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

# D3.6 mF2C Agent Controller block integration (IT-2)


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


| | |
|---|---|
| **Work Package** | **WP3, mF2C Agent Controller block design and implementation** |
| **Due Date:** | M33 |
| **Submission Date:** | *30/09/2019* |
| **Version:** | *3.0* |
| **Status** | Final |
| **Author(s):** | Roberto Bulla (ENG), Matija Cankar (XLAB), Cristovao Cordeiro (SIXSQ), Jordi Garcia (UPC), Jens Jensen (STFC), Eva Marín (UPC), Xavi Masip (UPC), Antonio Perra (ENG), Anna Queralt (BSC), Antonio Salis (ENG), Sergi Sánchez (UPC), Sašo Stanovnik (XLAB), Roi Sucasas (ATOS) |
| **Reviewer(s)** | Antonio Salis (ENG)<br>Sašo Stanovnik (XLAB) |


| **Keywords** |
|---|
| mF2C Agent Controller, Implementation, Integration, TRL |

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

*This document is issued within the frame and for the purpose of the mF2C project. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 730929. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.*

## Version History

| Version | Date | Comments, Changes, Status | Authors, contributors, reviewers |
|---|---|---|---|
| 0.1 | 26/6/2019 | Table of contents | Anna Queralt (BSC) |
| 0.2 | 12/7/2019 | Completed first version of section 6 | Cristovao Cordeiro (SIXSQ) |
| 0.3 | 16/7/2019 | Completed first version of sections 2, 3, 8 | Eva Marín, Xavi Masip (UPC), Matija Cankar, Sašo Stanovnik (XLAB) |
| 0.4 | 17/7/2019 | Completed first version of section 4 | Roi Sucasas (ATOS) |
| 0.5 | 23/7/2019 | Completed first version of section 5 | Jens Jensen (STFC) |
| 1.0 | 24/7/2019 | First integrated version | Anna Queralt (BSC) |
| 1.1 | 6/8/2019 | Completed section 7 | Roberto Bulla, Antonio Salis, Antonio Perra (ENG) |
| 1.2 | 29/8/2019 | Revised version of section 4 | Roi Sucasas (ATOS) |
| 1.3 | 2/9/2019 | Revised version of sections 3.1, 3.2, 3.3, 3.5, 9 | Eva Marín, Sergi Sánchez(UPC) |
| 1.4 | 5/9/2019 | Revised version of section 5 | Jens Jensen (STFC) |
| 1.5 | 12/9/2019 | Revised version of sections 3.1, 3.4 | Jordi Garcia, Eva Marín (UPC) |
| 2.0 | 12/9/2019 | Complete version for internal review | Anna Queralt (BSC) |
| 2.1 | 13/9/2019 | Version after review 1 | Antonio Salis (ENG) |
| 2.2 | 17/9/2019 | Version after review 2 | Sašo Stanovnik (XLAB) |
| 2.3 | 25/09/2019 | Updates according to reviewer's suggestions | Eva Marín, Xavi Masip (UPC) |
| 2.4 | 25/9/2019 | Final version ready for quality check | Anna Queralt (BSC) |
| 3.0 | 30/9/2019 | Quality check performed. Deliverable ready for submission. | María Teresa García (ATOS), Anna Queralt (BSC) |

# Table of Contents

## List of figures



## List of tables

## Executive Summary

This document describes the final implementation and integration of the mF2C Agent Controller, as well as the transversal security, event manager, and dashboard components.

The main focus of this document is the description of the implementation and integration of the components within the mF2C Agent Controller, and the security, dashboard and event manager transversal components, in order to support the mF2C architecture and functionalities planned for the second iteration phase of the project (IT-2). The implementation of the Sensor Manager, an added-value feature sitting on top of the architecture, is also provided. The deliverable includes also the technical maturity assessment of the individual components, and the lessons learnt throughout their development.

The outcome of this deliverable is the implementation of the aforementioned components and extensions, all of them available in a public repository.

# 1. Introduction

In D3.4 [1] the final design of the mF2C Agent Controller was reported, taking into consideration the set of requirements to be satisfied by the mF2C platform and resulting in a detailed set of sequence diagrams specifying the functionalities aimed at adding dynamicity to the mF2C platform developed in IT-1. Also, the final design of the Security, Event Manager, and Dashboard was provided.

Here, the final implementation and integration of the Agent Controller and the aforementioned transversal components are described. The implementation is available in the mF2C repository in GitHub [2], where the different partners have published their contributions during the project. Also, the technical maturity of each component is assessed, and results and conclusions drawn from the design and development process are provided. Thus, for each component we provide the following information:

- Description of its implementation
- Description of its integration with interacting components
- Technical maturity assessment
- Lessons learnt in the form of conclusions or scientific results

In this document we also describe the implementation of the Sensor Manager, with a functionality that, despite being beyond the scope of the mF2C platform, was found desirable for the Smart Boat use case. It has been implemented as a generic added-value extension, at application level, to facilitate the development of applications that consume data from sensors.

## 1.1. Purpose

The main objective of this deliverable is to provide the final implementation of each of the functionalities in the Agent Controller, the implementation of the security and event manager components that support the rest of the functionalities, and the implementation of the Dashboard GUI, all of them according to the design defined in D3.4 [1]. Also, an extension for applications that consume data from sensors is provided.

The final implementation of the interfaces for all components is provided in D4.8 [3].

The outcome of this deliverable, together with D4.6 [4], will be used for the final PoC integration and demonstration tasks in WP5.

## 1.2. Structure of the document

The structure of the present document is as follows:

- Section 1 describes the aim and the context of this document.
- Section 2 provides an overview of the AC integration in IT-2.
- Section 3 describes the final implementation of the Resource Management component.
- Section 4 describes the final implementation of the User Management component
- Section 5 describes the final implementation of the Security component.
- Section 6 describes the final implementation of the Event Manager component.
- Section 7 describes the final implementation of the Dashboard component.
- Section 8 describes the final implementation of the Sensor Manager extension.
- Section 9 provides the conclusions of the document.

## 1.4. Glossary of Acronyms

| Acronym | Definition |
|---|---|
| AC | mF2C Agent Controller |
| API | Application Programming Interface |
| CA | Certification Authority |
| CAU | Control Area Unit |
| CIMI | Cloud Infrastructure Management Interface |
| CPU | Central Processing Unit |
| DDoS | Distributed Denial of Service |
| DNS | Domain Name System |
| GUI | Graphical User Interface |
| HTTP | HyperText Transfer Protocol |
| IoT | Internet of Things |
| IT-1 | Iteration 1 of the mF2C project |
| IT-2 | Iteration 2 of the mF2C project |
| LAN | Local Area Network |
| MQTT | Message Queue Telemetry Transport |
| MUD | Manufacturer Usage Description |
| PM | mF2C Platform Manager |
| RAM | Random Access Memory |
| REST | Representational State Transfer |
| SSE | Server Side Events |
| TLS | Transport Layer Security |
| TRL | Technology Readiness Level |
| VSIE | Vendor Specific Information Element |

**Table 1. Acronyms**

## 2. Agent Controller integration

For IT-2, and already stated in D3.4 [1], the two main blocks of the Agent Controller are the Resource Management block and the User Management block. Regarding the resources, the main functionality of the Agent Controller is to synchronize information in the mF2C agent database related to the resources, including the resource discovery, identification and categorization of them, and the policies to be applied. The User Management block in the Agent Controller has all the functionalities related to the management of the profiles, Sharing model and assessment.

Figure 1 presents the two blocks in the Agent controller, and each of their components.

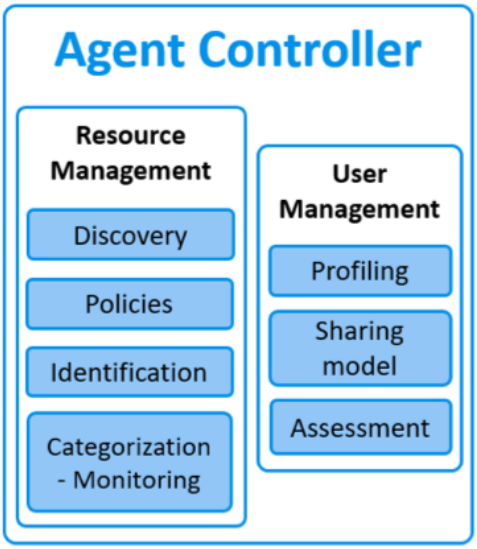


**Figure 1. Agent controller**

The interactions between blocks and components were described in depth in deliverable D4.7 [5], where the details of implementation of the interfaces were reported for IT-1, and which basically relies on the Cloud Infrastructure Management Interface (CIMI) interface, that allows the standardization of the communication between components. The updated definition of interfaces for IT-2 is defined in D4.8 [3].

