Università degli Studi di Pisa

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Computer Engineering

Tesi di Laurea Magistrale

Design and Development of a solution for QoS support in the FIWARE IoT architecture

Candidato:
Lorenzo Trisolini

Relatori:
Prof. Ing. Enzo Mingozzi

Prof. Ing. Giuseppe Anastasi

Anno Accademico 2014/2015

Chapter 1

Abstract

Now-days Internet of things (IoT) is changing the Internet vision. IoT architectures are employed in our daily lives in many applications and in a ubiquitous way. Technological advancements together with cloud computing paradigms are bootstrap in this process. In this context, the FIWARE platform, developed by the European FI-PPP project, aims to be a key player in this new internet vision, offering the means to build new applications that can benefit from high IoT service availability. Despite the QoS it has been identified as a key non-functional requirement to enable many IoT-related applications, the FIWARE IoT platform doesn't provide QoS negotiation and resource allocation functionalities. In this work we propose a solution for QoS support in the FIWARE IoT architecture. We start analyzing the different component of the FIWARE platform. Then we choose the RTTA heuristic algorithm to solve the problem of QoS-aware service selection and we modify it to obtain a larger set of service selection solutions. Finally, once we establish a deployment using a meaningful set of FIWARE IoT modules, we design and develop a QoS support in FIWARE, using the modified RTTA heuristic algorithm. At last, we show through simulations that the set of solutions of the new RTTA algorithm is larger than the original one and we validate the integration of our solution in FIWARE through a use case scenario.

Contents

Chapter 2

Introduction

Internet of things (IoT) is a new Internet concept that tries to connect everything that can be connected to the Internet, where everything refers to people, cars, televisions, smart cameras, microwaves, sensors, and basically anything that has Internet-connection capability. A recent study by Cisco predicts that IoT is projected to create $14 trillion net-profit value, a combination of increased revenues and lowered costs, to private sector from 2012 to 2022 [1]. IoT is not seen as an individual stand-alone system, but as a globally integrated infrastructure with many applications and services [2]. Now-days Internet of Things (IoT) architectures are composed by thousands of *smart* things or objects connected to the Internet. These objects are exploited as remote sensing and/or actuating components of a large distributed computing infrastructure. Remote sensing plays a key role in the acquisition of data about and from everything without needing of physical field visits and in real-time manner. In this scenario new applications has been conceived, that is machine-to-machine (M2M) applications, i.e., IoT applications exploiting direct interactions between things in order to monitor and control themselves and the surrounding environment with no or minimum intervention, are expected to play a major role, given their huge impact on many real-world application domains including industrial, healthcare, transportation, social and environment[3]. Technological advancements are bootstrap in this process; examples of systems made of inter-

connected physical objects are already employed in our daily lives: house climate control systems, alarm and intrusion detection systems, smart metering systems are only few examples of how communication capabilities of smart objects are exploited to provide empowered services to end users. Another bootstrap factor, in this process, is the cloud computing paradigm. It is being extended to accommodate the characteristics of the IoT architectures, thus introducing the so called Thing as a Service model[4]. It permits the deployment of a distributed IoT service infrastructure that shares globally sensing and/or actuating resources through the Internet. It also represents a shift from a set of vertical systems working in isolation to a horizontal platform integrating different physical objects from heterogeneous environments. Smart things are exposed through a service-oriented unified interface which allows applications to access things transparently regardless the physical layer technology and the location. Unlike conventional way of collecting and processing sensory data, Thing as a Service model now enables *decentralization of data sensing and collection, sharing of information and resources, remote access to global sensed information and its analytics* and *elastic provisioning of resources*. An interesting platform that offers services meaningful in this context is FIWARE (where FI stands for Future Internet). FIWARE platform is supported by the Future Internet Public-Private Partnership (FI-PPP) project of the European Union. It is a project that aims to design an open platform that makes sophisticated and innovative Internet application easy to build. It lowers the costs and complexity of serving globally a large number of users and handling data at a large scale. It assembles a set of "building blocks" that can create complex applications. These blocks called Generic Enablers (GE) are already available and ready to use. The GEs are organized in various technical chapters: *Cloud Hosting, Data/Context Management, Architecture of Applications/Services Ecosystem and Delivery Framework, Interface to Network and Devices, Security, Advanced Middleware and Web-based User Interface* and lastly *Internet of Things Services Enablement*. Our work has been focused on the last chapter. This set of GEs represents the bridge where future Internet services

interface and leverage on the ubiquity of heterogeneous, resources-constrained devices in the IoT environment [5]. These software components allows the integration of different IoT systems providing an abstraction level for the applications that can benefit from high IoT service availability. In each system, different smart things offer similar services with common functionalities but different QoS and costs. QoS is of particular importance since it has been identified as a key non-functional requirement to enable many IoT-related application scenarios. For example, Security and emergency content in smart city and home environments often has strict real-time requirements. As an example, latency (besides dependability) is a critical factor for applications such as real-time sensor monitoring in personal health-care or public safety systems. Other applications like, e.g, road traffic management applications for urban mobility, though less sensitive to delay bounds, may nevertheless benefit from receiving some form of real-time treatment, at least for a subset of their provided services. Moreover applications involving streaming of multimedia context like video surveillance that consumes high bandwidth will require full support from the platform not only to guarantee an acceptable level of service but also to avoid saturation and waste of network resources [6]. In this context efficient management of resources will be required to perform a proper selection of things matching applications requests, whilst guaranteeing to meet the respective QoS requirements. On the other hand, smart things are constrained devices in terms of computation, storage and energy. Furthermore, the IoT context is continuously changing because of intermittent availability of things. Therefore, service selection approaches are needed to address unique features when allocating things to application requests.

After the analysis, from scratch, of the FIWARE architecture and the available GEs, our work it has been focused on the IoT chapter. We have studied a meaningful deployment of the IoT GEs to build a FIWARE Instance platform. In this phase we have analyzed each GE offered in the IoT chapter of FIWARE library to understand what functions they offer and what are the interfaces that they use to communicate each other, analizing the protocol used for the communication and the management

of things data. Moreover, we had to modify some IoT software components because, we have used a deployment for which they weren't implemented to. So they are being adapted to communicate each other. Although FIWARE project aims to be a player in the IoT context, it doesn't include a support for the QoS. As we have seen, QoS is a key non-functional requirement to enable many IoT scenarios. So, once the deployment has been established and the communication between IoT components has been tested, we have used this FIWARE IoT platform instance to design and develop a QoS support in the IoT architecture. The development of the QoS solution has been developed in two phases. In the first one, we have analyzed the heuristics proposed to solve the problem of an efficient allocation of the service requests towards the things. We have choosed the heuristic proposed in [3], introducing a modification in the algorithm that extends the set of solutions of the selection problem, at the cost of a delay in the execution of the service requests. In the second one, stemming from the information model used in FIWARE, we have design a QoS model, introducing new operations that allow a negotiation phase in which a IoT application can negotiate a service level agreement specifying some QoS requirements. To this aim, we have implemented a wrapper for the main software component that, in our deployment, manages the services requests towards the things and its data.

Chapter 3

FIWARE

3.1 FIWARE platform

FIWARE is a software platform that provides enhanced OpenStack-based cloud hosting capabilities plus a rich library of components bringing a number of added-value functions offered “as a Service”. These library components provide open standard APIs that make the development of Future Internet applications much easier. The platform aims to increase the global competitiveness of the European ICT economy by introducing an innovative infrastructure for cost-effective creation and delivery of versatile digital services. The FIWARE goal is to make sure that an open platform alternative to existing proprietary platform (e.g. , Google or Amazon) will exist, around which a sustainable open innovation-driven ecosystem can be created. The FIWARE stakeholders can be Telecom Industries that can play the role of a Software Provider. They can accelerate the development of standards based on FIWARE results. They also may play the role of Application/Service Providers, developing new services and applications for large Usage Areas where telecom-based communication, security and availability levels as well as support to roaming users are required. Other stakeholders are IT Industries that play the role of Software Providers and/or Application/Service Providers.

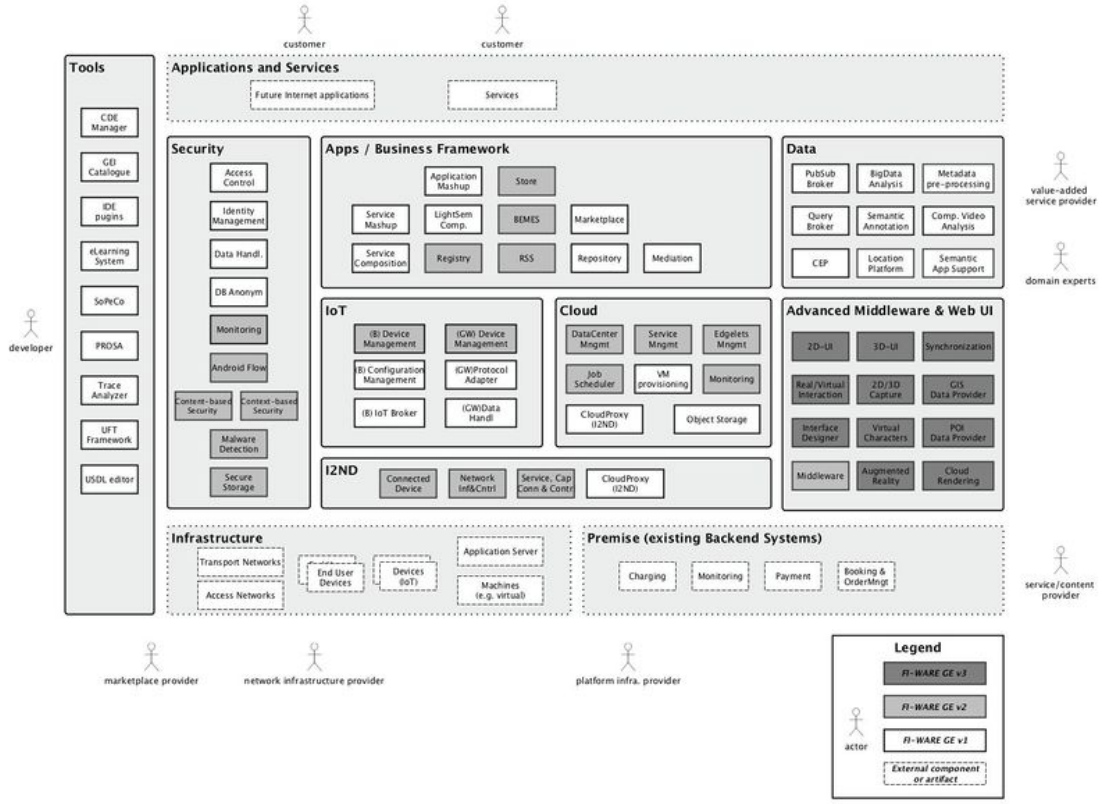


Figure 3.1: FIWARE Architecture

3.2 FIWARE structure

FIWARE is based upon elements called "Generic Enablers". They are the building blocks of the platform, made up of a set of components which together support a concrete set of functions and provides a concrete set of APIs and interoperable interfaces that are in compliance with open specifications publish for that GE. GE Open Specifications contain all information required in order to build compliant products which can work as an alternative implementations of GEs developed in FIWARE project. GE Open Specifications typically include:

- Description of the scope, behaviour and intended use of GE;
- Terminology, definitions and abbreviations to clarify the meanings of the specification;
- Signature (Restful interface) and behaviour of operations linked to APIs that the GE should export;
- Description of protocols that support interoperability with other GE or third party products.

The various components are organized in a rich library, the FIWARE Catalogue, where it can be found also the reference implementation of each component allowing the developers to put into effect functionalities such as the connection to the Internet of Things or Big Data analysis, making the implementation much easier. The GEs can be combined in different ways to create a particular type of product or service. The types of products/services are:

- *FIWARE Compliant Platform Product*: A product which implements, totally or in part, a FIWARE GE or composition of FIWARE GEs (therefore, implements a number of FIWARE Services). Different FIWARE compliant Platform Products may exist implementing the same FIWARE GE or composition of FIWARE GEs;

- *FIWARE Instance*: The result of the integration of a number of FIWARE compliant Platform Products. It comprises a number of FIWARE GEs and supports a number of FIWARE Services. Provision of Infrastructure as a Service (IaaS) or Context/Data Management Services are examples of services that a particular FIWARE Instance may support, implemented combining a concrete set of Platform Products. FIWARE Instances are built integrating a concrete set of FIWARE compliant Platform Products;

- *Future Internet Application*: An application that is based on APIs defined as part of GE Open Specifications. It should be portable across different FIWARE Instances that implement the GEs the application relies on.

In the overall value chain envisioned around FIWARE, it can be identified different roles, classified in:

- *GE Provider*, that is any implementer of a GEs;

- *Instance Provider*, that is any company or organization which deploys and operates a platform Instance and establishes some sort of business model around that particular Instance. Note that FIWARE Instances may not consist only of the integration of FIWARE compliant Platform Products but their integration with other products which allow the FIWARE Instance Provider to gain differentiation on the market (e.g. integration with own Operating Support Systems or with other products supporting services that are complementary to those provided by FIWARE GEs) or to enable monetization of its operation (e.g., integration with own Billing or Advertising systems);

- *Application/Service Provider*, any company or organization which develops Future Internet applications and/or services based on GE APIs and deploys those applications/services on top of platform Instances.

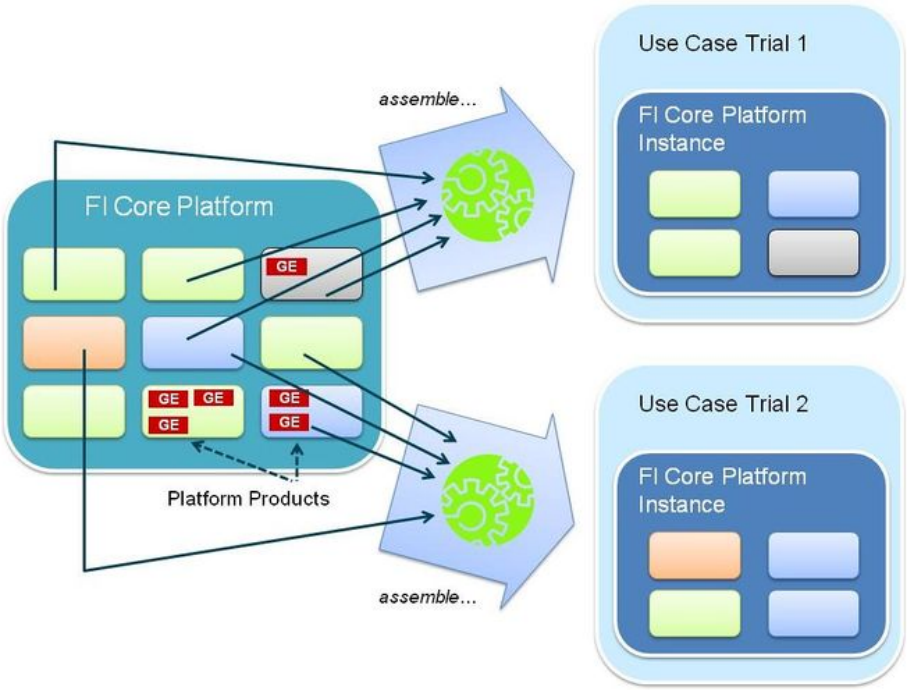


Figure 3.2: examples of FI-WARE Instances

3.3 FIWARE IoT Services Enablement

Among all FIWARE modules, we have focused our study on Internet of Things chapter that provides the Generic Enablers to allow Things to become available, searchable, accessible, and usable context resources fostering FIWARE-based Applications interaction with real-life objects. Before go through this chapter, we introduce some real-scenarios:

- *Scenario 1.* 7:25 AM, starting the car, David's car device asks for activation regarding traffic and pollution applications. David chooses traffic application and the car device sends a signal to the closer gateway that the vehicle is active and in the traffic. The gateway, based on the planned route home-office defined in the car device, diffuses to the gateways network that a new actuator is alive and the expected time instant when it will roam from the first area (gateway 1) to the second area (gateway 2);

- *Scenario 2.* Sensor measurement sharing during the journey to David's office, the car device receives a short message from gateway 7 to activate weather data collection to draw snow storm progress on the city map. As David is driving, the car device launched a request to the profile database to validate David choices. Based on the latest information, from yesterday 10:00 AM, David has no objection to communicate weather information. Immediately, the car device begins to send temperature, windscreen activity, humidity level;

- *Scenario 3.* Tomorrow, David wants to go sooner to his office because the home clock screen advertises that weather forecast was too optimistic today and that bad weather will come during the night. The picture, based on 357 climate sensors, shows clearly that the snow storm will arrive around 7:30 AM. When he begins to cook his dinner, his home gateway informs the city eco-management application that a new consumption cycle is planned in this district. All cities till around 100 km from his home are sending information. The 357 sensors indicate clearly that air pressure is falling quickly and some mobile sensors in this area are providing real-time temperature indications. Night would be very cold and a snow storm is now forecast for tomorrow evening.

3.4 IoT concepts

In the IoT technical chapter are defined various concepts representing a series of related actors playing in the IoT context as we can see in the figure 3.3.

They can be divided in different interface abstraction levels:

- *Device Level,* an hardware unit having the capability to either perform a measurement or an actuation. Device-level data is the raw data provided by the device. Management functionality like sensor calibration, firmware updates, and battery status monitoring is also taking place on the device level. In this level it is included also the IoT gateway, that is a device that additionally to or

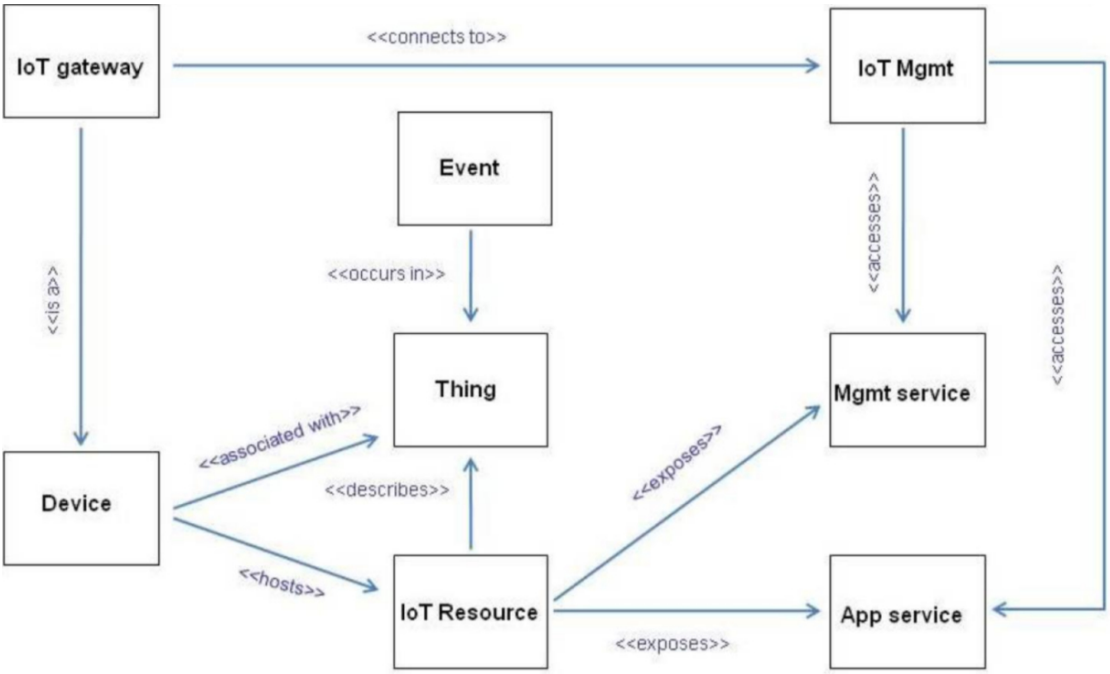


Figure 3.3: Concepts defined in the IoT technical chapter

instead of sensing/actuating provides inter-networking and protocol conversion functionalities between devices and IoT backend. It is usually located at proximity of the devices to be connected. An example of an IoT gateway is a home gateway that may represent an aggregation point for all the sensors/actuators inside a smart home;

- *IoT Resource level*, an IoT resource is a computational element providing access to sensor/actuator devices. An information model for the description of IoT resources can include context data like location, accuracy, status information, etc. IoT resource level data consists not only of the measured data, but also context information like the data type, a time stamp, accuracy of measurement, and the sensor by which the measurement has been performed;

- *Thing level*, A thing can be any object, person, or place in the real world. Things are represented as virtual things having an entity ID, a type and several

attributes. Sensors can be modelled as virtual things, but other real-world things like rooms, persons, etc. can be modelled as virtual things as well. So thing level data consists of descriptions of things and their attributes, while information on how the data has been obtained might be contained as meta data (Usually not provided by typical gateways).

It can be identified also an application level, in which there are the following elements:

- *Management service*, It is the feature of the IoT resource providing programmatic access to readable and/or writeable data belonging to the functioning of the device;
- *Application service*, It is the feature of the IoT resource providing programmatic access to readable or writeable data in connection with the thing which is associated with the device hosting the resource. The application service exchanges application data with another device (including IoT gateway) and/or the IoT backend;
- *Event*, An event can be defined as an activity that happens, occurs in a device, gateway, IoT backend or is created by a software component inside the IoT service enablement.

3.5 Things as NGSI context entities

All informations about Things and/or IoT resources are managed through the OMA NGSI standard, that describes these objects as Context Entities. The OMA NGSI (Next Generation Service Interface) Context Management standard provides the NGSI-9 and NGSI-10 interfaces to manage and exchange Context Information about Context Entities. A Context Entity is any entity which has a state. Values of attributes of defined entities becomes the Context Information that applications have to be aware of in order to support context-awareness. Context Information is any

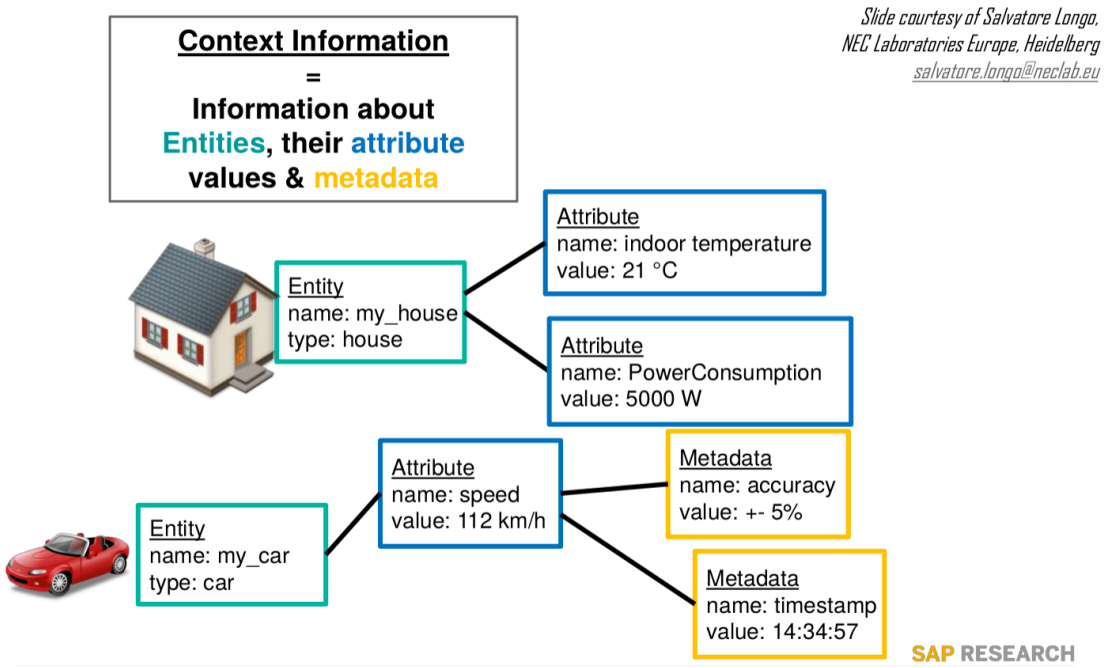


Figure 3.4: NGSI vision

volatile or persistent information, which describes a state of a Context Entity. An example of NGSI use case can be visualized in the figure 3.4.

Of course, Context Entities could be users, devices, places, buildings, therefore "things" as defined before. Context Information related to things can be measured by sensors, and combined with other context information manually set by humans, derived from interactions with the user, operations on handsets or terminals, inferred from other information, or requested from databases. Adoption of OMA NGSI enables the management of configuration of, and the data associated to, arbitrary physical objects in the real world, that are Devices and Things. It enables this at a level of abstraction that allows getting rid of the complexity of managing connections with gateways and devices. Updates on the state and configuration of those physical objects will come as updates on the Context Information (i.e., updates on attributes of Context Entities representing those physical objects) and Configuration (i.e., updates on information about available Context Entities). The purpose of the

NGSI-9 interface is to exchange information about the availability of Context Information and Context Entities, while NGSI-10 is designed for exchanging the Context Information itself [7].

3.6 IoT chapter Architecture deployment

IoT chapter Architecture deployment varies from simple scenarios in which few devices are connected using standard IoT communication protocols to more complex scenarios distributed across a large number IoT networks connecting IoT Gateways and IoT nodes and providing advanced composition and discovery functions. IoT GEs are spread over two different domains:

- *IoT Backend*: It comprises the set of functions, logical resources and services hosted in a Cloud datacenter. Up north, it is connected to the data chapter ContextBroker, so IoT resources are translated into NGSI Context Entities. South-wise the IoT Backend is connected to the IoT edge elements, that is all the physical IoT infrastructure;

- *IoT Edge*: It is made of all on-field IoT infrastructure elements needed to connect physical devices to FIWARE Applications. Typically, it comprises: IoT end-nodes, IoT gateways and IoT networks. The IoT Edge and its related APIs will facilitate the integration of new types of gateways and devices, which are under definition in many innovative research projects.

