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

D4.3 Design of the mF2C Platform Manager block components and microagents (IT-1)

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

Work Package	**WP4, mF2C Platform Manager block design and implementation**
Due Date:	*M9*
Submission Date:	*30/09/2017*
Version:	*0.8*
Status	*Final*
Author(s):	*Anna Queralt (BSC)*
Reviewer(s)	*Roberto Bulla (ENG)* *Laura Val (WOS)*

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	21/07/2017	Completed section 4.4 and initial workflows definition	Rosa M. Badia, Toni Cortes, Ana Juan, Alexander Leckey, Daniele Lezzi, Anna Queralt, Román Sosa
0.2	14/07/2017	Completed sections 3.1 and 3.3	Ana Juan, Román Sosa
0.3	22/08/2017	Completed full section 6 and updates in workflows	Cristovao Cordeiro, Román Sosa
0.4	25/08/2017	Completed sections 1, 2, 3, 4, 5 and 7	Alexander Leckey, Daniele Lezzi, Anna Queralt
0.5	14/09/2017	Additional content and updates in sections 2, 3, 4 and 5	Jens Jensen, Alexander Leckey, Anna Queralt, Román Sosa
0.6	15/09/2017	Draft ready for 1st revision	Anna Queralt
0.7	22/09/2017	Several updates on workflows and text modified accordingly	Rosa M. Badia, Toni Cortes, Ana Juan, Alexander Leckey, Daniele Lezzi, Anna Queralt, Román Sosa
0.8	27/09/2017	Incorporated reviews and comments from ENG, STFC, UPC and WOS	Anna Queralt

Table of Contents

List of figures

List of tables

Executive Summary

This document developed by the mF2C project describes an initial design of the mF2C Platform Manager (former Gearbox) and the microagents for iteration IT-1.

The main focus of this document is the design of each of the Platform Manager internal components, in such a way that their expected functions are fulfilled. Special attention is paid to the interactions between different components; either in the same or in different agents, to provide a design that takes into account the requirements and goals of the mF2C architecture. Also, a standard approach is used to design interfaces for the communication between different agents. Finally, a design for the microagents is provided according to the current architecture.

The outcome of this document is a detailed design of the Platform Manager components and functionalities, including illustrative workflows that will also be essential for the next stages of the development, and a design for the interfaces and microagents in mF2C.

1. Introduction

1.1. Introduction

In D2.6 the requirements and architecture of mF2C were defined, identifying the Platform Manager (PM) and Agent Controller (AC) as the main building blocks of mF2C agents. The functionalities required within these blocks were also identified, as well as an initial set of interactions between components.

In this document, each of the identified functionalities in the PM are designed in detail, paying special attention to the interactions between the different functionalities, either in the same or in different agents, in order to perform the expected functions. These interactions have been identified by means of the definition of a set of workflows that depict a number of representative scenarios for the mF2C platform, and will be essential for the upcoming integration of the different components.

This document also provides the design of the interfaces that will enable the communication between components.

Finally, and according to the mF2C architecture in D2.6, this document establishes the concept of microagent that was envisioned in the initial project proposal.
The structure of this document is as follows:

- Section 1 describes the aim and the context of this document.
- Section 2 provides an overview of the PM functionalities and security aspects.
- Section 3 details the design of the Service Orchestration component.
- Section 4 details the design of the Distributed Execution Runtime component.
- Section 5 details the design of the Telemetry and Monitoring component.
- Section 6 details the design of the interfaces.
- Section 7 details the design decisions taken on microagents.

1.2. Purpose

The objective of this deliverable is to provide a design of each of the functionalities in the Platform Manager of an agent or a microagent, according to the architecture defined in D2.6 [1], as well as the design of the interfaces that will enable communication between components, either in the same or in different agents.

1.3. Glossary of Acronyms

Acronym	Definition
AC	mF2C Agent Controller
CIMI	Cloud Infrastructure Management Interface
CPU	Central Processing Unit
DE	Decision Engine
DMTF	Data Management Task Force
EBNF	Extended Backus-Naur form
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
JSON	Javascript Object Notation
NIC	Network Interface Card
O/S	Operating System
PM	mF2C Platform Manager
QoB	Quality of Business
QoS	Quality of Service

REST	Representational state transfer
SDK	Software Development Kit
SDN	Software Defined Networking
SDS	Software Defined Storage
SLA	Service Level Agreement
SLO	Service Level Objective
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
VM	Virtual Machine
XML	eXtensible Markup Language

Table 1 Acronyms

2. Platform Manager Description

The Platform Manager is one of the two main building blocks that comprise the agent entity, along with the Agent Controller. The main responsibilities of this block are collecting all application or service[1] requests, translate them into internal calls, orchestrating the different tasks and performing resource optimal allocation. At run time it controls service execution, according to define SLAs, and other supporting tasks like telemetry collection and management.

In its interaction with other agents, it is defined as a global entity that works as a controller, when it is managing agents in lower layers, or that acts as a receiver of control data, when it is being managed by agents from upper layers (described in deliverable D2.6 [1]).

2.1. Survey of main Platform Manager Functionalities

The Platform Manager is the block responsible for the orchestration of services based on the compute, storage and network resources and using a full-stack monitoring system, which receives telemetry data from different sources. This block is also responsible for coordinating the distributed execution of services and applications within the mF2C infrastructure.

As shown in Figure 1, the Platform Manager is divided into three main components according to these responsibilities: Service Orchestration, Distributed Execution Runtime, and Telemetry.

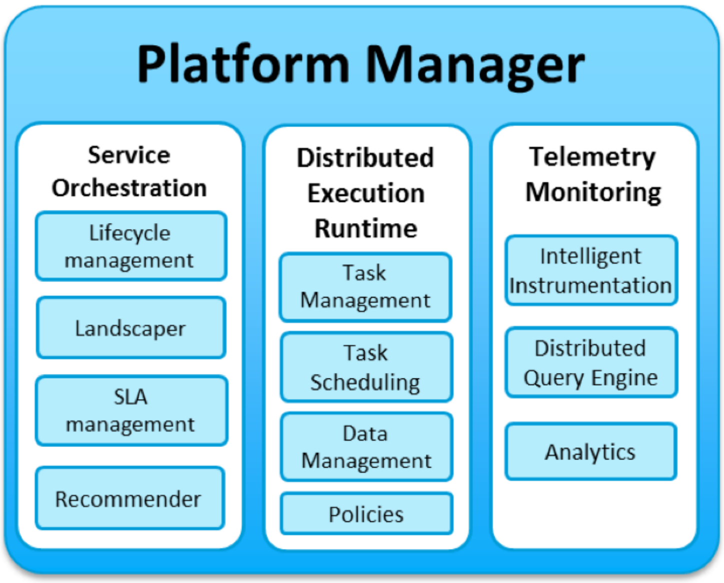

Figure 1. Platform Manager components and functionalities

The *Service Orchestration* component is responsible for allocating services to the most suitable resources producing optimal performance and efficient use of those resources, and for controlling the different phases of a service being executed. Its main functionalities are:

- Lifecycle management: controls the different phases of the execution of a service, namely the initialization, operation, and termination phases.
- Landscaper: expresses both the physical and logical infrastructure topology of the different layers of the F2C hierarchy in the form of a graph.
- SLA management: guarantees that user's expectations in terms of QoS and QoB are satisfied.

[1] The terms "application" and "service" will be used indistinctly throughout the document, meaning a piece of software whose execution is managed by the mF2C platform.

- Recommender: derives scheduling recipes from a set of heuristics and models, and provides them to the Lifecycle management.

The second component in the PM is the *Distributed Execution Runtime*, which is responsible for coordinating the execution of end-user applications or services following a task-based approach. The main functionalities in this component are:

- Task management: identifies the tasks that compose an application and detects the dependencies between them.
- Task scheduling: selects the most appropriate resources to execute each task and monitors task execution to keep the application flow consistent.
- Data management: stores and provides access to the data used by services or applications.
- Policies: define how tasks are assigned to resources, and how data is distributed.

Finally, the *Telemetry and Monitoring* component is decomposed in the following functionalities:

- Intelligent instrumentation: provides the telemetry collectors that capture the raw data from the system's hardware and software, and derives metrics.
- Distributed query engine: allows for the querying of telemetry data.
- Analytics: derives heuristics and models from the instrumentation data and the Landscaper, and provides them to the Recommender.

The design of each of these functionalities will be detailed in the following sections.

2.2. Security Provisioning

As the PM is responsible for a range of planning and orchestration functionality, it is potentially vulnerable to a number of attacks against the system. For example, an attacker could register a malicious service as a resource, and make it look attractive to the system (low cost, high QoS). If the service then fakes the billing/accounting records, or steals information from the tasks it runs, it could make a lot of money before a human operator discovers it and intervenes.

On one hand there is nothing new here: after all, e-commerce works the same way. Resources must be trusted to deliver what they promise, and to process accounting and billing fairly. In order to do this, there are financial controls on participants, communicated to the end user through EV certificates (extended validation). In practice, when users access a web site protected by such a certificate, they get a nice green security icon on the address bar, pretty consistently across all browsers; if the EV extensions are absent, the bar will (typically) be blue, still indicating security but visibly different, thus hopefully warning people not to trust the site with their credit card details.

In e-commerce this level of protection is not achievable to smaller companies as it would be far too expensive. Instead, they have their financial services hosted by another company which is registered and whose entire business model is to support their customers' financial transactions (e.g. WorldPay, etc.). In the context of IoT, there is no possibility of securing everything with high assurance certificates; billing must, as in cloud services, be handled centrally by a trusted service. Nevertheless, accounting records must be PROTECTED or higher (following the terminology in the Security Policy defined in D3.1 [2]). As a corollary, it also follows that trusted devices must have means of securely (a) authenticating themselves to each other, (b) signing data, (c) encrypting data for others and (d) decrypting data for themselves. This scenario usually suggests a PKI (Public Key Infrastructure) but leaves the question of bootstrapping the PKI. In terms of capabilities of devices, most of the PM functionality described in this deliverable already requires computational capabilities beyond the smallest devices (such as microcontrollers), so implementation of a PKI for devices that require it should be computationally feasible.

The threat scenarios were described extensively in D2.4 [3]. The data security policy was described in D3.1 [2], and applied to the mF2C use cases in D4.1 [4]. We have already discussed how mF2C functionalities like the Landscaper and Recommender need to check their data, in order to prevent an attacker from publishing a malicious service. Data management obviously also needs to follow the data policy set out in D3.1 [2]. For accounting purposes, usage data may have to be marked confidential (i.e. PRIVATE), in order to protect the user's privacy. Beyond the authentication, it may be necessary also to have *delegation* in the traditional sense of access rights, that the user has rights to a service or resource, and these are delegated to the tasks, to enable them to access (or create) data on behalf of the user. Delegation may also be needed in the pragmatic sense, that a token or delegated credential is needed for a service to "impersonate" the user, albeit usually within a restricted time, and possibly with restricted rights, in order to allow a scheduler to run tasks, and the lifecycle management to manage the orchestration of tasks.

3. Service Orchestration Design

The Service Orchestration component will be responsible for allocating services to the most suitable resources producing optimal performance and efficient use of those resources. Service placements will be dependent on the analysis of historical invocations of similar service descriptors, and configuration of available resources in real time, including SLA, and QoS.

The following sections are intended to design the components inside the Service Orchestration block in order to support the mF2C architecture for IT-1, extending the content of D2.6 [1], and focusing on the interactions with other components of the platform in order to facilitate their upcoming integration.

3.1. Lifecycle Management

The Lifecycle Management component manages the lifecycle of a service to be executed by the mF2C infrastructure, enabling the control of the diverse phases in the execution (initialization, operation and termination as explained in D2.6 [1]). A service at this level corresponds to an application executed by the Distributed Execution Runtime.
The Lifecycle Manager performs this action coordinating the different components in both the Platform Manager and the Agent Controller. The lifecycle of a service in the mF2C platform is shown in Figure 2.
The main operations provided by the Lifecycle Management are:

- submit a service
- stop a service
- re/start a service
- terminate a service

A submitted service is processed on the Leader PM of the level where the user is. The Lifecycle needs to orchestrate the following tasks:

1. Find the resources where the service tasks are going to be executed. This step makes use of the Recommender and Landscaper.
2. Initiate the SLA management, according to the expected SLA for the service. The SLA is accepted if the needed resources from the AC are being shared by the user. This step makes use of the SLA Manager in the leader, and the User Management of each resource (the latter is not shown in the diagram, and will be further explained in section 3.3).
3. Start the allocation of resources and make the needed deployment of runtime framework. The deployment may include the deployment of the task itself. This step uses the Service Management in the AC of each node where a task will be executed, under request of its corresponding PM.
4. The execution of the service is finally delegated to the Distributed Execution Runtime in the leader.

