5

An Open Source Framework for IoT Analytics as a Service

John Soldatos[1], Nikos Kefalakis[1] and Martin Serrano[2]

[1]Athens Information Technology, Greece
[2]Insight Center for Data Analytics, National University of Ireland, Galway, Ireland

5.1 Introduction

Earlier chapters have illustrated the importance of IoT and cloud computing convergence, as a means of achieving scalability and meeting QoS (Quality of Service) constraints. IoT deployments in the cloud are motivated by two main business drivers:

- **Business Agility**: Cloud computing alleviates tedious IT procurement processes, since it facilitates flexible, timely and on-demand access to computing resources (i.e. compute cycles, storage) as needed to meet business targets. In the case of IoT analytics applications, IoT developers and deployments can flexibly gain access to the storage and processing resources that they need in order to support their applications.
- **Reduced Capital Expenses**: Cloud computing leads to reduced capital expenses (CAPEX) (i.e. IT capital investments), through converting CAPEX to operational expenses (OPEX) (i.e. paying per month, per user for each service). This is due to the fact that cloud computing enables flexible planning and elastic provisioning of resources instead of upfront overprovisioning. Among the benefits of such flexibility is that it enables small and medium size enterprises (SMEs) to adopt a pay-as-you-go and pay-as-you-grow model to infrastructure acquisition and use, through paying for the computing resources and capacity that they need. This can be particularly important for the proliferating number of SMEs (including

high-tech startups), which exploit IoT analytics as part of their products or services.

Similarly to cloud computing infrastructures [1], integrated IoT/cloud infrastructures and related services can be classified to the following models:

- **Infrastructure-as-a-Service (IaaS) IoT/Clouds**: These services provide the means for accessing sensors and actuator in the cloud. The associated business model involves the IoT/Cloud provide to act either as data or sensor provider. IaaS services for IoT provide access control to resources as a prerequisite for the offering of related pay-as-you-go services.
- **Platform-as-a-Service (PaaS) IoT/Clouds**: This is the most widespread model for IoT/cloud services, given that it is the model provided by all public IoT/cloud infrastructures outlined above. As already illustrate most public IoT clouds come with a range of tools and related environments for applications development and deployment in a cloud environment. A main characteristic of PaaS IoT services is that they provide access to data, not to hardware. This is a clear differentiator comparing to IaaS IoT clouds.
- **Software-as-a-Service (SaaS) IoT/Clouds**: SaaS IoT services are the ones enabling their uses to access complete IoT-based software applications through the cloud, on-demand and in a pay-as-you-go fashion. As soon as sensors and IoT devices are not visible, SaaS IoT applications resemble very much conventional cloud-based SaaS applications. There are however cases where the IoT dimension is strong and evident, such as applications involving selection of sensors and combination of data from the selected sensors in an integrated applications. Several of these applications are commonly called Sensing-as-a-Service, given that they provide on-demand access to the services of multiple sensors. Note that SaaS IoT applications are typically built over a PaaS infrastructure and enable utility based business models involving IoT software and services.

Although the Sensing-as-a-Service paradigm is a special case of an SaaS deployment, it is in practice applicable to IoT applications only. Indeed, Sensing-as-a-Service applications involve on-demand collection, processing and analysis of information from sensors (i.e. IoT devices) [2]. The on-demand and dynamic nature of Sensing-as-a-Service applications in reinforced by the location dependent and time dependent nature of such IoT applications, which permit the dynamic selection of the IoT resources (sensors) that will provide the data streams to be processed. As such Sensing-as-Service can be seen as a case of an "IoT Analytics as a service" paradigm, where the IoT application users is allowed to dynamically specified data processing and analytics functionalities, along with the IoT devices on which they will be executed.

In this chapter we present a framework for implementing Sensing-as-a-Service applications based on the open source OpenIoT project [3]. The OpenIoT framework enables the dynamic selection of sensors and resources, as well as the subsequent specification of processing functionalities over the data of the selected sensors. In essence it enables the specification of dynamic sensor queries, which can be considered the first step towards IoT analytics as a service [4]. In addition to facilitating the dynamic definition and deployment of such Sensing-as-Service (or IoT analytics as a service) services, OpenIoT provides:

- Semantic interoperability and unification of data from diverse IoT sensors and other data sources, through ensuring their conversion and compliance to a common ontology, namely the OpenIoT ontology, which is an extended version of the W3C SSN (Semantic Sensor Networks) ontology [5].
- A range of easy-to-use tools for the visual specification of the Sensing-as-a-Service services. The tools enable the definition and deployment of SPARQL based sensor queries, through exploiting sensors registered within the OpenIoT framework.

Note that OpenIoT does not provide sophisticated data analytics functionalities, but it can well be extended on the basis of frameworks for data mining and machine learning, in order to support more advanced analytics functionalities. Such extensions are worked out in the scope of the H2020 FIESTA-IoT project, which provide functionalities for semantically interoperable IoT experimentation i.e. the execution of data-centric IoT experiments based on data streams from multiple IoT experimental facilities. Following sections of the chapter focus on the description of the OpenIoT framework and capabilities for Sensing-as-a-Service, along with a practical example of constructing and deploying a relevant sensor query based on the OpenIoT tools. Moreover, the enhancement of the Sensing-as-a-Service paradigm with more sophisticated analytics functionalities, towards an IoT Analytics as a Service paradigm is also discussed.

5.2 Architecture for IoT Analytics-as-a-Service

5.2.1 Properties of Sensing-as-a-Service Infrastructure

Service formulation and delivery in the scope of OpenIoT is characterized by the following properties:

- **On-demand**: Service formulation and delivery in OpenIoT should be performed on-demand. This implies the need for on-demand expressing requests for IoT services formulation, which shall be fulfilled by the OpenIoT middleware infrastructure. Therefore, service formulation should provide the means to dynamically selecting sensors and ICOs needed in order to satisfy the demanded service requests.
- **Cloud-based**: OpenIoT services are provided in a cloud environment. At the heart of this environment lies a scalable sensor cloud infrastructure, which shall provide sensor data access services. Thus, the OpenIoT service formulation strategies must take into account the need to access, use and combine services residing within the sensor/ICO cloud.
- **Utility-based**: Service delivery in OpenIoT is utility-based, which is in-line with the on-demand and cloud-based properties. As a result, OpenIoT should provide the means for calculating utility, through making provisions for storing a range of utility parameters (e.g., usage parameters for the employed ICOs) during the process of service formulation.
- **Service-Oriented**: OpenIoT requests will result in the deployment of services. The latter may be the composition of other services, such as services for accessing data streams in the cloud. Overall, OpenIoT has a service-oriented nature.
- **Optimized**: OpenIoT incorporates a wide range of self-management and self-optimization algorithms. The service formulation process ensures that information about resources reservation and usage is recorded in order to enable the implementation of utility-based optimization algorithms.

5.2.2 Service Delivery Architecture

The architecture of the OpenIoT platform is illustrated in Figure 5.1, while a more detailed overview of the interactions between the various modules is depicted in Figure 5.2. As already outlined, OpenIoT enables the cloud-based delivery of IoT data processing services, through enabling the creation of dynamic on-demand services. These services select and process data from a multitude of different data sources.