What we have said before, can be figured out in the figure 3.5. From a functional point of view, IoT Backed GEs perform the following functions:

- *Provision of Things as NGSI Context entities*: This means to create an NGSI Context Entity for each IoT resource. This function is typically carried out by the Backend Device Management GE. The GE in charge of this function plays the role of Context Producer for all Entity attributes related to IoT resources

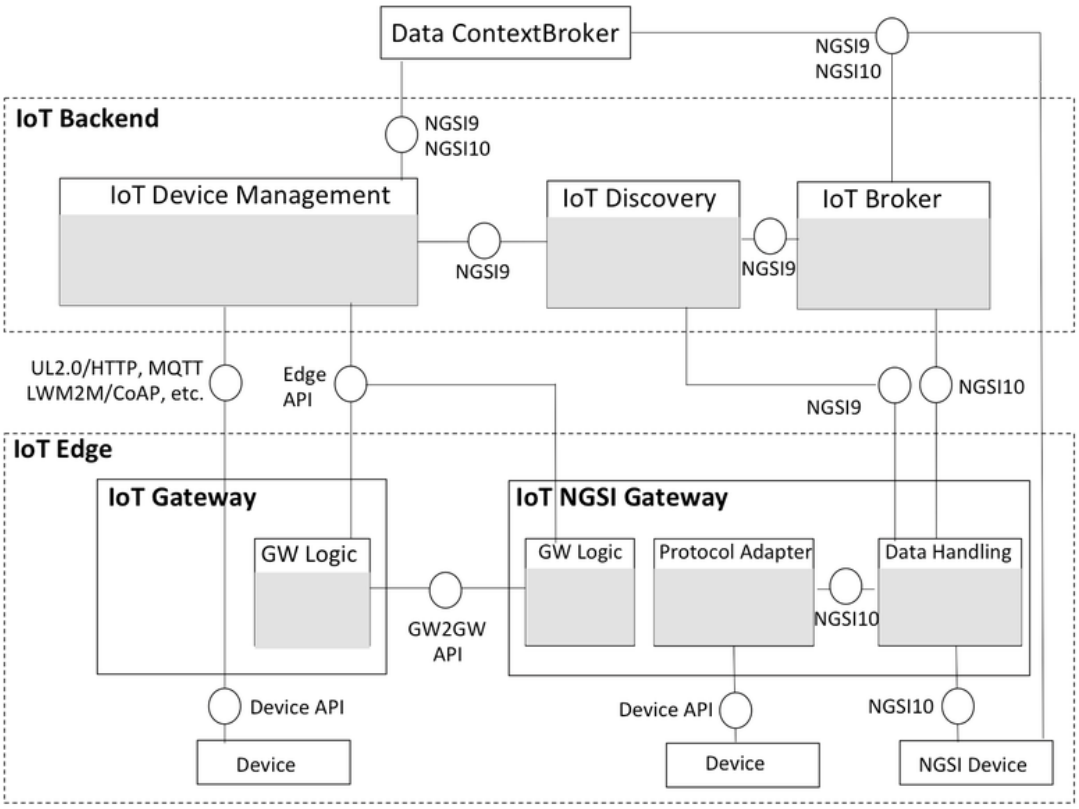


Figure 3.5: FIWARE IoT architecture

sensing/observations. On the other hand, it registers itself as Context Provider for those Entity Attributes related to actuation capabilities;

- *IoT Southbound protocol adaptation*: The variety of communication protocols for IoT devices and/or gateways is extremely high today. FIWARE provides an easy way to extend a number of supported communication protocols regardless they are based on open or proprietary specifications. This function (in the backend) is carried out by the Backend Device Manager GE so it handles the translation of IoT southbound protocols (sensing and actuation) into/from the OMA NGSI protocol;

- *IoT Edge management*: Some configuration, operation and monitoring functions regarding IoT Edge elements (Connectivity/Networks, Gateways, End-nodes) might be controlled from the Backend Device Management GE and thus exposing a convenient API to IoT integrators in addition to the NGSI API for ContextBroker interconnection;

- *IoT Devices composition and discovery*: This function is provided by the IoT Broker GE in combination with the IoT Discovery GE.

On the other hand, IoT Edge GEs implement the following functionalities [7]:

- *IoT Southbound protocol adaptation*: This functionality means to translate specific protocols into NGSI within a gateway device so that native NGSI entities might be pushed directly to the backend enablers (typically the IoT Broker or directly the Data chapter ContextBroker). This function is provided by the Protocol Adaptor GE. Current supported technologies are Zigbee and RFID tags;

- *Complex (NGSI) Event Processing*: Although there is one specific GE in the Cloud for this functionality (Data Chapter CEP GE), for some scenarios it might be useful to reduce the network traffic exchanged between the IoT edge elements and the Cloud infrastructure. It is provided by the Data Handling GE

within a Gateway device. Data Handling GE consumes, processes and delivers NGSI events so it is typically used together with the Protocol Adapter in the same gateway device;

- *Gateway Logic*: This function handles a gateway-to-gateway API (currently under definition) and the IoT Edge configuration API at the gateway level (currently under definition too). For the IoT Edge configuration of gateways existing protocols such as OMA-LWM2M management interfaces, OMA-DM or BBF TR.69 are to be considered. It will be provided by the Device Management GE;

- *IoT end-nodes Configuration*: Some IP-capable nodes will not traverse gateways and therefore they might be operated and monitored directly. In this specific case the management interfaces of OMA-LWM2M are expected to be considered due to the constraint nature of IoT-end nodes;

- *IoT Networks Configuration*: In complex scenarios such as smart-cities, IoT-end nodes are not expected to be connected over a unique connectivity network. On the other hand, several IP alternatives are expected to co-exist, namely: cellular (2G, 3G, 4G and soon 5G), meshed-radio networks (6LowPAN/IEEE.802.15.4), IP/BLE, LPWA, etc. It is important to note that this control plane is not the IoT devices or end-nodes control plane, but a different one that may include also interaction with network APIs. This function will be implemented at the IoT Edge module in the Backend Management GE, the Gateway Logic module at Gateways and in a similar module at IoT-end nodes;

  A key design statement is that the IoT context is continuously changing in-fact IoT Gateways are not permanently connected to the Backend. Besides, IoT Gateways can be constrained devices in some scenarios. Therefore, it is required, in the gateway domain, a light-weight implementation of some GEs.

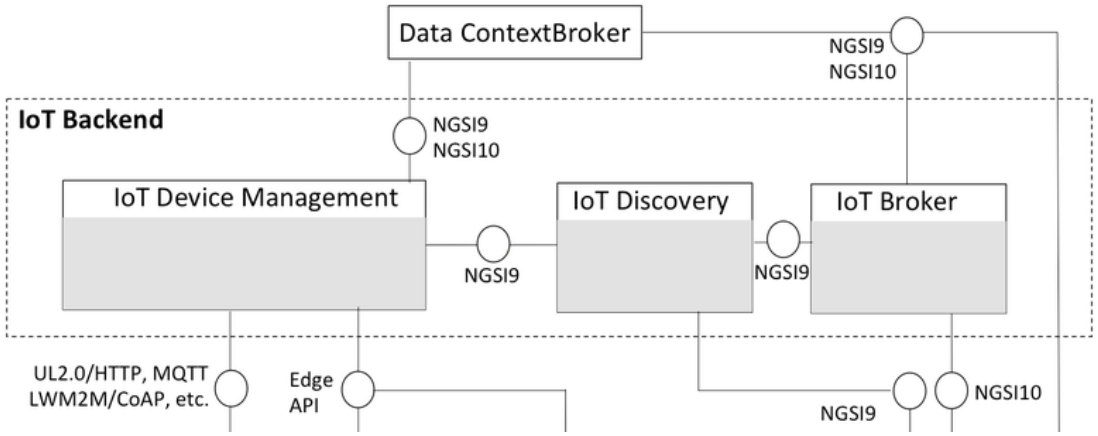


Figure 3.6: IoT Backend GEs

3.7 IoT Backend

The IoT Backend includes that GEs that are executed in a cloud platform and eventually communicate with the Data Context-Broker. They are the IoT Device Management, the IoT Discovery and the IoT Broker (figure 3.6).

Starting from the right we have the *IoT Device Management*, that connects devices and/or gateways (they may use different standard or proprietary communication protocols and API) to FIWARE platform. Moreover, it manages *NGSI Context Entities*, in-fact it handles, on the northbound side,the connection to an instance of the Data chapter ContextBroker to create one Context Entity per physical connected device (for most cases application developers, will only interact with the NGSI Entities). Lastly, it provides an *IoT Edge Manager module*, that give the possibility to IoT integrators to configure, operate and monitor IoT end-nodes, IoT Gateways and IoT networks.

The Device Management GE is composed by a set of components that are the *IoT Agent modules*, the *IoT Agent Manager module* and the *IoT Edge Manager module* (figure 3.7).

The *IoT Agent Manager* is an optional module that interface with all the IoT Agents installed in a datacenter throughout their Administration/Configuration API.

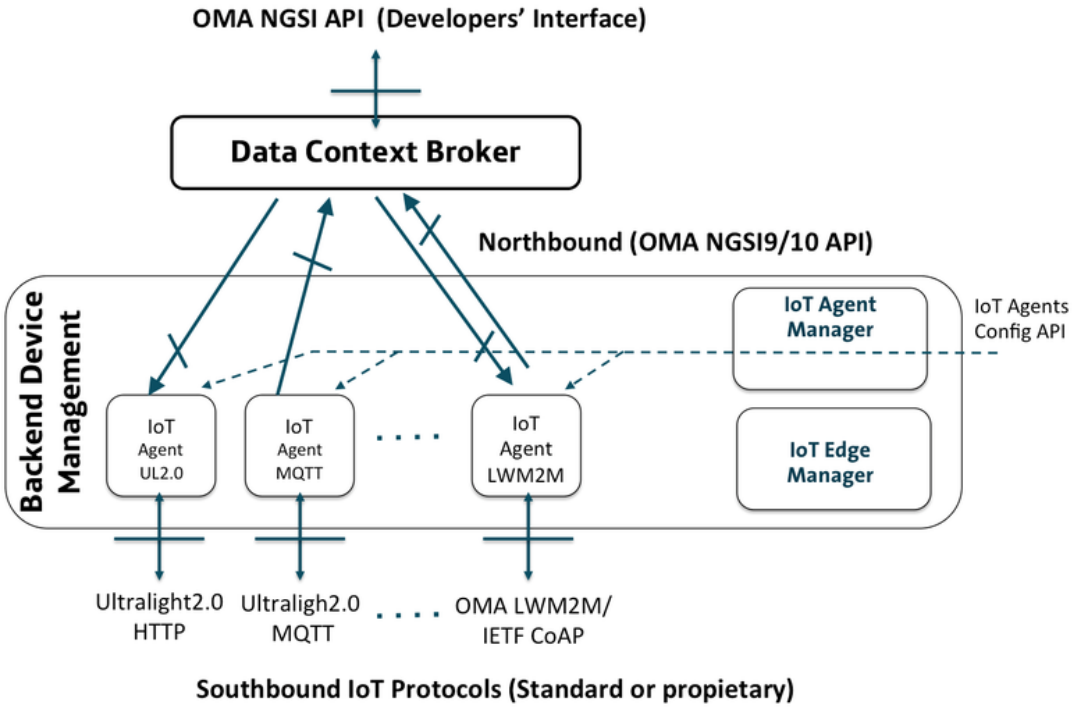


Figure 3.7: Device Management GE

This will enable a single point to launch, configure, operate and monitor all IoT Agents in a FIWARE Ecosystem

3.7.1 IoT Agent

As we have said before the Device Management GE is composed by IoT Agents that are software modules handling South IoT Specific protocols and North OMA NGSI interaction. The minimum configuration of Device Management GE includes at least one IoT Agent. The functions of this component are:

- It handles the creation of an NGSI Context Entity in a ContextBroker (at its northbound) per each one of the connected IoT Devices;
- It acts as Context Producer for those attributes related to sensing capabilities or observations;
- It provides an Administration/Configuration API.

Among all, the IoT Agent that we have analyzed is the *LWM2M/CoAP IoT Agent*. This IoT Agent connects Lightweight M2M Clients (devices), communicating over CoAP, to the NGSI Context Broker in the Data Chapter. The main interactions with the agent (without entering in low-level details) are:

- Device Provisioning;
- Service creation;
- Device Registration;
- Device lazy observation;
- Device active observation;
- Device command.

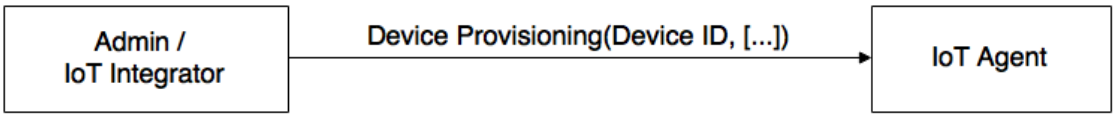


Figure 3.8: Device Provisioning

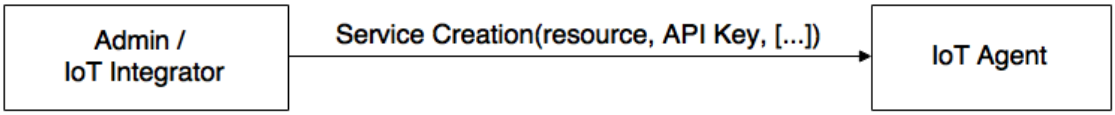


Figure 3.9: Service Provisioning

Device Provisioning In order for the system to recognize the device and do the appropriate mapping to NGSI context elements, one of two things must happen: either the device is provisioned in advance into the system, or a service is provisioned in the system and the device assigned to it. In the former case, the Device Id will be used to identify the device, using it as the Endpoint name in LWM2M. The following pieces of data can be specified in provisioning: device Id, security informations, NGSI entity mappings and types of attributes provided (figure 3.8).

Service creation For those cases where the devices are not specified individually, services can be provisioned as a whole. A service is identified by a pair of (resource, APIKey) attributes, and can contain roughly the same kind of information as the device provisioning requests. The specified resource corresponds to a LWM2M server endpoint where the clients will send their requests. Each time a device arrives to the specified endpoint, it will be assigned to the proper service based on the endpoint (figure 3.9).

Device Registration Every device in LWM2M must be registered to the LWM2M server before starting any interaction. In this registration the LWM2M client has to indicate the following information to the server: Endpoint name(Device Id) and supported objects (a list of links for every OMA LWM2M object that can be accessed by the server). The LWM2M server uses this information to assign the device to a

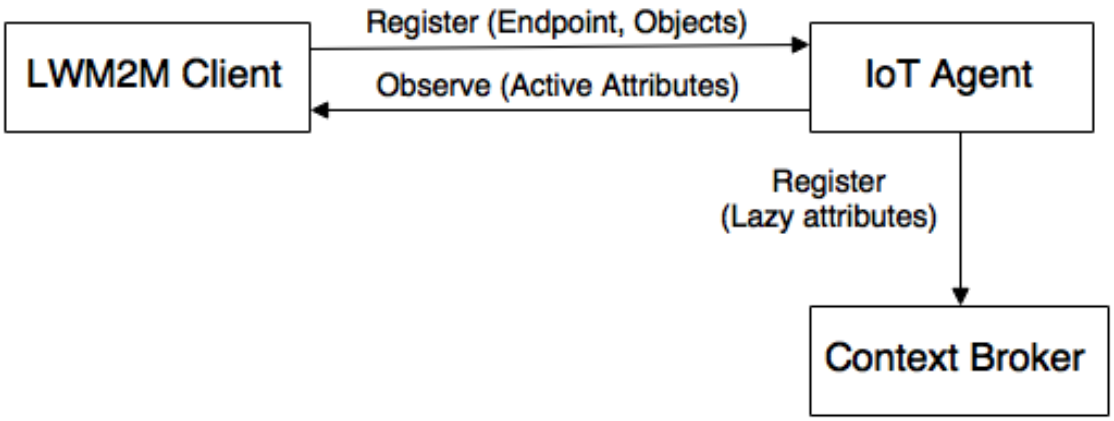


Figure 3.10: DeviceRegistration

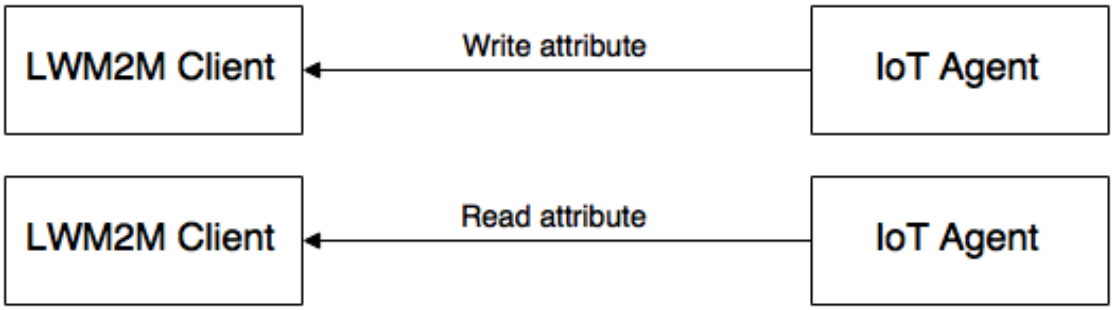


Figure 3.11: Device lazy observation

service (or to retrieve the device information in case it has been individually provisioned). Based on the service or device information, the server will decide what attributes of the objects provisioned by the device are active, lazy or commands and it will register itself in the Context Broker as the Context Provider of those in the two last categories (lazy and commands). For those attributes defined as active, the server will emit an Observe request (figure 3.10).

Device lazy observation For those attributes of the devices that marked as lazy, the updates and reads operations of Context Entities in the Context Broker GE will be mapped to Read and Write operations from the Device Management Interface in LWM2M (figure 3.11).

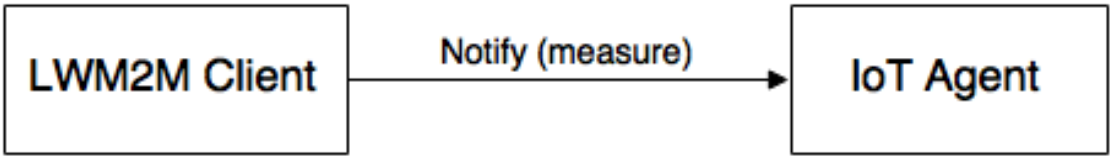


Figure 3.12: Device active observation

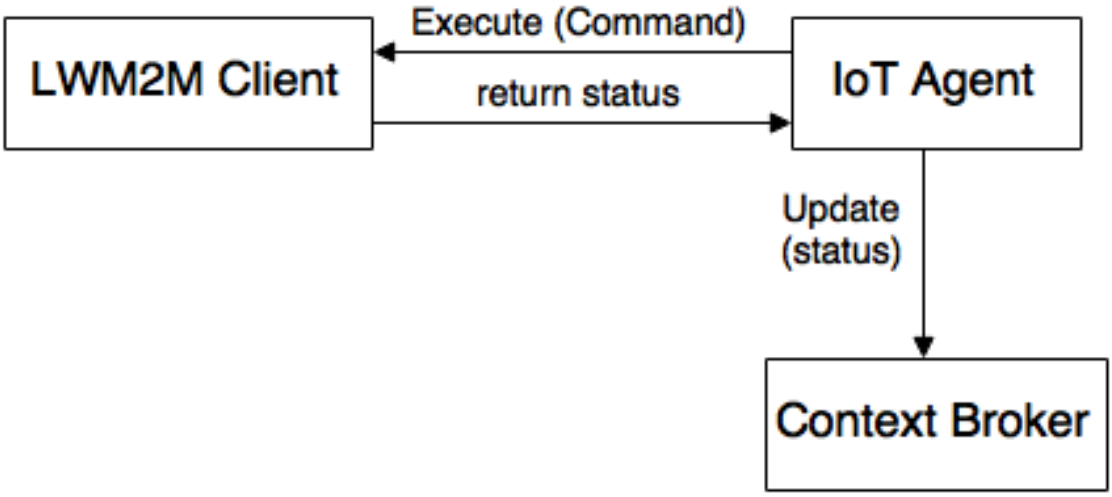


Figure 3.13: Device command

Device active observation For those attributes of the devices marked as active, a Observe operation is issued from the server upon registration, expecting subsequent measures to be issued as Notify responses to the Observe message (figure 3.12).

Device command Commands are issued as changes in a particular value in the Context Entity, and are mapped to Execute operations from the server to the client. The status of the execution is maintained in a special attribute in the Context Entity, that is updated with any upcoming information to the server (figure 3.13).

3.7.2 IoT Broker

The IoT Broker GE is an IoT Backend enabler. It is foreseen to run on a machine in a datacenter, where it serves as a middle-ware. Instead of having to deal with the

technical details of existing FIWARE IoT installations, application developers only need to set up their application to communicate with the IoT Broker in order to retrieve the data they need. The main interface exposed is the FIWARE NGSI. This API has been developed by the FIWARE community as a binding of the OMA NGSI Context Management standard, in particular the interfaces used are OMA NGSI-9/10. The IoT Broker GE retrieves information from IoT Gateways and Devices via the FIWARE NGSI protocol, while the same protocol is can be used by applications to retrieve information from the IoT Broker GE. The two FIWARE NGSI context management interfaces distinguish between two types of information. The first type is called context information and consists of attribute values and associated metadata. This kind of information is exchanged using the operations defined in OMA NGSI-10. The second type of information is context availability information, i.e., information on where context information can be retrieved by OMA NGSI-10 operations. This kind of information is exchanged using the operations defined in OMA NGSI-9. In the figure 3.14 are shown the internal components of the IoT Broker.

As we can see, It represents the point of contact for accessing information about things. Applications can access this information using the NGSI-10 interface exposed. In-fact it interacts potentially with a large number of Gateways, other Backend instances, Devices, and of course data consumers. Furthermore, it typically interacts with at least one instance of the IoT Discovery GE. The latter is where the IoT Broker GE retrieves information about where context information is available in the IoT installation. In the FIWARE architecture, the role of the data consumer is played by the Context Broker GE. The IoT Broker GE communicates with the Context Broker GE via the Northbound Interface. The Southbound interface is used to communicate with the IoT Agents, which are providing data. The role of IoT Agent can be played by either the Backend Device Management GE (IoT Backend), or by the Gateway Data Handling GE (IoT Edge). In case of advanced usage IoT Broker keeps certain kinds of states as represented by the two repositories, the *Subscription Storage* and the *Registration Repository*, as depicted in the figure 3.14. Firstly,

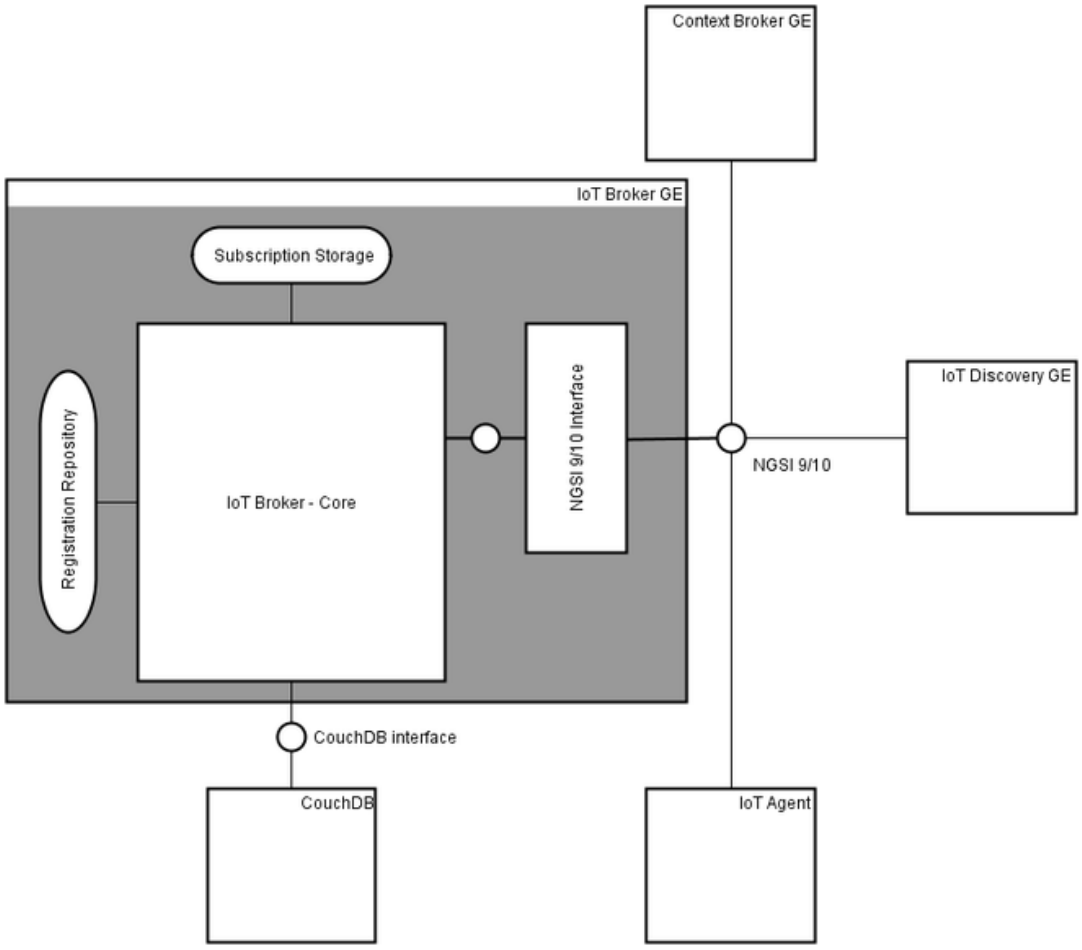


Figure 3.14: IoT Broker GE - internal

it needs to keep track of existing subscriptions. This includes both subscriptions received from applications and subscription issued to IoT Agents by the IoT Broker. Secondly, it includes a repository of context registrations, that are, informations about where certain context information can be retrieved. The operations, exposed by the IoT Broker, for exchanging context informations are described in the NGSI context management interface:

- The first kind of operation is a *Query*. When an application invokes the query, it expects to receive context information as the response;
- The second kind of operation is a *Subscription*. When an application subscribes to certain context information, it just receives a subscription Id as response. Context information is then sent to the application in the form of notifications. Depending on the kind of subscription, context information can be sent whenever attribute values change or simply at fixed time intervals (in these cases the IoT Broker receives a *Notification* message) or whenever an attribute, requested in a previous subscription, is now available (instead in this case the IoT Broker receives a *Availability Notification* message);
- Finally, there is a mode of information exchange defined in the NGSI context interface where information updates are sent asynchronously, called *Update*, by the IoT Agents. When the IoT Broker receives such updates, it forwards the information to Context Broker GE that is responsible for further processing and/or storing such updates.