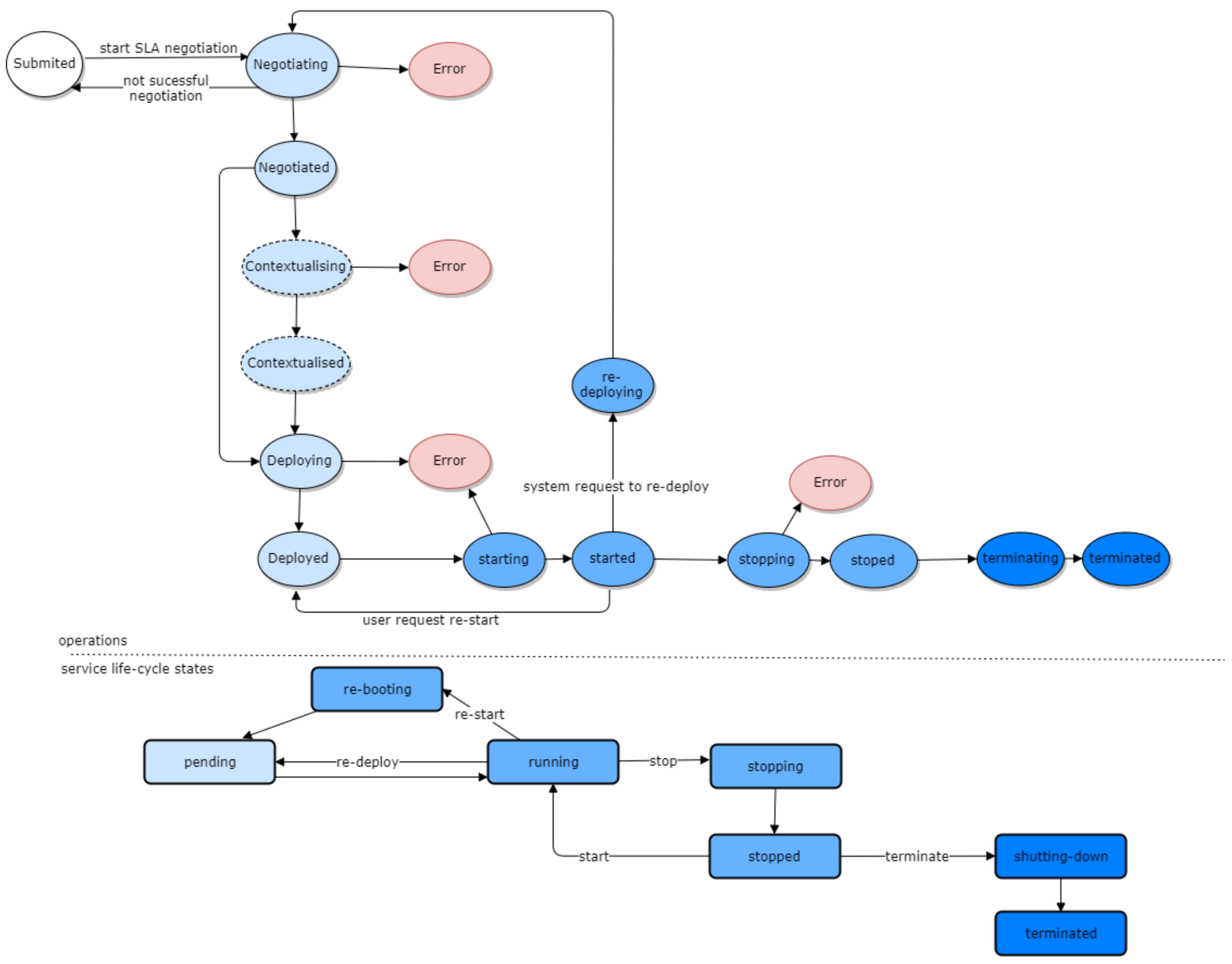

Figure 2. mF2C service lifecycle

The diagram in Figure 3 illustrates the interactions between components needed to perform these functions, corresponding to the initialization phase of a service.

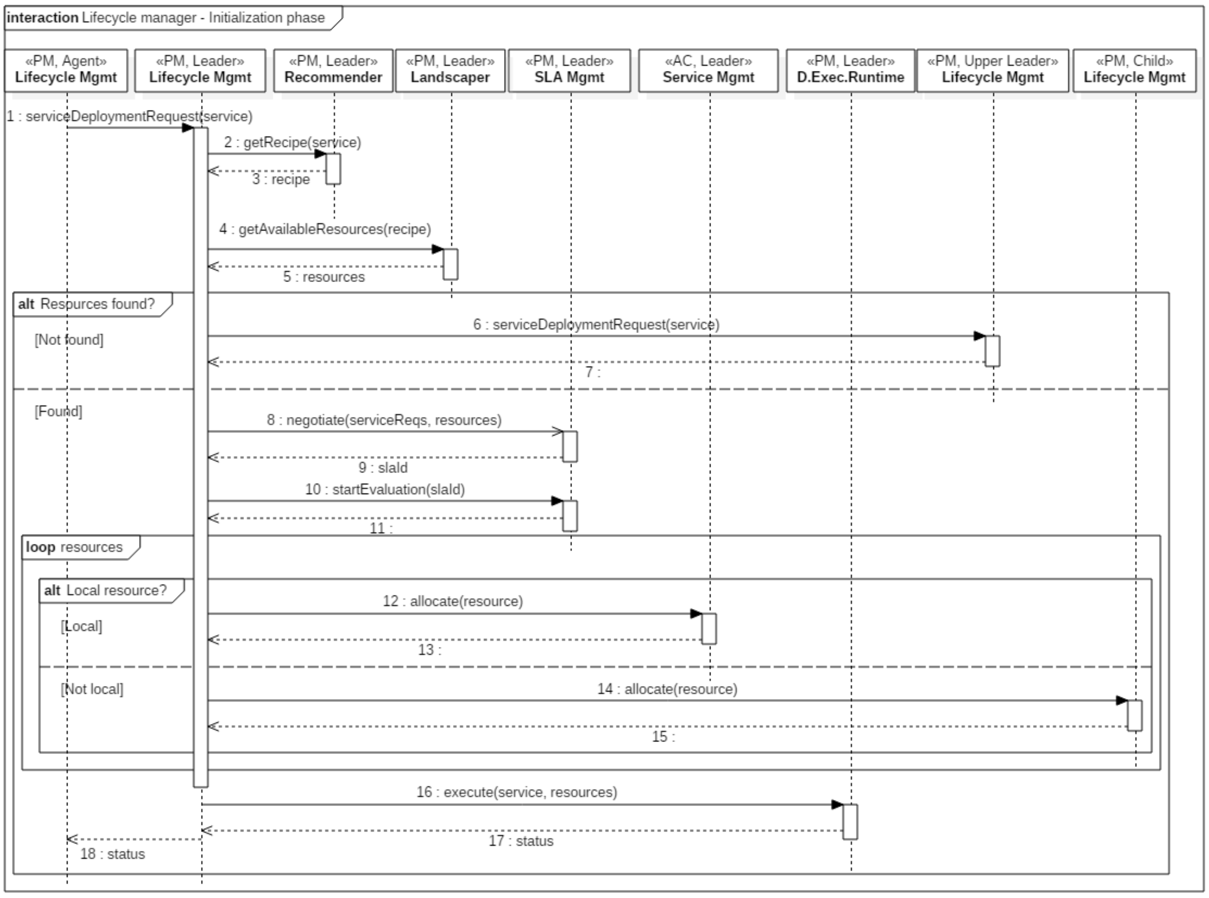
interaction Lifecycle manager - Initialization phase
«PM, Agent» Lifecycle Mgmt
«PM, Leader» Lifecycle Mgmt
«PM, Leader» Recommender
«PM, Leader» Landscaper
«PM, Leader» SLA Mgmt
«AC, Leader» Service Mgmt
«PM, Leader» D.Exec.Runtime
«PM, Upper Leader» Lifecycle Mgmt
«PM, Child» Lifecycle Mgmt
1 : serviceDeploymentRequest(service)
2 : getRecipe(service)
3 : recipe
4 : getAvailableResources(recipe)
5 : resources
alt Resources found?
[Not found]
6 : serviceDeploymentRequest(service)
7 :
[Found]
8 : negotiate(serviceReqs, resources)
9 : slaId
10 : startEvaluation(slaId)
11 :
loop resources
alt Local resource?
[Local]
12 : allocate(resource)
13 :
[Not local]
14 : allocate(resource)
15 :
16 : execute(service, resources)
17 : status
18 : status

Figure 3. Lifecycle Management - Service initialization

When a user requests a service to be run by mF2C, the PM of the agent receiving the user request forwards it to its leader. As a first step, the Recommender is asked for an optimal deployment configuration to run the service. This configuration is returned as a *recipe*, which states characteristics of resources –not specific instances of resources– best matching the service demands. Then, a list of resources matching the recommendation is obtained from the Landscaper. If the Landscaper is not able to find the appropriate resources in the current cluster, then the service deployment request is forwarded upwards and the process starts again from the upper cluster. Otherwise, if the current cluster has resources matching the recipe, a service SLA is negotiated as will be seen in section 3.3. Afterwards, the resources are allocated. For those resources in the current agent, the Service Management at the AC is contacted to perform the allocation (i.e., deployment) of the resources. For resources in children, the allocation request is forwarded to their PM, which, in turn, forwards them to their Service Management module in the AC. Finally, the Distributed Execution Runtime starts the execution of the service in the resources provided.

During the operation phase, the user can decide to stop the execution of the service, and restart it later. These actions, when received by the leader, are forwarded to the Distributed Execution Runtime to make the status change effective, as shown in Figure 4. When a start or a stop request is received, the Landscaper is notified of the event so that it can update its information about the deployed services. SLA violations may be used in future iterations for adaptation. In this case, a new recipe is requested from the Recommender and the same steps followed in the initialization phase are executed.

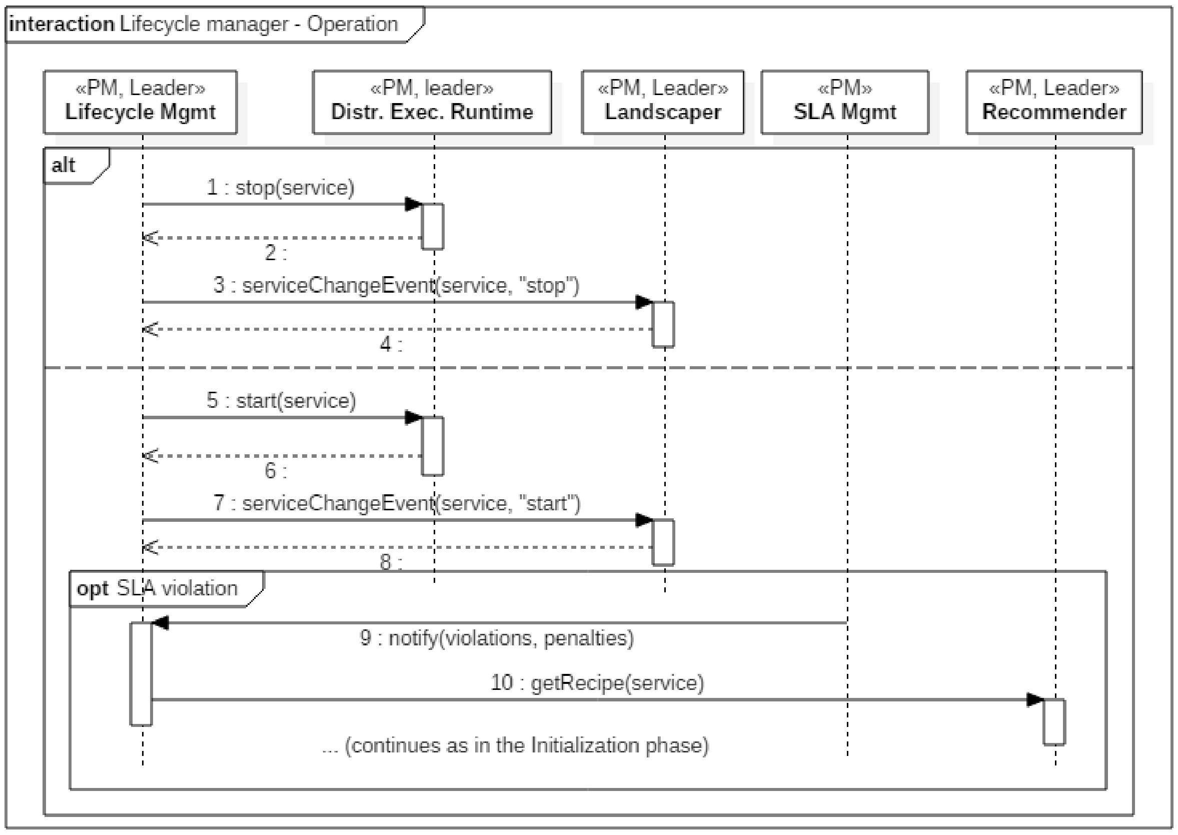
interaction Lifecycle manager - Operation
«PM, Leader» Lifecycle Mgmt
«PM, leader» Distr. Exec. Runtime
«PM, Leader» Landscaper
«PM» SLA Mgmt
«PM, Leader» Recommender
alt
1 : stop(service)
2 :
3 : serviceChangeEvent(service, "stop")
4 :
5 : start(service)
6 :
7 : serviceChangeEvent(service, "start")
8 :
opt SLA violation
9 : notify(violations, penalties)
10 : getRecipe(service)
... (continues as in the Initialization phase)

Figure 4. Lifecycle Management - Service operation

Finally, in the termination phase, the Lifecycle Manager deallocates all the resources used by the service and terminates the SLA agreement, as shown in Figure 5. The termination workflow is started either by a user who wants to abort execution of their services, or when the execution of the services finishes and the Lifecycle Manager is notified of the termination. If the resources are local to the leader, the Service Management is contacted to perform the deallocation. If the resources belong to a child, its Lifecycle Management in the PM is contacted so that it can forward the request to its Service Management module in the AC. Termination is also possible when the user's rights to run services are revoked (or they run out of credits, if it's prepaid), although this functionality may be out of scope for IT-1.