# 3. Resource Management

The resource management block in the Agent Controller is composed of four main components:

- Identification
- Discovery
- Categorization
- Policies

In the following subsections, we detail the implementation of these four components, and how the implementation of the registration software has been done. Even though it is not considered a component per se in the resource management block, and in fact registration is done in cloud, we also describe the implementation of registration, since all resource management procedures are triggered by the registration process.

## 3.1. Registration

The registration is the component of the mF2C system where the users register with the platform. Each user wishing to use the pool of resources available in the mF2C environment or share their own resources must register in mF2C in advance.

### 3.1.1. Implementation

The implementation of the Registration module is published in GitHub in: https://github.com/mF2C/Registration-Identification

The registration is done through a web site hosted in the mF2C cloud agent. The component has been developed using open source tools, such as Apache as the webserver, using PHP as the programming language, and JavaScript to enrich the user experience.

During the registration process, the existence of a secure communication channel between the dashboard and the CIMI server is assumed. Such communication channel is used to send the data provided by the end users through a web form in the registration site to the CIMI service. In the current implementation, sensitive data such as the user password is hashed by CIMI before storing it into dataClay. The hashing method used provides a sufficient degree of security in the current implementation, aimed at showing the operation of the full system. More complex methods can be applied in the future.

### 3.1.2.Integration

The registration component interacts with the CIMI module each time is required to store/retrieve information. For example, during the new account creation to store the new user or during the login into the dashboard to validate the user credentials.

Registered users who have activated their account will gain access to the dashboard from which they will be able to download the agent docker-compose file. This file includes the IDKey environment variable in the identification module section, which is unique for each user. This variable is used by the identification module to acquire a new deviceID. Additionally, information related to the hardware components in the target device will be passed in the same way to the categorization module.

#### 3.1.3.Technical maturity assessment

The registration component has been validated in the UPC's lab premises using desktop computers and other mobile devices, such as laptops, smartphones and tablets, and as such it has a TRL 4: Technology validated in lab.

#### 3.1.4.Lessons learnt

Even though it is conceptually decoupled from the rest of the system, the registration website is an essential component of the mF2C environment. It is there where users have the first contact with the environment and subscribe to the service by providing some personal data.

During the development of the component we learned about the importance of safeguarding users' private data. In this sense, the main source of information were face-to-face meetings with experts in the field. They gave us some valuable experiences and some tips to protect stored data as well as minimize the impact in case of any leakage.

### 3.2. Identification

In mF2C, the identification module is responsible for managing the device (deviceID) and user (IDKey) identifiers. In this context, managing means to acquire both, the deviceID and the IDKey, when the agent is executed for the first time in the device, to store the obtained identifiers and to share them with other modules that require it.

#### 3.2.1. Implementation

The Identification module has been developed using the Java programming language and is available in GitHub in: https://github.com/mF2C/Registration-Identification

The module is used through RESTful calls. Using the GlassFish application server, the module exposes two endpoints:

- registerDevice:
  - **Description:** the main purpose of this sub module is to acquire a deviceID for the host device. In the case where the docker-compose file has been obtained through the web dashboard, the method input parameter is available as an environment variable and thus, the input data detailed below is optional.
  - **Method**: POST
  - **Input data:** user username, user password (optional)
  - Input format: JSON
  - **Output data:** status, message
  - Output format: JSON
  - **Input example:** {"usr":"user_username", "pwd":"user_password"}
  - **Output example:** {"status":"201", "message":" Agent IDs have been saved"}
- deviceID:
  - **Description:** once the identification module has acquired the deviceID through the registerDevice method, the deviceID endpoint will make it available.
  - Method: GET
  - Input data: N/A
  - Input format: N/A
  - **Output data:** status, IDKey, deviceID
  - Output format: JSON
  - Input example: N/A

- **Output example:** {“status”: ”op_status”, “IDKey”: ”user_IDKey”, “deviceID”: ”acquired_deviceID”}

#### 3.2.2.Integration

The identification module interacts with two system components: the policies module and the registration component.

The identification module interacts at least twice with the polices module. The first interaction is when the policies module triggers the identification module the first time the agent is executed in the device. The second interaction takes place when the polices module requests the deviceID and IDKey to subsequently send them to the categorization module. After these events, the identification module is still available through the deviceID endpoint, not only for the polices module but for any other service that needs the device and user identifiers for future queries.

#### 3.2.3.Technical maturity assessment

The identification module has been validated in the UPC’s lab premises (using desktop computers and Raspberry Pis 3) and as such it has a TRL 4: Technology validated in lab.

#### 3.2.4.Lessons learnt

The use of hash values as identifiers provides some key advantages in comparison with other naming techniques. The main advantages are: i) hashes are not reversible, so in case of using sensitive information as input, it cannot be inferred or obtained; ii) from a practical perspective, the risk of collision is zero; iii) easy to compute, even for resource constrained devices; iv) the text is meaningless and thus, no information is exposed.

### 3.3. Discovery

The discovery component is responsible for allowing leaders to advertise their presence so that regular agents can become aware of their existence. This is done by leveraging the 802.11 protocol, and more specifically, by embedding leader information within the vendor-specific information element (VSIE) of the 802.11 beacon, as explained in D3.3 [6]. This beacon would then be broadcast by the leader so that nearby agents can detect it by scanning for the mF2C VSIE.

#### 3.3.1. Implementation

The discovery component is written in Python and available in GitHub in: https://github.com/mF2C/ResourceManagement

It exposes a REST API through a lightweight Flask web app. To ensure its expected functionalities, it relies on the following wireless tools provided by the Linux operating system:

- hostapd: It allows the creation and configuration of software access points in Linux. We use its inherent capability to manage 802.11 beacons to allow a leader to broadcast custom mF2C beacons.
- dnsmasq: We use its DHCP server feature to allow a leader to assign IP addresses to agents within its area.
- iw: a Linux tool for managing wireless devices. We particularly make use of the scan functionality provided by iw to allow agents to scan for nearby leaders.
- wpa_supplicant: It handles wireless network association and authentication. We use it to allow an agent to associate to a nearby leader.

For a smooth management of such wireless tools, the following classes have been defined as part of the discovery component.

**Broadcaster**

This class facilitates the beacon broadcasting functionality. It contains the following methods:

- start_broadcast(): It starts the hostapd service, which is responsible for broadcasting beacons.
- start_dhcp(): It starts the dnsmasq service, which is responsible for assigning IP addresses to agents who joined the leader.
- stop_broadcast(): It stops the broadcasting functionality, by stopping the hostapd service.
- check_active(): It checks if the hostapd service is active.
- check_inactive(): It checks if the hostapd service is not active.
- fill_beacon_fields( broadcast_frequency, interface, config_file): It uses the parameters provided as an input, such as the beacon frequency, the wireless interface name and the configuration file containing the leader information, to fill the hostapd.conf file that will be used to start the hostapd service properly.

**Scanner**

This class is responsible for scan-related operations. The methods that it provides are presented as follows:

- start_scan(interface): It starts the scan on the provided wireless interface using the iw wireless tool.
- is_mf2c_oui_found(line): It checks if the mF2C organizationally unique identifier (i.e. ff:22:cc) is present within a given line of the scan results.
- get_f2c_ie(item): It retrieves the mF2C information element from the scan result that was provided as an input.
- get_mac(item): It retrieves the MAC address found in the scan result that was provided as an input.
- return_list_of_results(content): It takes the output of the iw wireless tool and organizes it into a list containing separate scan results.
- parse_scan_results(scan_results): for each scan result, it checks if it contains the mF2C OUI. If so, the subsequent vendor content is decoded and a new leader is added to the leaders_list.

**JoinConfig**

This class has the necessary operations allowing a regular agent to join/unjoin a leader's area. It is comprised of the following methods:

- config(bssid): It writes the wpa_supplicant.conf configuration file
- join(interface): It starts the agent-leader wireless association process, using the wpa_supplicant tool.
- unjoin(): It stops the agent-leader association, by terminating wpa_supplicant.
- check(): It checks the current status of the agent-leader association (i.e. connected, disconnected).
- get_ip(): It returns the IP address of the agent.

**Watcher**

This class contains the operations allowing the monitoring of wireless events on the leader and agent sides. This is done through the following methods:

- on_topology_changed(): This method is only relevant in the leader side. It allows it to monitor the events of agents joining/ leaving it by means of the hostapd_cli tool.

- update(mac_addr): When a leader detects the event of an agent leaving its area, it updates the corresponding CIMI resource of that agent by changing its status to DISCONNECTED.
- stop(): It stops the monitoring on the leader side by stopping the hostapd_cli tool.
- on_leader_connection_changed(): This method is only relevant in the agent's side. It allows the agent to detect the events of whether its leader is connected or disconnected by means of the wpa_cli tool.

**VSIE**

This class hosts the operations for managing the encoding of the leader beacon information into the proper format of the 802.11 vendor-specific information element, as described in D3.5 [7].