Query Queries are one-time requests for information. They are realized by the *queryContext* operation of OMA NGSI-10. In reaction to a query, the IoT Broker determines the set of IoT Agents that can provide the requested information. This will be done by sending a *discoverContextAvailability* request to the IoT Discovery GE. After this step, the IoT Broker queries the identified IoT Agents, it aggregates the results and it forwards them to the GE or application that has issued the query.

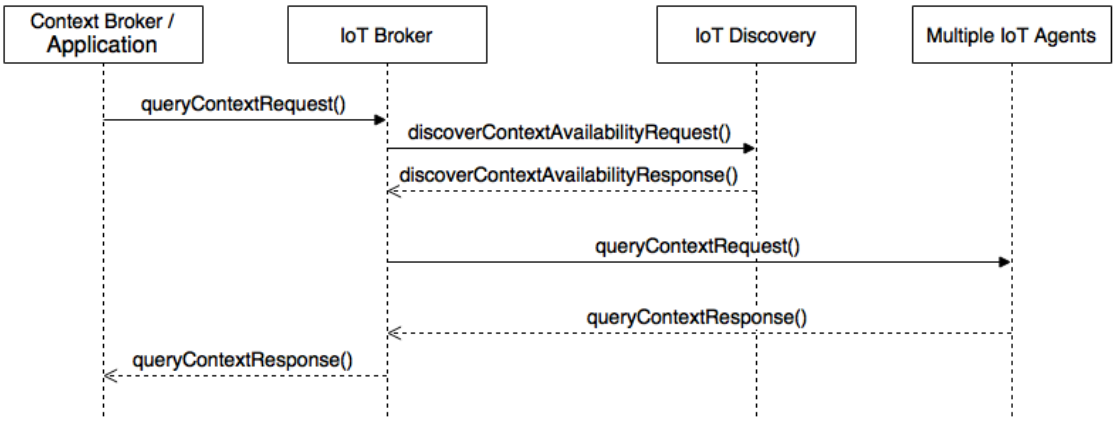


Figure 3.15: IoT Broker - Query

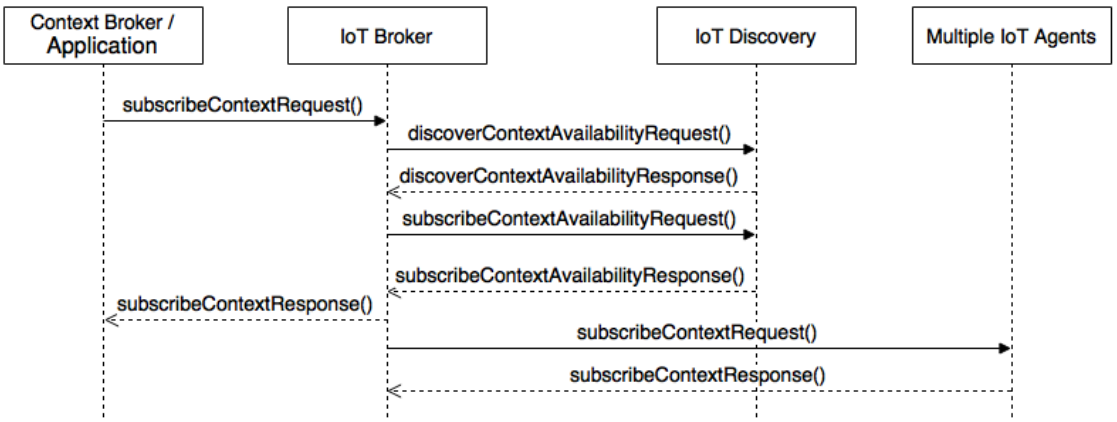


Figure 3.16: IoT Broker - Subscription

In the figure 3.15 are represented the query operations.

Subscription Subscriptions are requests for information updates the issuer wishes to receive, under certain conditions to be specified in the request message. Also in these interactions, like in *Query* operation, the role of the IoT Discovery GE is to provide the relevant information sources, such as the IoT Agents. In the figure 3.16 are depicted the main subscription interactions.

Update In general, updates received by the IoT Broker GE are forwarded to the IoT Discovery GE. Before forwarding, the IoT Broker discovers the Thing-level en-

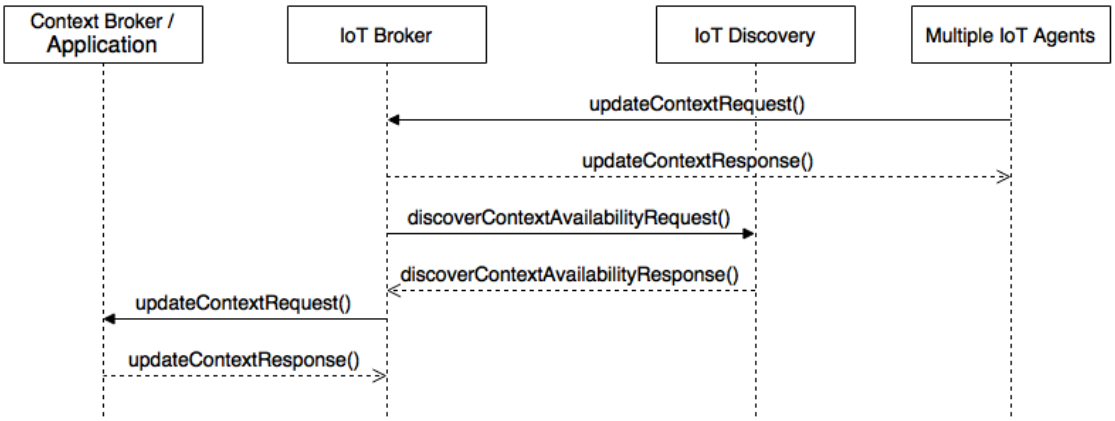


Figure 3.17: IoT Broker - Update

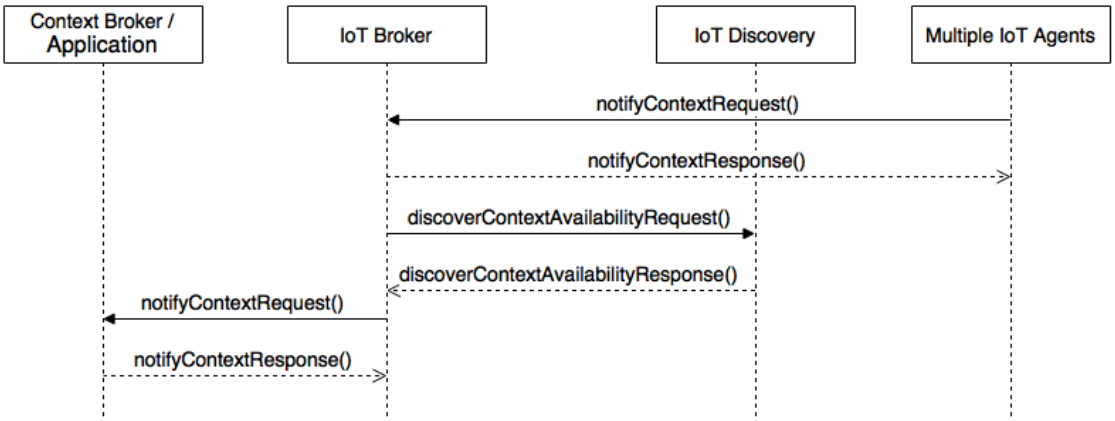


Figure 3.18: IoT Broker - Notification

tities associated to the Device-level entities about which it has received an update. For that reason the IoT Broker contacts the IoT Discovery GE using a discoverContextAvailability request. In the figure 3.17 are depicted the main update interactions.

Notification Notifications are the counterpart of subscriptions. A notification is sent whenever the condition that has been specified in the subscription is satisfied (figure 3.18).

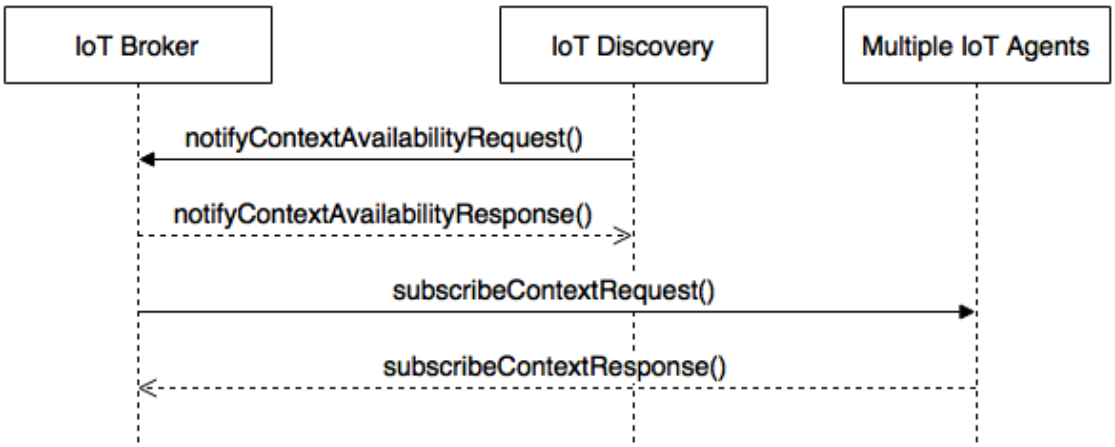


Figure 3.19: IoT Broker - Availability Notification

Availability Notification When a new IoT Agent having informations that are relevant for an existing subscription becomes available, the IoT Broker is notified about this through the availability notification, so that it can then make a subscription to the new IoT Agent (figure 3.19).

3.7.3 IoT Discovery

The IoT-Discovery GE is the part of the Backend tier of the IoT Architecture which is responsible for context source availability management. The underlying data model of this GE is based on the OMA NGSI-9 Context Management Information Model. This model relies on the concept of context entities, which are generic entities whose state is described by the means of values of attributes and associated meta-data. In the context of IoT, context entities and context entity attributes can be used to model IoT resources and the variables they measure. The IoT Discovery GE is responsible for the context availability registrations from IoT Agents, thus making it the access point for information about entities and their attributes. In particular, the context availability information is forwarded from IoT Agents that expose the FIWARE NGSI-9/10 interfaces. The role of IoT Agents can be played by either the Data Handling GE in IoT Gateways, or the Backend Device Management

GE. Using the FIWARE NGSI-9 interface, that the IoT Discovery GE provides, applications and services will be able to register, discover and subscribe to updates on context availability information. In the figure 3.20, we can see the main components of the IoT Discovery GE, that are the *Configuration Manager* and the *Configuration Repository*.

The Configuration Manager component consists of a context information registry in which context provider applications can be registered. In addition, components interacting with the Configuration Manager can perform discovery operations on that context registration information or subscribe to changes on it. The Configuration Repository stores information on the availability of context information and can be accessed through the Configuration Management. When a NGSI-9 client sends a discoverContextAvailabilityRequest operation in order to find out where the desired context information can be found, the Configuration Manager returns zero or more ContextRegistration instances that match the client's request. IoT-Discovery has a series of interfaces for interacting with applications, users, and other GEs within the FIWARE architecture. For example, the GE communicates with the Context Broker GE via the Northbound Interface and with the IoT Agents on the Southbound Interface. The role of IoT Agent can be played by either the Backend Device Management GE, or by the Gateway Data Handling GE. There is also an interface (NGSI-9) between the IoT Broker GE and the IoT Discovery GE. In the context of NGSI communication, IoT Discovery plays the role of the server, and any other actor, interacting with it, play the role of the client.

The operations implemented by the IoT Discovery are:

- Registration;
- Registration Update;
- Discovery;
- Subscription;

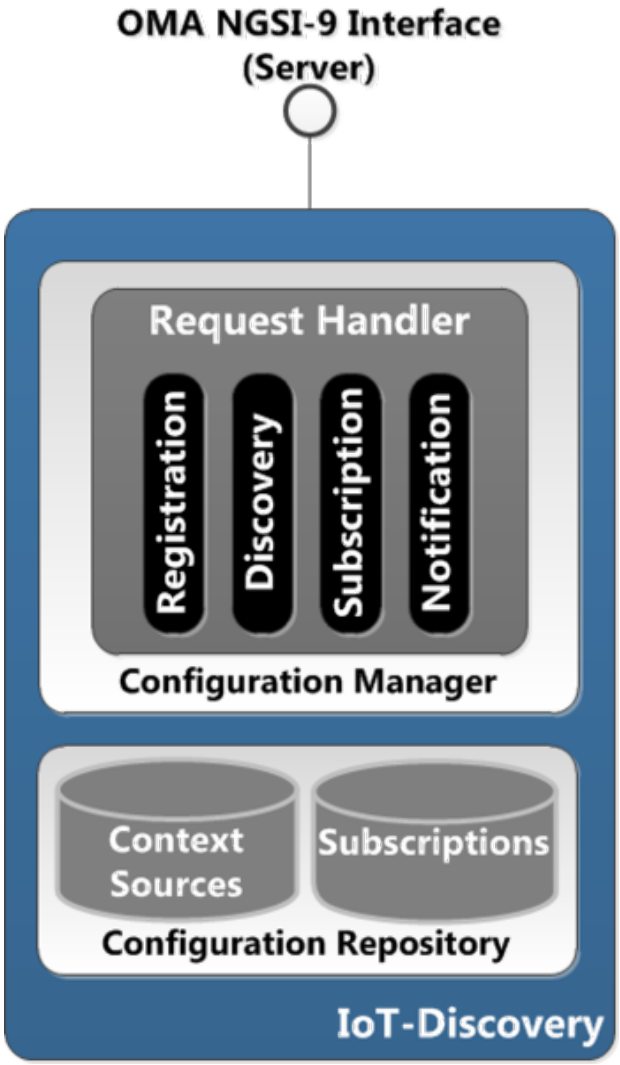


Figure 3.20: IoT Discovery

- Notification.

In the following diagrams all IoT Agents, GEs, and applications, interacting with the IoT Discovery, are abstracted using the term "NGSI-9 Client".

Registration In order for things informations to be available at the Backend, IoT Agents need to register their information to the IoT-Discovery GE. This is done via the registerContext operation (figure 3.21).

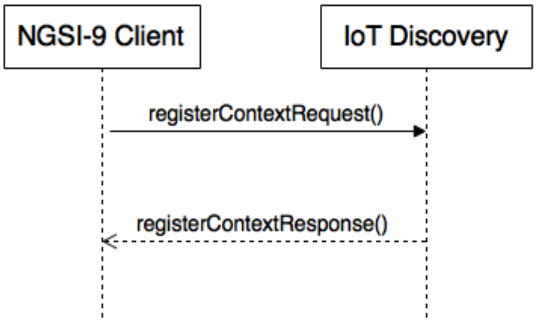


Figure 3.21: IoT Discovery - Registration

Registration Update A context provider can update a registration by using the registration Id that is returned from the previous registration response (figure 3.22).

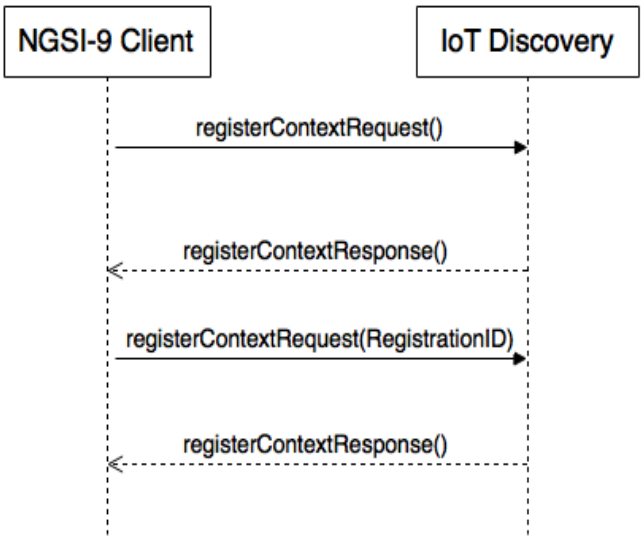


Figure 3.22: IoT Discovery - Registration Update

Discovery The entities that typically play the "NGSI-9 Client" role in this case are the IoT Broker GE (when the requested context availability information to process NGSI-10 query/updates) or any other NGSI application that wants to know the context availability associated with a given entity/attribute (figure 3.23).

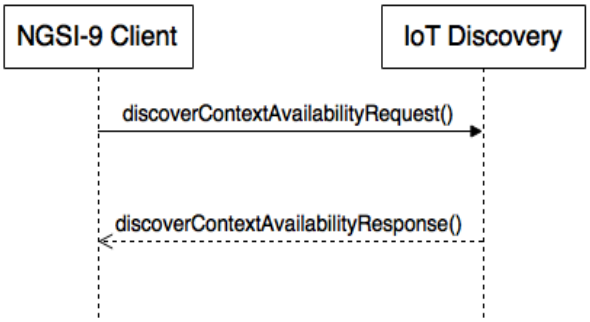


Figure 3.23: IoT Discovery - Discovery

Subscription The entities that potentially could play the "NGSI-9 Client" role are any application (i.e. a Subscriber App) that needs to be aware of changes in context availability information (figure 3.24).

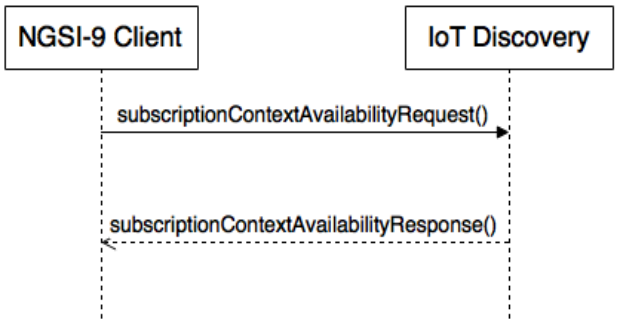


Figure 3.24: IoT Discovery - Subscription

Notification The entities that potentially could play the "NGSI-9 Client" role are IoT Agents, while "Subscriber Application" are applications subscribed to context availability information changes (figure 3.25).

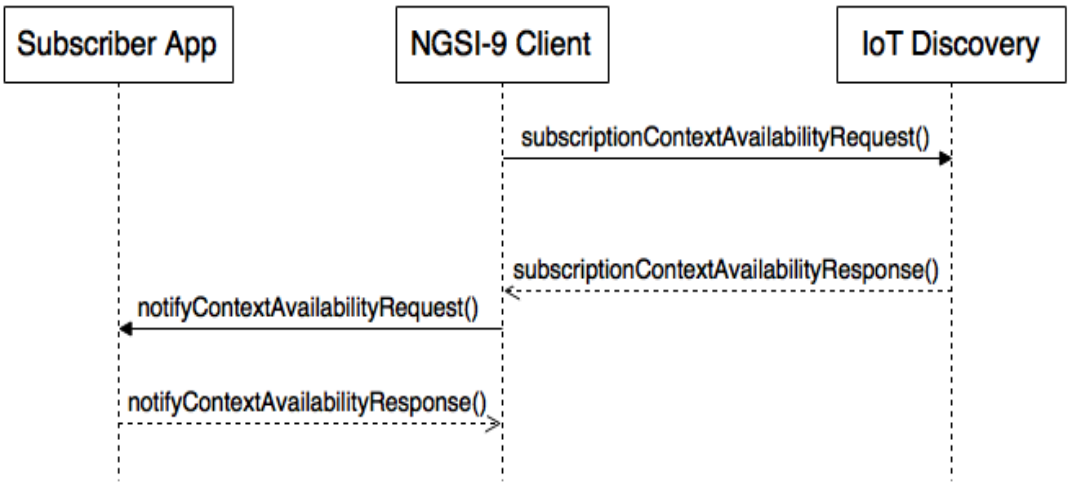


Figure 3.25: IoT Discovery - Notification

3.8 IoT Edge

The IoT Edge includes that GEs that are executed in the IoT Gateways that communicate with the IoT Backend which runs in the cloud. They are the Gateway Logic, the Protocol Adapter and the Data Handling (figure 3.26).

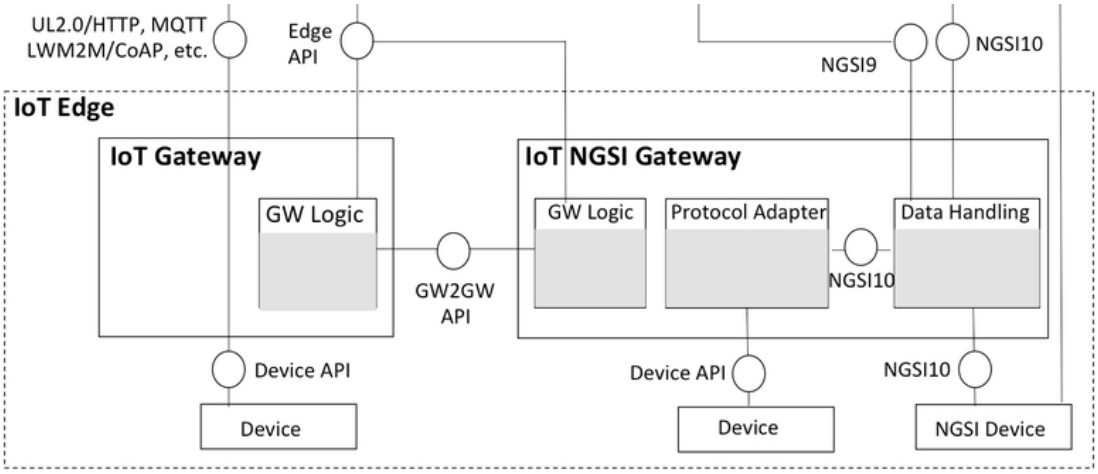


Figure 3.26: FIWARE IoT edge

3.8.1 Gateway Logic

The Gateway Logic GE will handle the IoT Edge management API and functions plus the gateway-to-gateway API. For the first (Edge Management) it needs the correspondent function/module to be activated and configured in the Backend Device Management GE. As we have said before, all these functions are under definition.

3.8.2 IoT Protocol Adapter

The Protocol Adapter GE deals with the incoming and outgoing traffic and messages between the IoT Gateway and registered devices. There may be multiple instances of Protocol Adapter GEs capable of serving not fully IoT compliant devices, i.e. devices that do not support ETSI M2M. These devices can be IP-based devices, that communicates using the IP stack (IPv4 or IPv6), or "legacy devices", meaning devices communicating using non-IP based protocols, for instance ZigBee, or Z-Wave. The Protocol Adapter GE receives these device specific protocols and translates them to a uniform internal API. The exposed API handles capabilities to read and write to the resources, as well as IoT specific management and configuration services such as resource discovery. In particular, the ZigBee Protocol Adapter pro-

vides a communication into a ZigBee PAN(s) (Personal Area Network). It supports a mechanism whereby a gateway can interact with individual ZigBee nodes.

3.8.3 IoT Data Handling

The Data Handling GE addresses the need of filtering, aggregating and merging real-time data from different sources. A lot of applications expect to receive value-added data that are relevant to their needs, and this can be easily achieved in a de-coupled manner thanks to the Complex Event Processing technology (CEP). Events are published by Event Producers and subscribed to by Event Consumers. Typical applications that require Data Handling GE are sensor network applications, RFID readings, supply chains, scheduling and control of fabrication lines, air traffic, smart buildings, home automation, and so on. The CEP Engine component of the Data Handling GE typically processes input data, in order to generate a smaller set of output data, which avoids upper level software overloading and network flooding.