Overall, the architecture makes provisions for the creation and fulfillment of requests for services to the OpenIoT system. It is empowered by the following components:

- **Service Request Definition Component (“Request Definition”)**: Service Request Definition is the component where requests for IoT

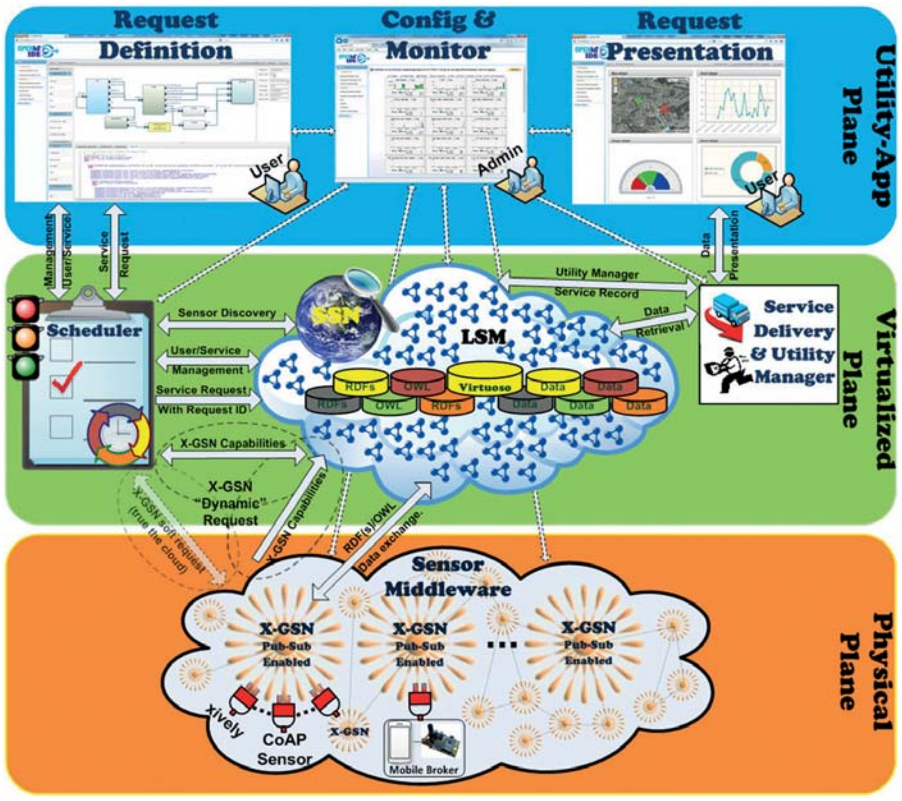


Figure 5.1 OpenIoT architecture.

services are formulated (by end-users) and accordingly submitted to the OpenIoT system. This component comes with an appropriate graphical user interface (GUI), which facilitates the service request customization.

- **(Global) Scheduler**: The Global Scheduler is in charge of accepting and prioritizing the various service requests (by one or more end-users) and accordingly generates the list of sensors (and other Internet Connected Objects (ICO)) that participate in the delivery of the service. Furthermore, the global Scheduler performs the required reservations of resources, which facilitate utility calculation and resource optimizations.
- **Service Discovery**: Service discovery refers to the OpenIoT directory services. It maintains the semantically annotated descriptions of the sensors that are known to the OpenIoT system. Service discovery

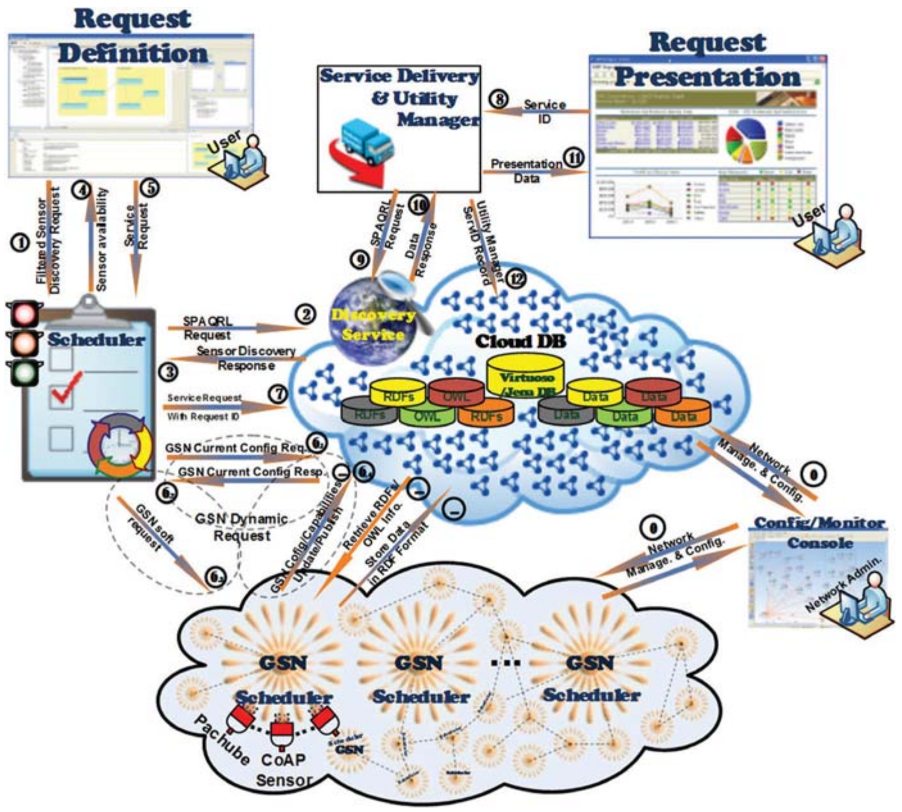


Figure 5.2 Functional blocks of openIoT's project analytics as a service architecture.

relies on the registration of sensors in the directory service repository. The structure of the service directory is based on the OpenIoT ontology (an enhanced version of the W3C SSN ontology).

- **Cloud Infrastructure**: This refers to the cloud computing environment (functional and operational), which ensures sensor cloud integration and streaming of sensors and ICO data to cloud storage, the operations performed in the cloud infrastructure are independent of the infrastructure management and the infrastructure modifications.
- **Global Sensor Networks (GSN) Nodes**: GSN nodes refer to deployment instances of the GSN middleware [6]. They play a significant role in the data provisioning for the IoT service delivery, since they enable the interfacing of physical-world devices to the OpenIoT system (via the cloud infrastructure). At the same time, GSN nodes perform a range of

local-level optimizations, on the basis of the ICOs that they comprise of and on how these participate and contribute to the various services, prioritizing and transforming the data acquired form the physical sensors into normalized data.

- **Service Delivery and Utility Manager (SD&UM)**: The service delivery manager ensures the proper assembly and delivery of the services subject to the various constraints imposed either for physical infrastructure restrictions or service customization definitions. To this end, it uses the selected sensors and ICOs and combines them as specified in the service request sent to the system. The combination depends also on the optimizations performed by the OpenIoT infrastructure, given that these optimizations may, for example, regulate the frequency of accesses to the various underlying data services.
- **Service Presentation ("Request Presentation")**: This component facilitates the implementation of the presentation layer of the service, on the basis of mashups and other visualization libraries. It can be considered as an optional component aiming at easing the presentation of the services according to the preferences and needs of the end-user.

5.2.3 Service Delivery Concept

In-line with the main components of the OpenIoT architecture outlined above, service delivery is based on the selection and orchestration of multiple services (including cloud services) that provide data and/or instigate tasking or actuation functionalities. The orchestration and combination of those services is based on the following factors:

- **Type of (requested) service**: The service request specifies different possible operations on ICOs, such as selection, retrieval and processing of their data, or execution of actuation commands. The OpenIoT sensor cloud infrastructure can be seen as a large-scale distributed sensors and ICO database. Service requests can be thought of as queries and operations over this database (i.e. SQL (Structured Query Language) can be thought as a representative metaphor). Depending on the query and operation, the OpenIoT infrastructure will instigate alternative paths within the service delivery strategies. For example, query operations (i.e. «SELECT» in SQL terms) will lead to the combination of sensor data access services, while actuating operations (i.e. «UPDATE» or «EXECUTE» operations) will lead to the invocation of the actuating services. Furthermore, requests combining both actuation and selection functions should trigger alternative paths within the OpenIoT service formulation strategies.

- **Optimizations**: The OpenIoT sensor cloud is a self-managing infrastructure, which provides opportunities for optimal delivery of the services. Therefore, the resource management and optimization capabilities of the OpenIoT infrastructure affect service formulation and delivery. Alternative service delivery and execution paths are likely to be considered in the scope of the optimization of the OpenIoT services.
- **Sensor and ICO selection**: The selected sensors and ICOs influence the formulation and delivery of services. Different ICOs may provide different capabilities in terms of data selection and actuating services execution. Therefore, the OpenIoT service delivery environment deals with the heterogeneity of data being collected from the various ICOs. In particular, the OpenIoT cloud and the underlying GSN nodes provide a virtualized interface for accessing the low-level capabilities of the ICOs acting as data collectors for the OpenIoT system.

Finally, the service formulation and delivery mechanisms consider the need to support both service deployment and service un-deployment. Service un-deployment should be implementing as an integral element of the service management and governance functions in OpenIoT. The un-deployment process is therefore addressed in later paragraphs as well.

