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

D3.3 Design of the mF2C Controller Block (IT-1)

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

Work Package	**WP3, mF2C Controller block design and implementation**
Due Date:	*M9*
Submission Date:	*30/09/2017*
Version:	*1.4*
Status	*Final*
Author(s):	*Eva Marín (UPC)*
Reviewer(s)	*Alexander J. Leckey (INTEL)* *Cristóvão Cordeiro (SixSQ)*

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)	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	26/07/2017	Internal version in UPC	Eva Marín Tordera, Jordi García Almiñana, Zeineb Rejiba, Souvik Sengupta, Alejandro Gómez Cárdenas
0.2	28/07/2017	First draft to distribute to the mF2C list	Eva Marín Tordera, Jordi García Almiñana, Zeineb Rejiba, Souvik Sengupta, Alejandro Gómez Cárdenas, Xavi Masip
0.3	28/08/2017	Internal UPC version with STFC, BSC, ATOS, UPC and TUBS contributions	Eva Marín Tordera, Jordi García Almiñana, Zeineb Rejiba, Souvik Sengupta, Alejandro Gómez Cárdenas, Xavi Masip, Jens Jensen, Jasenka Dizdarevic, Roi Sucasas Font, Toni Cortés, Anna Queralt.
0.4	29/08/2017	2nd Internal UPC version with STFC, BSC, ATOS, UPC and TUBS contributions.	Eva Marín Tordera, Jordi García Almiñana, Zeineb Rejiba, Souvik Sengupta, Alejandro Gómez Cárdenas, Xavi Masip, Jens Jensen, Jasenka Dizdarevic, Roi Sucasas Font, Toni Cortés, Anna Queralt
0.5	01/09/2017	Second draft to distribute to the mF2C list including contributions from: STFC, BSC, ATOS, UPC, TUBS, XLAB and WoS.	Eva Marín Tordera, Jordi García Almiñana, Zeineb Rejiba, Souvik Sengupta, Alejandro Gómez Cárdenas, Xavi Masip, Jens Jensen, Jasenka Dizdarevic, Roi Sucasas Font, Toni Cortés, Anna Queralt, Matic Cankar, Laura Val.
0.51	04/09/2017	Revision and ESA contributions on sections 4.3, 5.1	Antonio Salis, Glauco Mancini, Roberto Bulla
0.6	05/09/2017	Revision and BSC contributions on section 4	Daniele Lezzi
0.7	07/09/2017	Revision and ATOS contributions on section 5	Roi Sucasas Font
0.71	07/09/2017	Comments and style correction from STFC	Jens Jensen
0.8	08/09/2017	Modified section on runtime constraints in section 4.1, and contribution to section 6 from BSC and ESA	Daniele Lezzi, Anna Queralt and Antonio Salis
0.9	11/09/2017	Updated section 3.6 from INTEL	Alexander J. Leckey
1.0	12/09/2017	Contribution from TUBS in section 4.4 and comments from ATOS	Jasenka Dizdarevic and Roi Sucasas Font
1.1	15/09/2017	Third draft ready for 1st revision	Xavi Masip
1.2	22/09/2017	Revised document	Cristóvão Cordeiro
1.3	26/09/2017	Revised document	Alexander J. Leckley
1.4	28/09/2017	Revised document	Matic Cankar and Jens Jensen

Table of Contents

List of figures

List of tables

Executive Summary

This document has been developed by the mF2C project, intended to clearly describe the preliminary design of the mF2C Agent Controller, as envisioned for iteration IT-1.

The main objective of this document is to provide a comprehensive understanding about the Agent Controller, its main functionalities for IT-1, and a preliminary approach for IT-1 development. The document starts by summarizing the key decisions for the mF2C architecture assumed for IT-1, to make the design contributions proposed in the deliverable easy to understand, later introducing the set of main blocks and functionalities foreseen for the Agent Controller. Special attention is paid to describe each one of the three different blocks gathering all functionalities for the Agent Controller, namely, the Resource Management, the Service Management and the User Management, including design aspects related to the core functionalities envisioned for IT-1 as well as the interfaces required for the Agent Controller. Such functionalities are illustrated through different workflows, as the main foundation for further development. Finally, security is also considered as previously assessed in D2.4 for the whole mF2C system and particularly for the Agent Controller in D3.1.

The outcome of this document is a detailed design of the Agent Controller, including approaches for the different functionalities (monitoring, data management, resources discovery and identification, policies, SLA, QoS enforcement, etc.), including illustrative workflows that will also be essential for the next stages of the development, as well as a preliminary design of the data base to be considered in mF2C.

1. Introduction

1.1 Introduction

In deliverable D2.6 [1] we defined and designed the mF2C architecture considering some assumptions for the implementation in iteration IT-1. The main component of this hierarchical architecture is referred to as the agent. A key disrupting idea coming up from the mF2C project leverages the fact that any kind of device with enough computing capacity could participate in the mF2C system, from a server to a single board computer such as a Raspberry or a smart phone. To make it possible, the device would only be required to install the agent and start a registration process. The agent will have all the management and control functionalities for the device to become a participant in the mF2C system.

In the architecture presented in D2.6, there are two main concepts to be highlighted. The first refers to the hierarchical approach, introducing the concept of an agent serving as leader. All agents are grouped in clusters (fog area) managed and controlled by a leader. The second one, refers to the fact that the whole set of management and control functionalities has been divided into two big blocks, the Platform Manager (PM) described in deliverable D4.3, and the Agent Controller (AC) described in this deliverable. In short, the PM provides high-level functionalities, responsible for inter-agent communications (agents communicate through their PMs) and thus, with the capacity to take decisions with a more global view. On the other hand, the AC has a local scope, that is, focusing on the device's local resources. It is worth mentioning what local resources for an Agent Controller are. In an agent, the local resources are only its own local resources. In a leader, the local resources are all resources within the cluster.

As a short summary, main functionalities for the Agent Controller are:

- Filling the resources database with information about local resources
- Filling the services database with information about services/tasks being executed/or to be executed in its local resources
- Filling the user database with information about the users participating/utilizing resources and services in its local resources

These three basic functionalities will require additional functionalities to make it work, such as: discovery, monitoring, identification, categorization, etc., all described in detail in the different sub-sections of this deliverable. It is also worth highlighting that the mentioned database is unique per agent and is shared between the Agent Controller and the Platform Manager.
Other functionalities are:

- Assigning resources to requested execution tasks (Mapping), in this case only to its own local resources.
- Guaranteeing that the mF2C services running in the device meet the constraints defined by the user.
- Setup and management of such user's profiles with roles and permissions

The exact functionalities and the detail of the proposed way of implementing them are described in the different sections of this deliverable.

1.2 Purpose

The objective of this deliverable is to provide a deep description of the Agent Controller (AC) block, which is one of the two main components of the agent architecture proposed in deliverable D2.6 [1]. To this end, and after the review of the proposed architecture for IT-1 in section 2, the different sections correspond to the main components of the agent controller sub-blocks., which are the resource management, the service management and the user-management sub-blocks, described in Sections 3, 4 and 5 respectively. Beyond the mentioned sub-blocks, another main component of the

Agent Controller is the database. The database is designed to be shared by both, the Agent Controller (AC), described in this deliverable, and the platform manager (PM), described in the deliverable D4.3. The design of this common database and its interfaces are proposed in Section 6. All the interfaces for communicating the internal sub-blocks of the Agent Controller are described in Section 7. Finally, in section 8 we present an illustrative example with a preliminary implementation of some of the AC functionalities.

For the sake of better understanding about the operation of the different proposed blocks, this deliverable also includes some workflows describing the functionalities of the sub-blocks.

1.3 Glossary of Acronyms

Acronym	Definition
AC	Agent Controller
ACK	Acknowledgement
API	Application Programming Interface
APP	Application
CA	Certification authority
CPU	Central Processing Unit
DB	Data Base
device_ID	Device identificator
ESD	Extreme Studentized Deviate
FA	Fog Area
GPU	Graphics Processing Unit
HDD	Hard Disk Drive
ID	Identificator
ID_key	User identificator
IoT	Internet of Things
JSON	JavaScript Object Notation,
mF2C	Towards an Open, Secure, Decentralized and Coordinated Fog-to-Cloud Management Ecosystem
MQTT	Message Queue Telemetry Transport
OID	Object ID
PCM	Performance Counter Monitor
PM	Platform Manager
QoS	Quality of Service
SDD	Solid State Drive
SLA	Service Level Agreement
SLO	Service Level Objective
SQL	Structured Query Language
UML	Unified Modelling Language
USK	User Secret Key

Table 1. Acronyms

2. Summary of mF2C architecture for IT-1

The mF2C architecture has been described in deliverable D2.6 [1], as a layered architecture where resources are categorized according to a certain policy and a so-called agent entity is responsible for deploying the management functionalities in every component within the system. That is, all devices participating in the mF2C architecture should install the agent software to make the agent entity work.

On the other hand, the mF2C project development has been defined following an iterative approach, thus setting two iterations, the first, IT-1, ending in M18, and the second, IT-2, in M36 –as a final implementation of the mF2C system. Figure 1 depicts the functional control architecture we identify for IT-1, mainly considering the following assumptions:

- Only one cloud is considered.
- Only three layers are considered. These layers are logical layers considered for management and control.
- There is no horizontal control communication among nodes, so service requests are only vertically forwarded.
- Fog area stands for the set of nodes managed by a leader.
- Only one node acts as leader in each fog area. In Figure 1, nodes in fog layer 1 are leaders of the fog areas formed by each of them and the agents in fog layer 2.
- Only one backup node, in each fog area, is considered for robustness. In Figure 1, one of the agents in fog layer 2, acts as a backup node. And in case that the leader in fog layer 1 fails, the backup node will become leader and will belong logically to layer 1.
- Resources information will be stored in a database and its management will follow a simple approach just to verify the system works.
- Mobility is only considered at fog layer 2.
- Functionalities to be included will be a subset of the total identified, sufficient to show mF2C system performance and benefits.
- IoT devices can be connected to any of the agents in the mF2C system.
- Control communication among agents is always done through the Platform Managers.

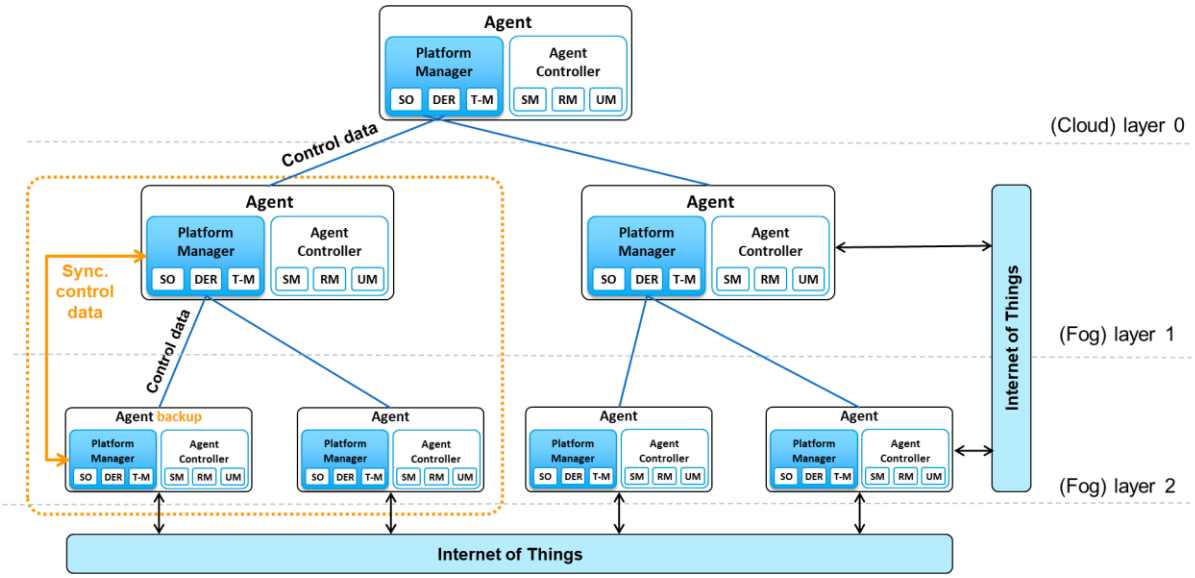

Figure 1. mF2C architecture for IT-1

As it can be seen in Figure 1, the agent software is installed in all the devices (nodes) participating in the mF2C system. That is, all the nodes in the architecture are agent[1]. The two main building blocks that comprise the agent entity are: the Platform Manager and the Agent Controller. The Platform Manager is the block responsible for the orchestration of services –based on the computing, storage and network resources –through a full-stack monitoring system that receives telemetry data from different sources–, which is in depth described in deliverable D4.3. On the other hand, the Agent Controller only manages the local resources, services and users of each individual agent and will be largely described in this document.

2.1 Survey of main Agent Controller Functionalities

This section reviews the main components for the Agent Controller as proposed in deliverable D2.6 [1]. The set of functionalities for the Agent Controller is split into three main blocks, Resources, Services, and Users. The first block, the Resources, will mainly encompass all functionalities dealing with resources management, including for example, discovery, naming or categorization, just to name a few. The second block, the Services, will include functionalities related to services, such as task execution, mapping, etc. Finally, the third block, the User, will include functionalities, such as QoS enforcement or user profiling, highly related to the user expectations, demands and characteristics. The main functionalities of the Agent Controller are shown in Figure 2.

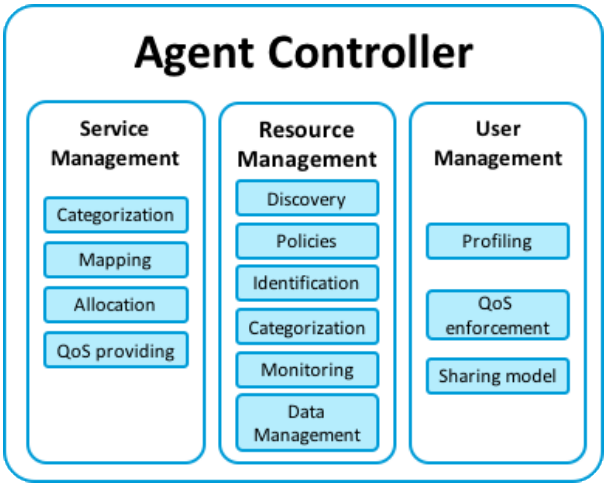

Figure 2. Agent controller main functionalities

Regarding the resources, the main functionality of the Agent Controller is to fill in the tables information in the agent database, related to the resources. It is worth mentioning that the agent database is shared by both the Platform Manager and the Agent Controller. This database must include local information related to the local resources, and its size would depend on the strategy decided to manage the data (cache with minimal information, aggregation policy, etc.).

Related to the services, the Agent Controller acts as a "slave" of the Platform Manager. It is important to remind two different aspects of the proposed architecture: 1) Control communication is always done among Platform Managers, and; 2) All the smartness of the system is also located at the Platform Manager. Figure 3 is shown to illustrate these two characteristics.

[1] In fact, other kind of devices can participate in the mF2C system if they are attached to an mF2C agent, for example sensors (represented as IoT in Figure 1) or even devices with computing capacity but without enough capacity to have an mF2C agent installed.

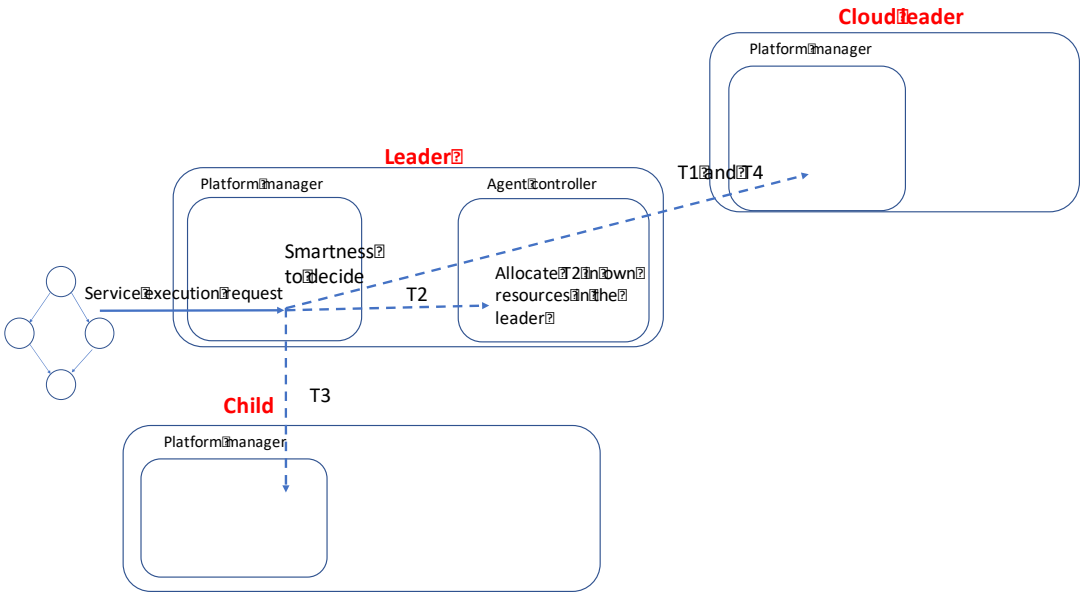

Figure 3. Example of PM and AC functionalities.