**InformationElementAttribute**

This class manages the information element attributes to be included in the mF2C beacon, as described in D3.5.

### 3.3.2.Integration

As shown in Figures 9 and 10 in D3.4 [1] the discovery component has been integrated with the policies component. In fact, the discovery component exposes two API calls allowing the policies component to trigger the scanning process and the beaconing process, at the agent and the leader sides, respectively. These calls are the following:

- /broadcast: It is a HTTP POST call, taking as input a JSON with the configuration parameters, e.g.: {"broadcast_frequency":"100","interface_name":"interface_name", "config_file": "mF2C-VSIE.conf",”leader_id”:”the_leader_ID”}. The broadcasting is then started using this configuration.
- /scan/<interface_name>: It is a GET REST call taking as a parameter the name of the wireless interface to be used.

### 3.3.3.Technical maturity assessment

The discovery component has been validated in the UPC's lab premises (using laptops and Raspberry Pis) and as such it has a TRL 4: Technology validated in lab.

### 3.3.4.Lessons learnt

As mentioned in 3.3.3, the discovery component has been tested using laptops and Raspberry Pis, using the Linux operating system. Testing on other operating systems has not been possible since, unlike Linux, they do not allow access to and manipulation of the 802.11 wireless protocol, which is needed for the proper functioning of the discovery component. In addition, while performing our tests, we noticed that certain Wi-Fi network cards (either built-in or external dongles) do not support the feature for broadcasting customized vendor-specific information elements, which limited the set of devices where could test this component.

| | Raspberry Pi 3 with built-in Broadcom chip | Laptop with built-in Broadcom chip | Laptop with built-in Atheros chip |
|---|---|---|---|
| Average execution time (sec.) | 1.041 | 4.21 | 3.353 |

**Table 2. Scan execution time**

In Table 2, we show the scan execution time when 3 devices with different wireless chips were used as agents and a laptop with a built-in Atheros wireless chip acting as a leader. As it can be noted, the

obtained results highly depend on the used wireless chip and they account for the duration spent to go through all the wireless channels to receive beacons.

Certainly, the implementation of our approach depends on the manipulation of the 802.11 protocol, which is not available out-of-the-box in some operating systems. However, unlike other discovery solutions, it does not assume that the agent and the leader are part of the same local area network (LAN), as is done in multicast DNS-based discovery approaches. It is also edge-centric, unlike approaches relying on the use of a remote cloud-based repository hosting the set of available agent resources.

## 3.4. Categorization

The job of the resource categorization module is not only characterizing resources according to their hardware, software, network or Internet of Things (IoT) related information, it is also responsible for the continuous monitoring of resources. Apart from monitoring the resources of each agent, when the agent is the leader of an area, it is also in charge of aggregating all resource information of the Fog Area (including its child information) and publishing it as the available resource capacity of that corresponding Fog Area.

### 3.4.1. Implementation

The implementation of this component is published in GitHub in: https://github.com/mF2C/ResourceManagement

According to the participatory nature of the different layer's agent, they can be classified into two types: Normal Agent and Leader Agent. For the Normal Agent, the resource-categorization module is responsible for creating two kinds of information: Static information ('device`) and dynamic information ('device-dynamic'). For all the participating devices, their maximum resource capacity is fixed at a certain bound and as they are performing some computational tasks, so the available resource capacity may differ. For that reason, we realized that, before classifying the participating devices according to their resource capacity, it could be relevant to know their total capacity and also the available capacity, and that's why we split the resource information into two parts. Also, the resource-categorization module is responsible for updating the `agent-resource` information with the proper device IP for every agent.

In the case of a Leader Agent, this module not only creates the aforementioned two information types (static and dynamic) but also it is responsible for aggregating its child resource information. Similarly, for Leader Agent side it is also responsible for updating the 'agent-resource' information along with its child device IPs and own IP.

In the following sections we are going to briefly discuss the different types of information, which are geing generated and updated by the resource-categorization module.

**'device (device_static)**

This information contains the static part of resource information. For example, CPU core, networking standards, total RAM capacity etc. It contains three functions:

- hwsw_info - This function is in-charge of collecting the total RAM size, OS information, total Disk size and CPU related info.
- net_stat_info - This function is collecting the available network standard information of the device.
- hwloccpuinfo - This function is designated to collect the `hwloc` and `cpuinfo` of the device.

To collect these pieces of information, we use some existing python packages - 'cpuinfo', 'psutil', 'platform' etc.. These packages are helping us get the resource information as per the desired way.

**'device_dynamic'**

As the name suggests, this part contains the dynamic information of the resource components. For example, CPU usage, storage availability, RAM availability etc. Similar to the `device_static`, we also created some functions to collect the aforementioned resource information. Below, we are going to briefly discuss their functionality:

- hwsw_dyna_info - This function is in-charge of collecting the information of power consumption, available disk size, available CPU capacity, available RAM size information.
- net_dyna_info - This function is designated to collect the information of device IP and the data packet transfer (i.e, throughput info) and data packet loss.

For collecting these pieces of information, we are using some of the existing python packages - 'psutil', 'platform' etc.

**'fog_area'**

This information is basically created by the Leader agent of the corresponding Fog area. Basically, this aggregated resource information, which is created and updated by considering the 'device-dynamic' and 'device' information of all the resources within a fog area. To create this aggregated information, we build a python function named - `fogarea` to collect the information of the aggregated fog area resources.

- fogarea - It is designated to form the aggregation of fog-resources, which are located in the same fog-area. Considering their `device-dynamic` and `device` (device-static) information, this function is aggregating the resource information and giving the output to know the available capacity of fog area resources It is also helping to identify the fog area resources which have the highest and lowest computational capability (i.e., CPU, RAM, Power etc.), within the same fog area.

### 3.4.2. Integration

As shown in Figures 7, 8 and 9 in D3.4 [1], the resource-categorization component has been closely integrated with the Policies component, basically the resource-categorization module starts working once it is triggered by the policies component. As we said earlier, once the CIMI container is up and the Policies component gets the device-id, leader-id and the parameter 'isleader' info, then it triggers the resource-categorization module, both at the agent and the leader sides.

### 3.4.3.Technical maturity assessment

The resource-categorization component has been validated in the UPC's lab premises (using laptops and desktop machines) and as such it has a TRL 4: Technology validated in lab. Also, it has been tested in TISCALI-ENGINEERING cloud.

### 3.4.4.Lessons learnt

The biggest contribution of resource-categorization module is the taxonomy of mF2C resources, which has been presented in D3.3 [6], and the corresponding class diagram presented in D3.4 [1], (Fig. 5). Though in some of the works [8] [9] [10] [11] [12] [13] [14], researchers separately provide some resource classification model for their considering system (i.e., Fog, IoT and Cloud), we haven't found any work where researchers provide some taxonomical model for the hierarchical, combined, diversified and distributed system such as mF2C. So, that's why it is necessary for building our taxonomic model for mF2C platform. To fulfil that demand the resource-categorization module is in

charge of collecting various resource information and helping classify the resources according to the collected information.

## 3.5. Policies

Policies block is in charge of defining the set of rules that manage the overall mF2C agent and the architecture, and the mechanisms associated to enforce these policies.

### 3.5.1. Implementation

The implementation of the Policies component is available in GitHub in: https://github.com/mF2C/ResourceManagement

The modular design and implementation of the component is composed of three different principal sub-modules:

**Area Resilience and Leader Reelection**

Set of mechanisms in charge of the protection of the control of the areas and leaders, following the defined policies on Leader Selection and Backup Selection, and the replacement of the active leader.

Area Resilience is designed to follow a proactive approach, electing a “backup” leader that will overtake the Leader responsibilities in case of failure. To make this possible, a primary routine evaluates the capability of the agent to be a leader and to switch to leader if required, while the leader has a special routine to select backups and the health checks between them.

Leader Reelection is a routine that is only active when triggered, that replaces the active leader with another capable agent. Most of the functions and procedures executed are part of the Area Resilience sub-module, requiring coordination and communication between them.

**Resource Manager Orchestration**

Control of the different modules involved in the resource manager depending on the actual state of the agent (active leader, backup or normal agent). Moreover, defined mechanisms to select a leader in the absence of another one nearby is implemented inside this sub-module.

**Policies**

Sub-module defining all the rules and parameters active on the system. This includes the implementation to request the active rules of the system and to make changes on them.

A set of tests have been proposed and used to validate any new changes on the implementation, to assure that the main functionalities are fulfilled. For instance, the backup-leader communications and failures are tested to check that the proposed mechanisms resolve the anomalies on the system.

### 3.5.2. Integration

To make the communication between other modules and Policies possible, a REST API call is provided for each responsibility required. The interface of the module that makes possible the integration between modules is explained in D4.8 [3]. In Figure 2, the integration between Policies module and other modules is presented.

