1.4.), we are going to develop an ontology-based CBR planner and adopt, initially, the Flat Memory model.

In the use case that we are going to present in Section 5, the way the Cases are created is by using the method **thermocase** in the Monitor class of the DemoProject. This method generates random numbers at fixed intervals for the values of temperature, time etc.

2.2.2. Reasoner

CBR is based on finding solutions to new problems by reusing older cases. As a result, the only component needed besides the CB in order to achieve CBR at its simplest form is a reasoner (the Planner) that can identify similarities between cases. The Planner will become part of the VEs during their registration time and will run locally.

In order to measure similarity between cases, we need to follow two steps. First of all, the similarity of the attributes of the new case with the various cases in the CB has to be measured. This is done in the source code by using the method **compareProbParams** in the **Planner** class. This method takes as input the names of the parameters of a new incomplete case (a case that consists of a problem and no solution) and the percentage of acceptable similarity. The similarity is calculated by using the Jaccard coefficient.

If the **compareProbParams** returns as an answer “true” (meaning that there are cases in the CB that have sufficiently similar attributes to those of the new case), the **searchSimilarCase** in the **Planner** class is calculating a new similarity index in order to find the most appropriate case. This method takes as input a list of the names of the attributes describing the new case, a list of the values of the attributes, a list of the weights of these attributes (if any) and a new similarity threshold regarding the values of the attributes and returns the most appropriate solution (values of the attributes of the solution, a URI or a message). In this case, the similarity is calculated by using the Bray-Curtis distance.

It should be noted at this point that the Planner has two modes. Whenever a VE creates a new case on its own, the **searchSimilarCase** method is used. However, if a VE accepts a request for the search of a new case from another VE, the **searchSimilarCaseRequest** method is used. In the last case, the reasoner searches only for shareable cases, defined by flags. These flags can be set either by the application developer or the owner of the VE.

To conclude, in our case, the simple flat memory k-Nearest Neighbor retrieval (K-NN) is used. In this approach, the assessment of similarity is based on a weighted sum of features.

3. Decentralized Discovery Mechanisms

3.1. Description

The concept of adapting past solutions is one of the main requirements of CBR. For this reason, we introduce the idea of allowing VEs to share their cases, thus producing an environment where the knowledge is distributed and knowledge flow is supported. In that sense, the cases produced via CBR become one form of Experience of the VEs, which can not only aid in the planning of future solutions, but can also be shared between different VEs.

Because of the great distribution of the knowledge among the VEs, the development of decentralized discovery mechanisms becomes essential. For this reason, we use mechanisms such as ttl (time to live) in order to support a controlled flow of requests among the VEs. As a result, the first steps towards decentralized discovery of Cases, IoT-services and Friends are made.

3.2. Functionalities

3.2.1. Cases discovery

The method **experienceSharePOST** is the starting point of the Case discovery mechanism. It uses the Http-POST method to pass data into a remote VE targeting the URI of the Experience Sharing mechanism. The method structures the request into a POST body by creating a String variable containing the required variables and attributes.

Upon receiving the request, the remote VE extracts the data and checks the value of the ttl number. If by reducing it the ttl number is above zero then it calls on the suitable Planner Component methods, in order to check if a similar problem structure exists inside the local CB (compareProbParams) and after a positive reply, initiates the searchSimilarCase method of the Planner.

If a similar case is not retrieved then the remote VE becomes a “broker” by initiating recursively a new call of the Experience Share mechanism, using the new ttl calculated by the method described in 3.2.4 of this document. Thus, we ensure that the entire process will operate until either a suitable solution has been discovered or until it reaches a dead end (ttl time out, no case present in our friend VE cluster).

It is worth noting that in the case of outward generated case retrieval, the VE will only retrieve cases marked as shareable in the local CB by the “isShareable” data-type property. Below is a brief outline of the input and output of the Experience Sharing mechanism:

- Input: Problem parameter names, Problem parameter values, Solution parameter names, Address (IP) and port, ttl.
- Output: Solution parameter values, Similarity percentage, URI, Message, Who gave the solution.

3.2.2. IoT-services discovery

The DiscoverIoTService mechanism is initiated by the **discoveryMechanism** method that contacts a remote VE through the Http-POST method in order to extract information about the semantic description of services pertaining to a specific domain. When the remote VE is contacted, it uses the Planner's **searchIoTService** method as described below to provide the answer. No recursiveness has been added into the mechanism yet. Below is a brief outline of the input-output parameters of the mechanism.

- Input: domainName, IP address & port
- Output: Service name & URI, Names of input parameters of the service, Names of output parameters of the service

The **searchIoTService**, implemented in the Planner component, uses the input provided by the discoveryMechanism method above (more specifically, the domain name) and retrieves from the service local store the semantic description of any relevant IoT-services. The answer is a structured ArrayList variable, containing all relevant data as stated above.

3.2.3. Friends discovery

The **requestMoreFriends** method initiates the Friend Discovery mechanism in a VE with the input being only the address of a remote VE, which will provide us with a suitable friend candidate. The philosophy of the previous discovery mechanisms is used, meaning RESTful communication through Http-POST. When the remote VE receives the request and begins to process it in its doPost method, (part of the same FriendRecommendation class) it calculates its most subjectively dependable friend, based on its own subjective Dependability Index and returns the information necessary as a recommendation to the original VE. Further code development will include the ability of the original VE to ask the Social Analysis Component of the COSMOS platform for relevant data of the recommended new friends, so that it can establish new social links.