5.3 Sensing-as-a-Service Infrastructure Anatomy

5.3.1 Lifecycle of a Sensing-as-a-Service Instance

As part of the OpenIoT system, the management and requests operations for dynamically creating and deploying IoT services (i.e. Sensing-as-a-Service and IoT-Analytics-as-a-Service services) perform the following main tasks:

- **Formulation of the request**: As part of this task the request is formed on the basis of the specification criteria for particular sensor selection, as well as of the processing of the resulting collected data.
- **Parsing and validation of the request**: This task processes the request and ensures its validity. The validation of the requests ensures that they refer to existing sensors and ICOs or that the criteria set lead to the selection of a group of sensors and ICOs.

- **Discovery of resources**: In the scope of this task the criteria for selecting sensors are applied against the OpenIoT directory services i.e. the sensor directory is used to select a set of sensors that fulfill the relevant criteria and when need it update the OpenIoT directory sensor services.
- **Instantiation of a new OpenIoT service**: With the selected sensors/ICOs at hand, a new OpenIoT service instance is created as a cloud service. This results in the establishment of the service that is associated with the Sensing-as-a-Service request.
- **Population of information and structures associated with utility metering and resource management**: Along with the creation of the OpenIoT service, the appropriate resources are reserved. This is denoted in the various structures that comprise information about the resources of the OpenIoT system. Furthermore, structures/records for the utility metrics are used.
- **Deployment/Delivery of the service**: As part of this task, the OpenIoT service is deployed and becomes available on the OpenIoT system. Consequently, it becomes ready to be invoked by end-users.

Figure 5.3 illustrates the main system actions entailed in the course/process of deploying an OpenIoT service (i.e. service request, sensor(s) selection, scheduling and resources reservation and ultimately service deployment). Following the successful deployment of an OpenIoT service, end-users can invoke and use it. As part of the service lifecycle, it is also likely that the service will be uninstalled and deactivated from the system, in which case all resources associated with the service will be released.

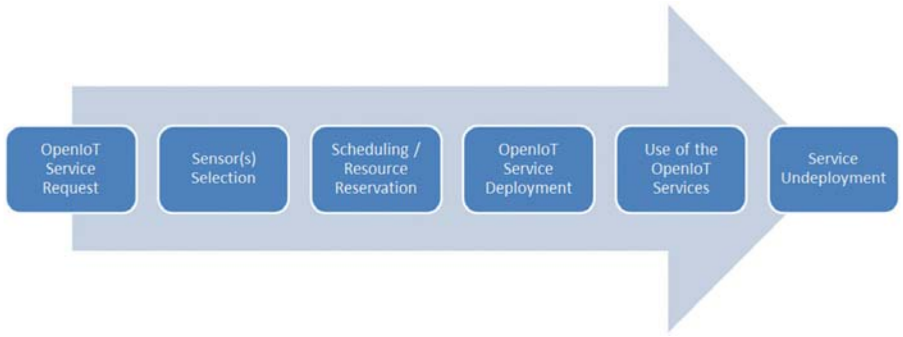

OpenIoT Service Request
Sensor(s) Selection
Scheduling / Resource Reservation
OpenIoT Service Deployment
Use of the OpenIoT Services
Service Undeployment

Figure 5.3 IoT data analysis services request lifecycle.

5.3.2 Interactions between OpenIoT Modules

OpenIoT is a sensor cloud environment. Along with the data of the various sensors and ICO streams, this cloud stores a wide range of meta-data enabling the deployment, delivery and optimization of IoT services within the sensor cloud. This meta-data is updated during the operation of the sensor cloud system, as new services are requested and deployed, while others go out of scope. The sensor cloud system will be responsible to frequently check if data are required from the system's deployed services from the provided mechanisms. Furthermore, this meta-data will regulate the interactions between the various components of the OpenIoT architecture. Figure 5.4 [10] illustrates the various modules of the OpenIoT architecture, along with their interactions (indicated based on uni-directional and bi-direction arrows). Furthermore, the figure illustrates the various entities/classes, whose values/data are used in the scope of the interactions of the modules. In particular, given the entities illustrated in Figure 5.4, each of the OpenIoT modules interacts with the others as follows:

- **Request Definition**: The request definition module is the user interface that enables the user to formulate the requests in the OpenIoT system. This module interacts directly with the Scheduler's API which is described in detail in following sections.
- **(Global) Scheduler**: The Scheduler formulates the request based on the user inputs (request definition). It interacts with the rest of the OpenIoT platform through the Cloud Database (DB). In particular, the Scheduler performs the following functions:
    - Retrieving the available sensors from the GSN nodes through the "availableSensors" entity,
    - Informing the GSN nodes abut which of their virtual sensors are used by the service being scheduled. Relevant information is includes in the "sensorServiceRelation" entity,
    - Informing the Service Delivery & Utility Manager (SD&UM) about what services to deliver based on the "serviceDeliveryDescription" entity,
    - Notifying the user, via itself and the SD&UM module, about the status of a defined service through the "serviceStatus" entity, and
    - Implementing access control mechanisms with the help of the "user" entity.

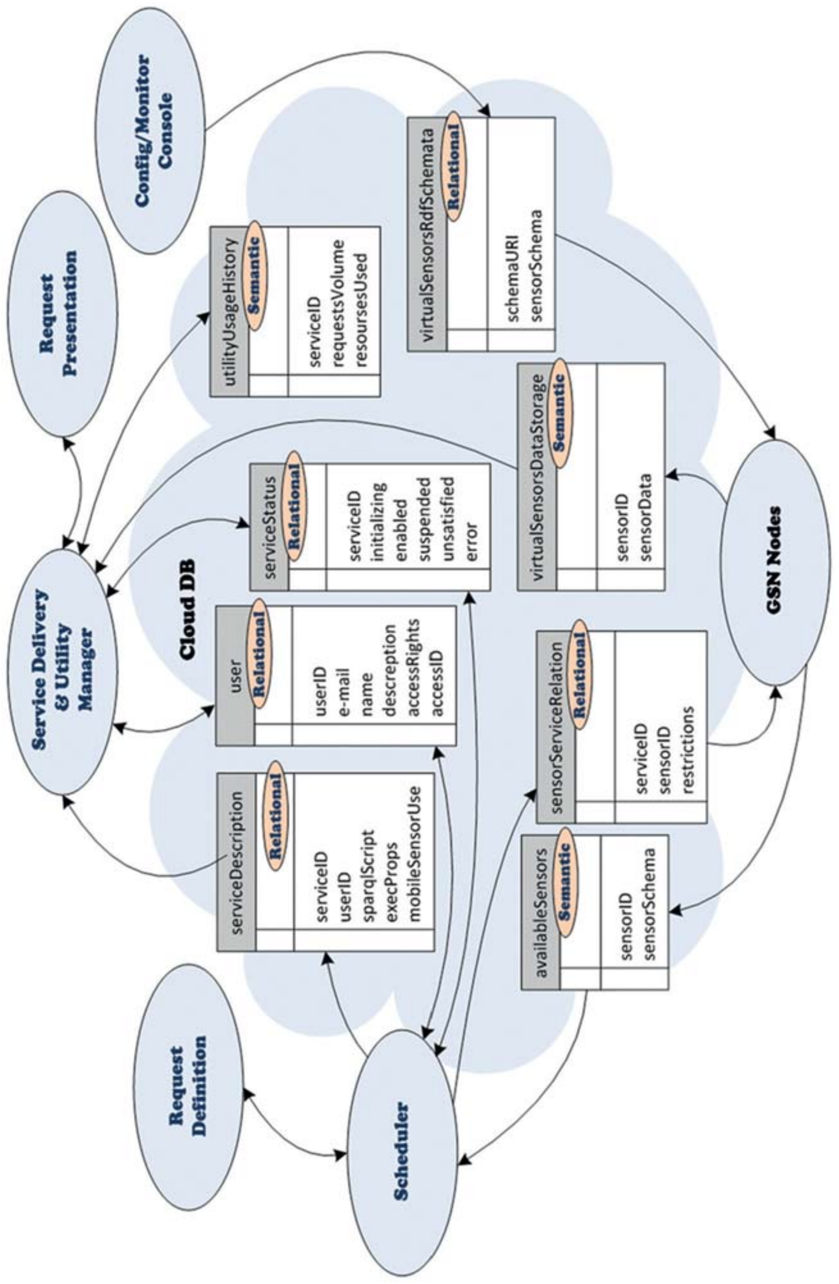


Figure 5.4 Main entities and modules.