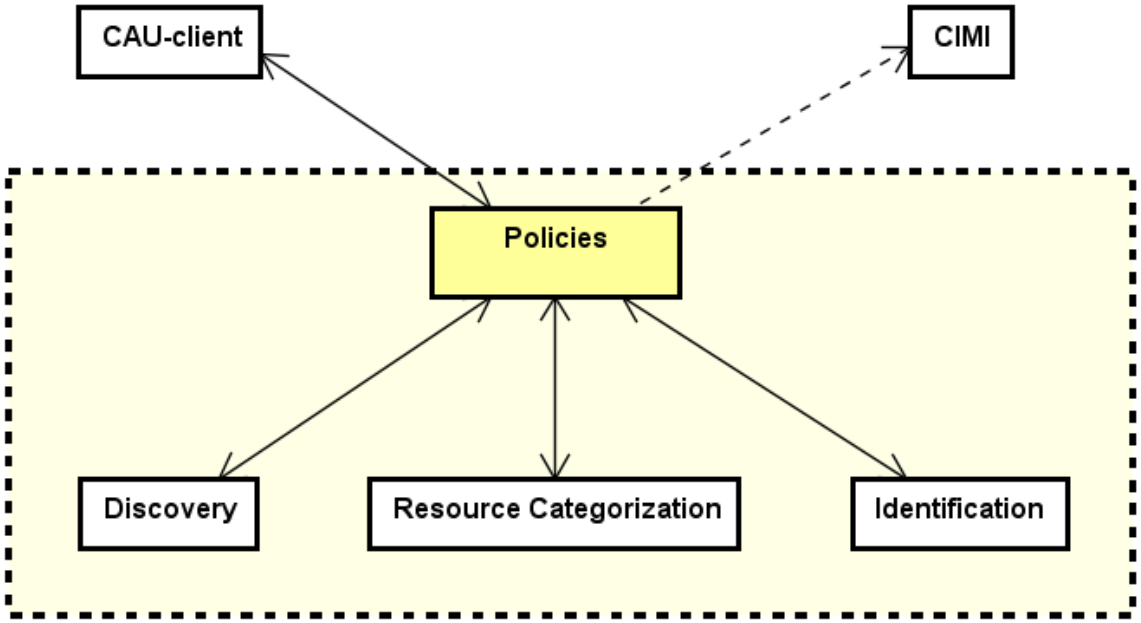


Figure 2. Policies integration with other mF2C modules

Policies has dependencies on all the Resource Management modules. In other words, the Policies module is integrated with Discovery, Resource Categorization and Identification, using their respective interface specifications. Moreover, Policies module depends on CIMI to receive and store rules and parameters used by other modules. The data generated by the module, result of the current policies and interaction between modules, is stored as a resource into CIMI, represented as a discontinuous line in Figure 2. Finally, as the Policies module has an important role on the start-up or role change of the Agent, there is also a dependency with the CAU client module, resulting on the corresponding integration. All the interactions between the mF2C components is done through the provided REST APIs.

### 3.5.3.Technical maturity assessment

TRL 4 – technology validated in laboratory

The module has been validated in laboratory premises, using a testbed to emulate the functionality of the module in a scenario with hundreds of others, and the integration between mF2C components.

### 3.5.4.Lessons learnt

One of the challenges of this component is the definition and implementation of the rules that affect how the mF2C architecture is built and the behaviour of other components. As the specifications of the architecture are quite unique, a custom implementation of the component was required. The implementation is made from scratch, requiring some tests and emulation to verify that the functionality is fulfilled correctly. The resulting component has been shown to work correctly, based on individual tests and multiple deployment in a closed laboratory environment, where the interaction between components has been verified and the expected outcome form each action has been successfully archived. However, there is room for improvements in terms of performance and scalability.

# 4. User Management

The User Management module is responsible for defining how an mF2C agent participates in mF2C and which agent's resources (RAM, CPU, application running, etc.) will be shared with other agents. It is also responsible for doing the corresponding assessment to assure that the agent shares only the resources defined by its owner.

This module is composed by the following subcomponents:

- Profiling: user profile properties that define how the user's device (agent) participates in mF2C.
- Sharing Model: sharing model properties that define the resources to be shared with other agents.
- Assessment: process that monitors the resources shared by the agent.

All of them have been integrated and released as a single component written in Python. A more detailed description of this module, its components and functionalities can be found in previous deliverables.

## 4.1. Implementation

All the components of this module have been implemented in Python 3, and the resulting code is available and published in GitHub under Apache License version 2.0 in: https://github.com/mF2C/UserManagement

The User Management has been also released as a docker image together with the Lifecycle Management component in https://cloud.docker.com/u/mf2c/repository/docker/mf2c/lifecycle-usermngt

The code is composed of two main blocks or packages. The package 'data' is responsible for managing the resources (CIMI) and all the information used by the User Management. The other package, called 'modules', contains all the logic of the User Management. This logic includes the creation, management and deletion of the resources managed by this module, and the background processes that are permanently checking the resources shared with other mF2C agents.

## 4.2. Integration

The User Management module presents some changes with respect to the dependencies diagram described in deliverable D3.5 [7]. According to the workflows described in previous deliverable D3.4 [1], the new dependencies diagram can be found in Figure 3.

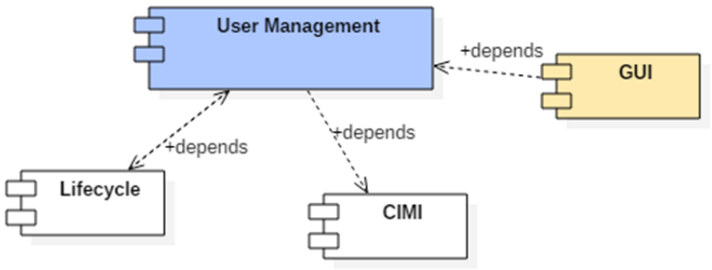


**Figure 3. User Management dependencies**

The Profiling and Sharing Model subcomponents are used to create and define the properties related to how one mF2C agent participates in mF2C and which resources it will share with the other mF2C agents. Thus, these subcomponents only depend on CIMI. The final users can view and edit these properties through the Dashboard GUI, as defined in deliverable D3.5.

The Assessment subcomponent makes use of the information managed by the other two subcomponents of the User Management module together with other resources stored in CIMI to continuously check and assess the resources shared with other agents. If it detects a violation, like more mF2C applications than allowed (defined by the user) running in the agent, then it sends a warning notification to the Lifecycle.

## 4.3. Technical maturity assessment

This component was successfully tested in a laboratory environment together with the Lifecycle Management module and other components of mF2C, achieving a TLR 4 (technology validated in lab) at the time of writing. It is expected to improve this module to achieve at least a TLR 5 (technology validated in relevant environment) at the end of the project.

## 4.4. Lessons learnt

During the design and development phases of the User Management component we had to face several challenges, like the decision of the information that belongs to the Sharing Model, the Profiling and the User. The difference between the properties that define these three different entities is not always very clear, and we had to make several changes related to the definition and location of them during the two iterations. Apart from that, the main challenge we had to face during the development and testing of the User Management module has relation with the tests needed to integrate it with other components. Another challenge we had to face is related with versioning and dependencies between the components. Each mF2C component has been released as a single microservice application that depends on others, with multiple versions each one. Thus, we had to overcome these difficulties when testing new features of a specific component or when doing integration tests.

# 5. Security

Getting security right is important for any IT platform. If security is not implemented correctly, people either will not trust the platform, or the operator might find resources misused, data compromised, customers leaving the platform, etc., all of which can be costly. Obviously, security has been described at great length in earlier deliverables – D2.5 (security and privacy for IT-2) [15], D2.7 (IT-2 architecture) [16], D3.2 (security for AC) [17], D4.2 (security for the PM) [18].

## 5.1. Reverse Proxy

The Traefik reverse proxy was described in D3.2, section 4.6.1. It is an already existing external component that has been integrated into mF2C to secure inter-agent communications.

### 5.1.1. Implementation

The implementation is Traefik (https://hub.docker.com/_/traefik). It is deployed in the Agent's and Cloud docker-compose.

### 5.1.2. Integration

As described in D3.4, section 5.1, the only remaining task was to use the Agent credential as the proxy credential (as opposed to a self-signed certificate). The agent credential is available in a volume where it is placed by the CAU-client component. The implications of this are that (a) the credential needs to be available (through the CAU-client) before the reverse proxy can be active, and (b) the client credential needs to have the right names and sufficient extensions to enable it to be used as endpoint protection. This also means CAU-client cannot require the reverse proxy.

### 5.1.3.Technical maturity assessment

Traefik is a widely used external component, so we can classify it as TRL9 – Actual system proven in operational environment.