3.2.4. Calculation of max ttl

For each one of the discovery mechanisms described previously, one of the main goals is refining of their recursive "abilities". A big step in this direction is the use of the ttl variable, not only as a static although customizable input, but as an actual dynamic representation of the in-between stages of "information brokering", based on the theory of the six degrees of separation and the actual social capabilities of affected VEs. This means that if a ttl begins high enough and the recursive discovery uses nodes (broker VEs) with many outward connections, this can have an adverse effect on network overhead and discovery overlap. Therefore, by using the **calculateMaxTTL** method of the Social Monitoring component we can take into account not only the initial recursiveness targeted but other criteria too (like the number of friends). Thus, we can actually adjust this number if we deem it too high. By initially using a simple mathematical function of a logarithmic nature, we can adjust the ttl by using the friend count of the current VE as well as a "target audience" to be reached (maxhits variable). This variable will be extracted per domain, and further code development and further experimentation with real data will refine the function used. Finally, if the experimental data points to such a possibility, we will use an absolute upper limit for ttl in the sense of the six degrees of separation theory. Following the calculation of the ttl from the previous method, the **getMaxTTL** method will check whether the inputted ttl is above bounds and if so then adjust it accordingly.

4. Social Analysis

4.1. Description

In order to support effective experience sharing and discovery, social links establishment becomes necessary. Sending a request only to VEs that are directly involved to its content is the most efficient way of communication, as only VEs that can directly help or can offer the new required knowledge are informed. This way, the communication overhead is kept to a minimum, which is important for keeping the communication channel alive. Thus, the VEs need Friends and COSMOS has to develop a social environment that can support their discovery. For this reason, services like Friend Recommendation are developed.

Moreover, the concepts of reliability, trust and reputation are of outmost importance whenever a VE has to choose among cases offered as an answer to its request by different VEs. For this reason, services like Dependability calculation are provided too.

4.2. Functionalities

4.2.1. Friends list and friend recommendation service

During the registration process of a VE in the COSMOS platform, it will be possible for the user to define pre-existing friends inside the friend-list of the localstore. This is the most basic way a VE forms social bonds with others and such friends will have a number of benefits during the social monitoring or discovery mechanism phases (e.g. greater priority). Another way of acquiring friends will be through the friend discovery mechanism as explained in 3.2.3.

However at any given point during the VE's lifecycle, it can also petition the COSMOS Social Analysis component for a renewal of its friend list based on the new social metrics stored in the COSMOS Social Ontology for each VE. Such a process is initiated by the VE acquiring a list of its friends through the **retrieveFriends** method.

After the list of friends is created, the VE will also instantiate a List of weights to be used in the calculation of the new Dependability Indexes by the Social Analysis component. Once the component receives the input from the **initiateFriendListRenewal** method, it will use the RESTful input to calculate Dependability Indexes for existing friends through the method described below. After this step, the Social Analysis component will make use of a "threshold" of Dependability (currently hard-coded), in order to purge the list of no longer reliable friends.

If indeed VEs have been purged, then the Social Analysis will proceed to calculate new Dependability Indexes for other, similar VEs inside the Social Ontology. After sorting the returned VEs by their Indexes, it will append the data of the most dependable one, until the original friend-list size is reached. Finally the data returned to the original VE will then be used by the **buildFriendFile** method in order to recreate the friend list with the new friends and their new Dependability Indexes.



4.2.2. Dependability Index calculation

Having described the process through which a VE can contact the COSMOS Social Analysis component and have its friend list renewed, it is at this point wise to actually describe the way new Dependability Indexes are calculated. At every VE, the stored Dependability of its friends is subjective, meaning that although it is derived by the same universally calculated and stored data-type properties “hasReliabilityIndex”, “hasTrustIndex” and “hasReputationIndex”, the weights used in its calculation are most probably different for each VE.

The Social Analysis component retrieves the three mentioned Indexes and firstly calculates a normalized number by dividing trust and reputation. This happens because by definition reputation is greater than trust. Afterwards, by using the weights provided by the VE, it calculates the weighted sum of the normalized number and the reliability index in order to produce the new Dependability Index as a normalised number.

5. DEMO: Efficient Heating Service

The purpose of this section is to describe the use case that has been chosen in order to demonstrate the use of the main components and services of WP5. The different steps of the scenario are given below:

5.1. London Scenario:

In smart home environment, total energy consumption is measured in real time with the help of smart meters.