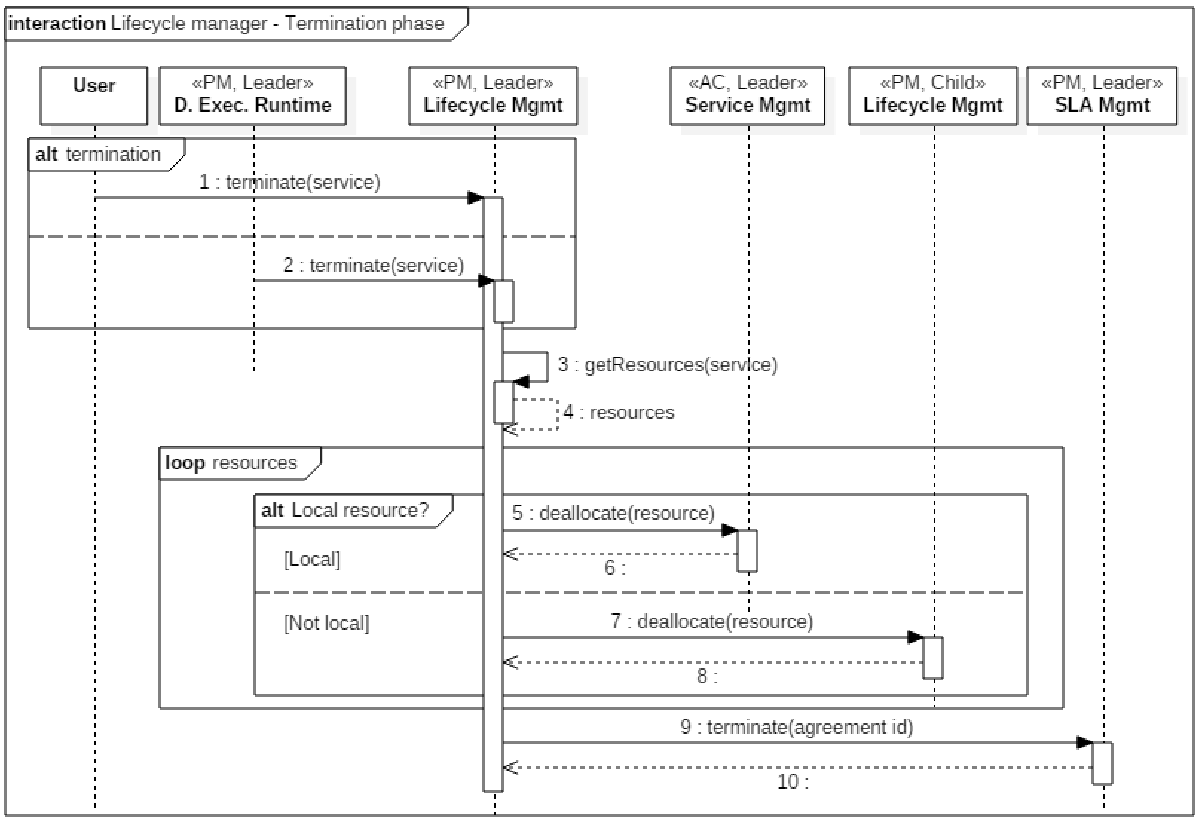
interaction Lifecycle manager - Termination phase
User
«PM, Leader» D. Exec. Runtime
«PM, Leader» Lifecycle Mgmt
«AC, Leader» Service Mgmt
«PM, Child» Lifecycle Mgmt
«PM, Leader» SLA Mgmt
alt termination
1 : terminate(service)
2 : terminate(service)
3 : getResources(service)
4 : resources
loop resources
alt Local resource?
[Local]
5 : deallocate(resource)
6 :
[Not local]
7 : deallocate(resource)
8 :
9 : terminate(agreement id)
10 :

Figure 5. Lifecycle Management - Service termination

The Lifecycle Management component is comprised of the following modules:

- API. It is the interface of the Lifecycle Manager to the rest of components.
- Controller. It is responsible for the behaviour of the component, in charge of the communication with the rest of mF2C components.
- Service Registry. It keeps a registry of the services in the mF2C platform being handled by this instance.

3.1.1. Communication with the Agent Controller

According to the defined workflows, the Lifecycle Management functionality in the PM of an agent interacts with the following functionalities in the AC of the same agent:

Service Management. The Service Management of an agent is contacted in order to perform the allocation of needed resources for the execution of a service, and the deployment of the needed software.

3.1.2. Communication with the Platform Manager

The Lifecycle Management needs to interact with the following functionalities in the PM:

Recommender. The Lifecycle Management contacts the Recommender in order to retrieve an optimal resource configuration where to execute an application.

Distributed Execution Runtime. The Distributed Execution Runtime is the component in charge of orchestrating the execution of the service tasks. The Lifecycle Management contacts it when the resources and software needed for the execution have been allocated, in order to start the execution of the service.

Landscaper. It is contacted to maintain the status of resources and to retrieve a set of resources that match the recommendation.

SLA Management. The SLA Management is in charge of observing the SLA of the execution at the service level. The Lifecycle Management contacts the SLA Management when a service is submitted, in order to prepare the assessment of the expected QoS.

3.2. Landscaper

The Landscaper module generates and stores the physical and logical infrastructure topology of the different layers of the F2C hierarchy. This includes the cloud, fogs and IoT devices layers.
As shown in Figure 6, the first step for the component is to query the list of agents within its current cluster and send requests to retrieve a list of resources that are available. Afterwards, the Landscaper requests the set of deployed services from the Lifecycle Manager. The next step is to then generate a usable model divided into three layers:

- Service Layer: store services/applications currently deployed.
- Virtual Layer: stores SDx (software-defined) entities eg, containers, VM's, vSwitches, SDN, SDS.
- Physical Layer: these are mapped to the actual physical resources of the compute node such as memory, disks, NICs, CPUs, etc.

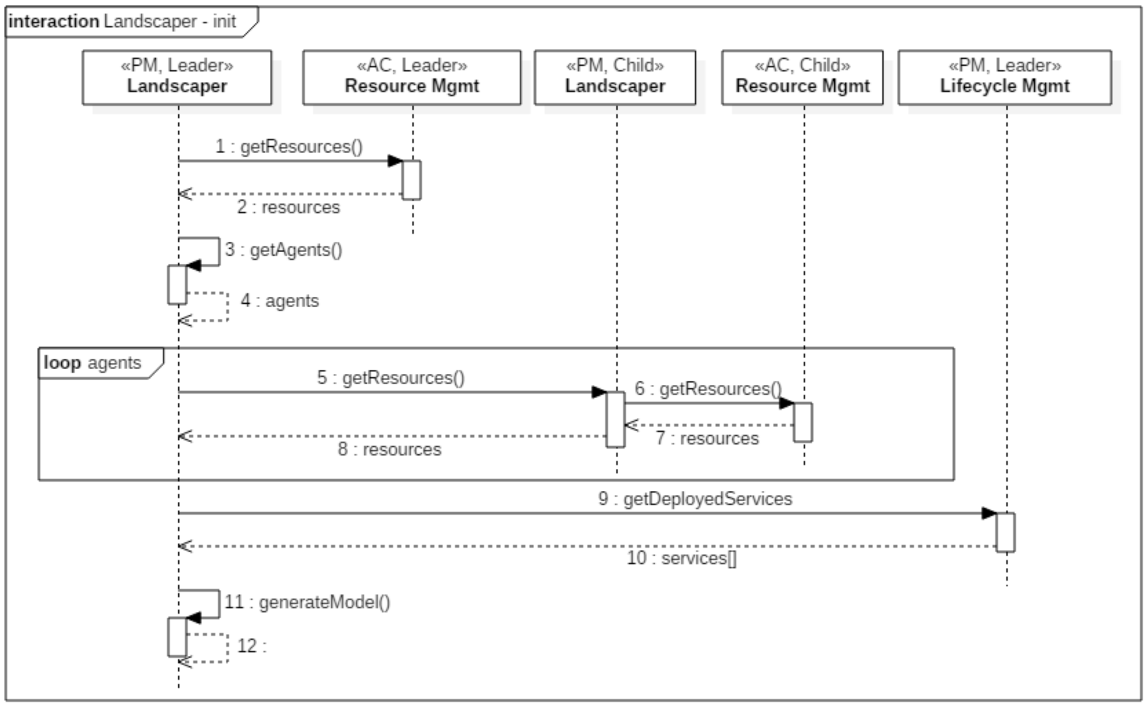

Figure 6. Landscaper – Initialisation

The Landscaper will store historical values so that a landscape snapshot for that cluster can be generated for a given date/time. This allows for performance analysis on historical invocations to be executed by mapping the associated telemetry. There are two main events of interest:

- Physical Resource: the Resource Management component of each Agent Controller will generate a resourceChangeEvent which will be sent to the Landscaper. This will contain all the relevant information for that physical resource.

- Deployed Services: Lifecycle Management will raise a serviceChangeEvent that contains all the information for that service (new deployed service, service termination, re-start, etc).

These change events allow the Landscaper component to make the appropriate changes to the persisted model, archiving the old values as shown in Figure 7.

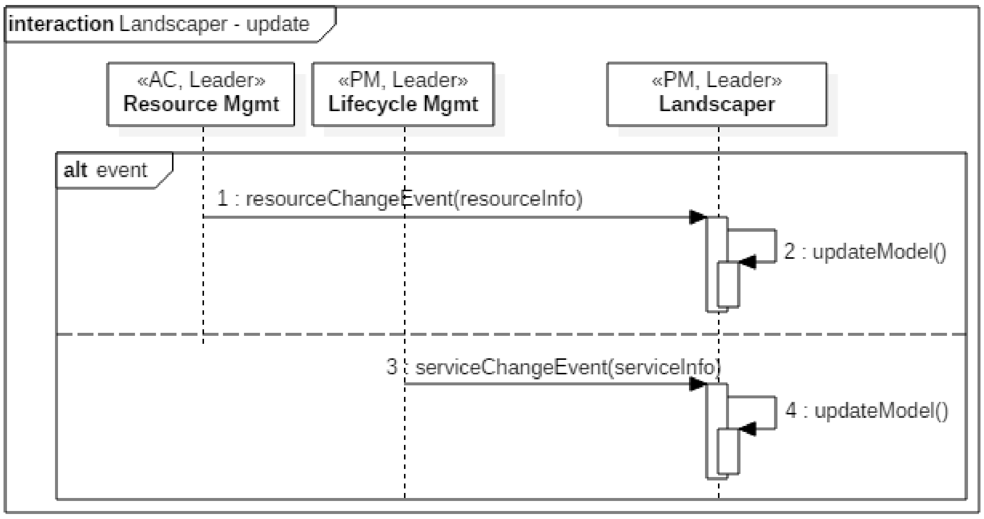

Figure 7. Landscaper – Updating the model

3.2.1. Communication with the Agent Controller

According to the workflows, the Landscaper functionality in the PM of an agent interacts with the following functionalities in the AC of the same agent:

Resource Management. The Landscaper needs to communicate with the Resource Management module to retrieve the list of compute resources available on that node.

3.2.2. Communication with the Platform Manager

The Landscaper needs to interact with the following functionalities in the PM:

Lifecycle Management: The Landscaper requires the list of currently deployed services so that it can map them to the underlying infrastructure that they are executing on.

Landscaper: To query the Resource Management of each Agent Controller, the Landscaper must first contact the Landscaper of the agent to be queried so that it forwards the request to the AC.

3.3. SLA Management

The SLA Management component is responsible for managing the SLAs between the parties involved in a service on the mF2C platform: the platform and the platform users. The component is in charge of generating, storing and observing the electronic documents that describe the expected service level of the execution of a service.

The agreements contain functional and non-functional terms that describe the service being delivered. In mF2C, we are mostly interested in non-functional terms, where a Service Level Objective (SLO) is defined as a constraint on a metric. A non-fulfilled constraint for a metric datum is considered a violation. An SLO violation implies that the agreement has been violated, and this can imply business penalties (e.g., a discount) if they have been defined in the agreement.

The list below shows an example of the schema proposed for agreements on mF2C.

```
{
   type: "AGREEMENT",
   id: "agreement-mf2c",
   name: "Example of agreement",
   provider: "mf2c-atos",
   client: "mf2c-client",
   creation: "2017-08-15T00:00:00",
   expiration: "2018-08-15T00:00:00",
   guarantees: [
      {
         name: "ResponseTimeTerm",
         constraint: "responsetime LT 1500",
         penalties: [
            {
               type: "discount"
               value: "10"
               unit: "%"
            }
         ]
      }
   ]
}
```

The lifecycle of an agreement is shown in Figure 8, which includes the main operations provided by the SLA Management:

- Create an agreement
- Stop an agreement evaluation
- Re/start an agreement evaluation
- Terminate an agreement

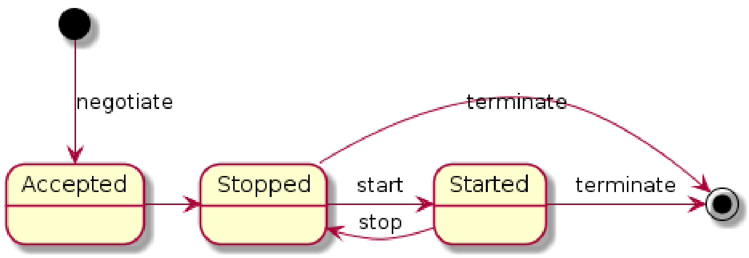

Figure 8. States of an agreement

The creation of an agreement is initiated by the Lifecycle Management when a service is initialized (see Figure 3). As shown in Figure 9, the SLA Management functionality receives the SLA description (involved parties and SLOs), and builds the formal SLA document, saving it and returning an identifier that will allow to access the document. The SLA Management checks, with the User Management block of each device where a resource is being allocated, that the allocation is allowed according to the sharing constraints expressed by the user. Otherwise, the SLA is rejected.