In Figure 3 we show an example where a service is requested to be executed in the mF2C system. Let's assume the service is composed of four tasks, i.e., T1, T2, T3 and T4 – the PM will also have the capacity to divide the requested service into different tasks, not shown in the figure though. In the example, we see that the PM has the capacity to decide what should be done with the requested tasks. Let's also assume that the service is initially requested to a node acting as leader (this is not a constraint though, and other examples may be proposed where the service is requested to any node in the architecture). The PM in this node (leader), according to the resource information in its database, which includes information about its own resources and about its children, decides the following for each one of the tasks for the requested service:

- it cannot handle tasks T1 and T4 (this means they cannot be handled in the node, the leader in this case, nor in the set of nodes under its responsibility) and consequently these tasks are forwarded to the upper layer, in this case the cloud agent.
- Task T3 can be executed in one of its children, and then task T3 is forwarded to the lower layer, to one of its children.
- Task T2 can be executed in the own device's resources and then the task execution request is forwarded to its Agent Controller.

It is worth mentioning that, for tasks T1, T3 and T4 (two first bullets), when the task is forwarded up or down in the hierarchy, the PM is the one forwarding the request to the PM located in the upper or lower layer.

With this example, we can see the mentioned two characteristics of the architecture, but also the main functionality of the Service Management component in the Agent Controller, namely, to allocate, execute and control the requested tasks in local resources, understanding local resources as its own resources.

Finally, the User Management block in the Agent Controller has all the functionalities related to the management of the profiles, the QoS enforcement and the sharing model of the users who have access to the mF2C system and the various services running on top –including aspects related to the definition of rules for the shareable resources residing in the devices (CPU, disk, battery, etc.) and the user roles definition. It is interesting to comment that, in an mF2C system a user could play a role as a consumer, a provider, or most usually play both roles. In other words, a device in the mF2C system may be a service consumer, using the mF2C system to execute services, but it can also be a provider, i.e., a resource provider under a collaborative sharing model yet to be defined. In this sense, the user profile should include all these particularities. For example, the Profile component in

the User Management block could include which is the QoS agreed with the user to execute services (QoS enforcement), and also the conditions to share his/her resources with the mF2C system (sharing model).

2.3 Security provisioning

Deliverable D2.4 [2] gives a comprehensive background into IoT security in general. Deliverable D3.1 [3] and D4.1 [4] are more specific and cover security and privacy for the controller and platform manager respectively. D3.1 describes the security policy for data in mF2C, taking into account the different capabilities of the devices, and lists the (expected) security features of each of the architectural layers; D4.1 also looks at security from the perspective of the use cases. The basic idea is to have three levels of protection: "private" for personal or sensitive data; "protected" for data which is not secret but needs integrity protection (such as advertised services), and "public" for data which needs no special protection.

The Agent Controller must with no doubt implement this security model. For example, registering resources should be permitted only for authenticated and authorized entities, and the information needed for each entity to authenticate itself is typically private. However, once a list of resources is available, this list needs only to be protected: every potential consumer of the resource should be able to see what is available, but an unauthorized entity should not be allowed to modify the list or database of resources (otherwise it could elevate its privileges, e.g. through phishing.) Statistics on overall resource availability could be made public (which doesn't say data must be published, merely that it needs no special protection.)

2.3.1. Controller Security Prototype

From March to August 2017, STFC had a small team working on a prototype security implementation, working alongside with the team writing the associated deliverables (D3.1 and D4.1). The purpose was to have a proof of concept implementation of the security policy and Agent Controller, with code being tested in Arduinos and Raspberry PIs [5] [6]. The implementation used MQTT over private networks; the idea being that the Leader agent would provide the MQTT broker which would enable the remaining parties to communicate, and the security classification was introduced through the MQTT topic. The payload of the message was JSON (RFC 7159 [7]), as it is simple enough to parse even for an Arduino, and JSON also supports signatures (RFC 7515 [8]) and encryption (RFC 7516 [9]). However, signatures and encryption were not implemented in the prototype.

2.3.2. Controller Security API

Building on this work, the next step is to design an API upon which agents and applications for IT-1 can be implemented. After discussion within the mF2C consortium, the API will be initially implemented in Python and Java. The AC security API must enable its caller to:

- bootstrap its identity;
- in particular, for services that need to be able to sign messages, a certificate is generally needed;
- authenticate;
- classify its generated, stored, or sent data according to the security policy (see 3.5);
- determine the security requirements of received data.

3. Resource Management Design

3.1 Discovery

In order to make it possible for users to contribute and to share their resources with the mF2C system, it is necessary to adopt suitable mechanisms to notify them about the existence of mF2C instances close to their location. This functionality is achieved by the discovery module.

3.1.1 mF2C discovery in proximity

In the following, we describe the process proposed to enable mF2C discovery close (i.e., in proximity) to the network edge, along with the motivations behind the proposed approach.

mF2C discovery in proximity is made possible through the use of the beacon stuffing approach [10], which consists in embedding customized information inside specific fields of standard-compliant 802.11 beacons. More specifically, as shown in Figure 4, information about mF2C-support will be advertised within the vendor specific information element of the 802.11 beacon [11]. Following the element ID (equal to 221, DD in hexadecimal) and the length fields, an Organizationally Unique Identifier (OUI) will be included. At this stage, we can use the FF:22:CC OUI, which has not been assigned to any organization yet, as an OUI for mF2C. The actual beacon payload carrying the mF2C-related information is provided next. In addition to the ID of the leader broadcasting the beacons, this payload will also contain information that will help the contributor decide whether to contribute or not, such as a user-friendly description of the kind of tasks in which its resources may be used, a field describing how urgently the resources are needed, etc. This information will be embedded in an efficient manner in order to keep the frame size as small as possible. After that, the mF2C-enhanced beacon will be broadcast by the leader within its area according to a schedule strategy to be dynamically determined by the policies.

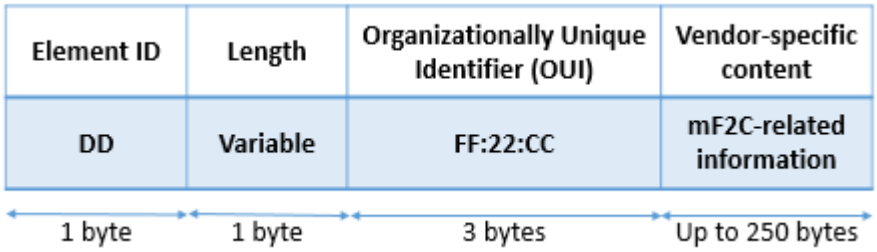

Figure 4. 802.11 Vendor-specific information element adapted to mF2C

From a contributor's perspective, a device with the mF2C agent installed, will start to scan for these special frames as soon as the agent is turned on. Consequently, the device will be able to detect them whenever it enters an mF2C-capable area where the advertisements are transmitted. After extracting the mF2C-related information from the corresponding frame fields, the agent at the contributor's device will verify whether the message is mF2C-compliant and if it is indeed coming from a valid mF2C device (using the advertised leader ID). It will then check the advertised characteristics and send an acknowledgment to the corresponding leader in case these characteristics match its contribution preferences.

As it may be noticed, the proposed discovery process does not involve any extra message to be exchanged. The necessary discovery-related information is instead included inside the beacon message, without requiring a pre-established network connection. The only additional processing at the contributor's side will result from the analysis of the information embedded inside the beacons. With that in mind, this information processing will be designed in a resource-efficient manner. It is worth also mentioning that the 802.11 standard is used. This is mainly motivated by the fact that most of the computing-capable devices (consequently mF2C candidate devices) are equipped with a

802.11 interface. In addition, it is characterized with a ~100m range, which is well-adapted to local area mF2C deployments.

It is worth mentioning that a strategy should be designed to prevent misbehaviour driven by potential fake leaders. Indeed, a device can replicate the beacon and act as a false agent.

3.1.2 mF2C general discovery framework

The different layers of the mF2C hierarchy are characterized by both a heterogeneous landscape of networking technologies and the different roles and interactions among them (leader - contributor at the lowest level, leader_level_N – leader_level_N+1 at higher levels). As a result, a unique discovery strategy cannot be applied to all levels; rather different discovery approaches should be designed to cover such heterogeneity and roles diversity. To that end, an approach which is conceptually similar to the above described process might be used, tuned according to the inherent characteristics of each layer, turning into the following tentative general framework:

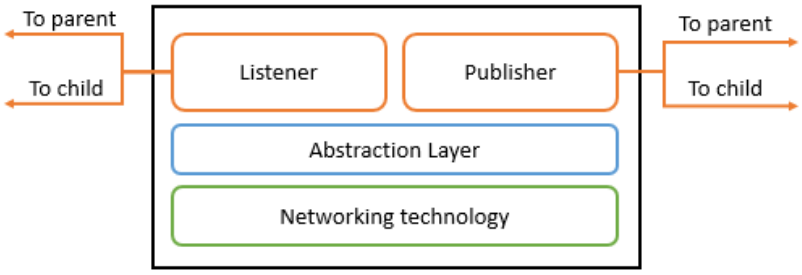

Figure 5. mF2C general discovery framework

The abstraction layer represented in this figure is responsible for hiding the details of the underlying network technologies from the upper layer. This upper layer consists in two elements: a publisher and a listener. The publisher is mainly in charge of framing mF2C-compliant discovery messages and sending them either to its parent or to its children. On the other hand, the listener is responsible for capturing these mF2C-compliant messages and interpreting them.

However, other possible strategies may be also used, based on enriching existing contributions such as for example NDP (Neighbour Discovery Protocol [12]) in IPv6.

3.2 Identification and Naming

The identification process by which every device participating in the mF2C environment gets a unique identifier (ID_key), is the first action to perform in order to get the device ready for using the mF2C system.

Previous to the registration, the user should download and install the mF2C agent in every device he/she wants to use in the mF2C system. It is worth noting that the download and installation method may vary depending on the device's operating system.

Once all user's devices have the mF2C agent installed, the user must obtain a valid ID for each one of his/her devices (device_ID). To that end, every user (being user a person, institution or entity) must register into the website enabled for the mF2C service provider. The registration process will be done only once per user, institution or entity, regardless the number of devices he/she/it wants to use in the network.

During the registration, the users will access to a public website hosted by the mF2C service provider. Registration will start when the user registers a valid electronic mail address and accepts the use conditions. Afterwards the system will generate a unique user secret key (USK), so-called ID_key, which will be unique per user.

The ID_key, will be downloaded into the user's device after the successful registration and in addition, an email will be sent to the user for ID_key recovery purposes. Options to recover, update and revoke the ID_key will be included in the same website.

Once the user has installed the mF2C agent software, registered and obtained the ID_key in his/her device, he/she will be asked for loading the key in the agent, specifically to the ***Identification*** sub-block in the Agent Controller (AC). Also, an additional string that the user can type or generate randomly will be required. In no case the same string can be used more than once with the same ID_key, but the same ID_key can be used with different strings for different devices of the same user to generate different device_IDs. The Identification sub-block is the component responsible for generating different strings for the different devices of the same user.

The concatenation of both the ID_key and the additional string will be used for the agent as the input of a hash function that uses the SHA-256 algorithm [13] to generate a hash value, which will be the device_ID (Figure 6).

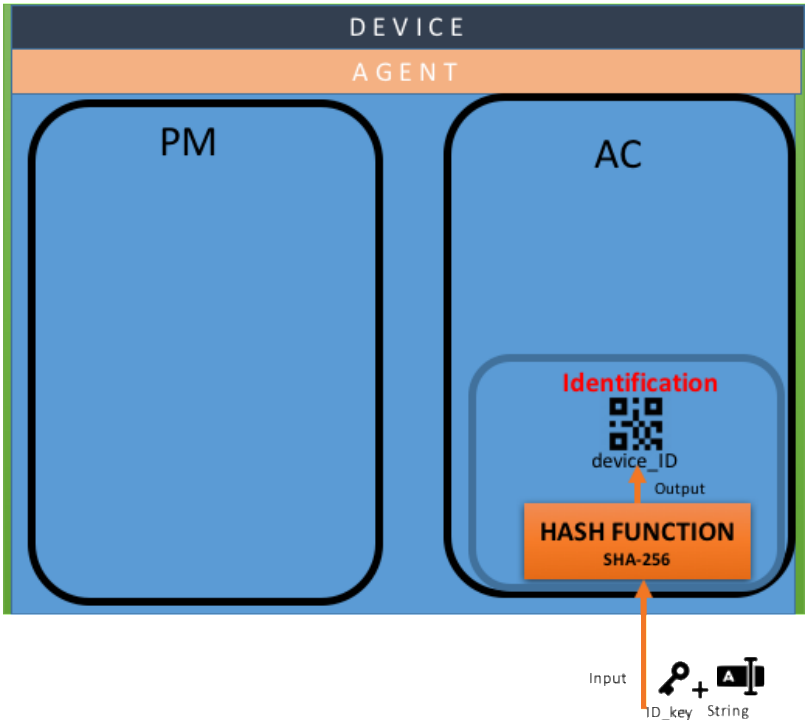

Figure 6. ID calculation in the agent

3.3 Categorization

Devices participating in the mF2C system with the agent software installed, so-called mF2C Capable resources, should be categorized according to an ontology yet to be designed. The results of this resource categorization/classification will be necessary to properly and accurately handling the matching between requested services and resources where services are to be allocated to. Figure 7 presents a preliminary approach to this ontology, showing the class diagram of mF2C resource categorization. It is worth highlighting that the proposed classification is designed to be open, hence to include any potential new component to be categorized in the future.

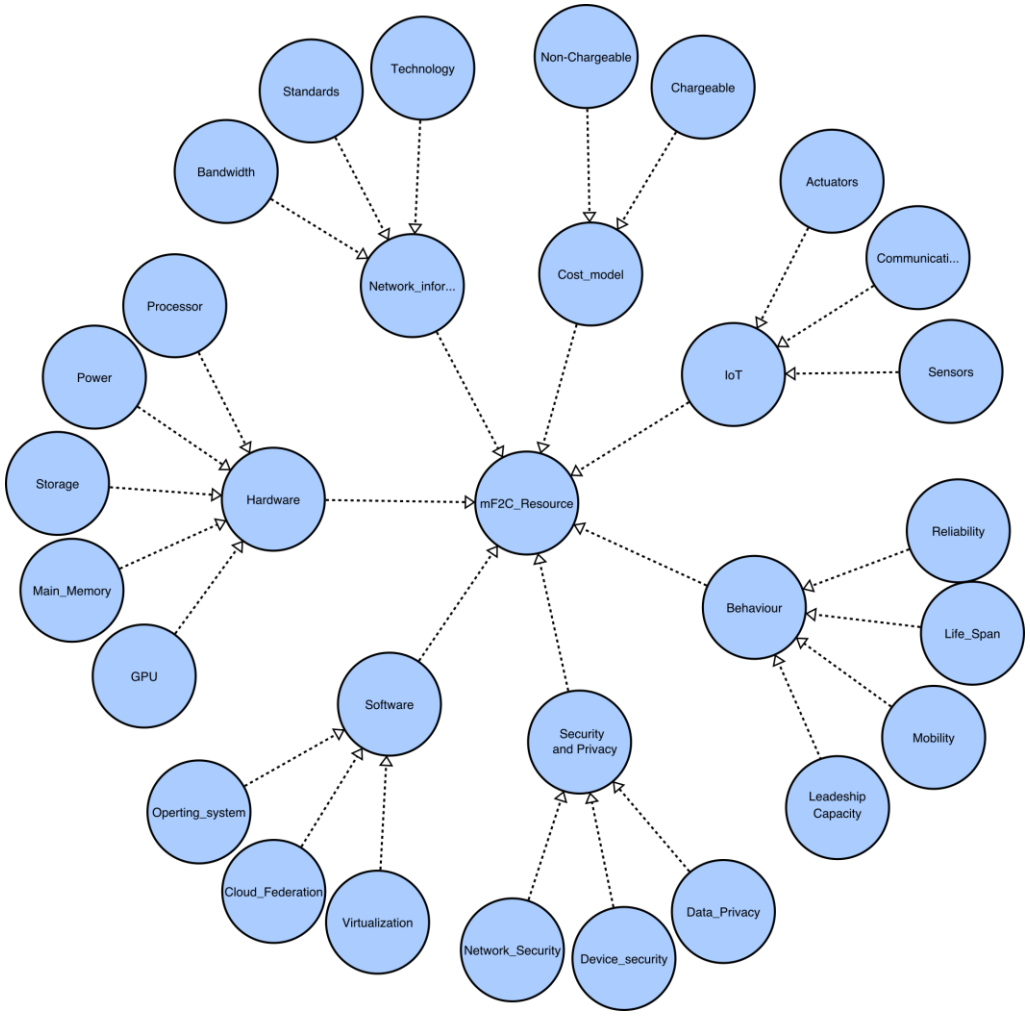

Figure 7. Resource Categorization

In this first level of classification, the next characteristics are proposed to categorize the resources:

- Hardware: Representing the hardware characteristics of the device. In a second level, we could consider, for example:
  - Power (information of power source, remaining power, etc.)
  - Main Memory capacity (total, available, shared)
  - Storage capacity (total capacity, available capacity, shared)
  - Processor information (number of cores, architecture, Clock speed etc.)
  - GPU Information (Graphic card owner information, model, GPU memory, etc.)
- Software: Representing the software included in the device. In a second level the next subcategories might be considered:
  - Operating System
  - Virtualization (does it support virtualization?)
  - Cloud federation (does the cloud agent support cloud federation?)
- Network: Representing the means and characteristics to connect with the device. In a second level, we'd propose to consider:
  - Technology (WiFi, Bluetooth, Zigbee, LORA, 3G, 4G, 5G, etc.)
  - Standards
  - Bandwidth (or data rate)
- Cost model: Representing the potential cost the device utilization may bring in. Subcategories for this bullet may be:
  - Non-Chargeable

    - Chargeable (in terms of price, compensations, advantages, etc.) based on per unit usage of Processor, Storage, Power, Network etc.
- Security: Representing the security level at the device. Tentative subcategories may be:
    - Network security: YES/NO
    - Data security capabilities: Private, Protected and Public
    - Device Security: The Device that participates in the system has the hardware level security or not: YES/NO
- IoT: Representing the set of potential devices connected to the device. For example, does the device have IoT devices connected? Which are the characteristics of these devices? A second level may include:
    - Sensors (Type: Temperature, Humidity, Proximity)
    - Actuators
    - Communicating Node: Access points, gateways to transmit data between sensors/actuators to the mF2C capable device.
- Behaviour: Representing how the device behaves according to different parameters. Some of these parameters can be characterized by either the same node, or the node's leader. Below a list of potential characteristics is presented:
    - Lifespan. This can be handled by the node itself.
    - Mobility (It is mobile or not). This can be handled by the node itself
    - Leadership capacity: A leader or backup should be able to attach multiple mF2C agents.
    - Reliability. This is ranked by the leader.

The design of the ontology to be used for resource categorization will have a direct impact on the database design proposed in Section 6.

3.4 Policies

In this section, some of the policies that will rule the functioning of the resource management modules are described. Any entity (virtual or physical devices) willing to join the mF2C system –therefore operating under the command of these polices–, must have the agent properly installed and configured.

At this stage, policies presented below are very generic, thus they can be used in a wide range of use cases. Nevertheless, new policies with a higher complexity level are expected to come up in the future.

3.4.1 Leader and backup node selection

Every fog area will comprise a leader node, at least one agent, and a backup node when possible. Below are the policies that describe how those roles are assigned:

- Every node must calculate a function value FA_{func}, using as input the information obtained from the resource database.

  The FA_{func} is based on a set of elements and their respective weights. The elements of the function may vary according to the use case, for example:

$$FA_{func} = (w_1 \times cpu) + (w_2 \times battery\ power) + (w_3 \times mobility) + (w_4 \times bandwidth) + (w_5 \times storage) + (w_6 \times memory) + (w_7 \times location)$$

  The weight factor (w_i) assigned to every element of the equation enables weighting the entities according to the administrator preferences or use case needs. It is yet to be designed how the distinct elements, for example mobility, are measured.

- The leader of a community will be that entity with the higher FA_{func} value.
- In the case of two or more nodes with the same FA_{func} value, the leader will be the non-mobile node. If more than one satisfies this condition then the leader will be chosen randomly.
- The backup node will be the node with the second highest FA_{func} value.
- When two or more nodes share the second higher FA_{func} value, the backup will be the non-mobile node. If this condition is satisfied for more than one node, then the backup will be the node with lower hops to the leader. In the case of two or more nodes meeting both conditions the backup node will be chosen randomly.
- The nodes that are not playing the role of leader or backup will assume the regular node role.
- Every *x* seconds the backup node will check with the leader node if new updates are available. In this case, *x* can be a fixed or a dynamic value. It will be fixed when it is chosen by the system administrator and dynamic when the system determines the optimal value according to the size of the community population or any other parameters.

The fog areas will consist of different nodes (devices members of the mF2 system) with a common leader node and backup node. It is still under study how to decide which are the nodes belonging to a fog area, but in any case, these nodes must have connectivity to the leader node.

3.4.2. Discovery policies

Regarding discovery strategies, the mF2C system should cover different kind of policies. For example, considering the policies related to the leader some of them are 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.

The first policy pertaining to the discovery module is related to the transmission frequency of advertisement beacons. This frequency will be dynamic and may be adjusted based on historical arrival patterns of contributors in a leader's area. It can also be tuned according to how urgently the leader needs resources from the potential contributors.

The second policy is related to the maintenance of the discovery state and it consists in determining how often the leader transmits "keep alive" messages. These messages are used by the leader to infer the user absence, when the latter stops sending them for a long period of time. This policy will take into account the need to minimize the transmission cost at the user device side while ensuring that the user-leader association is kept alive as needed.

3.5 Data Management

In an IoT context, most data is generated at the edge of the network and may be transferred more or less frequently to a repository, either centralized or distributed. Such data can then be accessed by the users' applications to implement the catalogue of smart services. In addition, these services can also generate other kinds of data at any other level in the platform, such as new data derived from calculations using different IoT sources, which may also be stored and consumed by applications.

Since data may be of arbitrary types not known a priori (sensors measuring different kinds of observations under different parameters, different kinds of data generated by applications, …), and their intended behaviour may vary depending on the application, the application developer must define the classes and methods that implement the most appropriate data structures in each case. For instance, one such class may represent a sensor, or a dataset containing a certain kind of data such as temperatures, to which the application can request access according to the access methods defined (e.g. get all the temperatures within a period of time, get the highest temperature in the last

month, get the current temperature, …). We may also consider the definition of some common attributes to all data to ease the data discovery, such as for example geographic device location, type, time stamp, etc.

Internally, this class may either connect to a sensor to get the data on demand, or just return it because it is already stored in the objects of the class. The developer will choose one of the two options depending on the application requirements. For instance, if a complete history of observations is required, the class needs to periodically get data from one or more sensors and store it. The data management functionality will enable access to the data in this class from any node where the application using the class runs. If only real-time observations are needed, then the class might request data from the sensor(s) when required, without the need to store it.

With this mechanism, the data management functionality provides a uniform object-oriented interface for applications to access any kind of data in the platform, regardless of its location, of its source (data provided by an IoT device or generated by the application itself), and of whether it is persistent or volatile. Part of the functionality required to achieve this behaviour corresponds to the Platform Manager and is described in D4.3. The data manager in the Agent Controller is in charge of the following functionality, when receiving the corresponding request from the Platform Manager in the same agent:

- Storing data in the form of objects
- Retrieving an object (or part of it) given its OID (Object ID)
- Storing/modifying an object (or part of it) given its OID and the new/modified data
- Implementing “queries” over a collection of objects
- Deploying a set of classes when the platform manager requests it

These functionalities are implemented by extending the data service component in dataClay [14], and its deployment in the overall mF2C architecture is to be decided. It seems reasonable to consider that fog caches more recent data and cloud stores persistent data.

Storing/retrieving/modifying data objects

In order to store objects, dataClay uses a key/value store where the key is the OID and the value is the serialized version of the object data. In the current version we are using PostgreSQL, though any other Key/value store would also work.

With respect to the serialization, dataClay does not use the standard serialize method because it uses reflection and adds too much overhead to the process. Instead of using the standard serialization, dataClay automatically builds an optimized one for each class. To be able to build this optimized serialization, classes need to be registered into the system before an object belonging to that class can be stored. With the information of the class (fields, types, and methods), dataClay can implement a perfectly optimized serialization method that does not use reflection because it already has all needed information.

It is important to understand that complex objects such as collections or objects with references are not stored as a single object. For instance, a collection with N objects will imply storing the N objects using their own OID, and then another one with the collection information that will also have an OID. This behavior mimics the way data is stored in memory by object oriented programming languages where OIDs are object references and each object is stored in its own portion of memory.

Implementing queries over a collection of objects

DataClay offers two mechanisms for querying on collections. The first one is to implement an iterator that iterates over some of the collection objects. This can be implemented when implementing the collection class. The second option is to implement a method that iterates over all elements in the collection and checks which ones fulfil the query constraints.

Furthermore, like in all databases, we can implement indexes (or similar structures) to speed up queries.

It is important that all these query methods are registered with the class so they can be deployed in any agent controller with objects of that class, and thus enable them to execute such queries.

Deploying a set of classes

As we can see, it is important to have information on the class fields and type of all objects as well as the implementation of some methods such as iterators, queries, serialization, etc. For this reason, agent controllers need to be able to deploy classes (currently Java and python) in order for the class loader to be able to load the important information of the class and thus use the right serialization and query methods.

3.6 Monitoring

Instrumentation of each resource will be carried out by the *Monitoring* framework which will deploy a collection of probes that will capture telemetry relating to the "full stack" of a service deployment. This will feed the *Analytics* module of the Platform Manager with a view to extracting optimal hardware and software configurations for the placement of services. Metrics will include: benchmark (e.g. per query latency), Middleware/Database, system metrics (e.g. per core, per socket, system utilization), hardware counters (e.g. Processor Counter Monitor) and environment (e.g. temperature, power consumption).

Each probe will be categorized as a *Collector*, *Processor*, or *Publisher*.

- *Collectors* are software probes collecting any form of metric from any software source. Examples include the host operating system, hypervisor or Virtual Machine, guest operating system, middleware, or hosted application.
- *Processors* are responsible for passing captured data through aggregators that analyze and perform some action on the data, e.g., calculating mean, standard deviation, etc. The results of one probe may influence the collection frequency of another, e.g., battery life impact the collection of a disk I/O probe.
- *Publishers* will take processed data to be published to arbitrary destinations. Typical endpoints will include time series databases (eg, InfluxDB), message queues (eg, Mosquito/MQTT), or analytics engines.

A typical example of a probe would be one that queries Intel PCM (*Performance Counter Monitor*). This utility provides a set of metrics that allows the acquisition of data regarding internal processor events, including information about components found on the processor chip such as the memory controller, cache controller, etc.

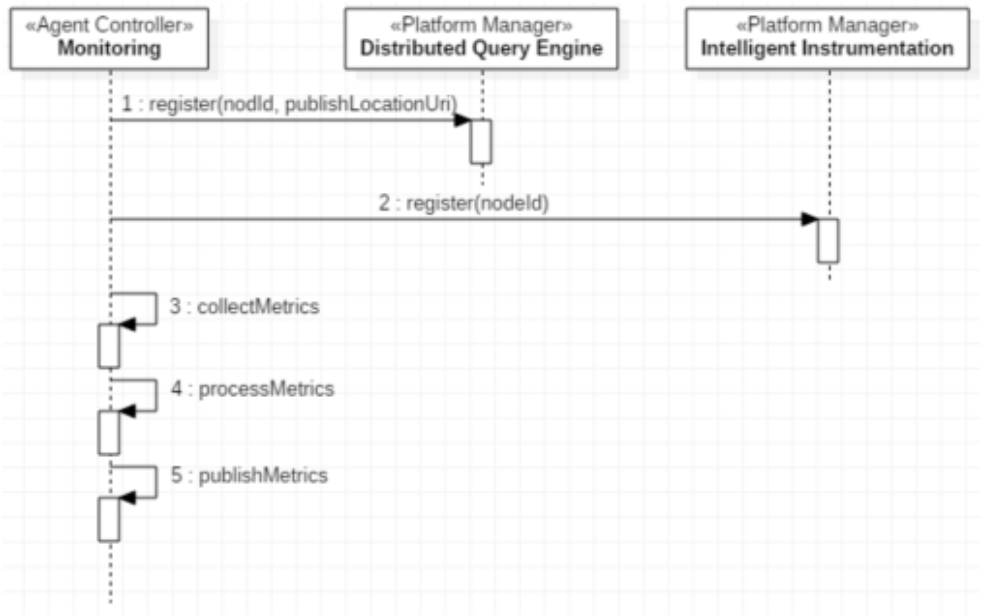

Figure 8. Each monitoring probe will be required to register before collecting

As each probe on the Agent Controller has the potential to publish telemetry at any arbitrary location in the hierarchy of the mF2C system, this will make querying of these metrics difficult within the Platform Manager. As such, all probes will be required to register with the *Distributed Query Engine* module of the Platform Manager so that it knows where to retrieve metrics for that particular node. This will allow the query engine to provide a single API that abstracts access to publish locations of metrics.

Each probe will also register with the Platform Manager's *Intelligent Instrumentation* module so that it can analyse the output with a view to throttling publishing frequencies depending on the output of analysis, e.g., anomaly detection, battery degradation, etc.

A typical example might be a probe that collects 10mb of telemetry data per minute. However, if nothing of interest (anomalies, data spikes, etc) has occurred during that time window, the *publisher* could be prompted to only publish that minute's average therefore utilizing less storage space. Similarly, if an anomaly was detected during this 1 minute window, the publisher could actually publish the entire 10mb of telemetry data to aid the identification of the root cause.

A number of approaches exist to aid adjusting publishing frequencies based on anomaly detection include:

- "*Tukey*" statistical analysis
- Contextual using Term Frequency

Seasonal Hybrid ESD techniques for detecting anomalies in seasonal univariate time series.

3.7 Core Resource Management operation

In Figure 9 we show a workflow putting together the main (designed to the core functional) operations of the resource module in the Agent Controller, showing the leader registration, pre-registration, user registration, device identification, device discovery, post-discovery maintenance and disconnection operations.

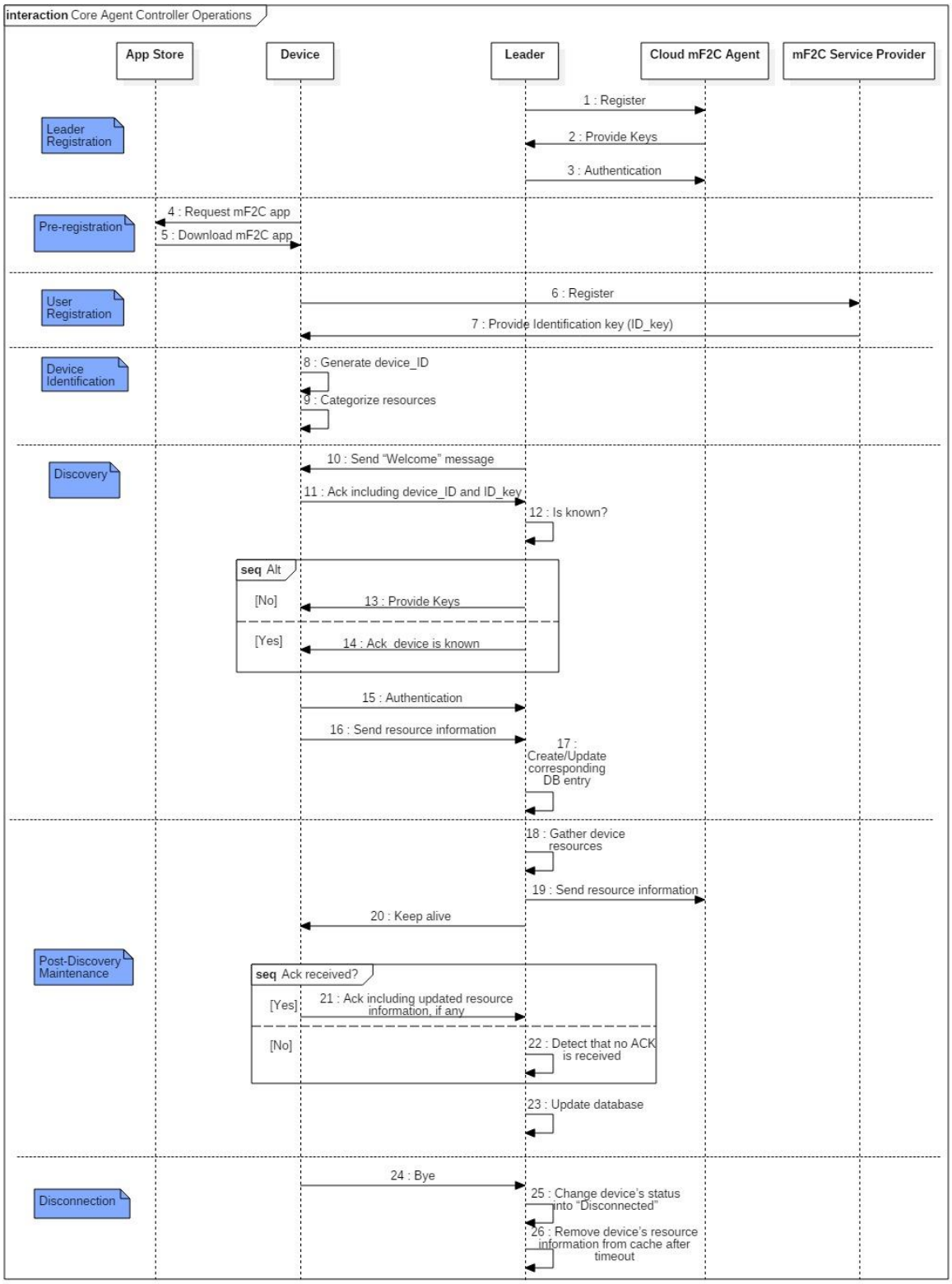

Figure 9. Workflow including: Registration, identification, Discovery, Key distribution.