- **Service Delivery & Utility Manager**: The SD&UM module provides results to the request presentation module by retrieving SPARQL scripts [7], that the Scheduler has provided to the "serviceDescription" entity. Furthermore, this module retrieves data from the GSN nodes by executing the retrieved scripts to the "virtualSensorsDataStorage". Moreover it is able to store resource-usage history for accounting, metering and billing purposes.
- **Request Presentation**: the Request Presentation module is the User Interface that enables the user to retrieve data from the Cloud Database (DB). The Request Definition has described the request and the data is delivered using the SD&UM API described below.
- **Configuration Console**: The Configuration/Monitoring console is the system administrators' tool, which enables administrators to deploy, configure and manage the OpenIoT platform. It interacts directly with several other modules (Scheduler, SD&UM and GSN nodes) for monitoring purposes. Finally, it is also capable to set up RDF schemata for new virtual sensors. The schemata are stored within the "virtualSensorsRdfSchemata" entity and enable GSN nodes to access this information during their configuration.
- **GSN Nodes**: The GSN nodes (or virtual sensors) are:
    - Providing the available sensors to the Scheduler module through the "availableSensors" entity,
    - Informed about the sensors in use from the Scheduler based on the "sensorServiceRelation" entity,
    - Retrieving new virtual Sensors RDF schemata from the Config/Monitor Console through the "virtualSensorsRdfSchemata" entity, and
    - Providing sensor data to the SD&UM through the "virtualSensorsDataStorage" entity.

As part of these interactions the above modules create and consume data associated with the entities listed in the following table [10]. Note that the table differentiates between semantic and non-semantic data entities. Semantic data entities are implemented on the basis of ontologies (i.e. RDF), while non-semantic data structures are represented on the basis of relational database tables. Note that all the structures that hold sensor information follow semantic descriptions, given that all sensor descriptions in OpenIoT will be semantically annotated and represented.

Data Entity	Type	Description
serviceDescription	Relational (SQL) or RDF	Holds the description and properties of all the services that are executed through the OpenIoT system
availableSensors	Semantic (RDF)	Constitutes the directory database of the OpenIoT sensor cloud system.
serviceStatus	Relational (SQL)	Maintains a list with the status of the services, in order to provide relevant feedback to end-users
sensorServiceRelation	Relational (SQL)	Maintains the (many-to-many) associations of the services to the various sensors and ICOs available in the system (i.e. information about which sensors are used in the scope of a given services).
virtualSensorsDataStorage	Semantic (RDF)	Maintains the data of the various data streams i.e. data corresponding to the data streams of the sensors and ICOs that provide services to OpenIoT users
virtualSensorsRdfSchemata	Semantic (RDF)	Holds the structure of specific sensors/ICO types to allow for the management and instantiation of the sensors.
utilityUsageHistory	Semantic (RDF)	Used to records utility/usage related parameters, in order to boost accounting, billing and (utility based) resource optimization
user	Relational (SQL) or RDF	Used to store the available users and their access rights to implement access control mechanisms.

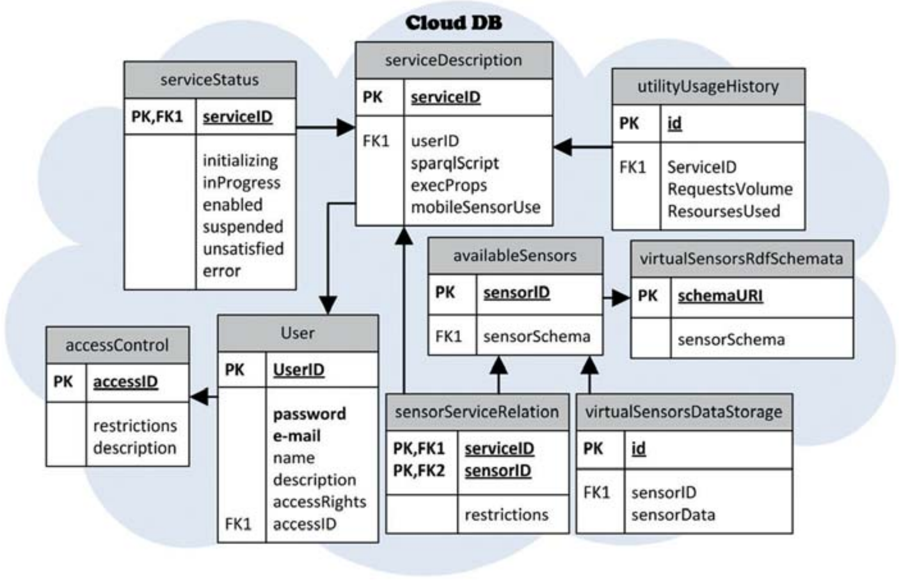


Figure 5.5 Relationships between the main OpenIoT data entities.

The relationship between the main OpenIoT data entities is depicted in Figure 5.5 [10].

5.4 Scheduling, Metering and Service Delivery

The modules that are responsible for the services formulation within the OpenIoT platform are the “Scheduler” and the “Service Delivery & Utility Manager”. Following paragraphs provide a detailed description of these modules, including the functionalities that they offer to end-users. Note that the term end-user can either denote the final user of the IoT services or the solution provider exploiting the OpenIoT capabilities in order to integrate and deploy a Sensing-as-a-Service solution.

5.4.1 Scheduler

The Scheduler is the main and first entry point for service requests submitted to the OpenIoT cloud environment. This component receives the service requests from the service definition components as part of the process of creating a new cloud service based on the Sensing-as-a-Service paradigm. It parses each service request and accordingly performs two main functions towards the delivery of the service, the sensor/ICO selection and the scheduling/resource reservations.

The API of the scheduler supports the lifecycle of the OpenIoT service, which has been presented in earlier paragraph. In particular, it provides the means for:

- Constructing an OpenIoT service on the basis of existing sensors and ICOs.
- Registering an OpenIoT service within the OpenIoT sensor cloud. In this case the OpenIoT system assigns a service identifier (serviceID) to the service, which uniquely identifies the service within the OpenIoT service delivery system.
- Unregistering a (previous registered) OpenIoT service. This is a counterpart function to the one registering the service. The unregistration/deregistration function moves the service out of the scope of the OpenIoT system.
- Enabling an already registered service, thereby commencing its operation within the OpenIoT sensor cloud.
- Disabling an OpenIoT service, thereby leading to its deactivation within the sensor cloud. Disabling a service does not however imply that the service goes out of the scope of the sensor cloud i.e. it still remains available for activation.
- Querying the status of a given service, as a means of accessing the state of the service within the sensor cloud.

The above functions change the state of the OpenIoT services according to rules and dependencies specified within the various states. For example, only registered services can be enabled, and only enabled services can be disabled. At the same time, only registered services can be unregistered.

Figure 5.6 [8] illustrates the lifecycle of the IoT services within the OpenIoT system. The transitions between the different states occur on the basis of invocations to the Scheduler API.

