Towards an Open, Secure, Decentralized and Coordinated Fog-to-Cloud Management Ecosystem

D3.5 mF2C Agent Controller Block integration (IT-1)

Project Number	**730929**
Start Date	**01/01/2017**
Duration	**36 months**
Topic	**ICT-06-2016 - Cloud Computing**

Work Package	**WP3, mF2C Agent Controller block design and implementation**
Due Date:	*M12*
Submission Date:	*22/12/2017*
Version:	*0.9*
Status	*Final*
Author(s):	Gregor Cimerman (XLAB), Breogan Costa (WOS) Jasenka Dizdarevic (TUBS), Alejandro Gómez (UPC) Jens Jensen(STFC), Alexander Leckey (INTEL) Eva Marín (UPC), Anna Queralt (BSC) Zeineb Rejiba (UPC), Antonio Salis (ENG) Roberto Bulla (ENG), Souvik Sengupta (UPC), Roi Sucasas (ATOS)
Reviewer(s)	*Roberto Bulla (ENG)* *Matej Artač (XLAB)*

Keywords
mF2C Agent Controller, Internal components, Implementation

Project co-funded by the European Commission within the H2020 Programme		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 730929. Any dissemination of results here presented reflects only the consortium view. The Research Executive Agency is not responsible for any use that may be made of the information it contains.

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	14/11/2017	Table of contents and assignments	Anna Queralt (BSC)
0.2	22/11/2017	First round of contributions	Jasenka Dizdarevic (TUBS) Alejandro Gómez (UPC) Eva Marín (UPC) Anna Queralt (BSC) Zeineb Rejiba (UPC) Souvik Sengupta (UPC) Roi Sucasas (ATOS) Jens Jensen(STFC)
0.3	29/11/2017	Second round of contributions	Jasenka Dizdarevic (TUBS) Alejandro Gómez (UPC) Jens Jensen(STFC) Alexander Leckey (INTEL) Eva Marín (UPC) Anna Queralt (BSC) Zeineb Rejiba (UPC) Souvik Sengupta (UPC) Roi Sucasas (ATOS)
0.4	10/12/2017	Comments and pending issues solved	Gregor Cimerman (XLAB) Breogan Costa (WOS) Jasenka Dizdarevic (TUBS) Alejandro Gómez (UPC) Jens Jensen(STFC) Alexander Leckey (INTEL) Eva Marín (UPC) Anna Queralt (BSC) Zeineb Rejiba (UPC) Antonio Salis (ENG) Roberto Bulla (ENG) Souvik Sengupta (UPC) Roi Sucasas (ATOS)
0.5	11/12/2017	Final version to internal review	Anna Queralt (BSC)
0.6	13/12/2017	Formatted and homogenized version	Anna Queralt (BSC)
0.7	14/12/2017	Internal review	Matej Artak (XLAB) Roberto Bulla (TISCALI)
0.8	18/12/2017	Incorporated reviews and comments from ENG, XLAB	Jasenka Dizdarevic (TUBS) Eva Marín (UPC) Anna Queralt (BSC) Roi Sucasas (ATOS)
0.9	22/12/2017	Quality check and final version to submit	Lara López (ATOS)

Table of Contents

List of figures

List of tables

Executive Summary

This document developed by the mF2C project describes the implementation of the mF2C Agent Controller block. For each of the Agent Controller functionalities, already designed in D3.3, the document provides a set of UML diagrams that detail the purpose of the functionality and how it is implemented to fulfil its purposes.

The outcome of this document is a detailed description of the implementation of the Agent Controller components and functionalities in IT-1, focusing on the internals of each component. The details of the interactions between components are provided in D4.7.

Together with D4.5, the results of this document (and of D4.7) will be used as inputs in D5.1 mF2C reference architecture, leading into a complete integrated solution for IT-1 ready to be validated.

1. Introduction

1.1 Introduction

According to the architecture defined in D2.6 [1], the main building blocks of an mF2C agent are the Platform Manager (PM) and the Agent Controller (AC). In D3.3 [2] each of the functionalities of the AC were described in detail, paying special attention to the interactions between the different functionalities, either in the same or in different agents.

In this document, we detail how each of the functionalities in the AC is developed in IT-1. To do so, we use a set of diagrams that describe their implementation. Each functionality is extensively described by means of the following artefacts:

- *Requirements*: the description of the requirements that the functionality addresses
- *Use case diagram*: shows the functions that the component offers to external actors, which can be users or other software components.
- *Component diagram*: shows the detailed internal architecture of the component. For complex components, a class diagram containing the classes that implement the component is also provided.
- *Internal sequence diagrams*: a set of sequence diagrams that describe the internal interactions within the component in order to fulfil its functions.
- *Data model*: when applicable, a class diagram containing the mF2C information managed by the functionality.
- *Baseline technologies*: a table containing the already existing technologies used to implement the functionality (programming languages, tools, libraries, …), and how are they used in the implementation.
- *Test cases*: for those components that will be implemented in IT-1, the description of a set of test scenarios to test the correct behaviour of the functionality.

The structure of this document is as follows:

- Section 1 describes the aim and the context of this document.
- Section 2 provides an overview of the AC functionalities.
- Section 3 details the implementation of each functionality within the Service Management component.
- Section 4 details the implementation of each functionality within the Resource Management component.
- Section 5 details the implementation of each functionality within the User Management component.
- Section 6 provides the conclusions of this work.

The implementation of the mF2C Platform Manager block is described analogously in D4.5 [3].

1.2 Purpose

The objective of this deliverable is to describe in detail the internal implementation of each of the functionalities in the Agent Controller of an agent, according to the design provided in D3.3 [2].

1.3 Glossary of Acronyms

Acronym	Definition
AC	mF2C Agent Controller
ACK	Acknowledgement message
API	Application Programming Interface
CPU	Central Processing Unit
IE	Information element
IoT	Internet of Things
IT-1	Iteration 1 of the mF2C project
MQTT	Message Queuing Telemetry Transport
OUI	Organizationally Unique Identifier
PM	mF2C Platform Manager
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
SLA	Service Level Agreement
SM	Service Management
UM	User Management
VSIE	Vendor-Specific Information Element

Table 1. Acronyms

2. Agent Controller Integration

As it was proposed in the mF2C architecture the agent entity deploys all the main control and management functionalities in each one of the devices forming part of the mF2C system. The agent controller (AC) is one of the two main building blocks of this agent entity; the other one is the platform manager (PM) which is described in detail in deliverable D4.5 [4].

In the distributed and coordinated fashion proposed in mF2C and where services are executed in different devices, the PM includes the logic of the system, taking decisions based on a more global view. On the other hand, the AC has a more local scope, focusing on local resources, services and users. From the point of view of the Agent Controller, the local resources mean its own resources and the resources of its children if the device is the leader of a cluster of devices (children). However, if the device is not a leader and then it does not have children, local resources are only its own resources. Regarding the services, the AC only controls and manages the services being executed in the own device. Finally, the AC also manages the preferences, roles, profile, etc. of the user owner of the device.

Figure 1 shows the AC controller and its set of functionalities. This set of functionalities is split into three main blocks, Resources, Services, and Users. In this deliverable, Section 3, Section 4 and Section 5 describe the internals of the Service Management, the Resource Management and the User Management blocks respectively and each one of their functionalities (sub-components). This description includes requirements, detailed architecture description, use case diagrams, internal sequence diagrams, data model and, if it exists, the baseline technology.

On the other hand, the interactions between blocks and between subcomponents are deeply described in deliverable D4.7 [5], where the details of implementation of the interfaces between components are reported.

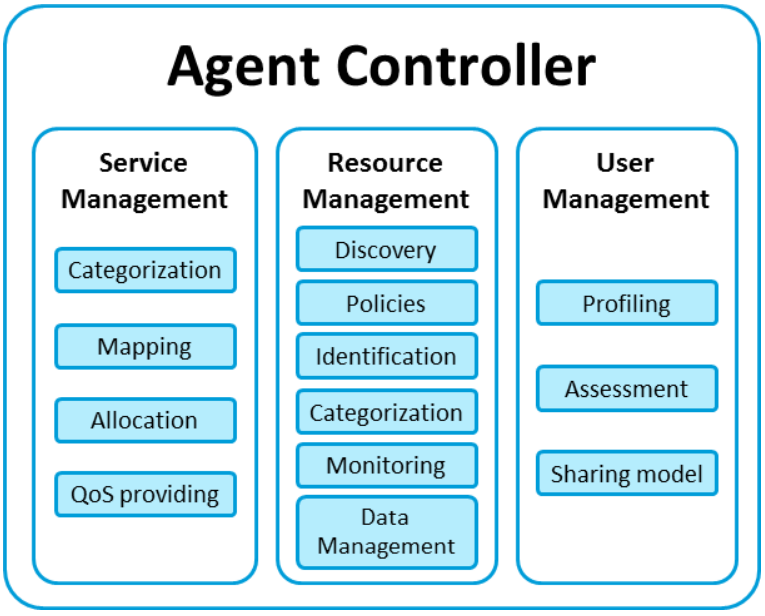

Figure 1 Agent Controller functionalities

2.1 Security provisioning

The AC security component is designed to abstract as much as possible of the required security features from the caller, in order to minimise the security developments in each of the components. The core design is shown in Figure 2.

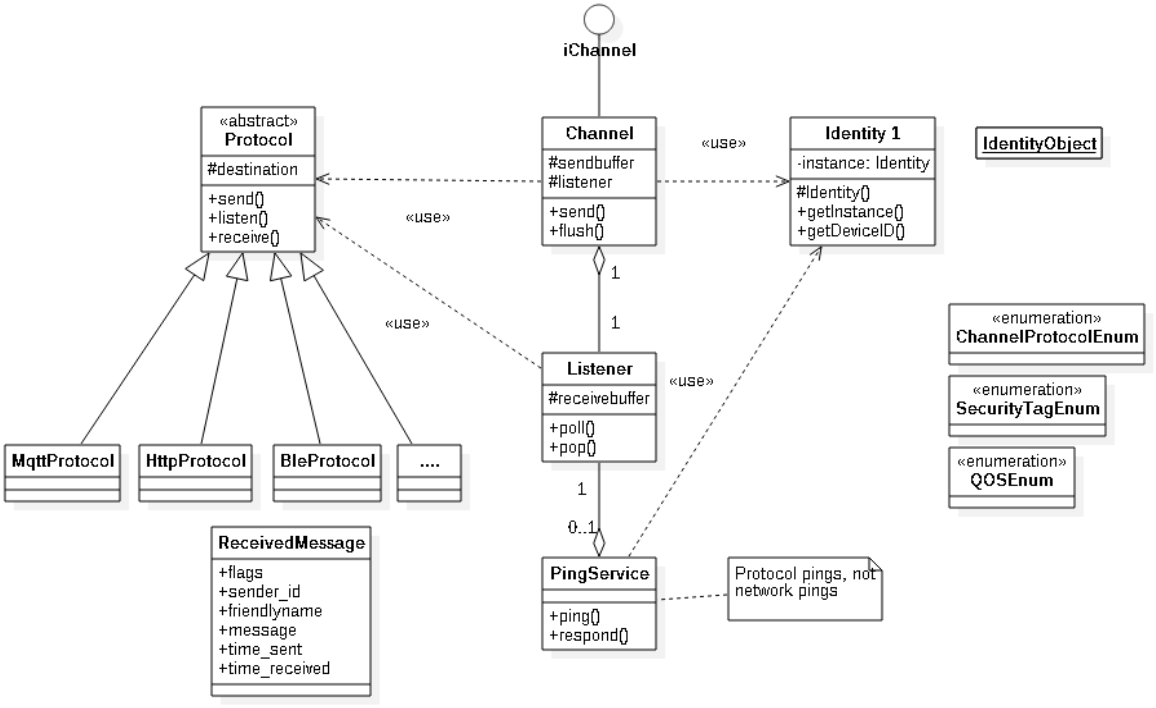

Figure 2 Agent Controller security

The image shows a communications channel (equivalent to a socket in standard network communications, but not necessarily implemented through a socket.) Indeed, channels can be implemented in different protocols, hence the "protocol" abstract class with implementations in HTTP, MQTT, etc. In IT-1 it is proposed that the caller select the required protocol when instantiating the channel, so the caller will need to know whether to ask for an MQTT channel (e.g. a Leader communicating with its Fog) or an HTTP channel (e.g. for accessing a REST web service in a cloud.)

Also notable in Figure 2 is the Identity class: the idea being this is a singleton class, so is instantiated at most (or precisely) once for each instance of the application/agent. The idea is that it would bootstrap the identities if needed - the device id could be calculated locally (in most proposals) but if available (and needed), could also call out and obtain a X.509 certificate. Moreover, the class should also implement a credentials store which securely stores - and activates - any previously obtained credential.

3. Service Management

The Service Management block in the Agent Controller is responsible for the orchestration of local services. This section describes main functionalities of this block, which are represented with the following subcomponents: mapping, categorization, allocation and QoS provisioning. The service requests are decomposed into tasks in the Task Management block of the Platform Manager. After that the Task Scheduler block decides where each individual task will be executed and deploys them to adequate agent controller by using mapping subcomponent of the SM. The categorization subcomponent provides the information about the type of service and its requirements that must be used by the mapping strategy to allocate the optimal resources for a successful service execution. The allocation subcomponent is responsible for the allocation of available resources to the various requests, trying to meet security and privacy rules, cost models, while guaranteeing overall optimal resources usage. The SM block also needs to check and guarantee that the service tasks meet certain constraint requirements by using QoS provisioning subcomponent.

3.1 Categorization

3.1.1 Requirements

The requirements for the categorization subcomponent:

- The categorization component should be able to receive a service task from the mapping subcomponent.
- Prior to that the Lifecycle Manager processes the information from the Recommender and the Landscaper and sends attributes that will be used for categorization in AC's Service Management.
- These attributes are sent to the mapping subcomponent which then sends them to the categorization.
- The attributes defined for the first iteration are CPU, Storage, Network, Memory, Priority, Time limit and Location.

3.1.2 Use case diagram

Next picture depicts the use cases associated to this component:

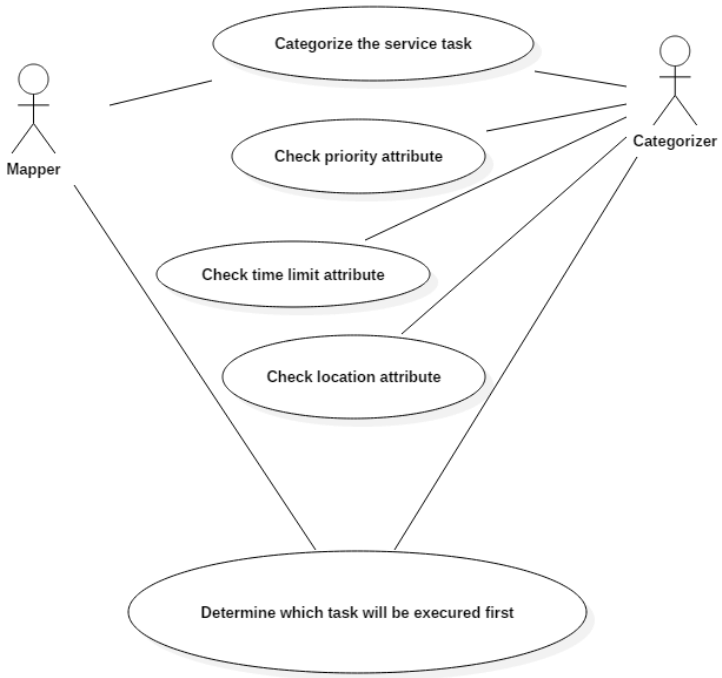

Figure 3 Use case diagram for the Categorization functionality

Scenario:

1. The mapping subcomponent (mapper) asks the categorization subcomponent (categorizer) to classify the received service tasks (for example two tasks) according to its characteristics.
2. Categorizer checks priority and proceeds with the one with the highest priority (and break). If both have the same priority, continue to the next step.
3. Categorizer checks time limit and proceeds with the one with the lower time limit (and break). If both have the same time limit, then continue to the next step.
4. Categorizer checks location (the level the request comes from, is it cloud, fog etc.). If the service task comes from the cloud, it is forwarded to be executed first.

3.1.3 Component detailed architecture

The Categorization module is implemented in a single component, with the class diagram depicted in Figure 4.

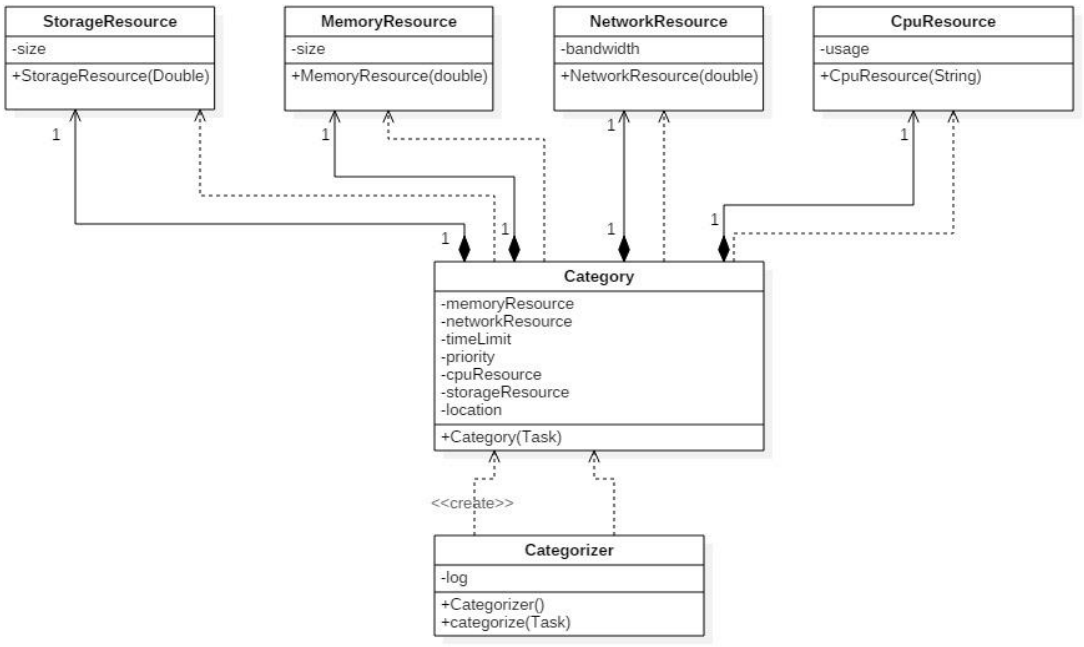

Figure 4 Component diagram for the Categorization functionality

The mapping subcomponent sends a task to the categorization component (class Categorizer in the Figure) in order for it to categorize it based on the attributes from the Lifecycle Manager. Categorizer creates a category for each task with 7 defined attributes: memoryResource, networkResource, time Limit, priority, cpuResource, storageResource and location. Category creates four objects: StorageResource, MemoryResource, NetworkResource and CpuResource. For now, each of these objects define one attribute, StorageResource and MemoryResource define size (in Bytes), NetworkResource defines bandwidth and CpuResource defines usage.

3.1.4 Internal sequence diagrams

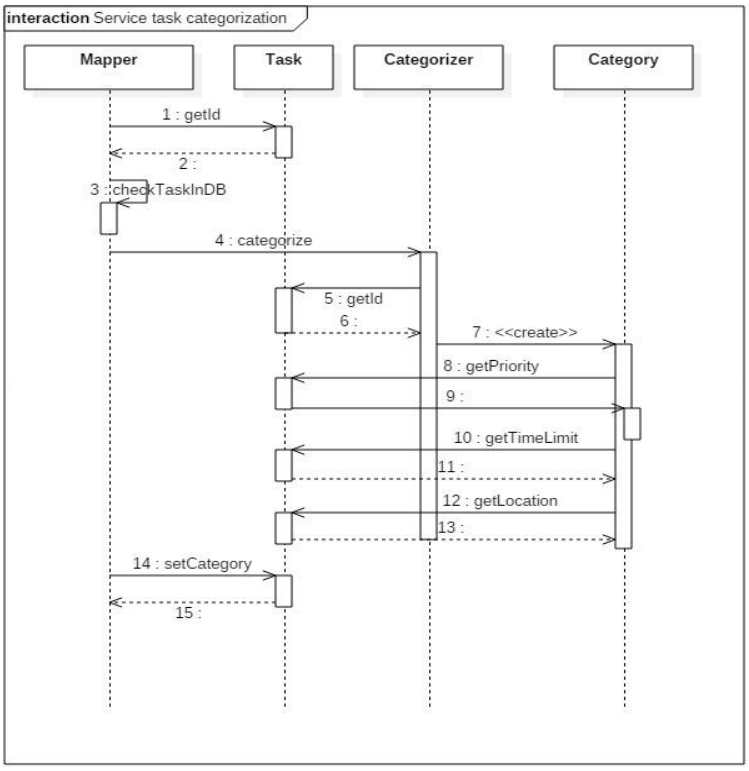

Figure 5 Sequence diagram for Service task categorization

3.1.5 Data model

This component will use the information about service categories that is managed by the Platform Manager. It will later add new attributes on a task level, which will not be implemented in the first iteration.

3.1.6 Baseline technology

The following baseline technologies are used within this component:

Technology	Usage	Reference
Java	Implementation language	http://www.oracle.com/us/technologies/java

Table 2. Service Management baseline technology

3.1.7 Test cases

This component will not be implemented during the first iteration.

3.2 Mapping

3.2.1 Requirements

The requirements for the mapping subcomponent:

- The mapping component should have an interface with the PM from where it receives the service task
- The mapping component should be able to acquire information from the local database (that each device will have a local database, managed by dataClay) if the task already exists

- The mapping component should be able to forward the task to the categorization component in the Service Management for it to be classified according to its characteristics.
- The mapping component should be able to contact the block Policies in the Resource Management to see which rules should be applied to match the task requirements and determine if the execution of the task can be continued
- The mapping component should be able contact the Profiling block in the User Management in order to find out if there are some user constraints and determine if the execution of the task can be continued
- The mapping component should be able to contact the allocation component in the Service Management that will reserve the resources for the task (when there are distinct resources in the same device)
- The mapping component should be able to contact the QoS provisioning component in the Service Management to check for the QoS constraints
- The mapping component should be able to update the local database
- The mapping component should be able to receive the request for the reservation of previously selected resources from Lifecycle model in the PM. After receiving this request it will call the allocation subcomponent that will do the actual allocation of the resources.

3.2.2 Use case diagram

Next figure shows the mapping use case-scenario 1:

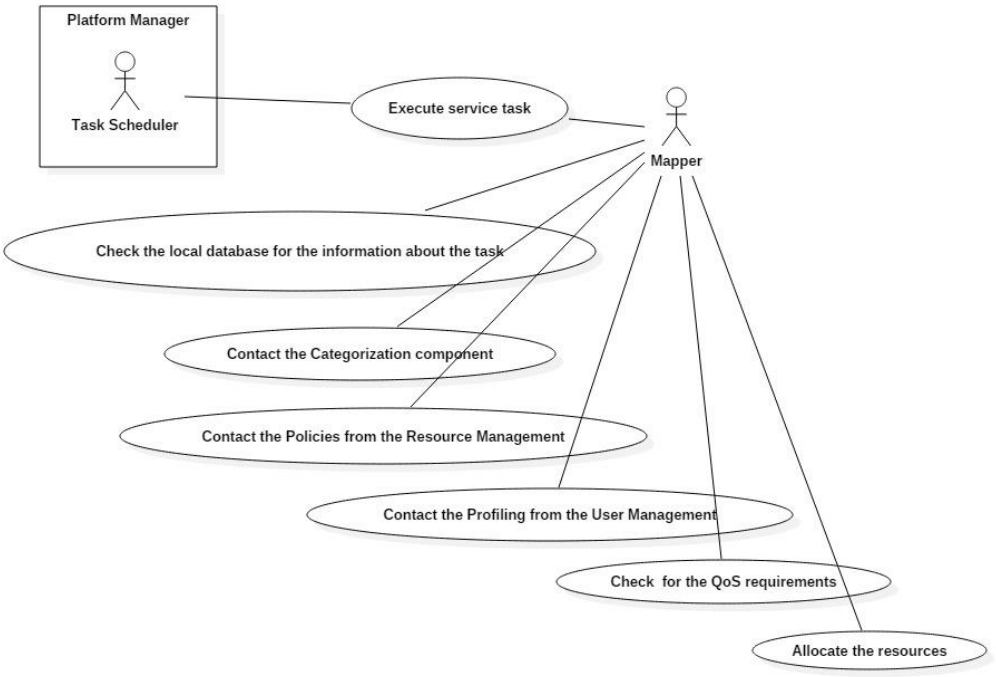

Figure 6 Use case diagram for the Mapping functionality – scenario 1

Scenario 1:

1. The mapping component (mapper) receives the task request from the Task Scheduler in the Platform Manager.
2. After that, mapper checks in the local database check if the requested task already exists, and in this scenario it returns a negative answer.
3. Mapper will then contact the Categorization component to categorize the received task according to its characteristics.
4. Contact the Policies component in the Resource Management in order to find out policies for the task.

5. Contact the Profiling component in the User Management in order to find out if there are user constraints for the task.
6. Check for the QoS requirements with the QoS provisioning component.
7. Contact the Allocation component to allocate the resources for the task.

Next figure shows the mapping use case-scenario 2:

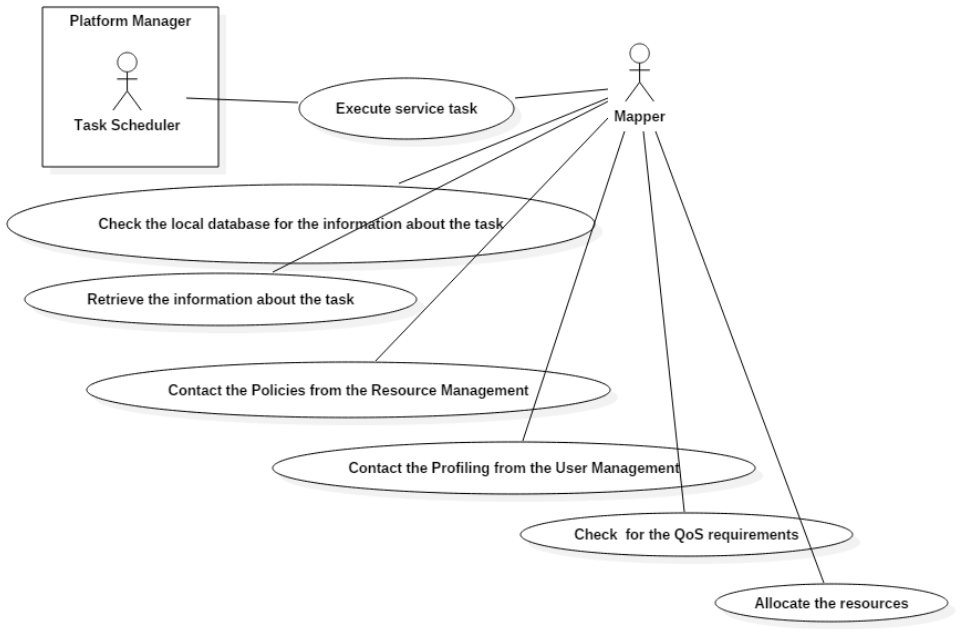

Figure 7 Use case diagram for the Mapping functionality – scenario 2

Scenario 2:

1. The mapping component (mapper) receives the task request from the Task Scheduler in the Platform Manager.
2. After that, mapper checks in the local database check if the requested task already exists, and in this scenario it returns a positive answer.
3. Retrieve the information about the task (it has already been categorized).
4. Contact the Policies component in the Resource Management in order to find out policies for the task.
5. Contact the Profiling component in the User Management in order to find out if there are user constraints for the task.
6. Check for the QoS requirements with the QoS provisioning component.
7. Contact the Allocation component to allocate the resources for the task.