### 5.1.4.Lessons learnt

The agent credential is designed to be a client credential, which enables the agent to connect to a server and authenticate itself through (usually) TLS. To achieve this, the credential needs to:

1. Be a valid certificate, issued by a trusted CA;
2. Have the associated private key available to the client;
3. Have client extensions.

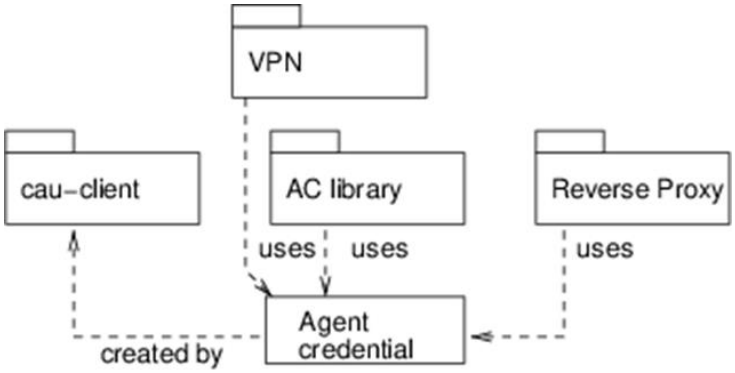


**Figure 4: Reverse Proxy use of Agent credential**

It is not obvious that the credential can also be used as a server credential: for HTTP connections (in Traefik, in our case): RFC 2818 [19] states that the client must check the server's identity by validating the name or address in the server's certificate against the name or address which the client thinks it is connecting to. To be a server credential protecting a HTTP endpoint, the credential must have a private key and contain a valid certificate issued by a trusted CA, as above, but it must also contain the name or address of the server. Every other use of the agent credential (Figure 4) by other components is as a client credential.

However, it turns out that clients connect to the Traefik reverse proxy without a server name or address in the credential. The credential is issued with both HTTP server and client extensions (through the extendedKeyUsage extension), and this is sufficient.

## 5.2. CAU Client

The CAU client enables an agent to interact with the CAU. The most important role of the CAU-client is to ensure that the agent credential is obtained from the fog CA, and is made available to other components that need it. In turn, the purpose of the CAU is to enable the connection of the agent to the CA issuing the credential: as discussed in previous deliverables, by default, an agent connecting to a fog would not be granted access to the Internet, but it does need to connect to the CA in the cloud to get a trusted certificate issued to it. The CAU is the trusted gateway that enables this to happen.

### 5.2.1. Implementation

There is a CAU-client repository in GitHub (https://github.com/mF2C/cau-client). Note that as of this writing, it is the `IT2` branch which contains the code used in IT2, not the `master` branch. It consists mainly of four sub-components:

1. Configuration and utilities
2. A credentials manager which creates keys and certificate signing requests, and stores the certificate with the private key when the certificate is available.
3. A CAU interface.
4. A CIMI interface, used to register the agent

### 5.2.2.Integration

The cau-client is integrated into the agent, storing credentials into a mounted volume called 'pkidata' and mounted on /pkidata. Other components that need the credential are expected to mount the volume (preferably read-only.) The certificate is stored in a file `server.crt`, but PEM formatted. The corresponding private key is stored in a file called `server.key`; the private key is also PEM formatted (PKCS#8), and stored in unencrypted form.

### 5.2.3.Technical maturity assessment

The CAU-client has remained continuously in service, shielding its clients from the re-engineering of the CAU and the changes to how clients are registered in CIMI. We estimate the TRL at 4 (Technology validated in lab), based on the operational experiences with CAU-client in the testbed, the state of the code documentation and robustness (checking errors etc.), the current state of the documentation, and the associated design documentation in deliverables.

### 5.2.4.Lessons learnt

The CAU client picks up its coordinates (location of CAU and Leader CAU) from the environment when the container is launched. More generally, this information needs to be defined in the docker-compose file, when the agent is launched with docker-compose: it relates to the general discussion

about the types of information that should be available in the docker-compose file: if we take the view that it should not be agent-specific, but it can be fog-specific, it makes sense for the docker-compose file to provide the information.

More generally, as has been highlighted before, mF2C lacks a trust anchor distribution mechanism, thus leading to images with hardcoded CA certificates in them. This is fine for the mF2C infrastructure itself and the mF2C use cases, but makes the components less reusable. A recent modification to the CA has made the CA certificate(s) available for download, which makes the trust anchor management more dynamic. However, if a certificate is downloaded dynamically, we still need to configure the relevant containers with the endpoint from which they can download this certificate, the means to validate that they are talking to the correct endpoint (e.g. another CA certificate for the endpoint security, or a fingerprint), and of course it assumes that the containers can reach the endpoint. We note that the current state is not radically different from the IoT platforms that come with trust anchors embedded in hardware.

## 5.3. AC Library

The purpose of the AC (Agent Controller) library is to provide a means for agents (and components of agents, and for applications running alongside the agents) to communicate in a way which is consistent with the security policy described in D3.1 [20]. Specifically, when using the AC library, the caller can specify whether the message is PUBLIC, PROTECTED, or PRIVATE.

### 5.3.1. Implementation

There is an AC library repository in GitHub - https://github.com/mF2C/aclib

### 5.3.2. Integration

AC library implements the security policy originally outlined in D3.1 [20]. It supports the security policy in D3.1 and the description of the requirements of the security policy in D4.1 [21], so should be used by every component and application that communicates over the network except for communication with dataClay, CIMI, and the Reverse Proxy, which are secured by separate means.

### 5.3.3.Technical maturity assessment

The AC library is documented and tested and has changed little over the integration of IT-2 components, but has primarily been tested in the testbed (which is noted for having more permissive communications, it is less aggressively firewalled than a production environment would be). However, since most of the development and integration of IT-2 has focused on inter-component control communications and less on applications and data classified according to the policy, the AC component has not been used extensively by other developers in mF2C. Consequently, we estimate the TRL at 4, because the AC library has not been used extensively outside of its test environments.

### 5.3.4.Lessons learnt

In the security design deliverables, we have discussed the need for understanding how messages communicate through the infrastructure. For the sake of argument, suppose we had a customer from outside the project who wanted to use all of mF2C, but they need to control communication through MUD (Manufacturer Usage Description [22]) in order to reduce the risk of the infrastructure being used to Distributed Denial of Service (DDoS) attack, etc. To implement this for the customer's application, it would be much easier if we could implement it in the AC library, knowing that all messages were communicated through this library.

# 6. Event Manager

As described in deliverable D2.7 [16], the Event Manager provides a functionality to the mF2C agents whereby other components can be notified whenever a certain event occurs.

## 6.1. Implementation

The Event Manager is open source and its implementation can be found in GitHub at https://github.com/mF2C/cimi-server-events.

This microservice will deploy a Redis server and a Flask-based SSE server. Events are generated by jobs, which are scheduled at start-up, according to the user configuration. These jobs are written in Python and they basically use CIMI's REST API to query for a specific resource and examine the differences between consecutive calls, i.e. if for example a new CIMI resource has been added, then the subsequent job will find a new entry with respect to its previous execution.

There is one channel per job, thus any client subscribed to the event stream of an existing channel, will receive CIMI-based events whenever the conditions described in the job are met.

## 6.2. Integration

The Event Manager is a standalone service that only depends on the CIMI server.

A configuration example is shown here:

```
{
  "cimi_api_url": "http://cimi:8201/api",
  "port": 8000,
  "channels": [
    {
      "job": "demo.py",
      "frequency": 5
    }
  ]
}
```

The API URL of the CIMI server that will be used is defined in "cimi_api_url". The HTTP port to be used by the other mF2C components when subscribing to this service is defined by "port". And for each job of the Event Manager there should be an entry in the channel list defined by "channels".

The existence of the CIMI endpoint "cimi_api_url/cloud-entry-point" indicates that the CIMI server is ready, and at this point, the Event Manager starts its own SSE service for the other mF2C components to subscribe to.

For another component to subscribe to the Event Manager, given that these microservices run within the same Docker network, all that is required is for the respective component to:

- Select the channel name to subscribe to (in the case of the configuration file above, the channel name would be "demo")

- Open a streaming connection to the Event Manager on *http://event-manager:8000*/stream?channel=demo
- Start an open ended thread listening to this stream.

When an event occurs, the Event Manager will send the respective information throughout the corresponding channels and the subscribed components will receive it.

Note that the Event Manager is a SSE service, which means that other components cannot send any messages through this service, but only receive.

## 6.3. Technical maturity assessment

The Event Manager has been implemented and tested quite extensively, and is now an intrinsic microservice of the mF2C Agent. There are mF2C components making use of it in testing and pre-production operational environments. Therefore, the Event Manager is considered to be TRL 5 – technology validated in relevant environment.