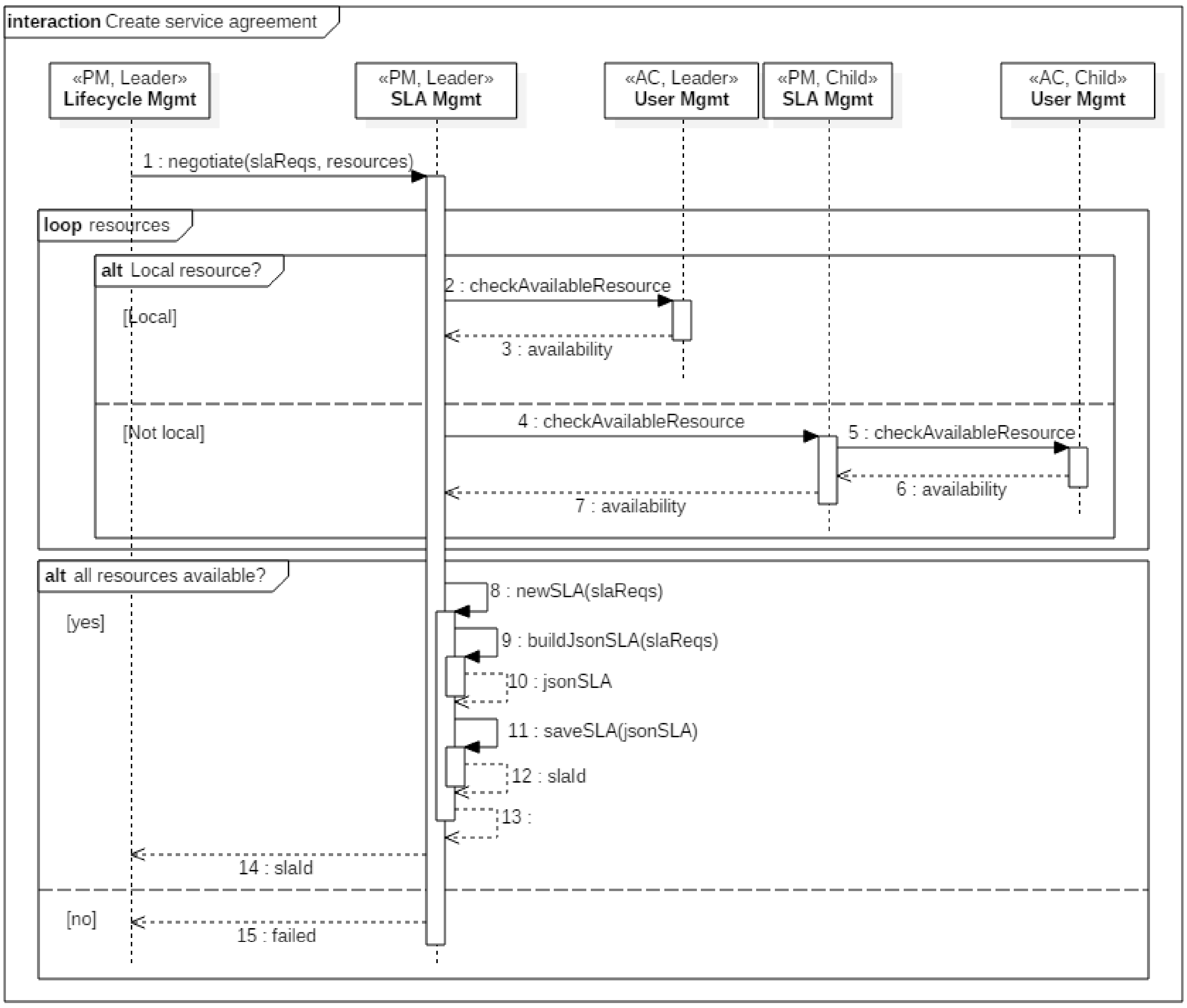

Figure 9. SLA Management – Create agreement

The evaluation of an SLA is initiated when a service is started, which is notified by the Lifecycle Management. SLA Management subscribes to the Telemetry and Monitoring component to be notified about telemetry data, and the agreement is marked as started, as shown Figure 10.

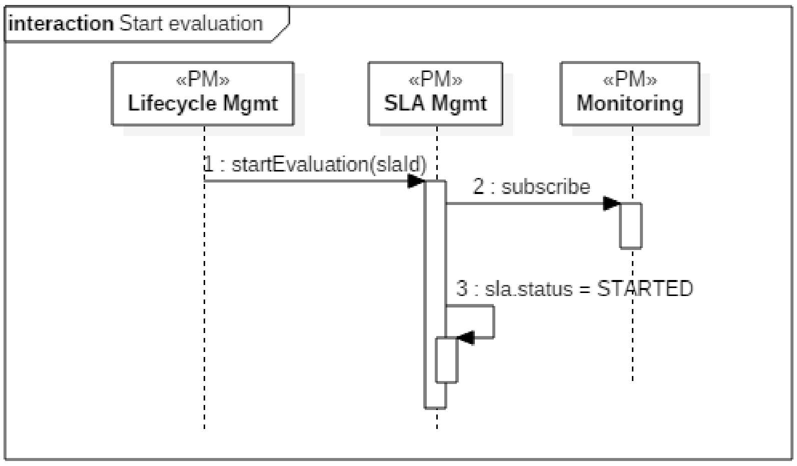

Figure 10. SLA Management – Start evaluation

The same process is followed to stop an evaluation, marking the SLA as stopped.

The evaluation of an agreement is initiated when data is received from Monitoring. Found violations and penalties are informed to the Lifecycle Management (Figure 11), which can then react accordingly.

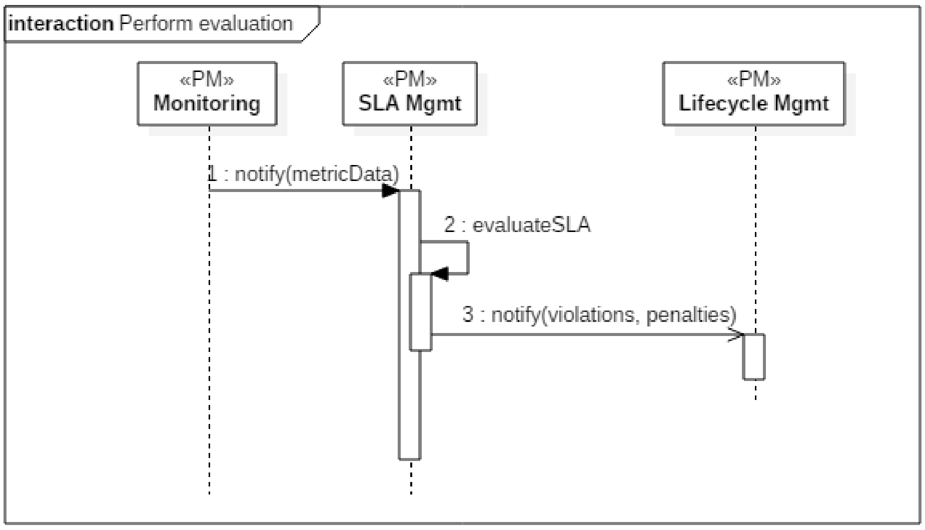

Figure 11. SLA Management – Evaluate agreement

Besides the functionality offered to other components by the API, the SLA Management component periodically evaluates the SLOs contained in the agreement. This evaluation is supported by the Telemetry and Monitoring. The detected violations and penalties are sent as messages to interested observers.
The SLA Management component is comprised of the following modules:

- API. Offers the functionalities explained above.
- SLA Registry. Keeps a registry of the agreements being handled by this instance and their status.
- Evaluator. Performs the periodic evaluation of the started agreements.
- Monitoring Adapter. Translates monitoring data from the Monitoring component to an internal representation. This makes it possible to use a different Monitoring with minor modifications.
- Notifier. This module is responsible for sending violations and penalties to external components.

3.3.1. Communication with the Agent Controller

User Management. An interaction with the User Management functionality in the AC is needed in order to check if a resource (or part of it) is allowed to be shared or not.

3.3.2. Communication with the Platform Manager

Lifecycle Management. It manages the agreement lifecycle according to a service lifecycle. It also receives notifications about violations, which could be used to perform adaptation actions.

Telemetry Monitoring. The Monitoring supplies the needed monitoring data to the SLA in order to assess the respective SLOs.

3.4. Recommender

The Recommender module will store the heuristics and models derived from analyzing service deployments. It is intended that these will take the form of decision trees. Before the Lifecycle Management deploys a service, it will need to query the Recommender for a deployment recipe that maps to that particular service type (eg, Service Descriptor), as shown in Figures 3 and 4 .

It is assumed that recipes are constantly changing and evolving as newer analysis takes place. Also, there may not be an exact match of available hardware or software configuration at a given moment, so sometimes a best fit may be required on that occasion.

3.4.1. Communication with the Agent Controller

No communication with any AC functionalities currently planned.

3.4.2. Communication with the Platform Manager

Analytics: The Recommender will provide an API that allows the Analytics module to store the output of the service performance analysis.

Lifecycle Manager: The Recommender will provide an API to retrieve an optimal hardware/software configuration to deploy a given service descriptor.

4. Distributed Execution Runtime

The Distributed Execution Runtime (DER) participates in the operation phase of the services executed in the platform, and is responsible for coordinating the execution of end-user services or applications in the mF2C infrastructure. The DER implementation takes as its basis the COMPSs programming framework [5]. COMPSs considers applications as composites of invocations to pieces of software encapsulated as methods called Core Elements (CE). The main purpose of the runtime toolkit is to orchestrate the execution of CE invocations (tasks), and thus to fully exploit the available computing resources. In mF2C the adaptations of COMPSs include enhancements in the way the resources are managed by the runtime due to their volatility that imposes requirements to data management and work balancing, and integration with the security framework.

The following sections detail how the COMPSs runtime is adapted to support the mF2C architecture for IT-1. These sections extend the content of the architecture document D2.6 [1] focusing on the connections with other components of the PM and of the AC.

The workflow in Figure 12 shows the interaction between the components within the Distributed Execution Runtime, as well as with other external components. The different steps in the diagram will be explained throughout the following subsections.

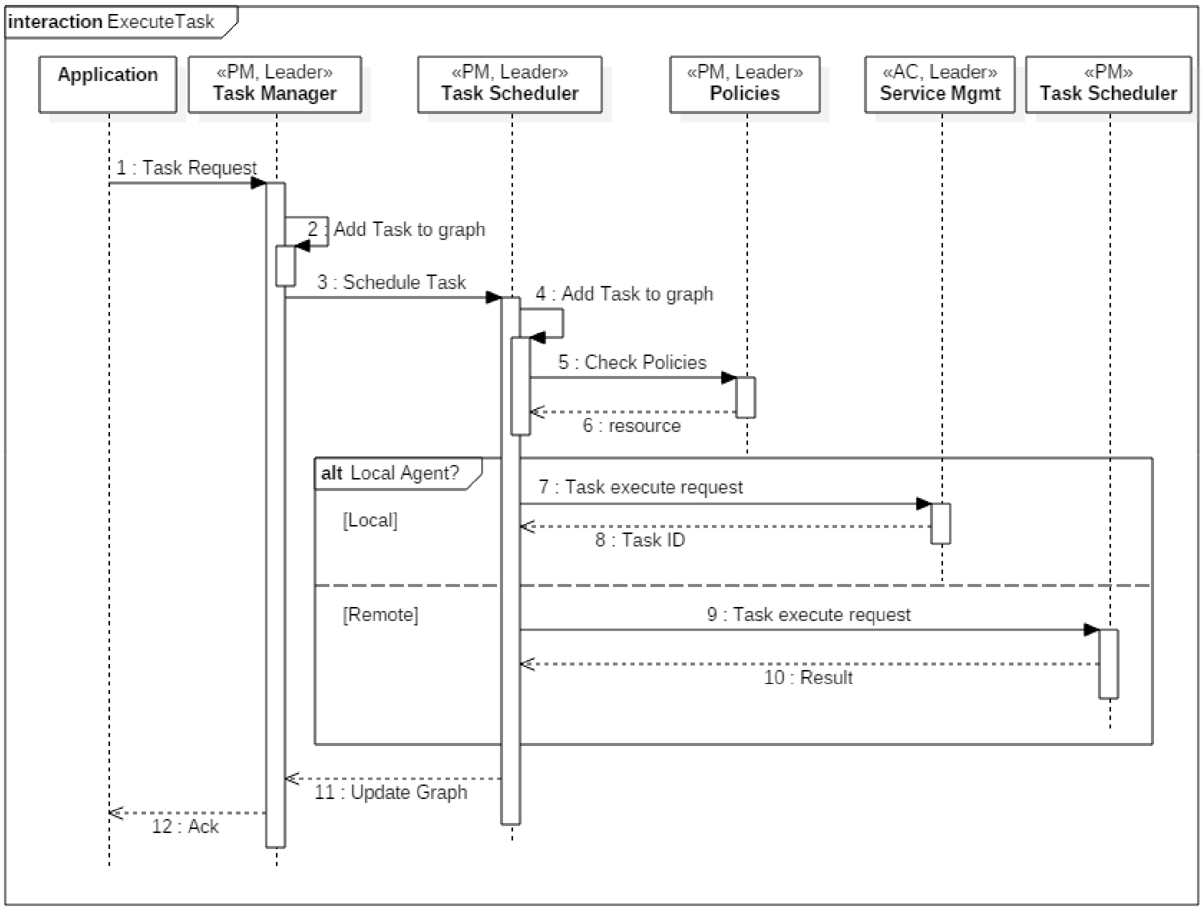

Figure 12. Distributed Execution Runtime – Execute task

4.1. Task Management

In the COMPSs model, applications share computing resources and, potentially, data values; therefore, the runtime library is split into two parts. The front-end of the runtime, instantiated in every application, manages the private aspects of the applications: monitors accesses to private pieces of data, such as objects, and detects the CE invocations. A single instance of the back-end (Runtime Process), running as an independent process shared by all the running applications in the device, manages all the aspects shared amongst the applications, including computing resources (CPU, GPU, nearby nodes or VM instances on the cloud) and data (currently only files, but we envisage managing access to databases and Content Providers).