3.7.1 Workflow description

It must be highlighted that since all control communication is supposed to be managed by the PM, all messages included in Figure 9 generated by the different AC blocks are first sent to the PM to be forwarded outside the agent towards the leader.

Leader registration

1. Register

The chosen leader would be registered at the cloud agent. The policies to select the leader and the backup leader are still to be defined. Either way, these policies may depend not only on the agents' capabilities but also on the services and their requirements. Policies may require the backup leader to be registered at cloud or may not.

2. Provide keys

The cloud agent provides keys to the leader, and to the backup leader when necessary, for providing authentication and secure communication. The functionality to provide such keys at the cloud agent is still to be defined. One potential approach as described in D3.1 focusses on using a decoupled security architecture responsible for generating and distributing keys.

3. Authentication

According to the keys, the leader and the cloud agent will be mutually authenticated and the leader is authorized to become a leader.

Pre-registration

4. Request mF2C app:

This is a one-step process involving the mF2C user and an app store such as the conventional ones, be it an existing one or a new one. The goal is to get the agent software to be installed on the user's device to make it mF2C-capable.

5. Download mF2C app

The mF2C app is downloaded in the user device and ready to be configured. The mF2C app should support different systems (Windows/Linux/Android/IOS).

User Registration

6. Register

The user registers at cloud basic information that can be used for statistical purposes and as a result the system gives a secret key to the user, referred to as Identification key (ID_key). The information that the user must register and the moment of the registration are not defined yet. Indeed, the registration process could take place before, during or after the user is downloading the mF2C app. In the two first cases, the registration must be done through a web form, in the third one, using a form included in the application.

7. Provide Identification key (ID_key)

Once the user has submitted the registration form to the mF2C service provider (wherever it is), it will return the Identification key, (ID_key) for the user.

Device Identification

8. Generate device_ID

The device_ID will be calculated using a hash function with two inputs: the Identification key (ID_key) and an additional string (any string the user wants to include to differentiate among his/her devices, policy yet to be defined). The output of the hash function will be a hash value, the device_ID, identifying the specific device of that user. If the user has different devices the same ID_key is used but different strings will be used to generate the different device IDs.

9. Categorize resources

The resource categorization module includes a function collecting the information about storage, processing, memory, networking, platform, power source etc. This information will be sent to the leader at a later stage.

Device Discovery: Ready to be discovered

Once the configuration steps are successfully accomplished, the device is ready to be discovered as soon as it enters an mF2C area.

10. Send "Welcome" message

The leader periodically broadcasts "Welcome" messages within its range, in order to announce that mF2C is enabled in the area.

11. Ack including device_ID

Since the device (with the mF2C app) is passively listening to "Welcome" messages, it automatically detects these special messages when it reaches an mF2C area. So, it acknowledges receipt of these messages by sending an ACK including its device_ID and ID_key identifying both, device and user respectively.

12. Is known?

The leader checks the device_ID in its database. Two cases may occur:

Case1: No

13. Provide keys:

If the device_ID does not exist in the leader database, then the leader provides keys according to the device_ID (and any other attribute if needed). Same as step 2, the functionality responsible for keys provisioning is yet to be defined

Case2: Yes

14. Ack device is known

If the device_ID exists in the leader database, the device already has its keys.

15. Authentication

The leader and the device will be mutually authenticated and the device is authorized to join the leader's area.

16. Send resource information

After the successful authentication of a device, the resource categorization block will retrieve the current status of the device by updating the information collected in step 9. That means, remaining storage, power, processing capabilities, information about the current availability of Memory, Networking facilities etc. This information is forwarded to the leader.

17. Create/Update corresponding DB entry

According to whether the device is known or not, the leader will either create or update the DB entry corresponding to that device, using the information received at the previous step.

Now that the necessary preparation steps have been done and the device is associated with the leader, the device becomes an mF2C agent and is ready to receive execution assignments from the leader.

Post-discovery maintenance

18. Gather device resources

The leader gathers information about all its associated devices (i.e. mF2C agents in Layer 2 in the resource topology of Figure 1) and processes it to be then forwarded to the cloud agent. The processing may require aggregation or compressing strategies to reduce the amount of information to be forwarded, while guaranteeing accuracy enough.

19. Send resource information

Periodically the leader will send information about its "children" (associated devices) to the cloud agent. This will allow the cloud agent to acquire a global (aggregated/compressed or not) view of the different resources contributing to the system.

20. Keep alive

This Keep alive is periodically sent by the leader to check the status of the associated devices it is managing. If no response is received from a particular device, the leader assumes that it has left the area or that it is dead for some reason. The keep alive is different from the welcome messages in the sense that it is likely performed as a unicast and after the connection between the device and the leader has been established.

21. Ack including updated resource information, if any is received

The device acknowledges receipt of the keep alive. To that end, the device sends an ACK message. This message may be enriched with resource information updates.

22. Detect that no ACK is received

The leader detects that no ACK is received. There will be a policy including the number and timing of re-attempts.

23. Update database

Depending on result on steps 21 and 22, the database will be updated with the resource information when the ACK is received, or when it is not, marking the device as disconnected or finally removing the device from the database.

The leader may also include a policy looking for a finer device state description, for example defining states like "ok" (active) /"down" (if they have been gracefully shutdown, drained or retired)/ "unreachable" (if consecutive keep alive messages are not coming back, in which case the device will automatically be considered as down after X re-attempts)/ "intermittent" (if the keep alive messages are flapping, like if the device has a weak signal), etc.

Disconnection

24. Bye

The device explicitly sends to the leader a "Bye" message indicating its intention to leave the area.

25. Change device's status into "Disconnected"

The leader will update the database record for that device to indicate that it is "Disconnected".

26. Remove device's resource information from cache after timeout

For efficiency reasons and in order not to keep obsolete information in its cache, the leader will remove the device's resource information when a pre-configured timeout elapses (yet to be defined).

4. Service Management Design

As described in deliverable D2.6 the service management block in the agent controller is responsible for the orchestration of local services. Its main functionalities include:

- Categorization of services and management of categories.
- Mapping of tasks.
- Allocation of tasks
- QoS provisioning

4.1 Categorization

As described in the Resource Management block, the Agent Controller oversees filling the tables to be accessed by both, the Agent Controller and the Platform Manager. As for resources, a tentative classification of mF2C capable resources according to a categorization strategy is presented in Section 3.3. As for the mF2C services, we must remind that the request for executing a service is always reaching out to the Platform Manager of an Agent. Once the service request is received, the Platform manager will start a set of steps, basically consisting on decomposing the service on tasks, selecting the suitable agents where executing these tasks, etc., all deeply described in the deliverable D4.3, focusing on the Platform Manager. Besides that, the Platform Manager will also forward the service to the Agent Controller, specifically to the Service Categorization module, where the service will be classified according to a defined categorization policy, as the one proposed in Figure 10 (a different representation of Service Categorization, as a class diagram is shown in Annex I). Afterwards, the information about the service and its characteristics will be added to the Service Database. Similar to the Resources, the Agent Controller is in charge of filling the databases to be consulted by both the Agent Controller and the Platform Manager.

The service categorization process included in the mF2C system, is considered critical to efficiently and successfully execute services. Indeed, the service categorization process 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. A preliminary approach to categorize the services in a mF2C system is based on the following aspects:

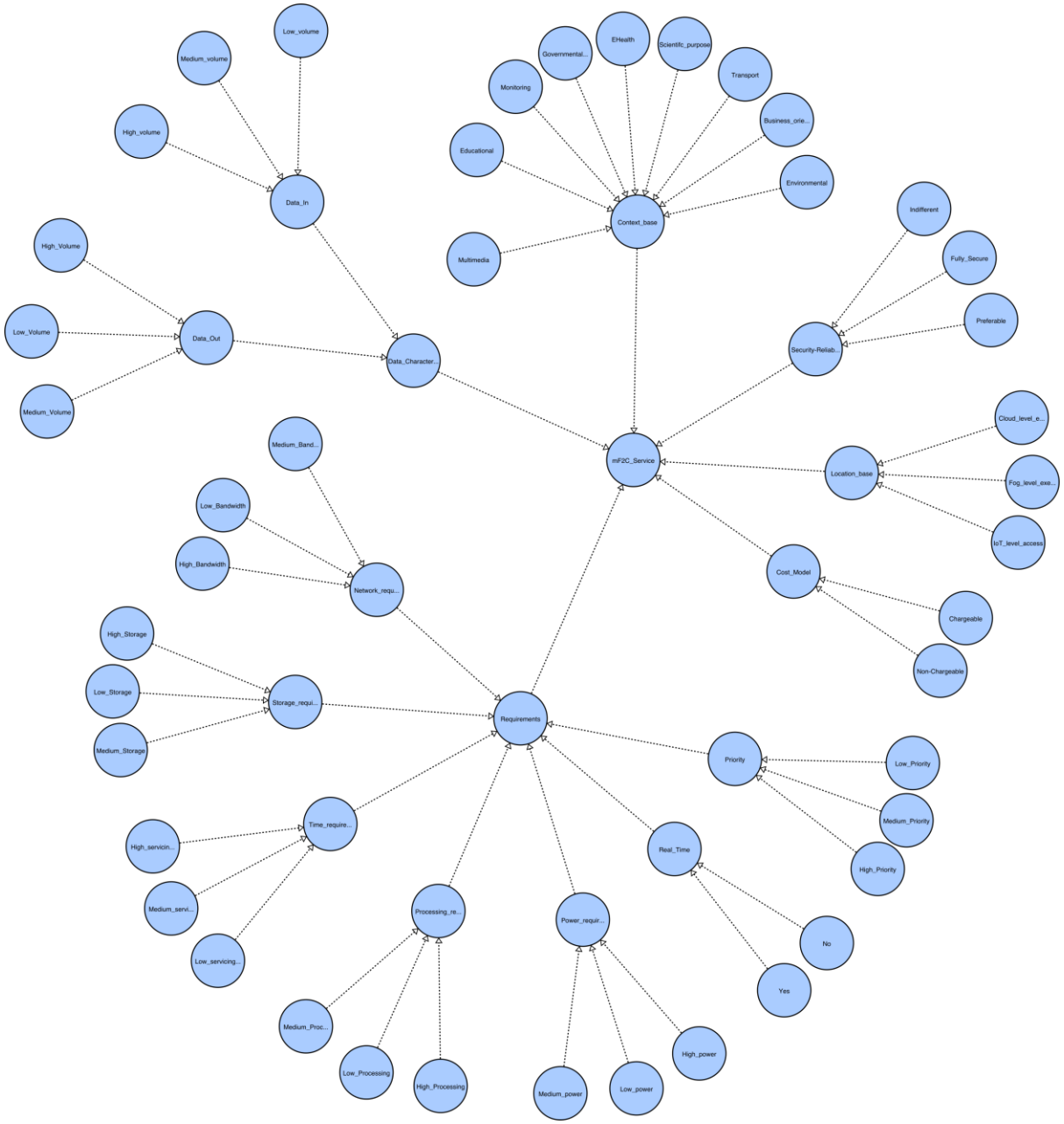

Figure 10. Service categorization

- Context Based: Services can be classified according to the Context scenario as: Multimedia, Educational, Monitoring, Health, Governmental, Transport, Scientific Purpose, Business Oriented, Environmental, etc.
    - Location Based: Based on the location of the service execution and the IoT's data source, we can categorize the services as:
    - Cloud Level: The service execution has been performed on the Cloud side agent.
    - Fog Level: The service has been executed on the Fog agent.
    - IoT Level: Real time applications require data collected from IoT devices.
- Cost Model Based: Services might be free of cost or chargeable. So, based on the Cost Model services can be further classified into two parts:
    - Non-Chargeable: Free of Cost
    - Chargeable: Payment required for using the service or to execute the application.

- Security & Reliability Based: Services may be classified as:
  - Fully Secure: Services requiring "fully" network, data and device security as well as reliability guarantees.
  - Preferable: Services requiring limited security and reliability.
  - Indifferent: Service not requiring security and reliability
- Data Characteristics Based: Related to services requiring data to be executed and producing data once executed. Hence, based on the data input and output for the service, services may be categorized as:
  - Data In: Based on the input data, services can be categorized into three types: i) High Volume, ii) Medium Volume or iii) Low Volume
  - Data Out: Based on the output data services can be categorized into three types: i) High Volume, ii) Medium Volume or iii) Low Volume
- Service Requirements: Considering hardware, network and time requirements, services can be classified according to:
  - Network Requirements: Requiring i) High or ii) Medium or iii) Low Bandwidth
  - Storage Requirements: Services may need: i) High or ii) Medium or iii) Low Storage.
  - Processing Requirements: Requiring i) High or ii) Medium or iii) Low Processing capacities.
  - Power Requirements: Requiring i) High or ii) Medium or iii) Low volume of power.
  - Time Requirements: It may be i) Long ii) Moderate or iii) Short duration time required to execute the service.
  - Real Time: Service demanding any real time constraint: YES/NO
  - Priority: Based on the priority, a service can be categorized into three classes: i) High or ii) Medium or iii) Low Priority Service.

Besides including the information above, an additional, deeper level of granularity is envisioned. Recognized the fact that agents will execute tasks (low level execution), a classification may also be added at task level, to specify tasks requirements and characteristics. Consequently, we propose the Service Database to include information not only related to the service but also to the tasks building the service, as follows:

- ComputingUnits: Required number of computing units. This is used when there is no need to specify constraints on the type of processors and number of cores.
- ProcessorName: Required processor name
- ProcessorSpeed: Required processor speed
- ProcessorArchitecture: Required processor architecture
- ProcessorType: Required processor type
- ProcessorPropertyName: Required processor property
- ProcessorPropertyValue: Required processor property value
- ProcessorInternalMemorySize: Required internal device memory
- processors: This is used to specify multiconstraints on the processor. A task, for example, can be executed on a CPU with 4 cores (computingUnits) or a GPU with 16 cores.
  - computingUnits: Required number of computing units
  - name: Required processor name
  - speed: Required processor speed
  - architecture: Required processor architecture
  - type: Required processor type
  - propertyName: Required processor property

    - propertyValue: Required processor property value
    - internalMemorySize: Required internal device memory
- MemorySize: Required memory size in GBs
- MemoryType: Required memory type (SRAM, DRAM, etc.)
- StorageSize: Required storage size in GBs
- StorageType: Required storage type (HDD, SSD, etc.)
- OperatingSystemType: Required operating system type (Windows, MacOS, Linux, etc.)
- OperatingSystemDistribution: Required operating system distribution (XP, Sierra, openSUSE, etc.)
- OperatingSystemVersion: Required operating system version
- WallClockLimit: Maximum wall clock time
- HostQueues: Required queues
- AppSoftware: Required applications that must be available within the remote node for the task

4.2 Mapping

The mF2C control architecture leverages a distributed and hierarchical control topology intended to map service requests into the most suitable resources for a successful service execution. The service requests can be decomposed into tasks in the Task Management block of the Platform Manager, and the Task Scheduler block decides where each individual task will be executed.