On the basis of the Scheduler API, the following functionalities are supported:

- **Resource Discovery**: This service will discover virtual sensor availability based on the “availableSensors” entity. It will provide the resources that match the requirements for a given service request.
- **Service User Management**: This Scheduler service will enable the management of the lifecycle of an OpenIoT service. This lifecycle management is performed based on the following Scheduler comments:

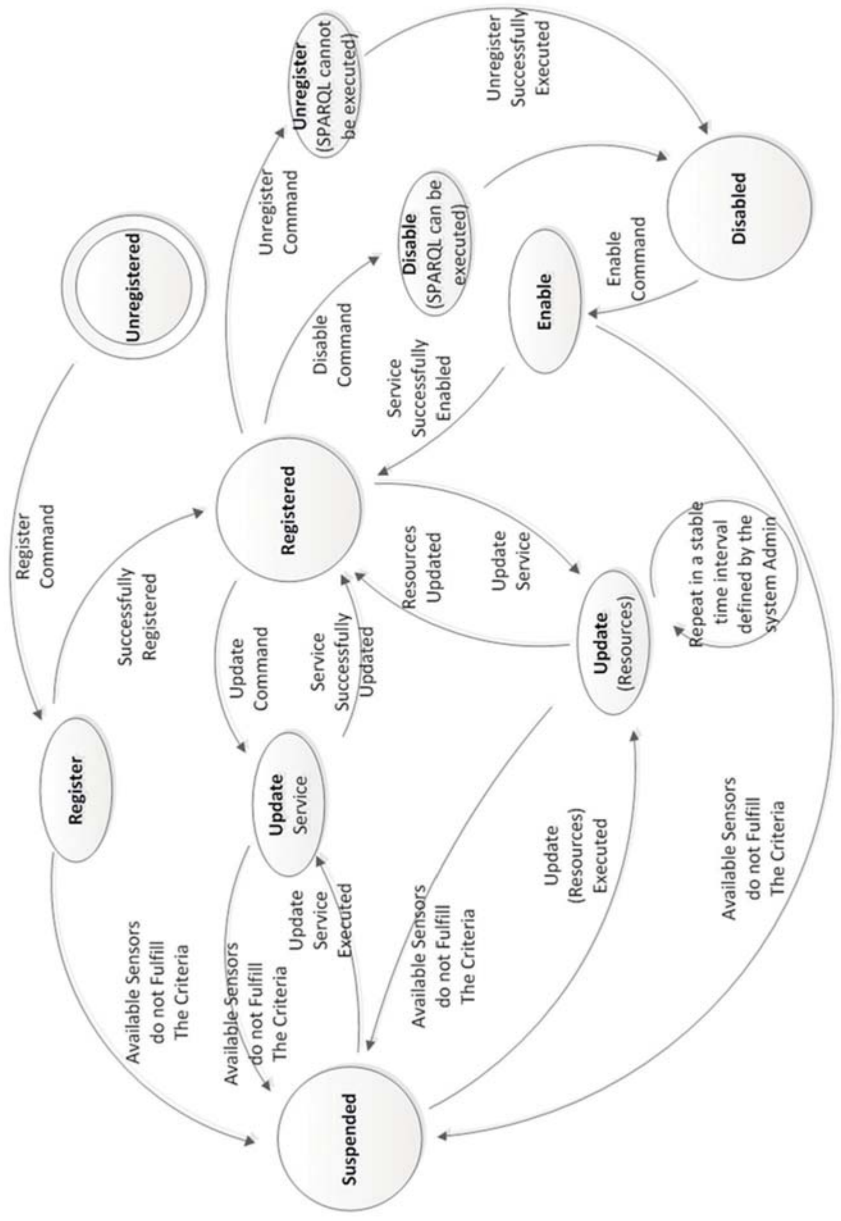

Unregistered
Register Command
Register
Successfully Registered
Registered
Unregister Command
Unregister (SPARQL cannot be executed)
Unregister Successfully Executed
Disable Command
Disable (SPARQL can be executed)
Disabled
Enable Command
Enable
Service Successfully Enabled
Update Command
Update Service
Service Successfully Updated
Update Service Executed
Resources Updated
Update Service
Update (Resources)
Repeat in a stable time interval defined by the system Admin
Update (Resources) Executed
Available Sensors do not Fulfill The Criteria
Available Sensors do not Fulfill The Criteria
Available Sensors do not Fulfill The Criteria
Available Sensors do not Fulfill The Criteria
Suspended

Figure 5.6 State diagram of the OpenIoT services lifecycle within the scheduler module.

- *Register*: The "Register" service is responsible to identify all the required resources from the request and update the "sensorService-Relation" entity at the cloud database. The "Register" service shall formulate a SPARQL script, based on the user request, and shall store it to the "ServiceDescription" entity along with a Service ID and user's specific execution properties (the execution properties could include execution intervals, life of the service, etch). A new service instance shall get recorded at the "serviceStatus" entity in the cloud. Note that: (a) In case the request is satisfied the unsatisfied Boolean of the "serviceStatus" entity is set to false, whereas (b) If the request is unsatisfied the unsatisfied Boolean of the "serviceStatus" entity is set to true. Optionally more detailed information regarding the problem could be stored.
- *Unregister*: In the scope of the unregister functionality the user will have the ability to unregister a registered service. When a service gets unregistered the allocated resources shall get released. Therefore, the service-virtual sensor relation at the "SensorService-Relation" entity in the cloud shall get deleted. Furthermore, the service gets deactivated (set enabled as false) at the "serviceStatus" entity in the cloud.
- *Suspend*: As part of suspend functionality, the service shall get updated (set suspended as true) at the "serviceStatus" entity in the cloud.
- *Enable from Suspension*: As part of the suspension functionality the service is defined as enabled (enabled is true) at the "serviceStatus" entity.
- *Enable*: The enable functionality gives to the user will be given the ability to enable an unregistered service. When a service gets enabled the user request gets initialized and the related virtual sensors are identified and stored to the "SensorServiceRelation" entity. The service is set as enabled at the "serviceStatus" entity.
- *Update*: The update services permits changes to service. When a registered service gets updated the "Update" identifies all the required resources from the updated request and updates the "Sensor-ServiceRelation" entity at the cloud database. The "Update" service shall formulate a SPARQL script, based on the updated user request, and shall update it to the existing one along with the updated user's specific execution properties (the execution

properties could include execution intervals, life of the service, etch) at the “ServiceDescription” entity. The service status shall get updated as enabled at the “serviceStatus” entity in the cloud. Note that: (a) In case the request is satisfied the unsatisfied Boolean of the “serviceStatus” entity is set to false, (b) In case the request is unsatisfied the unsatisfied Boolean of the “serviceStatus” entity is set to true. Optionally more detailed information regarding the problem could be stored.

- **Registered Service Status**: This functionality enables the user to retrieve the status of a specific service by providing the ServiceID. The Registered Service Status service shall check the “serviceStatus” entity and send all the available information back to the user.
- **Service Update Resources**: Based on a service provider (i.e. administrator controlled) specified time interval this service/functionality shall check the enabled services from the “serviceDescription” entity and as a first step identify the ones that are using mobile sensors. As a second step it shall check if the mobile sensors fulfil the User’s request (e.g. in respect of a specific location). Note that: (a) in case the sensor fulfills the user’s request no further action is taken and (b) in case the sensor does not fulfil the user’s request this sensor is unrelated/removed from the specific service at the “sensorServiceRelation” entity and (c) as a third step a new sensor is searched that fulfils the user’s request (e.g. in respect of a specific location), (d) in case a new sensor is found it gets recorded at the “ServiceDescription” entity and the “serviceDescription” entity gets updated, (e) in case is no sensor available that fulfils the specific request the unsatisfied field shall get updated with “true” at the “serviceStatus” entity in the cloud.
- **Get Service**: This service is used to get the description of a registered service. Accessing the “serviceDescription” entity retrieves this information.
- **Get the Available Services**: This service provides the ability to a user to collect a list of registered services related with a specific user. These service IDs are available from the “serviceDescription” entity.
- **Get User**: This service is used by the OpenIoT platform’s access controls mechanisms so as to retrieve user’s information, access rights and restrictions to implement data filtering and access rights.

Note that for the user to be able to invoke the “Resource Discovery”, “Service User Management”, “Registered Service Status”, “Service Update

Resources", "Get Service", "Get User" and "Get the available Services" services, the user must first get logged-in to the system by authenticating with his/her ID. Moreover, the results provided to the user are prior filtered based on his/her account restrictions and the resources that are accessible based on his/her profile. The "user" and the "accessControl" entities provide the account restrictions data.