1. Every flat is modelled as a VE. Every flat contains a thermometer (sensor) and a boiler whose temperature can be measured (sensor), as well as set (actuator).
2. A VE registers to COSMOS and its IoT-services are defined and exposed through RESTful services. Moreover, COSMOS provides the VE with an initial set of friends. The user should be able to add friends to his/her VE manually too.
3. During the creation of an application, COSMOS offers the developer the opportunity to define how cases relevant to his application are going to be stored, created and maintained.
4. A user downloads an application for a VE. This application defines how the several cases (both complete and incomplete) are going to be created. For example, the flat VE continuously records the actions of a human user regarding the heating of the flat. In our case, it may record that at a room with a temperature of 6 °C, the user turned the heating on for a total of 10 minutes (600 seconds) and stopped at 20 °C, using hot water of 70 °C.
5. That way, the VE builds a case of {6, 600, 20, 70} and, in a similar, fashion its case base.
6. COSMOS manages the CBR cycle.
7. Each time a new incomplete case is created, the VE searches its Case Base for a solution. For example, the user may inform the VE that he/she will return after a specific time interval and request a certain target temperature in that certain time limit.
8. If no solution is detected, the VE initiates the Experience Sharing service and asks its friends for help.
9. A number of cases is returned by the friends.
10. Based on the dependability index of the friends and the similarity of the cases (the weights of these criteria can be defined), the cases are sorted and the best case is chosen.
11. The case is evaluated based on its results and the Trust of the friend that shared its case is recalculated.
12. At some point the VE may want to refresh the list of its followees (friends).
13. The VE sends a friend-recommendation request to the Social Analysis component.
14. The parameters passed to the request include weights for calculating the dependability (trust and reliability), a minimum acceptable limit of it and the list of followees the VE already has.
15. A new friend list is returned that includes the dependability indexes. Friends that have been set by users should not be thrown away, but isolated.

5.2. Madrid Scenario

In this use case, all buses are exactly the same, have the same owner and send all of their data centrally to EMT. As a result, the development of a nice scenario for showcasing important mechanisms of WP5 (like the flow and management of the VEs' knowledge) was proved to be really difficult and for this reason we focused during year 1 on the London scenario.

6. Used ontology files

For the prototype code's needs, we make use of the ontology files contained inside the `localstore` folder that accompanies the project files. Because of the fact that we do not yet have access on different VMs to test the code that has been developed until this point, we use the same "file storage" structure for each simulated VE but with different file names. All files used for storage and retrieval of data are `.owl` and their general structure and purpose is as follows:

- **List of Friends:** The list of friends is a collection of individuals along with the local VE and their data-type and object-type properties (i.e. "hasFriend" or "hasDependabilityIndex"). The files are named `friendlist.owl`, `friendlist1.owl` and `friendlist2.owl`, with the first belonging to Flat1, the second to Flat3 and the user VE (shared) and the third one to Flat2.
- **List of Cases (Case Base/CB):** These owl files are used for the storage and retrieval of data pertaining to the information that is described by Cases. Cases are represented as two separate individuals (problem and solution) connected through the use of the "solves" and "isSolvedBy" inverse functions. They also possess data-type properties containing their relevant data. The files are named `casebase.owl`, `casebase1.owl` and `casebase2.owl` with Flat1 owning the first, Flat3 the second and Flat2 the third.
- **Service stores:** These files, that may or may not be merged with the rest, dependent on the course of the code development, contain the semantic description of IoT Services and are used for the **DiscoverIoTService** mechanism of our prototype. They are `VE.owl`, `VE1.owl` and `VE2.owl` following the same pattern of ownership.
- **Cosmos Social Ontology:** This owl file contains all relevant data for VEs that can be used to infer social connectivity between them and evaluate their performance upon request from any VE through the **RebuildFriendList** mechanism as is described in detail bellow. The importance of maintaining such a file is mainly due to the fact that, in principle, socially dynamic behavior cannot be achieved without the platform being informed of changes in the social characteristics of VEs through the use of feedback mechanisms that will be developed in future prototypes.

At the moment all updates in the triple stores (not counting changes for testing purposes) are made through the use of the Jena libraries, provided by the `OntModel` class and all queries/retrievals are done using SPARQL. Test changes are done directly to the owl files through Protégé.

7. Delivery and usage

7.1. Package information

The deliverable contains the following projects and folders:

- Folders of Projects Developed
 - DemoProjectMaven
 - ExperienceGiverVEMaven
 - JennaPelletQueryMaven
 - SocialAnalysisComponent
 - FriendOfFriend
- Maven Configuration Files
 - DemoProjectMaven/pom.xml
 - ExperienceGiverVEMaven/pom.xml
 - JennaPelletQueryMaven/pom.xml
 - SocialAnalysisComponent/pom.xml
 - FriendOfFriend/pom.xml
- Package Source Code location
 - DemoProjectMaven/src
 - ExperienceGiverVEMaven/src
 - JennaPelletQueryMaven/src
 - SocialAnalysisComponent/src
 - FriendOfFriend/src
- Executables
 - /target/DemoProjectMaven-1.0-SNAPSHOT.jar
 - /target/ ExperienceGiverVEMaven-1.0-SNAPSHOT.jar
 - /target/JennaPelletQueryMaven-1.0-SNAPSHOT.jar
 - /target/SocialAnalysisComponent-1.0-SNAPSHOT.jar
 - /target/FriendOfFriend-1.0-SNAPSHOT.jar

7.2. Download and Installation instructions

The development took place in an x64 Windows 7 OS. The prerequisite for the execution of the applications is JDK 1.8. This is bundled along with the NetBeans 8.0 IDE that was used for code development. The entire bundle can be accessed at:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

After downloading and installing the previously mentioned components, the next step is to transfer all Project Folders to the newly created NetBeansProjects Folder inside the Documents section. After starting NetBeans and opening the Projects that use Maven for greater ease in transportability between workstations, the user may be asked to execute resolve problems, so that all dependencies are imported. By right clicking on the project names and selecting resolve problems, Maven will connect to the central repository and download to a new local repository all dependencies for current and future use. Also it is important to add that in order for the projects to run correctly, the accompanying localstore folder must be placed as is, in the main hard drive of the workstation.