## 6.4. Lessons learnt

Its basic implementation provides enough functionality for the mF2C components to be notified upon certain events. However, in some scenarios, the possibility to also send notifications and events has been requested, thus our understanding is that for mF2C, this SSE-based Event Manager is significantly useful and even critical for some components, but it could also be improved to work as a bidirectional messaging service, thus allowing for components to also publish notifications and have the Event Manager acting as a message broker.

# 7. Dashboard/GUI

As described in deliverable D3.4 [1], the Dashboard/GUI is a new component added to the mF2C agent for IT-2. The purpose of this Dashboard is to facilitate the supervision and the interaction with the mF2C system from a service provider perspective, giving an overall picture about how the system works and providing information such as agent deployments and hierarchy, resource usage, events, etc.

The Dashboard includes the User Management, through which the device owner can specify if the device in which the agent runs is a resource user, contributor, or both. The owner is also able to configure the amount or resources made available to the system, such as CPU, RAM and storage, etc.

Lastly, through the Dashboard it is possible to register and launch services, monitoring their execution and possible SLA violations. With all these key functionalities, the Dashboard is thought as the main tool for high level interaction with mF2C system, thus avoiding use of manual or automated scripts to perform operations.

## 7.1. Implementation

The Dashboard is implemented as a web application built on Apache/PHP stack and packaged as a Docker container that runs within the same Docker network of the other mF2C agent components.

The source code, including installation instructions and user documentation are available on GitHub at the following link: https://github.com/mf2c/dashboard .

The Dashboard implements the following main features:

First, it provides a common GUI for all implemented functionalities, using a navigation bar in order to facilitate the navigation between different web pages. This GUI has a home page which focuses on Leader agent and backup functionalities, showing overall information like a representation of the cluster composed by a Leader with its children and the backup agent, the characteristics of the device, resource usage status like CPU, memory, storage, etc. The home page is also able to show alert messages when specific events occur.

Second, it collects and aggregates data coming from CIMI and other APIs, and processes them in order to display information accordingly in the home page.

Finally, it provides the scaffolding (container) to include and link other web pages provided by other components like Service Manager, User Manager, etc., which implement their own functionalities.

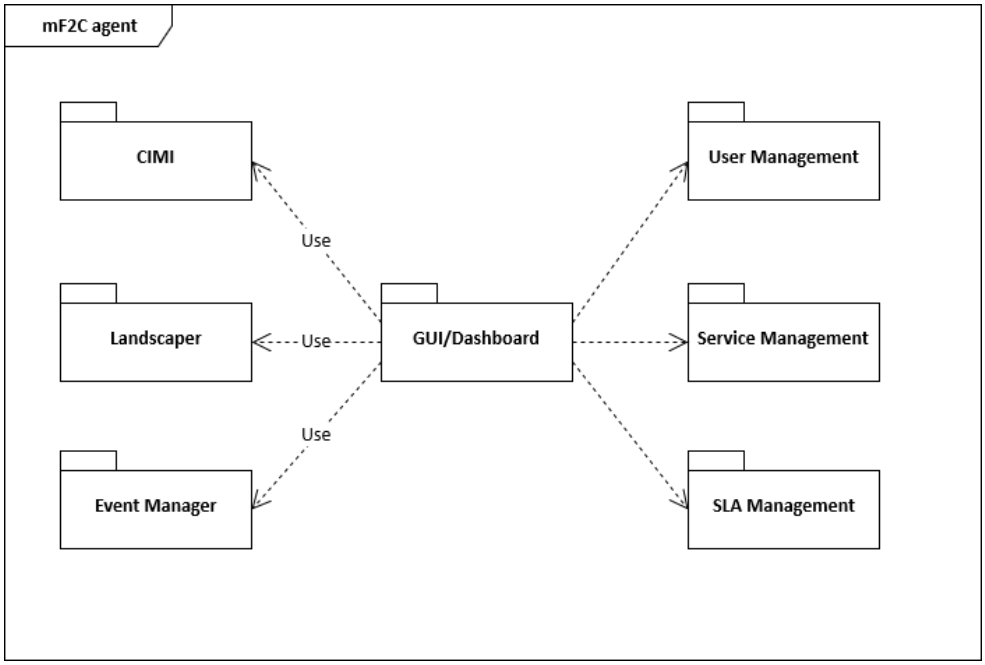


**Figure 5. GUI/Dashboard model diagram with relationships and dependencies**

Figure 5 illustrates the relationships between the Dashboard and other mF2C components. The Dashboard uses some APIs (indicated with <<Use>> in the arrows) and includes pages provided by other agent components.

As a standalone container, the Dashboard is deployed in all agents except for microagents.

The Dashboard can be loaded using a common web browser. Particular care has been taken to the technologies used for the development and the implementation in order to make it as much as possible backward compatible with old browsers.

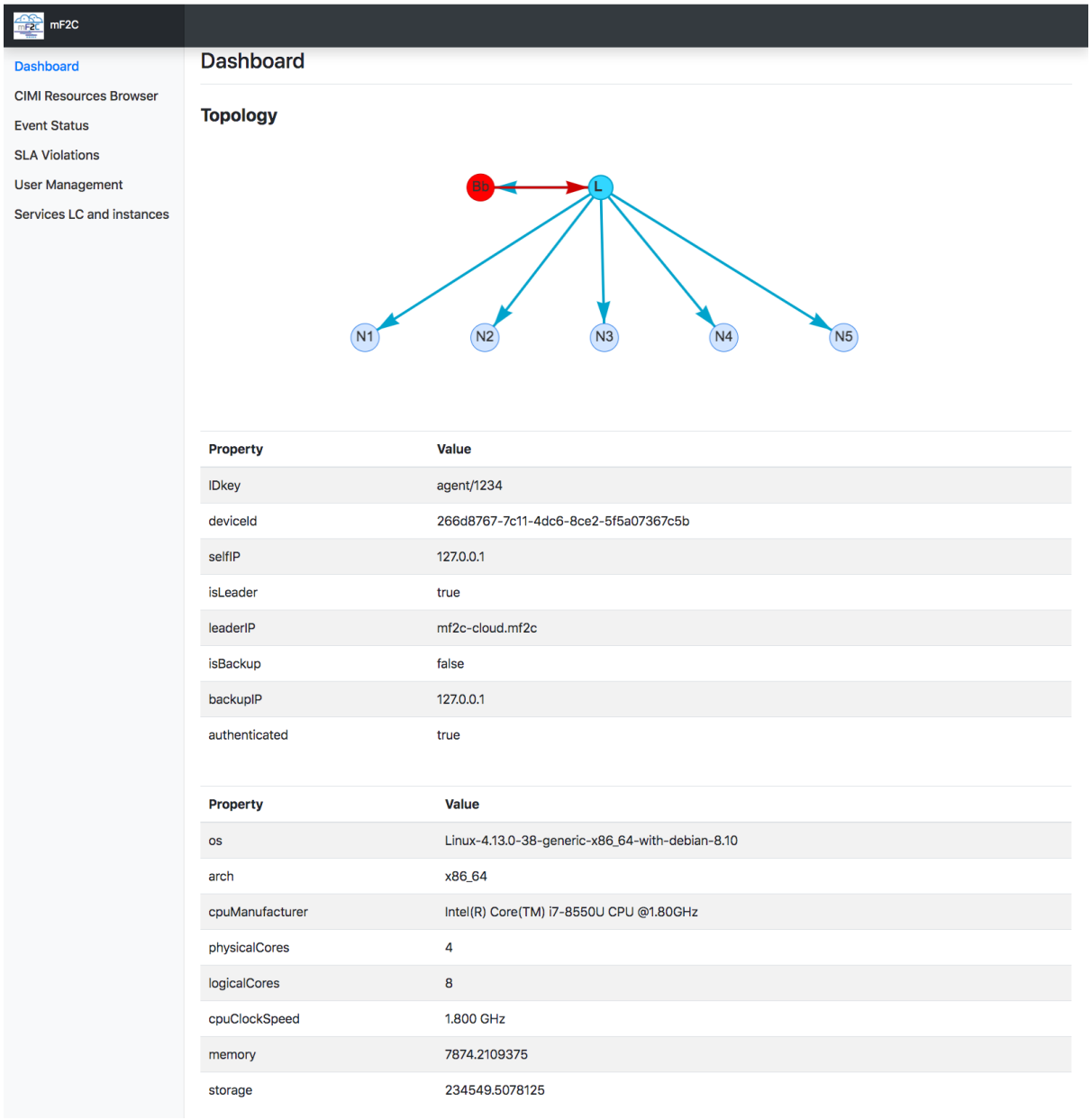


Figure 6. GUI/Dashboard home page

Figure 6 shows the current home page, which will be improved until the end of the project in order to fulfil all requirements.

## 7.2. Integration

To collect all data that has to be represented in the home page, the Dashboard uses basically the CIMI and Landscaper APIs. This allows to represent the features of an agent and the device in which it is running, along with its hierarchy and relationships with other agents (if the agent is a Leader).