In line with the Scheduler functionalities presented above, Figure 5.7 [8] illustrates the main workflow associated with the service registration process. In the scope of this process the Scheduler attempts to discover the resources (sensors, ICO) that will be used for the service delivery. In case no sensors/ICOs can fulfill the request, the service is suspended. In case a set of proper sensors/IOCs is defined the relevant data entities are updated (e.g., relationship of sensors to services) and a SPARQL script associated with the service is formulated and stored for later use. Following the successful

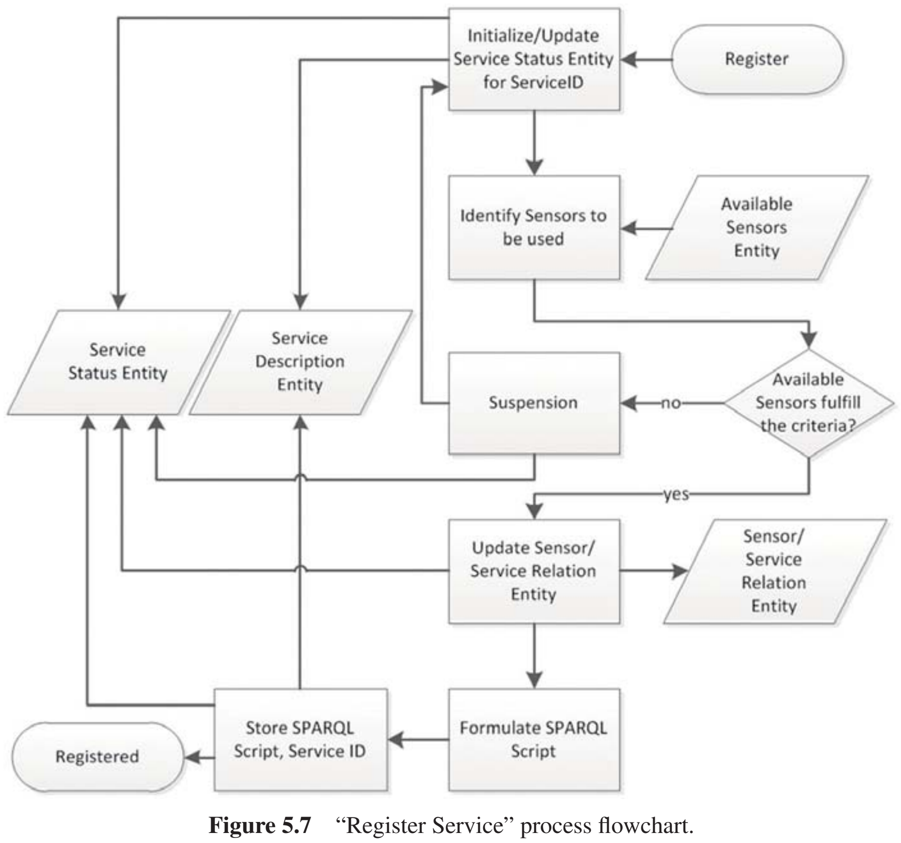

Initialize/Update Service Status Entity for ServiceID
Register
Identify Sensors to be used
Available Sensors Entity
Service Status Entity
Service Description Entity
Suspension
no
Available Sensors fulfill the criteria?
yes
Update Sensor/ Service Relation Entity
Sensor/ Service Relation Entity
Registered
Store SPARQL Script, Service ID
Formulate SPARQL Script

Figure 5.7 "Register Service" process flowchart.

conclusion of this process, the servicer enters the «Registered» state and is available for invocation.

Likewise Figure 5.8 [8] illustrates the process of updating the resources associated with a given service. As already outlined, such an update process is particularly important when it comes to dealing with IoT services that entail mobile sensors and ICOs i.e. sensors and ICOs whose location is likely to change within very short timescales (such as mobile phones and UAV (Unmanned Aerial Vehicles)). In such cases the update resources process could regularly check the availability of mobile sensors and their suitability for the registered service whose resources are updated. The workflow in Figure 5.7 assumes that the list of mobile sensors is known to the service (i.e. the sensors' semantic annotations indicate whether a sensor is mobile or not).

Even though the process/functionality of updating resources is associated with the need to identify the availability and suitability of mobile sensors, in principle the update process could be used to update the whole list of resources that contribute to the given service. Such functionality could help OpenIoT in dealing with the volatility of IoT environments, where sensors and ICOs may dynamically join or leave. In the scope of an IoT application, one cannot rule out the possibility of the emergence of new sensors that can be associated with an already established service.

Finally, Figure 5.9 [8] illustrates the process of unregistering a service, in which case the resource associated with the service is released. The data structures of the OpenIoT service infrastructures are also modified to reflect the fact that the specified service no longer using its resources. As already explained, this update is important for the later implementation of the OpenIoT self-management and optimization functionalities.

5.4.2 Service Delivery & Utility Manager

The Service Delivery & Utility Manager has (as its name indicates) a dual functionality. On the one hand (as a service manager) it is the module enabling data retrieval from the selected sensors comprising the OpenIoT service. On the other hand, the utility manager maintains and retrieves information structures regarding service usage and supports metering, charging and resource management processes. The following paragraphs elaborate on the main functionalities/services of the Service Delivery & Utility Manager.

The API of the Service Delivery & Utility Manager (SD&UM) serves as the point where the OpenIoT platform provides its outcome. In particular, the module provides the means for:

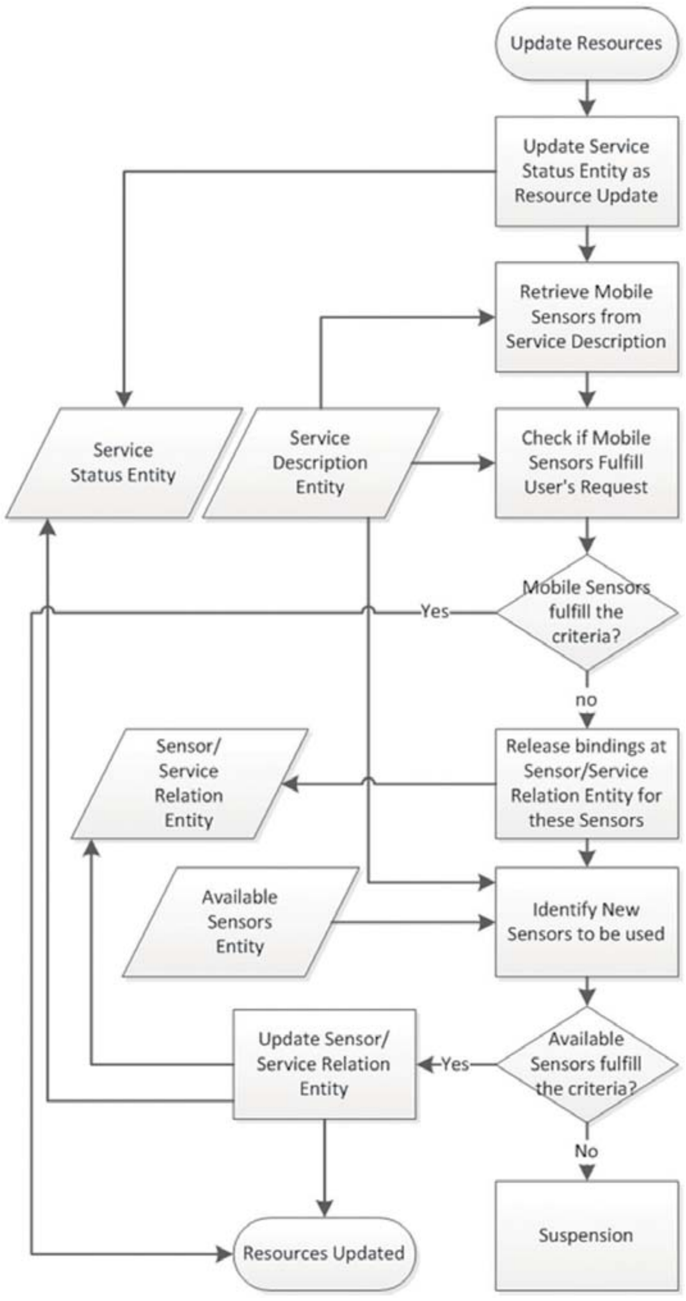

Update Resources
Update Service Status Entity as Resource Update
Retrieve Mobile Sensors from Service Description
Service Status Entity
Service Description Entity
Check if Mobile Sensors Fulfill User's Request
Mobile Sensors fulfill the criteria?
Yes
no
Sensor/ Service Relation Entity
Release bindings at Sensor/Service Relation Entity for these Sensors
Available Sensors Entity
Identify New Sensors to be used
Update Sensor/ Service Relation Entity
Yes
Available Sensors fulfill the criteria?
No
Resources Updated
Suspension

Figure 5.8 "Update Resources" service flowchart.