7.3. User Manual

Though the applications can be run by their accompanying executable jar files inside the respective target directories of the Project files, it is best to run them through NetBeans. This way, the user can observe all underlying messages in the output window of the IDE which would not appear during execution just by jar.

The first component to be demonstrated is the use of the **Planner** responding to a notification pertaining to the scenario of Efficient Heating as described in section 5. The first phase should be the running of the ExperienceGiverVE and JennaPelletQuery Projects as they simulate two auxiliary VEs that participate in the experience sharing process that may be deemed necessary by the values given during the demonstration. Finally, the user must run the DemoProject in order to start a GUI that provides the means to begin the process. The Demoproject simulates the GUI waiting for user input, as well as the primary Flat VE receiving the notification.

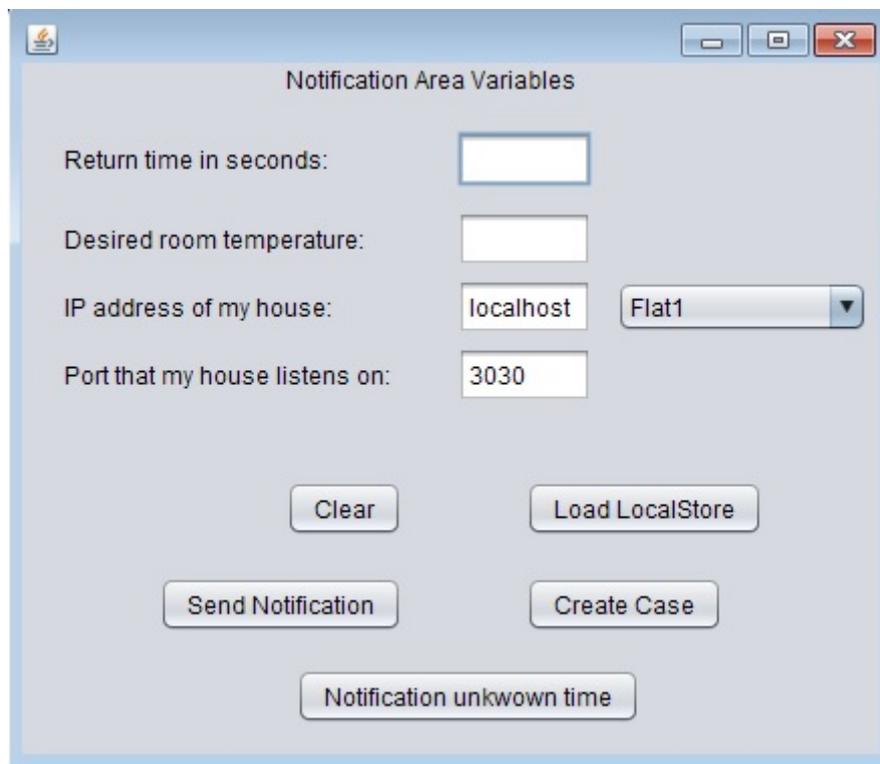


Figure 1: Main interface of the GUI.

By adding the values 3535 in the first box and 26 in the second box, we can begin the notification process as soon as we hit the Send Notification button. At that moment, the application sends an http POST request (through the **sendNotification** method inside the Servlets.java file) to the **recieveNotification** Service (implemented inside the same file). After the **POST** I received, the VE calls an instance of the **Planner** and begins searching for a similar case as the one provided by the user. The case structure is {TempBefore-time-TempAfter} and {boilerTemp-URI of Service-message}. Current temp is always set at 5 degrees. The **searchSimilarCase** method will run without first checking the **compareProbParams** function as it is expected the VE will contain such a structured case by design. With the provided values, the VE searching the local case base will indeed find a similar case and thusly return:

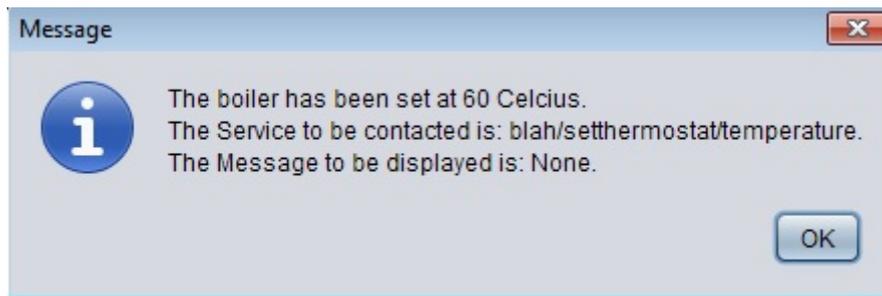


Figure 2: Message returned after a successful case retrieval from the local case base.