For example, let's assume that the service execution request reaching out the Platform Manager of the agent serving as leader in area 1, is decomposed into four tasks. A potential decision about the mapping procedure is shown in Figure 11 and may turn into:

- Task 1 (T1) requires data only available in area 2 of the network. This means that the request to execute T1 must be forwarded to the agent in the cloud (according to the mF2C hierarchical architecture), to be later forwarded towards the agent in area 2, responsible for the mapping strategy in that area.
- Task 2 (T2) requires resources available in the local resources in area 1, so the request to execute T2 is forwarded in this agent.
- Task 3 (T3) requires resources available in one of the agents in area 1, so the request to execute T3 is forwarded to this agent.
- Task 4 (T4) imposes high demanding resources capacities, so the request to execute T4 is forwarded to the agent in the cloud.

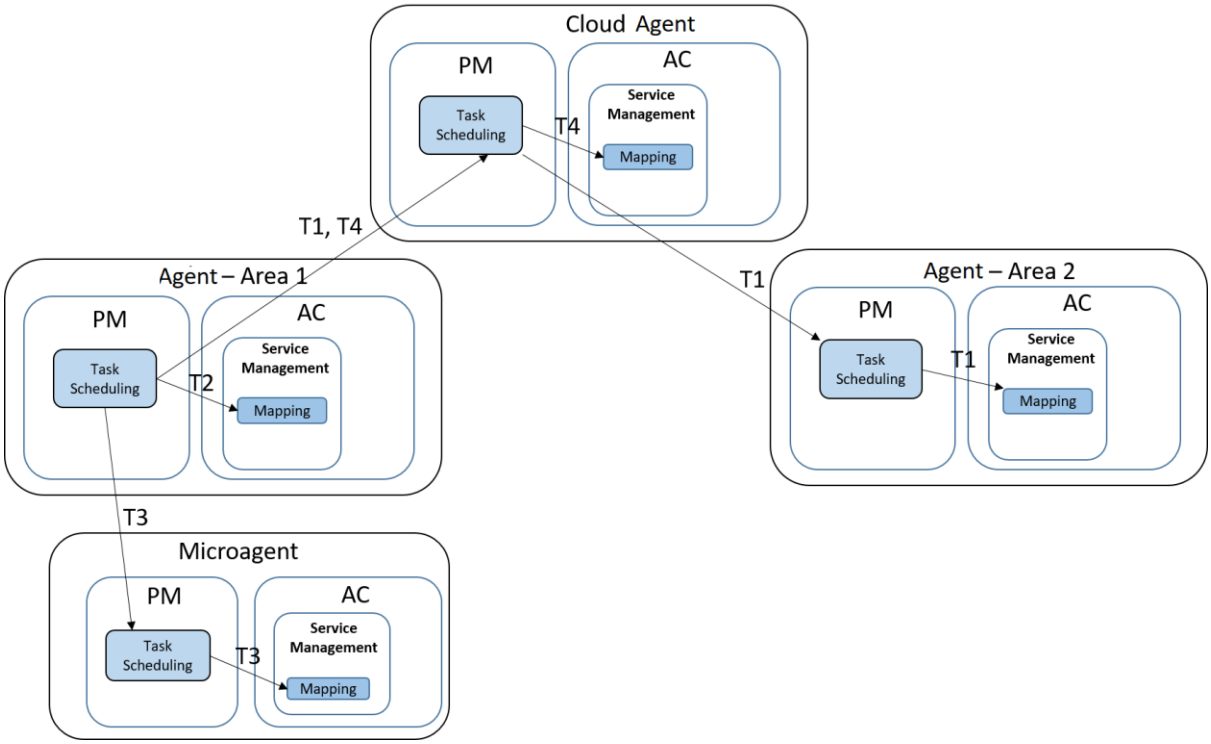

Figure 11. Mapping of different service tasks

The process of mapping each of the mentioned tasks can be presented in the following steps, see Figure 12:

Step 1. After receiving the task request from the Platform Manager, the Agent Controller forwards it to the Mapping block.

Steps 2 and 3. The Mapping block will then look at the database to check if the requested task already exists, thus getting a positive or negative answer. If the task exists it means that this task or method has been already executed in this device and it has information stored about it.

Step 4. If the answer is negative, the Mapping block will ask the Categorization block in the Service Management, for classifying the received task according to its characteristics. This block will work in tight cooperation with the Categorization block in the Resource Management. If the answer is positive this step is skipped.

In principle, from this step the mapping would consist in selecting the resources matching the requested task. However, when there is no possibility for choosing resources in this specific device, there is no need for running again a matching between the requested task and the local resources – the Platform Manager has already done this matching when selecting this agent controller for execution. In this case, the Mapping functionality finishes in step 4.

On the other hand, when distinct resources may be selected in the same device, it is necessary an additional matching step in this mapping module. 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. By an mF2c non capable device we mean devices with computing capacity but without capacity to have the mF2C agent installed. In that case, the AC is in charge of mapping some of the tasks to these small devices. This matching will consist on steps 5, 6, 7 8 and 9 which are described in the next paragraphs.

Step 5. The next step is intended to contact the block Policies in the Resource Management to see which rules should be applied to match the task requirements.

Step 6. Moreover, for each task request, the Mapping block must cooperate with the Profiling block in the User Management in order to find out if a user constraint for the expected QoS is in place. If such constraint exists, it must be implemented.

Steps 7 and 8. The Mapping block requests the database for information about local resources to select the appropriate resources.

Steps 9: And finally, the selected resources are allocated. It is worth mentioning that this allocation module not only runs during this mapping process in the runtime execution phase, which are the steps described in this section; but also during the deployment phase when the lifecycle manager (in PM) deploys the selected resources.

This entire process is shown in Figure 12.

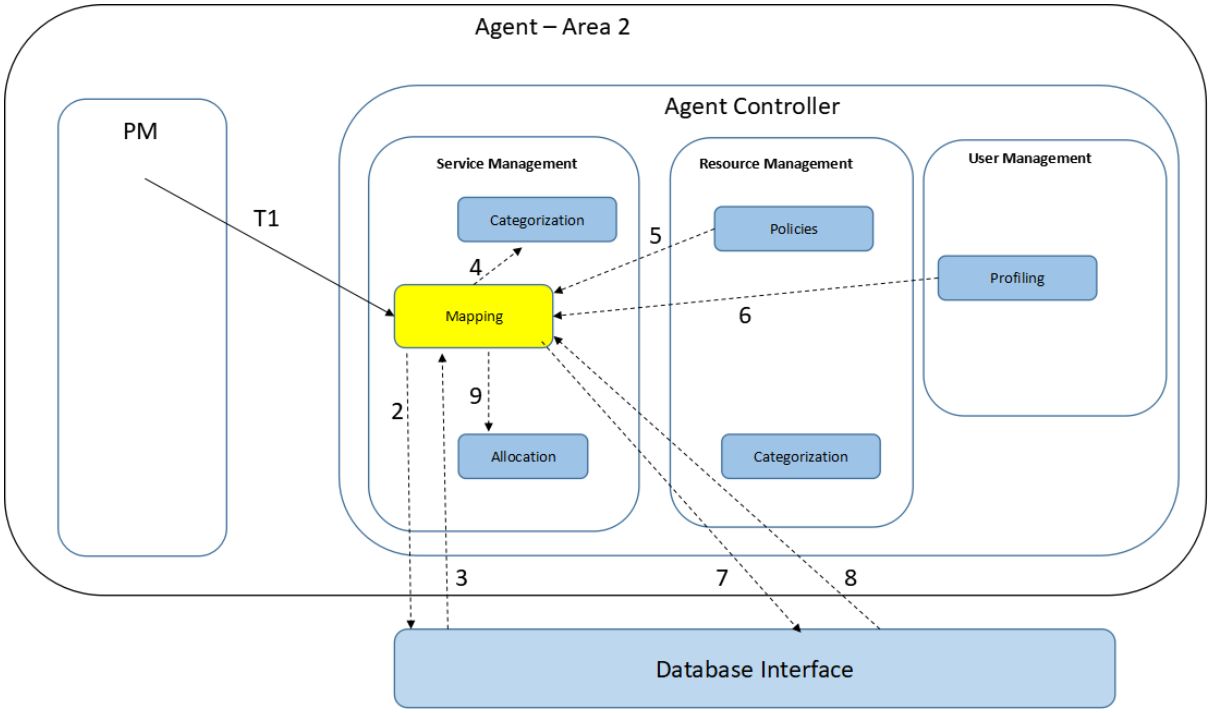

Figure 12. Mapping block in coordination with other blocks in Agent Controller

4.3 Allocation

This module 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.

This module can be call during the deployment phase by the Lifecycle model in order of reserve selected resources, as it is described in D4.3. As well as during the mapping process, in the runtime execution phase. In this last case, it is called once the database is read (thus providing local resources availability) and the mapping request for the available resources has been determined. It inherits general categorization in step 4, policies verification in step 5, and user profiling requirements and constraints in step 6.

Previous steps should guarantee to cover Priority, Time related and Critical (low latency) requirements.

The Allocation block assigns and allocates the appropriate computing (CPU, RAM), storage (disk), networking (bandwidth), and energy (battery slots) resources to be able to run the services in local resources. Finally, the Agent Controller will monitor the execution of the services checking the fulfilment of QoS attributes.

4.4 QoS Provisioning

The service management block also needs to check and guarantee that the service meets the requirements specified by the user and the resource management module. This block 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.

The QoS parameters (aligned to successfully meet the SLOs defined for a particular SLA) to be considered are not limited to the parameters used in traditional networking. Indeed, in addition to latency, jitter, bandwidth, among others, various features may also impact the observed QoS, such as:

- *Energy consumption*: high energy consumption may result in the rapid unavailability of resources dependent of battery.
- *Data quality:* quality of data provided in a response to the service request.
- *Bandwidth, Capacity and Throughput*: indicate the capacity of data which can be sent over a link within a given time.
- *Reliability:* measures the ability of the system or its individual components to perform its required functionalities under stated conditions for a specified period of time.

These are only some of the parameters that may be used for QoS measuring, and thus the list can be extended to add any additional attribute to meet future services demands. However, it is worth mentioning that QoS requirements must be service-aware, in other words, for different service tasks distinct sets of parameters may be used. For the first iteration, the QoS deployment will focus on the service execution time as the main critical parameter.

5. User Management Design

The User Management module is part of the Agent Controller component and it is responsible for managing the profiling and the sharing model properties of the users who have access to the mF2C system and the applications running on top of it. This module is also responsible for enforcing the QoS related to these properties.

This module is composed by three subcomponents (***Profiling***, ***QoS Enforcement*** and ***Sharing model***) that will interact with other Agent Controller and Platform Manager modules. On one hand, it will interact with other Agent Controller subcomponents in order to set and get the profiling and the sharing model properties. And on the other hand, if some of these properties or constraints are not being fulfilled, then this module will call the Platform Manager component in order to take the required actions.

Baseline Technologies

The user interface must be bespoke and then it needs to be designed and developed from scratch being a part of the Agent Controller component.

Internal Architecture

Next picture depicts the User Management's subcomponents and their interactions with other mF2C components, see Figure 13.

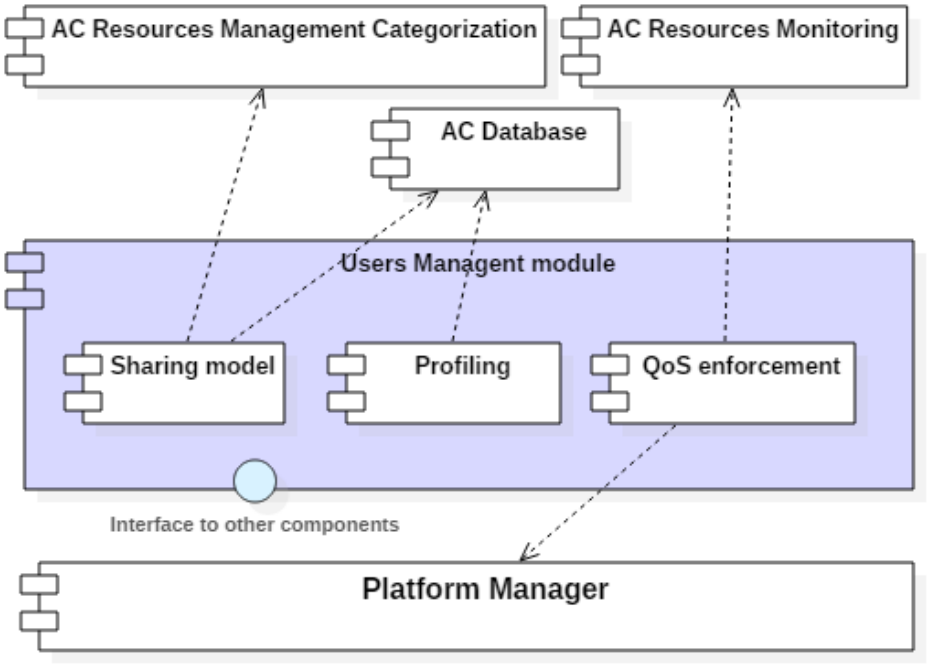

Figure 13. User Management's subcomponents

5.1 Profiling

A user profile is the collection of personal data related with a specific user (be it both, the user owner of the device and the mF2C consumer client executing a service) and digital representation of person's identity. It can be also considered as a logical representation of a user model. The user profile information can be extended or refined with user's preferences and behaviour.

This subcomponent is responsible for the setup and management of such user's profiles with roles and permissions. The functionalities included in this module are the following:

- Set / store the profiling properties
- Get / read these properties

The properties related to these roles and permissions/authorizations managed by this module include the following:

- Access control list (ACL), including both
  - Services allowed to be used by the user
  - Services allowed to run in their devices
- Max. number of services allowed to run in their devices
- How to join the mF2C system:
- As a consumer client, as a contributor, or both
- Use of GPS or any other device to determine the location or context
- Use key functions related to data encryption to preserve the identity and secure communications of the user
- Security levels defined for different data flows related to the user
- Ability for data to be shared
- Users group definitions
- Resource parameters related to QoS conditions, to be guaranteed in case of executing services, both hosted or offloaded

All these properties should guarantee the modelling of roles, permissions and authorities for the fulfilment of user's needs.

When the user downloads and installs the mF2C components in the device, all these profiling properties are configured with default values, as we can see in the sequence diagram in Figure 14.

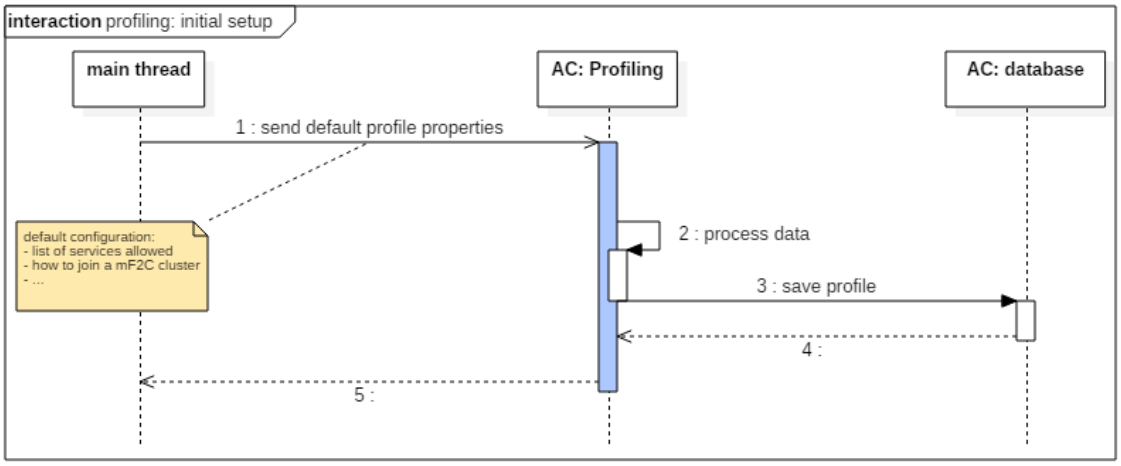

Figure 14. Profile properties configuration with default values

Later on, the user will be able to modify them, Figure 15.