3.2.3 Component detailed architecture

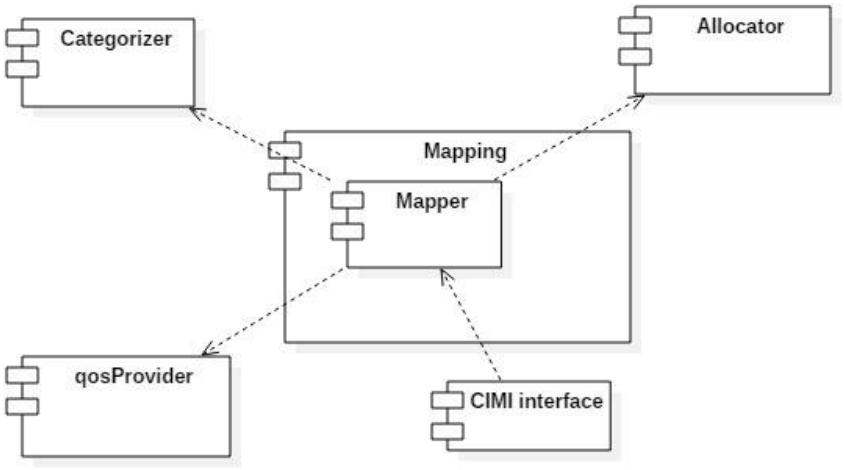

Figure 8 Component diagram for the Mapping functionality

The mapping component (Mapper in the Figure) is the only entry point for Service Management. It connects to the CIMI interface in order to receive the tasks from Platform Manager. After that it sends the tasks for further processing to the Allocator, Categorizer and qosProvider components.

Mapping class diagram

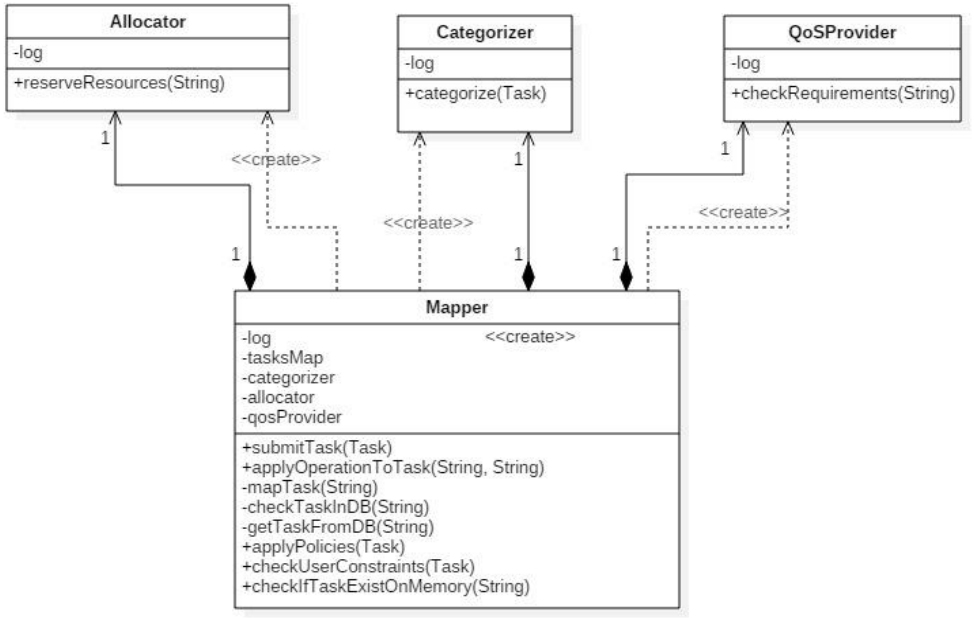

Figure 9 Class diagram for the main Mapping component

3.2.4 Internal sequence diagrams

The following diagrams show interaction between the mapping and other components in Service Management (the interaction with the components in Resource and User Management is not included). The first diagram shows the case when the task that is not in the database, so the first step is to categorize the incoming task (this process is shown in the service task categorization sequence diagram in 3.1.4), after which the mapping component will proceed with checking the

requirements with QoS provider and allocating the resources. The second diagram is the same except the task has already been in the database so there is no need for the categorization step.

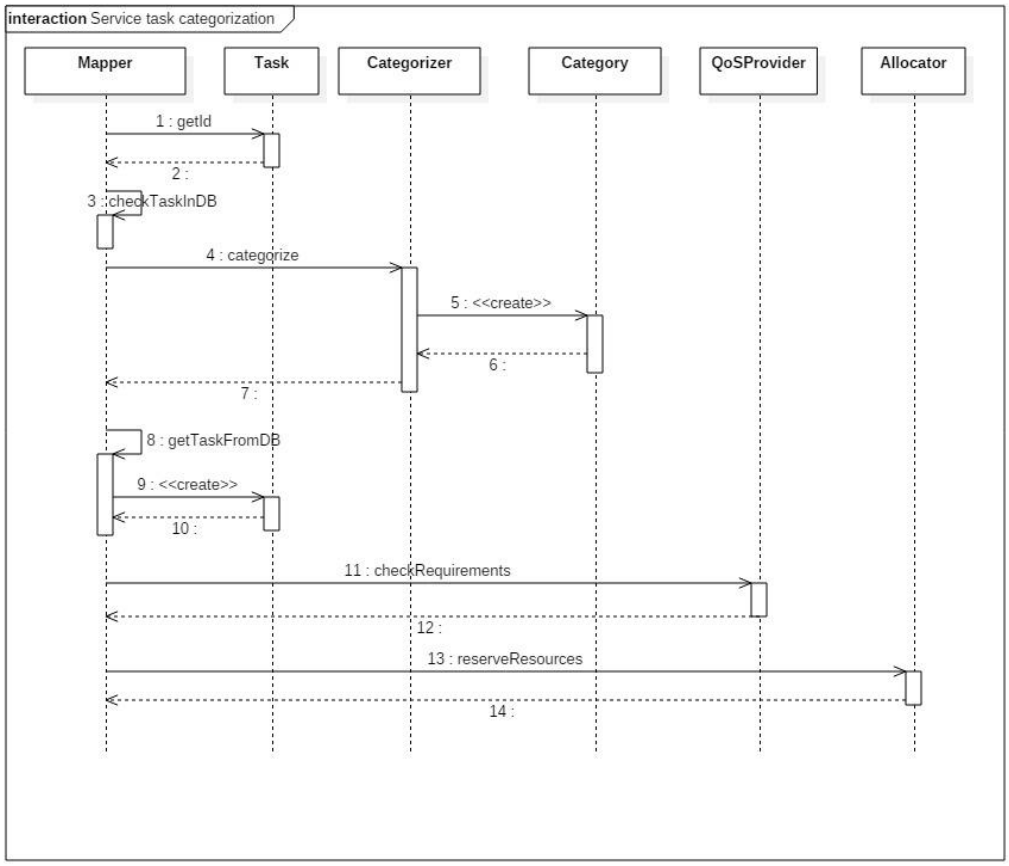

Figure 10 Sequence diagram for Mapping of a task that is not in the database

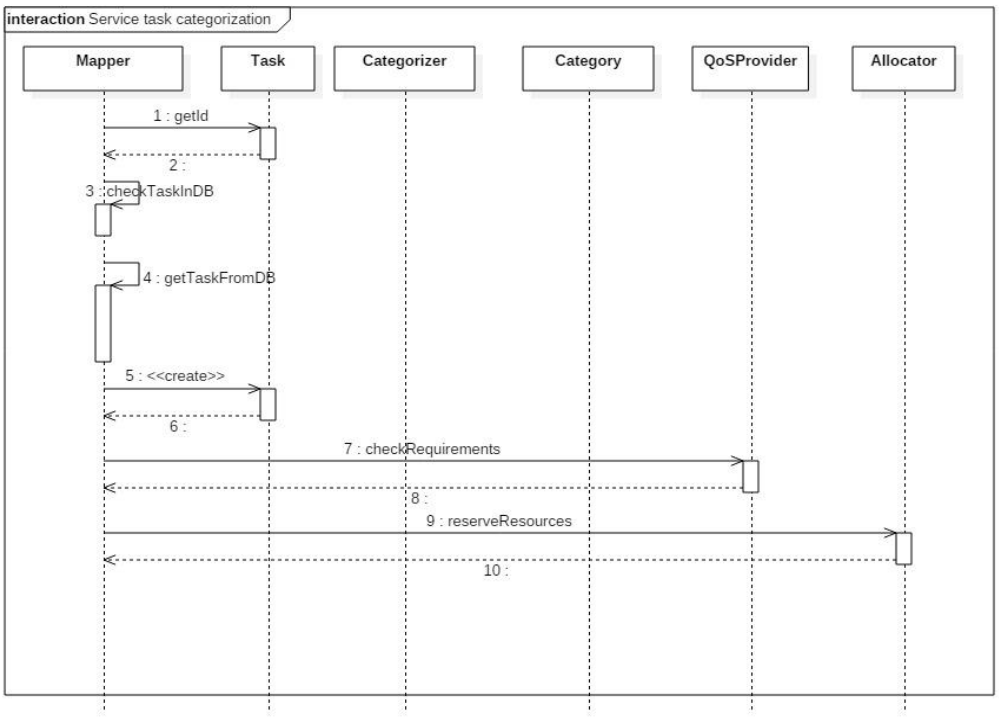

Figure 11 Sequence diagram for Mapping of a task that is already in the database

3.2.5 Data model

None for the first iteration.

3.2.6 Baseline technology

Technology	Usage	Reference
Java	Implementation language	http://www.oracle.com/us/technologies/java

Table 3. Mapping baseline technology

3.2.7 Test cases

This component will not be implemented during the first iteration.

3.3 Allocation

This subcomponent is responsible for the allocation of available resources to the various requests, trying to meet security and privacy rules, cost models, while guaranteeing overall optimal resources usage.

3.3.1 Requirements

The allocation subcomponent will communicate with other components only through the mapping subcomponent. This component is called upon in the cases when there is a possibility for choosing resources in the specific device of the AC, that is, when the distinct resources may be selected in the same device. Possible options of the AC for selecting resources are when either some virtualization is enabled or when the agent controller has attached computing devices non mF2C capable (without capacity to have the mF2C agent installed).

The allocation subcomponent should be able to reserve previously selected resources on the request from the Lifecycle model during the deployment phase. The Lifecycle module will first contact the mapping subcomponent who will then contact the allocation subcomponent. Also it can be contacted during the mapping process, in the runtime execution phase. In any case, all the communication will go through the mapping component, and its only functionality is to allocate available resources to the various requests, trying to meet security and privacy rules, cost models, while guaranteeing overall optimal resources usage.

3.3.2 Use case diagram

Next figure shows the allocation use cases:

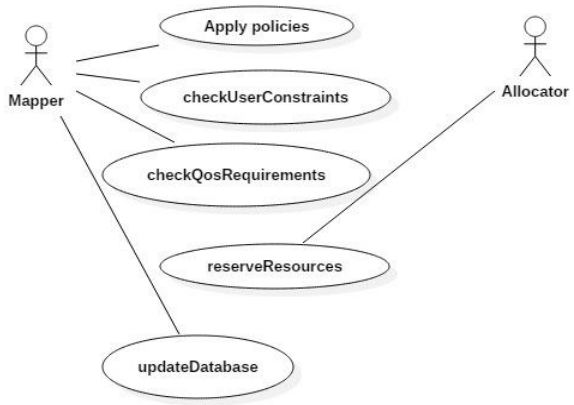

Figure 12 Use case diagram for the Allocation functionality

After task categorization the mapping component (mapper) contacts the Policies component in the Resource Management in order to find out policies that it needs to apply for each of the tasks, the Profiling component in the User Management in order to find out if there are user constraints for the task and check for the QoS requirements with the QoS provisioning component.

If all requirements are completed the allocation component can reserve the resources for the task.

The mapping component will update the database with the information about the allocated resources.

3.3.3 Component detailed architecture

Next figure shows Allocator component that at the moment has only the function of reserving resources after all the requirements have been fulfilled (each task has been categorized and checked if it satisfies Policies requirements from Resource Management, Profiling requirements from User Management and QoS requirements from the Service Management).

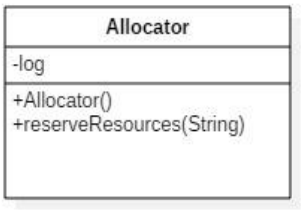

Figure 13 Component diagram for the Allocation functionality

Main Allocation class diagram

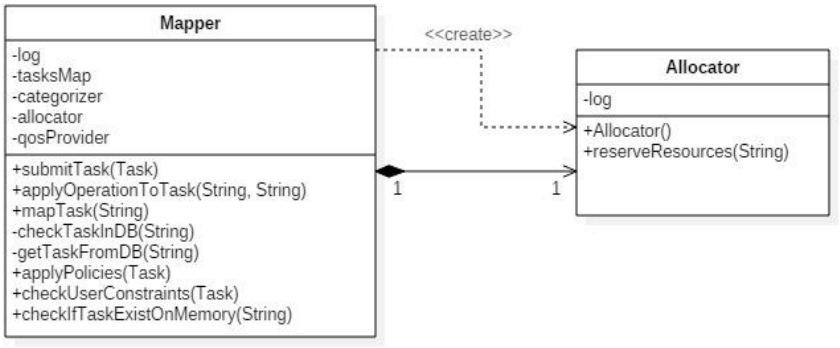

Figure 14 Class diagram for the Allocation component

3.3.4 Internal sequence diagrams

The following diagram shows the interactions between the allocation and other components. The direct interaction exists only with a mapping subcomponent that initiates the resource reservation once all the requirements have been checked.

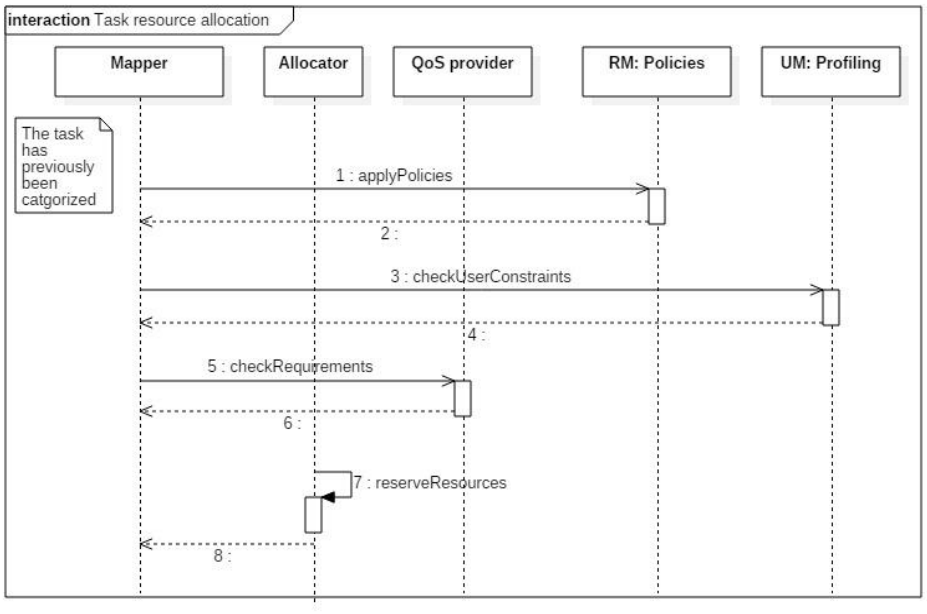

Figure 15 Sequence diagram for Task resource allocation

3.3.5 Data model

None for the first iteration.

3.3.6 Baseline technology

Technology	Usage	Reference
Java	Implementation language	http://www.oracle.com/us/technologies/java

Table 4. Allocation baseline technology

3.3.7 Test cases

This component will not be implemented during the first iteration.

3.4 QoS provisioning

The QoS provisioning subcomponent is responsible for QoS provisioning on a service task level. For each service, it will contact the SLA Management block in the Platform Manager to get information about the expected service.

3.4.1 Requirements

The QoS provisioning subcomponent will be contacted by the mapping subcomponent for each individual task that arrives from Task Scheduler. Since Task Scheduler is in the PM, the only component it can directly communicate from the Service Management is the mapping subcomponent. As one of the requirements it should be able to communicate with a higher instance component to get the information about the expected service. The parameters that should be defined will depend on the different service tasks, but for the first iteration the focus will be on the service execution time.

3.4.2 Use case diagram

None for the first iteration.

3.4.3 Component detailed architecture

Next figure shows QoS provisioning component. The component will not be implemented in the first iteration, since there was no agreement on how to define QoS requirements that could be used on a task level.

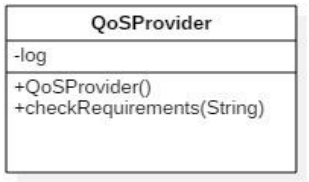

Figure 16 Component diagram for the QoS provisioning functionality

Main QoS provisioning class diagram

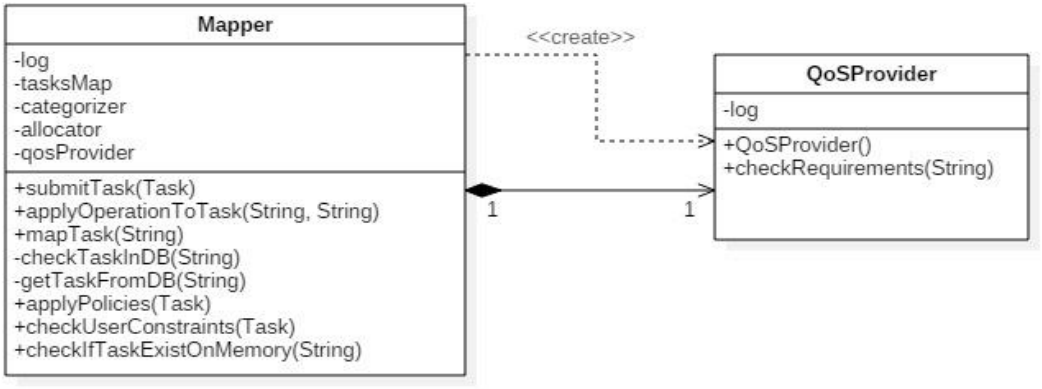

Figure 17 Class diagram for the QoS provisioning functionality

3.4.4 Internal sequence diagrams

None for the first iteration.

3.4.5 Data model

None for the first iteration.

3.4.6 Baseline technology

Technology	Usage	Reference
Java	Implementation language	http://www.oracle.com/us/technologies/java

Table 5. QoS provisioning baseline technology

3.4.7 Test cases

The component will not be implemented during the first iteration.

4. Resource Management

This section describes each one of the sub-components of the Resource Management block of the Agent Controller (AC). As stated in Section 2, the AC has a local view of the resources. However, due to the hierarchical nature of the proposed mF2C architecture, resources are grouped in clusters of devices, being one of these devices the leader. For this reason and despite the local scope of the AC, in the case of a device being leader, its 'local view' includes its own resources and the resources of the devices forming part of this cluster.

It is also worth mentioning that, although the PM includes all the smartness of the system, in the agent entity, the database is unique, and both, the PM and the AC share this database. For these two reasons, the hierarchical architecture and the shared database, one of the main functionalities of the AC regarding the resources can be summarized as:

- To fill the resource database, to be used by both PM and AC, with information about:
  - Own resources if the devices is part of the cluster but it is not a leader
  - Own resources and resources of the devices forming part of the cluster if the device is the leader of the cluster.

This database is managed by dataClay and we assume that each device will have a local database with its local information which is periodically copied/summarized (or following a policy still to be defined) to a Aggregated database, AGGR in Figure Figure 18, also in the device, which is also periodically (or following a policy still to be defined) synchronized with the local database of its leader. Finally, in the 3 hierarchical layer proposed for IT-1, the leader will also copy/summarize the information contained in its local database (about its own resources and resources of its children devices) to the leader's aggregated database, and this AGGR information will be periodically (or with a policy still to be defined) to the cloud leader.

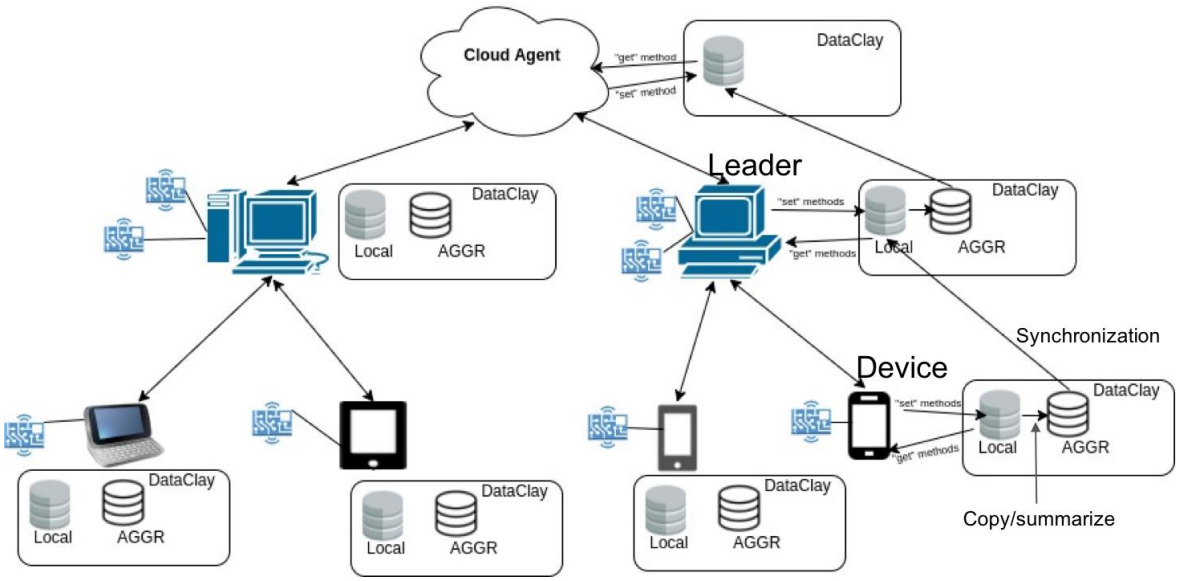

Figure 18 Leader and device databases synchronization

This basic functionality of filling resource database with information about local resources will require additional functionalities to make it work, such as: discovery, monitoring, identification, categorization and data management, described in the next sub-sections. We also describe an additional Resource Management sub-component, called Policy, which contains some of the policies followed by the rest of Resource Management sub-components to perform their functionalities.

4.1 Discovery

4.1.1 Requirements

Functional requirements

- The discovery component should allow the leader to advertise its presence
- The discovery component should allow the agent to detect a leader in its vicinity
- The discovery component should allow a leader to keep track of the discovery state of attached agents
- The discovery component should allow an agent to update its discovery state to the leader it is associated with
- The discovery component should provide an agent with the option to express its intention of leaving a leader's area
- The discovery component should allow the leader to stop beacon advertisements if the policies require so.
- The discovery component should allow a leader to detect beacons sent by neighbouring leaders
- The discovery component should allow a leader to detect when a new agent has joined the area as a result of receiving a beacon

Non-functional requirements

- The discovery component should guarantee short discovery times
- The discovery component should have a light footprint on the agent's device
- The discovery component should be able to scale to a potentially high number of agents

4.1.2 Use case diagram

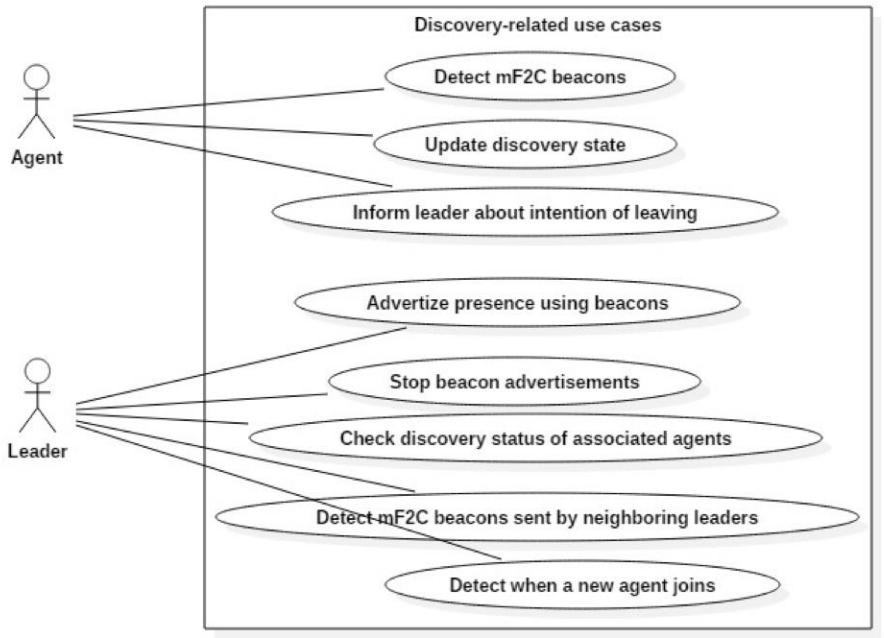

Figure 19 Use case diagram for the Discovery functionality

Figure 19 presents the discovery use case diagram, considering 8 use cases.

Use case #1

Title: Advertise presence using beacons

Actor: Leader

Scenario:

1. Leader encodes mF2C-related information in the proper information element (IE) format
2. Leader appends IE to other beacon fields
3. Leader starts broadcasting beacons according to a frequency that is retrieved from the policies.

Use case #2

Title: Detect mF2C beacons

Actor: Agent

Scenario:

1. Agent starts the scan for beacons
2. Agent goes through VSIEs contained in beacons and looks for the mF2C OUI
3. If found, the agent extracts the advertised attributes
4. Agent decodes them
5. Agent fills a list with the found leaders and their information

Use case #3

Title: Detect mF2C beacons sent by neighbouring leaders

Actor: Leader

Scenario:

1. Leader starts the scan for beacons
2. Leader goes through VSIEs contained in beacons and looks for the mF2C OUI
3. If found, leader extracts the advertised attributes
4. Leader decodes them
5. Leader fills a list with the found leaders and their information

Use case #4

Title: Check discovery state of the associated agents

Actor: Leader

Scenario:

1. Leader periodically sends the keepalive message to check the state of the associated agents it is managing.
2. Leader waits for the response: if no response is received from a particular agent within a predefined timeout, the leader assumes that it has left the area or that it is off for some reason.