To monitor the data accessed from each task and the data dependences among tasks, the runtime processes the parameters of each task. The Private and Public Data Registers, respectively located on the front-end and back-end of the runtime, record the accessed data values and assign a unique identifier for each version of the value. Once all the accessed values are registered, the task is scheduled for execution.

4.1.1. Communication with the Agent Controller

No communication with any AC functionalities currently planned.

4.1.2. Communication with the Platform Manager

Lifecycle Management. In the current architecture a service is instantiated by the Lifecycle Management component that can provide a first list of resources based on the recipes obtained from the Recommender (Figure 3). The runtime can decide during the execution to request more resources to the Service Management.

4.2. Task Scheduling

To decide which resources host the execution of a task, the runtime is based on the concept of Computing Platform: a logical grouping of computing resources capable of running tasks. The decision is made on the Decision Engine (DE), which is agnostic to the actual computing devices supporting the platform and the details of how to interact with them.

The DE acts as a metascheduler in the runtime deciding which of the available resources is best suited to run a task. To this aim, the information provided by the Policies component of the PM is used to create an updated list of resources, and for each one the runtime maintains information on the expected end time, energy consumption and economic cost of the execution. Once the target resource is selected, the scheduling of the task is delegated to the actual implementation that it is responsible for monitoring the data dependencies of the task and scheduling the execution of the task on its resources. For example if a Cloud Platform is selected, then a provider specific connector is used to instantiate VMs.

In the COMPSs programming model, the tasks of an application share data through objects in memory and files. A relevant feature of the runtime is the ability to orchestrate applications whose tasks share big amounts of data through a storage API that interoperates with different backends. In particular, the integration with a persistent object storage platform is the more relevant to the mF2C project. This integration allows an application to make objects persistent, that is, objects initially allocated in memory can be backed by the persistent storage layer. From that point on, changes to the object will be forwarded to the backend as well. On the other hand, objects that were made persistent can be retrieved by other applications. This enables interactive sharing of data between applications that run concurrently.

4.2.1. Communication with the Agent Controller

Service Management: the interaction with the Service Management is needed to instantiate the task on the selected resource. This step requires first deploying and configuring the environment in which the task is to be deployed. For example, if the target resource provides virtual machines, the interaction includes the request for the instantiation of the VM through a VM image and the contextualization of the machine to properly receive the runtime commands to execute the tasks. In the case of container based environments, a Docker container will be deployed and configured. All these steps of deployment and configurations are performed by the internal components of the Service Management of the AC, as described in D3.3 [6].

Categorization: the diagram in Figure 13 depicts the interaction of the runtime with the Categorization component in order to retrieve the characteristics of the resources that are made available to the runtime to schedule the tasks. As depicted, the list of resources is stored in a local configuration file of the runtime that periodically reads it to have an updated picture of the available resources.

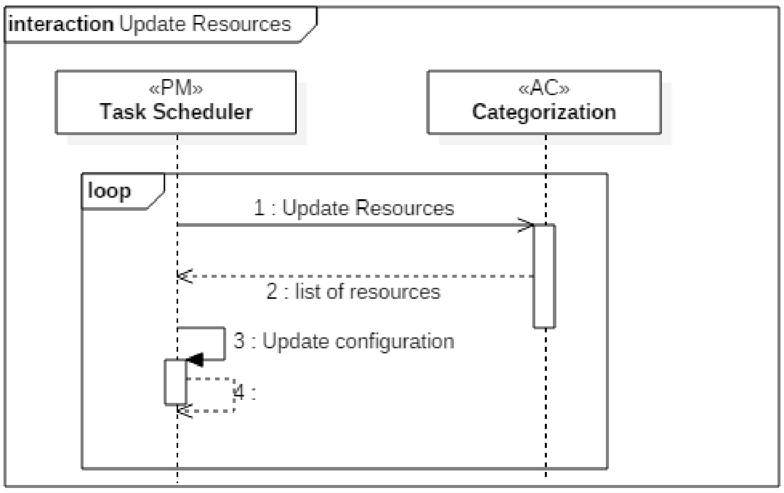

Figure 13. Distributed Execution Runtime – Update resources

4.2.2. Communication with the Platform Manager

Policies: as explained before, the Task Scheduler needs a list of resources to distribute the execution of the tasks. In COMPSs a configuration file is used to provide the runtime with a list of providers (endpoints of cloud providers or cluster management middleware) that is used to match the constraints of a task with the available resources. In mF2C the list of resources is maintained through the AC Categorization component, while the PM Policies information is used to identify the appropriate resource. As depicted in the diagram in Figure 12, a task can be executed on the same node as the leader agent, or if a different node is selected as worker, the Task Scheduler forwards the task to the PM of that node. As explained in the next paragraph, the Policies could also drive the selection of the execution node based on data locality.
Data Management: the Task Scheduler will contact the data manager in the same PM in order to perform the following operations:

- Get an object by its identifier (objectID). It will send the objectID in string format, and a reference to the persistent object with the corresponding objectID will be returned. The task scheduler uses this operation to get all the objects that are needed for the execution of a task.
- Get the locations of an object. Given an objectID, the data manager returns a list with the locations of the corresponding object.
- Replicate an object. The Task Scheduler will send the objectID and the destination node, and the object will be replicated in the specified destination. When the node that contains the

object is too busy to execute a task, the task scheduler uses this operation to replicate the object in the node selected for execution and thus take advantage of parallelism, if possible according to the task dependencies. This operation is used for tasks that do not modify the object.

- Create a version of an object. It will send the objectID and the destination node, and the object will be copied in the specified destination, with a new objectID. The purpose of this operation is analogous to the replica, but for tasks that modify the object.
- Consolidate a version of an object. Given the objectID of a version, the data manager "commits" the changes made to this version in the original object that was versioned to guarantee the consistency of the parallel execution.

4.3. Policies

The policies in the execution environment define how tasks are assigned to resources, and how data is distributed in the storage platform. These policies help to identify the most appropriate resource to execute a task, taking into account its current state and the data it contains. Also, data policies such as replication or distribution policies will be taken into account to improve performance and reliability. The runtime will prioritize policies according to the current state of the execution.

4.4. Data Management

The data management functionality of the PM is in charge of providing a global view of the data, the communication between different managers, and enforcing the data management policies. This manager will be implemented by an extension of the logic module in dataClay [7].

Data objects should be referenceable and accessible from any component in the mF2C infrastructure. In order to offer this global view, we need to implement a global namespace to reference objects. For this reason, whenever an object is created, dataClay assigns an objectID that is unique in the whole infrastructure. We have decided to offer a flat name space where the ID of an object is a 16 byte number created using the UUID algorithms offered by Java. This option is highly scalable as no centralized interaction is needed, as required by mF2C. Nevertheless, if a different naming scheme is needed (i.e some kind of hierarchical name space), dataClay can be modified to support it. Object creation is shown in Figure 14.

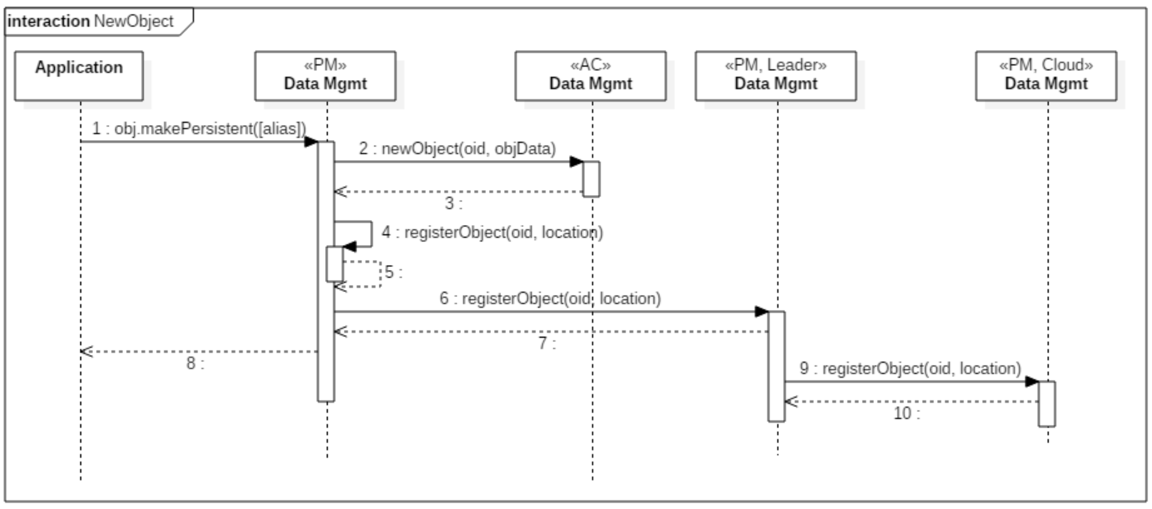
interaction NewObject
Application
«PM» Data Mgmt
«AC» Data Mgmt
«PM, Leader» Data Mgmt
«PM, Cloud» Data Mgmt
1 : obj.makePersistent([alias])
2 : newObject(oid, objData)
3 :
4 : registerObject(oid, location)
5 :
6 : registerObject(oid, location)
7 :
8 :
9 : registerObject(oid, location)
10 :

Figure 14. Data Management – New object

As can be seen in the diagram, an object that is created, either by an application or by some mF2C functionality, is sent to the Data Management functionality of an agent to be registered and cached. The leader in the same cluster is immediately informed of the new object and, eventually (e.g when there is connection, or when the agent is not busy), this information will get to the cloud.

UUIDs are good for machines but not for humans, thus we also offer a secondary namespace where users can associate strings to objectIDs, and thus be able to refer to objects using its textual name. This textual name (referred to as an "alias" in Figure 14) also implements a flat address space. If more complex textual names are needed, this could also be easily changed to partition them (i.e. per user, per cluster, …). It is important to note that this textual namespace does not need to include all objects. For instance, it may make sense for a collection to have a textual name to be able to retrieve it or share with other parts of the service, but not all objects in the collection need to have a textual name because they will be accessible by iterating over the collection.

Once the application has the objectID we need to find its current location (that can change over time, or be replicated in different resources) to be able to access it. To find this location, the data management component in the PM will keep a table (that will be stored persistently) with the information (objectID, locations) for all the objects in its resources, or in resources underneath in case of a leader.

To avoid unnecessary searches at higher levels when searching for a data object, platform managers will cache object metadata (objectID, locations). Given that this metadata is either immutable or easy to detect if erroneous, it can be cached in any PM that has seen it. The objectID is immutable and, although the locations can change, if a location is erroneous in a cached piece of metadata (because the object has been moved since the last time it was accessed by the PM), the error will be detected when requesting the object. In that case we can discard the cached information and request the information from the root platform manager that should have the means to find the correct versions of all data locations. This process is shown in Figure 15.

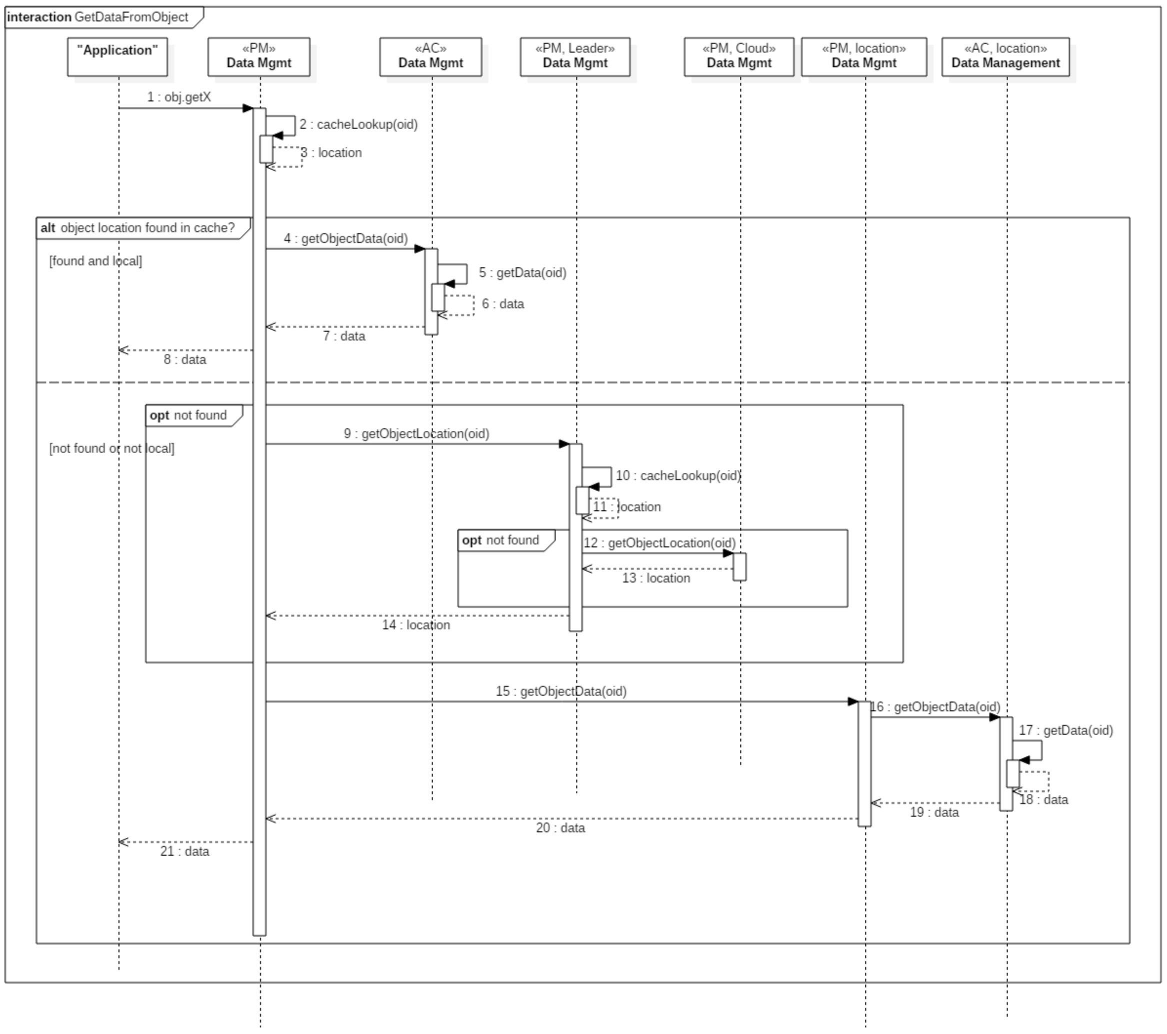

Figure 15. Data Management – Get data from object

It is important to understand that once a PM knows the location of an object, all accesses to the object will go directly to the node where the object is located following a flat data communication. The same sequence of interactions is performed when the data held by an object is updated.