Being structured as a single page application, the Dashboard uses a proper view to include the web pages provided by other components, that are used to implement specific functionalities, like, for example, register and launch a service or define user management policies.

The Dashboard is available via an URL mapped through Traefik component of the mF2C agent.

## 7.3. Technical maturity assessment

At the time of writing this document, this component is under testing in the Engineering testbed, together with the Service Management, Lifecycle Management and other mF2C modules, to reach a TRL 4 (Technology validated in lab) shortly.

It is expected to improve this module in the coming months to be used for the final demonstration of the use cases in real environments, such as an airport, thus achieving TLR 5 (Technology validated in relevant environment) at the end of the project.

## 7.4. Lessons learnt

The basic implementation of the Dashboard put in place is enough to show the benefits of using GUI when the supervising, monitoring and administration of an mF2C agent or a cluster of agents in the system is needed.

As mentioned before and stated in D3.4 [1] also, the Dashboard is basically a tool aimed at service provider and operators, rather than mF2C platform end users.

The experience gained in the project suggests that to manage complex scenarios or even to use the platform in a commercial solution, the access management capability - based on roles and privileges – must be added to the Dashboard in order to define how the different types of users (consumers, service providers, operators) can use specific system features, and at the same time to prevent and avoid misuses.

# 8. Sensor Manager

The Sensor Manager is an added-value extension to mF2C that manages sensor readouts in the agent. Although being implemented at application level, i.e. sitting on top of the mF2C platform, and having identified the need within the scope of a particular use case, it is a general extension that helps applications to consume data from the sensors attached to an agent. The Sensor Manager features an abstraction layer to sensors, allowing simple, concurrent, hardware- and language-agnostic way to access sensor data without having to write any low-level sensor handling code. It is packaged as a mF2C application and builds on top of the platform to significantly expand the functionality available in IoT scenarios.

Multiple clients (applications, users) can subscribe to the same set of sensors simultaneously without disrupting others. Reading data is push-based, clients receive data as soon as it is provided by the sensors with no delay. Clients do not subscribe to reading values of a particular sensor, rather they subscribe to values of a particular type, e.g. temperature, without regard of the underlying sensor. This provides a powerful abstraction layer, reduces complexity, eases the burden on development complexity and reduces Time To Market due to the usage of readily-available drivers for existing sensors.

Data is accompanied by metadata of SI units and is normalised to its base unit. This unifies reading different sensors of the same type, where different hardware might report values of the same type in different units; the Sensor Manager automatically takes care of converting values to their base representation.

Access policies are available to restrict sensor data access to privileged clients. While not production ready, this can be easily extended to direct or delegated ACLs, integrated with the mF2C platform, to unify access management with the platform and allow sensor owners to restrict access to their own applications or to applications of developers they trust.

An example sensor driver and example application are distributed alongside the Sensor Manager to aid development.

## 8.1. Implementation

The application is written in Golang and available on GitHub at https://github.com/mF2C/sensor-manager. There are three compilation units and one external application, each officially packaged in a separate Docker image:

- the Sensor Manager,
- the Sensor Manager Example Driver,
- the Sensor Manager Example Application and
- the Multi-client Authenticated Messaging Broker Application.

Starting with the last, which is a completely separate application, it is based on the Mosquitto messaging broker. Modified with an open-source authentication plugin, it is the component that distributes sensor data to clients and provides a standard Message Queue Telemetry Transport (MQTT) authentication interface. It is configured to listen on a WebSockets port, which is able to be integrated into Traefik, mF2C's API entrypoint.

The example driver and example application are standalone examples of components sensor driver developers or application developers must develop to interface with the Sensor Manager. They utilise all features of the mF2C Sensor Manager and the Multi-client Authenticated Messaging Broker Application.

These two components are abstracted away from an external user through the use of a reverse proxy instance, Traefik, that translates requests from a single HTTP address-based endpoint to the two backend services. This simplifies integration of this service into mF2C and external systems.

The Sensor Manager serves multiple functions:

- interfacing with the mF2C API,
- providing an authentication backend for the Multi-client Authenticated Messaging Broker Application,
- managing the sensor values published by the sensors to be read by clients.

Interfacing with the mF2C API will be described in the next section as part of the integration.

The authentication backend is accessed by both users and the Multi-client Authenticated Messaging Broker Application. Users contact the server to obtain credentials and connection parameters for sensor streams they have access to. The broker connects back to the Sensor Manager to validate access, as it does not maintain an internal access list, but delegates authentication and authorisation per-topic to the Sensor Manager through the HTTP backend.

Managing the sensor values is the core part of the Sensor Manager. Each sensor driver (explained in following sections) must conform to a specified interface over which it publishes sensor values. This is done directly to a separate authenticated MQTT topic, which the sensor value handling component of the Sensor Manager reads. Metadata about the sensor, such as the SI quantity of value published, its unit and timestamp are included with each reading; this is then processed by the Sensor Manager to normalise values and make them available under the appropriate topic for each type of sensor with the correct authentication parameters.

Internally, SQLite is used for the authentication database. This avoids using a separate server for the database, simplifying deployment and minimising resource usage, while retaining good-enough performance and availability. The application's main artifact, the Sensor Manager docker container, is only 7 MB in size, which could be further reduced to 4 MB by using the scratch Docker base image, but this would make debugging more difficult. Memory consumption is also minimal, not exceeding 10 MB of active memory even with multiple sensors and clients publishing and reading values.

## 8.2. Integration

Sensor Manager primarily interacts with mF2C to spawn sensor driver containers and get the information about the attached sensors. The goal of this interaction is to launch sensor driver containers in response to sensors appearing in the listing made available by the Resource Manager in CIMI API's device-dynamic resource. The data and control flow of the sensor driver, along with how it is created, is displayed in Figure 7.

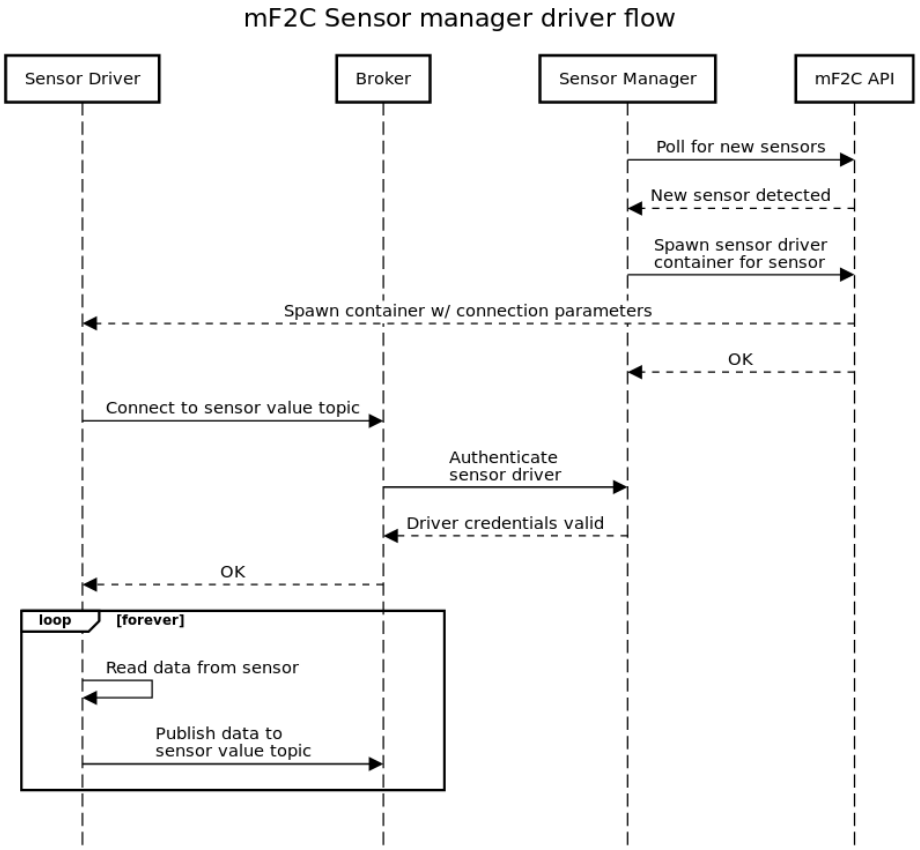


**Figure 7. Sensor Manager sequence diagram: sensor driver**

The most important endpoint mF2C provides is device-dynamic in the CIMI schema. The resource categorisation module identified the hardware sensors connected to a device, along with their type and connection parameters. This information is then used by any available sensor drivers, spawned by the Sensor Manager, to connect to each sensor and read its values.

Sensor drivers are distributed as Docker images that read sensor data and publish it back to the Sensor Manager. An example sensor driver written in Golang is included in the sensor driver repository and published as a Docker image.