Use case #5

Title: Update discovery state

Actor: Agent

Scenario:

1. Agent listens for keepalive messages
2. If the agent receives a keepalive message, it sends the leader an ACK message to acknowledge receipt of the keepalive

Use case #6

Title: Inform leader about intention of leaving

Actor: Agent

Scenario:

1. Agent explicitly sends “bye” message as a sign of its intention to leave the leader’s area.

Use case #7

Title: Stop beacon advertisements

Actor: Leader

Scenario:

1. Leader stops sending beacons as a result of a policy change

Use case #8

Title: Detect when a new agent joins

Actor: Leader

Scenario:

1. When an agent receives a beacon and becomes aware of the presence of the leader, it can decide to associate with the leader. The leader should detect when such an event happens.

4.1.3 Component detailed architecture

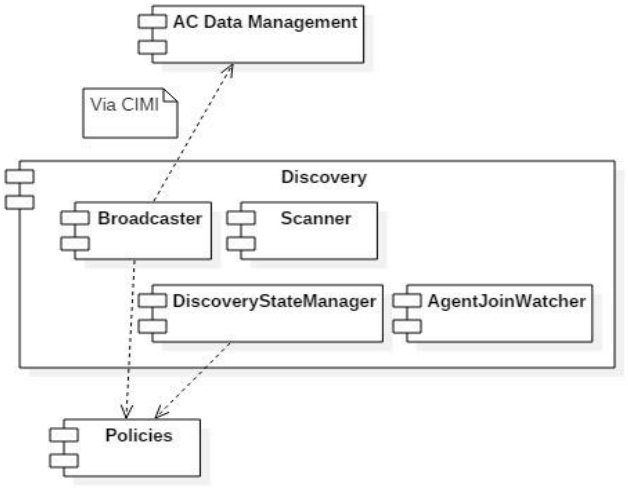

Figure 20 Component diagram for the Discovery functionality

The discovery component is comprised of the 3 following sub-components, as depicted in Figure 20:

- Broadcaster: This component provides the underlying mechanisms allowing a leader to advertise its presence via mF2C beacon broadcasts.
- Scanner: The scanner component allows an agent to scan for mF2C beacons and to properly decode the retrieved scan results
- DiscoveryStateManager: This component is in charge of keeping track of the discovery state of agents, either with regards to managing keepalive messages or to explicit disconnection requests made by agents towards the leader.
- AgentJoinWatcher: This component is executed at the leader side. It watches the event of a new agent associating with the leader and as a result forwards this information to be added to dataClay.

It is also dependent on inputs retrieved from dataClay and the policies module. For instance, the Broadcaster needs to retrieve the ID (stored by the Identification Component in dataClay) in order to transmit it within the beacon.

Main Discovery class diagram

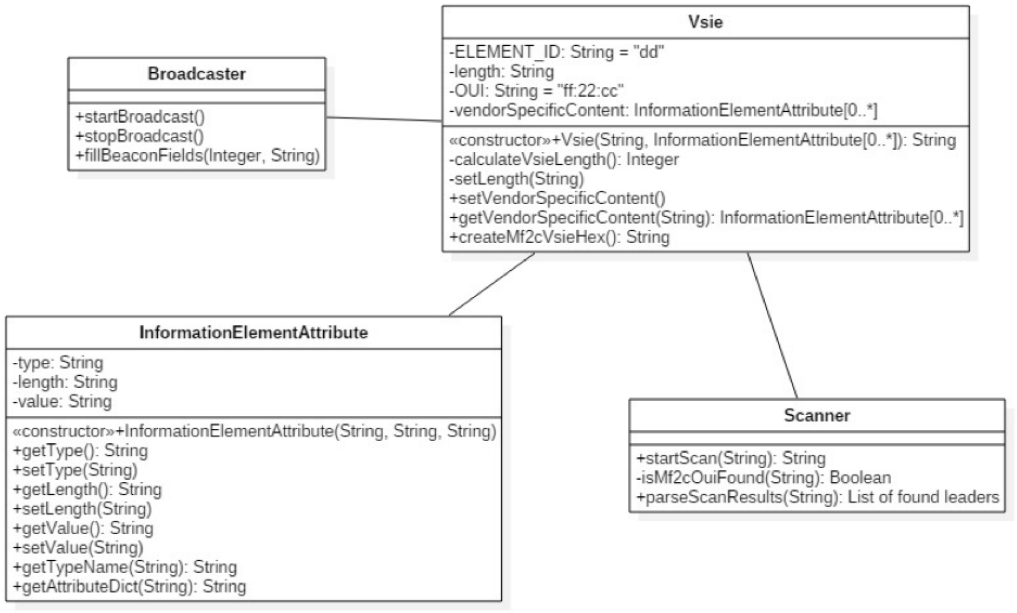

Figure 21 Class diagram for the Discovery functionality

As depicted in Figure 21, the main discovery class diagram is comprised of the Broadcaster class, the Scanner class, the Vsie class, as well as the InformationElementAttribute class. The Broadcaster class and the Scanner class use the Vsie class to embed a VSIE within the beacon and to parse its content, respectively. An object of the Vsie class is made up of a list of objects of the InformationElementAttribute class.

Broadcaster class

Methods:

fillBeaconFields(Integer,String): This method will fill the necessary fields into the beacon in order to prepare it to be sent. It will take as input the broadcastFrequency and the name of the wireless interface

startBroadcast(): this method will trigger the transmission of beacons

stopBroadcast(): this method will be used to stop beacon broadcasts

Scanner class

Methods:

startScan(String): trigger scan and retrieve corresponding results. Takes as input the name of the wireless interface

isMf2cOuiFound(String): search for the mf2c OUI in the scan results

parseScanResults(String): parse the scan results and return the list of found leaders, if any. Otherwise, output a message indicating that no leader was found.

Vsie class

It represents the characteristics of the Vendor Specific Information Element.

Attributes

ELEMENT_ID: a constant representing the vendor-specific element ID which is equal to DD in hexadecimal (221 in decimal)
length: the length in bytes of the subsequent fields of the VSIE
OUI: a constant representing the Organizationally Unique Identifier which is equal to ff:22:cc
vendorSpecificContent: a list of objects of the InformationElementAttribute class

Methods

Vsie(String,InformationElementAttribute[0..])*: creates a new Vsie object using the provided length and the list of the InformationElementAttribute objects that it will contain
calculateVsieLength(): calculates the length of the VSIE based on the attributes that it contains
setLength(String): sets the length of the VSIE in the corresponding field
setVendorSpecificContent(InformationElementAttribute[0..]):* sets the InformationElementAttribute objects to be included in the VendorSpecificContent field
getVendorSpecificContent(String):InformationElementAttribute[0..]:* gets the list of the InformationElementAttribute objects included in the VendorSpecificContent field
createMf2cVsieHex(): creates the final hexadecimal version of the mF2C VSIE

InformationElementAttribute class

It represents an information element attribute to be included in the vendor specific content field. It is represented as a TLV(Type-Length-Value).

Attributes:

type: type of the attribute
length: length in bytes of the attribute
value: actual value of the attribute

Methods:

InformationElementAttribute(String,String,String): creates a new InformationElementAttribute object, with the type, length and value as inputs
getType(): returns the type of the attribute
setType(String): sets the type of the attribute, for example setType("01") means this attribute represents the leader ID.
getLength(): returns the length of the attribute
setLength(String): sets the length of the attribute
getValue(): returns the actual value of the attribute
setValue(String): sets the actual value of the attribute
getTypeName(String): takes the type as an input and returns what this type represents, for example if it takes "01" as an input, it returns "Leader ID", meaning that this TLV represents a Leader ID
getAttributeDict(String): takes the actual attribute represented in hexadecimal and returns a dictionary-like structure where the type and the value of the attribute are extracted

DiscoveryStateManager class diagram

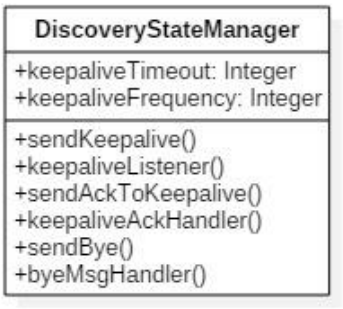

Figure 22 Class diagram for the DiscoveryStateManager sub-component

Attributes

keepaliveTimeout: if this timeout expires and an agent has not responded yet, the leader will consider it as disconnected

keepaliveFrequency: represents how often a keepalive message should be sent

Methods

sendKeepalive(Integer): executed by a leader to send a keepalive message according to the keepaliveFrequency

keepaliveListener(): executed by an agent to listen for keepalive messages

keepaliveAckHandler(): executed by a leader to handle acknowledgments to keepalive messages, taking into account the keepaliveTimeout

sendBye(): executed by an agent to express its intention of leaving a leader's area

byeMsgHandler(): executed by a leader to properly handle the outcome of the reception of "bye" messages

AgentJoinWatcher class diagram

Figure 23 Class diagram for the AgentJoinWatcher sub-component

Methods:

watchAgentJoinEvent(): executed by a leader to detect the event of an agent associating with it, as a result of receiving the beacon.

4.1.4 Internal sequence diagrams

Advertise presence using beacons

Figure 24 shows the process performed at the leader side in order to be able to advertise its presence using beacons. In fact, the fillBeaconFields method provided by the Broadcaster class will initially be called. This process involves creating InformationElementAttribute objects, representing all the mF2C information that needs to be advertised. Then a Vsie object will be created using the list of the previously-created information element attributes. In order to put this VSIE into the proper hexadecimal format, the createMf2cVsieHex method will be called. Once this process is finished, the startBroadcast method can be executed to effectively initiate beacon transmissions.

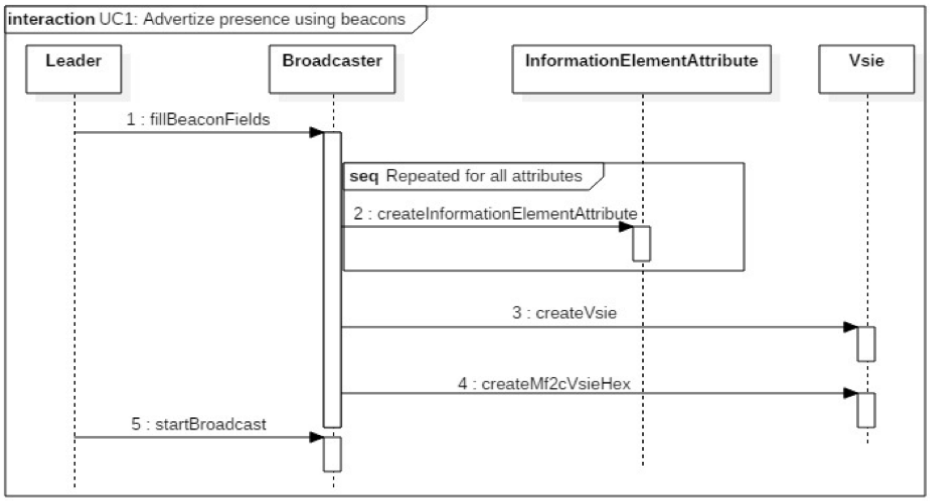

Figure 24 Sequence diagram for Advertise presence using beacons

Detect mF2C beacons

In order to be able to detect mF2C beacons, the agent performs a wireless scan, initiated by the startScan method of the Scanner class. The list of the retrieved results will be returned next to the agent, who will start to parse them, using the parseScanResults method. This process involves going through the list of all found results. Then, for each one of these results, the isMf2cOuiFound method will be called to check if the result has a vendor-specific information element (VSIE) containing the mF2C OUI. If such a VSIE is found, then the getVendorSpecificContent method will be called, returning a list of VSIE attributes (belonging to the InformationElementAttribute class). For each one of these attributes, the getAttributeDict method will be called twice, to get the real name of the attribute as well as the value that it contains.

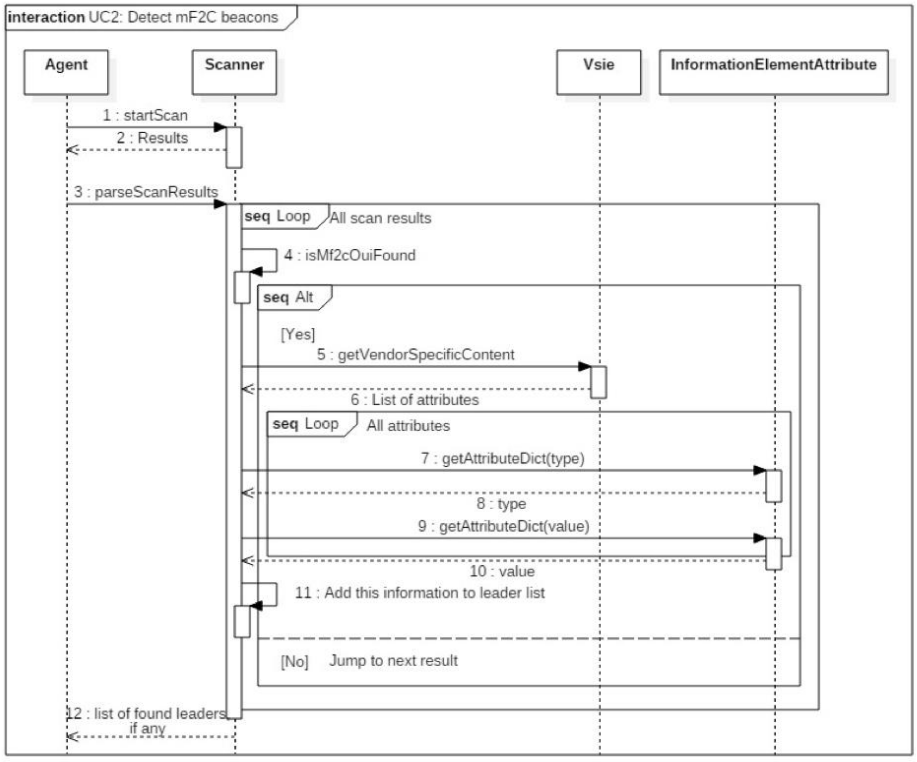

Figure 25 Sequence diagram for Detect mF2C beacons

This information will be added to the characteristics of the leader within the list of found leaders. If the mF2C OUI has not been found, then the following scan result will be examined in the same way, until all results have been processed. Consequently, the list of the leaders which have been found will be returned to the agent. Otherwise, a message indicating that no leader was detected will be shown to the agent.

Detect mf2c beacons sent by neighbouring leaders

The same process as UC2 takes place.

Check discovery state of the associated agents

The leader periodically sends keepalive messages using the sendKeepalive message. In the meantime, the agent has a keepaliveListener method running. Upon receipt of a keepalive message, the agent will send an acknowledgement to the leader. The keepalive ackHandler method is executed by the DiscoveryStateManager at the leader side to properly handle the received acknowledgements, if any or to infer that the agent is off if the predefined timer has elapsed. In either case, it will update the state of the agent into dataClay.

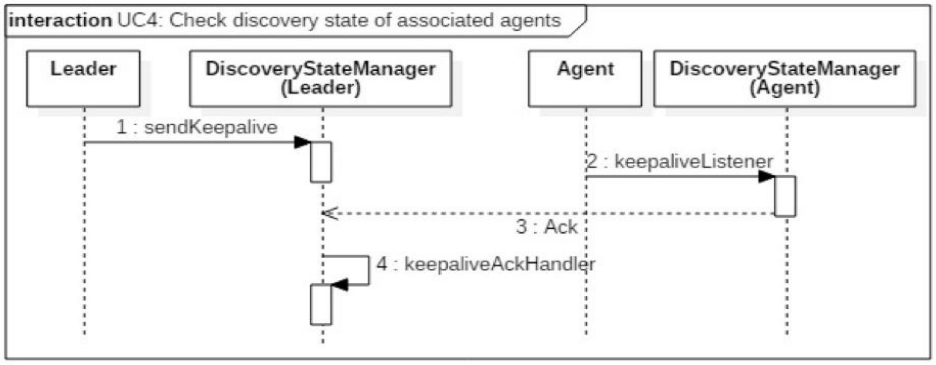

Figure 26 Sequence diagram for Check discovery state of associated agents

Update discovery state

This use case is included in the previous sequence diagram (Figure 26).

Inform leader about intention of leaving

When an agent decides to leave a leader's area, it will call the sendBye method. When the leader receives such a message, the ByeMsgHandler method will be called, resulting in updating the entry corresponding to that agent in dataClay.

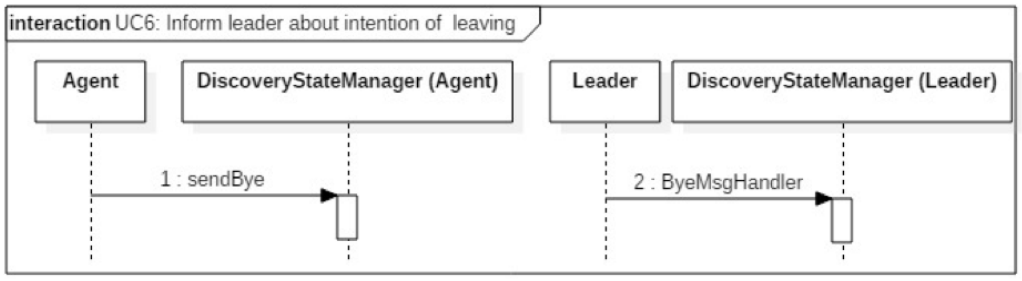

Figure 27 Sequence diagram for Inform leader about intention of leaving

Stop beacons advertisements

As there may be periods when broadcasting beacons are temporarily not needed, a call to the stopBroadcast method could be initiated by the policies component, resulting in disabling beacon transmissions.

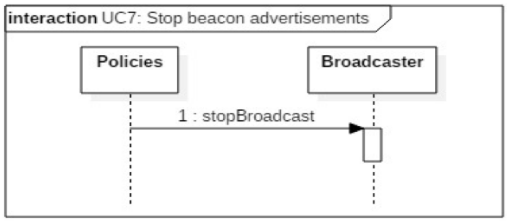

Figure 28 Sequence diagram for Stop beacon advertisements

Detects when a new agent joins

Figure 29 depicts the interaction occurring when a new agent joins a leader's area. In fact, the watchAgentEvent method is continuously called in order to detect when a new agent associates with a leader, as a result of receiving a beacon. If such an association occurs, this update is propagated to dataClay to store information about the newly-added agent.

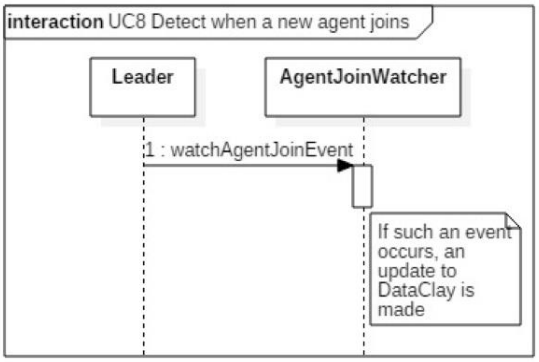

Figure 29 Sequence diagram for Detect when a new agent joins

4.1.5 Baseline technology

Technology	Usage	Reference
Python	Implementation language	https://www.python.org/
iw	Iw is a Linux wireless configuration utility. It is the underlying tool used by the Scanner sub-component to perform a wireless scan and get the corresponding results. Amongst others, these results include Vendor-Specific Information Elements, which have been advertised within beacons and probe response frames. These VSIEs are inspected in order to find out if they contain the mF2C OUI.	https://wireless.wiki.kernel.org/en/us use/documentation/iw

hostapd	Hostapd is a Linux access point implementation. It allows the Broadcaster sub-component to broadcast beacons. It has a built-in option to configure vendor-specific information elements, which we use to embed mF2C-related information that will be advertised by the leader	https://wireless.wiki.kernel.org/en/users/documentation/hostapd

Table 6. Discovery baseline technology

4.1.6 Test cases

In this section we define a set of test cases for the Discovery functionality, based on the use cases previously described.

Test Conditions #1 for Use Case #1: Leader advertise presence using beacons

1. Test that the leader encodes mF2C related information in the proper IE format
2. Test that the leader appends IE mF2C related information in the proper IE format
3. Test that the leader call for broadcasting beacons mF2C in the right frequency

Test Conditions #2 for Use Case #2: Agent detects mF2C beacons

1. Test that the agent call for starting scan for beacons
2. Test that the agent appends IE mF2C related information in the proper IE format
3. Test that the agent call for broadcasting beacons mF2C in the right frequency
4. Test that the agent can decode the given attributes (scenario 2.3)
5. Test that the agent, given the leaders and their info, fill a list with that information in the right format

Test Conditions #3 for Use Case #3: Leader detects mF2C beacons sent by neighbouring leaders

1. Test that the leader start scanning beacons
2. Test that the leader looks for the mF2C OUI through the beacons VSIEs
3. Test that the leader, having the OUI (scenario 3.2), can extract the advertised attributes
4. Test that the leader is able to decode the given attributes (scenario 3.3)
5. Test that the leader is able to fill the list using the right format with the found leaders and their info (see scenario 3.4)

Test Conditions #4 for Use Case #4: Leader checks discovery state of the associated agents

1. Test that the leader periodically sends keepalive message to check the state of the associated agents it is managing.
2. Test that the leader, if there is no response from a particular agent within a predefined timeout, then assumes that the agent has left the area or that it is off for some reason.

Test Conditions #5 for Use Case #5: Agent updates discovery state

1. Test that the agent listens for keepalive messages
2. Test that, if the agent receives a keepalive message, it sends the leader an ACK message to acknowledge receipt of the keepalive

Test Conditions #6 for Use Case #6: agent informs leader about intention of leaving

1. Test that the agent explicitly sends “bye” message as a sign of its intention to leave the leader’s area.

Test Conditions #7 for Use Case #7: agent stops beacon advertisements

1. Test that the leader stops sending beacons as a result of a policy change

Test Conditions #8 for Use Case #8: leader detects when a new agent joins

1. Test that the leader detects the decision of an agent of associating with him after receiving the beacon.

4.2 Policies

The policies will be a set of rules to be applied and used by different blocks in the Agent Controller. This set of rules could be changed by the PM, according to a high level policy managed by the PM.

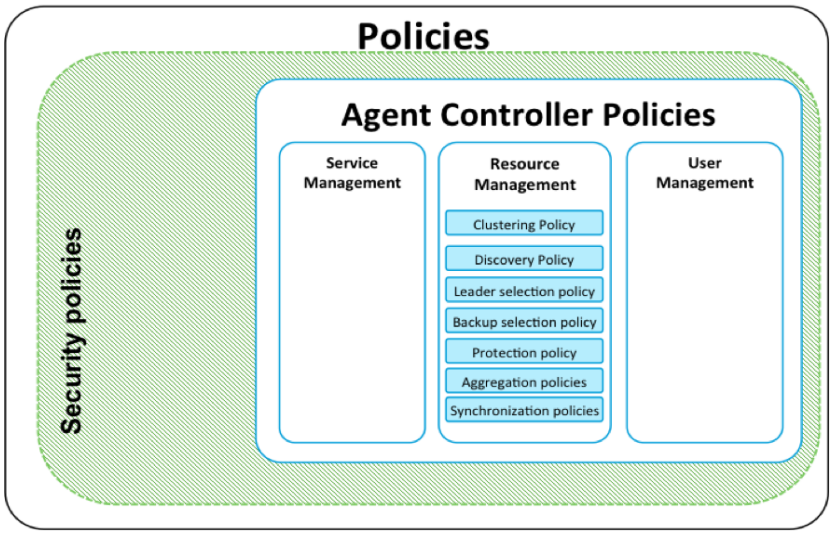

Figure 30 Policies component in the Resource Management block

The identified policies regarding the resources are:

- Clustering policy: It is the policy used to group devices in a cluster under the management of a leader. It could be by proximity, connectivity, etc. For IT-1 we will follow a simple policy:
  - The clustering is determined by the attachment with a leader. If an agent receives the beacon from a leader and acknowledges participating in the mF2C system, it will belong to the cluster of that leader. That is, an agent will participate in the cluster of the first leader from which it receives a beacon.
- Discovery Policy: The discovery policies includes the policies related to the frequency of advertisement beacons to find new "children" as well as the frequency of the "keep alive" messages to know if the leader's children are still there. In a first approach for IT-1, we propose the next simple policies:
  - Beacons sent by the leader to discover new 'children' are sent periodically with a period T_B **.** This period can be adjusted changing the value. This adjustment should be done by the Policies block in the PM.
  - "Keep alive" messages sent by the leader are sent periodically with a period T_K. This period can also be adjusted by the Policies Block in the PM.
- Leader selection policy: It is the policy to select which device is the leader among a cluster of devices. For IT-1 we will follow a simple policy completely related to the policy used to cluster the devices:
  - The leader will be randomly (and manually the first time) selected among the well-known fixed nodes (nodes without mobility). Once the leaders are selected, the cluster of devices will be determined by the devices that these leaders can discover.

- Backup selection policy: It is the policy to select which device is the backup among a cluster of devices. For IT-1 we will follow a simple policy completely related to the policy used to cluster the devices:
    - The backup will be randomly (and manually the first time) selected among the well-known fixed nodes (nodes without mobility).
- Protection policy: It is the policy followed when a leader fails to recover the leader's database, guaranteeing that the required information stored in a leader is accurately shifted from a leader to a backup, reacting to a leader failure. We propose three protection policies:
    - Zero-Knowledge (ZK) strategy. In this strategy when the leader fails, the backup node detects this fail and sends messages to the rest of nodes asking for their state information to update its own database, becoming the backup the new leader.
    - Keep Updating (KU) strategy. In this strategy the backup node has a copy of the leader database, which is periodically updated. When the leader fails it is not necessary to send the leader's database to the backup node because it already has all the information. However, it requires an additional policy setting the periodicity for the synchronization between the leader's and the backup's databases.
    - High-Layer download (HLD). In this strategy a copy of the leader's database is maintained in a node in an upper layer of the hierarchy, which also requires the periodic update and synchronization between both databases. When the leader fails the backup node must download the leader's database.
- Aggregation policies: These are the set of policies used to copy/summarize the content of the local database to the aggregated database (AGGR). Notice that this AGGR database is the one that upper layer can see, that is, the one shared with the leader. These aggregation policies should different in the different layer of the hierarchy. For IT-1, one propose the next set of simple aggregation policies:
    - For layer 2: (Normal agents) The aggregation will be simply the COPY. The content of the local database only contains information about the device itself, and it is copied to the AGGR database which is shared with the leader.
    - Fog layer 1: (Leaders) In this case in the local database the leader has information about its own resources, as well as, information about the resources of its "children". In the leader's local database will an entry for the leader itself and also for each one of its 'children'. However, for the aggregated database there will be only one entry (Notice this is the information that cloud agent in layer 0 sees). In this unique entry there will be aggregated information about the leader and its children, A simply aggregation policy for this layer could be:
        - For quantifiable information such as CPU, Storage, etc the aggregated value will be the addition of the values of the leader and all its children.
        - For qualitative parameters such as type of IoT attached devices (sensors, actuators, cameras, etc.) the value will be the union (U) of all the values of the leader and its children. In Figure 31 we show an example of this aggregation policy in the leader.
    - Fog layer 0: (Cloud agent) The local database of the cloud agent will have information about resources in cloud, as well as resources of its children which are leaders. In the architecture proposed in IT-1 we proposed to have only a cloud, then in this case it is not necessary to have an aggregated database (AGGR) to share with

other clouds. But in any case, for homogeneous design we decided to maintain both databases in the entire layer, Local and AGGR. Then, for IT-1, the simplest aggregation policy can be also: COPY.