In order that the system is aware of the structure of the stored objects, the classes that define these objects need to be registered in all the agents that will manage their objects. In particular, classes related to mF2C metadata should be present in all agents, while classes related to a specific application or service running on top of the infrastructure should be deployed only in those affected agents. In both cases, the Data Manager in the cloud knows about all the classes registered in the system to be able to deal with the dynamicity of the resources. This process is shown in Figure 16.

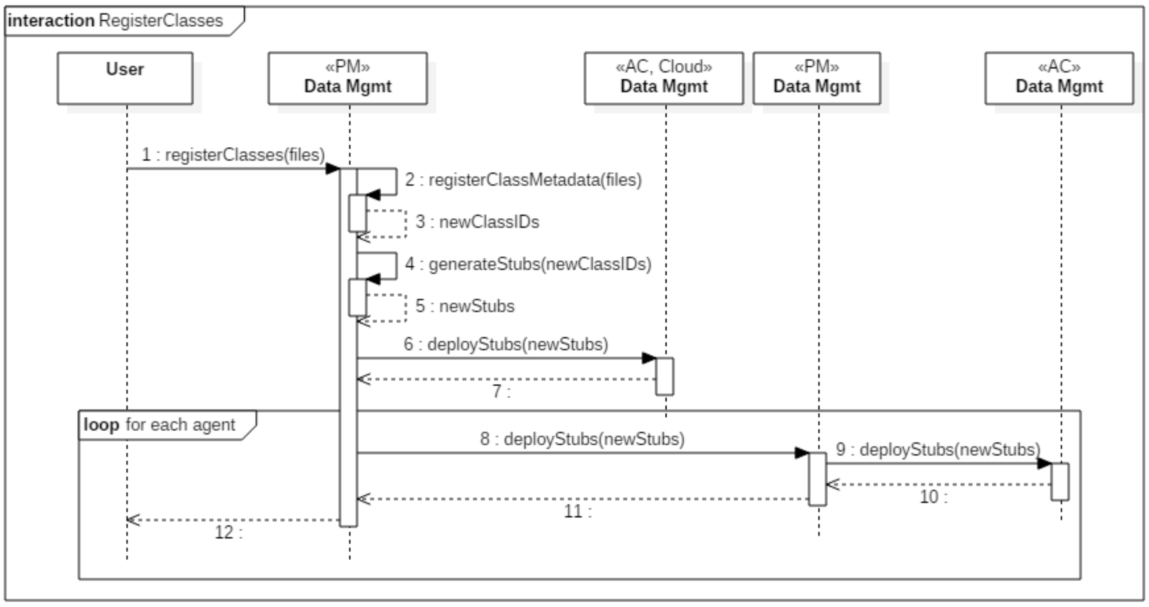
interaction RegisterClasses
User
«PM» Data Mgmt
«AC, Cloud» Data Mgmt
«PM» Data Mgmt
«AC» Data Mgmt
1 : registerClasses(files)
2 : registerClassMetadata(files)
3 : newClassIDs
4 : generateStubs(newClassIDs)
5 : newStubs
6 : deployStubs(newStubs)
7 :
loop for each agent
8 : deployStubs(newStubs)
9 : deployStubs(newStubs)
10 :
11 :
12 :

Figure 16. Data Management – Register classes

Also, when a new resource with storage capacity joins the infrastructure (Figure 17), the classes that hold mF2C data need to be deployed so that the new resource is able to perform the mF2C functionalities.

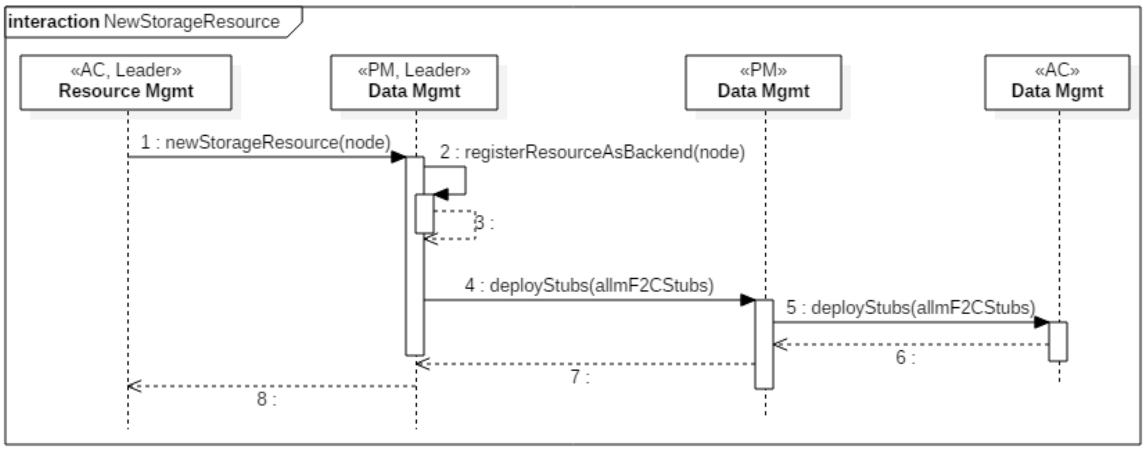
interaction NewStorageResource
«AC, Leader» Resource Mgmt
«PM, Leader» Data Mgmt
«PM» Data Mgmt
«AC» Data Mgmt
1 : newStorageResource(node)
2 : registerResourceAsBackend(node)
3 :
4 : deployStubs(allmF2CStubs)
5 : deployStubs(allmF2CStubs)
6 :
7 :
8 :

Figure 17. Data Management – New storage resource

Another important task of the data manager in the PM is to be able to enforce data movements and replications. On the one hand, the data manager will offer the possibility for other functionalities to decide when data needs to be copied or migrated to other nodes. For instance, the distributed execution runtime may decide that in order to increase locality, it will request a copy of a given data to a given node or cluster.

On the other hand, it will also be in charge of applying global policies with respect to data movement and replication from the policy manager. For instance, this manager will be able to react to policies such as make sure that there are always two replicas in the system or that moveable data is moved to the cluster that most recently used it.

4.4.1. Communication with the Agent Controller

Data manager: the data manager in the PM will contact the data manager in the AC in order to perform the following operations:

- Create a new object. It will send the objectID and the required data, and the object will be created in the agent controller.
- Get the data (the whole object or part of it). It will send the objectID and information on the portion to be accessed, and the data manager in the AC will recover it from its persistent devices and return it.
- Modify the data (the whole object or part of it). It will send the objectID and information on the portion to be modified, and the data manager in the AC will modify it and make the modifications persistent.

Resource manager: the data manager in the PM will be contacted by the resource manager in its AC in the following cases:

- Whenever a new resource willing to store data for the system joins, a message will be sent by the resource manager in the AC to the data manager in the PM in order to guarantee that this new resource is known by the storage system and to deploy all needed classes.
- Whenever an existing resource with persistent data leaves the system, the data manager in the PM will be informed so it can be removed from the list of potential resources.

4.4.2. Communication with the Platform Manager

Data manager northbound: the data manager in PM will contact the data manager in the PM of the leader (northbound) in order to perform the following operations:

- Register a new object to make it visible to the rest of the cluster. It will send the objectID, the location, and optionally an alias, and the object will be registered in the metadata information with all the other objects in the cluster.
    - Eventually, if a given object needs to be accessed from outside the cluster, the data manager in the PM of the leader will forward this registration also to the higher level (cloud in IT-1) to make it available from any component in the system.
- Get the location of an object. If a data manager needs to access an object that is not in the current node, and it has no cached information about it, it will contact the northbound PM to get the location of the object. Then it will cache this information to avoid having to make the same request in the future.
    - Eventually this request may be forwarded to the higher PM (the cloud in IT-1) if the object is outside the current cluster.

Data manager southbound: the data manager in PM will contact the data manager in the PMs on the cluster (southbound) in order to perform the following operations:

- Deploy a class to all nodes that may need it once the class is registered in the cloud. This deployment will require sending the stubs of the class to all nodes that may store objects of that class.

5. Telemetry and Monitoring

The Telemetry and Monitoring component is responsible for measuring service performance across the distributed infrastructure it is running on. By increasing the resolution and simultaneously monitoring the entire system stack, it generates metrics that allows us to review issues in context. The modules of this component are divided into three main components. The Intelligent Instrumentation will provide the telemetry collectors and aggregators capturing the raw data, adjusting the frequency depending on differing factors. The Distributed Query Engine will provide a centralised location to retrieve metrics that are actually stored throughout the local cluster. Finally, the Analytics module will review service performance metrics against the infrastructure it actually ran on to identify optimal placement recipes.

5.1. Intelligent Instrumentation

Instrumentation will be carried out by a telemetry framework and a collection of customized modules will allow the instrumentation of all the relevant elements of the deployed service on the infrastructure. The lifecycle of instrumentation will follow 3 steps:

- Collectors: Software probes will be used to capture metrics from hardware (in-band/out-of-band), from any software source: host O/S, middleware, hosted application.
- Aggregators: Captured data can be passed through filters to perform an action on the data, generate average, standard deviation, etc.
- Publishers: Processed data will be published to defined destinations, eg, file, database, message queue.

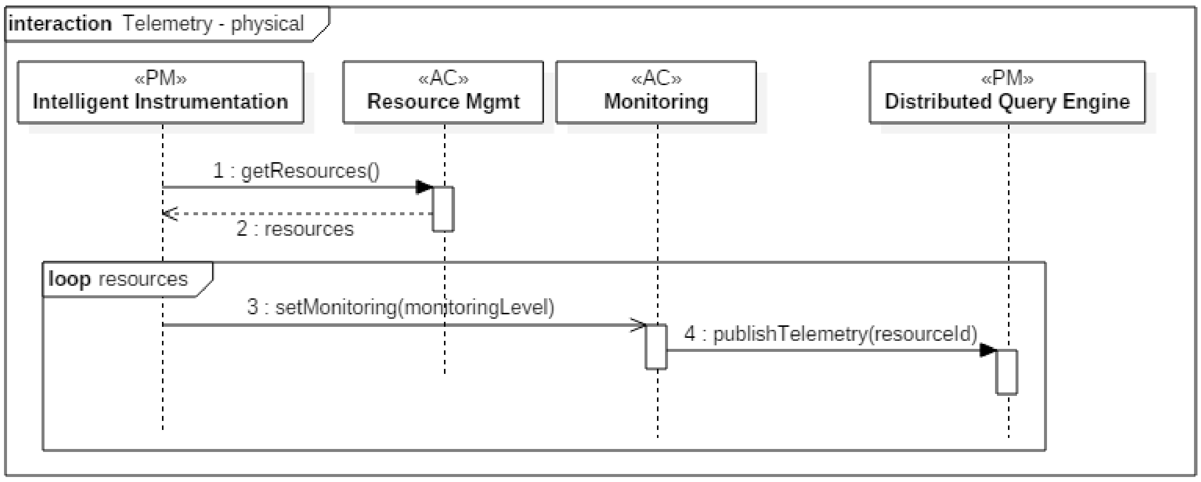
interaction Telemetry - physical
«PM» Intelligent Instrumentation
«AC» Resource Mgmt
«AC» Monitoring
«PM» Distributed Query Engine
1 : getResources()
2 : resources
loop resources
3 : setMonitoring(monitoringLevel)
4 : publishTelemetry(resourceId)

Figure 18. Intelligent Instrumentation - Setting instrumentation

It is envisioned that the level of instrumentation will vary periodically:

- The results of one probe, eg, battery life, can impact the collection of a separate probe, eg, Disk I/O, on the same device with one metric potentially causing the throttling of the other.
- Sending all collected data all of the time has the potential to overflow the system with “useless” information. Analysis may suggest sending not just aggregated values, but only when necessary
- Instrumentation can increase the measuring and transmission rate when outliers occur, eg, Tukey” statistical analysis, when anomalies are detected.