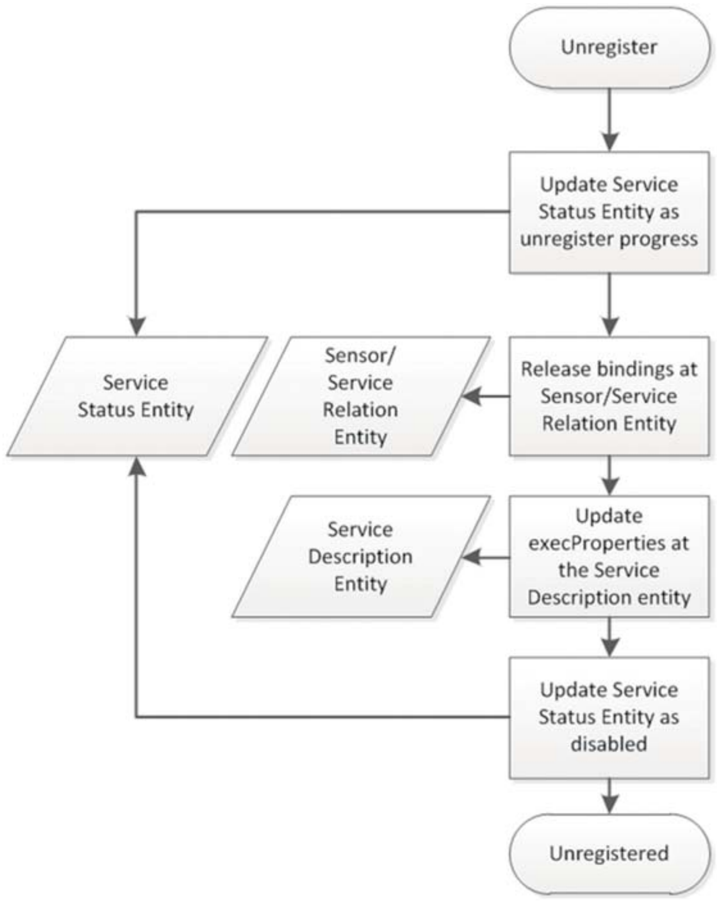

Unregister
Update Service Status Entity as unregister progress
Service Status Entity
Sensor/ Service Relation Entity
Release bindings at Sensor/Service Relation Entity
Service Description Entity
Update execProperties at the Service Description entity
Update Service Status Entity as disabled
Unregistered

Figure 5.9 "Unregister" service flowchart.

- Executing and delivering the requested services.
- Accessing and processing data streams from the cloud.
- Taking into account processing instructions specified during the request formulation.
- Keeping track of utility parameters associated with the service, for example: the time the service is used, the volume of data transmitted, as well as the number and type of sensors used.
- Managing and maintaining utility data records.

On the basis of the Service Delivery & Utility Manager API, the following functionalities are supported:

- **Subscribe for a report**: This service enables the user to invoke an already defined service from the "ServiceDescription" entity. By providing an application's destination address (URI) this service will collect the results from the predefined query (sparqlScript), which is stored at the "ServiceDescription" entity, and deliver it to the application via the Callback Service.
- **Callback Service**: this service is instantiated by the "Subscribe for a report" service and invoked based on the schedule defined by the user at the service registration time. If the query is executed normally, the callback service invokes the callback results service.
- **Callback results**: By invoking the callback results the SD&UM will attempt to deliver results to the subscriber application.
- **Unsubscribe for a report**: This service is invoked by the user and deactivates the "Subscribe for a report" one. The previously registered subscription removal is identified by the user by providing a unique subscription ID.
- **Poll for a report**: This service enables the user to invoke an already defined service from the "serviceDescription" entity. The difference with the "subscribe for a report" service is that it enables the user to execute the predefined query with modified parameters (i.e. give me the results of the last 30 min) and that this call will produce a single Result Set (it will be executed only once and then it will be dropped). In case the query executes normally, the "Poll for a report" service invokes the callback results service.
- **Get the utility usage of a user**: This service enables the user to retrieve the utility usage involved for a specific user. By providing the user's ID the "Get the utility usage of a user" service retrieves the related services with the specific user from the "serviceDescription" entity. It then collects the usage history from the "utilityUsageHistory" entity and by using special utility usage algorithms and in relation with the policies applied for the provided services, it returns the overall usage/cost of the platform for the selected user.
- **Get the utility usage of a registered service**: This service enables the user to retrieve the utility usage related with a specific registered service. By providing the "serviceID" it collects the usage history from the "utilityUsageHistory" entity and by employing special utility usage algorithms combined with the charging policies specified for the provided services it returns the usage/cost of the platform for the selected service.

- **Record utility usage of a service**: This service is invoked from the “Poll for a report” and the “Callback service” services. On its invocation the volume of the requested data and the type of resources used, are stored to the “utilityUsageHistory” entity for later use from the “Get the utility usage of a registered service” and the “Get the utility usage of a user” services.
- **Get service status**: This service enables the user to retrieve the status of a specific service by providing the service ID. The registered service status service shall check the “serviceStatus” entity and send to the user all the available information.
- **Get service**: This service is used to get the description of a registered service. This information is retrieved by accessing the “serviceDescription” entity.
- **Get the available services**: This service provides the ability to a user to collect a list of registered services related with a specific user. These service IDs are available from the “serviceDescription” entity.
- **Get User**: This service is used by the OpenIoT platform’s access controls mechanisms so as to retrieve a user’s information, access rights and restrictions in order to implement data filtering and access rights.

Note that to be able to invoke the “Subscribe for a report”, “Unsubscribe for a report”, “Poll for a report”, “Get the Utility Usage of a User”, “Get Service”, “Get User” and “Get the available Services” services the user must first get logged-in to the system by authenticating with his/her ID. Moreover the results provided to the user are prior filtered based on his/her account restrictions and the resources which are accessible based on his/her profile. The account restrictions data are provided by the “user” and the “accessControl” entities.

5.5 Sensing-as-a-Service Example

Following paragraphs illustrate the process of establishing a fully deployable service (from data Capturing to Visualization) using the OpenIoT reference framework and its Sensing-as-a-Service capabilities.

5.5.1 Data Capturing and Flow Description

In this example, weather sensors are deployed in the central area of Brussels producing data (wind chill temperature, atmospheric pressure, air temperature, atmosphere humidity and wind speed).

The data are captured using the GSN middleware[1] through a special wrapper (i.e. residing at the physical plane of the architecture depicted in Figure 5.1) which collects the Weather Station's data every 4 hours. This is where the first level of data filtering occurs, whereas the weather station produces data in a higher rate, in this scenario we are interested in a four hour sampling rate. The captured data are following a sensor type created for this occasion (named after "Weather"). The "Weather" sensor type is used to semantically annotate the captured data at the GSN level. One GSN instance is running for every weather station so after X-GSN announces the existence of each sensor (bound with a specific sensor id) it starts to push the captured data to Linked Sensor Middleware (LSM) components, which comprises an RDF Store and is deployed in a private cloud environment (the virtualized plane of the architecture Figure 5.1).

Then it is time to set up the service by using the Request Definition (the utility/application plane of Figure 5.1) tool with the help of which we will discover these sensors (by using the Scheduler), describe the request and send it to the Scheduler (the virtualized plane of Figure 5.1) to handle it. The Scheduler decomposes the request and registers it to LSM. The information that should be accessed and processes in this scenario is the wind chill temperature versus the actual air temperature in the area of Brussels for the dates between 01/07/2014 and 01/28/2014.

The SD&UM (the virtualized plane in Figure 5.1) retrieves on demand the formulated request executes the involved queries and feeds the Request Presentation (i.e. the utility/application plane of the OpenIoT architecture) with presentation data. The last step would be for the Request Presentation to presents the received data in the predefined widgets.

The presented high level description of the data flow at the virtualized and utility/application planes is in following paragraphs built and presented as an OpenIoT Sensing-as-a-Service application.

5.5.2 Semantic Annotation of Sensor Data

The association of metadata with a virtual sensor is performed through an appropriate metadata file. For example, a virtual sensor named Brussels_weather.xml will have an associated metadata file named Brussels_weather.