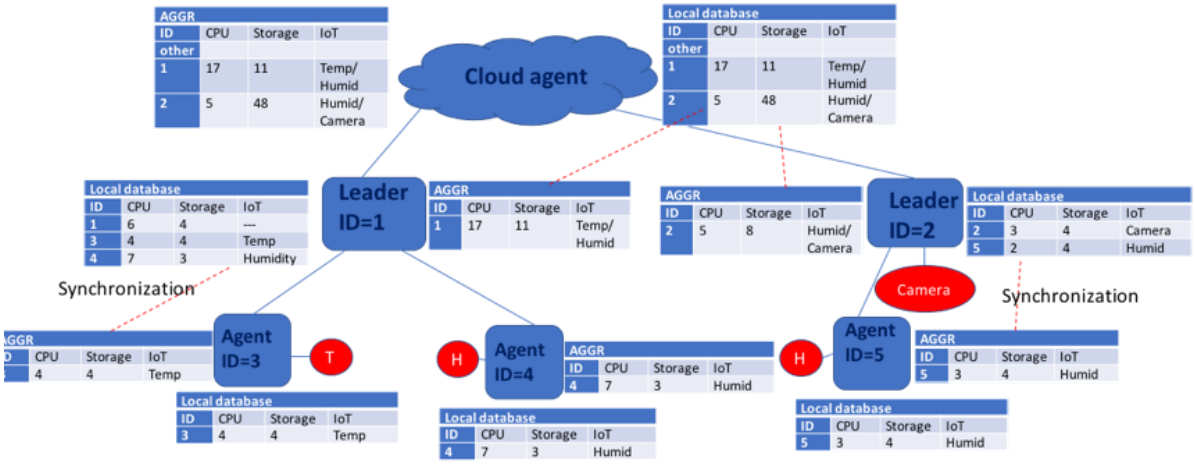

Figure 31 Aggregation policies example

In Figure 31 we can see that each agent has a local database and an aggregate database (AGGR) which is the one synchronized with the higher layer local database. In order of simplifying the example we have considered only three type of parameters: CPU, Storage and IoT. CPU and Storage are quantifiable, but IoT we consider that is a qualitative parameter, with information about the sensors, actuators, cameras, etc. that the agent has embedded or attached.

In the case of normal agent we observe that both databases, local and aggregate, contain the same information, because the aggregation policy is COPY. The same happens with cloud agent, local database and aggregate database have the same content.

However in the case of leaders (layer 1), the local database contains an entry for its own resources, but as many entries as children agents the leader has. On the other hand, applying the policy of aggregating by means of a sum the quantifiable parameters and the union (U) of the qualitative parameters, in the aggregate database we have only one entry with this aggregated information. For example, for leader 1, for CPU and Storage we have the added value of all the CPU and Storage values of the leader and its children (17 and 11), whereas for IoT the leader informs that in its cluster it has temperature sensor and humidity sensor. This aggregated information, with only one entry per leader, is the one that the cloud agent sees, and which is synchronized with the local database in the cloud agent, as it can be observed in Figure 31. The cloud agent in the local database has entries (specified by 'other' in the table) for its own information, as well as an entry for each one of the leaders it is controlling.

In the case of the leader, in the future, we will try to improve the aggregation of quantifiable resources, trying making some range-based window for them. That is, in the previous example, and in order of leaving a margin for inaccuracies due to the mobility/intermittent connectivity of agents, we do not consider the exact sum of 17 units of CPU in leader 1, but we can consider it is in a range between 10-15.

- Synchronization policies: These policies determines both, when the information in local database is aggregated and passed to the AGGR database, and also when AGGR database is

synchronized with local database in higher layer (see Figure 18). In a simple approach for IT-1, we propose the next policies:

- Synchronization between local and AGGR databases are performed any time there is a change in resources.
- Synchronization between AGGR database and local database in higher layer is performed periodically or after N changes in resources; and synchronization between AGGR database in leader and local database in cloud is performed with the same policy (after T time or N changes) but also when a leader discovers a new device.

- User Management Assessment's Warnings policy: It is the policy that defines when to send a warning to the Service Orchestration component, after the Assessment module detects that one or more applications are using more resources than defined by the user.

4.3 Identification

4.3.1 Requirements

The identification component is divided into two sub-components, the registration and the identity management. While the registration takes place in a cloud server, the identity management is executed by the agent in the resource itself.

During the registration phase the IDKey is generated (or recovered in the case of already registered users) and delivered to the user according with the chosen registration method. Lately, that IDKey is used as input during the calculation of the resource identifier (ID) in the identity management sub-component.

The objectives of this module are to provide every device participating in the mF2C network with a globally unique ID that allows to recognize a specific resource unambiguously and to establish mechanism to update and/or revoke the resource ID.

4.3.2 Use case diagram

Under this section the possible user cases that may take place during the identification process are shown in Figure 32 (2 for registration and 3 for the identity management).

Use case #1 – Registration / IDKey recovery through the webpage

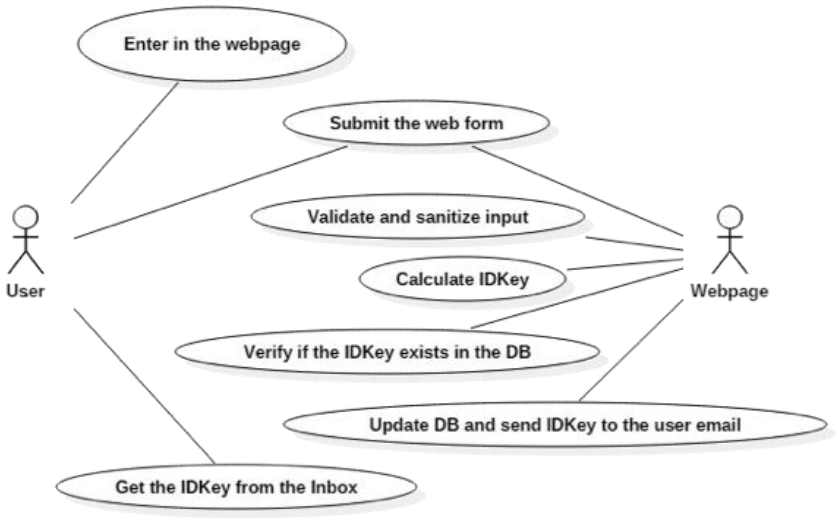

Figure 32 Use case diagram for Registration

Actors: User, WebPage

Steps:

1. The user enters in the webpage, fill the form with his email and submit it.
2. The webpage validates that the email address provided by the user meets the standard format and if it necessary sanitizes the string.
3. The user email is concatenated with a random string and used as input of a hash function to calculate the IDKey.
4. The webpage verifies whether the IDKey is already stored in the database or not. If it not stored then it is a new user registration and a new row is created with the corresponding information, otherwise it is a recovery. In that case the field “last idkey recovery” in the user IDKey row is updated with the actual date-time.
5. The IDKey is sent to the user via email.
6. The user gets his IDKey from his email.

Use case #2 – Registration / IDKey recovery through the webservice

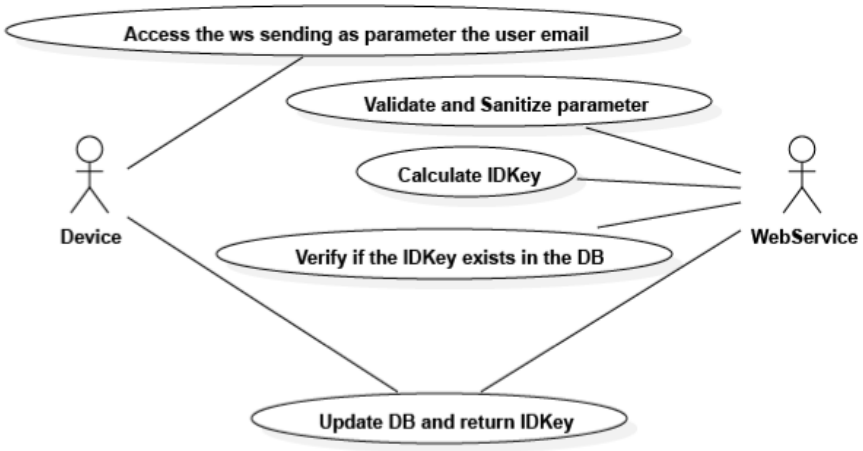

Figure 33 Use case diagram for Registration using webservice

Actors: Device, Webservice

Steps:

1. The device accesses the web service and sends the stored user email as parameter.
2. The web service validates that the email address provided by the device meets the standard format and if it necessary sanitizes the string.
3. The user email is concatenated with a random string and used as input of a hash function to calculate the IDKey.
4. The web service verifies whether the IDKey is already stored in the database or not. If it not stored then it is a new user registration and a new row is created with the corresponding information, otherwise it is a recovery. In that case the field “last idkey recovery” in the user IDKey row is updated with the actual date-time.
5. The IDKey is returned to the device.

Use case #3 – Resource Identifier Calculation

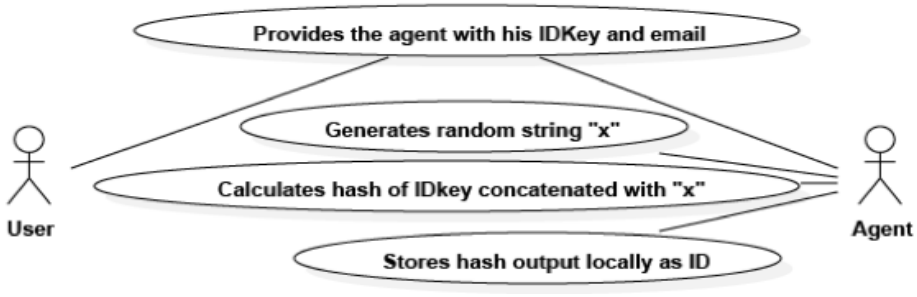

Figure 34 Use case diagram for Resource ID calculation

Actors: User, Agent

Steps:

1. The user provides the mF2C agent with his IDKey and email address.
2. The agent generates a random string “x” that concatenates with the IDKey.
3. The agent uses the hash function with the concatenated string as input and the resulting hash output is stored locally as ID of the res. where the agent is running.

Use case #4 – Resource Identifier Update

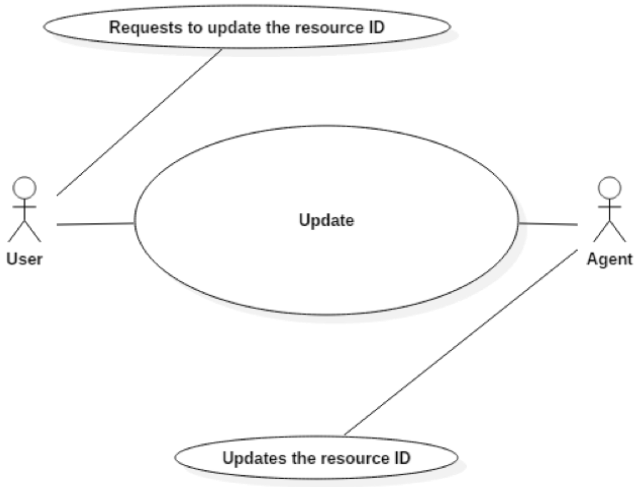

Figure 35 Use case diagram for Resource ID update

Actors: User, Agent

Steps: Process under discussion, this use case will not be implemented in IT-1.

User case #5 – Resource Identifier Revocation

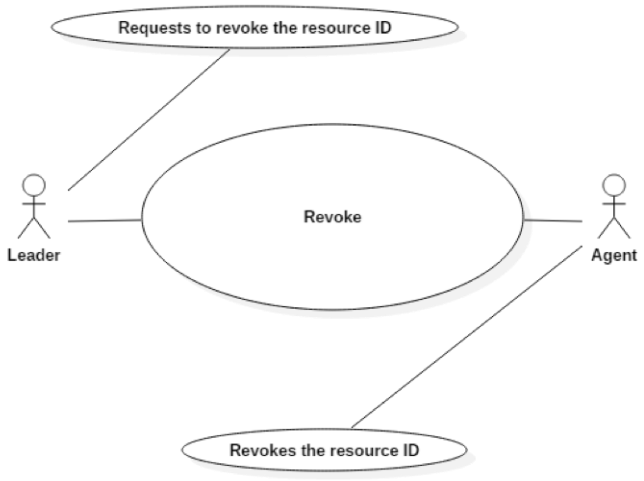

Figure 36 Use case diagram for Resource ID revocation

Actors: User, Agent

Steps: Process under discussion, this use case will not be implemented in IT-1.

4.3.3 Component detailed architecture

The three functionalities required in the identity management will be contained in the class named Identification. The Identification class will implement one method for each functionality, as shown in the next Figure 37.

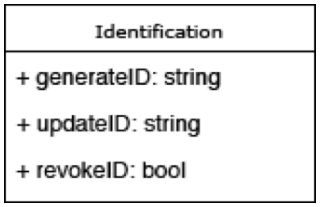

Figure 37 Identification class

Methods

generateID: this method will take as input the user IDKey and will produce a globally unique resource identifier. The output will be the resource ID.

updateID: in case the user requires to change the resource ID, this method will be in charge of perform that action. In order to update a resourceID, the method will need as input the old resource ID and the user IDKey. Finally, the output of this method will be the new resource ID.

revokeID: The last method will be responsible of revoke resources ID. This action can be requested by the resource owner, the leader agent or even the agent running in the cloud. The input of the method is the actual ID and the output is a Boolean value.

4.3.4 Internal sequence diagrams

In this subsection the sequence diagrams of the use cases related with the identification process are presented.

Registration / IDKey recovery through the webpage

During the registration or IDKey recovery process through the webpage the user interacts with the webpage and after some background tasks the webpage sends the IDKey to the user.

The interactions between the user and the webpage are those marked with the numbers 1, 2, 3 and 9 in the diagram below. In those interactions the user requests the webpage and a webserver responds sending the html registration form. Once the user has filled the required information in the web form he/she submits it, the webpage performs a set of background task.

The first task the webpage does is to validate and sanitize the user input, that is, to verify that the user input corresponds with a valid email format and to remove invalid characters such as blank spaces and the beginning or ending of the string, etc.

After the input validation, the webpage calculates the user IDKey using the SHA-512 algorithm. Once it is calculated, the webpage checks if the generated IDKey already exists in the database. If it exists the lastTimeRecovered field is updated accordingly, otherwise a record for the new registration is created in the database.

The last action, the webpage does, is to send IDKey to the user's email, from where the user can download it.

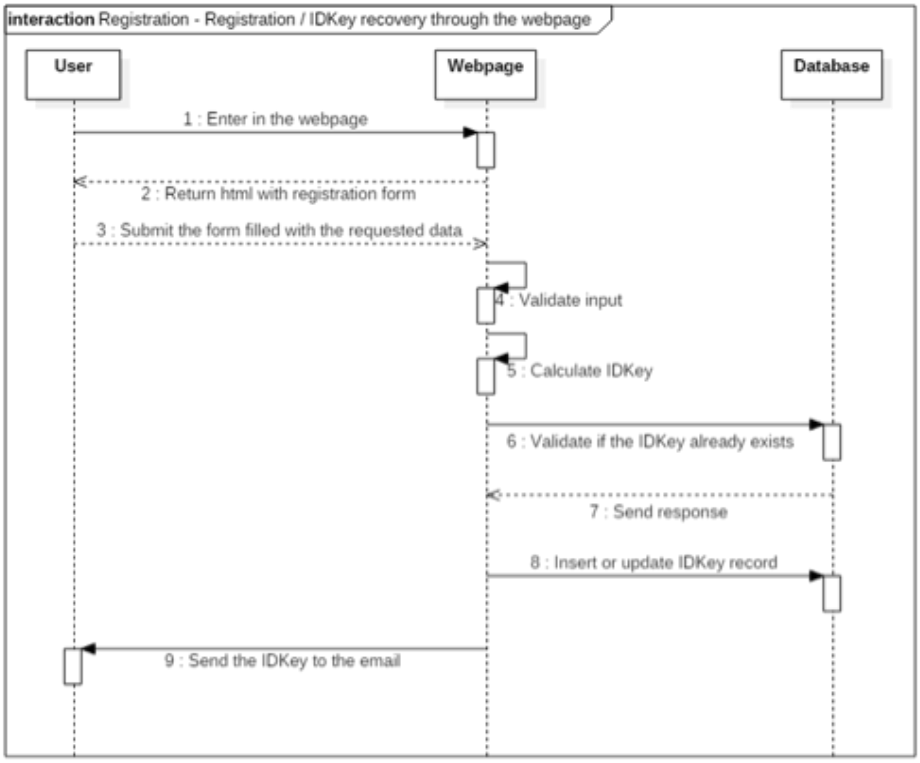

Figure 38 Sequence diagram for Registration/IDKey recovery through the webpage

Registration / IDKey recovery through the webservice

In the registration / IDKey recovery through the web service a device takes the user's place and a web service executes the same set of tasks that the webpage in the previous sequence diagram.

The main difference with the Sequence diagram #1 is that in this one there is not human intervention and thus, the process can be automatized.

As can be seen in the next diagram, in the first message the device sends to the web service the user's email is attached.

Once all the tasks that the web service must run have been completed, the web service sends the IDKey to the requesting device.

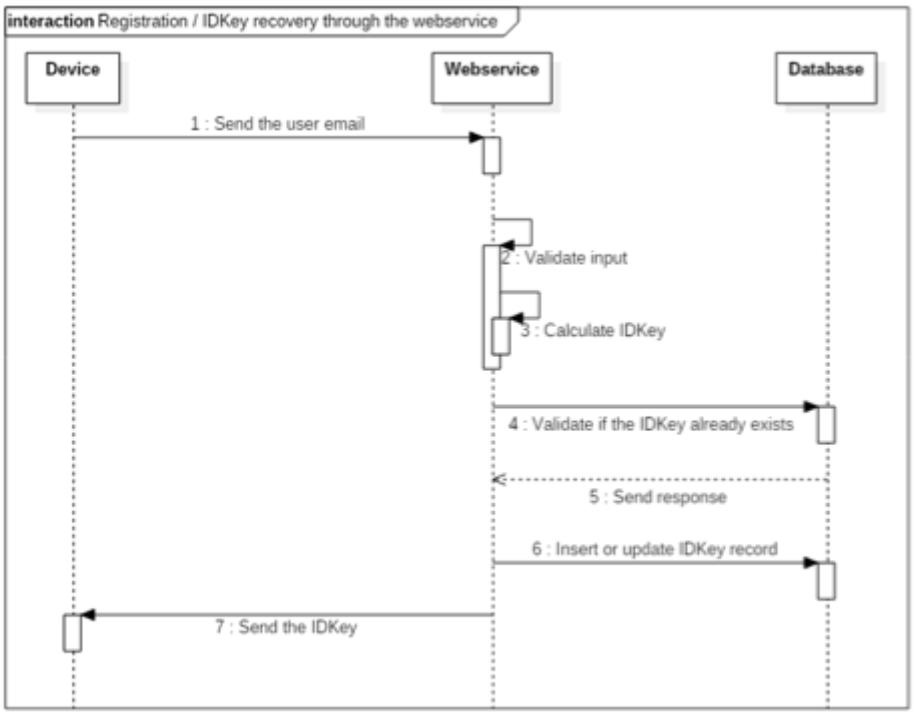

Figure 39 Sequence diagram for Registration/IDKey recovery through the webservice

Resource Identifier Calculation

In order to allow the agent to calculate the resource ID, the user provides the obtained IDKey during the registration process as well as his/her email.

When the agent receives the user's email and the IDKey, it generates a random string (x) and concatenates it with the IDKey. The resulting string is used as an input to the hash algorithm and the hash output is stored as the resource ID in the local database.

Finally, the user receives a confirmation notifying that the process has finished successfully.

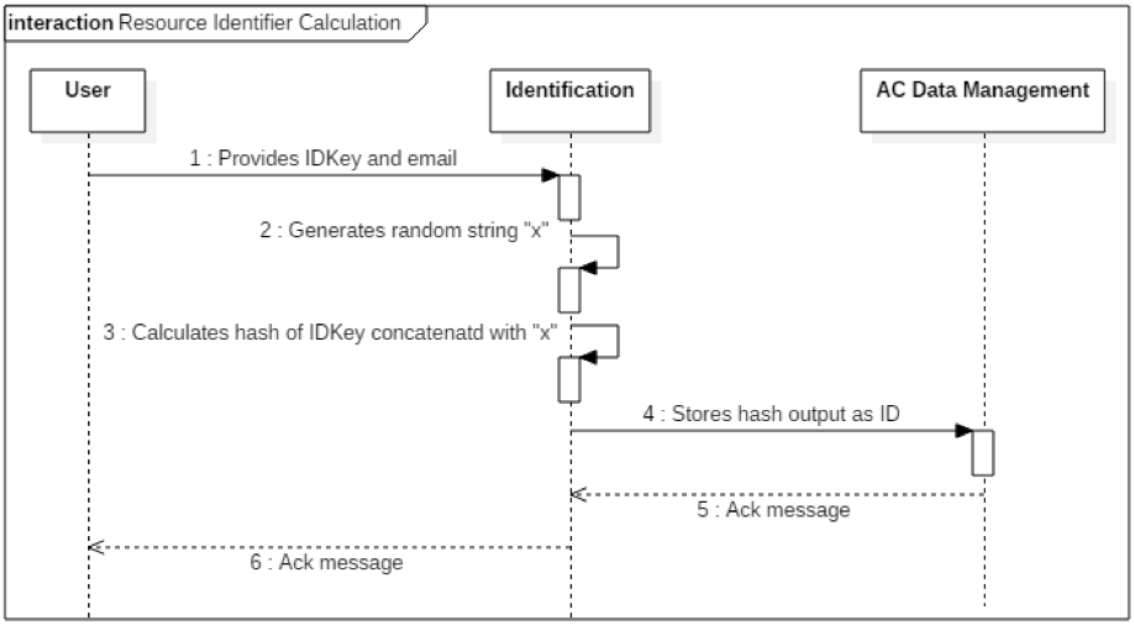

Figure 40 Sequence diagram for Resource identifier calculation

Resource Identifier Update

The exact sequence for this task is under discussion. In the next figure we present a first approach for solving this task.

In the diagram the user requests to the agent to update the resource ID. The agent attends the request modifying accordingly the corresponding field in the database.

Finally, the user receives a confirmation notifying that the process has finished successfully.

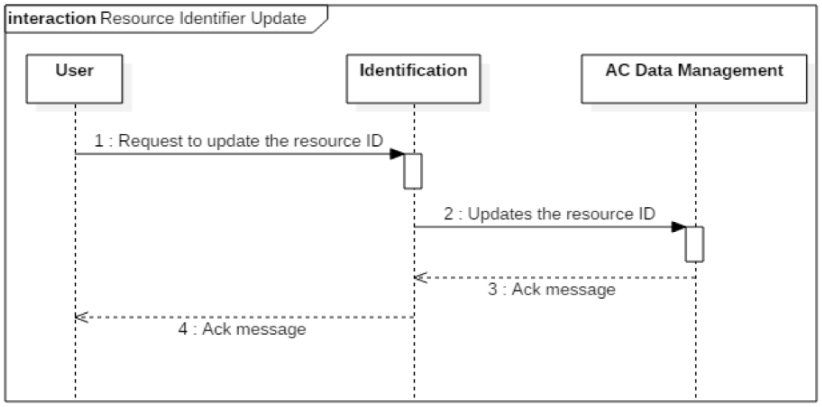

Figure 41 Sequence diagram for Resource identifier update

Resource Identifier Revocation

The exact sequence for this task is under discussion. In the next figure a first approach for solving this task is presented.

In the diagram the leader requests to the agent to revoke the resource ID. The agent attends the request deleting accordingly the corresponding field in the database.

Finally, the leader receives a confirmation notifying that the process has finished successfully.

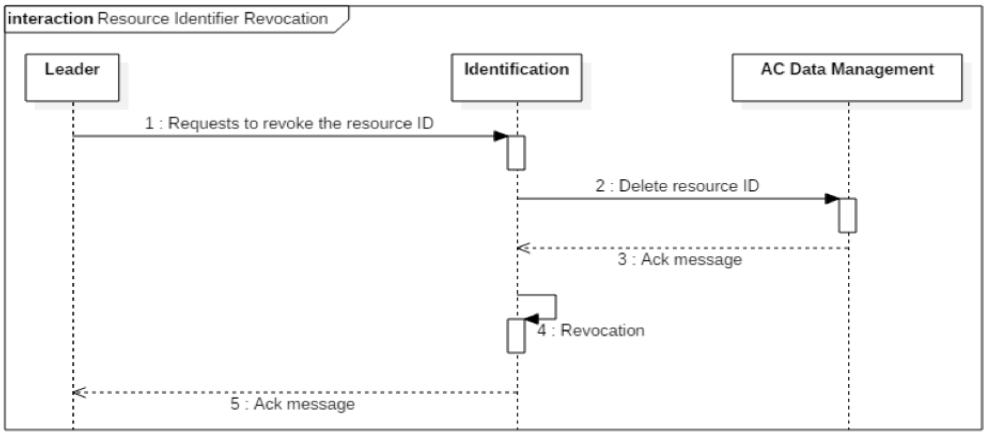

Figure 42 Sequence diagram for Resource identifier revocation

4.3.5 Data model

The only data that will be managed in the identification functionality are the IDs that will be stored in the database when they are generated and / or updated and the instruction to delete them in the case of revocation.

Since the three events aforementioned are very sporadic and they only involve a connection with the database, there is not data model for the identification function.

4.3.6 Baseline technology

Technology	Usage	Reference
Apache	Web server used to provide the registration form and service to the users.	https://www.apache.org/
PHP	Programming language used to implement the registration service.	https://secure.php.net/
MySQL	Database management service used in the cloud server during the registration.	https://www.mysql.com/
OpenJDK	Programming language used to implement the ID assignation method in the client side.	http://openjdk.java.net/

Table 7. Identification baseline technology

4.3.7 Test cases

In this section we define a set of test cases for the Identification functionality, based on the use cases previously described.