Afterwards, we enter the values 1800 and 26. Such a case is not present inside the first VE so after failing to retrieve it, the VE will call the experience share service and contact through http **POST** its friend, "Flat2", implemented by the ExperienceGiverVE project. The second VE, after receiving the request, will also call on its **Planner** in order to check the structure of the received experience request. If the **compareProbParams** method returns true then and only then it will proceed to access its case base, in order to locate a similar problem-solution pair. After locating such a case, the Experience Sharing Service returns to the first VE, which displays the answer as follows:

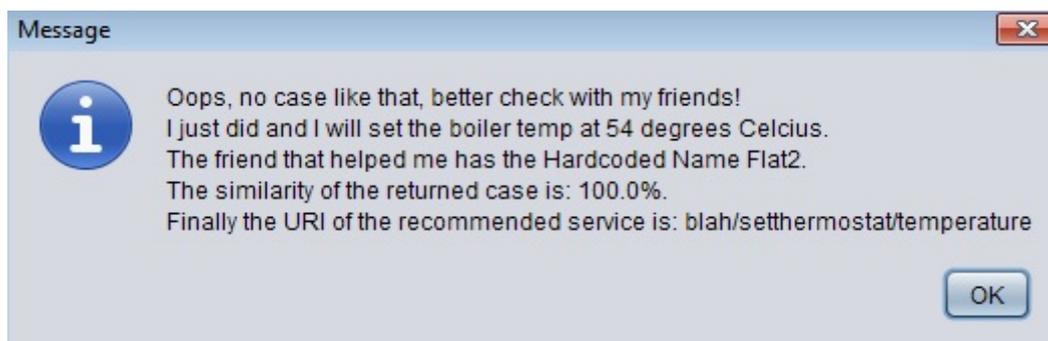


Figure 3: Case returned from Flat2.

Finally, in order to check the recursiveness of the Experience Sharing Service, we will have to enter the values 3535 and 28 in the GUI fields. Such a combination of values is present in neither Flat1 nor Flat2, so after failing to find a similar case, Flat2 will call Experience Share recursively on Flat3, implemented by the JennaPelletQuery project. While searching the local CB, Flat3 will in turn locate a similar case and return it to Flat1 VE, with Flat2 acting as a **broker** between the two of them:



Figure 4: Case returned from Flat3 with Flat2 acting as a broker.

It is worth mentioning that, during this example, the output tabs inside NetBeans will at some point display the output of the **DiscoverIoTService** mechanism that Flat1 implements in a basic

fashion. The service contacts a VE (itself at this example) and requests the values of the semantic description of any service associated with the domain provided as input.

Until this point, the **Social Monitoring's** function was to streamline the communication between VEs by choosing the appropriate friends from the local friend list. As described, another main target for the component is to provide a mechanism to dynamically change the depth of the recursiveness in Discovery Services, based on the number of friends (popularity) of intermediary broker VEs. Such a mechanism will call on the component's **getMaxTTL()** function that retrieves the previously calculated value and during the reception of incoming REST communications, will modify the incoming **ttl** if it is deemed necessary. Also, the calculation of the max value will take place ideally after each friend list renewal. In our example, such checks for the **maxttl** variable are made during the Experience Sharing mechanism (**Case Base Discovery**).

Another functionality that is implemented is a first draft of the **Social Analysis (SA)** component (expected to run on COSMOS) and more specifically, its function as a means to provide a renewal mechanism for the friend list of individual VEs, based on the information of the **COSMOS Social Ontology**. This code is implemented in the **SocialAnalysisComponent** Project which simulates a VE sending the names of its friends to the Platform in order to recalculate their Dependability Indexes. The SA uses the VE provided weights to calculate the new Index based on the reputation, trust and reliability that each VE has inside the Social Ontology. After checking the new Dependability Indexes with the threshold set by the requesting VE and if this has led to the purge of friends from the original list, it gives an analogous number of new friends that are above the threshold probabilistically, which means not ordered by the new index. (**RebuildFriendList** mechanism) After the entire process has been completed it returns the new friends to the VE along with relevant data, so that the VE can rebuild its friend list ontology. (**buildFriendFile** method) Any changes in the friend list can be observed by using Protégé to view the friend list owl file and can be triggered by modifying values in the Social Ontology owl file that the component accesses to recalculate the Dependability Index. Namely the values that are used to calculate the Dependability Index are stored in the datatype properties of "hasReputationIndex", "hasReliabilityIndex" and "hasTrustIndex" of each VE. The only pre-requisite is that by definition the reputation index must be equal or less than the trust index and the reliability index is a float number between [0, 1]. **Note that all data in the ontology is stored as strings.** The file names used are friendlist.owl and CosmosSocial.owl inside the localstore folder.