Chapter 4

State of the Art

4.1 Existing QoS algorithms and Models

QoS-aware selection problem has been widely studied in recent years and it has been applied in traditional platforms and using different frameworks. In the literature it can be found many approaches with which the problem has been addressed. In the [8], it is proposed an adaptive method of selecting services based on the hardness of QoS constraints. The basic idea is to sample services that represent a specific quality-value range. Then it is calculated the utility of candidate services in a QoS sub-range and it is sampled the highest utility service. This process of sampling services and evaluating their utility value is repeated until it makes a composite service that has the highest level of global utility for a task. In the [10] we can read about an adaption mechanism that allows a service broker, offering a composite service, to bind at runtime each task of the composite service to a corresponding concrete implementation, selecting it from a set of candidates which differ from one another in terms of QoS parameters. The proposed policy is a load-aware per-request approach which aims to combine the relative benefits of the well known per-request and per-flow approaches. It exploits the multiple available implementations of each abstract task, and realizes a runtime probabilistic binding. Instead, in the [11] the authors address the selection problem, providing a service broker, with

a forward-looking admission control policy based on Markov Decision Processes. This mechanism allows the broker to decide whether to accept or reject a new potential user in such a way to maximize its gain while guaranteeing non-functional QoS requirements to its already admitted users. Another work based on a broker is [13], where authors propose a QoS broker for web service composition that aims at finding the best combination in order to maximize the end-user satisfaction. The proposed approach is based on a utility function that takes into account only end-users without analyzing constraints from server providers. The server providers constrained are take in account in the [14], where the authors propose a QoS-aware adaptive load balancing strategy. Finally, in [15] the authors present a heuristic that aims at finding the optimal allocation that minimizes the overall response time. The proposed solution, however, provides only probabilistic guarantees that cannot support hard real-time applications. The [10], [11], [15] are solutions proposed in the field of SOA and Web Service architectures but they are not suited for IoT deployments, as IoT systems are composed of service providers characterized by constrained capabilities with limited battery capacity [3]. From the our study we have identified, the [9], the [12] and the [3] as works proposing a QoS aware scheduling designed for service oriented IoT platforms. In the [9] the authors propose a solution based on the *Wukong* middleware. It is used to map the abstraction of a smart application onto physical smart devices and actuators. The work is focused on the design of a QoS oriented mapping algorithm for applications with a set of QoS attributes. Wukong find the best mapping solution according to the requirement, after developers specify how each attribute contributes to the overall QoS. In the [12] a multi-layered scheduling model is proposed to evaluate the optimal allocation that meets the QoS requirements of the applications. Different solutions are deployed at different layers to manage different systems resources, such as network resources and IoT services. However, the proposed approach cannot be used for online IoT-service selection, since it is mainly suited for offline provisioning and planning [3]. An online IoT-service selection solution is designed in the [3] where it is formulated

a QoS-aware selection problem for IoT cloud platforms, whose solution minimizes the energy consumption so as to maximize the lifetime of battery powered devices, whilst guaranteeing the fulfilment of real-time QoS requirements. An heuristic algorithm, called *Real Time Thing Allocation* algorithm (RTTA), is implemented to solve the problem that is formalized as an Integer Linear Problem and that results to be an instance of the Agent Bottleneck Generalized Assignment Problem (ABGAP)[16]. At the end of the analysis of all previous works, we have concluded that the [3] is the only one that offers an heuristic that can be directly integrated in the FIWARE IoT platform. In-fact, it has been already integrated in the *BETaaS* platform, designed for the development and execution of M2M applications in the IoT context.

4.2 Real Time Thing Allocation algorithm

4.2.1 Problem Formulation

As it is described in the [3], the problem is formulated on a given system of *n* things, each exposing a subset of IoT services and a set of *k* requests for service invocation. Each service *j* is assumed to be invoked periodically with period p_j. Thing *i* is assumed to be a constrained device capable of satisfying only one service invocation at time. Moreover, a thing may be battery-powered. Let b_i, be the battery capacity on a thing *i*. Each invocation of a service *j* on a thing *i* has a fixed execution time t_{ij}, including both communication and computation times, and a fixed energy cost c_{ij}, representing the overall amount of energy needed to accomplish the invocation of service *j* on thing *i*. The cost of execution of a service on a thing has a different impact depending on the initial battery level of the thing. In order to make a fair comparison among costs, they consider costs of execution over a common (hyper-)period *h* and normalized with respect to the available energy b_i. The hyper-period *h* is computed as the least common multiple among all request period p_j; the normalized energy cost f_{ij} of executing service *j* on thing *i* is given by:

$$f_{ij} = \frac{h}{pj} \frac{c_{ij}}{b_i}$$

Not all services can be invoked on any thing, but the same service can be invoked on multiple equivalent things. Equivalence among things is based on context information associated to thing services, which we assume is provided by m_{ij} as follows

$$m_{ij} = \begin{cases} 1 \text{ if service } j \text{ can be invoked on thing } i \\ 0 \text{ otherwise} \end{cases}$$

They assume that each service request can be executed by at least one thing. The utilization u_{ij} of allocating requests of service *j* on thing *i* is defined as

$$u_{ij} = \begin{cases} \frac{t_{ij}}{p_i} \text{ if } m_{ij} = 1 \\ +\infty \text{ otherwise} \end{cases}$$

The thing allocation problem is then to allocate the *k* requests to the *n* things so as to minimize the maximum (normalized) energy cost among things over a hyperperiod *h*, whilst guaranteeing that all service invocations are completely executed before their implicit deadline. As we have seen, the problem is an Integer Linear Problem that results to be an instance of the Agent Bottleneck Generalized Assignment Problem (ABGAP) [3].

4.2.2 RTTA Algorithm

The proposed heuristic, named *Real Time Thing Allocation algorithm* (RTTA), is a novel greedy polynomial-time heuristic algorithm to solve the ABGAP. The *RTTA* pseudo-code can be viewed in the [3]. The input to RTTA are: the number of things *n*, the number of requests *k*, the normalized energy cost matrix **F** = f_{ij}, the utilization matrix **U** = u_{ij}, a precision *threshold* ε, and, finally, an optional priority matrix **P** = p_{ij}. The latter is used to steer the thing allocation procedure *Feas* described in detail in [3]. The output is: a boolean *isFeasable*, which takes the *True* value if at least one allocation exists, the allocation vector **y** that maps service requests to things, and the corresponding residual battery vector **z**. As explained in the [3], the rationale behind RTTA is to iteratively search for the first feasible allocation that

guarantees the highest minimum level of residual battery for all things. To this aim, RTTA leverages the procedure *Feas* that, given a threshold θ, finds an allocation so that the residual battery on each thing after service invocation is no lower than θ. On every iteration, the threshold θ is decreased until a feasible solution is found with an acceptable precision level measured by ε. More specifically, a binary search strategy is used to reduce the time needed to execute the overall procedure. The core of the RTTA algorithm is the procedure *Feas*. The pseudo-code of the *Feas* algorithm is reported in [3]. The allocation is based on the values of a priority matrix **P** passed as input. In particular, element p_{ij} of **P** is a measure of the desirability of allocating request *j* to thing *i*. All requests are then considered iteratively for allocation. At each step, the next request to allocate, say *j*, is the one having the maximum difference between the largest and the second largest p_{ij} (for all things *i* such that deadline constraint is met). Request *j* is then allocated to the thing *i* for which p_{ij} is a maximum. If a service request for which no feasible assignment is found, i.e., any possible allocation to a thing implies its residual battery level goes below θ, the algorithm returns *isFeasable* equal to False.

Chapter 5

Design QoS solution in FIWARE IoT Architecture

Before to get in deep with the design aspects, we introduce the main NGSI operations and data structures considered to implement our FIWARE QoS solution.

5.1 NGSI Information model

The basic element of the NGSI Information model is the *Context Entity*. There are two types of Context Entities: the *Context Element*, used to represent the state of a resource (i.e. battery level, coordinates) and the *Context Registration Element*, used to reach information on a resource, in particular where they can be reached. The Context Entity is composed by *attributes*, used to represent the informations about a resource. In the our model, a Context Entity represents the abstract view of a thing. The attributes represent the resources offered by a thing (i.e temperature, pressure measurements). To each attribute are associated metadata, that represent parameters about a thing service (i.e. latency, energy cost). The metadata are used to differentiate equivalent services in terms of QoS and costs. In the following we present the main operations and data structures used by the NGSI-9/10 standard for the context management.

5.1.1 NGSI-9 operations

registerContext This operation allows registering and updating of registered Context Entities, their attribute names and availability. The *ProvidingEntity URI* is used to identify the entity that provides the values of context attributes for registered Context Entities.

- **Input Message: registerContextRequest**

Element name	Element type	Optional	Description
ContextRegistrationList	ContextRegistration [1..unbounded]	No	List of ContextRegistration structures.
Duration	xsd:duration	Yes	Desired availability period.
RegistrationId	xsd:string	Yes	Registration identifier used to update previous registrations.

- **Output message: registerContextResponse**

Element name	Element type	Optional	Description
Duration	xsd:duration	Yes	Confirmed availability period.
RegistrationId	xsd:string	No	Registration identifier that could be used to update this registration.
ErrorCode	StatusCode	Yes	Error reported by the operation.

discoverContextAvailability This operation allows the synchronous discovery of the potential set Context Entities, types of Context Entities and related Context Information that can be provided. This is only checked against the registrations issued in the register operation, not against the real source of the Context Information. In other terms, the discovery operation provides an aggregated list of Context Registration.

- **Input Message: discoverContextAvailabilityRequest**

Element name	Element type	Optional	Description
EntityId	EntityId [1..unbounded]	No	List of Modifier to identify the Context Entity(ies) to discover.
AttributeList	xsd:string [0..unbounded]	Yes	List of attributes or group of attributes to discover.
Restriction	Restriction	Yes	Restriction on the attributes and meta-data of the Context Information.

- **Output message: discoverContextAvailabilityResponse**

Element name	Element type	Optional	Description
ContextRegistration ResponseList	ContextRegistrationResponse [0..unbounded]	Yes	List of Context Registration responses.
ErrorCode	StatusCode	Yes	Error codes for general operation errors.

subscribeContextAvailability This operation allows the asynchronous discovery of the potential set Context Entities, types of Context Entities and related Context Information that can be provided. This does not guarantee that Context Information about a Context Entity within this set is currently available. In other terms, this operation allows to subscribe to the notification on the availability of an aggregated list of Context Registrations.

- **Input message: subscribeContextAvailabilityRequest**

Element name	**Element type**	**Optional**	**Description**
EntityId	EntityId [1..unbounded]	No	List of identifiers or name patterns of the Context Entity(ies) to discover.
AttributeList	xsd:string [0..unbounded]	Yes	List of attributes or group of attributes to be discovered.
Reference	xsd:anyURI	No	The interface reference for the notifyContextAvailability operation.
Duration	xsd:duration	Yes	Requested duration of the subscription.
Restriction	Restriction	Yes	Restriction on the attributes and meta-data of the Context Information.
SubscriptionId	xsd:string	Yes	Used in the notification message and subsequent requests.

- **Output message: subscribeContextAvailabilityResponse**

Element name	Element type	Optional	Description
SubscriptionId	xsd:string	No	The identifier of the subscription.
Duration	xsd:duration	Yes	Negotiated duration of the subscription.
ErrorCode	StatusCode	Yes	Error reported by the operation.

notifyContextAvailability This operation allows receiving the notification about the potential set of Context Registrations subscribed to by the subscriber that implements the notification interface.

- **Input message: notifyContextAvailabilityRequest**

Element name	Element type	Optional	Description
SubscriptionId	xsd:string	No	The identifier of the subscription to which the notification belongs to.
ContextRegistrationResponse List	ContextRegistration Response [1..unbounded]	Yes	List of Context Registration responses.
ErrorCode	StatusCode	Yes	Error codes for general operation errors.

- **Output message: notifyContextAvailabilityResponse**

Element name	Element type	Optional	Description
ResponseCode	StatusCode	No	Status codes for general operation errors.

5.1.2 NGSI-10 operations

queryContext This operation allows for the synchronous retrieval of Context Information. The requestor of queryContext operation shall specify a list of entity identifiers. Such identifiers may represent unique entities or entity identifier patterns. An entity identifier patterns shall be represented as regular expressions. The requestor of this operation may also specify the list of attributes to be retrieved by this operation. The entity identifiers may include type attributes to specify the type of target entities. It is assumed that the requestor is aware of possible entity types and attributes through *Context Entity Discovery* operations or by other means. The requestor of this operation may specify restrictions on the returned Context Information. Restrictions are based on the values of attributes and meta-data of the Context Information.

- **Input message: queryContextRequest**

Element name	Element type	Optional	Description
EntityIdList	EntityId [1...unbounded]	No	List of identifiers of the Context Entity(ies) for which the Context Information is requested. Identifiers can contain patterns represented as regular expressions.
AttributeList	xsd:string [0...unbounded]	Yes	List of ContextAttributes and/or AttributeDomains that are queried.
Restriction	Restriction	Yes	Restriction on the result set of the query. Restrictions are based on the values of attributes and metadata of the Context Information.

- **Output message: queryContextResponse**

Element name	Element type	Optional	Description
ContextResponseList	ContextElementResponse [0...unbounded]	Yes	List of Context Information, related attributes (or group of attributes) and metadata.
ErrorCode	StatusCode	Yes	Error codes for general operation errors.

subscribeContext This operation allows the asynchronous retrieval of Context Information. It is used for subscription to Context Information. The subscription triggers the notifications about the matching ContextEntities based on the defined NotifyCondition information passed in the subscribeContextRequest operation. In the subscribeContextResponse operation a subscription id is returned, which is used for notifications and in update and unsubscribe operations. The subscription duration is negotiated during the subscription request/response operation.

- **Input message: subscribeContextRequest**

Element name	**Element type**	**Optional**	**Description**
EntityIdList	EntityId [1...unbounded]	No	List of identifiers of the Context Entity(ies) for which the Context Information is requested. Identifier can contain patterns represented as regular expressions.
AttributeList	xsd:string [0...unbounded]	Yes	List of ContextAttributes and/or AttributeDomains to which the requestor wants to subscribe.
Reference	xsd:anyURI	No	URI that identifies the interface where the notifyContext operation shall be invoked.
Duration	xsd:duration	Yes	Requested duration of the subscription.
Restriction	Restriction	Yes	Restriction on the attributes and metadata of the Context Information.
NotifyConditions	NotifyCondition [0...unbounded]	Yes	Conditions when to send the notifications.
Throttling	xsd:duration	Yes	Proposed minimum interval between notifications.

- **Output message: subscribeContextResponse**

Element name	Element type	Optional	Description
SubscribeResponse	SubscribeResponse	Yes	Response to the subscribeContextRequest.
SubscribeError	SubscribeError	Yes	The error reported by the receiver of the request.

notifyContextRequest This operation allows receiving the notification about the Context Information subscribed to by the subscriber that implements the notification interface.

- **Input message: notifyContextRequest**

Element name	Element type	Optional	Description
SubscriptionId	xsd:string	No	The identifier of the subscription to which the notification belongs to.
Originator	xsd:anyURI	No	The original requestor of the subscription which caused this notification.
ContextResponseList	ContextElementResponse [0...unbounded]	Yes	List of Context Information, related attributes (or group of attributes) and metadata.

- **Output message: notifyContextResponse**

Element name	Element type	Optional	Description
ResponseCode	StatusCode	No	The response message reported by the receiver of the request.

updateContext This operation allows updating a set of Context Information, related attributes and metadata. For each ContextElement of the list of Context Elements received in the updateContextRequest, if an empty Context Value is provided, the operation behaviour shall be:

1. if the UpdateAction is set to “update” or “append”, the receiver shall reject the related changes requested for the specific ContextElement and report an error in the response;

2. If the UpdateAction is set to “delete”, the receiver shall ignore the ContextValue parameter, perform the related changes requested (delete) and report a success in the response.

- **Input message: updateContextRequest**

Element name	Element type	Optional	Description
ContextElementList	ContextElement [1...unbounded]	No	List of Context Elements containing only the subset of Context Information (related attributes (or context domain) and metadata) to be modified.
UpdateAction	UpdateActionType	No	Indicates the type of action that is performed within the update operation.

- **Output message: updateContextResponse**

Element name	Element type	Optional	Description
ErrorCode	StatusCode	Yes	Error codes.
ContextResponseList	ContextElementResponse [0...unbounded]	Yes	List of response containing the indication of the Context Element and the related statusCode.

5.1.3 Data Structure definition

ContextElement structure In the *Context Element* are stored context values. It is used in the *ContextElementList* parameter of the *updateContext Request* operation and the ContextElement field of the *ContextElement Response* structure. *ContextAttribute* element is used to store values about a thing, such as battery level and position in terms of latitude and longitude. Metadata can be stored in *Metadata* field of the ContextAttribute element or in the *DomainMetadata* element.

Element name	**Element type**	**Optional**	**Description**
EntityId	EntityId	No	Identifies the Context Entity for which the Context Information is provided.
AttributeDomainName	xsd:string	Yes	Name of the attribute domain that logically groups together set of Context Information attributes. Examples of attribute domain are: device info (battery level, screen size, ...), location info (position, civil address, ...).
ContextAttribute	ContextAttribute [0...unbounded]	Yes	List of Context Information attributes. Note: In case of the attributeDomainName is specified all contextAttribute have to belong to the same attributeDomainName.
DomainMetadata	ContextMetadata [0..unbounded]	Yes	Metadata common to all attributes of the logical domain (related to the AttributeDomain).

ContextRegistration structure *Context Registration* structure is used in the *ContextRegistrationList* parameter of registerContext operation and the *ContextRegistration* field of the ContextRegistration response structure. This structure can be used either to register/update the information about Providing Application or to register/update the availability of ContextEntities and their related attributes.

Element name	**Element type**	**Optional**	**Description**
EntityIdList	EntityId [1...unbounded]	Yes	List of identifiers for the Context Entities being registered.
ContextRegistration AttributeList	ContextRegistrationAttribute [0...unbounded]	Yes	List of ContextAttributes and/or AttributeDomains which are made available through this registration.
RegistrationMetadata	ContextMetadata [0...unbounded]	Yes	Metadata characterizing this registration.
ProvidingApplication	xsd:anyURI	No	URI identifying the application that provides the values of the context attributes for the target Context Entities.

EntityId structure ContextEntities are identified using an entity identifier. The optional entity type may be needed when the EntityId doesn't contain type information or when the EntityId is only unique per entity type.

Element name	**Element type**	**Optional**	**XML Type**	**Description**
ID	xsd:string	No	element	Identifier of the Context Entity(ies). This value may be a string following the anyURI restrictions or a pattern represented.
Type	xsd:anyURI	Yes	attribute	Indicates the type of Context Entity(ies) for which the Context Information is requested. If EntityId uniqueness is only guaranteed in combination with Type, then Type shall be present.
IsPattern	xsd:Boolean	Yes	attribute	Indicates whether the EntityId is a pattern or an id. If this attribute is omitted, it shall be treated as false.

ContextAttribute structure A context attribute represents atomic Context Information. An attribute is defined as a set of information, namely a name, a type, a value and a set of associated metadata.

Element name	Element type	Optional	Description
Name	xsd:string	No	Name of the Context Information attribute.
Type	xsd:anyURI	Yes	Indicates the type of the value field.
ContextValue	xsd:any	No	The actual value of the Context Information attribute.
Metadata	ContextMetadata [0..unbounded]	Yes	Metadata about the Context Information attribute.

ContextRegistrationAttribute structure Equal to the previous element except for the ContextValue. This element is part of the ContextRegistration structure so it is used to expose the services offered by a resource.

Element name	Element type	Optional	Description
Name	xsd:string	No	Name of the ContextAttribute and/or AttributeDomain.
Type	xsd:string	Yes	Indicates the type of the ContextAttribute value.
isDomain	xsd:boolean	No	Indicates if this structure refers to a ContexAttribute or a AttributeDomain.
Metadata	ContextMetadata [0..unbounded]	Yes	Metadata about the Context Information attribute.

ContextMetadata structure It is used to represent any type of metadata.

Element name	Element type	Optional	Description
Name	xsd:string	No	Name of metadata.
Type	xsd:anyURI	Yes	Indicates the type of the value field.
Value	xsd:any	No	The actual value of the metadata.

Restriction structure It is used to express constrains to reduce the search space of the Context management component. It is used in the *queryContext* request, *discoverContextAvailability* request and *subscribeContextAvailability* request. There are two kind of restrictions:

- *AttributeExpression*, that filters the result set based on expressions on the values of the context attributes.
- *Scope*, that compared to AttributeExpression, a-priori limit the set of context sources that are needed for serving the request.

Element name	Element type	Optional	Description
AttributeExpression	xsd:string	No	Name of metadata String containing an XPath restriction.
Scope	OperationScope [0..unbounded]	Yes	List of scope definition.

OperationScope structure

Element name	Element type	Optional	Description
ScopeType	xsd:string	No	Name of the scope type.
ScopeValue	xsd:any	Yes	Contains the scope value for the defined scope type.

5.2 Design of the FIWARE IoT QoS support

Given the set of IoT GEs, we have addressed the problem to identify a meaningful deployment of these software components. The IoT GEs selected are: *IoT Discovery*, *IoT Broker* and *Device Management*, of which we use the IoT Agent- *LightWeightM2M over CoAP* component (that works as adapter to communicate with LWM2M/CoAP devices). Moreover, the IoT Broker has been wrapped in the new implemented module called *QoSBroker*. The deployment is depicted in the figure 5.1.

We can see as the deployment configuration can be split in two side. The lower one in which there are *IoT Agent* and devices that communicate using the LightWeight M2M protocol. The main function of the IoT Agent component is mapping between the LWM2M protocol and the NGSI protocol. In-fact it is evident as, all the other GEs in the upper side, exchange context information using the NGSI-9/10 protocol.

5.2.1 QoS model design

The QoS model has been built starting from the data structures defined in the NGSI-9/10 information model. In a QoS context, a client application should negotiate a SLA if it wants to use a specific service with a guaranteed QoS level. In our model we provide two phases, *Allocation* and *Dispatching*. In the first one, the client forwards a service request specifing a series of restrictions and a QoS level. For example, an application can subscribe a temperature measurements service, specifing a geographical scope and a maximum response time but especially a rate with which it wishes to receive data. The previous parameters represent the content of the *Service Agreement Offer* that the client sends to the instance of the *QoSBroker* to establish a SLA for a particular service. Naturally, the offer must be compliant to a template that provides sections in which it can be specified the QoS requirements and features of the required service. These will be used to identify a single/group of things that can satisfy the request respecting the parameters request.

A *Service Agreement* can be establish for the following NGSI-10 operations:

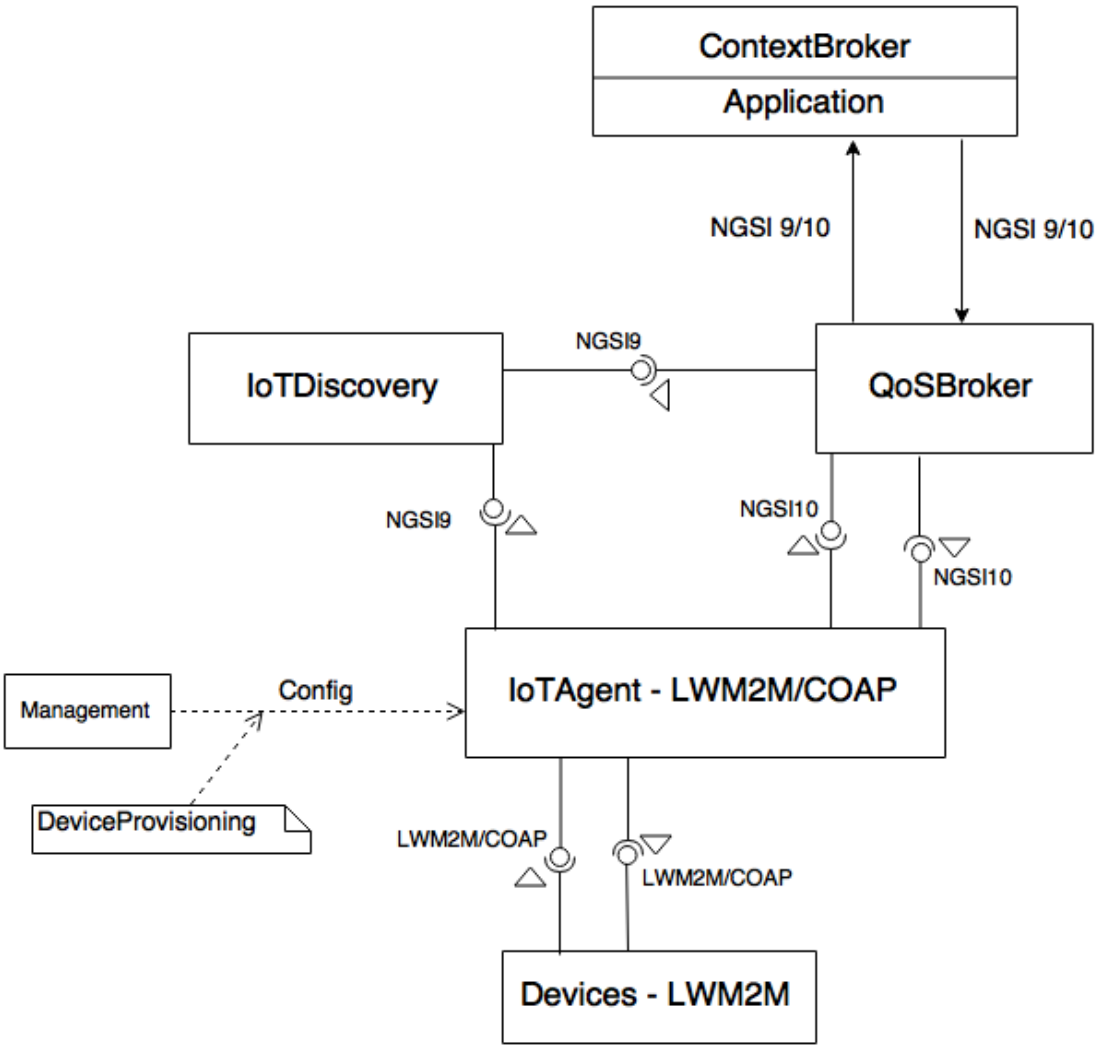


Figure 5.1: FIWARE IoT Deployment

- *queryContext*;
- *updateContext*;
- *subscribeContext*.

The *Dispatching*, represent the phase in which a service request, for which a SLA has been established, is carried out. The SLA is uniquely identified through the id generated in the *Allocation* phase. FIWARE IoT platform doesn't provide an *Allocation* phase of the resources, so we have created a new operation, called *Service Agreement*, through which a client can specified a SLA for a specific service, by means of the *QoSBroker*. This operation is carried out through a specific template built using data structures taken from the NGSI information model. In the following we describe the *Service Agreement* operation along with the data structures through which a SLA can be established.