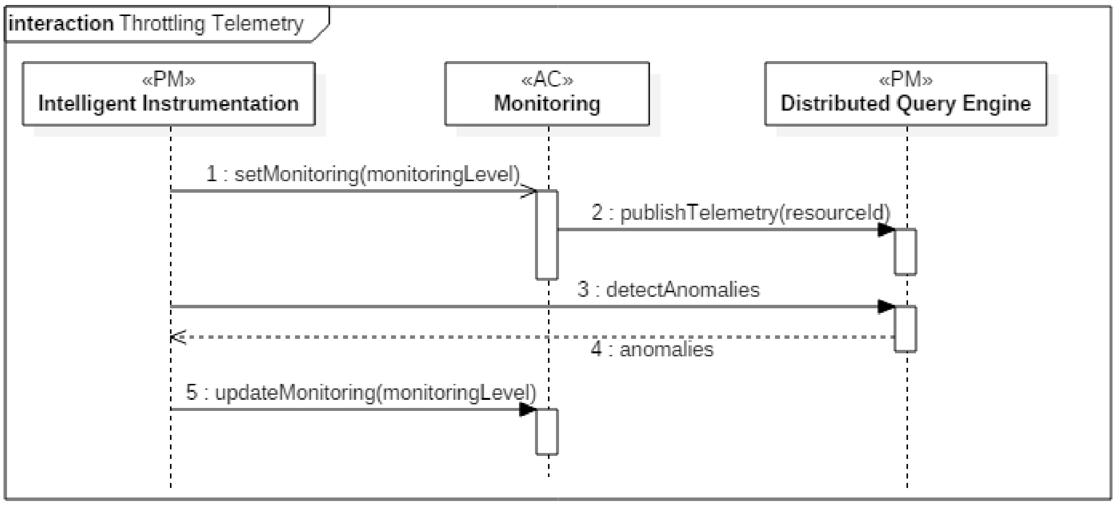
interaction Throttling Telemetry
«PM» Intelligent Instrumentation
«AC» Monitoring
«PM» Distributed Query Engine
1 : setMonitoring(monitoringLevel)
2 : publishTelemetry(resourceId)
3 : detectAnomalies
4 : anomalies
5 : updateMonitoring(monitoringLevel)

Figure 19. Intelligent Instrumentation - Adaptation

5.1.1. Communication with the Agent Controller

Monitoring: as instrumentation will only send aggregated data captured, detail metrics are only collected when necessary. So when anomalies are detected (eg, outliers), Monitoring will be contacted to increase capture frequencies and transmission rates.

5.1.2. Communication with the Platform Manager

Distributed Query Engine: as the Intelligent Instrumentation module is required to identify anomalies, it will poll metrics from the Distributed Query Engine to both analyse and identify which of the monitoring collectors need to throttle up or down their publishing frequencies

5.2. Distributed Query Engine

Monitoring collectors on each Agent Controller will capture and then potentially publish telemetry at various locations in the hierarchy of the mF2C system which would make querying of these metrics difficult. All probes will be required to register their node with the Query Engine so that it knows where to retrieve metrics for that particular node. The probe will also register with the Intelligent Instrumentation module so that it can throttle publishing frequencies depending on the output of analysis, e.g., anomaly detection, battery degradation.

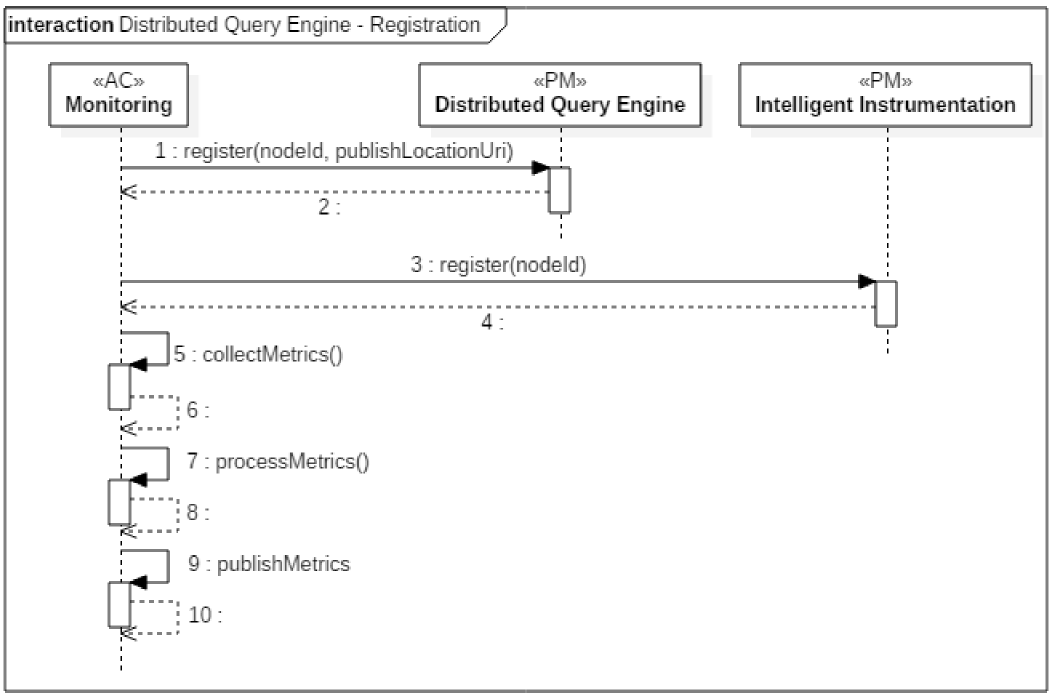
interaction Distributed Query Engine - Registration
«AC» Monitoring
«PM» Distributed Query Engine
«PM» Intelligent Instrumentation
1 : register(nodeId, publishLocationUri)
2 :
3 : register(nodeId)
4 :
5 : collectMetrics()
6 :
7 : processMetrics()
8 :
9 : publishMetrics
10 :

Figure 20. Distributed Query Engine - Registration of a monitoring probe

The Distributed Query Engine will provide a centralized API to allow for the querying of all telemetry data. All collectors will register with the query engine, notifying it of the identity of the probe and its publishing location. This provides an abstraction layer to accessing telemetry data to a single location.

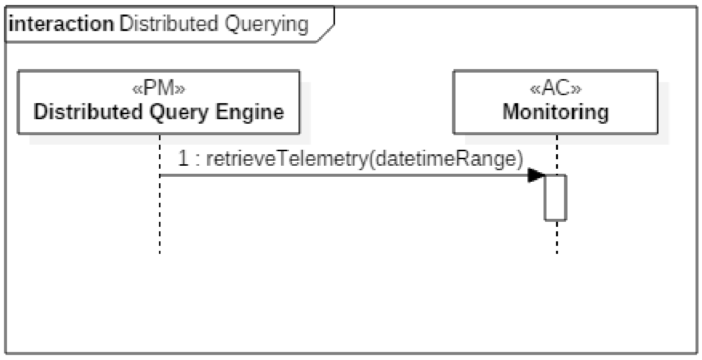
interaction Distributed Querying
«PM» Distributed Query Engine
«AC» Monitoring
1 : retrieveTelemetry(datetimeRange)

Figure 21. Distributed Query Engine - Query engine as an abstraction layer

5.2.1. Communication with the Agent Controller

Monitoring: each instrumentation probe will notify the Distributed Query Engine of its identity and publish location of metrics it collects.

5.2.2. Communication with the Platform Manager

Analytics: for each node in the landscape graph being analyzed, the Analyzer will query metrics - for the required time period - from the query engine for those node

5.3. Analytics

For a given service deployment, the Analytics module will query the physical deployment configuration from the landscape that the workload executed on. The associated telemetry for those physical nodes will then be queried for that same timeframe and mapped to that landscape sub-graph. A number of analytics algorithms will now be run against this data and the derived heuristics and models will then be fed back to the Recommender system. It is envisioned that the analytics module will take the form of an SDK and therefore be extensible, allowing new algorithms to be developed and added to the available collection of algorithms.

There are multiple methods to realize the actuation triggers and feeds, so the analytics system should support Streaming where analysis is performed on streamed data, offline mode where analysis could take a long period (e.g., large dataset), complex event processing analysing events in real time to derive learnings, continuous analysis of combinations of processes, or finally a user initiated analysis.

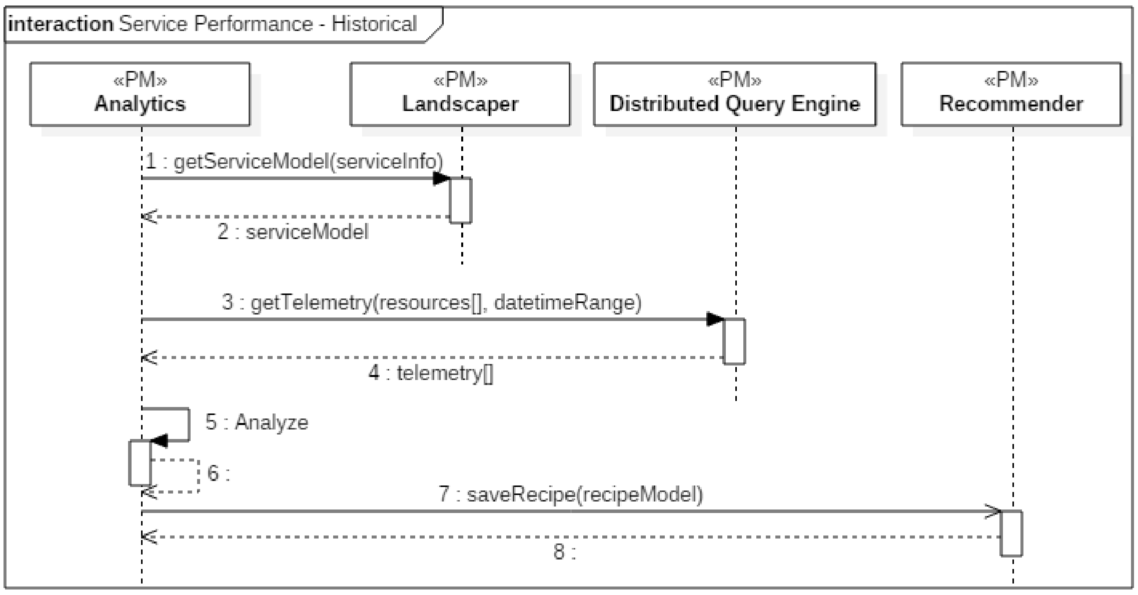
interaction Service Performance - Historical
«PM» Analytics
«PM» Landscaper
«PM» Distributed Query Engine
«PM» Recommender
1 : getServiceModel(serviceInfo)
2 : serviceModel
3 : getTelemetry(resources[], datetimeRange)
4 : telemetry[]
5 : Analyze
6 :
7 : saveRecipe(recipeModel)
8 :

Figure 22. Analytics - Service performance analysis (historical)

The output of any analysis will be in the form of heuristics and models, e.g. decision trees and will be stored in the Recommender module. These will be queried during service placement.

The analysis of currently executing services is similar to historical with the exception of notifying the Lifecycle Manager that a new deployment configuration is now available for a given Service Descriptor or service type. The Lifecycle Manager can now decide if it wants to re-place services matching this descriptor using this new deployment configuration.

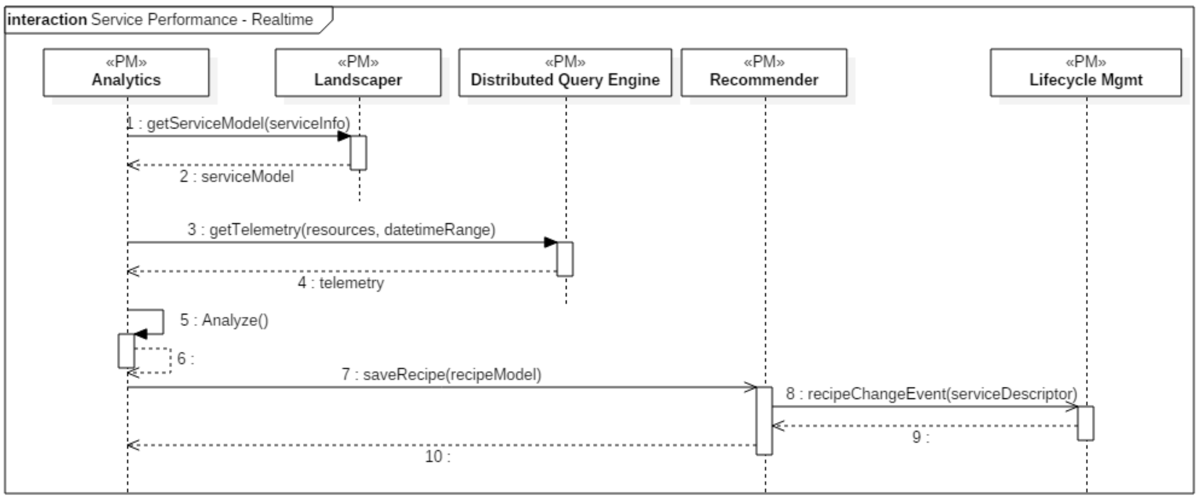
interaction Service Performance - Realtime
«PM» Analytics
«PM» Landscaper
«PM» Distributed Query Engine
«PM» Recommender
«PM» Lifecycle Mgmt
1 : getServiceModel(serviceInfo)
2 : serviceModel
3 : getTelemetry(resources, datetimeRange)
4 : telemetry
5 : Analyze()
6 :
7 : saveRecipe(recipeModel)
8 : recipeChangeEvent(serviceDescriptor)
9 :
10 :

Figure 23. Analytics - Service performance analysis (real time)

5.3.1. Communication with the Agent Controller

No interaction anticipated currently.