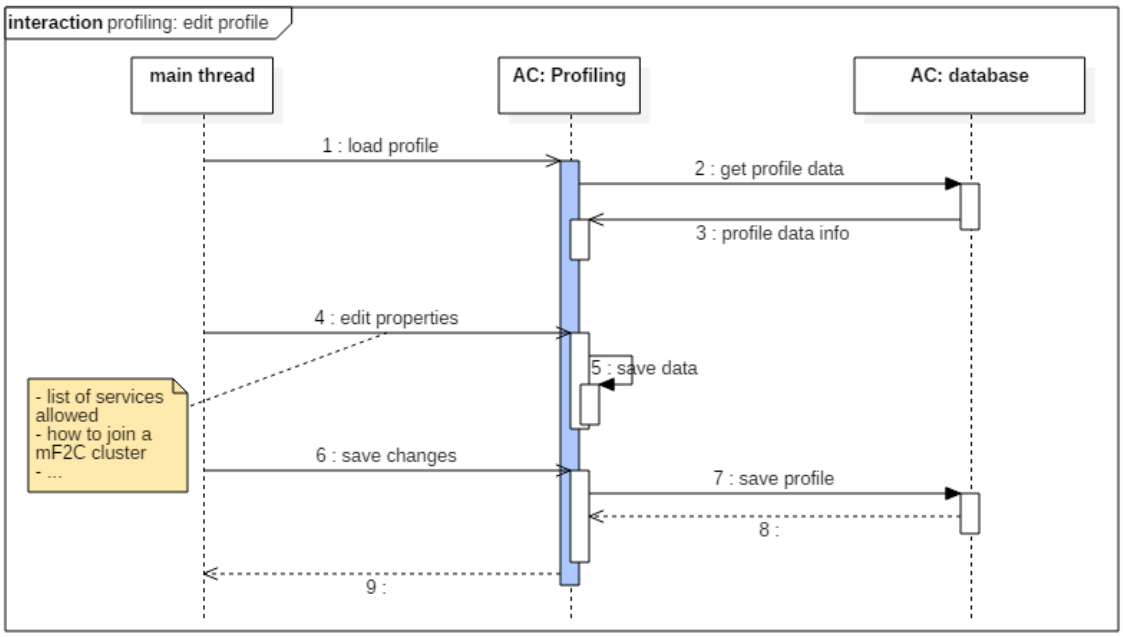

Figure 15. Updating profile properties

When setting the values of these profiling properties, this subcomponent will only interact with the Agent Controller database. Then it will also offer an interface to the other components in order to get / read the information about the user's profiling.

5.2 Sharing model

The sharing model subcomponent is responsible for the definition of the device's shareable resources. The functionalities included in this module are the following:

- Definition of the resources that will be available to the mF2C applications
- Definition of rules or constraints in order to establish not only the amount of resources to be shared, but also the conditions from which these resources should be increased, decreased or not shared at all
- Definition of reward mechanisms for these resource contributions, like some kind of service execution credits, economic rewards, etc.
- Get / read these defined shareable resources

The shareable resources managed by this module include the following ones:

- Max. CPU percentage usage
- Max. RAM usage
- Max. disk usage
- Max. amount of bandwidth usage
- Battery usage limits
- Local Data and/or file systems that could be shared with specified users or groups
- Other not yet envisioned

In the same way, as with the profiling module, when the user downloads and installs the mF2C components in the device, all these shareable resources are configured with default values, as we is shown in the following sequence diagram, Figure 16.

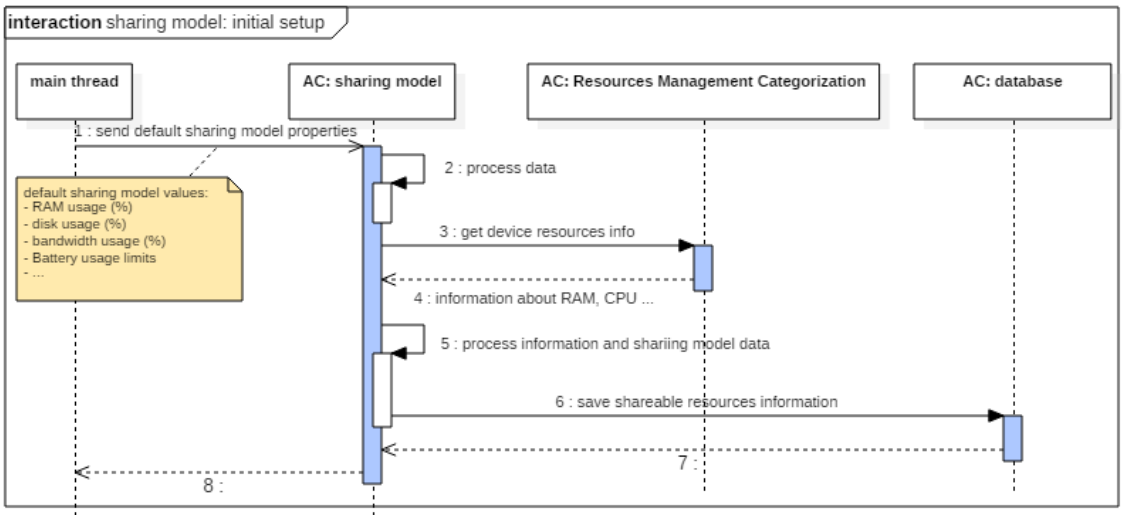

Figure 16. Configuration of shareable resources when installing mF2C software

The sharing model subcomponent will interact with the following Agent Controller modules:

- It will make use of the Agent Controller **Database**, in order to write and read the values of the defined sharing model properties.
- Then, it will also call the **Resources Management - Categorization** module interface, in order to get the information about the current device's resources.

Finally, this subcomponent will also offer an interface to the other Agent Controller components in order to get the information about these shareable resources defined by the user.

5.3 QoS Enforcement

The QoS Enforcement module is responsible for enforcing that the mF2C services running in the device meet the constraints defined by the user. The functionalities included in this module are the following:

- Check if the profiling properties are met
- Check if the shareable resources constraints are met
- Send a warning to the Platform Manager if some constraint is violated

As it is shown in the following sequence diagram, Figure 17, this module consists, basically, in a loop that is continuously checking that the mF2C applications running in the device meet the user's constraints.

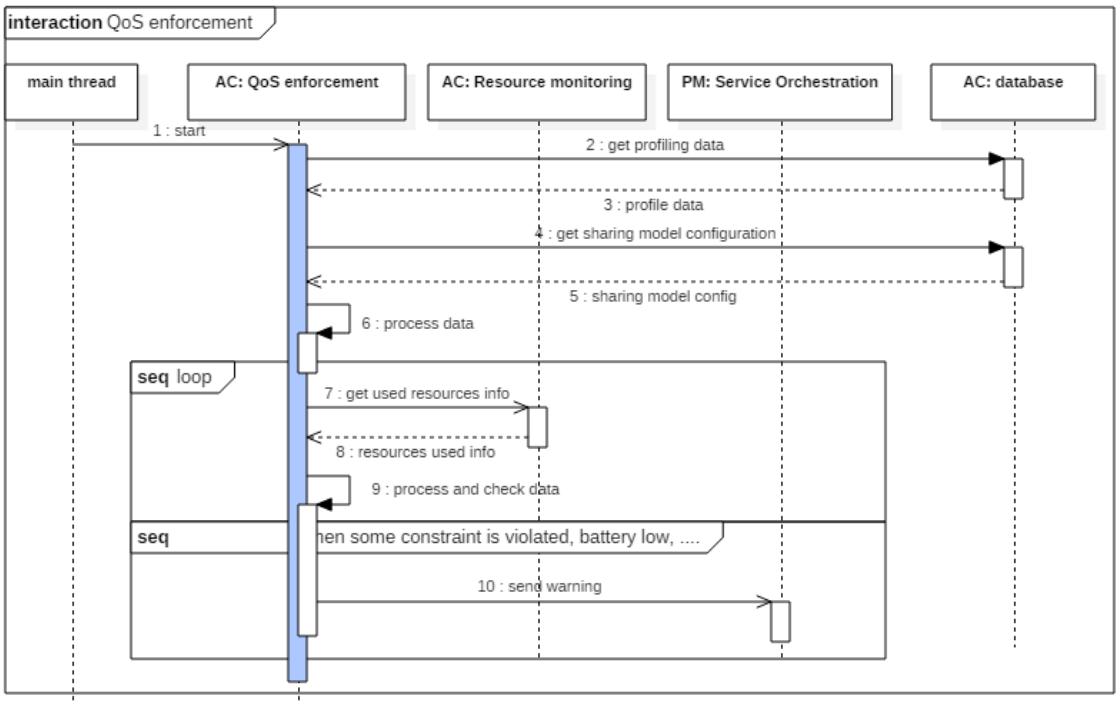

Figure 17. QoS enforcement working

This subcomponent will interact with the following mF2C components:

- It will make use of the Agent Controller Database, in order to get all the information about the user's profiling and sharing model.
- Then, it will also call the Resources Management - Monitoring module interface, in order to get the information about the resources used by the mF2C applications.
- Finally, if the applications running in the device do not meet the constraints defined by the user, this subcomponent should send a warning to the Platform Manager - Service Orchestration module in order to take the required actions, like removing or halting the application.

6. Data Base Design

In order to perform the designed functionalities in mF2C, a database including information about the services, resources and users participating in the platform is required. This database can be accessed both by the Agent Controller and by the Platform Manager, by means of a single interface that encapsulates the different kinds of data that the database contains, as seen in Figure 18.

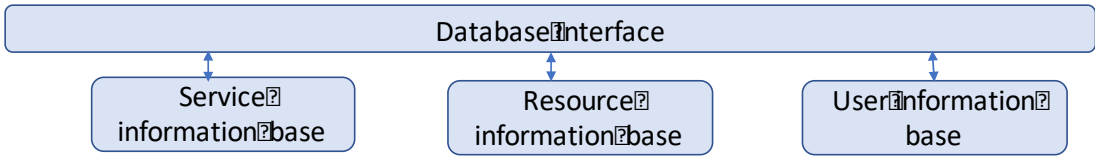

Figure 18. Conceptualisation of the database

Conceptually, the database must include three main kinds of information:

- Services, including their characteristics, requirements, and SLOs.
- Resources, including their characteristics and how they are interconnected in the infrastructure
- Users, including their relevant data and their preferences on how to participate in the infrastructure.

Since many of these user preferences depend on the resources that the user contributes to mF2C, and also on the services that are executed in the platform, there are strong relationships between the three kinds of information. Thus, we define a single schema that specifies these relationships and provides a global view of all the information that needs to be stored in mF2C. The fact that we define a global schema does not imply that we will have a single database that centralizes all the information. Instead, this schema and, more specifically, the database interface that encapsulates it, will provide access to the information distributed among the different resources in the platform, in such a way that any agent can access all the information it requires (about its own resources if it is a regular node, and about its own and its children's resources if it is a leader node).

6.1 Database schema

The UML class diagram in Figure 19 depicts the concepts, properties, and relationships between concepts that mF2C needs to store in order to perform its functions, as designed in D3.3 and D4.3, independent of the final technology used in its implementation.

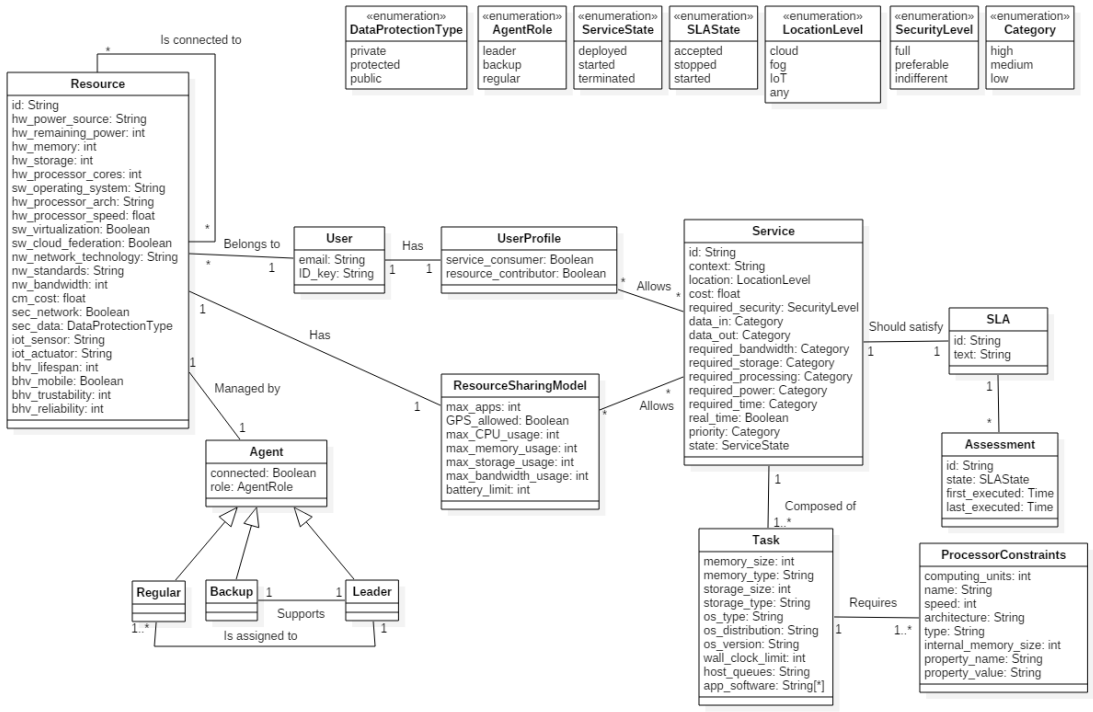

Figure 19. mF2C database schema

In the following we describe each of the entities included in the class diagram.

Resource

Represents a device that is part of the mF2C infrastructure. Each resource has an *id,* provided by the identification functionality, calculated from the ID_key of the user that owns the resource, represented by the association with the *User* entity.

The rest of properties of *Resource* correspond to the characteristics defined in the resource ontology used for the categorization of resources.

The association *Is connected to* indicates a physical direct connection between resources.

Each resource has an associated *ResourceSharingModel*.

Agent

The concept *Agent* and its subclasses represent the role that a resource plays at a given time in the infrastructure, and how the clusters in mF2C are organized. If a resource is regular then it has a leader, and a leader has at least one regular resource assigned. Also, each leader has a backup.

User

Represents a registered user, who can own several devices (resources in mF2C). An *email* is required for registration, and each user is assigned a unique secret key represented by *ID_key*. It has an associated *UserProfile*.

UserProfile

Represents the user's roles and permissions, applicable to the user participation in mF2C as a whole. A user can participate as a service consumer, as a resource contributor, or both.

The relationship with *Service* represents the services or applications allowed by the user.

Those permissions that depend on a specific device are included in the ResourceSharingModel.

ResourceSharingModel

Represents the conditions on how the owner of the associated resource wants to contribute such resource to the infrastructure.
Although not included in the figure, other attributes may be considered, for example depending on the business model, turning into: list/rent, subscription, pay per use, etc.
The relationship with *Service* represents the services or applications allowed in the associated resource.

Service

Represents a service or application that is executed in the infrastructure. It has an identifier and a state, required by the lifecycle management functionality, which can be deployed, started, terminated. The rest of its properties correspond to the characteristics used for the service categorization functionality.
A service is decomposed in tasks, which are the execution units managed by the distributed execution runtime, and has an associated SLA, to be meet in terms of distinct SLOs, required for SLA management.

Task

Represents an atomic part of a service that can be scheduled for execution. A task has a set of constraints or requirements for its execution, used by the task scheduling functionality. The associated entity ***ProcessorConstraints*** represents the set of constraints that refer to the processor characteristics. In case the task requires several processors, then it can have different processor constraints for each of them.

SLA

Represents the service level agreement associated to a service. It has an identifier and includes the content of the agreement, expressed in the JSON standard. The **Assessment** entity keeps track of the evaluations of an SLA and their state. Violation rules should be defined for each individual SLO.

6.2 Database Interface

In the following we define the core set of operations that the database interface needs to offer in order to access the data required by the different functionalities, according to their current design. This interface can be extended with additional operations required during the implementation.

new_resource(resourceID: String, hw_power_source: String, …)

Creates a new resource, with the id calculated by the identification functionality, and the values for its characteristics assigned by the categorization. It also creates its associated *Agent* instance.

get_resource_characteristics(resourceID: String): ResourceInfo

Returns the subset of characteristics of a resource needed to calculate the role of a node as specified by the policies, grouped in the ResourceInfo structure.

set_agent_role(resourceID: String, role: AgentRole)

Sets the attribute *role* of the agent related to *resourceID* to the value of the parameter *role*, which may be Leader, Backup, or Regular.

get_distance_to_leader(resourceID: String): int

Returns the number of hops to the leader of the cluster, obtained through the association “Is connected to”, needed to apply the node selection policies.

disconnect_resource(resourceID: String)

Sets to false the value of *connected* of the associated agent.

create_user_profile(email: String)

Creates a new user identified by *email*, and assigns it a unique *ID_key.* It also creates its associated profile with default values as specified by the profiling functionality.

set_user_profiling_properties(userID: String, profile: ProfileInfo)

Sets the properties of the profile of the user identified by *userID* to the values specified in the *ProfileInfo* structure.

get_user_profiling_properties(userID: String): ProfileInfo

Returns the current value of the profile properties of the user identified by *userID*.

new_resource_sharing_model(resourceID: String)

Creates a resource sharing model instance associated to *resourceID* with default values, as specified by the sharing model functionality.

set_resource_sharing_model(resourceID: String, model: SharingInfo)

Sets the properties of the sharing model of the resource identified by *resourceID* to the values specified in the *SharingInfo* structure.

get_resource_sharing_model(resourceID: String): SharingInfo

Returns the current value of the sharing model properties of the resource identified by *resourceID.*

new_service(serviceID: String, context: String, …)

Creates a new service, with its assigned id, and the values for its characteristics assigned by the service categorization.

set_service_state(serviceID: String, state: ServiceState)

Sets the state of *serviceID* to the value of the *state* parameter.

get_agents_in_cluster(resourceID: String): Set(Agent)

Returns all the agents in the cluster where *resourceID* belongs.

create_SLA(SLAid: String, text: String, serviceID: String)

Creates an SLA with SLA*id* and *text*, associated to *serviceID.*

create_assessment(assessmentID: String, SLAid: String)

Creates a new *Assessment* with *assesementID* associated to *SLAid.*

set_assessment_state(assessmentID: String, state: SLAState)

Sets the *state* of *assessmentID* to the value of *state.*

7. Interfaces Design

Here there will be described the interfaces of the subsystems. UML was used to make easy to visualize them.