ServiceAgreement operation This operation allows the creation of a service level agreement. The input for this operation is the *ServiceAgreementRequest* element, composed by a list of *ServiceDefinition* elements. This structure provides a list of *EntityId* through which discover a set of Context Entities. In our model they represent the abstraction of a thing. Each Context Entity is composed by a list of attributes that are the list of resources exposed by a thing. The discovery of the Context Entities can be carried out in three way:

- using the Context Entity type and the id of the Context Entity expressed as regular expression;
- using only the type of the Context Entity;
- using both the Context Entity type and the id of the Context Entity.

In the tables below, we present the data structures used in the Service Agreement operation.

ServiceAgreement Request

Element name	Element type	Optional	Description
ServiceDefinition List	serviceDefinition [0...unbounded]	No	Element containing the QoS requirements and features of the required service.

ServiceDefinition structure Data structure that distinguishes a particular SLA. It provides the list of required services along with the parameters that define the QoS level.

Element name	Element type	Optional	Description
operationType	xsd:string	No	Type of operation (queryContext, updateContext, subscribeContext).
EntityIdList	EntityId [1...unbounded]	No	List of Modifier to identify the Context Entity(ies) to discover.
AttributeList	xsd:string [1...unbounded]	No	List of attributes to discover.
Restriction	Restriction	No	Restriction NGSI element to specify QoS requirements and features of the required service.

Restriction structure We use this NGSI element to specify the QoS requirements (maximum response time and maximum request rate) of the required service. It is also used to filter the thing based on geographical scope, that can be describe as a specific point (latitude and longitude) or a circular area (center latitude, center longitude and radius). In particular, we have defined new *OperationScope* (NGSI information model) elements. One for the QoS parameters and two for the geographical scopes. *Scope* structure taken from NGSI information model:

Element name	Element type	Optional	Description
Scope	OperationScope [0..unbounded]	No	List of scope definition.

Here we can figure out the *OperationScope* structure.

Element name	Element type	Optional	Description
ScopeType	xsd:string	No	Name of the scope type.
ScopeValue	xsd:any	No	scope value for the defined scope type.

The new *scopeTypes* are:

- **Circle**: scope type used to specify a circular geographical scope for the selection of the things;
- **Point**: scope type used to specify things in a particular point of interest;
- **QoS**: scope type to specify QoS parameters in a *Service Agreement Request*.

For the **Circle** scope type, the structure is:

Element name	Element type	Optional	Description
centerLatitude	xsd:float	No	Latitude of the center point of the circular area.
centerLongitude	xsd:float	No	Longitude of the center point of the circular area.
radius	xsd:float	No	Radius of the circular area.

Instead for the **Point** scope type, the structure is:

Element name	**Element type**	**Optional**	**Description**
Longitude	xsd:float	No	Longitude of the thing that provides the service.
Latitude	xsd:float	No	Latitude of the thing that provides the service.

Finally, the **QoS** scope value described using the following structure:

Element name	**Element type**	**Optional**	**Description**
maxResponseTime	xsd:float	No	maximum response time of the required service.
maxRequestRate	xsd:float	No	minimum inter-request time between two different requests for the same service.

To clarify what we have said until now, we present an example of Service Agreement Request.

```
<?xml version="1.0" encoding="UTF-8"?>
<serviceAgreementRequest>
  <serviceDefinition>
    <operationType>queryContext</operationType>
      <entityIdList>
        <entityId type="environment" isPattern="true">
          <id>.*</id>
        </entityId>
      </entityIdList>
      <attributeList>
        <attribute>temperature</attribute>
      </attributeList>
      <restriction>
        <scope>
          <operationScope>
            <scopeType>Point</scopeType>
            <scopeValue>
              <latitude>30</latitude>
              <longitude>43.656998</longitude>
            </scopeValue>
          </operationScope>
          <operationScope>
            <scopeType>QoS</scopeType>
            <scopeValue>
              <maxResponseTime>15</maxResponseTime>
              <maxRequestRate>15</maxRequestRate>
            </scopeValue>
          </operationScope>
        </scope>
      </restriction>
  </serviceDefinition>
</serviceAgreementRequest>
```

In our model we use the *ContextRegistration* element, not only to represent the reachability information about a thing, but also to represent the services that are exposed by a thing (that is the abstraction of a physical device). So each attribute, in the ContextRegistration, matches a thing service. A series of parameters are associated to each service to characterize it from a QoS point of view. The parameters are: *latency*, that is compared with the maximum response time value of a service request and *energy cost*, that gives a measurement of how much the execution of the service reduces the thing battery level. A *ContextRegistration* structure, representing the information about a thing, is created in the IoT Discovery when it receives a *registerContext* message from the IoT Agent. In-fact the IoT Agent component carries out a registerContext operation every time a new device establishes a connection to it. As we will see in the next section, the data to build a ContextRegistration element are taken from the information provided in the Device Provisioning to the IoT Agent. In the following we show an example of ContextRegistration element used to describe the information about a thing.

```
<?xml version="1.0" encoding="UTF-8"?>
<contextRegistration>
  <entityIdList>
    <entityId type="temperature" isPattern="false">
      <id>sensor_1:environment</id>
    </entityId>
  </entityIdList>
  <contextRegistrationAttributeList>
    <contextRegistrationAttribute>
      <name>temperature</name>
      <type>temperature</type>
      <isDomain>false</isDomain>
      <metadata>
        <contextMetadata>
          <name>latency</name>
          <type>float</type>
          <value>2</value>
        </contextMetadata>
        <contextMetadata>
          <name>energyCost</name>
          <type>float</type>
          <value>0.07</value>
        </contextMetadata>
      </metadata>
    </contextRegistrationAttribute>
  </contextRegistrationAttributeList>
  <providingApplication>http://localhost:4041/ngsi10</providingApplication>
</contextRegistration>
```

The *ContextElement* is used to store the value of the battery level and the coordinates of a thing. These values are received by the IoT Agent through asynchronous updates coming from the devices. As specified in the next section, every time the IoT Agent receives a notify message from a device, it carries out an updateContext

operation that forwards to the *QoSBroker* which stores the updated values in its own repository. In this way a new ContextElement is created or, if the values were relative to an existing ContextElement structure, an update is carried out. Here we can see an example of ContextElement.

```
<?xml version="1.0" encoding="UTF-8"?>
<contextElement>
  <entityId type="temperature" isPattern="false">
    <id>sensor_1:environment</id>
  </entityId>
  <contextAttributeList>
    <contextAttribute>
      <name>battery</name>
      <type>float</type>
      <contextValue>80.0</contextValue>
    </contextAttribute>
    <contextAttribute>
      <name>coords</name>
      <type>coords</type>
      <contextValue>"45.0,56.1"</contextValue>
    </contextAttribute>
  </contextAttributeList>
</contextElement>
```

5.2.2 Allocation phase

In the sequence diagram of the figure 5.2 is depicted the allocation phase in which can be established a SLA for a specific service request. This operation is carried out by means of a *Service Agreement Request*.

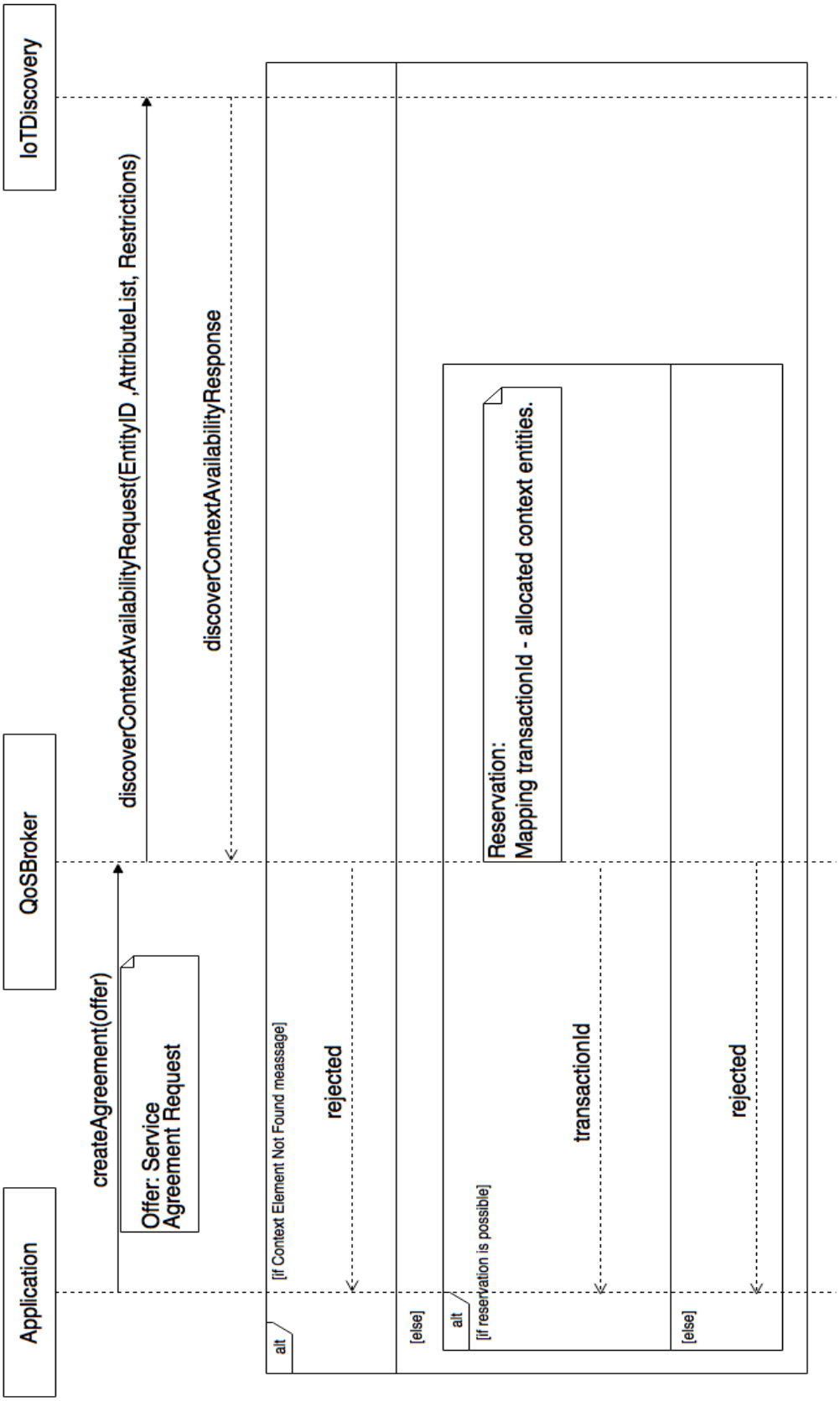


Figure 5.2: Allocation phase sequence diagram

As we can see in the previous figure, the negotiation starts with an service agreement offer sent by the client application. In the offer are expressed the required services (i.e. temperature measurements), the type of operation (i.e. queryContext), the QoS parameters (i.e. maximum response time and maximum request rate) and the geographical requirements. The *QoSBroker* starts a discovery procedure using a regular expression as entityId, specifying an attribute list (names of the required services) and the restrictions taken from the agreement offer (except for the QoS restrictions, that are used in the execution of the service selection algorithm). The IoT Discovery replies with the *discoveryContextAvailabilityResponse* that is processed by the *QoSBroker*. If the response is “ContextElement not found”, the allocation phase terminates immediately because no reservation is possible. In this case the service request is rejected. If the response is a "ContextRegistrationResponseList" the allocation phase can take place. For every *ContextRegistrationResponse* element the corresponding *ContextElement* is retrieved from the internal *QoSBroker* repository, to peak the the battery level and the coordinates values relative to each thing. Infact the previous values are used to represent the abstration of a device, that is *Thing*. This structure is composed by the battery level, the coordinates and the list of services offered by a device. This information together with the QoS restrictions (specified in the service agreement request) are used in the heuristic algorithm to compute an allocation schema that will represents the SLA of that service request. If a feasible allocation is not found, the service request is rejected. Otherwise, it means that a reservation is possible, so a transactionId is sent to the user. The transactionId uniquely identifies the SLA. It is used to uniquely identifies the Context Entity in which the allocation schema is stored (it will be the id of the entityId of the Context Entity). The transactionId is composed by the pre-position “QoS” to distinguish it, from any other entityId. The allocation schema is stored in the internal repository of the *QoSBroker* using a *ContextRegistration* element, as explain below:

ContextRegistration structure It is used to store the allocation schema. It represents the mapping between services guaranteed in the SLA and the things to which the services are allocated. The pointer to the things is the entityId of those Context Entities through they are described. We have used the same fields of the NGSI *ContextRegistration* structure but in our model they have a different meaning.

Element name	**Element type**	**Optional**	**Description**
EntityIdList	EntityId [1..unbound]	No	TransactionId to uniquely identify the SLA.
ContextRegistrationAttrbute List	ContextRegistration Attribute [0...unbounded]	No	List of attributes that represent the services granted in the SLA.
RegistrationMetadata	ContextMetadata [0...unbounded]	Yes	Metadata characterizing the SLA.

ContextRegistrationAttribute structure This element is used to describe the association between a service and the entityId that uniquely identifies a ContextRegistration element. It represents the thing that offers that service. A service can be associated to more than one things, in order to distribute the load on the resources available.

Element name	**Element type**	**Optional**	**Description**
Name	xsd:string	No	Name of the ContextAttribute that represents the service guaranteed in the SLA.
Type	xsd:string	Yes	Indicates the type of the ContextAttribute value.
isDomain	xsd:boolean	Yes	Indicates if this structure refers to a ContexAttribute or a AttributeDomain.
metadata	ContextMetadata [0..unbounded]	No	Metadata with entityId of the context entity that offer the attribute(service) and name of that attribute.

For the sake of simplicity, we present an example of allocation schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<registerContextRequest>
  <contextRegistrationList>
    <contextRegistration>
      <entityIdList>
        <entityId type="allocation" isPattern="false">
          <id>QoS_jshifhjsi898yhdjhd</id> (transactionID)
        </entityId>
      </entityIdList>
      <contextRegistrationAttributeList>
        <contextRegistrationAttribute>
          <name>temperature</name>
          <type>temperature</type>
          <isDomain>false</isDomain>
          <metadata>
            <contextMetadata>
              <name>equivalentEnt_1</name>
              <type>string</type>
              <value>tempSens_1,temp_1</value>
            </contextMetadata>
          </metadata>
        </contextRegistrationAttribute>
      </contextRegistrationAttributeList>
      <providingApplication></providingApplication>
    </contextRegistration>
  </contextRegistrationList>
  <duration></duration>
  <registrationId></registrationId>
</registerContextRequest>
```

The transactionId is used in the *Dispatching* phase when the service, granted by the SLA, is requested by the application. It is used to find out the corresponding allocation schema containing the reserved things. Each service correspond to a attribute of the context entity created in the *Allocation* phase. Each attribute has a list of metadata elements. Each metadata element points to a real Context Entity that describe the thing that offers a particular service. The pointer is represented by the entityId of the Context Entity stored in the IoT Discovery.

5.2.3 Dispatching phase

In the *Dispatching* phase the client application forwards the request for that services for which it was established a SLA in the previous phase. It carries out the request using the transactionId returned by the *QoSBroker*. In the following, we present the descriptions of the Dispatching for the NGSI operations take in consideration: *queryContext*, *updateContext* and *subscribeContext*.

Query Context In the *Dispatching* phase (figure 5.3), the user sends a queryContextRequest using the transactionId as entityId. It can be sent also a query with a standard entityId related to a normal Context Entity stored in the IoT Discovery (this case represents a best-effort service). The entityId uniquely identifies the allocation schema associated to the SLA. To each attribute is associated (through the ContextMetadata element) the list of pointers to the equivalent entities. At last, for each attribute, a *discoveryContextAvailability* operation is executed, using as entityId the one in the metadata element of the allocation list. If the discovery fails for that entity, the next entityId in the list is taken. The queryContextRequest in the next phase is the normal one as designed in FIWARE, using the entityIds of concrete entities. It is forwarded a queryContext for each attribute in the allocation mapping.

Update Context In the *Dispatching* phase (figure 5.4), the user sends a updateContextRequest using the transactionId as entityId. It can be sent also an update

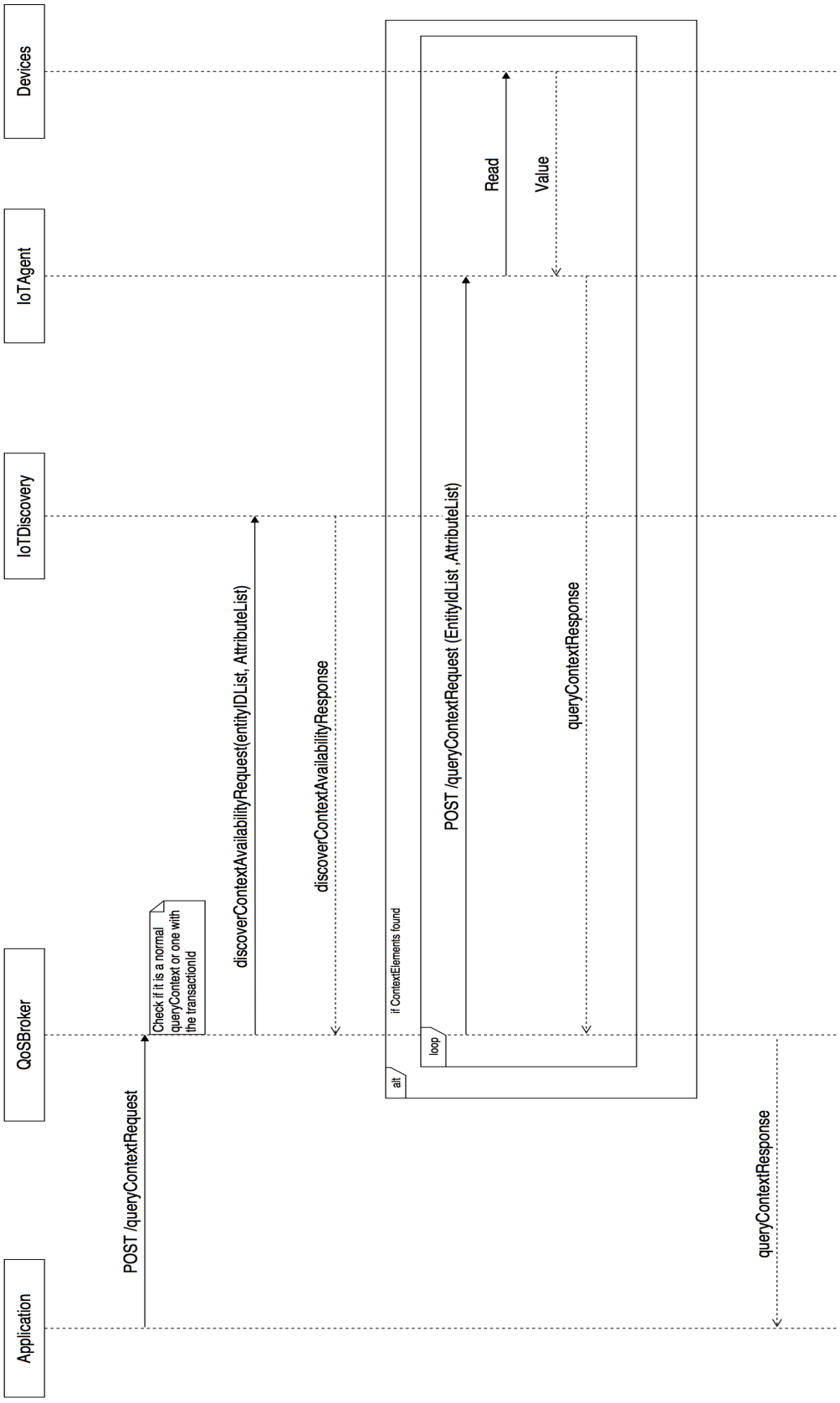


Figure 5.3: Query Context sequence diagram

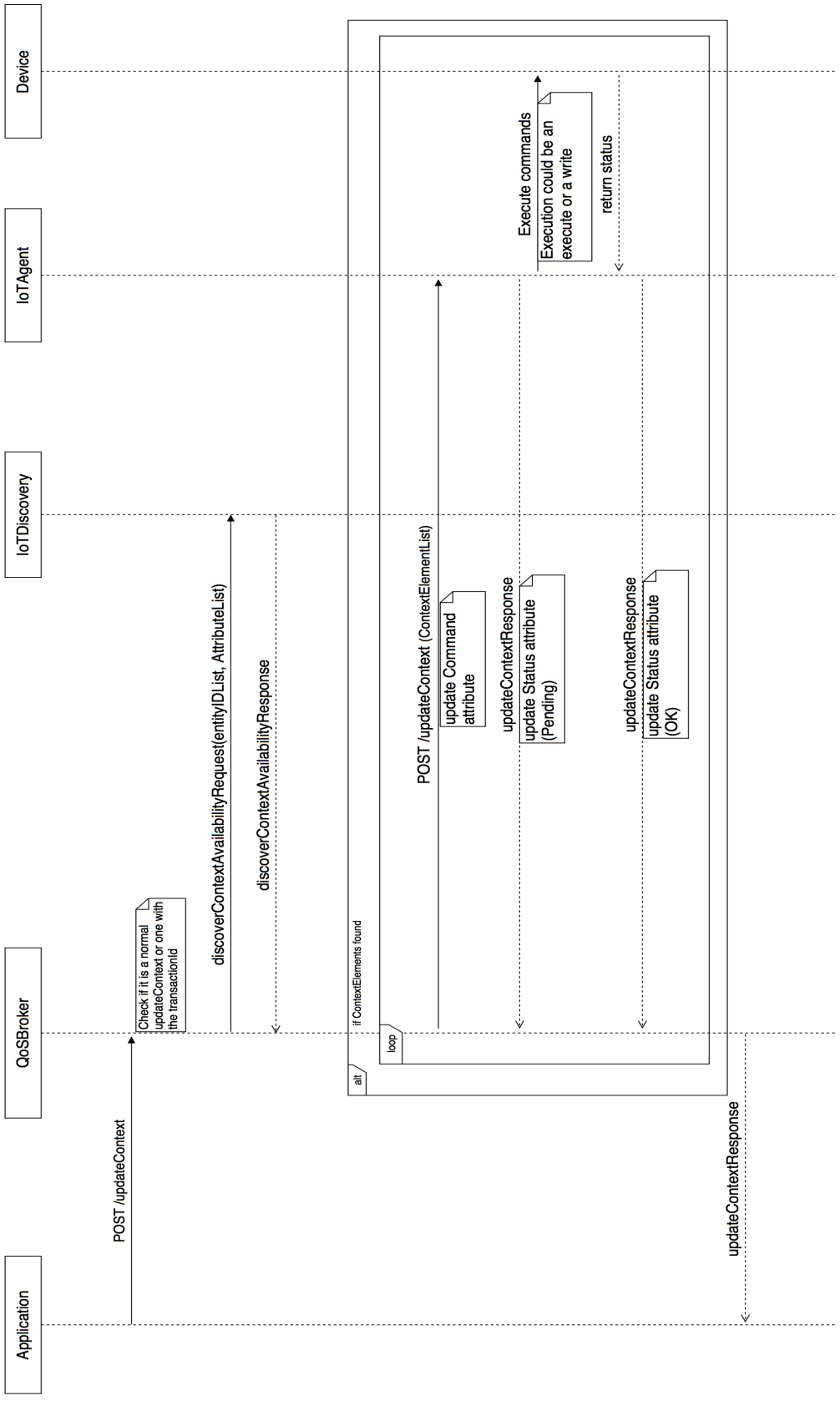


Figure 5.4: Update Context sequence diagram

standard entityId related to a normal Context Entity stored in the IoT Discovery (this case represents a best-effort service). The entityId uniquely identifies the allocation schema associated to the SLA. To each attribute is associated (through the ContextMetadata element) the list of pointers to the equivalent entities. At last, for each attribute, a *discoveryContextAvailability* operation is executed, using as entityId the one in the metadata element of the allocation list. If the discovery fails for that entity, the next entityId in the list is taken. The updateContextRequest in the next phase is the normal one as designed in FIWARE, using the entityIds of concrete entities. It is forwarded a updateContext for each attribute in the allocation mapping.

Subscribe Context In the *Dispatching* phase (figure 5.5), the user sends a subscribeContextRequest using the transactionId as entityId. It can be sent also a subscribe standard entityId related to a normal Context Entity stored in the IoT Discovery (this case represents a best-effort service). The entityId uniquely identifies the allocation schema associated to the SLA. To each attribute is associated (through the ContextMetadata element) the list of pointers to the equivalent entities. At last, for each attribute, a *discoveryContextAvailability* operation is executed, using as entityId the one in the metadata element of the allocation list. If the discovery/subscribeContextAvailability fails for that entity, the next entityId in the list is taken. The next subscribeContextRequest is like the normal one (as designed in FIWARE), using the entityIds of the concrete entities.

The subscribeContext operation provides also the notification phase. As we can see in the figure 5.6, there are two types of notifications. The first one is the normal one in which a new device value is notified to the client application. The second one is used to notify the availability of a new device to the *QoSBroker*.

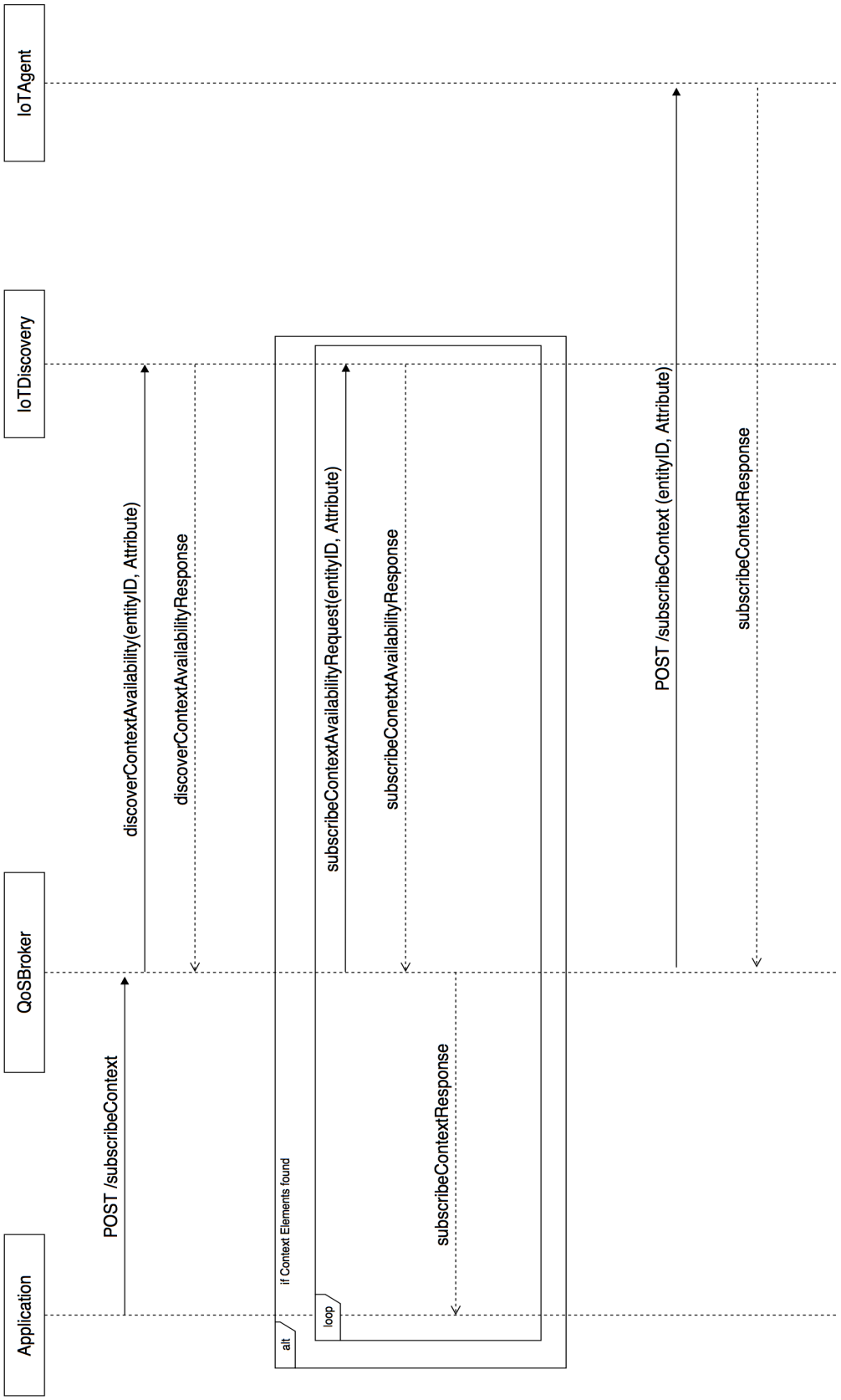


Figure 5.5: Subscribe Context sequence diagram

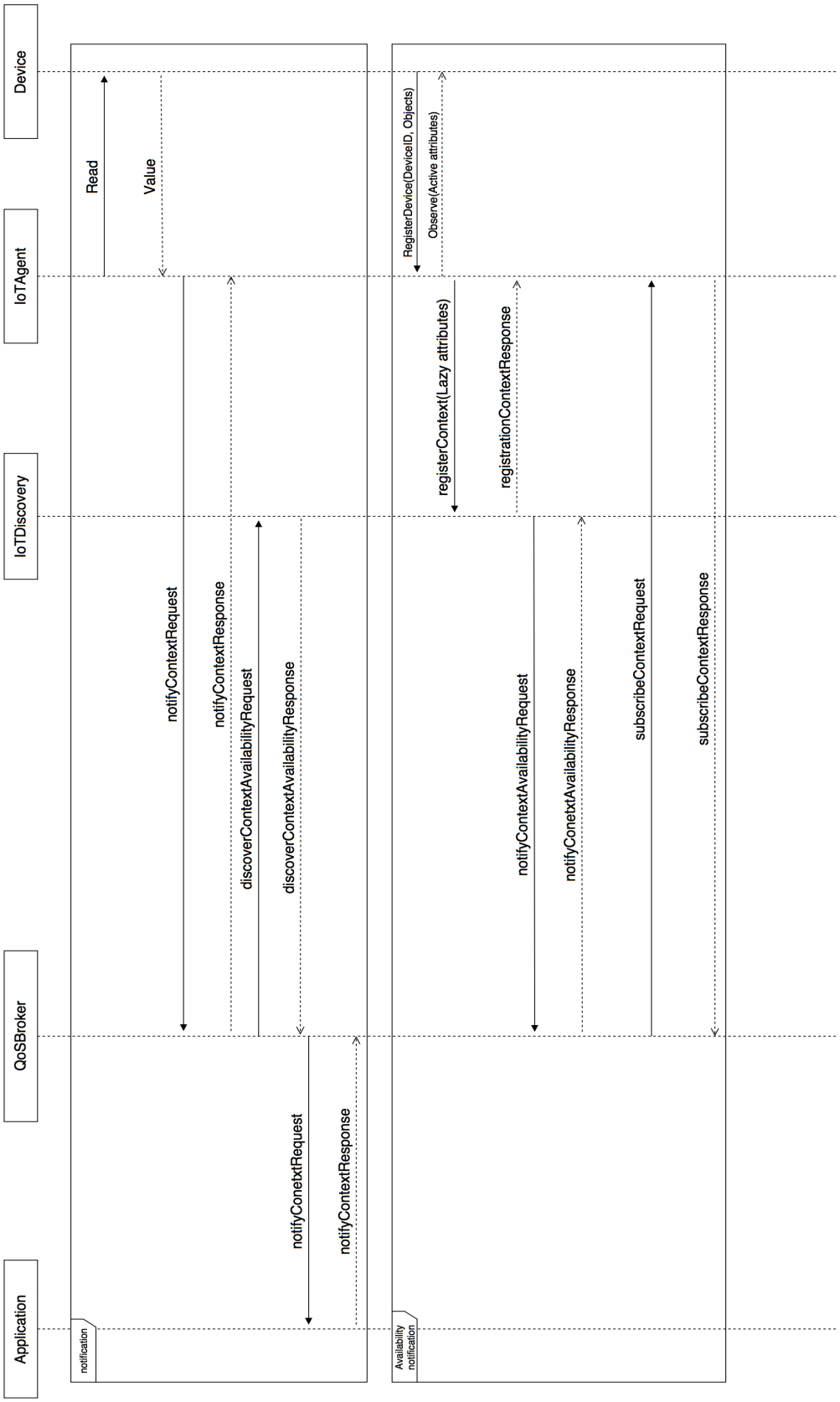


Figure 5.6: Notification and Availability notification sequence diagrams

5.2.4 IoT Agent LWM2M/CoAP

As we have said before we have chosen the IoT Agent LWM2M/CoAP version that works as protocol adapter for LWM2M devices. The LWM2M protocol is targeted in particular at constrained devices, e.g. devices with low-power micro-controllers. It provides a light and compact secure communication interface along with an efficient data model, which together enables device management and service enablement for M2M devices. CoAP is used as underlying transfer protocol. LWM2M defines a simple resource model where each piece of information made available by the LWM2M Client (a device that support LWM2M protocol) is a Resource. The Resources are further logically organized into Object. The LWM2M Client can have any number of Resources, each of which belongs to an object [17].

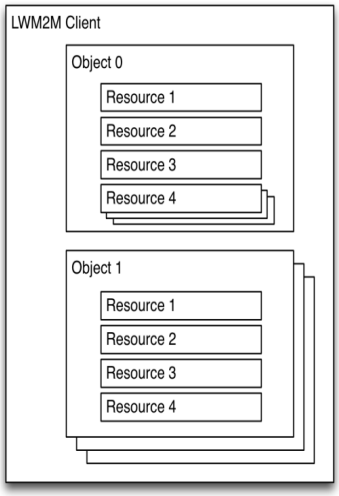


Figure 5.7: LWM2M Object model

In the starting phase, we have addressed the problem to test the communication between the IoT Agent, the IoT Discovery and the IoT Broker. The IoT Agent has been designed to communicate only with the Data Context Broker GE belonging to the Data FIWARE technical chapter. To this aim, we had to analyze the implementation of the IoT Agent to understand in which way set the communication with the

IoT Discovery and the IoT Broker. So we have modified the configuration file and the implementation of the IoT Agent in order to spread the exchange of information towards two GEs instead of only one. Now the IoT Agent carries out NGSI-9 operations with the IoT Discovery to register new devices and it carries out NGSI-10 operations with the IoT Broker. In the next phase, once the communication has been tested, we have put into communication the IoT Agent with our broker implementation, the *QoSBroker* (that wrap the FIWARE IoT Broker). In the next section we present a brief description of the operations to manage LWM2M devices, using the QoSBroker. We also describe the changes we have applied in the IoT Agent to enrich available informations about a device.

Device Provisioning The IoT Agent offers a provisioning API where devices can be pre-registered, so all the information about service and subservice mapping, attribute configuration and mapping NGSI-LWM2M attributes can be specified in a per device way. For each attribute are specified metadata that gives additional information about a thing resource. It can be also specified general metadata relative to an entire device. This operation allows to provide a new device in the IoT Agent's device registry. It takes a device in JSON format as payload. The information provided are also used in registerContext operation to register the availability of a new context entity and its attributes to the discovery module.

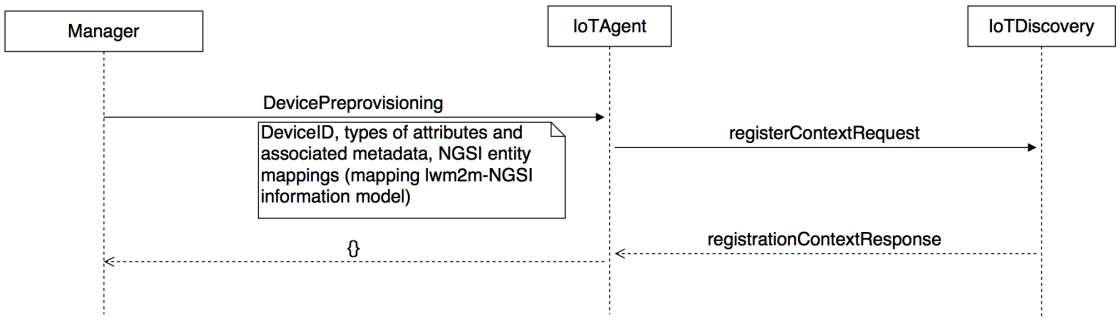


Figure 5.8: Device Provisioning

The *Device Provisioning* Operation is based on a *device model*, that specify the fields used to build the mapping between LWM2M information model and NGSI information model.

Attribute	**Definition**
name	Device Id that will be used to identify the device.
service	Name of the service the device belongs to.
service path	Name of the sub-service the device belongs to.
entity name	Name of the Context Entity representing the device.
entity type	Type of the Context Entity.
time-zone	Time zone of the sensor if it has any.
attributes	List of active attributes of the device.
lazy	List of active lazy attributes of the device.
commands	List of active commands of the device.
internal attributes	List of internal attributes with free format for specific IoT Agent configuration.

We have modified the previous model adding the possibility to specify metadata for the attribute field *lazy*. The aim is to register in the IoT Discovery, enriched device informations. In this way the *QoSBroker*, in the discovery phase, can retrieve parameters about the services exposed by a thing (i.e. latency, energy cost). In the following we can see an example of device provisioning request in which the field *metadata*, contains the parameters like cost or latency, relative to a thing service. In the following we can see an example of device provisioning request.