Finally, the Project FriendOfFriend implements the **Friend Discovery** mechanism described previously, in a basic way (at the moment it is self-called and executed), so that a VE can request other friend VEs for friend recommendations based on their own locally stored Dependability Indexes. In this case, the target VE returns a name (or possibly other identifier), of its best friend so that the original VE will be able to receive its data and add it to the friend list after communicating with the SA component. The code implements the return of the answer from the target VE, with further functionality to be added:

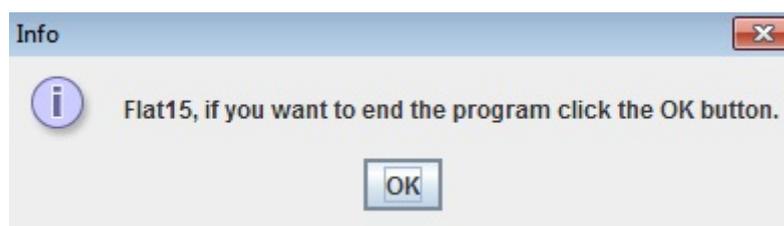


Figure 5: The result of a friend recommendation service provided by a friend of the initial VE.

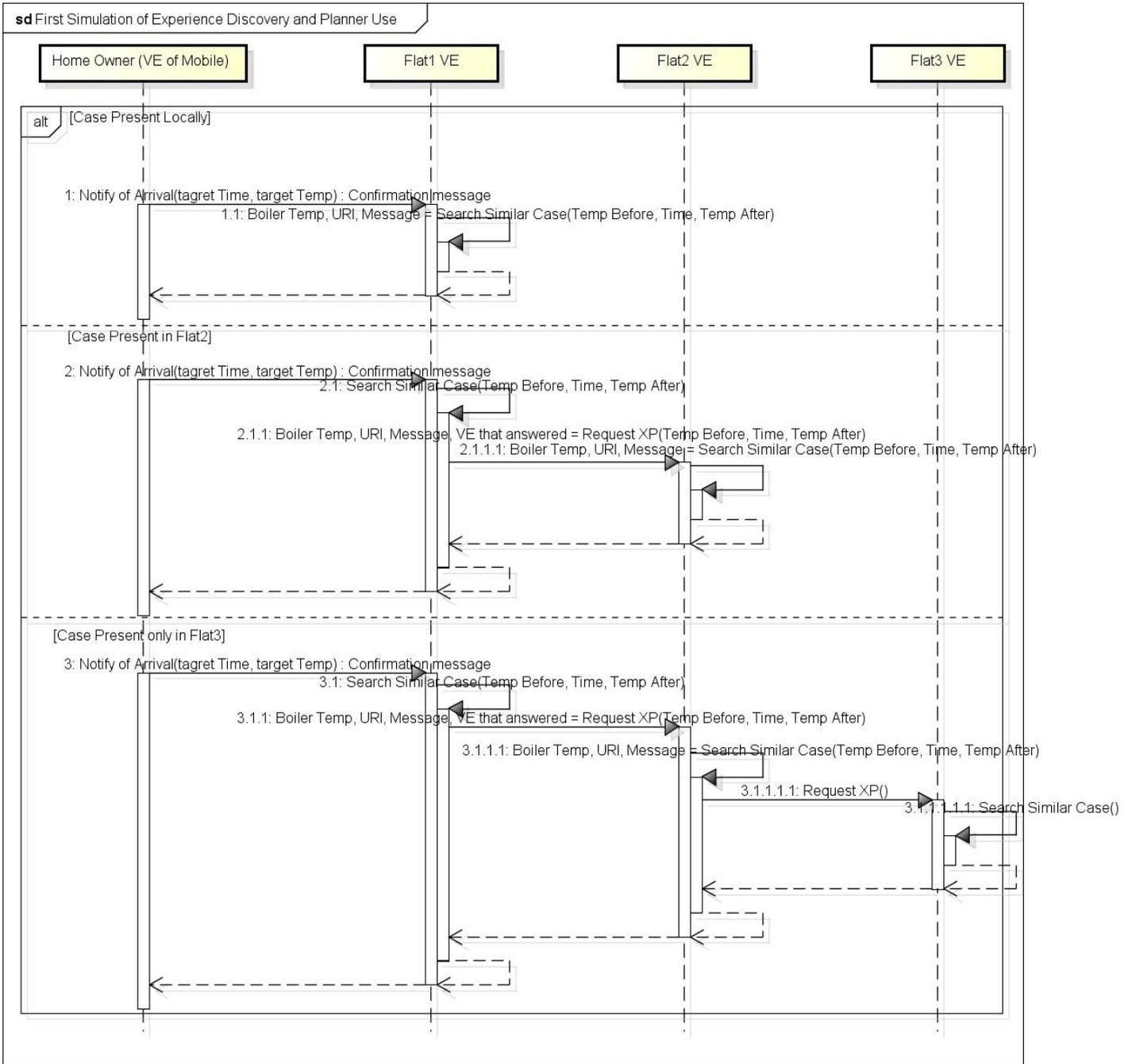
7.4. Licensing information

The libraries used in the development of these prototypes are listed along with their short descriptions and licences as follows:

- **APACHE-JENA-2.10.0:** Apache Jena is a API and toolkit for working with semantic web technologies such as RDF and SPARQL using Java. This artefact represents the source and binary distribution packages generated for releases. **Licenses:** Apache 2.0 License, <http://www.apache.org/licenses/LICENSE-2.0>
- **Jetty-Maven-Plugin-9.1.5-v20140505:** Administrative parent pom for Jetty modules. **Licenses:** Apache 2.0 License, <http://www.apache.org/licenses/LICENSE-2.0>
- **Pellet-Jena-2.3.2:** The Clark and Parsia Pellet OWL Reasoner. **Licenses:** GNU Affero General Public License 3.0, <http://www.gnu.org/licenses/agpl-3.0.html>
- **Maven Dependency Plugin 2.8:** Provides utility goals to work with dependencies like copying, unpacking, analyzing, resolving and many more. **Licenses:** The Apache Software License, Version 2.0, <http://www.apache.org/licenses/LICENSE-2.0.txt>
- **Maven JAR Plugin 2.4:** Builds a Java Archive (JAR) file from the compiled project classes and resources. **Licenses:** The Apache Software License, Version 2.0, <http://www.apache.org/licenses/LICENSE-2.0.txt>

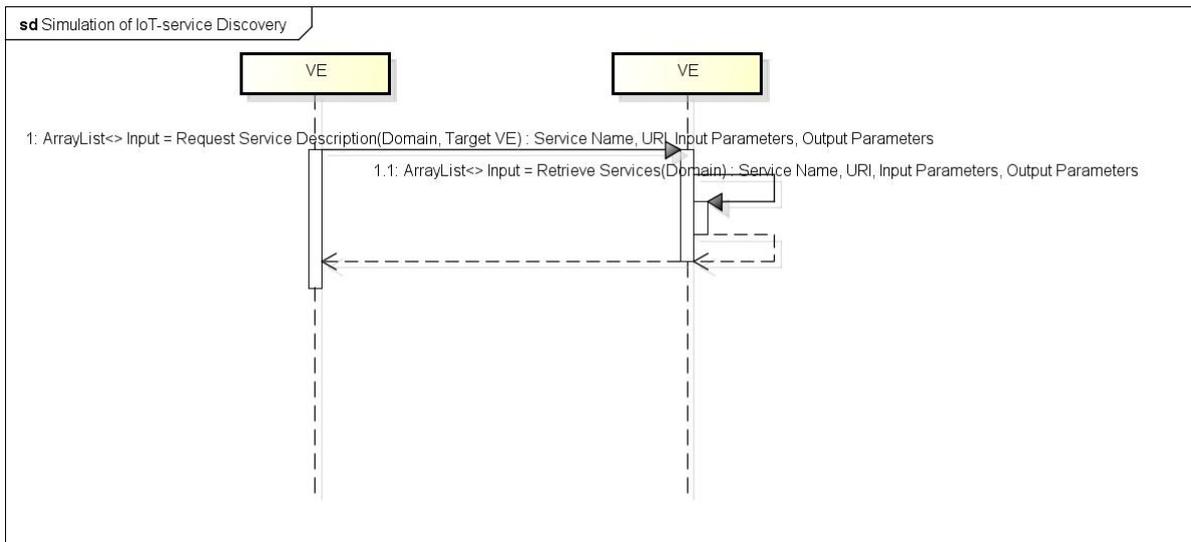
8. Architecture

Below there are some sequence diagrams that represent some of the mechanisms described in the previous sections:



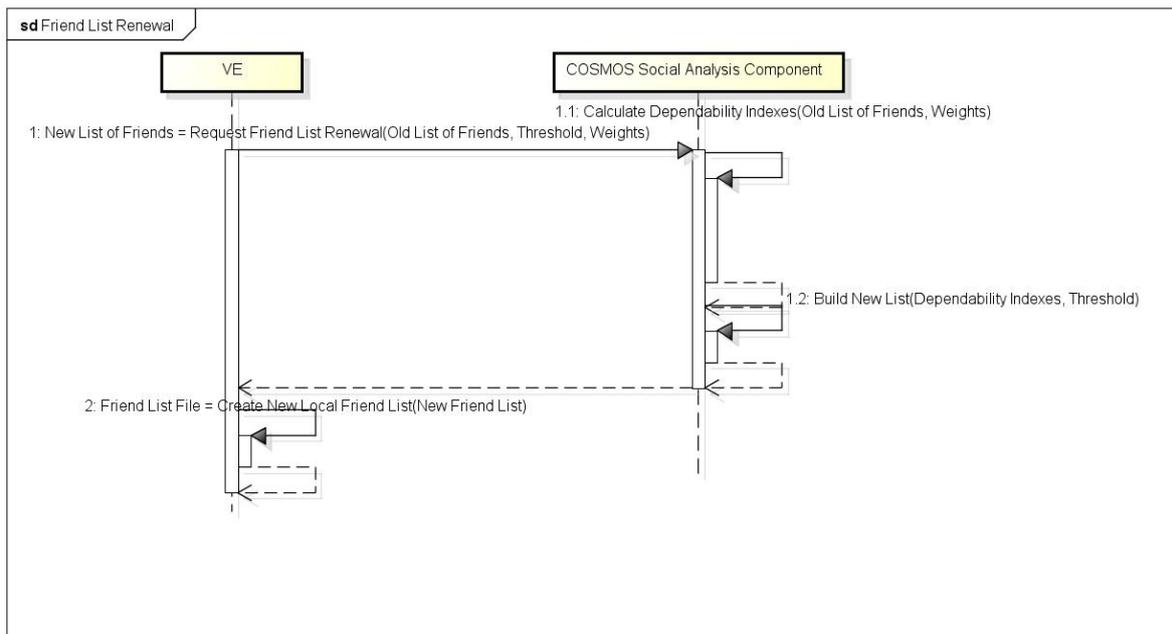
powered by Astah

Figure 6: Sequence diagram of the CBR and cases discovery mechanism.



powered by Astah

Figure 7: Sequence diagram of the IoT-service discovery mechanism.



powered by Astah

Figure 8: Sequence diagram of the Friend List Renewal mechanism.