7.1 Agent Controller Interfaces

7.1.1 Diagrams centered in Agent Controller Package dependencies

Below, there will be explained the dependencies between the different packages operating with the Agent Controller and between the Agent Controller classes themselves.

To make it easier to read, it was split into 3 different diagrams Figure 20, Figure 21 and Figure 22.

Please, note that the different colours of the dependency lines have no special meaning, used only with the purpose of making easier to understand the diagram and do not get confused with the multiple lines.

In the same way, Classes that did not have dependencies were eliminated to simplify the view.

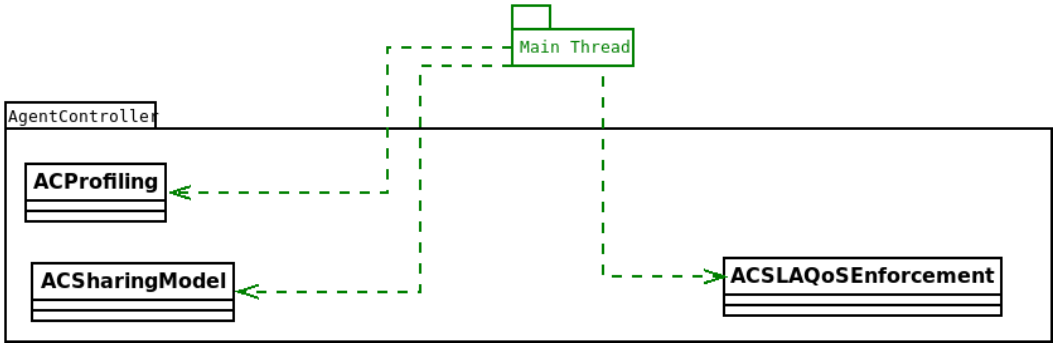

Figure 20. Agent Controller and Main Thread package diagram with dependencies

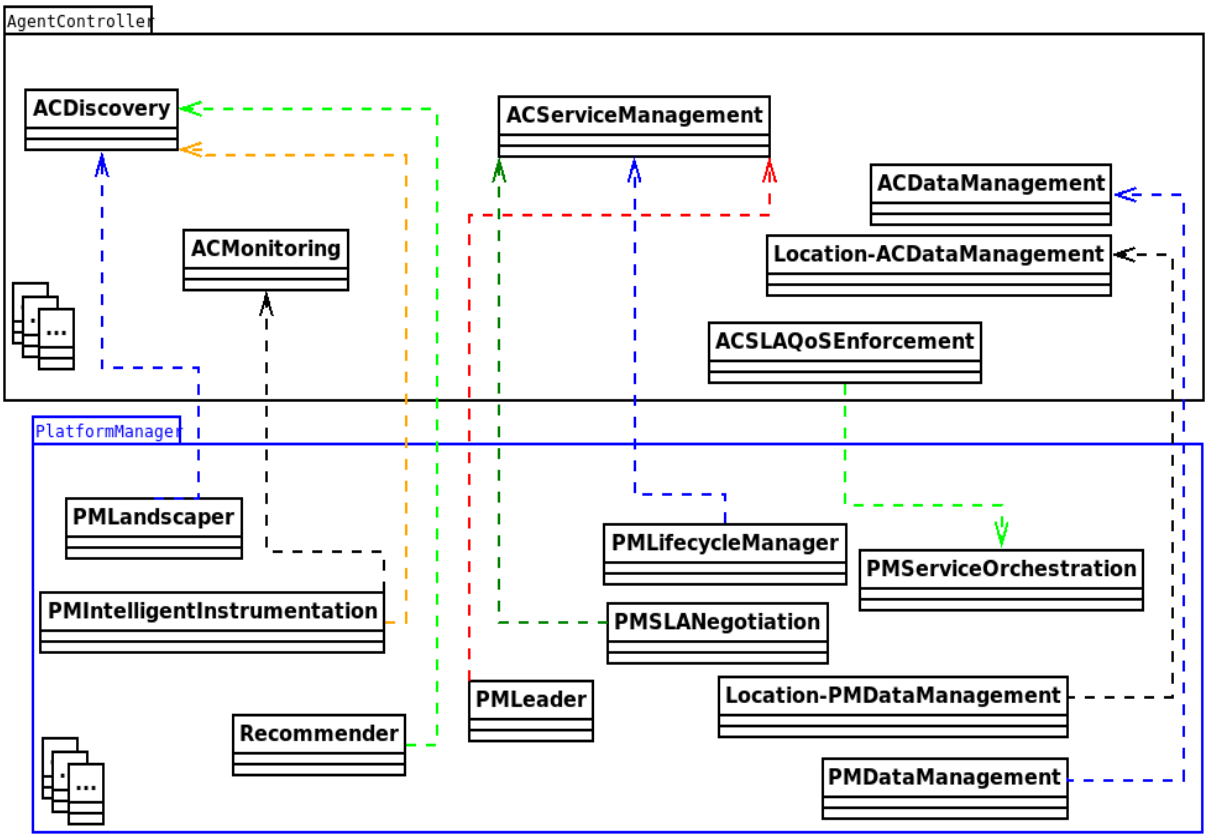

Figure 21. Agent Controller and Platform Manager package diagram with dependencies

In this last diagram, Figure 22 Agent Controller and Platform Manager package diagram with dependencies, you can observe that there are much more classes than in the simplified previous ones. This is because all classes of the Agent Controller package were included, even the ones which are not providing services to other packages objects or other local ones than themselves.

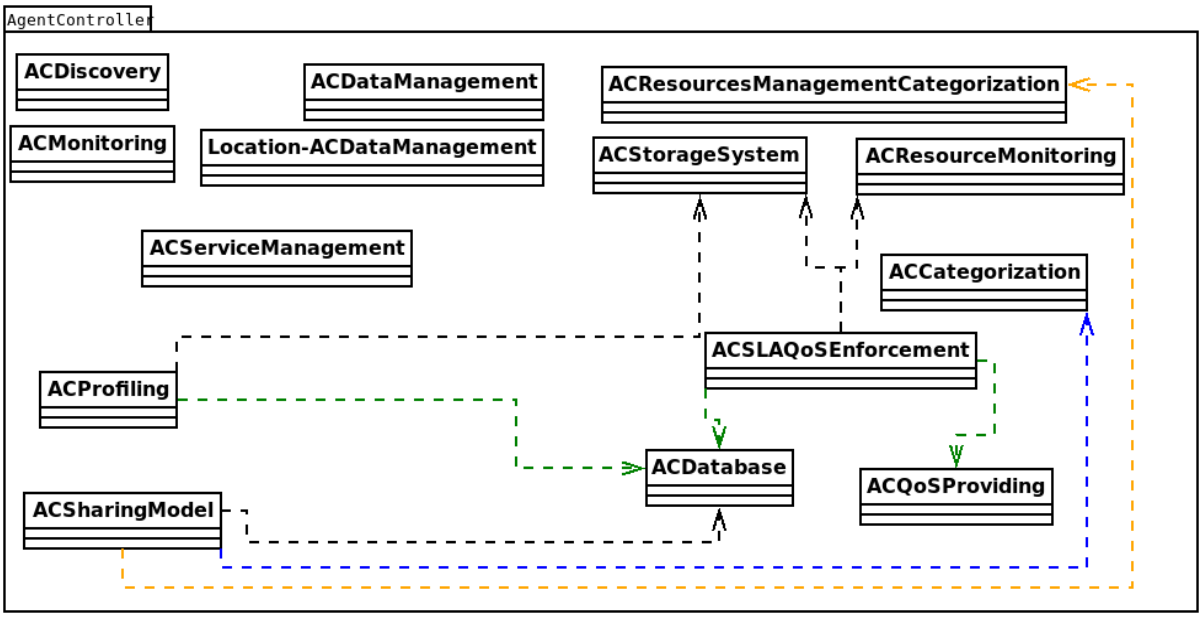

Figure 22. Abstract Class Diagram at packet level, centred in Agent Controller dependencies

7.1.2. Agent Controller's Class Diagram

This Class Diagram, Figure 23, is not included in the previously described dependencies due to the complexity to visualize the document with all those dependencies. The diagram below complements previous package one and details the methods to implement.

Some methods that apparently (for now), just provide services to their own class, were marked as private (-), the others were marked as public (+), defining the class interfaces.

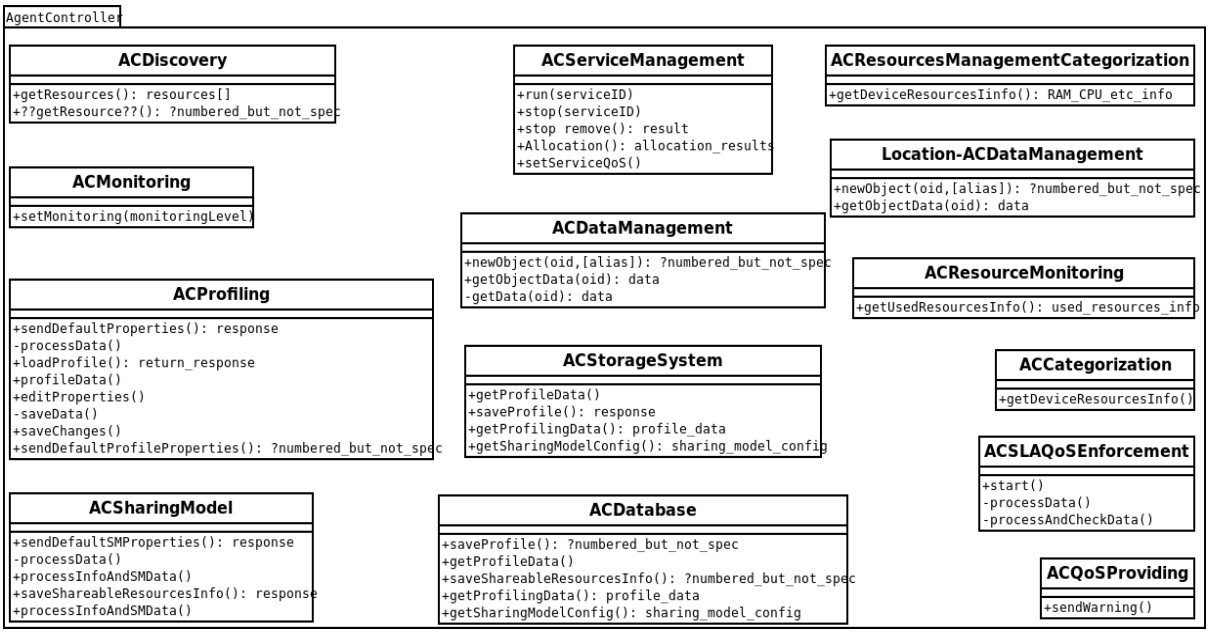

Figure 23. Agent controller's Class Diagram

8. Illustrative Example

A number of prototypes have been built to illustrate some of the functionalities presented in the "Core Resource Management Operations" workflow (Figure 9), such as the registration, device identification and the discovery process.

8.1 Identification

According with the section 3.2 the identification process can be divided in two logical steps: registration and device_ID calculation, which correspond to the steps 6, 7 and 8 of workflow of Figure 9. In the next sub-sections both processes are explained and preliminary frontends are presented.

8.1.1 Registration

The registration is the process by which the users will obtain the ID_Key. This key will serve two purposes: (1) identify uniquely users registered in the mF2C system and (2) will be a required input to calculate the IDs of all the user's devices, thus, registration must to be done only once per user, institution or entity.

The registration procedure will start when the user type his/her email address and press the button "register" in some kind of web page enabled for that purpose similar to the one shown in Figure 24.

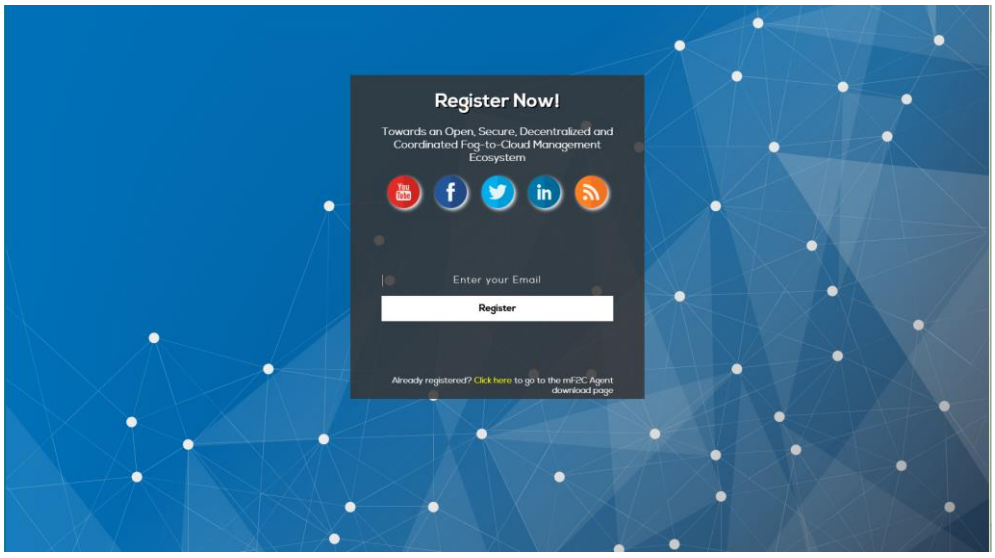

Figure 24. Registration frontend

After entering the email address, a process to generate the ID_key should start. This process will leverage some strategy particularly designed to the mF2C scenario, at this stage yet being an ongoing work. As a preliminary approach for in-lab testing the project proposes the following strategy:

- In the first step a script will validate that the user input meets the email address format.
- In a second step, unnecessary characters (like spaces, for example at the beginning and the end of the input) are removed.
- In a third step, the script will also put the whole string in capital letters and will separate the username from the email address.
- Fourth step: The username extracted from the email address is manipulated in order of getting from it four characters. In all the cases the extracted characters are the first, the last and the two characters in the middle. In the case of usernames which size is an odd number the three characters in the middle will be extracted and the character in the middle of these

three is ignored. The concatenation of these four extracted characters is called auxiliary string.

- Fifth step: Once the auxiliary string has been extracted from the user's email it will be concatenated with the whole user email. The result of this concatenation is the input to a hash function that uses the SHA512 algorithm. Finally, the hash output will be the user ID_Key.

The same script will verify if the calculated ID_Key belongs to an already registered user. If that is the case the script will update the database as a ID_Key recovery and will send the ID_Key as a file to the user email, otherwise a new record will be created in the database before the ID_Key is sent to the user.

The registration procedure concludes when the ID_Key has been calculated and sent to the user.

8.1.2 Device ID calculation

Once the user has registered in the system and has his/her ID_Key he/she can start calculating unique IDs for all his/her devices. The device ID (device_ID) will be a requirement that all the devices must meet before joining the mF2C system.

The process of computing each device ID is very similar to the activation of any software (Figure 25). The user will be asked to:

- type his/her email address (the same provided during the registration)
- to provide the ID_Key (received by email in this implemented prototype)
- to provide a unique string for every one of his/her devices (generated randomly in this prototype).

Before the ID assignation in the user device, an offline validation is performed to check if the email address is associated with the loaded ID_Key. If the ID_Key pass this validation then the device_ID can be generated. For this purpose, the SHA512 algorithm is again used. The input of the hash function will be the ID_Key concatenated with the string provided by the user. In Figure 25 this string is calculated randomly. The output of this hash is the device_ID.