```
{
  "name": "sensor_1",
  "entity_type": "environment",
  "attributes": [
    {
      "name": "battery",
      "type": "float" ,
    }
  ],
  "lazy": [
    {
      "name": "temperature",
      "type": "float",
      "metadata": [
        {
          "name": "latency",
          "type": "float",
          "value": "3"
        },
        {
          "name": "energy_cost",
          "type": "float",
          "value": "0.12"
        }
      ]
    }
  ],
  "commands": [
    {
      "name": "power",
      "type": "string"
    }
  ],

  "internal_attributes": {
```

```
    "lwm2mResourceMapping": {
      "battery" : {
        "objectType": 7392,
        "objectInstance": 0,
        "objectResource": 1
      },
      "temperature" : {
        "objectType": 7392,
        "objectInstance": 0,
        "objectResource": 2
      },
      "power" : {
        "objectType": 7392,
        "objectInstance": 0,
        "objectResource": 3
      }
    }
  }
}
```

Device Registration Operation to connect a device to the IoT Agent (figure 5.9). In this phase, it is done the mapping between the LWM2M resources and NGSI attributes provided in the device provisioning operation. The registerContext operation notify to the IoT Discovery the availability of a new context entity.

Device Active Observation For that attributes registered as active in the device provisioning, the IoT Agent receives asynchronous notify messages by the LWM2M devices (figure 5.10). The IoT Agent sends an updateContext to the *QoSBroker* to update the attribute value in the relative context entity element, that store the informations of the device. A ContextElement is created inside the *QoSBroker*. It represents a structure to store the updated values relative to a particular device.

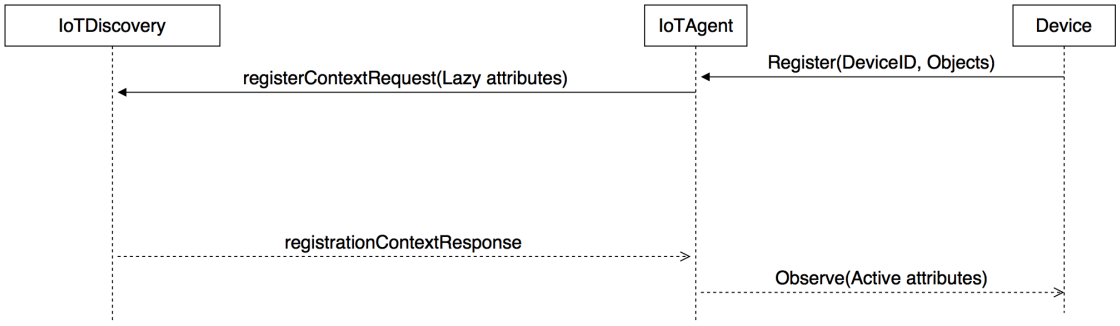


Figure 5.9: Device Registration

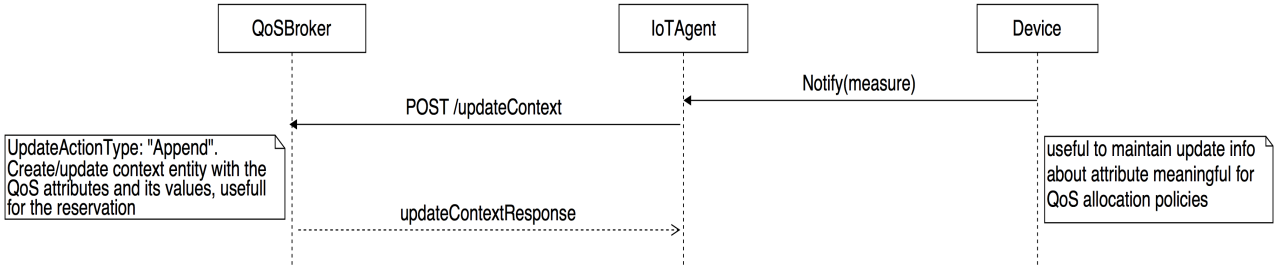


Figure 5.10: Device Active Observation

Device Unregistration Operation to disconnect a device from the IoT Agent. The latter send a registerContext to the IoT Discover with a duration of one second.

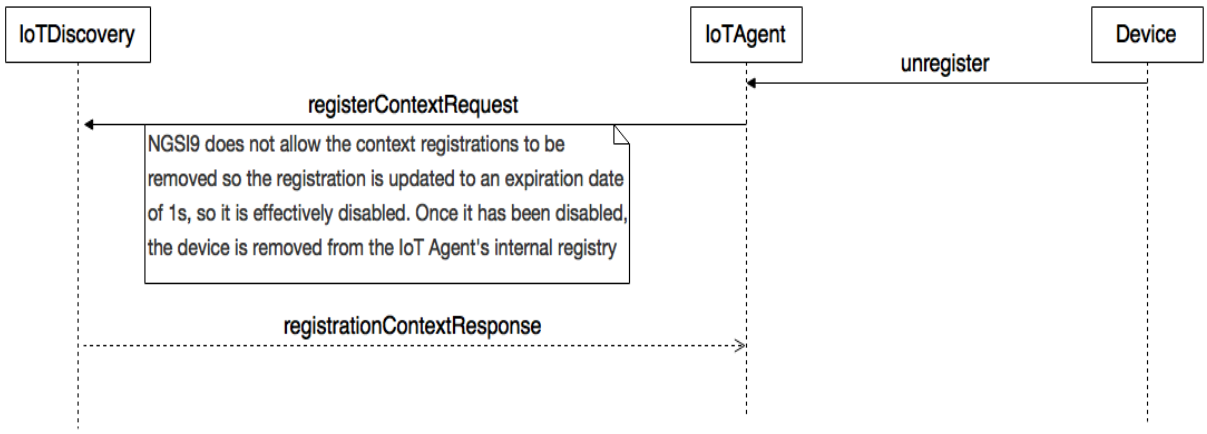


Figure 5.11: Device Active Observation

5.2.5 Development specification

In this section, we get in deep in the implementation details of the *QoSBroker*. This module is built as a wrapper for the IoT Broker, to extend its functionalities in order to guarantee a QoS support in the FIWARE IoT platform. The main components of the *QoSBroker* are:

- **RestController**, module that intercepts the restful calls to execute a series of operations coming from the client side
- **QoSBrokerCore**, module to manage NGSI-10 operations and service agreement request. It represents the core of the *QoSBroker* module, because it manages the allocation and dispatching phases;
- **QoSManager**, module that computes the incoming service agreement request and computes the set of inputs of the allocation algorithm;
- **QoSMonitor**, module that manages the asynchronous updates sent by the IoT Agent for that device attributes registered as active in the Device Provisioning operation. It stores or update the Context Elements that keep trace of the battery levels and/or coordinates of the devices. These values togheter with the informations in the ContextRegistration structures are used to build the thing data;
- **QoSCalculator**, module that executes the service allocation heuristic;
- **QoSCouchDB**, module to manages the repositories relative to the IoT context in which the *QoSBroker* works;
- **IoT Broker**, instance of the FIWARE IoT Broker.

The figure 5.12 shows all the interactions between the modules described in the previous list. It is evident how the *QoSBroker* represents an extension of the FIWARE IoT Broker. The presence of this new module is transparent for the other FIWARE modules, that are the IoT Discovery and the IoT Agent.

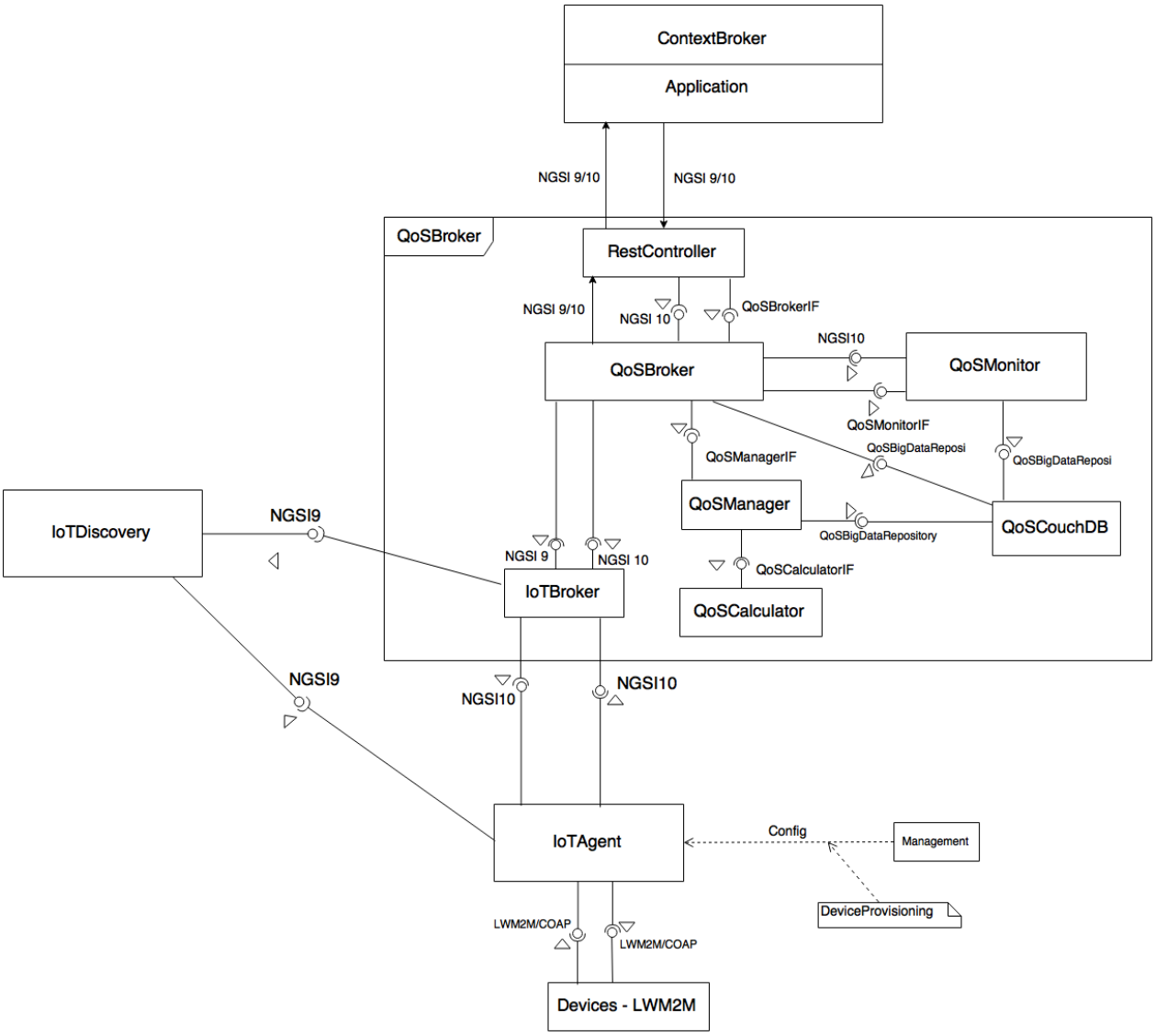


Figure 5.12: QoSBroker details

In the following we will analyze the interactions between all modules, for the operations that allow to realize a QoS solution in FIWARE.

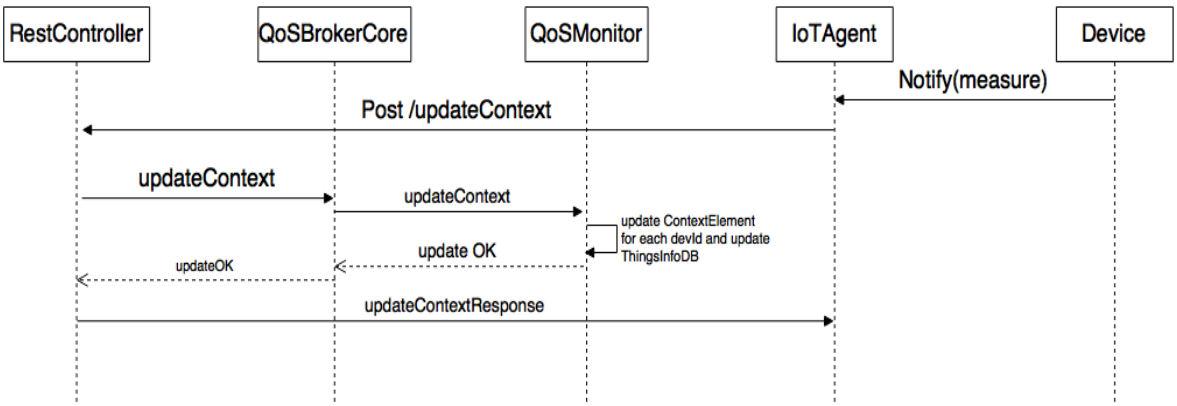


Figure 5.13: Device Active Observation

Device Active Observation After the notify coming from the device, the IoT Agent carries out an updateContext that is intercepted by the RestController component of the *QoSBroker*. The RestController replies the updateContext message to the QoS-BrokerCore, that sends the update values to the QoSMonitor. It creates or updates the ContextElement structures in the repository. Finally, the RestController replies with an updateContext Response to the IoT Agent. All the operations are depicted in the figure 5.13.

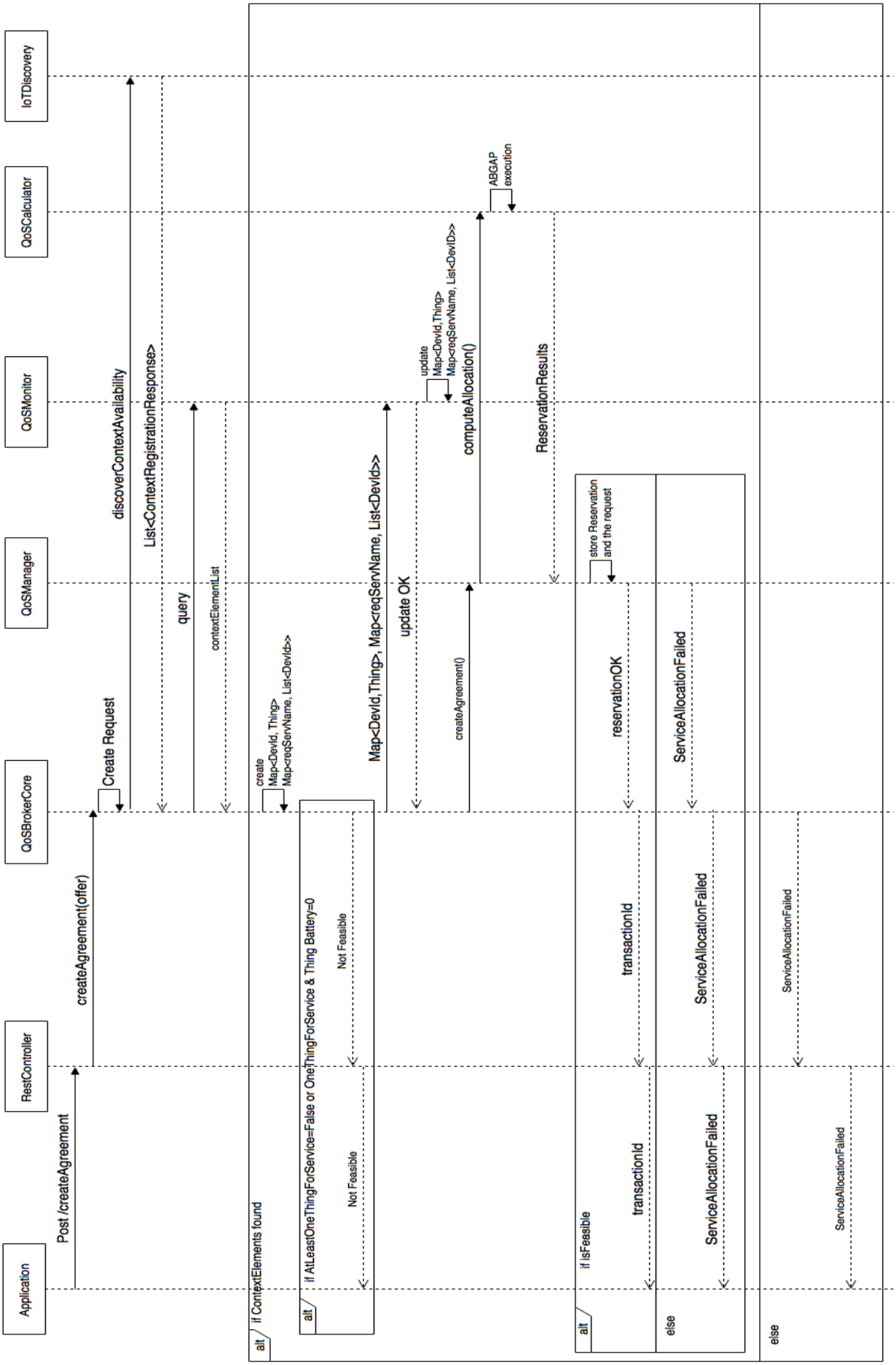


Figure 5.14: Allocation phase

Allocation phase The operations of the Allocation phase are depicted in the figure 5.14. As we can see, the client application invokes a *createAgreement* operation through which specifies a service agreement offer. The offer contains the type of operation, the QoS requirements and a geographical scope. The RestController intercepts the operation call. After the validation of the service agreement request, the RestController re-sends the createAgreemnt message to the QoSBrokerCore. This component carries out a discovery procedure to take all the informations (ContextRegistration and ContextElement structures), that are used to build the list of things object. The thing object represents the abstraction of a physical device. The negotiation procedure proceeds if the required services are offered by at least one thing for each service. The list of things object are sent to the QoSMonitor to update the repositories that keep track of the available things and of the equivalent things for each service. In the end, the QoSBrokerCore calls the createAgreement function of the QoSManager. This component computes the input parameters of the QoSCalculator for the heuristic algorithm, given the list of things objects, the list of equivalent things for each service and the list of service requests (containing the QoS requests parameters). The last operation is the computeAllocation that implies the execution of the service selection algorithm implemented by the QoSCalculator module. If a feasible allocation is found, a new allocation schema is stored in the repository and the transactionId is sent to the client application through the ServiceAgreementResponse structure. Otherwise, in case a feasible allocation is not found, a service allocation failed message is sent to the client.

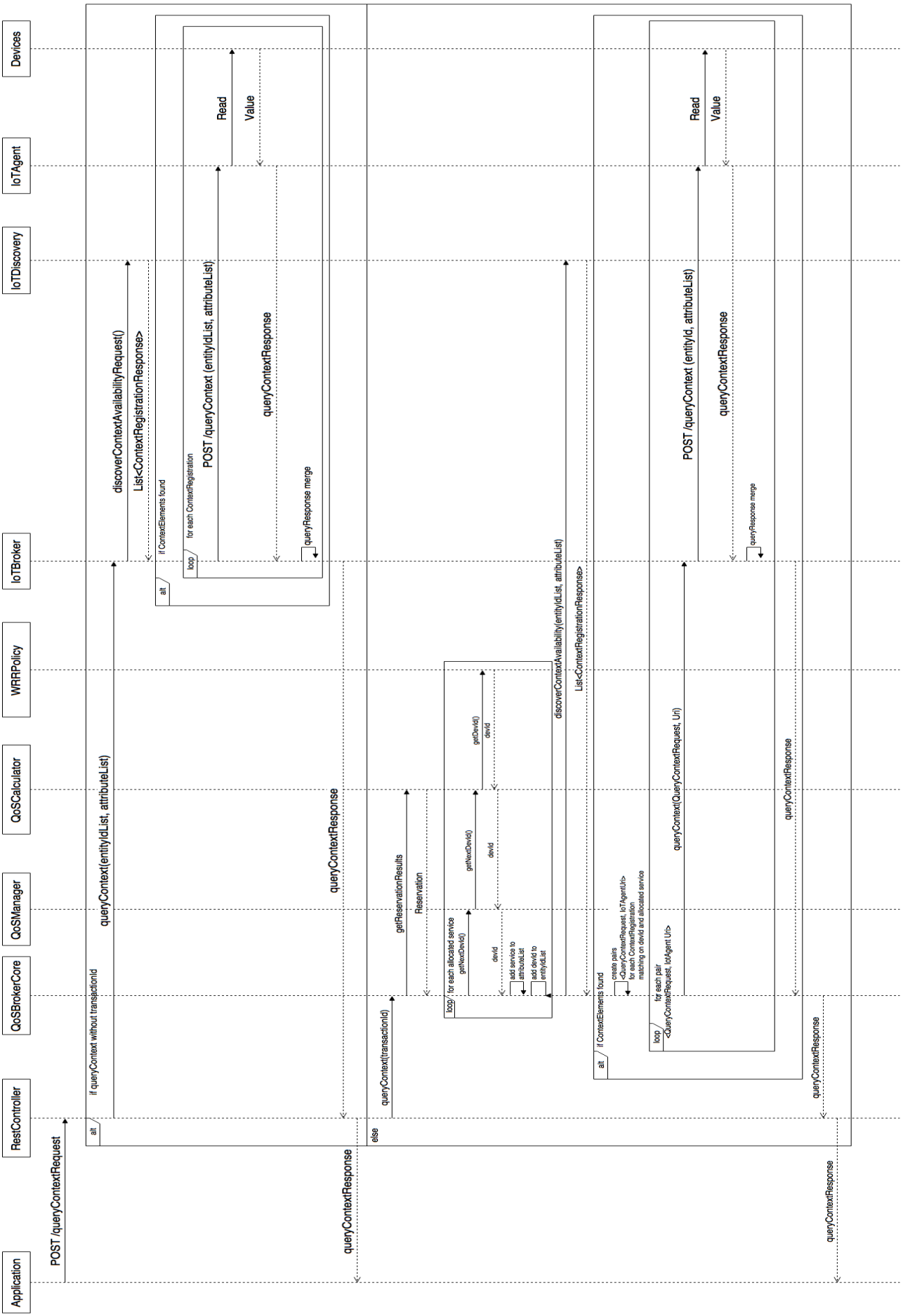


Figure 5.15: Query Context

Query Context In the figure 5.15 we can see the operations carry out in the queryContext Dispatching phase. There are two types of query context operation. The first one is carried out using a normal entityId, so it is sent to the IoT Broker and executed as a normal NGSI query context as originally implemented in the FIWARE platform. The second one is carried out using a transactionId as parameter. It is forwarded to the QoSBrokerCore component by the RestController. The QoSBrokerCore component queries the QoSCalculator module to retrieve the allocation schema associated to the transactionId. In the allocation schema for each granted service is associated a list of thing ids that represent the globally unique identifiers of the ContextRegistration structures associated to the devices. The Weighted Round Robin Policy component implements the policy mechanism to choose the next id of the list taken from the allocation schema. Once for each service in the SLA, an id is retrieved, the QoSBrokerCore carries out a discovery phase querying the IoT Discovery. This operation permits to retrieves the ContextRegistration structures associated to each thing id. Thanks to the ContextRegistration informations, the QoSBrokerCore can forward a series of query context towards the IoT Broker that will query the IoT Agent connected to the devices that expose the services read in the SLA. Finally, a queryContextResponse message, containing the values read through the queryContext, is sent to the client application.

5.2.6 Implementation details

In the following section is presented the list of data structures used to implement the FIWARE QoS support. The transactionId uniquely identifies a service agreement request and an allocation schema (if a feasible allocation is found). A transactionId is generated as an hash string every time a new service request is forwarded by the upper layer. The list of attributes in the *ServiceAgreementRequest* represent the list of required services. Each required service is expressed as a string that uniquely identify a service exposed by the IoT architecture. The informations of the *ServiceAgreementRequest* are parsed and a *Request* object is created. The fields of this object are *OperationType* (queryContext, updateContext, ..), QoS scope value (object containing the fields "maxResponseTime" and "maxRequestRate" that is the requested period p_j of a service), Restriction (NGSI object), EntityIdList (list of entityIds used to discover the ContextRegistration elements associated to the available things), list of string representing the required services (list created from the list of attributes in the *ServiceAgreementRequest*). Each *Request* object is uniquely identified through a transactionId. For each *ContextAttribute* in the *ContextRegistrationResponse* that matches a required service name, a *ServiceFeatures* object is created. This object has the fields latency (t_{ij}) and energy cost (c_{ij}). For each *ContextElement* (taken from the QoSMonitor repository) associated with a *ContextRegistrationResponse* a *Thing* object is created. The fields of this object are battery level, coordinates (values taken from the *ContextElement* stored in *QoSMonitor* repository) and a map to associate each service to an object *ServiceFeatures*. Each *Thing* is uniquely identified by an id (that is the id in the entityId structure of the ContextRegistration that represent the device). Every *ContextRegistrationResponse* element has a list of *ContextAttribute* (list of services exposed by a thing). In our model this feature is represented in the *Thing* object as a map of *ServicesFeatures* objects having the service name as key. The service name is considered globally unique. The list of *Thing* objects is stored in a repository in which each thing is uniquely identified through its id. It is also stored the list of equivalent things id