Environment variables, passed to the sensor driver container, define the connection credentials for both the sensor and the Multi-client Authenticated Messaging Broker Application, where it must publish values in a predefined JSON schema. This interface is described in detail in the official mF2C documentation.

The client applications connect through the WebSockets protocol wrapping MQTT communication; this can be done with any MQTT library supporting WebSockets. It must provide the correct credentials for access to the topic, previously obtained by contacting the Sensor Manager's management interface. Values are streamed in JSON format and include the reading timestamp, value and SI unit.

Figure 8 shows the complete flow of the client application.

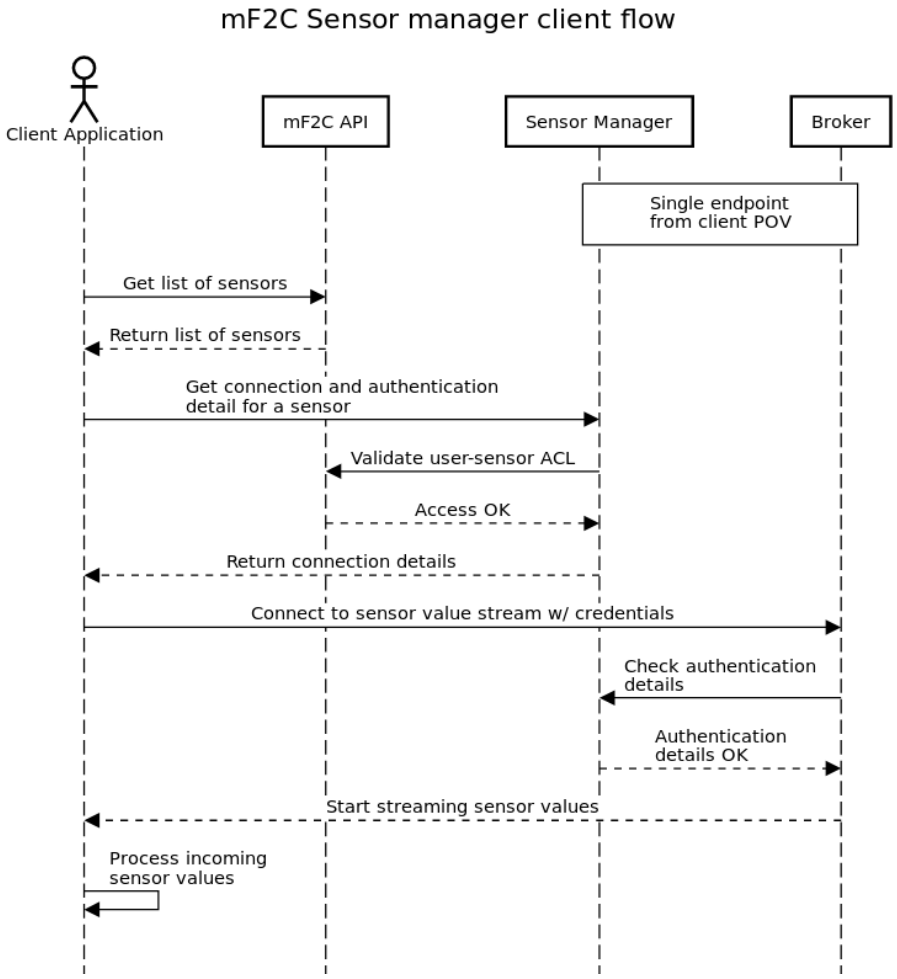


**Figure 8. Sensor Manager sequence diagram: client**

The Sensor Manager repository includes a script that deploys the Sensor Manager onto mF2C as an mF2C application. As this is a regular mF2C application (of the docker-compose type), this can be modified to be a globally registered mF2C service, with the ability to launch it via the "Launch" button in the mF2C dashboard.

## 8.3. Technical maturity assessment

The Sensor Manager is at TRL 4. We have confirmed it functions as intended and is integrated into the mF2C platform, but only lab-style tests have been conducted thus far.

## 8.4. Lessons learnt

A useful feature not currently present in the Sensor Manager is the ability of transparently accessing data from other nodes in the fog area. This could be done by manual shoveling of data between brokers or with direct federation of an enhanced version of the Multi-client Authenticated Messaging Broker Application. This would allow even better access to sensor data in the fog, enhancing the mF2C fog concept.

Detecting attached sensors through the Resource Manager has proved to be an issue. Sensor detection is currently solved by manual user input of sensors present on a device at registration.

# 9. Conclusions

In this deliverable, we have presented the implementation details of mF2C Agent Controller component, which is an update of previous deliverable D3.5 [7]. Unlike the previous deliverable, where the details of the functions each component offered to other components were described by means of use case diagrams, here we have gone deep into the details. We have described the different modules of each component, as well as the integration of these modules with others inside or outside the Agent Controller.

Besides that, describing the integration of Agent controller modules, we have also described the integration of other components, not being part of the initial design of the Agent controller. One of these components is the Sensor Manager, with a functionality that, despite being beyond the scope of the mF2C platform, was found desirable for the Smart Boat use case. Other components included in this deliverable are the security and event manager components, and the GUI/Dashboard.

For all the described components, we summarized the lessons learnt from their implementation and integration, providing the necessary input, jointly with deliverables D4.6 (Platform Manager) [4] and D4.8 (Interfaces) [3], for IT-2 validation in WP5.

## References


[1] "D3.4 Design of the mF2C Controller Block (IT-2)," June 2019. [Online]. Available: https://www.mf2c-project.eu/d3-4-design-of-the-mf2c-agent-controller-block-it-2/.

[2] "GitHub mF2C repository," [Online]. Available: https://github.com/mF2C.

[3] "D4.8 mF2C interfaces (IT-2)," September 2019. [Online]. Available: (to appear).

[4] "D4.6 Platform Manager blocks and microagents integration (IT-2)," September 2019. [Online]. Available: (to appear).

[5] "D4.7 mF2C interfaces (IT-1)," December 2017. [Online]. Available: https://www.mf2c-project.eu/d4-7-m12/.

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

[7] "D3.5 mF2C Agent Controller block integration (IT-1)," December 2017. [Online].

[8] C. Perera, Y. Qin, J. C. Estrella, S. Reiff-Marganiec and A. V. Vasilakos, "Fog computing for sustainable smart cities: A survey," *ACM Computing Surveys,* vol. 50, no. 3, pp. 32:1-32:43, 2017.

[9] B. Dorsemaine, J. P. Gaulier, J. P. Wary, N. Kheir and P. Urien, "Internet of things: a definition & taxonomy," in *IEEE 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, 2015.

[10] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang and R. Ranjan, "Fog computing: survey of trends, architectures requirements, and research directions," *IEEE Access,* vol. 6, pp. 2169-3536, 2018.

[11] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco and E. Theodoridis, "Smartsantander: Iot experimentation over a smart city testbed," *Computer Networks,* vol. 61, pp. 217-238, 2014.

[12] O. Hahm, E. Baccelli, H. Peters and N. Tsiftes, "Operating systems for low-end devices in the internet of things: a survey," *IEEE Internet of Things Journal,* vol. 3, no. 5, pp. 720-734, 2016.

[13] S. Singh and I. Chana, "A survey on resource scheduling in cloud computing: Issues and challenges," *Journal of Grid Computing,* vol. 14, no. 2, pp. 217-264, 2016.

[14] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management,* vol. 23, no. 3, pp. 567-619, 2015.

[15] "D2.5 mF2C Security/Privacy Requirements and Features," December 2018. [Online]. Available: https://www.mf2c-project.eu/d2-5-mf2c-security-privacy-requirements-and-features-it-2/.

[16] "D2.7 mF2C Architecture (IT-2)," January 2019. [Online]. Available: https://www.mf2c-project.eu/d2-7-mf2c-architecture-it-2/.

[17] "D3.2 Security and Privacy aspects for the mF2C Agent Controller Block (IT-2)," January 2019. [Online]. Available: https://www.mf2c-project.eu/d3-2-security-and-privacy-aspects-for-agent-controller-it-2/.

[18] "D4.2 Security and privacy aspects for the mF2C Platform Manager block (IT-2)," January 2019. [Online]. Available: https://www.mf2c-project.eu/d4-2-security-and-privacy-aspects-for-platform-manager-it-2/.

[19] E. Rescorla, "HTTP over TLS," May 2000. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2818.txt.

[20] "D3.1 Security and privacy aspects for the mF2C Agent Controller Block (IT-1)," June 2017. [Online]. Available: https://www.mf2c-project.eu/d3-1-m6/.

[21] "D4.1 Security and privacy aspects for the mF2C Platform Manager block (IT-1)," June 2017. [Online]. Available: https://www.mf2c-project.eu/d4-1-m6/.

[22] A. Cohen, E. Lear and B. Weis, "Help managing the growing number of Things on our networks has arrived," March 2019. [Online]. Available: https://www.ietf.org/blog/mud/.