Test Conditions #1 for Use Case #1: Webpage and IDKey recovery

1. Check IDKey generation
2. Test interactions between Webapp and (mocked) Database

Test Conditions #2 for Use Case #2: Webservice registers the / IDKey recovery

1. Test that Webservice detects when string needs to be sanitized and that it can do it
2. Test that it can calculate IDKey from hashing the e-mail and the random string
3. Test that the field 'last idkey recovery' is updated with the current date-time, using the right precision
4. Check that it returns the IDKey to the device

Test Conditions #3 for Use Case #3: Agent calculates the resource ID

1. Check that agent, given an user IDKey and e-mail, can generate random string x to concatenate with the IDKey
2. Check that the inputs are given in the right way and the hash result of the

Test Conditions #4 for Use Case #4: Agent updates the resource ID

1. Test that a given resource ID is updated correctly

Test Conditions #5 for Use Case #5: Agent revokes a resource ID

1. Test that the right resource ID is revoked, and the changes scale, but not affect other independent services

4.4 Categorization

4.4.1 Requirements

The basic objective of the Resource Categorization module is to provide the information about the resources. When running this module, the system is able to know the Hardware (Storage, RAM, Processor etc.), Power, Software (Operating System), Security requirements (Data, Device, and Network), Attached components (Webcam, Printer, Scanner etc.), Attached IoT information (Sensors & Actuators), and also information about resource behaviour and features (i.e. - Mobility). Therefore, the resource categorization module has two objectives:

1. Acquiring Resource information from a device
2. Aggregating this information from the local database and pass it to the AGGR database

4.4.2 Use case diagram

Figure 43presents the use cases for the categorization module for both, a normal agent and a leader.

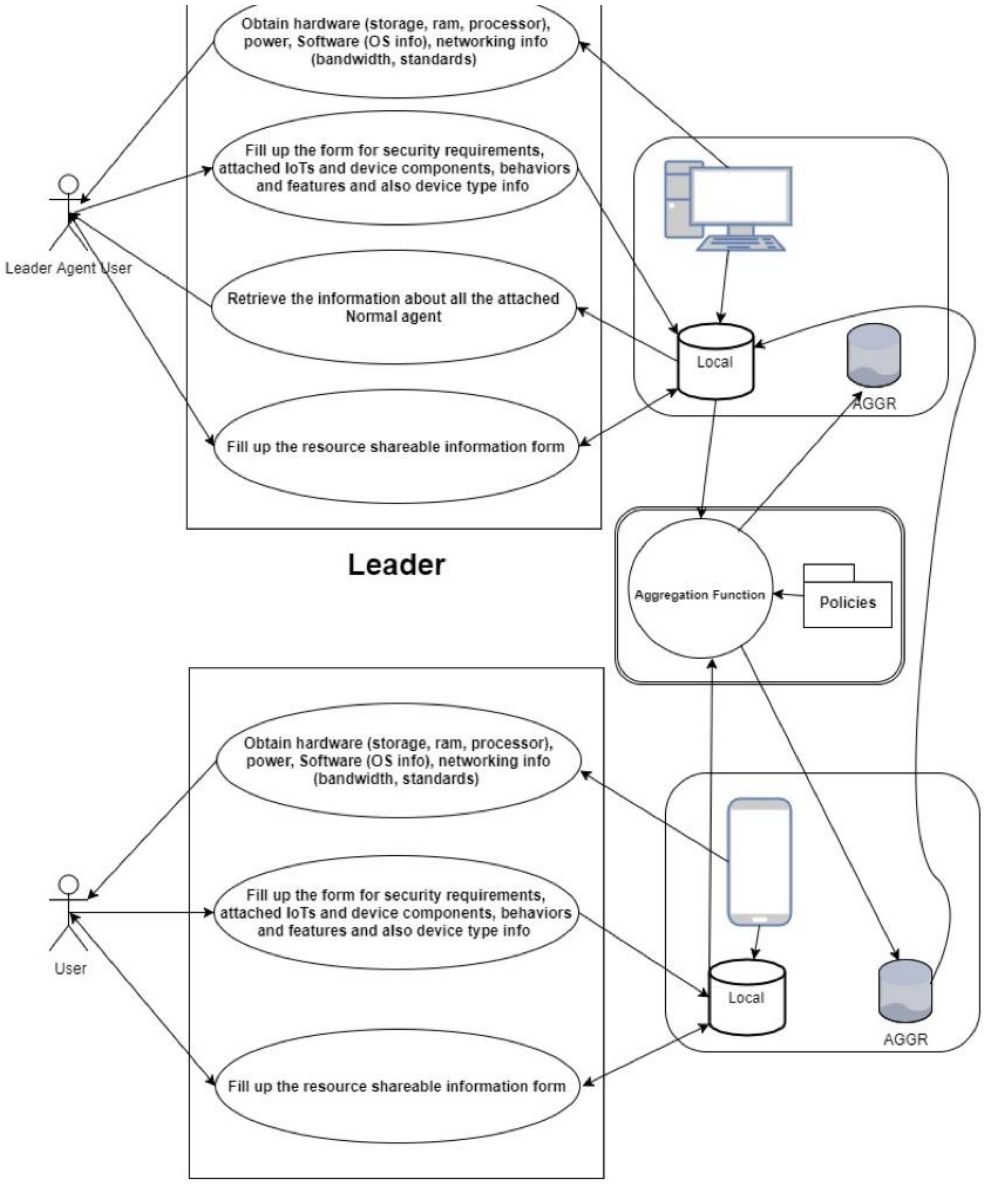

Figure 43 Use case diagram for the Categorization functionality

4.4.3 Component detailed architecture

Generally, the main functionality of the Resource Categorization module is: To acquire resource information from an Agent device.

In a posterior step this information is aggregated by the AC Data Management, which also stores the aggregation to optimize further accesses.

Before categorizing the mF2C resources, we need to know the basic attributes and characteristics of these resources. Therefore, for that purpose, we need to know the hardware specification, software specification, network specification, security requirements, attached components information,

attached sensors and actuators information, and features and behaviour information. To know this information in the mF2C system, for the IT-1, we consider a two-step procedure. In the first step after installing the agent software in a device, the resource categorization module will start working after the identification and discovery of the agent. In the first step after performing some python script the information about hardware (Total and available CPU, RAM, Power, Storage, information about System Architecture), Software (information about Operating System), Network (Standards & Communication technology and Bandwidth) will be acquired and stored in the Local database of the device. In the second step, the user of the device will fill a form with all the information about: Security requirements (Data, Device and Network security), Attached Components information (Webcam, Printer, and Scanner etc.), Attached IoTs information (Temperature Sensor, Humidity Sensor, Pressure sensor etc.), Features, and Behavior information (i.e. - Mobility). This information introduced by the user is also stored this in the Local Database of the device. The user of the Normal agent also fills the information about the amount of resources that the user wants to share with the mF2C system (according to the Sharing model detailed in section 5.3). All this information is stored in the local database of the device.

Aggregation of resource information is managed by the AC Data management module (section 4.6). The aggregation procedure will be different according to the different hierarchy of the system architecture. For an example, in the lower layer of the architecture

In a Normal Agent, aggregation functions only aggregate the information of its own resource information, attached IoTs information and also attached devices information (not being an agent).

In the Upper layer, Leader agent, aggregation functions not only aggregate its own resource information but also aggregate all the 'children' resource information.

In a similar way, in case of a Cloud agent, the aggregation function aggregates all the information of its attached Leader agents.

After this aggregation from data in the local database of the agent device, the aggregated information will be stored in the AGGR database and periodically synchronized with the local database of the upper layer agent.

Acquiring Static Resource Information from an Agent Device

The component will return the information about total quantities of Memory, Storage, Processor information including power information, OS name and version, information about system architecture. This information is stored in the local database of the agent.

Acquiring Dynamic Resource Information from an Agent Device

Here the component will return the current available resource specifications, such as – available memory size, available storage size, percentage of available processing capabilities, time to die-out or total percentage of power left etc. and then all this information will be stored in the local database.

Acquiring Network Resource Information from an Agent Device

Here the component will return the information about networking standards for a device and also will return the information about the current state of data rate or bandwidth of the device. Also this information will be stored in the local database.

Providing the attached IoTs and Components information an Agent Device

Here, the user will fill up some form, by providing the information of attached IoTs and attached device components (ex. Printer, webcam). This information will be also stored in the local database.

Providing the software & security requirements related information an Agent Device
User will fill up some form by providing the information about security requirements and also providing the information about applications software. This information will be also stored in the local database.

Providing the Features and Behaviors related information of an Agent Device
Users will fill up some form by providing the information about some features and behaviours (e.g. Mobile or not) of its agent device. After filling it up, this information will be stored to the local database of the device agent.
Retrieving all the attached agent information of a Leader agent (Only for leader or higher layer agent)
Here the component will return the resource information from all the attached agents for this leader agent. This information will be read from the local database of the leader agent.

4.4.4 Internal sequence diagrams

In case of the resource categorization in the mF2C platform, we have two sequence scenarios. The first scenario is – how the resource categorization module will work for a normal agent and in the second scenario is – how the categorization module will work in leader agent or upper layer agent. Below, we are going to present the two sequence diagram for the resource categorization module in mF2C.

Resource categorization in a normal agent
This diagram shows how the resource categorization module will work in a normal agent. By means of this sequence we describe the procedure to obtain the information of the resources component, behaviour and features and also the security requirements information. Take into account that steps 7,8 and 9 are managed by Sharing model module, and step 10 is managed by the AC Data Management module, however we have included them to show which will be the final information stored in the database, and how these modules interact.

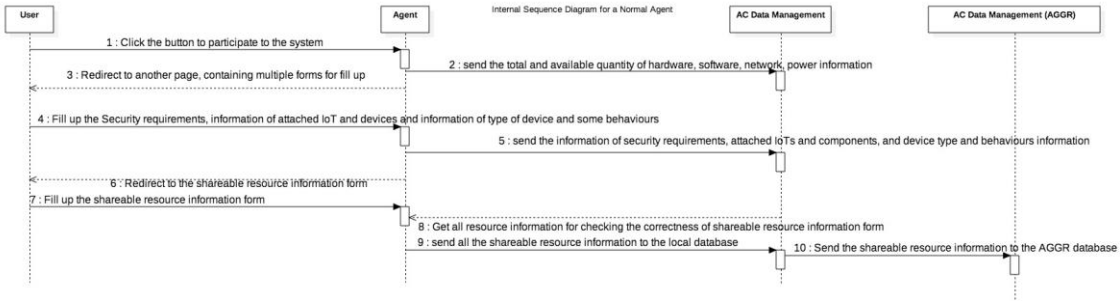

Figure 44 Sequence diagram for Resource categorization in a normal agent

This function is started by the Task Scheduling requesting the locations of an object through the *JavaClientManagementLib*. The request is forwarded to the *LogicModule,* which contacts the *MetadataManager* to get this information and return it. An object may be in several locations, if it has been previously replicated.

Resource categorization in a Leader
This diagram shows how the resource categorization module will work in a leader agent or upper layer agent. By means of this sequence we describe the procedure to obtain the information of resources component, behaviour and features and also the security requirements information for

own and also we describe how to get its children information. Take into account that steps 7,8 and 9 are managed by Sharing model module, and step 10 is managed by the AC Data Management module, however we have included them to show which will be the final information stored in the database, and how these modules interact.

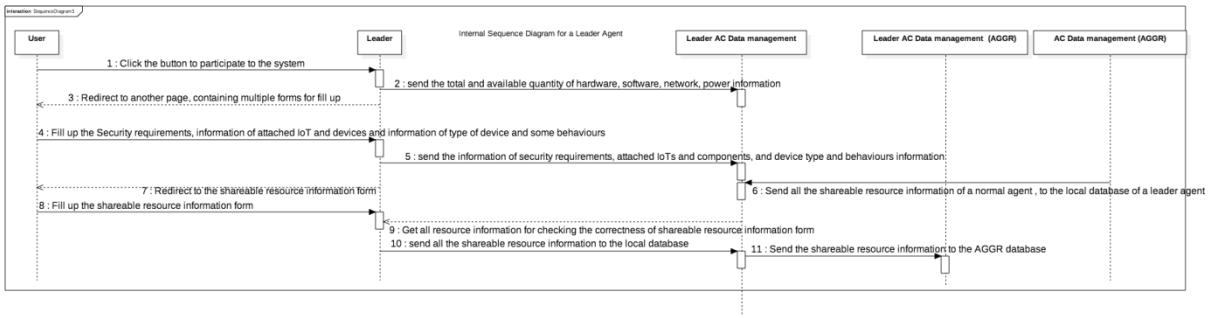

Figure 45 Sequence diagram for Resource categorization in a leader

4.4.5 Data model

mF2C resources can be represented as the composition of various resource components: hardware, software, features and behaviour, security related information, cost model and finally IoTs and attached device components. We can represent them by considering as a class of each component such as follows:

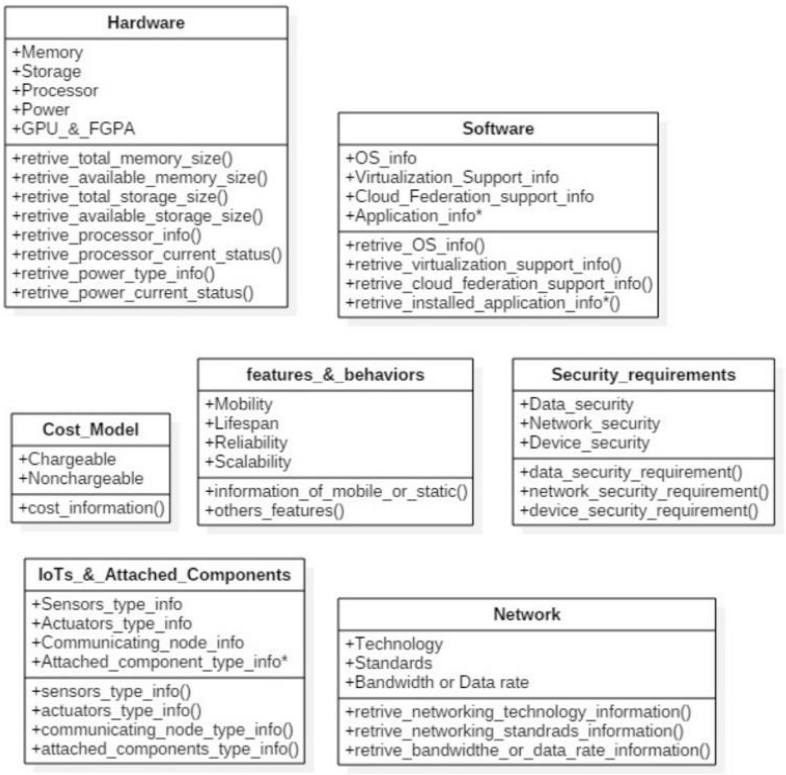

Figure 46 Class diagram for resource components

As shown in Figure 46, we initially classify the resources into various components such as – Hardware, Software, Network, Features & Behavior, Cost Model, Security requirements, IoTs and Attached

components. In the mF2C system, to represent the data model, we considered each of them as a Class.

From these initial classes we can derive the following classes:

- Static_resource_information
- Dynamic_resource_information
- Software_&_security_related_information
- Networking_resource_components_information
- Attached_IoTs_&_components_information
- Fetaures_&_behaviors_information)

These classes are included in the data model of the mF2C resource categorization, as shown in Figure 47.

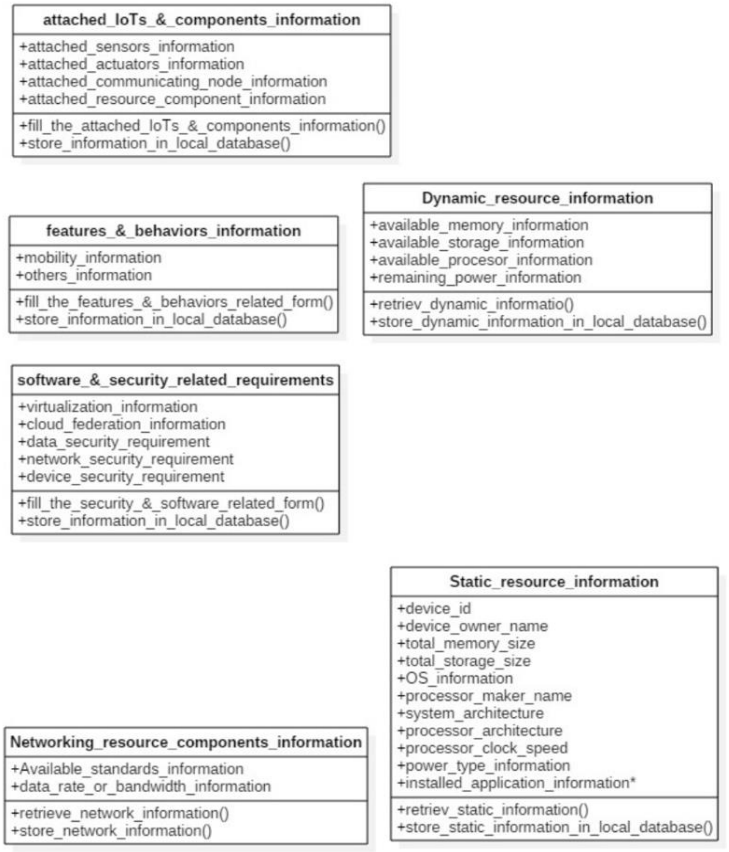

Figure 47 Data model for the Categorization functionality

Then, considering these six classes diagram we can derive the "Shareable_resource_information" (Figure 48), which is the most important class and basically this is the information to be shared or passed to the upper layer agent (leader).

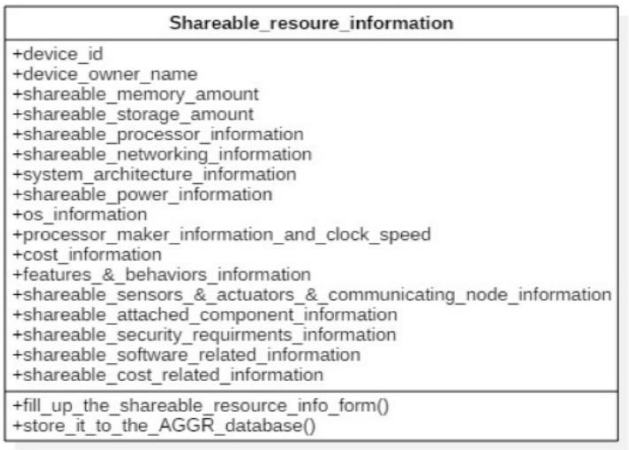

Figure 48 Shareable resource information

4.4.6 Baseline technology

Technology	Usage	Reference
Python3	Implementation language	https://www.python.org/downloads/
SQLite3	Considering Local Database Product	https://www.sqlite.org/

Table 8. Resource categorization baseline technology

4.4.7 Test cases

In this section we define a set of test cases for the Resource categorization functionality, based on the use cases previously described.

Test Condition #1 for Use Case #1: Leader Agent User obtains resources list

1. Test that the Leader Agent User can obtain the right hardware, software network and power resources

Test Condition #2 for Use Case #2: Leader Agent User fills up the security requirements form

1. Test that the Leader Agent User can fill up all the necessary security requirements form fields
2. Check that the input uses the right and same format than expected

Test Condition #3 for Use Case #3: Leader Agent User retrieves info about attached normal agents

1. Test that the Leader Agent User retrieves the all the fields with the necessary information about all the attached normal agents
2. Test that it includes all the attached normal agents
3. Test that the information has the right format (same than a mocked input)

Test Condition #4 for Use Case #4: Leader Agent User fills up the resource shareable form

1. Test that the Leader Agent User fills all the necessary shareable information form fields
2. Check that the input uses the right and same format than expected

Test Condition #5 for Use Case #5: User obtains resources

1. Test that the User can obtain the right hardware, software network and power resources

Test Condition #6 for Use Case #6: User fills up the security requirements form

1. Test that the User can fill up all the necessary security requirements form fields

2. Check that the input uses the right and same format than expected

Test Condition #7 for Use Case #7: User fills up the resource shareable form

1. Test that the User fills all the necessary shareable information form fields
2. Check that the input uses the right and same format than expected

Test Conditions #8 for Use Case #8: Resource categorization in a normal agent

1. Test, when the user accepts to participate to the system, that the agent sends the total available quantity of hardware, software, network and power information to the AC Data Management.
2. Test that the user is redirect to another page containing forms for fill up.
3. Test, when the user fill up the forms, that the agent send the information of security requirements, attached IoTS and components, device type and behaviours information to AC Data management.
4. Test that the user is redirect to the shareable resource information form.
5. Test, after the user fills up the form and the AC Data Management gives to the agent all information useful for the correctness of the shareable resources, that the agent sends all shareable resource information to the local database in AC Data Management.
6. Test that the AC Data Management sends the shareable resources information to the AGGR database.

Test Conditions #9 for Use Case #9: Resource categorization in a leader

1. Test, when the leader user accepts to participate to the system, that the leader sends the total available quantity of hardware, software, network and power information to the leader AC Data Management
2. Test that the user is redirected to another page containing multiple forms for fill up.
3. Test, when the user fills up the forms, that the leader sends the the information of security requirements, attached IoTS and components, device type and behaviours information to leader AC Data management
4. Test that the AC Data Management (AGGR) sends all shareable resource information of normal agent to leader AC Data management.
5. Test that the user is redirect to the shareable resource information form
6. Test, after the user fills up the form and the leader AC Data Management gives to the leader all information useful for the correctness for the shareable resources, that the leader sends all shareable resources information to leader AC Data management
7. Test that the leader AC Data management sends the shareable resources information to the AGGR database.

4.5 Monitoring

4.5.1 Requirements

The *Monitoring* module is responsible for instrumentation of each compute resource. A number of telemetry probes will capture performance metrics of the hardware/software that services are deployed onto. Each probe is required to perform 3 main functions:

Collect metrics:

Software probes must be able to capture metrics from hardware (both in-band and out-of-band), from any software source: host O/S, middleware, hosted application. Collectors should be able to query the output of other collectors as this could impact continued operation.

Process metrics:

Captured data should be passed through customised filters to perform a defined action on the data, eg, generate average, standard deviation, etc.

Publish metrics:

Processed data should be published to defined destinations, eg, file, database, message queue. The publish module should be able to analyse this data prior to publishing so as to decide how much data to publish, eg, publish averages, anomalies only.

4.5.2 Use case diagram

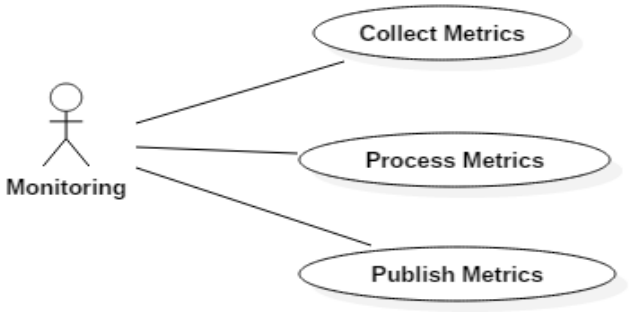

Figure 49 Use case diagram for the Monitoring functionality

Use case #1

Title: Collect Metrics

Actor: Admin

Scenario:

1. mF2C Admin will decide which metrics are required to be measured depending on service performance analysis requirements, SLA Management.
2. mF2C Admin will deploy relevant telemetry probes and initiate capture of metrics

Use case #2

Title: Process Metrics

Actor: Admin

Scenario:

Optional step. Original raw metrics may not be required so this step will generate aggregations, e.g., averages, prior to publishing

Use case #3

Title: Publish Metrics

Actor: Admin

Scenario:

Admin decides where the probes metrics should be published to and made available for analysis

4.5.3 Component detailed architecture

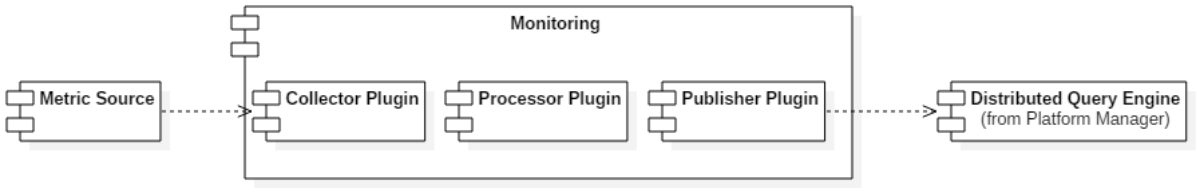

Figure 50 Component diagram for the Monitoring functionality

4.5.4 Internal sequence diagrams

Start Task

When a probe is started, the plugins which are referenced by the task manifest are subscribed to and a 'subscription group' is created. A 'subscription group' is a view that contains an ID, the requested metrics, plugins and configuration provided in the workflow.

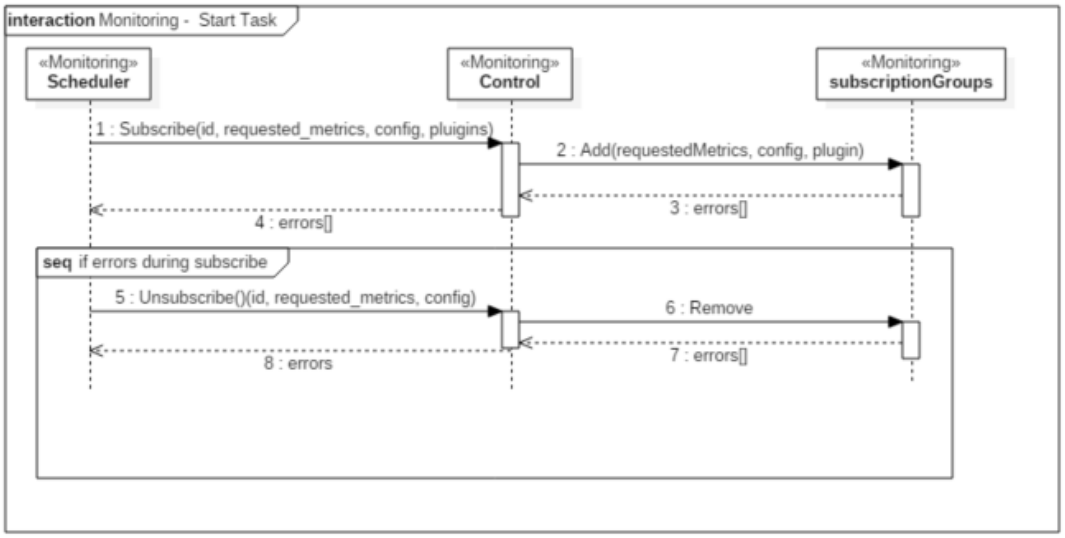

Figure 51 Sequence diagram for Start task

Load/unload Probes

When a probe is loaded/unloaded, this event triggers processing of all subscription groups. The requested metrics are evaluated and mapped to collector plugins. The required plugins are compared with the previous state of the subscription group triggering the appropriate subscribe/unsubscribe calls. Finally, the subscription group view is updated with the current plugin dependencies and the metrics collected based on the requested query.