associated to each required service. After this phase the *QoSBroker* invokes the *createAgreement* function of the *QoSManager* passing as paramenters the transactionId and *Request* object relative to the last service agreement request. The *QoSManager*, in the *createAgreement* function, reads the list of previous Request objects (they were created in the previous service requests in which a feasible allocation was found), the map of *Thing* objects and the map of equivalent things for each service. Then, the *QoSManager* calls the *QoSCalculator* function, that implement the heuristic algorithm. The *QoSCalculator* returns a reservation object that contains the result of the service selection algorithm. If the allocation is feasible the last *Request* object, computed in the last service request, and the allocation schema are stored in a repository by the *QoSManager*.

5.2.7 RTTA with multi-service allocation policy

In the [3], the authors present a service allocation algorithm for which a service is allocated only to one specific thing. To enlarge the set of solutions of the problem, we have modified the previous algorithm introducing the possibility to distribute the allocation of a service to multiple things instead of only one. The service requests can be spread over multiple things following a round robin policy. Taking in consideration the constrains of the original algorithm [3] about the battery level and the utilization, the distribution of the service requests permits to decrease the service execution energy cost but implies a delay of the deadline associated to each task relative to a request. The result is that the set of solutions of the allocation problem is greater, because the split factor represents a divider factor for the energy cost and the load of the execution of a service on a thing. In the following we present the new *Feas* procedure based on the *Feas* algorithm described in [3]. The inputs are the same of the original *Feas* procedure except for Φ. It represents the list of factors $\frac{h}{p_j}$ associated to each service request. These coefficients are factorized to compute the list of possible split factors of a service request on multiple things.

Algorithm 1 RTTA with multi-selection allocation policy

procedure *Feas()*

Algorithm RTTA with multi-thing allocation policy

Input: n, k, **P**, **F**, **U**, Φ, θ

Output: **z**, **y**, *isFeasible*

N $\leftarrow \{1...n\}$; K $\leftarrow \{1...k\}$; $\nu \leftarrow k(2^{1/k} - 1)$

$isFeas \leftarrow True$

for $i \leftarrow 1$ ***to*** n ***do*** $c_i \leftarrow 0$

for $i \leftarrow 1$ ***to*** n ***do*** $z_i \leftarrow 1$

while *isFeas = True* ***and*** $K \neq O$ **do**

  $d^* \leftarrow -\infty$

  for each $j \in K$ **do**

    $S_j \leftarrow Factorization(\varphi_j)$

    $s_p \leftarrow 1$

    while $S_j \neq O$ **do**

      $F_j^{s_p} \leftarrow \{i \in N : c_i + (u_{ij}/s_p) < \nu, z_i - (f_{ij}/s_p) > \theta\}$

      if $\sum c_{ij}^{s_p} < s_p$ **then**

        $S_j \leftarrow S_j \setminus \{s_p\}$

        if $S_j = O$ **then**

          isFeas $\leftarrow False$

        else

          $s_p \leftarrow S_j(0)$

          continue

        i'$\leftarrow \arg\max \left\{ p_{ij} :< i, c_{ij}^{s_p} >\in F_j^{s_p} \right\}$

        if $\sum c_{ij}^{s_p} - c_{ij}^{s_p} < s_p$ **then**

          $d^* \leftarrow +\infty$

          $W^{s_p} \leftarrow ComputeAllocation(F_j^{s_p}, s_p)$

          for each $< i, w_{ij}^{s_p} >\in W^{s_p}$ **do**

            $y_j \leftarrow < i, w_{ij}^{s_p} >$

            $z_i \leftarrow z_i - (f_{ij}/s_p) * w_{ij}^{s_p}$

            $c_i \leftarrow c_i + (u_{ij}/s_p) * w_{ij}^{s_p}$

          $K \leftarrow K \setminus \{j\}$

          break

33: **else**
34: $d \leftarrow getDiffMaxAndMax_2(F_j^{s_p}, s_p)$
35: **if** $d > d^*$ **then**
36: $d \leftarrow d^*$
37: $j^* \leftarrow j$
38: $W^{s_p} \leftarrow ComputeAllocation(F_j^{s_p}, s_p)$
39: **for each** $< i, w_{ij}^{s_p} > \in W_{ij}$ **do**
40: $I^* \leftarrow < i, w_{ij}^{s_p} >$
41: **break**
42: **if** $d^* = +\infty$ **break**
43: **if** *isFeas = True* **then**
44: **for each** $< i, w_{ij}^{s_p} > \in I^*$ **do**
45: $y_{j^*} \leftarrow < i, w_{ij}^{s_p} >$
46: $z_i \leftarrow z_i - (f_{ij}/s_p) * w_{ij}^{s_p}$
47: $c_i \leftarrow c_i + (u_{ij}/s_p) * w_{ij}^{s_p}$
48: $K \leftarrow K \setminus \{j^*\}$
49: **else**
50: **return**

The new *Feas* procedure performs the splitting of a service to multiple things, only if the allocation of a service to only one thing fails. Then a list of split factors, computed factorizing the coefficient $\frac{h}{p_j}$, give the list of factors each one representing the number of things on which a service request can be spread. The algorithm starts an iteration over the list of split factors until a feasible allocation is found. In other terms, in case of distribution of a service request, the allocation of a service is spread on multiple things starting from the thing with maximum priority. In this case, things must respect more relaxed constraints, being the utilization u_{ij} and the normalized energy cost f_{ij} divided by a split factor (line 15 of the *Feas* procedure

pseudo-code). The maximum load of a thing is computed as $c_{ij}^{s_p}$ factor that represents the number of times a service can be assigned to a specific thing respecting the constraints about utilization and energy cost in an hyperperiod.

In the following there are two procedures used in the new *Feas* algorithm. The first one computes allocation for a given service, starting from the thing with the maximum priority. So it computes the number of times it can be assigned a service to the same thing, respecting the $c_{ij}^{s_p}$ factor (it guarantees that the utilization and the energy cost constrains are respected). If the limit is reached and the allocation is not terminated, the service assignment proceeds to the next maximum priority thing. The second one computes the weighted difference between the thing with maximum priority and next maximum priority thing.

Algorithm 2 ComputeAllocation

1: **procedure** *ComputeAllocation()*
2: ***Algorithm*** ComputeAllocation
3: ***Input:*** F_{js_p}, s_p
4: ***Output:*** W^{s_p}
5: $w_{ij}^{count} \leftarrow s_p$
6: **while** $w_{ij}^{count} > 0$ **do**
7: $< i, c_{ij}^{s_p} > \leftarrow \arg\max \left\{ P_{ij} :< i, c_{ij}^{s_p} > \in F_j^{s_p} \right\}$
8: **if** $c_{ij}^{s_p} < w_{ij}^{count}$ **then**
9: $w_{ij}^{s_p} = c_{ij}^{s_p}$
10: **else**
11: $w_{ij}^{s_p} = w_{ij}^{count}$
12: $W^{s_p} \leftarrow < i, w_{ij}^{s_p} >$
13: $w_{ij}^{count} = w_{ij}^{count} - w_{ij}^{s_p}$

Algorithm 3 $getDiffMaxAndMax_2$

procedure $getDiffMaxAndMax_2()$
  Algorithm $getDiffMaxAndMax_2$
  Input: F_{js_p}, **P**, s_p
  Output: d
  $i \leftarrow 0$
  while $i < 2$ **do**
    $W^{s_p} \leftarrow ComputeAllocation(F_j^{s_p}, s_p)$
    for each $< i, w_{ij}^{s_p} > \in W^{s_p}$ **do**
      $\sum p_{ij} w_{ij}^{s_p} \leftarrow \sum p_{ij} w_{ij}^{s_p} + p_{ij} w_{ij}^{s_p}$
      $\sum w_{ij}^{s_p} \leftarrow \sum w_{ij}^{s_p} + w_{ij}^{s_p}$
    if i = 0 **then**
      $Max \leftarrow \sum p_{ij} w_{ij}^{s_p} / \sum w_{ij}^{s_p}$
    else
      $Max_2 \leftarrow \sum p_{ij}^{(2)} w_{ij}^{s_p} / \sum w_{ij}^{s_p}$
    $i \leftarrow i + 1$
  $d \leftarrow Max - Max_2$

Chapter 6

Tests results

The test phase has been carried out in two phases. In the first one we have validated the QoS support (with multi service allocation) developed in the FIWARE IoT platform, through a simple use case scenario. In the second one we have focused our work on the new version of the RTTA algorithm. We have validate the RTTA algorithm with multi service allocation, executing a series of simulations using different scenarios. The results shows as the new version of the RTTA heuristic algorithm implies a larger set of feasible allocations respect to the original one.

6.1 Validation test

The validation test is carried out using a simple use case scenario. In the scenario are used four temperature sensors. The battery of each thing is at the maximum level (50.710 mJ). In the table below we can see the main data about the sensors used in the use case scenario.

deviceId	**battery level**	**temperature**
$sensor_1$	100%	latency: 0.03; energy cost: 0.15
$sensor_2$	100%	latency: 0.05; energy cost: 0.3
$sensor_3$	100%	latency: 0.04; energy cost: 0.2
$sensor_4$	100%	latency: 0.07; energy cost: 0.1

Three client applications simulate the forwarding of three service agreement requests to the *QoSBroker*, all for the temperature service, with three different maximum request rates: 17 seconds, 2 seconds and 9 seconds. All these three requests results in a feasible allocation schema (two schemas it has been obtaining through the multi service selection). In the second phase starts the *Dispatching* phase. Each client application (emulated through three different threads) forwards a service request respecting the period specified in request sent in the *Allocation* phase. In the graphic below, we can see how the load is distributed on different things (represented using different colours). We have also emulated a load of the system, introducing a random delay on the response time of the things. We can see also that in two interval, because of the system load, the first client doesn't receive a response from the $sensor_3$ and the $sensor_1$.

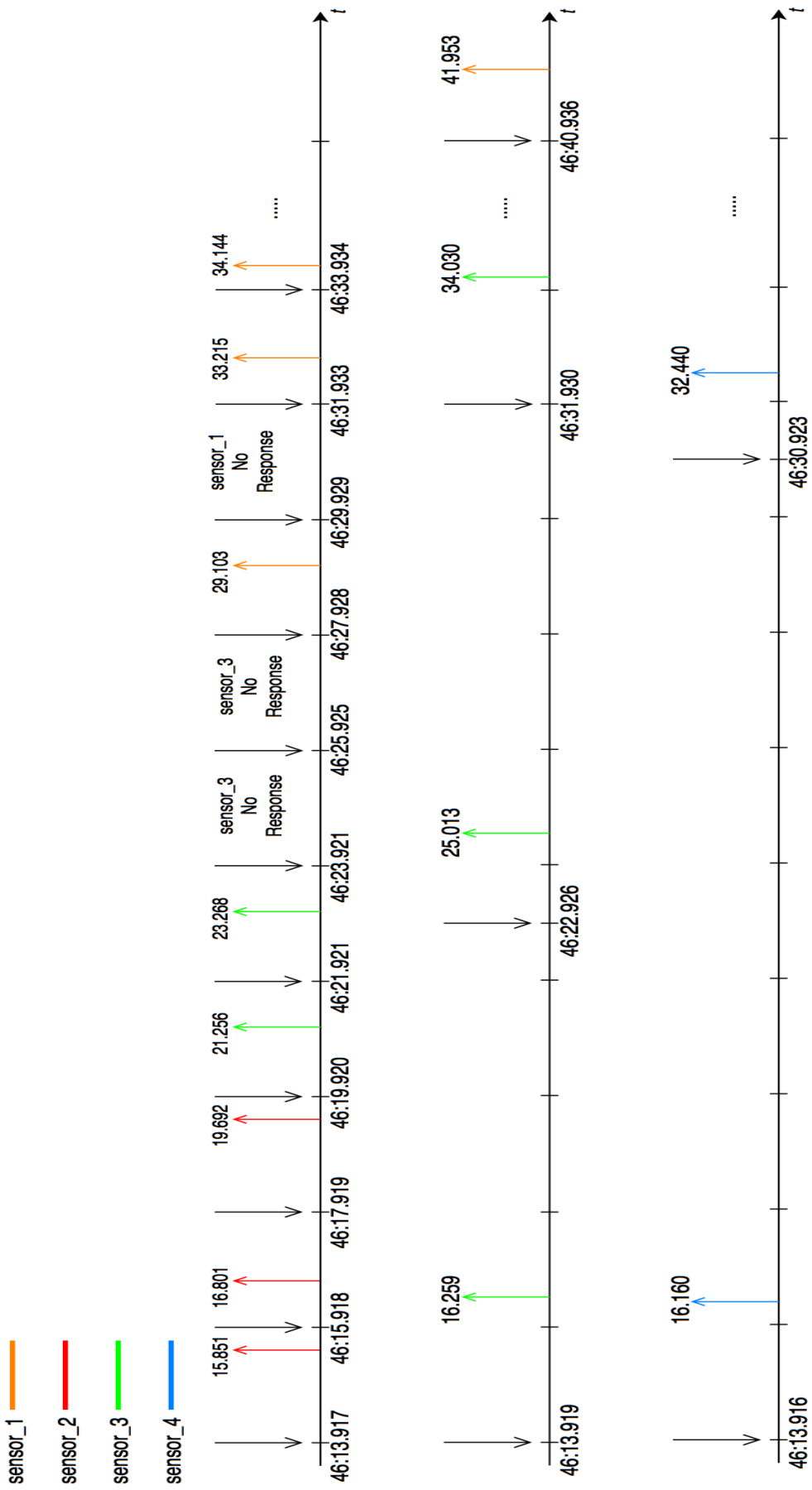


Figure 6.1: Test timeline

6.2 RTTA test

The second test is carried out on the new implementation of the *RTTA* heuristic (in which a service can be allocated to multiple things). In the test each result is compared with the one of the original RTTA version (described in [3]). The algorithm is evaluated in different scenarios, each one characterized by a number of service requests and available things. In each scenario the evaluation is carried out changing the average number of services exposed by each thing, expressed as a fraction of the overall number of service requests. For each scenario, first of all, the initial battery levels, computational costs and periods (from which are computed the list of coefficients $\frac{h}{p_j}$) are generated taking them from a uniform distribution with parameters as reported in the table below.