5.3.2. Communication with the Platform Manager

Landscaper: the *Analytics* module needs to query a service stack from the *Landscaper*. This is the all the relevant nodes (Service, Virtual, Physical) relating to an executing service.

Distributed Query Engine: for each node that makes up the service stack of the executing service, the *Analytics* module then needs to call the Distributed Query Engine to poll for all telemetry associate with those nodes allowing it to map performance to infrastructure.

Recommender: the Analytics module will store and update configurations that provide optimal performance for a given service descriptor.

6. Interfaces Design

The PM will be responsible for the orchestration of services based on the compute, storage and network resources and using a full-stack monitoring system, which receives telemetry data from different sources. This block is also responsible for coordinating the distributed execution of services and applications within the mF2C infrastructure.

The PM shall be a generic unit block which is able to both communicate with other mF2C Agents and with the Agent Controller, as shown in the following figure.

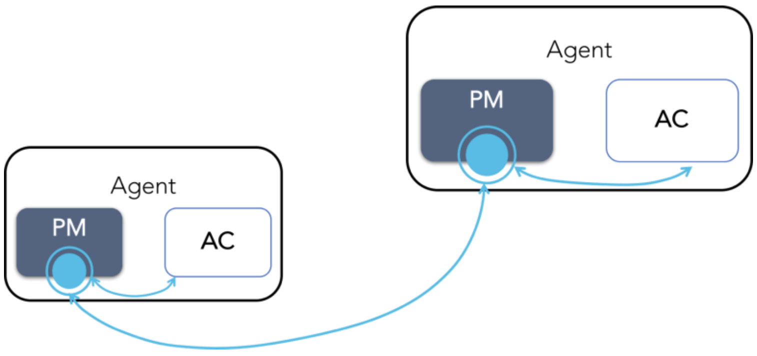

Figure 24. Communication between agents

The PM provides the interoperability required to make an efficient use of resources within heterogeneous, scalable and likely transient platforms like mF2C.
It is possible then to identify two different interaction paths that need to be taken into account before building the PM interface:

- **PM2PM**: all the requests that are sent back and forward between agents. This path is responsible for handling requests for managing services and applications, retrieving system-wide monitoring metrics, registering new resources, and any other PM functionality already described in D2.6 [1];
- **PM2AC**: all the interactions between the platform manager and the agent controller within the same agent. This path is responsible for all the requests concerning the allocation and deployment of services and applications, application monitoring, resource categorization and identification, and any other AC building blocks already described in D2.6 [1].

6.1. CIMI as the PM Interface

Since every mF2C agent will represent a node within an infrastructure, it is acceptable to conceptually address mF2C as an IaaS, which fits well with the CIMI model and RESTful HTTP-based protocol for management interactions, as described by DMTF [8].
CIMI provides a standard for the management of resources within an infrastructure. For mF2C and the PM this means that every possible resource that is to be managed by the PM (services, applications, credentials, devices, etc.), will be modelled and represented (in both JSON and XML) according to the CIMI specification. These are identified by URIs, whereby each resource representation shall have a globally unique ID attribute of type URI which acts as a reference to itself.
Beside its ease of use and wide industry support, this high-level interface provides:

- consistent resource management patterns, making it easy to develop small, lightweight and infrastructure agnostic clients;
- auto discovery of new resources without changing clients, enabling dynamic evolution of the platform;
- standard mechanism for referencing other resources;

- flexibility to cover a wider range of resources than strictly those related to cloud infrastructure management.

6.1.1. Protocol

With CIMI, all PM operations shall be HTTP based, adding to the usual PUT, GET, DELETE, HEAD and POST requests the possibility to have a JSON body, while covering all basic Search (or Query) and CRUD (Create, Read, Update, and Delete) operations plus the possibility to add custom operations which are mapped into POST requests. As an example, the CIMI standard does not mandate any authentication nor authorization process, but there is the possibility to extend the model to support authentication like "user" and "session" resources, which can also be then extended to provide access control lists, allowing fine-grained authorization. All the resource representations shall therefore include an "operations" attribute which explicitly states the PM operations allowed to the client on that resource. Obviously, the security here must integrate with the mF2C security platform, see section **Error! Reference source not found.**.

The use of the universally supported HTTP protocol makes CIMI the right interface for such an environment where multiple programming languages will be used to successfully integrate different devices, operating systems and architectures into a common infrastructure.

6.1.2. Entry Point Resource and Collections

For every resource type, there will be a Collection which groups all its resources, and because consumers and even developers should not be obliged to assume anything about the operations and collections available in the platform, there shall exist a well-known entry point resource allowing the discovery of the existing collections and operations. For the moment, we'll define this entry point as being the mF2C Entry Point resource, which programmatically shall map to *mf2c-entry-point*.

6.1.3. CIMI-defined Queries

The CIMI specification provides advanced features for manipulating results when searching collections. All the resource selection parameters are specified as HTTP query parameters. These are specified directly within the URL when using the HTTP GET method. The PM interface shall provide CIMI defined query parameters that will give users the possibility to at least:

- filter collections, e.g.

  `?$filter=expression`
  where "expression" is a mathematical expression compliant with the EBNF grammar defined in the CIMI specification;

- sort collections, e.g.

  `$orderby=attributeName[:asc|:desc], ...` ;

- define a range of resources (paging), e.g.

  `?$first=number&$last=number` ;

- specify a subset of a resource to be acted upon, e.g.

  `?$select=attributeName,...` ;

- expand references to avoid repeated requests to get referenced resources, e.g.

  `?$expand=attributeName,...` .

Unsupported, or unknown, query parameters shall be silently ignored by the PM.

6.2. Examples and Existing Implementations

SixSq has been incrementally adopting CIMI to cover all SlipStream resources and their management from any programming language [9]. Their implementation currently offers Clojure (Java), ClojureScript (JavaScript) and Python clients at various levels of maturity, while the server side implementation of the interface is written in Clojure and therefore might not be ideal for smaller mF2C agents. This implementation portability to ClojureScript would be however straightforward and could run on a lightweight Node.js server environment. The use of Node.js on the PM would also optimize the handling of the CIMI resources data, as there are a number of lightweight databases already made available and compliant with the framework.

6.2.1. Client Mockup

This section will attempt to simulate a typical interaction between a client and the PM, while proposing some key concepts and base structure for the CIMI implementation and resource representation in the mF2C PMs.

For this example, let's assume the PM API is running on port 80, on a device with the IP 1.2.3.4.

Discovery

Let's assume the client (a regular consumer, a developer or even an application) is completely agnostic of the PM's underlying resource model. The first step would be to discover which resource collections exist:

```
~# curl https://1.2.3.4/api/mf2c-entry-point --user "ADMIN:PWD" -H "Accept:
application/json"
```

```
{
  "id" : "mf2c-entry-point",
  "resourceURI" : "http://schemas.dmtf.org/cimi/1/dsp8009.xsd",
  "created" : "2016-06-21T17:31:14.950Z",
  "updated" : "2016-06-21T17:31:14.950Z",
  "baseURI" : "http://1.2.3.4/api/",
  "devices" : {
 "href" : "device"
  },
  "operations" : [ {
 "rel" : "edit",
      "href" : "mf2c-entry-point"
  } ],

  "other fields" : "..."
}
```

Note: even though the example uses authentication, the discovery entry point is always publicly available.

Add a new device

If we do not know what the device representation looks like, then we can query its collection:

```
~# curl -H "Accept: application/json" https://1.2.3.4/api/device
```

This will return the full list of "device" resources, with their structure. Once the resource structure is known and we actually verify (from the previous request) that the "devices" collection has an "add" operation, then our *new_device.json* will be:

```
{
 "name": "newDeviceRandomName",
 "description": "A new mF2C agent, running on a RPi"
 "networkName" : "default",
 "deviceRegion" : "Geneva",
 "deviceType" : "android",
 "appVersion" : "1.3.4",
 "deviceDisk" : 32,
 "deviceMemory" : 2048,
 "deviceCores" : 2,
 "deviceName" : "John-Doe-Smartphone",
 "deviceVisibility" : "public",
 "deviceRank" : "leader",
 "deviceIP" : "42.42.42.42",
 "nativeContextualization" : "linux-only",
 "canDeploy" : true,
 "status" : "running",
 "endpoint" : "http://42.42.42.42/",
 "tlsEnabled" : false
}
```

As described in **Error! Reference source not found.**, in practice adding a device will require p ermission, in order to prevent malicious people from adding nefarious services.

With this JSON content, we can then create a new "device" resource:

```
~# curl -XPOST -H content-type:application/json https://1.2.3.4/api/device/
-d@new_device.json
    {
      "status" : 201,
      "message" : "successfully created device/255acb74-1742",
      "resource-id" : "device/255acb74-1742"
    }
```

Select the new device

Once created, the new resource can be fetched from its respective collection, by using a filter:

```
~#           curl           -H           "Accept:           application/json"
https://1.2.3.4/api/device?$filter=id="device/255acb74-1742"

{
  "acl" : {
 "owner" : {
       "principal" : "ADMIN",
       "type" : "ROLE"
 },
 "rules" : [ {
       "principal" : "ADMIN",
       "type" : "ROLE",
       "right" : "MODIFY"
 } ]
  },
  "resourceURI" : "http://www.mf2c-project.eu/dev/DevicesCollection",
  "id" : "device",
  "operations" : [ {
 "rel" : "add",
 "href" : "device"
  } ],
  "devices" : [ {
"id" : "device/255acb74-1742",
 "updated" : "2017-08-18T14:47:23.953Z",
 "created" : "2017-08-18T14:47:23.953Z",
 "name": "newDeviceRandomName",
 "description": "A new mF2C agent, running on a RPi"
 "networkName" : "default",
 "deviceRegion" : "Geneva",
 "deviceType" : "android",
 "appVersion" : "1.3.4",
 "deviceDisk" : 32,
 "deviceMemory" : 2048,
 “deviceCores” : 2,
 "deviceName" : “John-Doe-Smartphone",
 "deviceVisibility" : "public",
 "deviceRank" : "leader",
 "deviceIP" : "42.42.42.42",
 "nativeContextualization" : "linux-only",
 "acl" : {
             "owner" : {
                   "principal" : "ADMIN",
                   "type" : "ROLE"
             },
             "rules" : [ {
                   "principal" : "ADMIN",
                   "right" : "ALL",
                   "type" : "ROLE"
             } ]
 },
 "operations" : [ {
             "rel" : "edit",
             "href" : "device/255acb74-1742"
 }, {
             "rel" : "delete",
             "href" : "device/255acb74-1742"
 } ],
 "resourceURI" : "http://www.mf2c-project.eu/dev/Device",
 "canDeploy" : true,
 "status" : “running”,
 "endpoint" : "http://42.42.42.42/",
 "tlsEnabled" : false
  }],
  "count" : 1
}
```

Some more complex workflows are also possible where resources are templated by other resource type. For example, if there is a PM resource called "credentials", there might be another one called "credentialTemplates". This is useful when there are resources which have a representation that is generated on the server side, based on a template and not the actual resource attributes (unlike what was just exemplified above, where the full "device" representation was known and defined directly by the client).

7. Microagents Design

In the mF2C project proposal, the need for a special kind of agents called microagents was foreseen. These agents were the ones interfacing with sensors and other devices that cannot host any significant piece of the software stack. Although the underlying concept remains, this initial idea was refined during the definition of the mF2C architecture described in D2.6 [1].

In particular, and for the sake of generality, the architecture makes no distinction between types of agents but, instead, considers only the concept of *agent*, which is the main entity in the mF2C architecture. The structure of an agent is the same regardless of the layer on which it is located (i.e in the cloud, next to sensors or actuators, or in an intermediate layer). Besides resulting in a cleaner and clearer design, not distinguishing between kinds of agents gives more flexibility to the framework, providing the possibility to reorganize the available agents according to the capacity of their associated resources, instead of having a pre-defined role.

In this way, the concept of microagent as a distinguished kind of agent with a specific functionality disappears, and becomes a role that any agent may play in the mF2C infrastructure, in the same way that any agent with sufficient capacity can be the leader of a cluster.

That is, microagents are just those agents, with the same structure than any other agent, which are located in the lowest layer of the infrastructure. As such, they will not perform the functions of a leader and, thus, they do not need to (but can) have the same amount of capacity as other agents in upper layers. The need for a specific design for this kind of agents has not been identified.

References

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

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

[3] “D2.4 Security/Privacy Requirements and Features,” [Online]. Available: http://www.mf2cproject.eu/wp-content/uploads/2017/05/mF2C-D2.4-Security-Privacy-Requirements-and-Features-IT1.pdf.

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

[5] F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Álvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia and R. M. Badia, “ServiceSs: An Interoperable Programming Framework for the Cloud,” *Journal of Grid Computing,* vol. 12, no. 1, pp. 1267-91, 2014.

[6] “D3.3 Design of the mF2C Controller Block (IT-1),” [Online].

[7] J. Martí, A. Queralt, D. Gasull, A. Barceló, C. Toni and J. J. Costa, “dataClay: A distributed data store for effective inter-player data sharing,” *Journal of Systems and Software,* vol. 131, pp. 129-145, 2017.

[8] “Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol. An Interface for Managing Cloud Infrastructure,” [Online]. Available: https://www.dmtf.org/sites/default/files/standards/documents/DSP0263_2.0.0.pdf.

[9] “SlipStream API Reference,” [Online]. Available: http://ssapi.sixsq.com/.