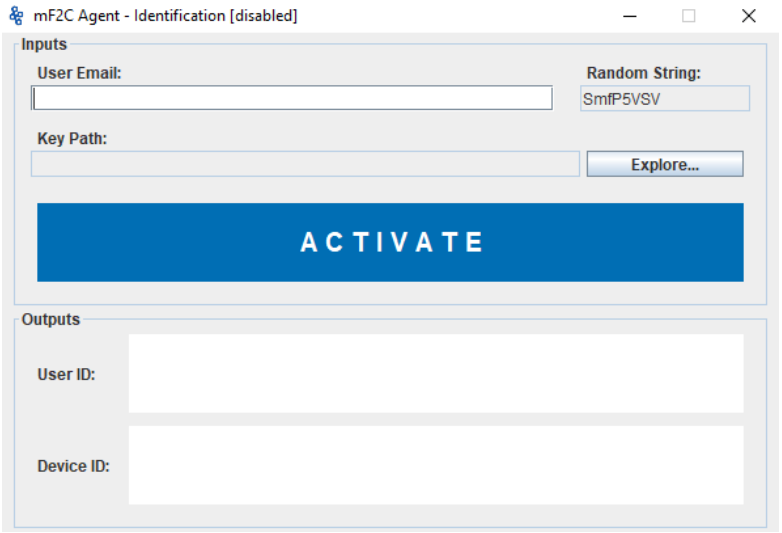

Figure 25. Device ID calculation frontend (before activation)

In the Figure 26, the fronted is presented once the Device ID (device_ID) has been calculated.

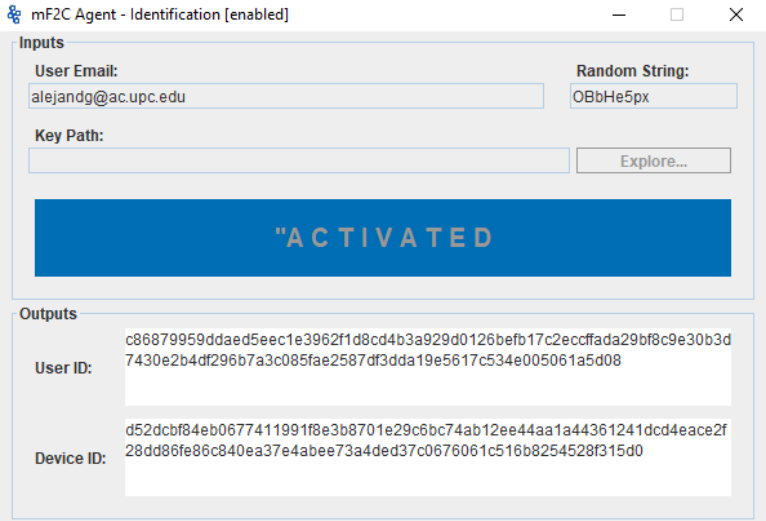

Figure 26. Device ID calculation frontend (after activation)

8.2 Discovery

In the following, we describe the current implementation of the discovery function detailed in section 3.1.1 and corresponding in the workflow of Figure 9 to the step 10. In fact, this function has been implemented using two Linux-based tools, namely *hostapd* [15] and *iw* [16]. While the former is an open source access point implementation offering a built-in option for appending information elements into beacons and probe response frames, the latter is a tool used for wireless device configuration allowing wireless scanning and retrieval of information (including vendor specific information elements) from scan results. Since both tools are run in “user space”, no kernel changes were needed to support the newly-added mF2C information. In addition, Python scripting has been used on top of these tools to encode and decode mF2C content at the leader side and the device side respectively.

In order to show that the mF2C beacon has been properly formed, we use *Wireshark* [17], a widely-known network protocol analyser tool. As it can be seen in Figure 27, frame number 664 (red box) represents an mF2C-enhanced beacon. In addition to the information elements normally contained in beacons, this frame also includes the mF2C vendor specific information element with the usual 221 tag number. It is recognized by the fact that it uses the mF2C OUI, which is followed by mF2C-specific data (highlighted in blue at the bottom of the figure).

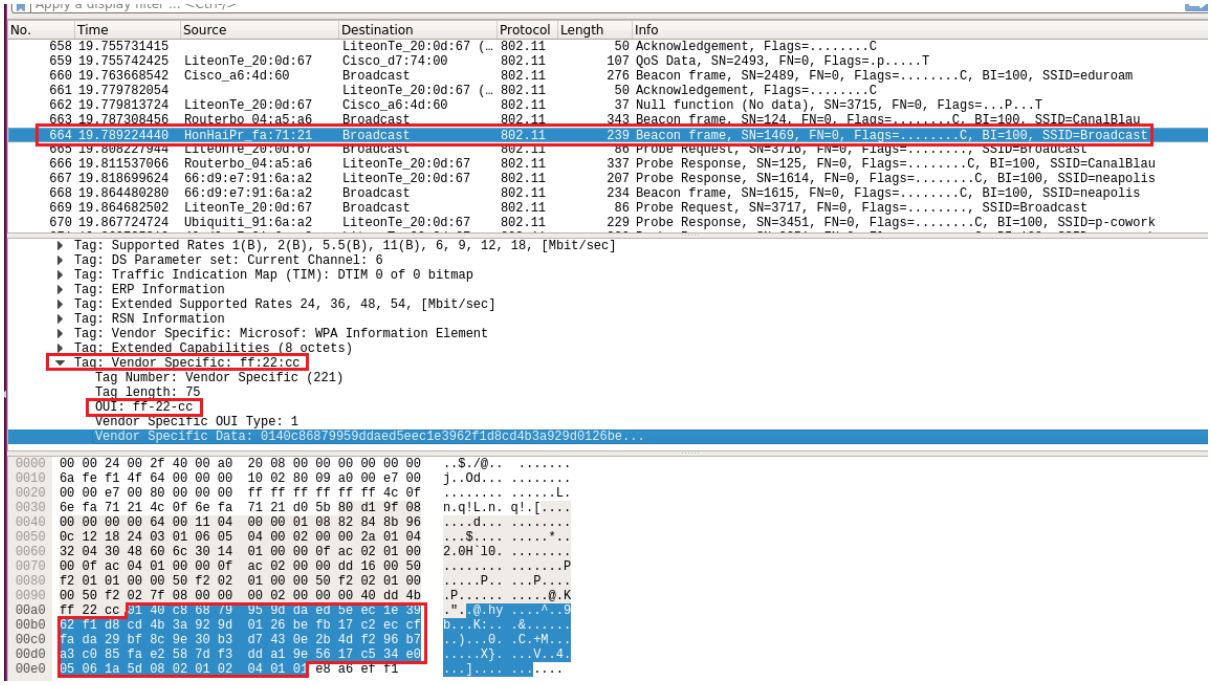

Figure 27. *Wireshark* Capture showing mF2C Beacon

In order to capture those beacons and to interpret the customized information they contain, a Python script is run on the receiver device side. Its main role is to parse the scan results returned by the *iw* utility looking for frames containing the mF2C OUI. If found, the subsequent payload is extracted and decoded into human-readable information that characterizes the leader. Figure 28 shows the decoding of an example payload corresponding to the previous beacon. It is comprised of the leader ID, the service type and the expected reward resulting from contributing resources to the system.

```
root@zeineb-X541UJ:/home/zeineb/mf2c-discovery# python3 mf2c_beacon_detector.py
Scan for mF2C beacons started...
mF2C Beacon detected!!!
+---+----------------------+----------------------+--------+
| # |  Leader ID(Trimmed)  |     Service Type     | Reward |
+---+----------------------+----------------------+--------+
| 1 | c86879959ddaed5eec1e | Smart Transportation |   1€   |
+---+----------------------+----------------------+--------+
root@zeineb-X541UJ:/home/zeineb/mf2c-discovery#
```

Figure 28. Beacon detection and content decoding

9. Conclusions

First of all, in this deliverable we review the main characteristics of the mF2C architecture proposed in deliverable D2.6 [1] and then we describe in detail all the main functionalities of the Agent Controller block. As stated in the mF2C architecture, the Agent Controller is one of the two blocks of the mF2C agent. One of the main characteristics of the proposed architecture leverages the fact that any kind of device with enough computing capacity could participate in the mF2C system as long as it has the mF2C agent software installed. Indeed, the mF2C agent will have all management and control functionalities to make a device become a participant in the mF2C system. Another important characteristic of the reviewed mF2C architecture is its organization into a hierarchical set of devices. Indeed, devices (mF2C agents) are clustered around one of them acting as the leader.

The set of management and control functionalities needed to become a participant in the mF2C system have been divided into two subsets of functionalities: Platform Manager (PM) functionalities, described in deliverable D4.3, and Agent Controller (AC) functionalities described in this deliverable.

In short, the PM provides high-level functionalities, including the intelligence to take decisions with a more global view. Whereas, the AC has a local scope, that is, taking decisions based on this local scope.

The subset of AC functionalities is divided into functionalities related to the resources, services and users management. Regarding the resources, the AC is responsible for managing local resources, considering as local resources the own resources when the agent is a usual mF2C agent; or its own resources and the resources of its clustered devices (children) when the agent is a leader. In this sense, the main functionalities of the Resource Management block have been described and tentative solutions for their implementation have been proposed. The list of the Resource Management functions is:

- Discovery
- Identification and Naming
- Categorization
- Policies
- Data Management
- Monitoring

In the case of the Services, the AC is only responsible for managing the services (or tasks) executed in its own local resources –that is, in the resources of the mF2C agent–, independently if it is a leader or not. If the device is a leader, the services executed in its children are managed by the PM. The functions of the Service Management are:

- Categorization
- Mapping
- Allocation
- QoS provisioning

Finally, the AC is also responsible for managing the users related to the device where the mF2C agent is installed. In the mF2C system, we consider that a user can be the owner of the devices, as well as the client (consumer) executing services in the mF2C system. The three functionalities of the AC related to the user management are:

- Profiling
- Sharing model
- QoS enforcement

This deliverable identifies all functionalities to be developed within the AC and provides different workflows showing the interactions of the different blocks for some basic operations. It also proposes a preliminary and basic design of the mF2C database, to be shared by the AC and the PM, as well as the UML diagrams of the interfaces between the different blocks of the AC.

Once the AC basic functionalities have been identified, the mF2C partners have proposed solutions for the implementation of some of them. The proposed solutions are a first approach towards the integration of the different blocks in the AC, as well as the integration between the AC and the PM.

Furthermore, some of these solutions have been preliminary implemented, for example the solution proposed for the Identification-Naming and Discovery functionalities.

It is important to highlight that all the proposed solutions, both at design level and at implementation level, are a first approach to the problem to be solved. Although there has been an intensive communication between partners working in deliverable D4.3 (devoted to PM) and partners working on this deliverable to avoid incoherencies, in the next steps of the mF2C project, which correspond to the AC blocks integration, we will refine the AC design in order of being completely compatible with PM designed in WP4.

Finally, the AC functionalities presented in this deliverable are a subset of the ones envisioned for the whole project. This set of functionalities corresponds to the first iteration of the project (IT-1) what intends to have a complete implementation of the mF2C system on month M18. After this first iteration, the entire set of functionalities of the different blocks of the mF2C agent will be improved, enriched and even increased.

Annex 1. Service Categorization

The diagram below shows the service categorization represented as a class diagram already shown in Figure 10.

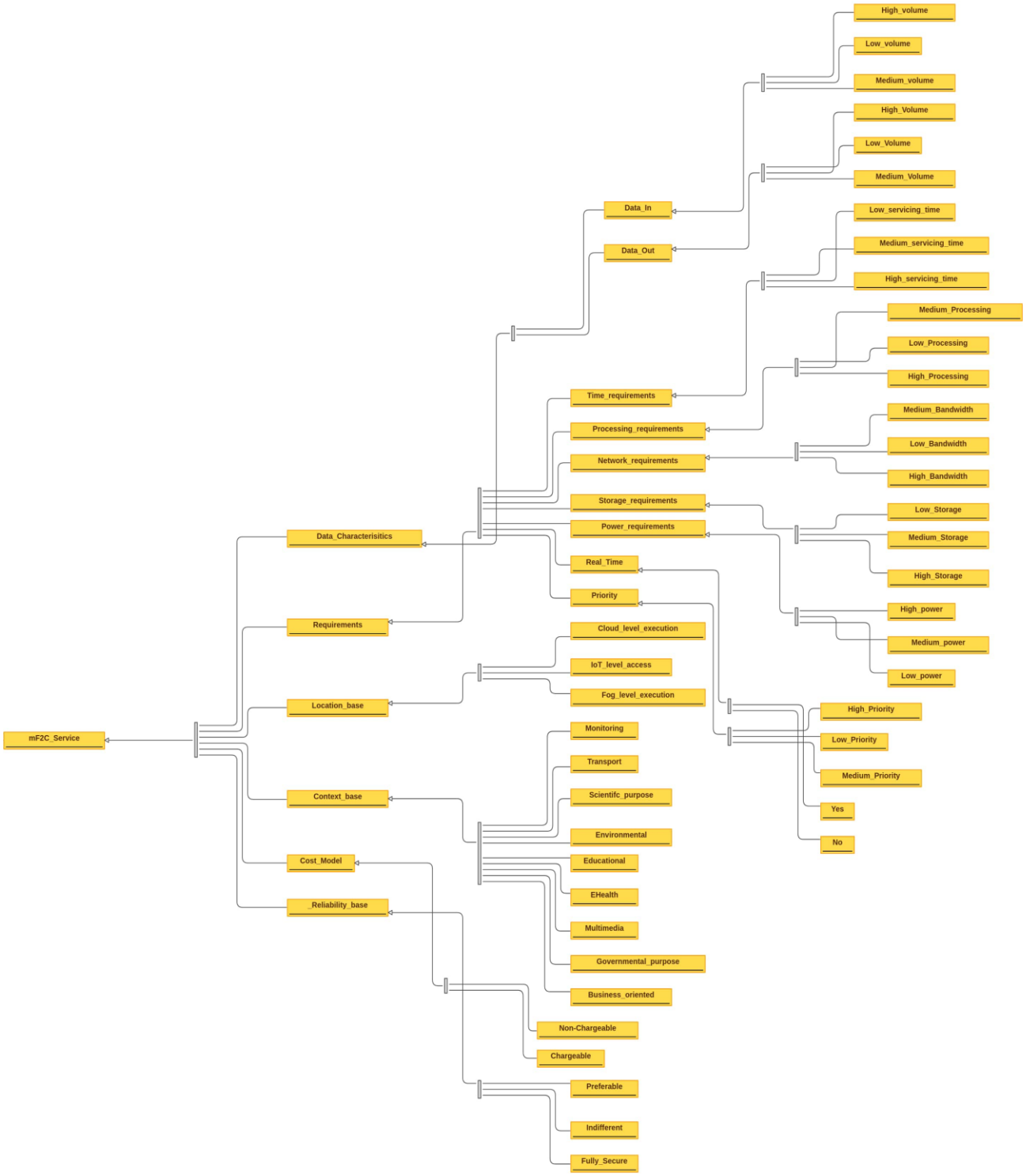

Figure 29. Service categorization

References

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

[2] «D2.4 Security/Privacy Requirements and Features,» [En línea]. Available: http://www.mf2c-project.eu/wp-content/uploads/2017/05/mF2C-D2.4-Security-Privacy-Requirements-and-Features-IT1.pdf.

[3] «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.

[4] «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.

[5] «Arduino,» [En línea]. Available: https://www.arduino.cc/.

[6] «RaspberryPi,» [En línea]. Available: https://www.raspberrypi.org/.

[7] «The JavaScript Object Notation (JSON) Data Interchange Format,» [En línea]. Available: https://tools.ietf.org/html/rfc7159.

[8] «JSON Web Signature (JWS),» [En línea]. Available: https://tools.ietf.org/html/rfc7515.

[9] «JSON Web Encryption (JWE),» [En línea]. Available: https://tools.ietf.org/html/rfc7516.

[10] J. P. L. R. a. A. W. R. Chandra, «Beacon-Stuffing : Wi-Fi Without Associations,» *Eighth IEEE Workshop on Mobile Computing Systems and Applications, 2007. HotMobile 2007. ,* p. 53–57, 2007.

[11] «IEEE Standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications».

[12] «Neighbor Discovery for IP version 6 (IPv6),» [En línea]. Available: https://tools.ietf.org/html/rfc4861#page-8.

[13] N. I. o. S. a. Technology, «Secure Hash Standard (SHS),» March 2012.

[14] J. Martí, A. Queralt, D. Gasull, A. Barceló, J. J. Costa y T. Cortes, «dataClay: A distributed data store for effective inter-player data sharing,» *Journal of Systems and Software,* vol. 131, pp. 129-145, 2017.

[15] «IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator,» [En línea]. Available: http://w1.fi/hostapd/.

[16] «Open Hub,» [En línea]. Available: https://www.openhub.net/p/iw.

[17] «WHIRESHARK,» [En línea]. Available: https://www.wireshark.org/.