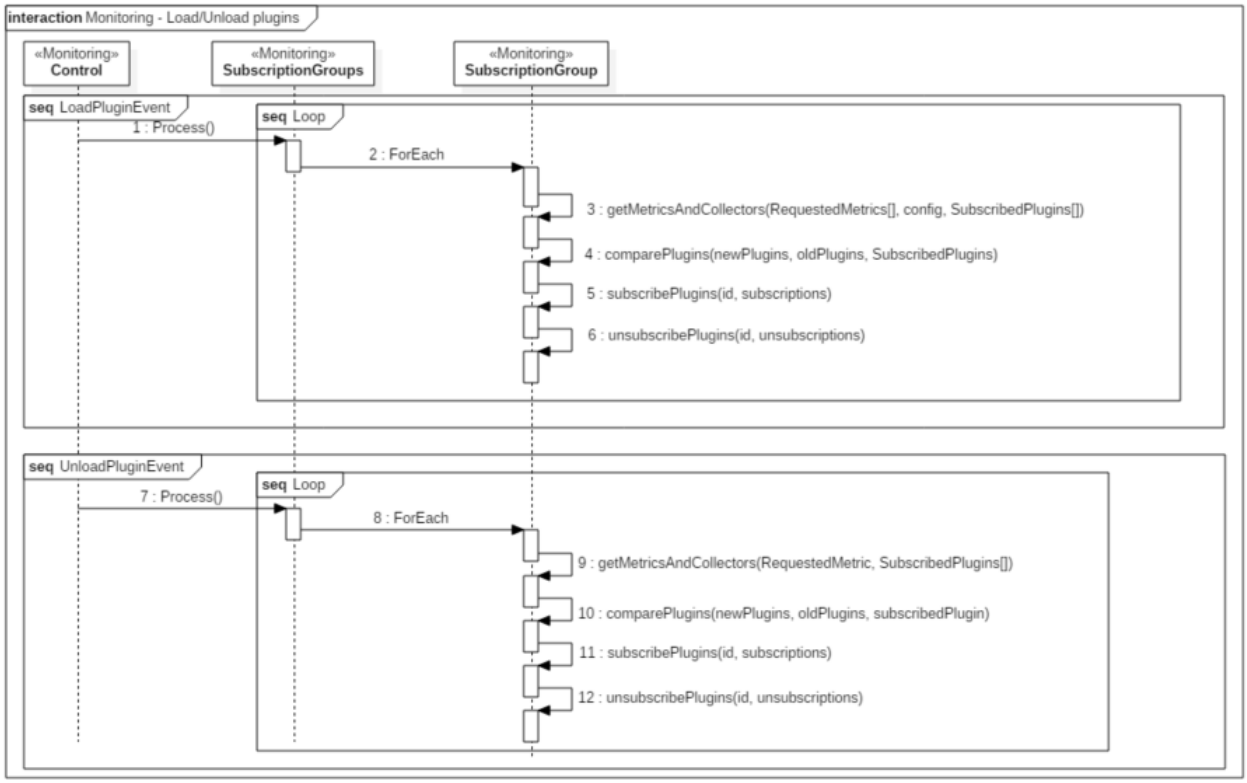

Figure 52 Sequence diagram for Loading/unloading a probe plugin

Collect Metrics

When a task starts and CollectMetrics is called, the Subscription Group is used to lookup up the plugins that will be called to ultimately perform the metric collection.

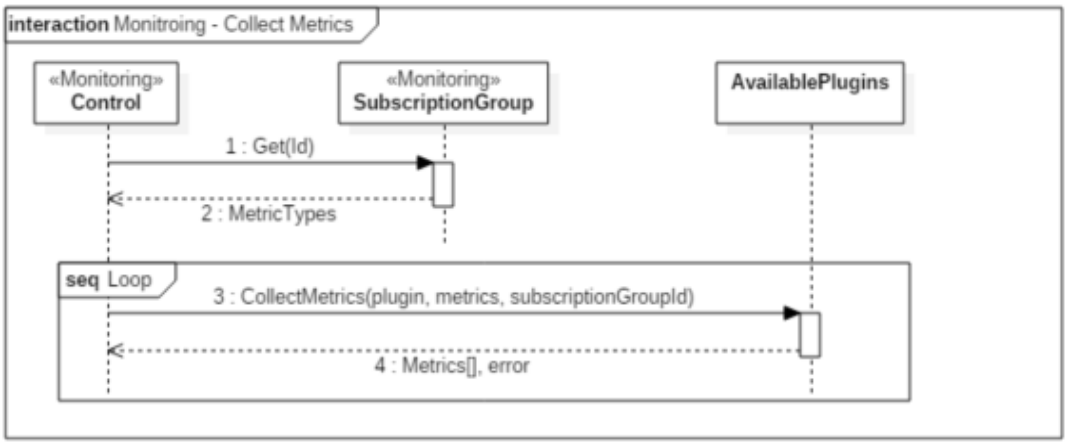

Figure 53 Sequence diagram for Collect metrics

4.5.5 Baseline technology

Technology	Usage	Reference
Go.lang	Implementation language	https://golang.org/

Table 9. Monitoring baseline technology

4.5.6 Test cases

In this section we define a set of test cases for the Monitoring functionality, based on the use cases previously described.

Test Condition #1 for Use Case #1: Admin collects metrics

1. Test that the Admin's monitoring component decides properly which metrics are required to be collected
2. Check that it to the right calls to collect metrics

Test Condition #1 for Use Case #2: Admin processes metrics

1. Test that the Admin's monitoring component processes input raw metrics in the way it is supposed to do, and produces the data/aggregations/etc. using the right format

Test Condition #1 for Use Case #3: Admin publishes metrics

1. Test that the Admin's monitoring component publishes the data taken from the metrics using the right format
2. Test that the Admin's monitoring component publishes the data taken from the metrics in the right place

4.6 Data Management

4.6.1 Requirements

The data management functionality of the AC is in charge of allowing applications or other functionalities to store, retrieve, and delete data in mF2C. This data can be either the information needed to manage the platform resources, services and users, or the data needed or generated by the services themselves.

For the data managed by the mF2C platform, this component is also in charge of managing the appropriate replicas of each piece of information, in such a way that data is accessible from the agent that will need it, which is not necessarily the one that created or updated it. This requirement is derived from the nature of the mF2C platform, where devices may lose connectivity, but they should be able to perform their functions even though they are isolated.

Also, this component must perform the requests coming from the Data Management component in the PM regarding the data locality requirements from the Task Scheduling component, and the deployment of classes so that objects can be accessed.

A non-functional requirement is that the data manager must support Python and Java, since these are the main programming languages that will be used in the development both of the mF2C platform and of the services supporting the use cases on top of the platform.

4.6.2 Use case diagram

The use case diagram in Figure 54 depicts the set of functionalities offered by the Data Management module in the AC. Some of them are used by other software components, which may be either applications or different functionalities in the agent, while another set of functionalities is offered specifically to the PM Data Management module in the same agent.

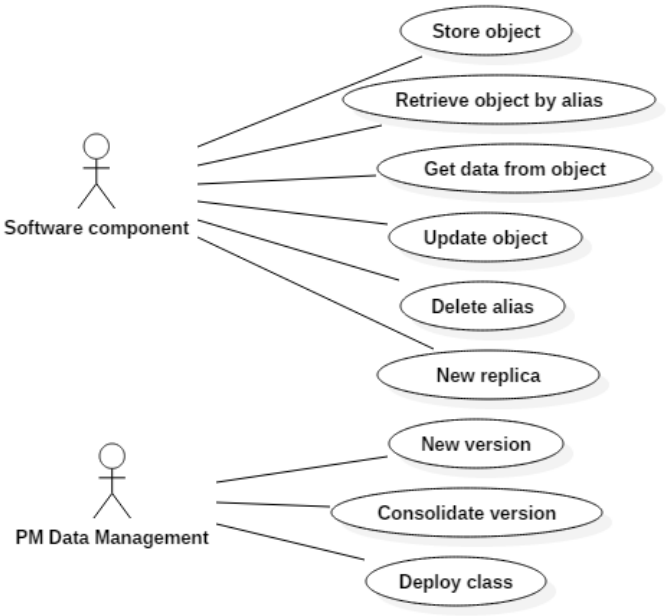

Figure 54 Use case diagram for the AC Data Management functionality

The Software component actor refers to any software component that can access the AC Data Management functionality in the context of mF2C. It can be another functionality defined in the Agent architecture, an application running on top of the platform, or a CIMI server that homogenizes interactions between components. The functionalities offered to this kind of actor are:

- Store object: persists an object and all its related objects in the Data Management, optionally with an alias that enables direct access to the object.
- Retrieve object by alias: obtains a reference to a stored object, given its alias
- Get data from object: once a reference to an object is obtained, this function allows reading the data contained in the object through the execution of a method that returns it. This method belongs to the registered class instantiated by the object.
- Update object: similar to the previous function, it allows to update the data contained in an object
- Replicate object: makes a replica of the object in the Data Management component of another agent, so it can be locally accessed avoiding communications.
- Synchronize replicas: this functionality is internally executed when an object is updated, and maintains consistency between the existing replicas, if any.
- Delete alias: removes the alias of an object so that, when it is not referenced by any other object, it will be deleted by the garbage collector.

There is another group of functionalities that are requested by the PM Data Management functionality, when a request is received from other mF2C components, as explained in D4.5 [4]. The AC Data Management is the responsible of effectively performing these actions on the backend.

4.6.3 Component detailed architecture

Figure 55 shows the component diagram for the AC Data Management.

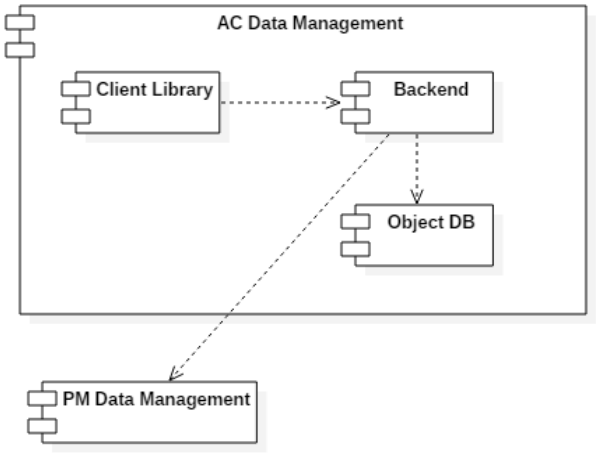

Figure 55 Component diagram for the AC Data Management functionality

The Client Library is the entry point to the Data Management functionality and provides a set of functions that allow executing the functionalities explained in the use case diagram.

The Backend is the core of the Data Management, and is responsible for managing all the objects that are stored and allowing to store and retrieve them. This component communicates with the PM Data Management to update the metadata of the objects (basically their locations), when changes occur. The Object DB is where objects are physically stored. In the next subsections each of these components is explained in detail.

Client Library class diagram

Figure 56 depicts the class diagram for the Client Library in the AC Data Management.

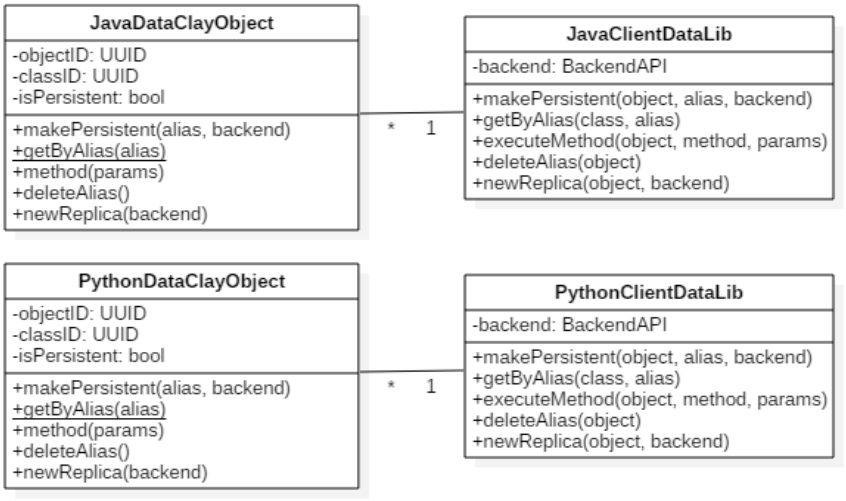

Figure 56 Class diagram for the Client Library sub-component

The JavaDataClayObject and the PythonDataClayObject represent the stubs that are used by the programmer, which are the API accessed by the software component that wants to perform CRUD operations on the objects. These classes also include the necessary information about the object they instantiate, so that it can be communicated to the respective ClientDataLib. All these classes are implemented in both languages because they deal with the objects of the programming language, not yet generic objects managed by the Data Management.

The ClientDataLib has a reference to the Backend component, accessible through the BackendAPI. When objects are handled to the Backend they are transformed into a language independent

representation and are managed in the same way, except for some specific functions, as will be explained in the next subsection.

Backend class diagram

The class diagram for the Backend component is shown in Figure 57.

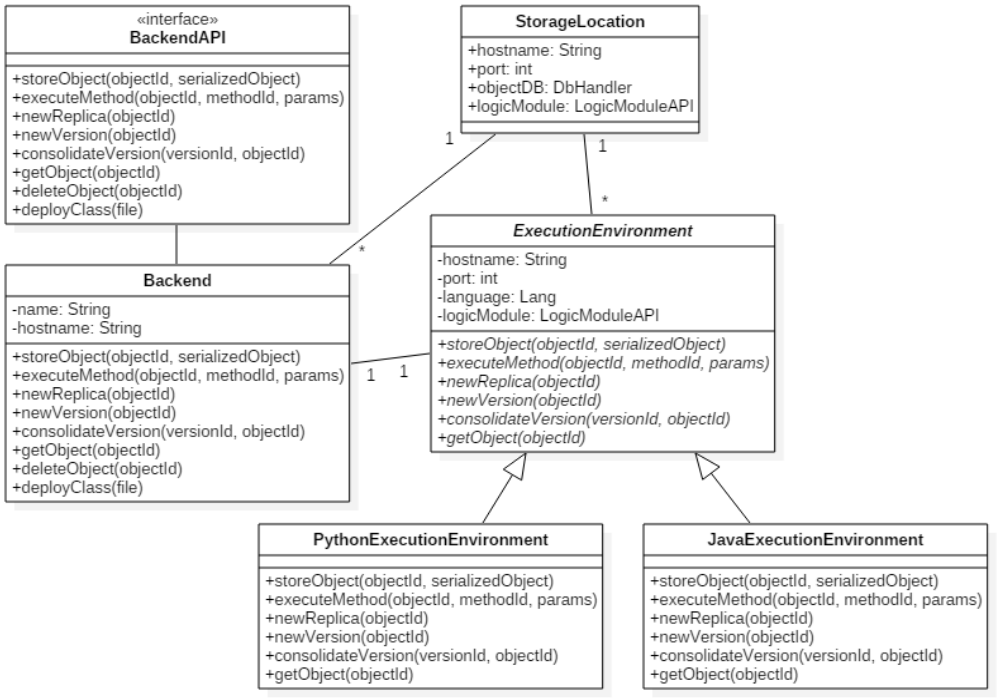

Figure 57 Class diagram for the Backend sub-component

The BackendAPI is implemented by the Backend concept, which represents a couple StorageLocation - ExecutionEnvironment. The StorageLocation represents a location where objects can be stored, and the ExecutionEnvironment is in charge of performing operations on these stored objects. Each ExecutionEnvironment has an associated StorageLocation, which may contain objects of different languages, while a StorageLocation has several ExecutionEnvironments, one for each language, that execute the methods of the registered classes on the stored objects. Thus, the abstract class ExecutionEnvironment has a different implementation for each language, each of them running either the Java Virtual Machine or the Python interpreter to enable the execution of the methods on the stored objects according to the language in which they were created. The StorageLocation contacts the ObjectDB through the reference to the DbHandler interface, explained in the next subsection.

The ExecutionEnvironment has a reference to the LogicModule (a component of the PM Data Managment explained in D4.5 [4]) to register the changes in the object metadata resulting from the execution of the methods that modify the location of the objects, i.e. newReplica, newVersion and consolidateVersion.

The StorageLocation contacts the ObjectDB to store and retrieve data through the reference to the DbHandler interface, explained in the next subsection.

ObjectDB class diagram

The class diagram in Figure 58 shows the internals of the ObjectDB component, responsible for managing the interaction with the database management system that stores the objects.

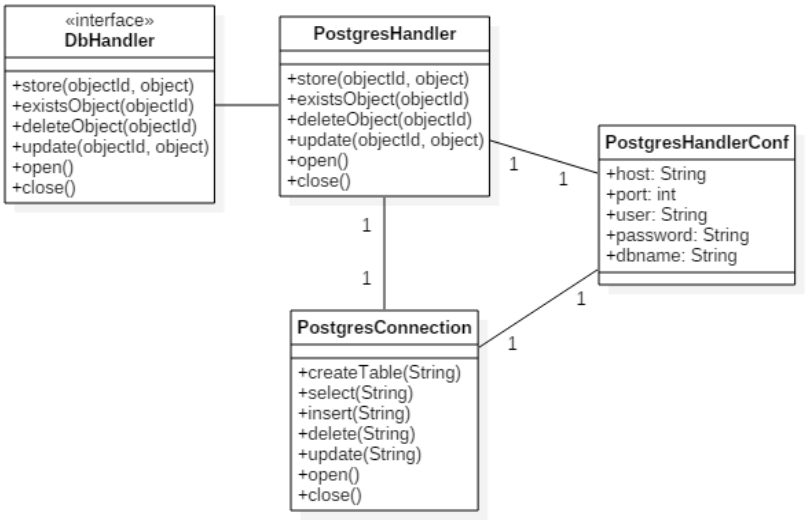

Figure 58 Class diagram for the ObjectDB sub-component

The point of entry is the DbHandler interface, which includes methods to create, read, update and delete (CRUD) objects, as well as methods to open and close a connection to the database. This interface can be implemented by different handlers that allow performing these functions on different kinds of database. In this case, we have chosen a relational database, in particular PostgreSQL, accessible through the PostgresHandler. This class translates the CRUD methods into SQL statements, and also handles the connection to the PostgreSQL database. All these actions are delegated to the PostgresConnection, which access the database. The PostgresHandlerConf contains the necessary configuration parameters to connect to the database management system.

4.6.4 Internal sequence diagrams

In all diagrams, the interactions from the DbHandler to the PostgresHandler have been omitted, since the PostgresHandler is just an implementation of the DbHandler interface. In this way, the diagram remains independent of the underlying database management system used.

For the sake of clarity, the diagrams show the success scenario where the objects in the ExecutionEnvironment are found in the cache, and those steps where the object is retrieved from the database by contacting the DbHandler are omitted. In contrast, the whole path of interactions is shown for the case of storing an object in the database, despite this being done asynchronously when the cache is full.

Finally, for language-dependent classes (e.g. JavaClientDataLib), the diagrams show only the Java case. For Python, the diagrams would be analogous, using the equivalent Python-related classes specified in the class diagrams (e.g. PythonClientDataLib), which have exactly the same methods.

Store object

The following diagram shows the interactions in the Store object function.

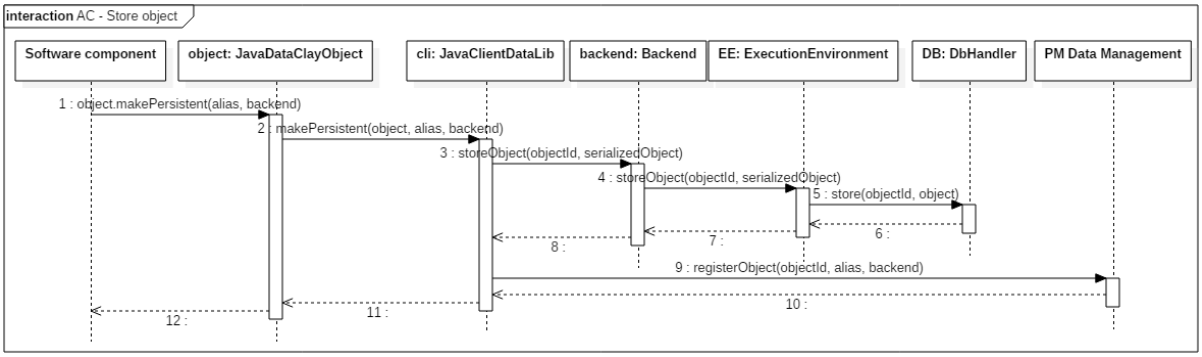

Figure 59 Sequence diagram for Store object

A software component starts the execution of this functionality by making a makePersistent call on an object, instance of a stub, represented by the class JavaDataClayObject. This stub internally calls the JavaClientDataLib, which serializes the object and contacts the Backend to store the object. This function is performed by the ExecutionEnvironment, which caches the object and also stores it in the database.

Retrieve object by alias

Figure 60 shows the sequence diagram for the Retrieve object by alias function. This functionality provides the software component with a reference to the object (its objectId), so it can be manipulated (read or modified, as will be seen in the next diagram) by executing its methods in the ExecutionEnvironment. In this way, data stays where it is stored instead of travelling between components. However, from the point of view of the requesting Software component, the object belongs to its address space.

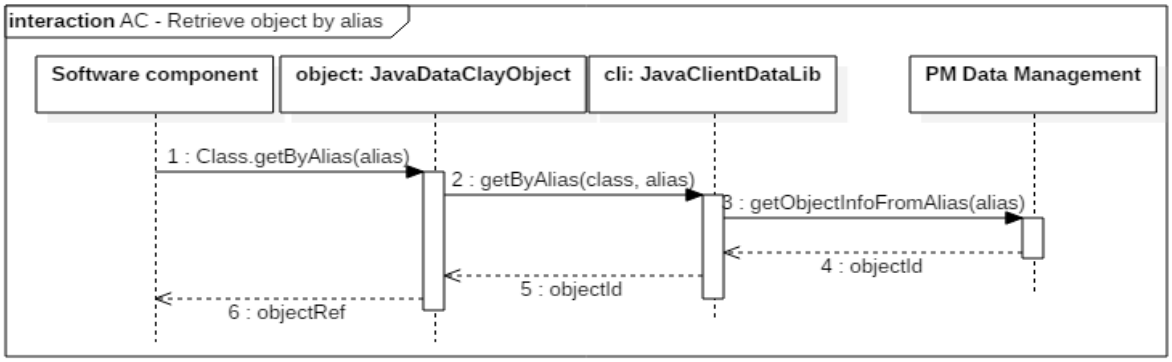

Figure 60 Sequence diagram for Retrieve object by alias

From the Software component, the class method getByAlias is called. This provides the JavaDataClayObject with the information needed to identify the object to be retrieved: the class where the object belongs, and its alias (identifier within the class). The JavaClientDataLib needs to contact the PM Data Management functionality (in particular the MetadataManager), since it is the component that has information about all the metadata.

The objectId is returned and incorporated in the JavaDataClayObject, which builds a stub instance of (reference to) the requested object for the Software component.

Get data from object / Update object

The functionalities Get data from object and Update object have been specified in the use case diagram for the sake of clarity. However, what they do internally is to execute a method on the specified object. This method belongs to a class previously registered and deployed by means of the New model functionality in the PM Data Management. Figure 61 shows the sequence diagram for executing a method on an object, either to read it or to update it.

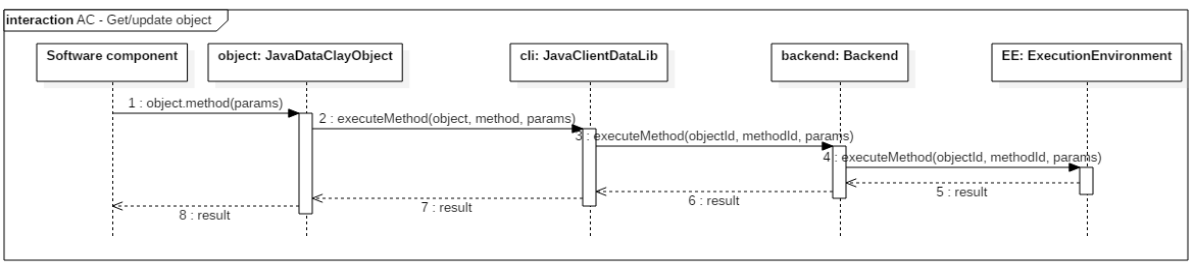

Figure 61 Sequence diagram for Get data from object and Update object

When the Software component calls a method on an object, the JavaDataClayObject provides the object on which the method is called, the method and the parameters of the call to the JavaClientDataLib. This library contacts the Backend component, which makes the call to its associated ExecutionEnviroment, where the objects are instantiated in a Java Virtual Machine that executes the methods on the objects.

It is worth to mention that the replica management is done as part of the registered classes. This means that methods in the registered classes can include a set of instructions, automatically added by means of decorators, which allow managing the synchronization between replicas. These instructions are a combination of functionalities that are already offered by the Data Management, namely Get locations and Update object.

Delete alias

Figure 62 shows the sequence diagram for the Delete alias function.

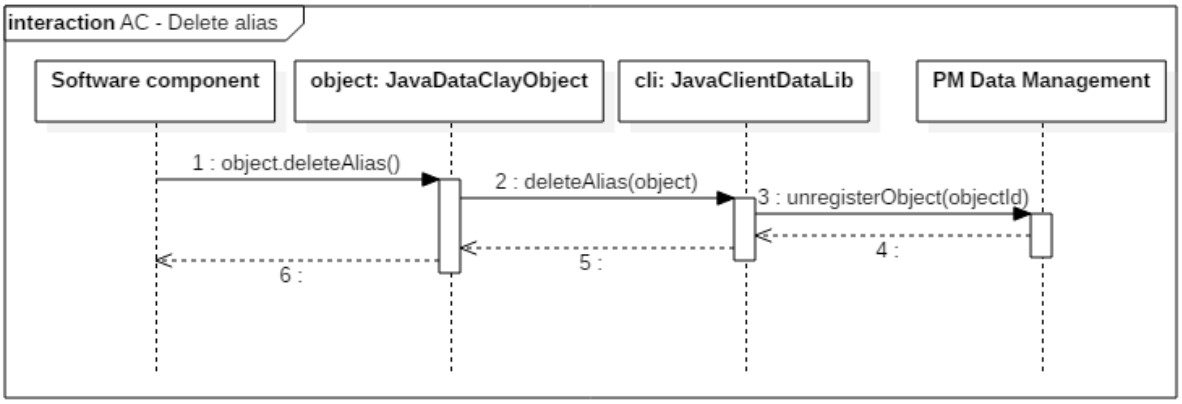

Figure 62 Sequence diagram for Delete alias

This functionality deletes the alias of an object, so that the garbage collector can remove it when it is not referenced by any other object. The call is made from a software component by means of the JavaDataClayObject, which will contact the JavaCientDataLib as usual. This functionality modifies the metadata of the object, which is managed by the Data Management component in the PM. Thus, the call is forwarded to the PM Data Management and will be handled by the MetadataManager class in the LogicModule, explained in D4.5 [4].

New replica

The diagram in Figure 63 shows the sequence of interactions for New replica, which makes a copy of an object in the AC Data Management component of another agent.