Finally, the conceptual view of the main cycles described above (CBR, discovery mechanisms and social links establishment) are depicted in the following figure:

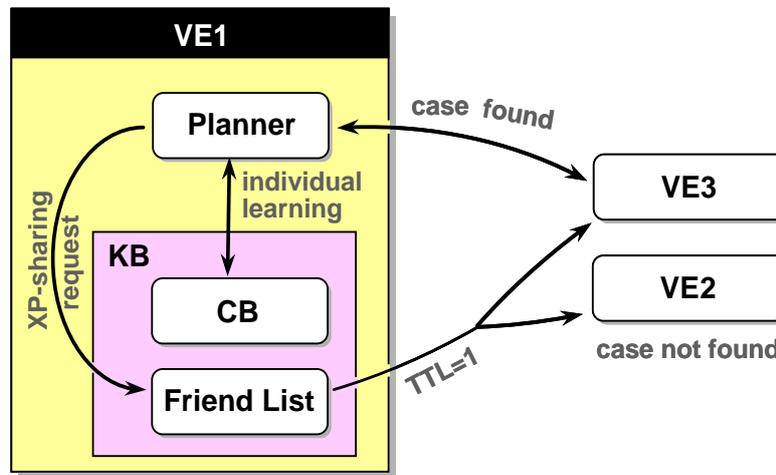


Figure 9: The components used from the CBR and cases discovery mechanisms.

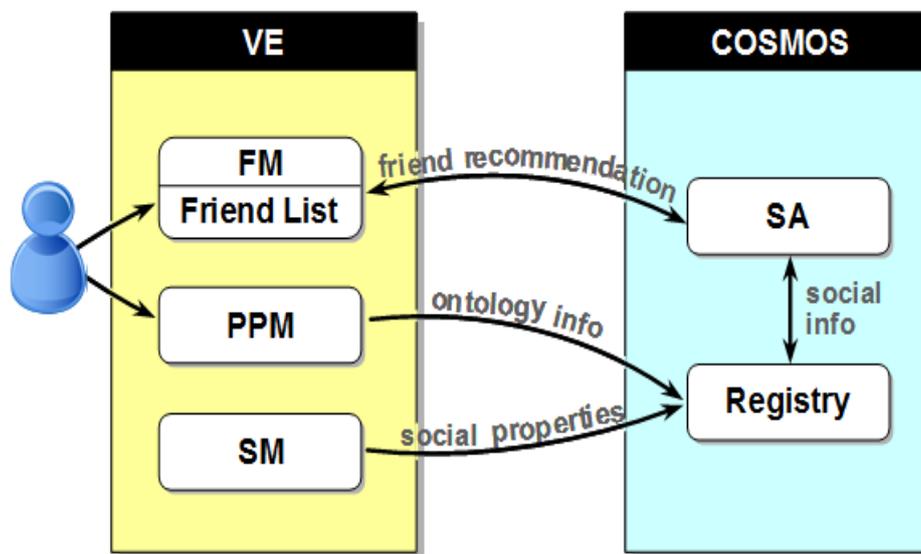


Figure 10: The components used for the social links establishment.

***Profiling and Policy Management (PPM)** component: It assigns a unique ID to the VE and enables the entry of all the information needed for the description of the physical entity through the domain ontology of the corresponding VE. Moreover, it enables the owner to determine the social “openness” of the VE: the IoT-services that can be used by other VEs, the kind of experience that can be shared, the sets of VEs which can access such information etc.

***Friends Management (FM)** component: It is responsible for creating and maintaining the list of friends that a VE has. In other words, it allows VEs to initiate, update and terminate their friendship with other VEs on the basis of the owner’s control settings. It provides the owner with the option of setting new friends to his/her VEs, offers friend-recommendation request services and monitors the friends list of a VE regularly or on demand in order to find any Friends whose Dependability is no more the desired one and thus should be removed. For this purpose, it communicates with the SA component.



***Social Monitoring (SM)** component: It contains all the main tools and techniques that are used for the monitoring of the social properties of the VEs, like Trust and Reputation. Its main objective is to collect, aggregate and distribute monitoring data (events) across the decision making components of the collaborating groups. The events are generated by interactions in response to - directly or indirectly - user actions (e.g. registering a new VE) or VEs' actions (XP-sharing). Social Monitoring "feeds" the VE Registry.

***Social Analysis (SA)** component: Based on the results of the Social Monitoring component and taking advantage of Social Network Analysis (SNA), the SA component is used for the extraction of complex social characteristics of the VEs (e.g. centrality), as well as models and patterns regarding the behaviour of the VEs and the relations between them.