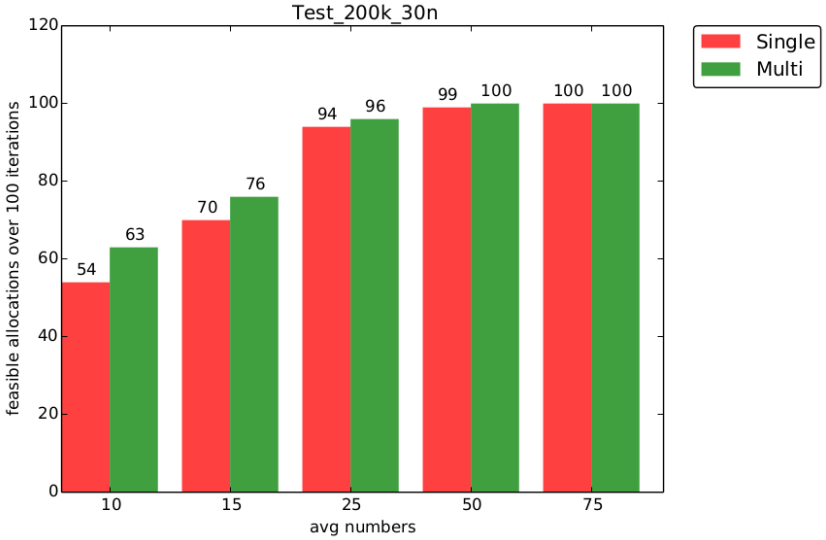
Parameter	**Range**
Period	10 – 100 s - step 10 s
Initial battery level	50 – 25 mJ - step 5 mJ
Execution cost	2*10^{-4} – 6*10^{-4} mW
Execution time	7 – 22.5 ms

The following average number of services exposed by each thing are considered (expressed as a fraction of the overall number of service requests): 10%, 15%, 25%, 50%, 75% respectively. For each average number of services, one hundred of m_{ij}'s, i.e. context information about which services can be invoked on which things, are also randomly generated so that the average number of services per thing characterizing the scenario is fulfilled. Each matrix **M** generated in each iteration is used to test both the single and multi service allocation algorithm version.

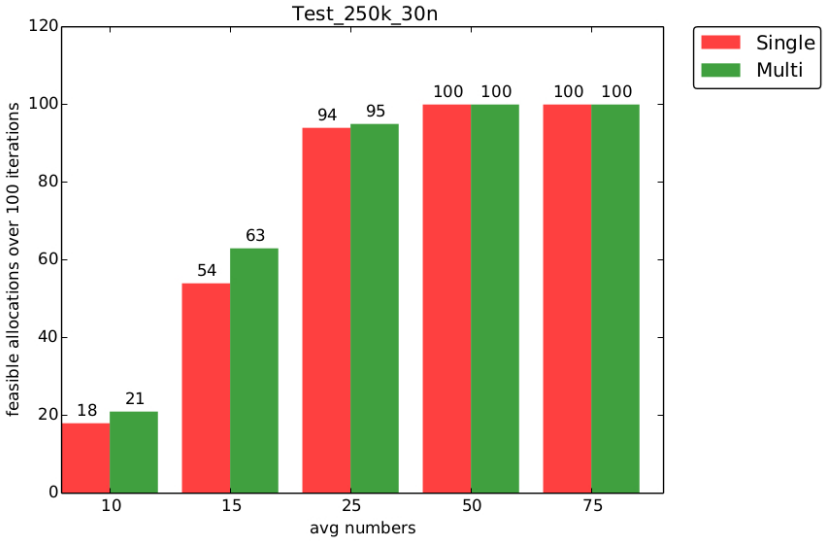
In-fact to evaluate the performance we consider, for each average number, the number of feasible allocations between the original and the modified version of the RTTA algorithm. The test scenarios are characterized by:

- 200 services and 30 things;
- 250 services and 30 things;
- 300 services and 35/40 things;
- 350 services and 45 things;
- 400 services and 35/40 things;
- 450 services and 45 things

In the previous scenarios, the used battery level distribution is composed by the following values 30.426, 25.355, 20.284, 15.213, 10.142, 5.071 mJ. From the results, depicted in the plots in the figure 6.2, we see as for each average number (10%, 15%, 25%, 50%, 75%), the number of feasible allocations computed with the new algorithm is greater than or equal to the number of feasible allocations obtained executing the original version of the *RTTA* algorithm. The difference in the number of feasible allocations is prominent especially for the average numbers 10% and 15%.

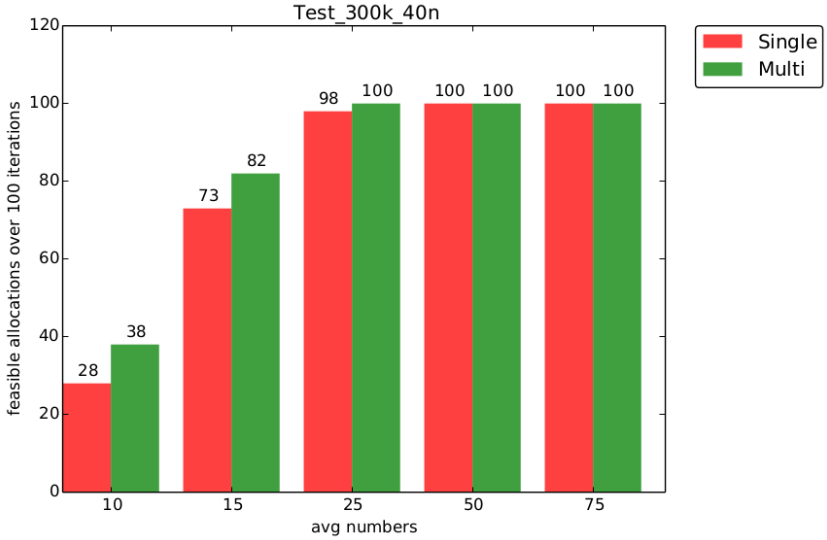


(a) scenario 200 services, 30 things

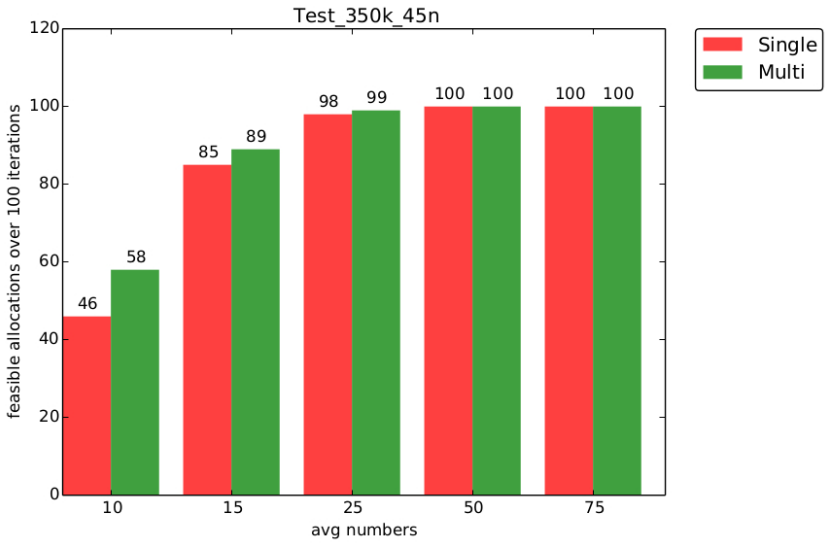


(b) scenario 250 services, 30 things

(c) scenario 300 services, 35 things

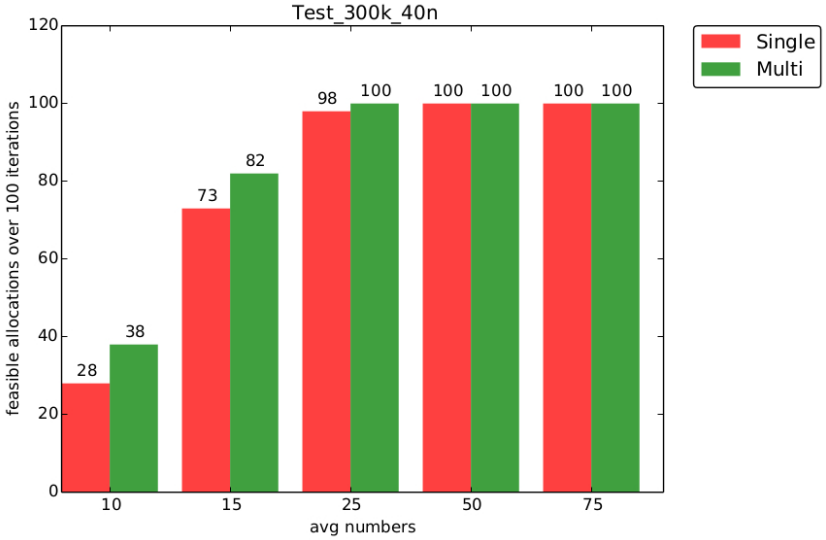


(d) scenario 300 services, 40 things

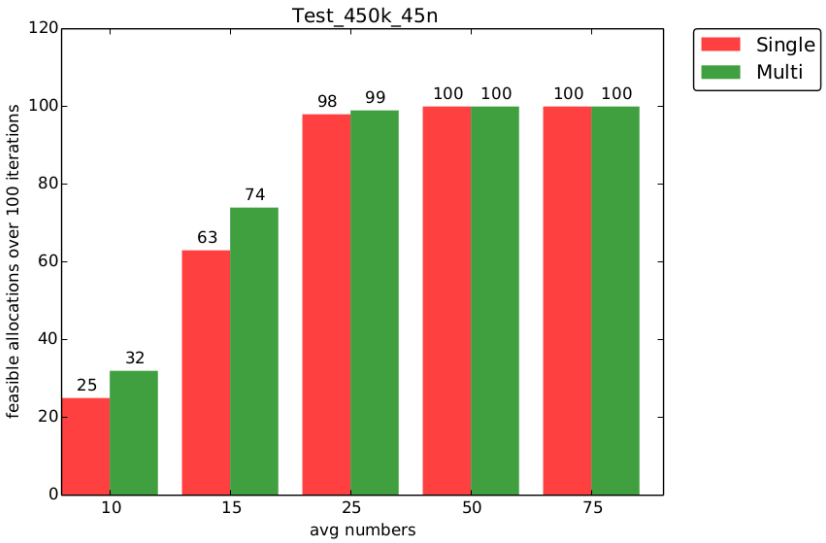


(e) scenario 350 services, 45 things

(f) scenario 400 services, 35 things



(g) scenario 400 services, 40 things

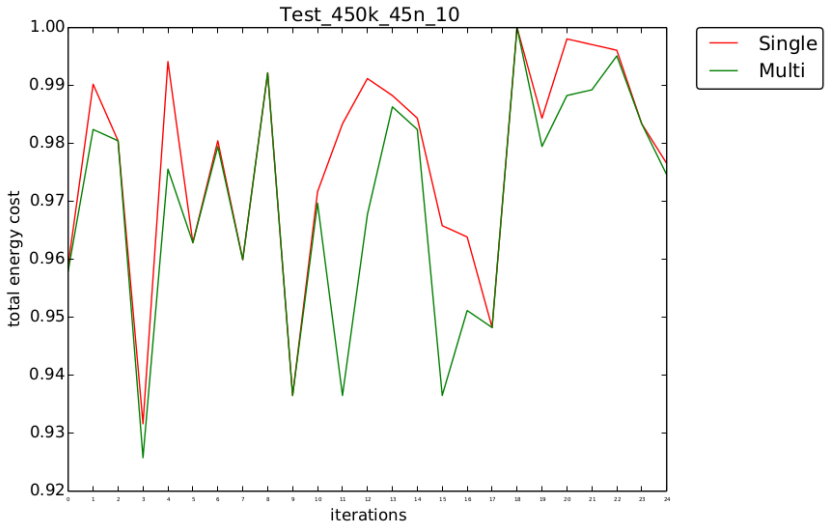


(h) scenario 450 services, 45 things

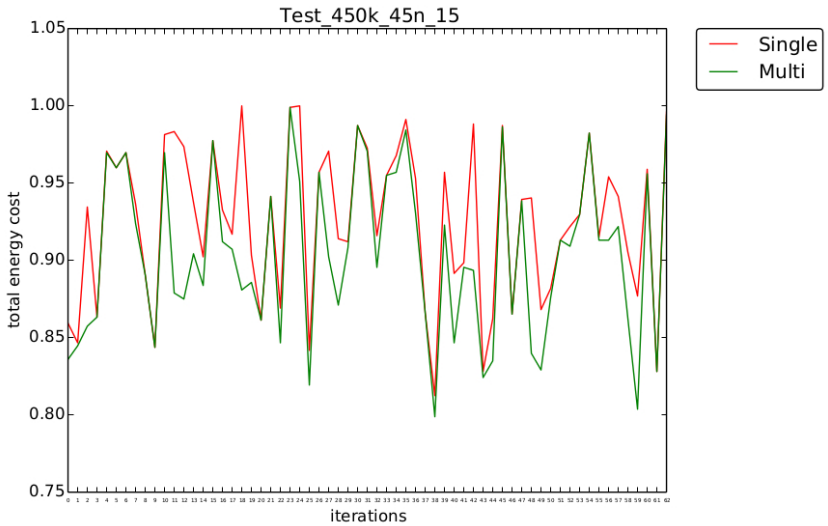
Figure 6.2: Feasible allocations plots over 100 iterations per Single/Multi service selection approach and per average services number on a thing

Moreover, in the figure 6.3 we show, for the last scenario, the plots relative to the energy cost behaviour (Single and Multi approach) for each average number, for those iterations in which both single and multi approach result in a feasible alloca-

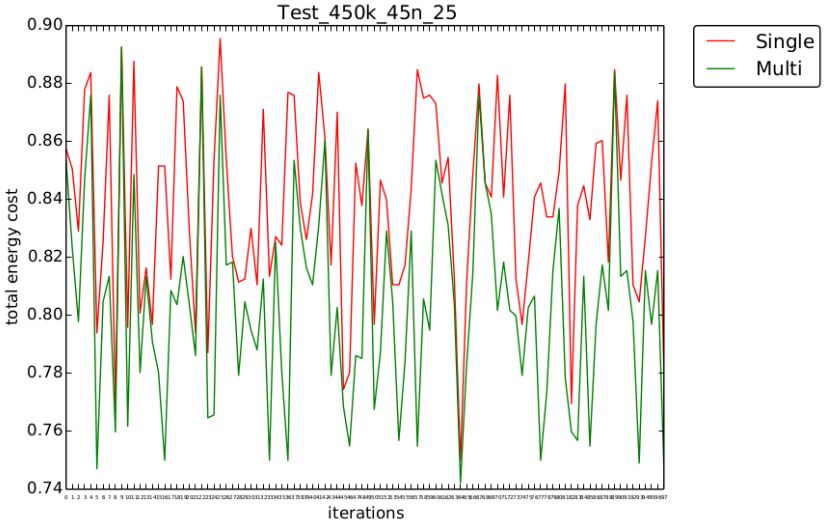
tion. We can see as the behaviour of the energy cost is more regular as the average number of services on each thing increases. For sake of brevity, we present the plots relative to one scenario being the behaviour for the other scenarios basically the same.



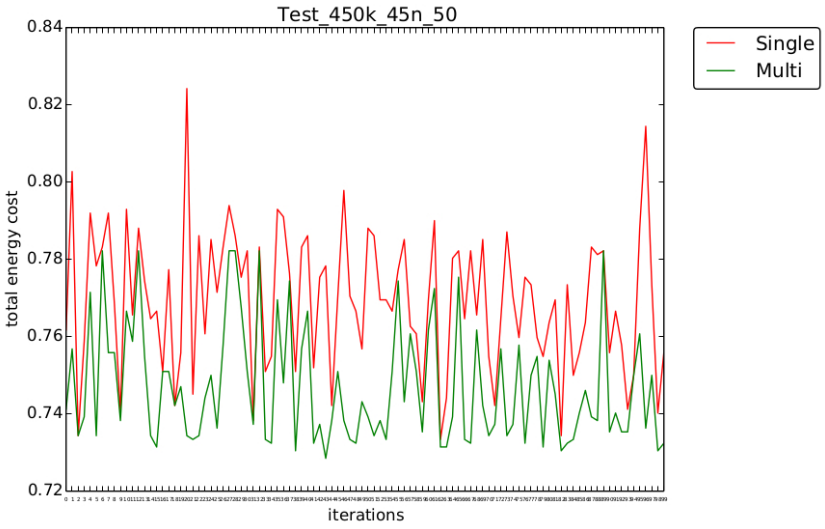
(a) scenario average number 10



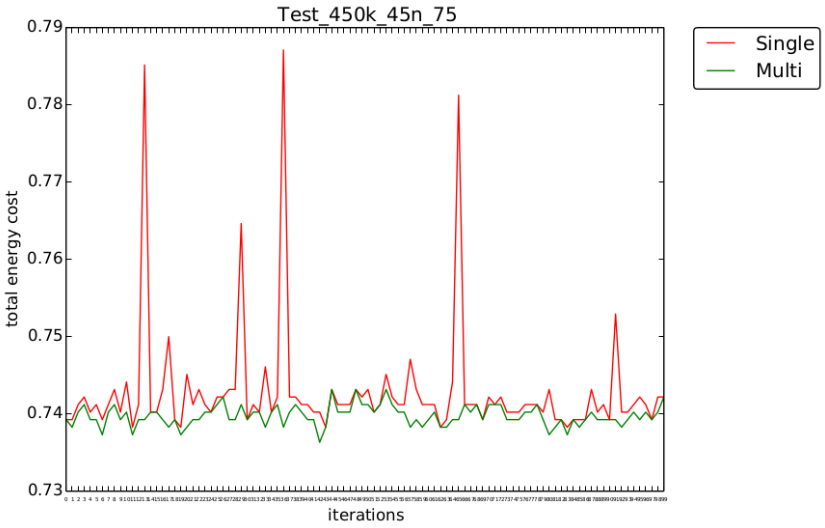
(b) scenario average number 15



(c) scenario average number 25



(d) scenario average number 50



(e) scenario average number 75

Figure 6.3: Energy cost behaviour for each average number

Finally, we can see a summary of the scenarios result tests in the plot in figure 6.4. For each scenario, with a number of services and a number of things given as inputs, and for each average number of services on a thing, we plot the average difference computed over the iterations where both the single and the multi service selection approach are feasible.

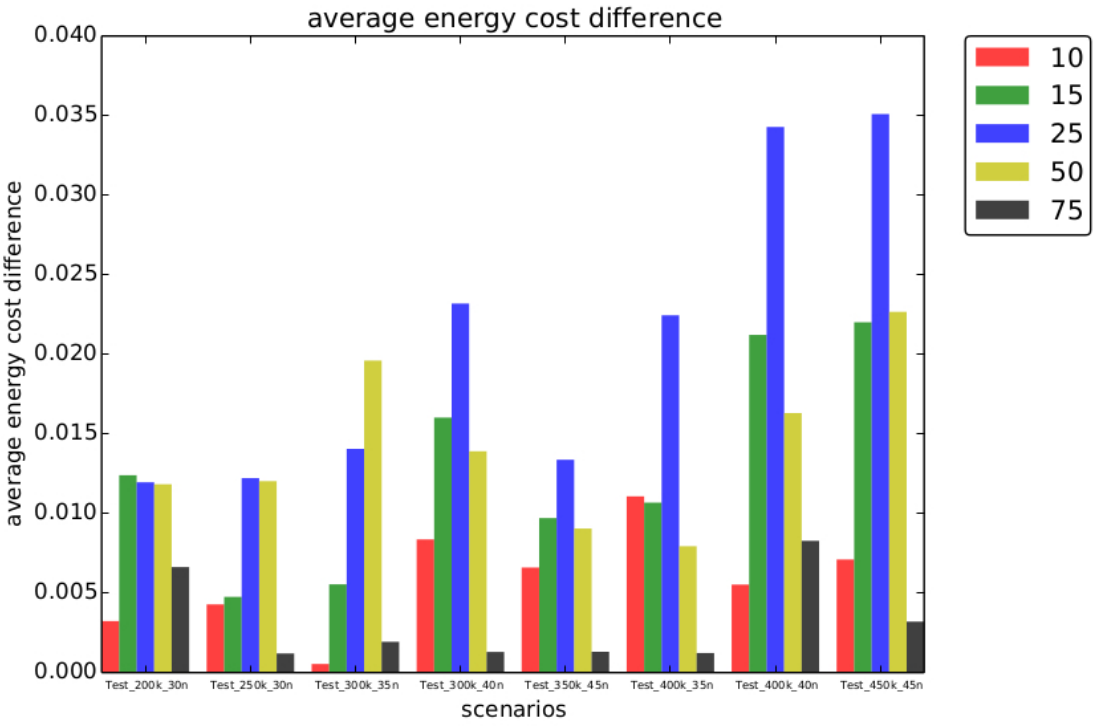


Figure 6.4: Average energy cost difference

From the previous chart, it is evident a bell behaviour in the average energy cost difference relative to each scenario with different average number of services per thing. For the first two average numbers, that are 10% and 15% there is a relative low difference in the average energy cost. With these percentages the solution of the service allocation problem is relative simple, so the gain between the single and multi service selection approach is low. Instead for the last two average numbers, that are 50% and 75% there is also a low difference but for another reason. In these case the solution of the allocation problem is more complex so it is lost the energy

cost gain between the multi and single service selection method. The maximum average gain is for the average number 25%. It represents a boundary scenario for which the gain between the single and multi service allocation method is maximum. As future works, we foresee new test scenarios in which the energy cost matrix and utilization matrix are computed in advance, in order to obtain only feasible allocations. The objective of this new kind of test scenario is to obtain a better comparison between the original RTTA version and the modified RTTA version with multi service allocation.

Chapter 7

Conclusions

In this work we tackled the study of the FIWARE platform and the problem of QoS-aware service selection applied to the FIWARE IoT architecture. First, we analyzed all the platform components and then we focused our attention on the IoT modules. We started studying the functioning of the IoT components and the communication protocols they use. Once we have chosen a group of FIWARE IoT modules, we have set up a deployment and tested it to understand what functions they offer. Once a deployment has been established, it is started the second phase of our work in which we have understood how implement a QoS solution in FIWARE IoT. To this aim, we have studied the available solutions proposed for the QoS-aware service selection in IoT cloud-based architectures. We have chosen the heuristic algorithm proposed in [3] and we have modified it in order to expand the set of solutions computed by the algorithm. Finally, we have integrated our solution in the FIWARE IoT architecture. The integration of the QoS support has been validated through a functional test. Instead, the new heuristic algorithm has been validated through simulations that demonstrated that it guarantees a greater set of solutions respect to the solution proposed in [3]. As future works, It is planned to extend the *QoSBroker* including the support for a larger set of operations, not only the queryContext. The *QoSBroker* could be extended to provides the *subscribeContext* operation. Another improvement concerns the heuristic algorithm that could be optimized trying

to compute to compute the optimal allocation in one shoot instead of iterating and stopping to the first feasible allocation. On the other side an extension can be developed on the discovery component. The actual implementation of the IoT Discovery doesn’t analyze restrictions about the geographical scope used to filter the available things.

Chapter 8

Acknowledgements

Ringrazio tutti coloro che mi hanno accompagnato in questo cammino universitario. Un ringraziamento al Prof.Enzo Mingozzi, a Carlo Vallati e a Giacomo Tanganelli per il loro aiuto e la loro disponibilità durante questi mesi di lavoro.

Bibliography

[1] J. Bradley, J. Barbier, and D. Handler, "Embracing the Internet of Everything to capture your share of $14.4 trillion," 2013.

[2] J. A. Stankovic, "Research directions for the Internet of Things," *IEEE Internet Things J.*, vol. 1, no. 1, pp. 3-9, Mar. 2014.

[3] G. Tanganelli, C. Vallati, E. Mingozzi, "Energy-Efficient QoS-aware Service Allocation for the Cloud of Things," 2014 IEEE 6th International Conference on Cloud Computing Technology and Science.

[4] E. Mingozzi, G. Tanganelli, C. Vallati, V. Di Gregorio, "An Open Framework for Accessing Things as a Service" in Proc. of the 16th International Symposium on Wireless Personal Multimedia Communications (WPMC 2013).

[5] E. Mingozzi, G. Tanganelli, C. Vallati, "A Framework for Quality of Service Support in Things-as-a-Service Oriented Architectures".

[6] FI-WARE basic guide, "http://www.slideshare.net/FI-WARE/fiware-basic-guide".

[7] FI-WARE wiki, "https://forge.fiware.org" Internet of Things (IoT) Services Enablement Architecture chapter.

[8] Jae-Hyun Cho, Han-Gyu Ko, In-Young Ko, "Adaptive Service Selection according to the service density in multiple QoS aspects," *IEEE Transactions on service computing*.

[9] Shih-Yuan Yu, Chi-Sheng Shih, Jane Yung-jen Hsu, Zhenqiu Huang, Kwei Jay Lin "QoS Oriented Sensor Selection in IoT System," *2014 IEEE International Conference on Internet of Things (iThings 2014), Green Computing and Communications (GreenCom 2014), and Cyber-Physical-Social Computing (CPSCom 2014)*.

[10] Valeria Cardellini, Valerio Di Valerio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, "QoS Driven Per-Request Load-Aware Service Selection in Service Oriented Architectures," *International Journal of Software and Inormatics*, vol. 7.

[11] Marco Abundo, Valeria Cardellini, Francesco Lo Presti, " Admission Control Policies for a Multi-class QoS-aware Service Oriented Architecture,".

[12] Ling Li, Shancang Li, Shanshan Zhao, "QoS-Aware Scheduling of Services-Oriented Internet of Things," *IEEE Transactions on industrial informatics*, vol. 10, no. 2, May 2014.

[13] Yu, Tao, Yue Zhang, and Kwei-Jay Lin. "Efficient algorithms for Web services selection with end-to-end QoS constraints." ACM Transactions on the Web. 2007.

[14] Boone, Bas, et al. "SALSA: QoS-aware load balancing for autonomous service brokering."Journal of Systems and Software. 2010.

[15] Menascé, D. A., E. Casalicchio, and V. Dubey. "On optimal service selection in service oriented architectures."Performance Evaluation 2010.

[16] Pentico, D. W. "Assignment problems: A golden anniversary survey."European Journal of Operational Research. 2007.

[17] Guenter Klas, Friedhelm Rodermund, Zach Shelby, Sandeep Akhouri, Jan Holler "Lightweight M2M: Enabling Device Management and Applications for the Internet of Things," White Paper.