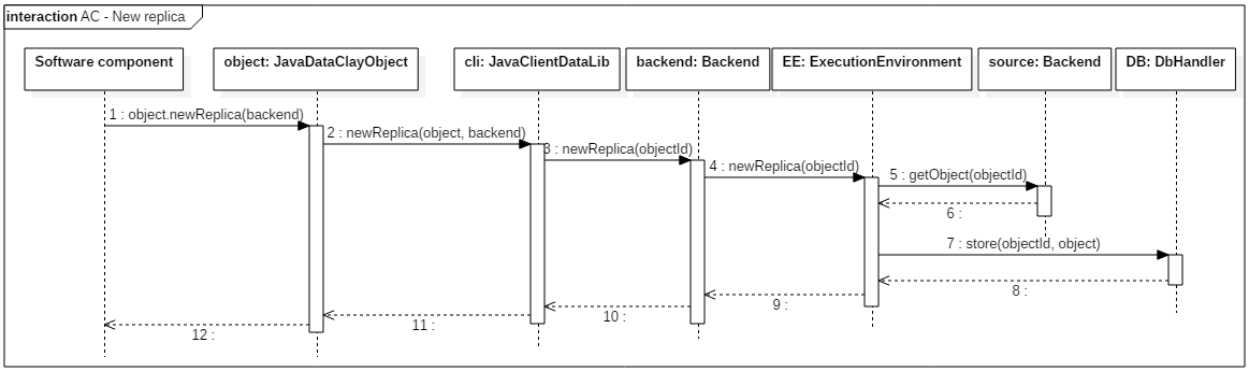

Figure 63 Sequence diagram for New replica

Once the call from the JavaDataClayObject is forwarded to the client library, the Backend is contacted to perform the necessary operations in its associated ExecutionEnvironment. This class

gets the object from the source (external) backend, caches it in the ExecutionEnvironment and, asynchronously, stores it in the associated object database through its DbHandler interface.

New version

The sequence diagram in Figure 71 shows the interactions for the New version function in the AC Data Manager.

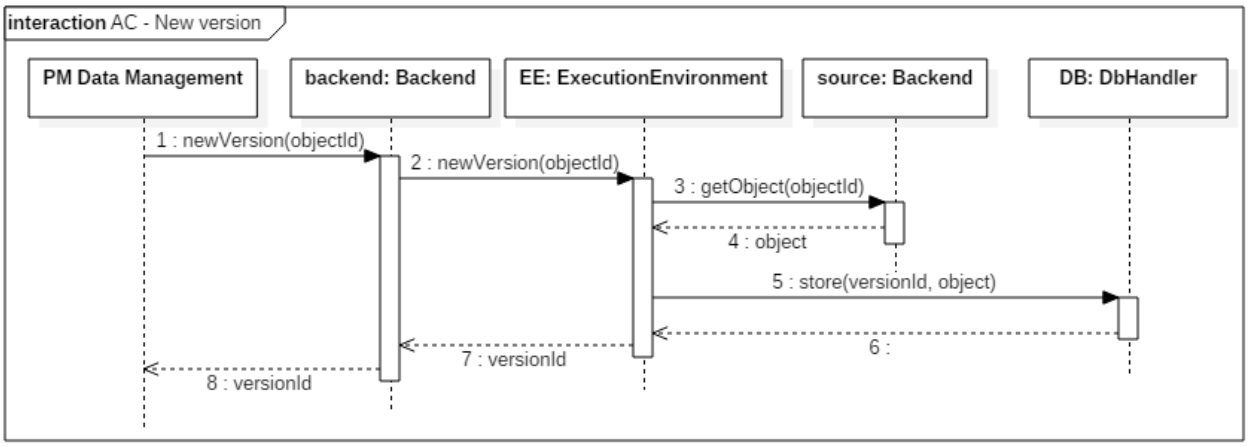

Figure 64 Sequence diagram for New version

The call to New version comes from the PM Data Management component, since this function is only used by the Distributed Execution Runtime of the PM. Similar to New replica, once the Backend is entered, the ExecutionEnvironment is responsible for getting the original object from its location and then store it in the local object database, but this time with a new versionId.

Consolidate version

The sequence diagram in Figure 75 shows the interactions for Consolidate version in the AC Data Manager. As happens with New replica, this functionality is required only by the Distributed Execution Runtime in the PM.

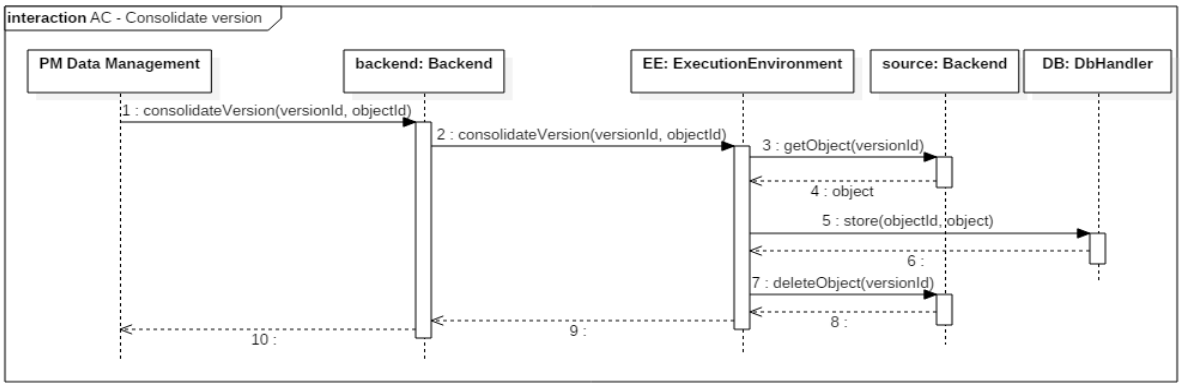

Figure 65 Sequence diagram for Consolidate version

The sequence of interactions once the Backend is reached is analogous to the previous functionalities, and orchestrated by the ExecutionEnvironment. In this case, however, the object that is got from the source backend is a version performed by the Distributed Execution Runtime, and its data is stored in the local database with its original objectId. Afterwards, the version is deleted.

Deploy class

The diagram in Figure 76 describes the interactions for deploying a class in a backend.

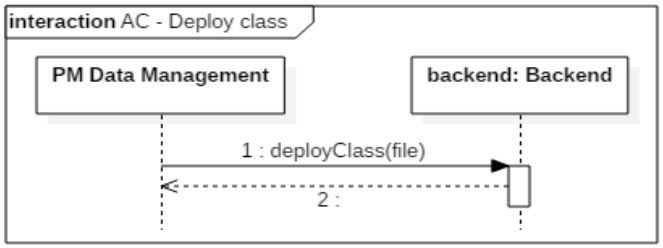

Figure 66 Sequence diagram for Deploy class

When a request comes from the Data Management in the PM with the Java or Python executable files to be deployed, the backend stores them in its local disk so that they are accessible by the Java or Python runtimes.

4.6.5 Data model

The Data Management functionality is the one used by other components to manage the information about mF2C. More precisely, the data models of other functionalities correspond to the classes that are going to be registered in the Data Management, and the data stored corresponds to the generic "object" instances that appear in the sequence diagrams. Thus, a specific class diagram about the kind of mF2C information managed by the AC Data Management is not applicable.

4.6.6 Baseline technology

Technology	Usage	Reference
Java	Implementation language	http://www.oracle.com/us/technologies/java
Python	Implementation language	https://python.org/
gRPC	Intercommunication RPC library	https://grpc.io/
Postgresql driver	DBMS to store metadata	https://jdbc.postgresql.org

Table 10. Data Management baseline technology

4.6.7 Test cases

In this section we define a set of test cases for the Data Management functionality, based on the use cases previously described.

Test Condition #1 for Use Case #1: Software Component store object

1. Test that the call to store the object is done
2. Test that the object has the right format and the sent and stored value are consistent

Test Condition #2 for Use Case #2: Software Component retrieve object by alias

1. Test that the right object is retrieved when it is requested an object by alias

Test Condition #3 for Use Case #3: Software Component get data from object

1. Test that the object provides the right data in the right format

Test Condition #4 for Use Case #4: Software Component update object

1. Test that the object update is done in the proper way, keeping consistency

Test Condition #5 for Use Case #5: Software Component delete alias

1. Test that the check that an alias is deleted and the status of the system is consistent afterwards

Test Condition #6 for Use Case #6: Software Component new replica

1. Test that the a new replica is created with the right format

Test Condition #7 for Use Case #7: PM Data Manager new version

1. Test that a new version is registered properly, keeping consistent status

Test Condition #8 for Use Case #8: PM Data Manager consolidates version

1. Test that a version is consolidated properly, keeping consistent status

Test Condition #9 for Use Case #9: PM Data Manager deploy class

1. Test that a class is deployed properly

5. User Management

This section describes the User Management module and all of its components. On one hand, this module is responsible for managing the user's profile and the definition of the user's device resources that will be shared in mF2C. On the other hand, this module is also responsible for checking that the mF2C applications act according to these sharing model properties and profile defined by the user.

This module is composed of three components that will be described in the next subsections:

- Profiling
- Sharing Model
- User Management Assessment[1]

5.1 Profiling

The profiling component is responsible for managing the information about the user's profile. This profile's information includes the user's identification key, the user's email, and how the user joins mF2C: as a contributor, as a consumer, or both.

5.1.1 Requirements

This component addresses the following requirements:

- The profiling component needs to be able to register a new user and create his profile in the device.
- This component needs to be able to edit or delete this profile when requested.
- Finally, the profiling component should provide an API to expose the methods and services needed by other mF2C components.
- In this first iteration, we consider that there will be only one user per device.

5.1.2 Use case diagram

The Profiling functionality defines four uses cases that are depicted in the next figure:

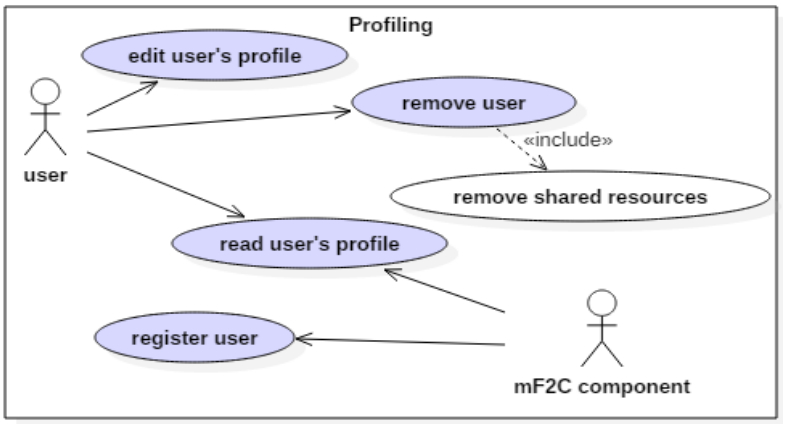

Figure 67 Use case diagram for the Profiling functionality

Use Case #1

Title: Register user

Actor: mF2C agent

Scenario:

[1] This module, previously known as "SLA / QoS Enforcement", was renamed to avoid misunderstandings with other components and functionalities.

1. During the initialization phase, the mF2C agent initializes the user registration process.

2. The Profiling module connects to the CIMI server to store the information.

Use Case #2
Title: Remove user

Actor: User

Scenario:

1. The user uses the GUI / interface to initialize the process for unsubscribing from mF2C.

2. The Profiling module connects to CIMI in order to remove the data related to the profiling and sharing model.

Use Case #3
Title: Edit user's profile

Actor: User

Scenario:

1. The user uses the GUI / interface to initialize the edition of the profiling data.

2. The Profiling module updates the content of the correspondent rows through CIMI.

Use Case #4
Title: Read user's profile

Actor: User, mF2C component

Scenario:

1. The Profiling module exposes through a REST API all the methods needed to perform the related CRUD operations.

5.1.3 Component detailed architecture

Next figure shows the different elements that compose this component and also those that interact with it.

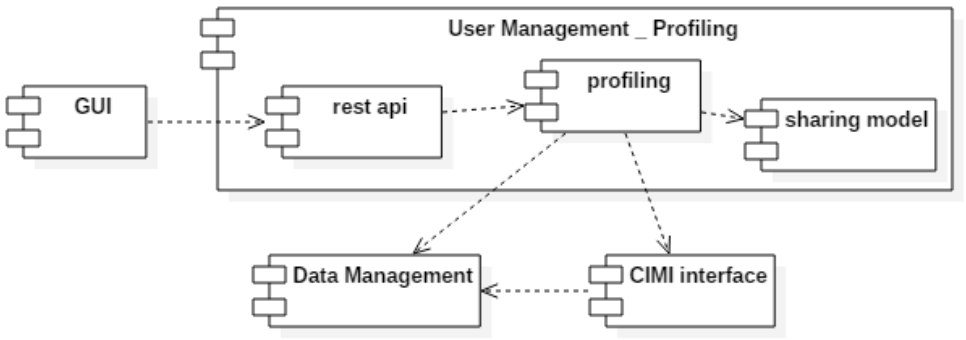

Figure 68 Component diagram for the Profiling functionality

The profiling component connects to CIMI (or the Data Management module, if CIMI is not available) in order to perform all the CRUD operations related to the user's profile. This component is instantiated by the User Management REST API sub component in order to expose all the methods needed by other mF2C components.

Profiling class diagram
Next picture depicts the class diagram of this component:

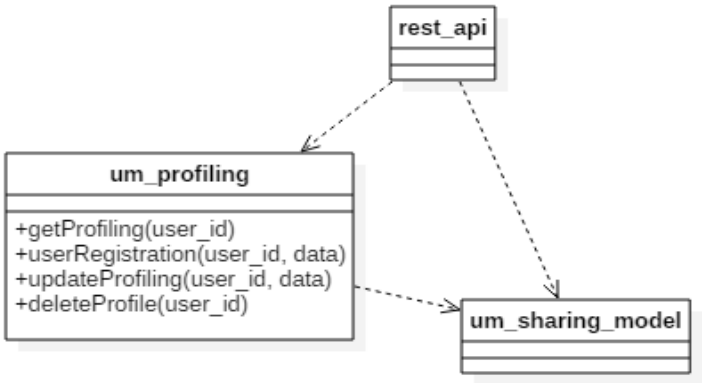

Figure 69 Class diagram for Profiling

While the *rest_api* class handles all the requests, the *um_profiling* class has the entire logic correspondent to this component.

5.1.4 Internal sequence diagrams

Register User

The rest_api component receives the "register user" request and redirects it to the profiling module. This module is the responsible for calling the CIMI server in order to create the new profile:

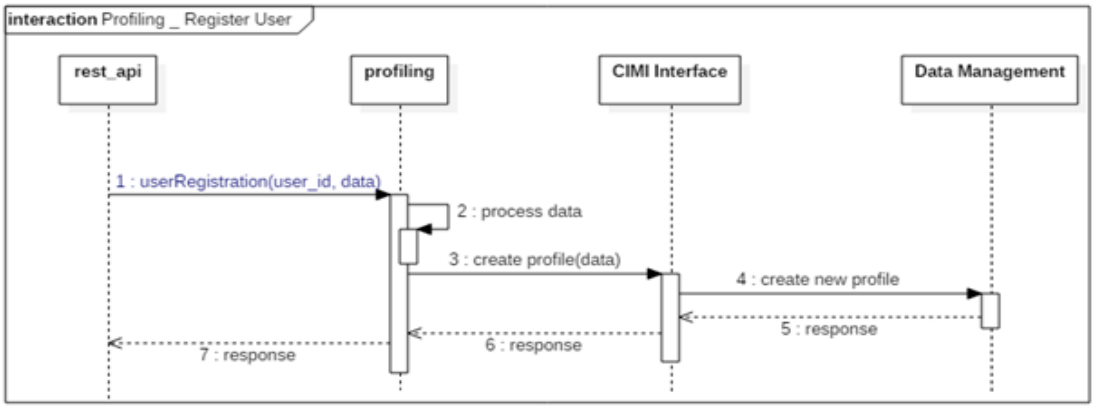

Figure 70 Sequence diagram for Register user

Remove User

First, the profiling module resets / removes the values corresponding to the user's sharing model. Then, it calls the CIMI server in order to remove the profile.

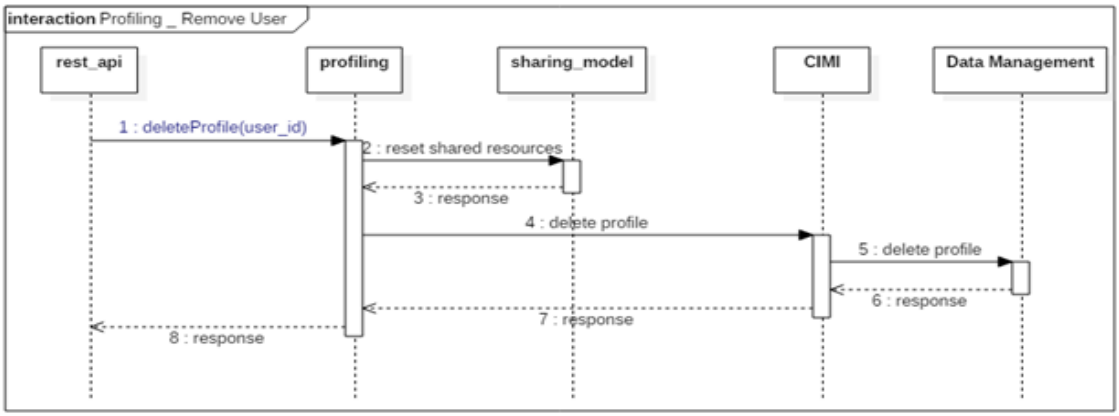

Figure 71 Sequence diagram for Remove user

Edit User's Profile

The rest_api component receives the "edit user" request with all the new profile values, and then redirects it to the profiling module. Finally, this module is responsible for calling the CIMI server in order to update the profile:

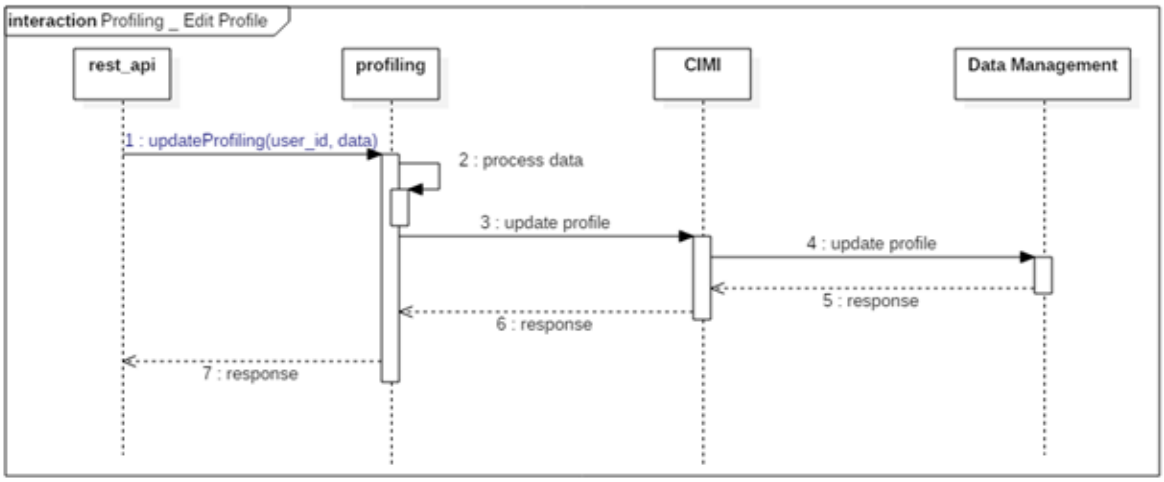

Figure 72 Sequence diagram for Edit user's profile

Read User's Profile

The profiling module is responsible for getting the user's profile.

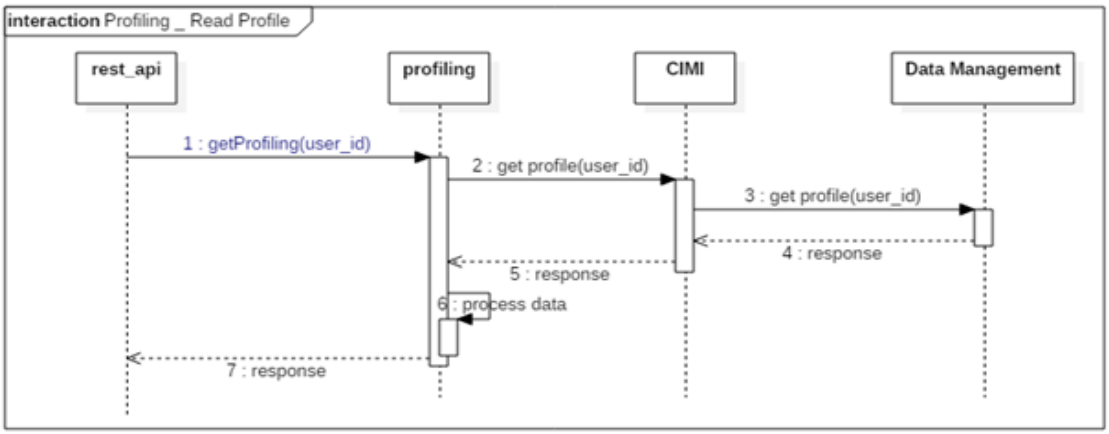

Figure 73 Sequence diagram for Read user's profile

5.1.5 Data model

Next diagram depicts the information managed by this component:

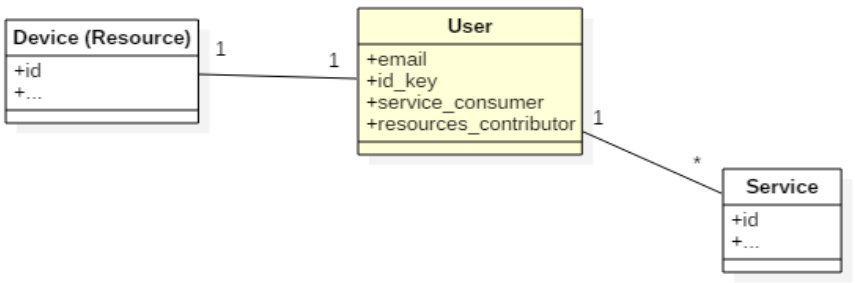

Figure 74 Data model for the Profiling functionality

Basically, this component will manage the information related to the device's user.

5.1.6 Baseline technology

Technology	Usage	Reference
Python	Implementation language	https://www.python.org/

Table 11. Profiling baseline technology

5.1.7 Test cases

In this section we define a set of test cases for the Profiling functionality, based on the use cases previously described.

Test Condition #1 for Use Case Profiling#1: agent registers user

1. Test that this is executed only during initialization phase
2. Test that the profiling module send the data in the right format to the right system

Test Condition #2 for Use Case Profiling#2: user removes himself

1. Test that the user can start the process to remove himself
2. Test that the change is applied in cascade and that the system stays in a consistent state after the user is removed
3. Test that the user is disconnected of the system after its removal, and that cannot use any of the system services that request to be registered

Test Condition #3 for Use Case Profiling#3: user edits user's profile

1. Test that the user can start the process to edit his data
2. Test the user data is updated in cascade if necessary

Test Condition #4 for Use Case Profiling#4: user or component reads user's profile

1. Test that the profile module exposes the right API and that this guarantees the CRUD operations
2. Test that the user or agent can read an user profile using the selected API

5.2 User Management Assessment

The User Management Assessment component is responsible for checking that the mF2C applications "respect" the sharing model and the profile properties defined by the device's user.

5.2.1 Requirements

The User Management Assessment component has the following requirements:

- It should detect if there are one or more applications using more resources than defined by the user, and if they are running according to the user's profile.
- It needs to be able to send warnings to the Platform Manager Service Orchestration module if it detects some violation (application using more resources than "allowed").
- The profiling component should provide an API to expose the methods and services needed by other mF2C components, including those to start and stop the assessment process.

5.2.2 Use case diagram

Next picture depicts the use cases associated to the User Management Assessment component:

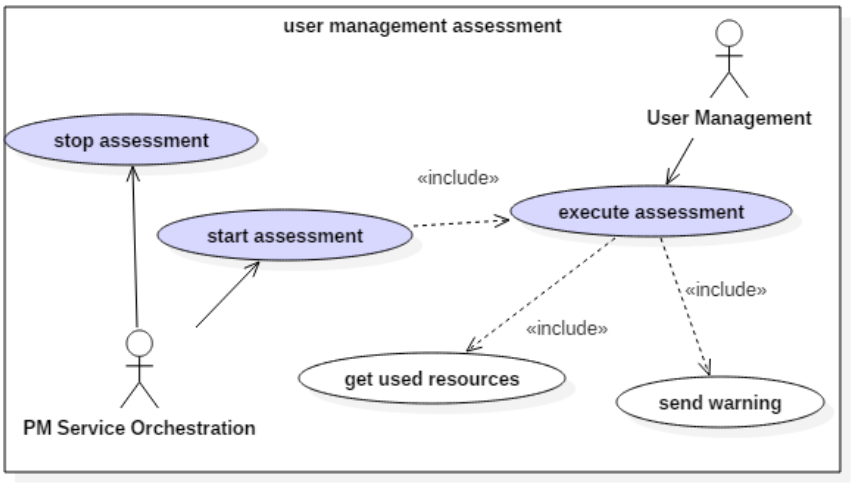

Figure 75 Use case diagram for the User Management Assessment functionality

Use Case #1

Title: Start (restart) Assessment

Actor: SLA Management

Scenario:

1. The Platform Manager Service Orchestration component uses the REST API to call the User Management Assessment module in order to start / restart the assessment process.
2. The Assessment module starts the execution of the assessment process.

Use Case #2

Title: Stop Assessment

Actor: SLA Management

Scenario:

1. The Service Orchestration calls the User Management Assessment module to stop the assessment process.
2. The Assessment module stops the execution of the assessment process.

Use Case #3

Title: Execute Assessment

Actor: User Management

Scenario:

1. It gets the profile and the list of shared resources defined by the user.
2. It gets the resources that are being used by the mF2C applications.
3. It checks if the resources used by these applications correspond to the resources defined by the user.
4. If these applications use more than defined, then it sends a warning to the Service Orchestration module.

5.2.3 Component detailed architecture

Next figure shows the different elements that compose this component.

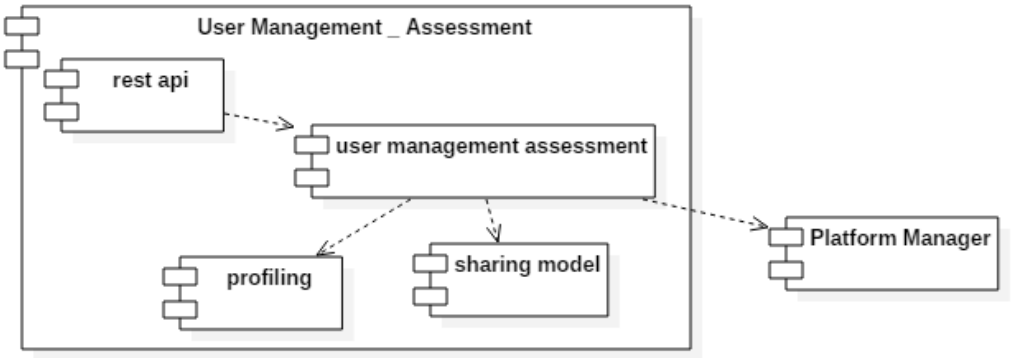

Figure 76 Component diagram for the User Management Assessment functionality

The Assessment module exposes, via the REST API component, all the services needed to manage its processes. The entire logic of this module is contained in the "user management assessment" sub component. It relies on the others User Management components, "profiling" and "sharing model", and also on the (Platform Manager) Service Orchestration in order to send the warnings.

User Management Assessment class diagram

Next picture depicts the class diagram of this component:

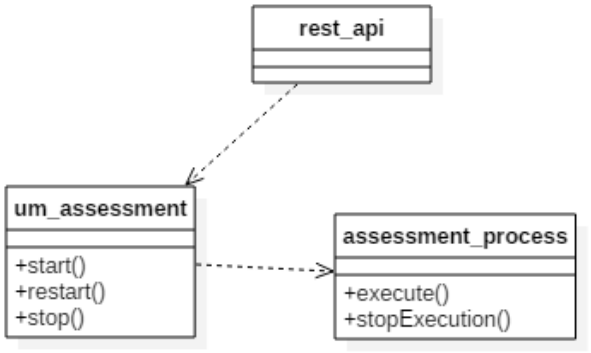

Figure 77 Class diagram for the User Management Assessment functionality

On one hand, the *rest_api* class handles all the requests. And on the other hand logic of this component is divided between the *um_assessment* and the *assessment_process* classes.

5.2.4 Internal sequence diagrams

Start Assessment

The *rest_api* component receives the "start assessment" request and redirects it to the *u_m_assessment* module, which is the responsible for starting and running the assessment process.

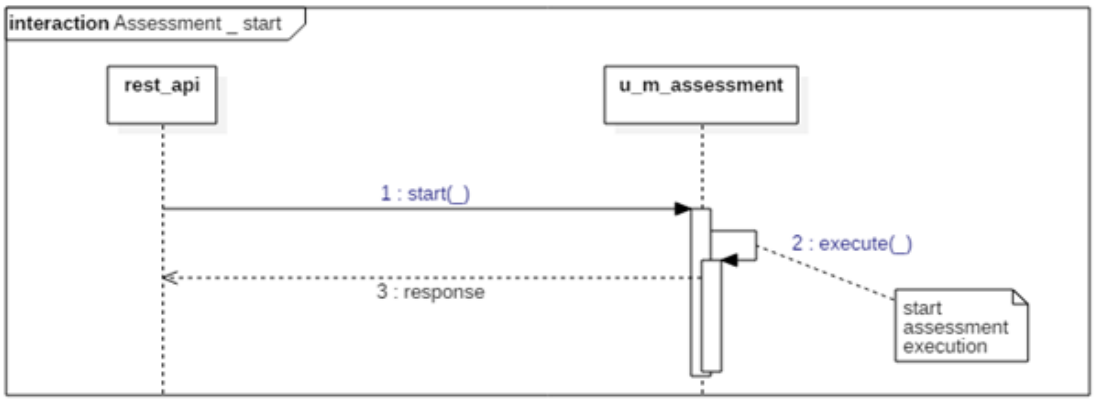

Figure 78 Sequence diagram for Start Assessment

Stop Assessment

The *rest_api* component receives the "stop assessment" request and redirects it to the *u_m_assessment* module:

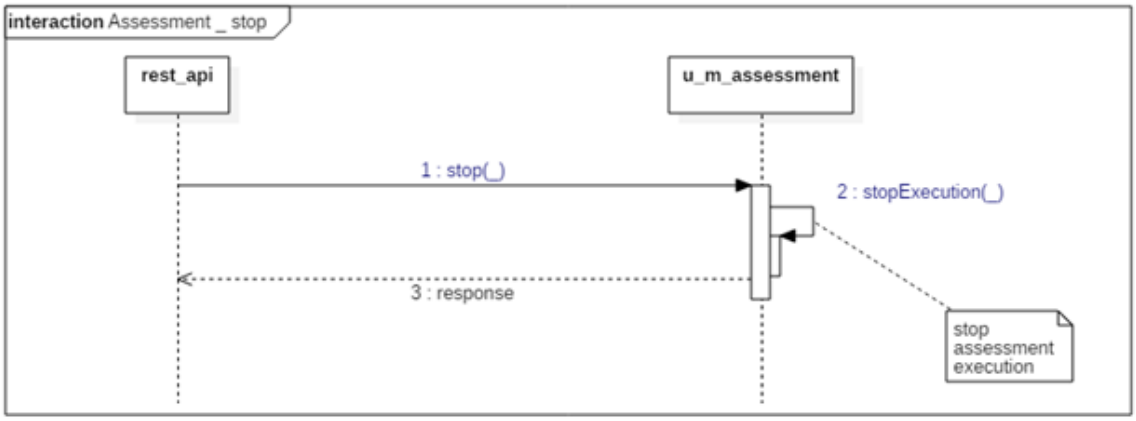

Figure 79 Sequence diagram for Stop assessment

Execute Assessment

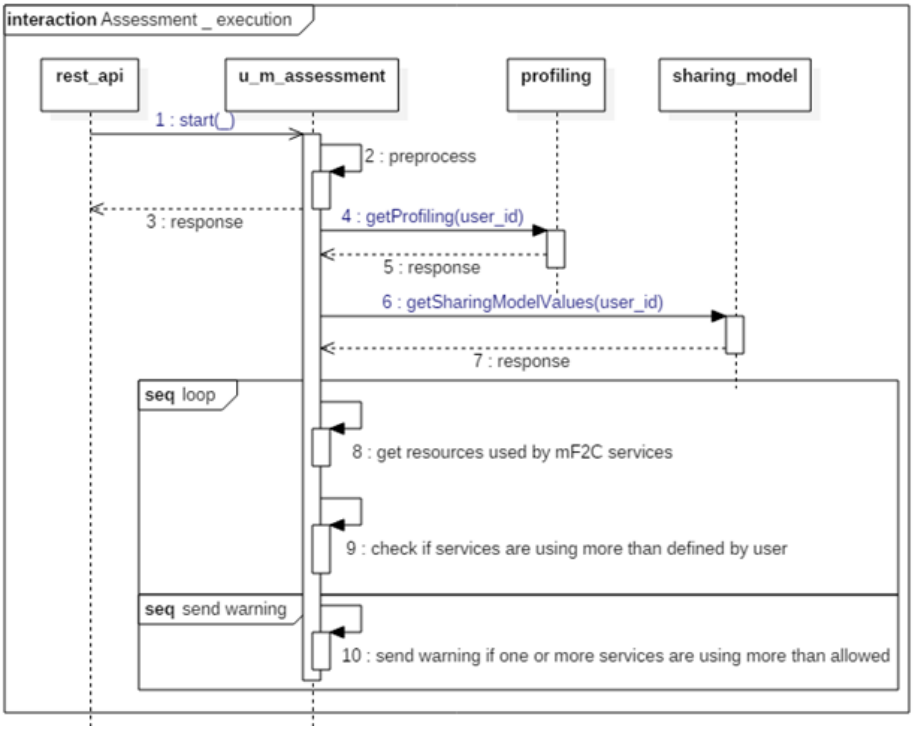

Figure 80 Sequence diagram for Execute assessment

In principle, the *u_m_assessment* component will interact with the Landscaper component (Platform Manager) in order to get the resources used by the services in a concrete device. Then, if it detects that services are not acting as defined by user (sharing model and profiling), it will send warnings to the Lifecycle.

5.2.5 Data model

This component will use the information managed by the Profiling and Sharing Model components.

5.2.6 Baseline technology

Technology	Usage	Reference
Python (including FLASK and FLASK_RESTFUL libraries / micro frameworks)	Implementation language	http://www.python.org

Table 12. User Management Assessment baseline technology

5.2.7 Test cases

In this section we define a set of test cases for the Assessment functionality, based on the use cases previously described.

Test Condition #1 for Use Case #1: PM SLA Management starts/restarts Assessment

1. Test that the PM SLA Management uses the same API than the UM Assessment module exposes
2. Test that the Assessment module start the process to answer to the API call (start)

Test Condition #2 for Use Case #2: PM SLA Management stops Assessment

1. Test that the Service Orchestration calls the same API than the UM Assessment module exposes
2. Test that the Assessment module start the process to answer to the API call (stop)

Test Condition #3 for Use Case #3: User Management executes Assessment

1. Test that the execution steps are called in the right order
2. (1st) Test that the profile and list of shared resources by the user are returned using the right format
3. (2nd) Test that the list resources that are being used by mF2C applications (mocked) are returned
4. (3rd) Test that the comparison between resources used by an application (mocked) are the same defined by the user
5. (4th) Test that the applications don't use more than defined
6. (4th) Test that if applications use more than defined, Service Orchestration module is warned

5.3 Sharing Model

The Sharing Model component is responsible for defining the resources that the device's user wants to share to mF2C.

5.3.1 Requirements

The Sharing Model component addresses the following requirements:

- First, it needs to be able to create information about the device's resources that the user wants to share in mF2C.
- Then, this component needs to be able to edit or delete this information.
- Finally, the Sharing Model component should provide a REST API to expose all the methods needed by other components.

5.3.2 Use case diagram

Next picture depicts the use cases associated to the Sharing Model component:

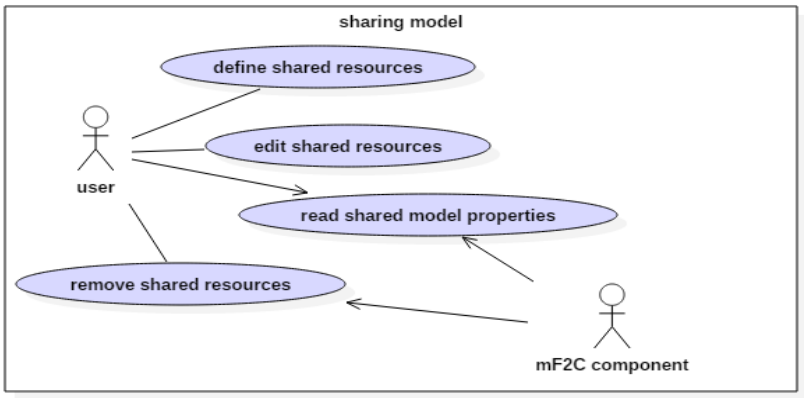

Figure 81 Use case diagram for the Sharing Model functionality

This component implements four use cases: define shared resources, edit shared resources, read shared model properties, and finally, remove shared resources. Next tables describe each of these use cases.

Use Case #1

Title: Define Shared Resources

Actor: User

Scenario:

1. The user uses the GUI / interface to define the resources he wants to share to mF2C
2. The Sharing Model component creates the correspondent rows through CIMI.

Use Case #2

Title: Remove Shared Resources

Actor: User, mF2C component

Scenario:

1. The user initializes, through the GUI, the process for removing the shared resources. Also another mF2C component can initialize this process by calling the corresponding services of the REST API.
2. The Sharing Model component resets the correspondent rows through CIMI server.

Use Case #3

Title: Edit Shared Resources

Actor: User

Scenario:

1. The user initializes the process for editing or updating the resources he wants to share to mF2C
2. The Sharing Model component updates the correspondent rows through CIMI.

Use Case #4

Title: Read Shared Resources

Actor: User, mF2C component

Scenario:

1. The Sharing Model module exposes through a REST API all the methods needed to perform the related CRUD operations.
2. It gets the requested values through the CIMI server.

5.3.3 Component detailed architecture

Next figure shows the different elements that compose this component.

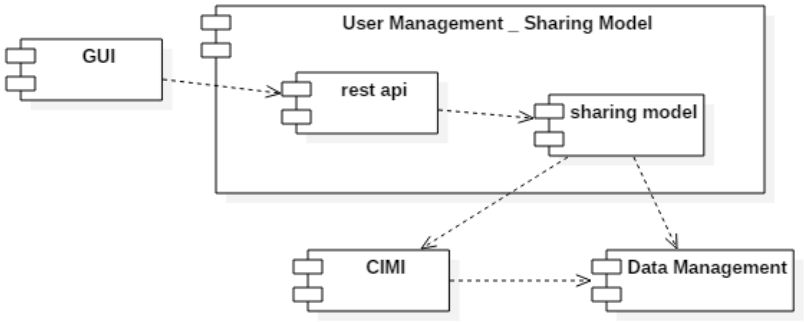

Figure 82 Component diagram for the Sharing Model functionality

In the same way as the Profiling component, the Sharing Model also connects to CIMI / Data Management to perform all the CRUD operations related to the device's resources shared to mF2C. This component is also instantiated by the User Management common REST API in order to expose the methods needed by other components.

Sharing Model class diagram

Next picture depicts the class diagram of this component:

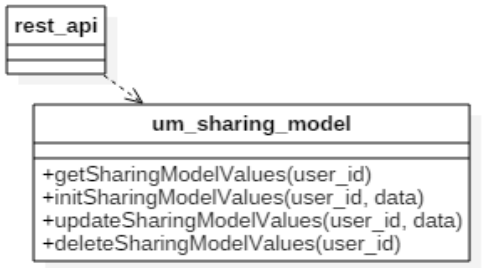

Figure 83 Class diagram for the Sharing Model functionality

5.3.4 Internal sequence diagrams

Define Shared Resources

The *rest_api* component receives the “define shared resources” request with all the user’s sharing model values, and redirects it to the *sharing_model* module. This module is responsible for first, processing these values, and then, for calling the CIMI server in order to save them:

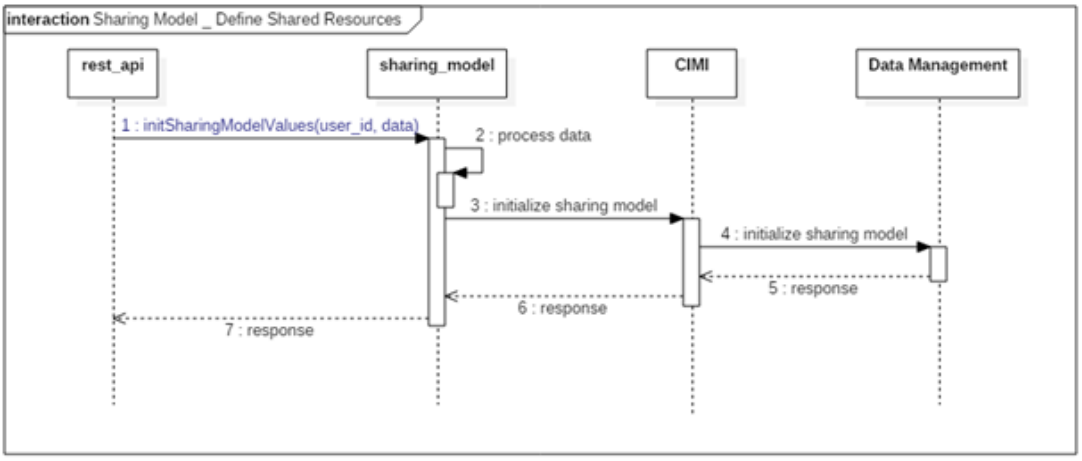

Figure 84 Sequence diagram for Define shared resources

Remove Shared Resources

The *sharing_model* module is responsible for deleting or resetting the information about the shared resources.

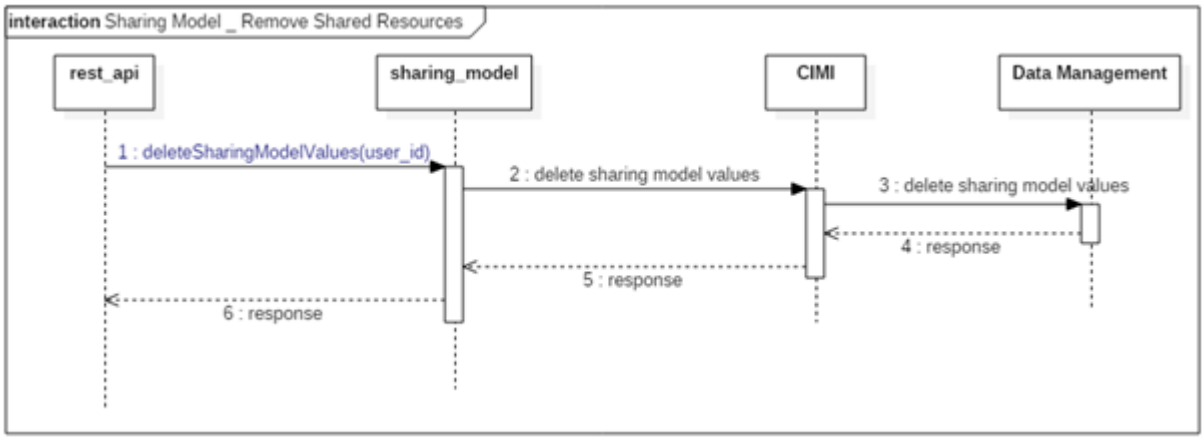

Figure 85 Sequence diagram for Remove shared resources

Edit Shared Resources

The *rest_api* component receives the "edit shared resources" request with the new sharing model values, and redirects it to the *sharing_model* module.

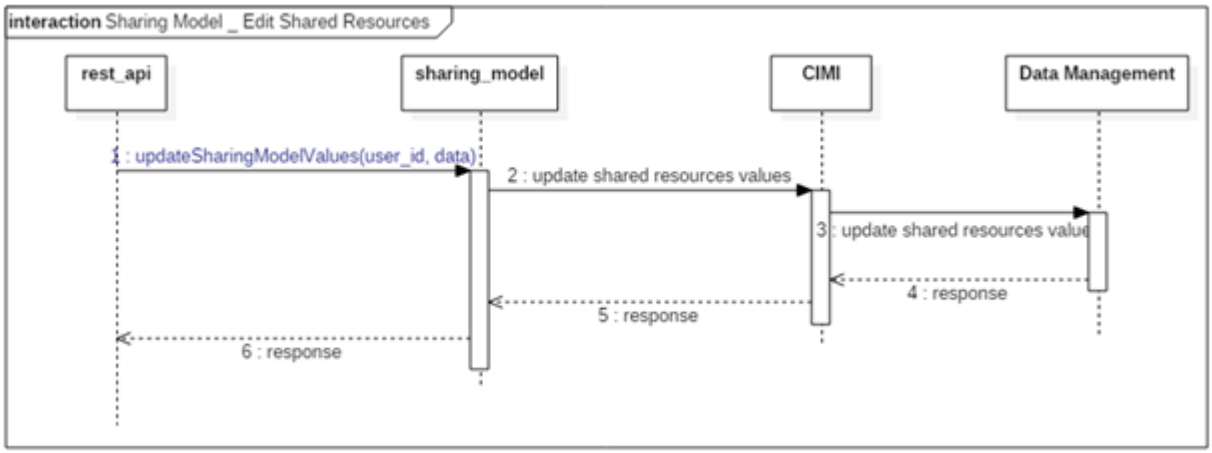

Figure 86 Sequence diagram for Edit shared resources

Read Shared Resources

The *sharing_model* module is responsible for getting the user's sharing model by calling the CIMI server.

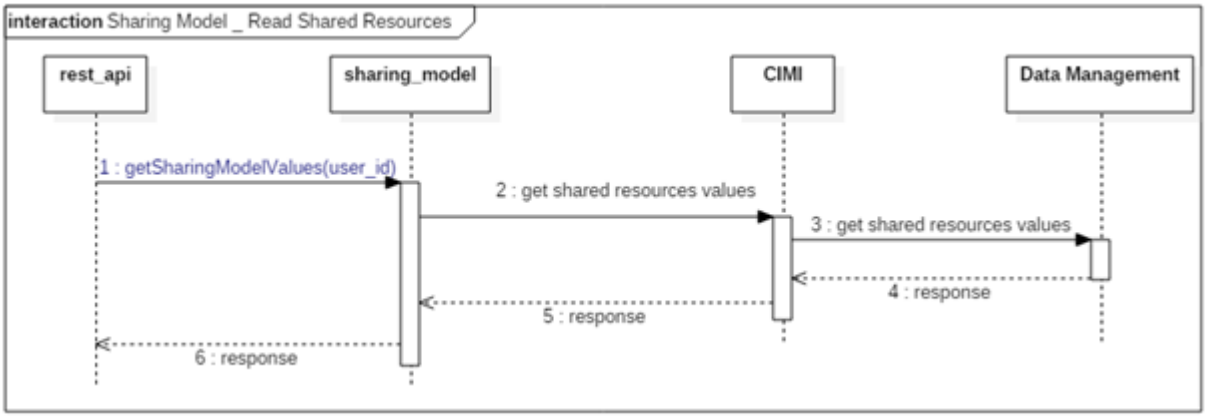

Figure 87 Sequence diagram for Read shared resources

5.3.5 Data model

Next diagram depicts the information managed by this component:

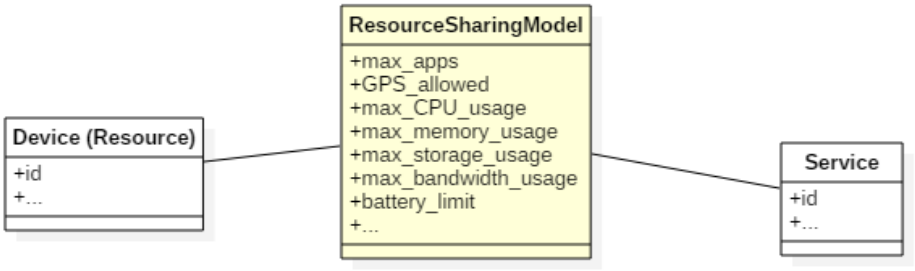

Figure 88 Data model for the Sharing Model functionality

5.3.6 Baseline technology

Technology	Usage	Reference
Python (including FLASK and FLASK_RESTFUL libraries / micro frameworks)	Implementation language	http://www.python.org

Table 13. Sharing Model baseline technology

5.3.7 Test cases

In this section we define a set of test cases for the Sharing Model functionality, based on the use cases previously described.

Test Condition #1 for Use Case #1: Sharing model is created

1. Test that the created sharing model is created when method is called.
2. Test if the CIMI call is executed and returns a correct response
3. Test if data management system received the data and the data is stored

Test Condition #2 for Use Case #2: Sharing model can be read via REST API

1. Test that the sharing model is fetched by ID
2. Test that the sharing model raises error when ID is not provided

3. Test that the sharing model is fetched using CIMI call and the correct response is returned

Test Condition #3 for Use Case #3: Sharing model can be updated via REST API

1. Test that the sharing model is fetched by ID
2. Test that the sharing model can be updated
3. Test that the sharing model calls CIMI interface
4. Test that the updated value is saved to Data Management system
5. Test that the update in Sharing Model information cannot be empty
6. Test that the update in Sharing Model is not called if the data does not change
7. Test validation details regarding the Sharing Model information

Test Condition #4 for Use Case #4: Sharing model can be deleted via REST API

1. Test that the sharing model is deleted by ID and is not present in Data Management system
2. Test that the error is raised when deleting not authorized Sharing Model

6. Conclusions

This deliverable has presented the implementation details of each of the mF2C Agent Controller functionalities. Each individual functionality has been described by means of a set of artefacts that detail the functions it offers to users or to other components, and its internal implementation to fulfil these goals. We have also provided a set of test cases for each functionality, which describe how it should be tested to ensure that it satisfies its functions.

The outcome of this document, together with D4.7 and D4.5, which specify the interactions between components and the implementation of the Platform Manager functionalities, will be the input for the integration and validation tasks in WP5.

Based on the experiences with IT-1 and evaluations of it, we can then add features and functionality for IT-2, and enhance the features already described in this document.

References

[1] "D2.6 mF2C Architecture (IT-1)," [Online]. Available: http://www.mf2c-project.eu/wpcontent/uploads/2017/06/mF2C-D2.6-mF2C-Architecture-IT-1.pdf.

[2] "D3.3 Design of the mF2C Controller Block (IT-1)," [Online]. Available: http://www.mf2c-project.eu/wp-content/uploads/2017/09/mF2C-D3.3-Design-of-the-mF2C-Controller-block-IT-1.pdf.

[3] "D3.5 mF2C Agent Controller block integration (IT-1)," [Online]. Available: http://www.mf2c-project.eu/blog/press-room/deliverables/.

[4] "D4.5 mF2C Platform Manager block and microagents integration (IT-1)," [Online]. Available: http://www.mf2c-project.eu/blog/press-room/deliverables/.

[5] "D4.7 mF2C interfaces (IT-1)," [Online]. Available: http://www.mf2c-project.eu/blog/press-room/deliverables/.

[6] mF2C. [Online].

[7] "D3.1 Security and privacy aspects for the mF2C Controller Block (IT-1)," [Online]. Available: http://www.mf2c-project.eu/wp-content/uploads/2017/06/mF2C-D3.1-Security-and-privacy-aspects-for-the-mF2C-Controller-Block-IT-1.pdf.

[8] "D4.1 Security and privacy aspects for the mF2C Gearbox block (IT - 1)," [Online]. Available: http://www.mf2c-project.eu/wp-content/uploads/2017/06/mF2C-D4.1-Security-and-privacy-aspects-for-the-mF2C-Gearbox-block-IT-1.pdf.

[9] "D4.3 Design of the mF2C Platform Manager block components and microagents (IT-1)," [Online]. Available: http://www.mf2c-project.eu/wp-content/uploads/2017/09/mF2C-D4.3-Design-of-the-mF2C-Gearbox-block-components-and-microagents.pdf.

[10] D2.4, D2.4 Security/Privacy Requirements and Features, http://www.mf2c-project.eu/wp-content/uploads/2017/05/mF2C-D2.4-Security-Privacy-Requirements-and-Features-IT1.pdf: [Online], 2017.

[11] mF2C project, D2.4 Security/Privacy Requirements and Features, 2017.

[1] «D2.6 mF2C Architecture (IT-1),» [En línea]. Available: http://www.mf2c-project.eu/wpcontent/uploads/2017/06/mF2C-D2.6-mF2C-Architecture-IT-1.pdf.

[2] «D3.3 Design of the mF2C Controller Block (IT-1),» [En línea]. Available: http://www.mf2c-project.eu/wp-content/uploads/2017/09/mF2C-D3.3-Design-of-the-mF2C-Controller-block-IT-1.pdf.

[3] «D3.5 mF2C Agent Controller block integration (IT-1),» [En línea]. Available: http://www.mf2c-project.eu/blog/press-room/deliverables/.

[4] «D4.5 mF2C Platform Manager block and microagents integration (IT-1),» [En línea]. Available: http://www.mf2c-project.eu/blog/press-room/deliverables/.

[5] «D4.7 mF2C interfaces (IT-1),» [En línea]. Available: http://www.mf2c-project.eu/blog/press-room/deliverables/.

[6] mF2C. [Online].

[7] «D3.1 Security and privacy aspects for the mF2C Controller Block (IT-1),» [En línea]. Available: http://www.mf2c-project.eu/wp-content/uploads/2017/06/mF2C-D3.1-Security-and-privacy-aspects-for-the-mF2C-Controller-Block-IT-1.pdf.

[8] «D4.1 Security and privacy aspects for the mF2C Gearbox block (IT - 1),» [En línea]. Available: http://www.mf2c-project.eu/wp-content/uploads/2017/06/mF2C-D4.1-Security-and-privacy-aspects-for-the-mF2C-Gearbox-block-IT-1.pdf.

[9] «D4.3 Design of the mF2C Platform Manager block components and microagents (IT-1),» [En línea]. Available: http://www.mf2c-project.eu/wp-content/uploads/2017/09/mF2C-D4.3-Design-of-the-mF2C-Gearbox-block-components-and-microagents.pdf.

[10] D2.4, D2.4 Security/Privacy Requirements and Features, http://www.mf2c-project.eu/wp-content/uploads/2017/05/mF2C-D2.4-Security-Privacy-Requirements-and-Features-IT1.pdf: [Online], 2017.

[11] mF2C project, D2.4 Security/Privacy Requirements and Features, 2017.