[1]Also called X-GSN (extended GSN) in the context of OpenIoT, where an enhanced version of the original GSN middleware that supports semantic annotation of virtual sensors has been deployed.

metadata. The metadata file contains information such as the location (in coordinates), as well as the fields exposed by the virtual sensor. This also includes the mapping between a sensor field (e.g. airtemperature) and the corresponding high-level concept of the ontology (e.g., http://openiot.eu/ontology/ns/AirTemperature).

```
sensorID="http://lsm.deri.ie/resource/61330620147099"
sensorName=979128
source="Brussels netatmo"
sensorType=weather
information=Weather sensors in Brussels
author=openiot
feature="http://lsm.deri.ie/OpenIoT/BrusselsFeature"
fields="pressure,airtemperature,humidity,visibility,win
dchill,windspeed"
field.airtemperature.propertyName="http://openiot.eu/on
tology/ns/AirTemperature"
field.airtemperature.unit=C
field.humidity.propertyName="
http://openiot.eu/ontology/ns/AtmosphereHumidity"
field.humidity.unit=Percent
field.visibility.propertyName="
http://openiot.eu/ontology/ns/AtmosphereVisibility"
field.visibility.unit=Percent
field.pressure.propertyName="
http://openiot.eu/ontology/ns/AtmosphericPressure"
field.pressure.unit=mb
field.windchill.propertyName="
http://openiot.eu/ontology/ns/WindChill"
field.windchill.unit=C
field.windspeed.propertyName="
http://openiot.eu/ontology/ns/WindSpeed"
field.windspeed.unit=Km/h
latitude=51.33332825
longitude=3.200000048
```

5.5.3 Registering Sensors to LSM

Sensors can be registered to the LSM middleware (and its cloud datastore) by executing an appropriate script (i.e. lsm-register.sh (on Linux/Mac) or lsm-register.bat (on Windows)). This script takes as argument the metadata file name. After this, the corresponding metadata in RDF will have been stored in LSM. An example is illustrated in the following table:

```
./lsm-register.sh virtual-
sensors/brussles_weather.metadata
lsm-register.bat virtual-
sensors\brussels_weather.metadata
```

5.5.4 Pushing Data to LSM

In order to push data to LSM, the LSMExporter processing class is internally used by GSN/X-GSN. This is specified in the virtual sensor configuration file:

```
<processing-class>
      <class-name>org.openiot.gsn.vsensor.LSMExporter
</class-name>
      <init-params>
          <param name="allow-nulls">false</param>
          <param name="publish-to-lsm">true</param>
      </init-params>
      <output-structure>
          <field name="airtemperature" type="double" />
          <field name="humidity" type="double" />
          <field name="pressure" type="double" />
          <field name="windspeed" type="double" />
          <field name="windchill" type="double" />
          <field name="visibility" type="double" />
       </output-structure>
</processing-class>
```

Then, when X-GSN starts, it begins to acquire the data through the wrapper and automatically generating the RDF data for each observation, storing it in LSM.

Each observation will be assigned a unique URI, e.g.
<http://lsm.deri.ie/resource/29925179667811>

Then, you can query the Virtuoso server, to see the updated data, with the SPARQL query shown in the following table:

```
select * where {
  <http://lsm.deri.ie/resource/29925179667811> ?p ?o
}
```

and get the results shown in the following table:

http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://purl.oclc.org/NET/ssnx/ssn#Observation
http://purl.oclc.org/NET/ssnx/ssn#observedBy	http://lsm.deri.ie/resource/29855158254802
http://purl.oclc.org/NET/ssnx/ssn#observationResultTime	2013-05-15T11:45:00Z
http://purl.oclc.org/NET/ssnx/ssn#featureOfInterest	http://lsm.deri.ie/resource/3797289123726234

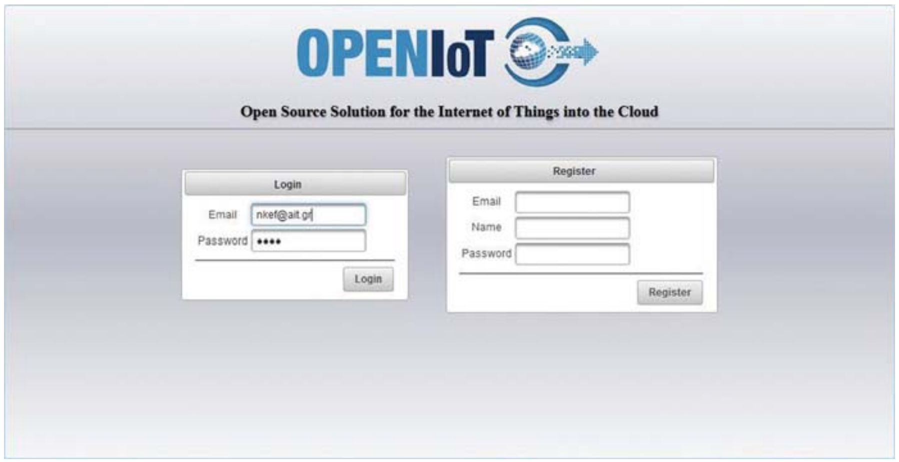
Once the data is in LSM, it can be accessed by the other OpenIoT components.

5.5.5 Service Definition and Deployment Using OpenIoT Tools

The first step, towards building a request for Sensing-as-a-Service, would be to log in to the Request Definition by using our credentials (Figure 5.10).

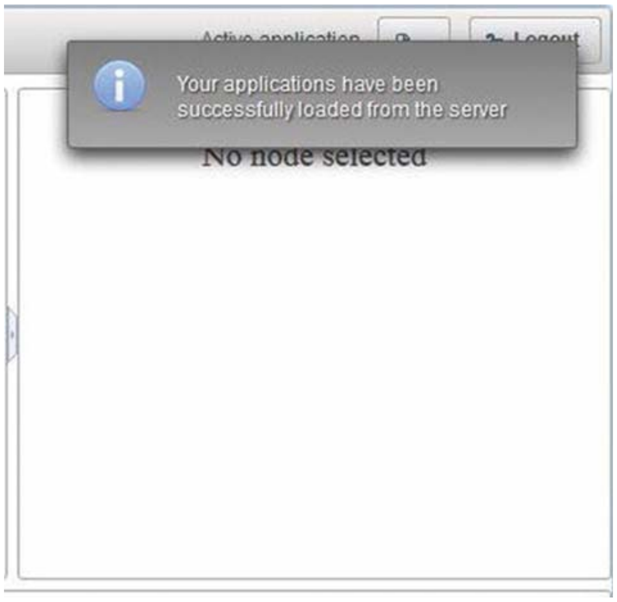
By logging in our profile is loaded and all our previously defined services are available to view or edit (Figure 5.11). A new Application can be created through the “File” menu (Figure 5.12).

As a first step, the available sensors should be discovered, using the magnifying glass at the data sources toolbox. In the map that appears we look up for the Brussels area and we add a pinpoint to the map. Then we set the radius of interest and we hit the “Find sensors” button (Figure 5.13 [9]).


OPENIoT
Open Source Solution for the Internet of Things into the Cloud
Login
Register
Email
Name
Password
Login
Register

Email: nkef@ait.gr
Password: ••••

Figure 5.10 Request definition log in.

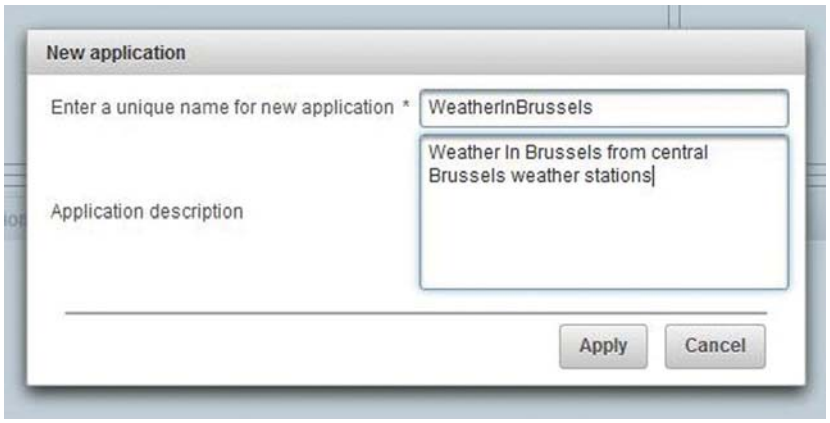

Active application

Logout

Your applications have been successfully loaded from the server

No node selected

Figure 5.11 Request definition loaded profile.

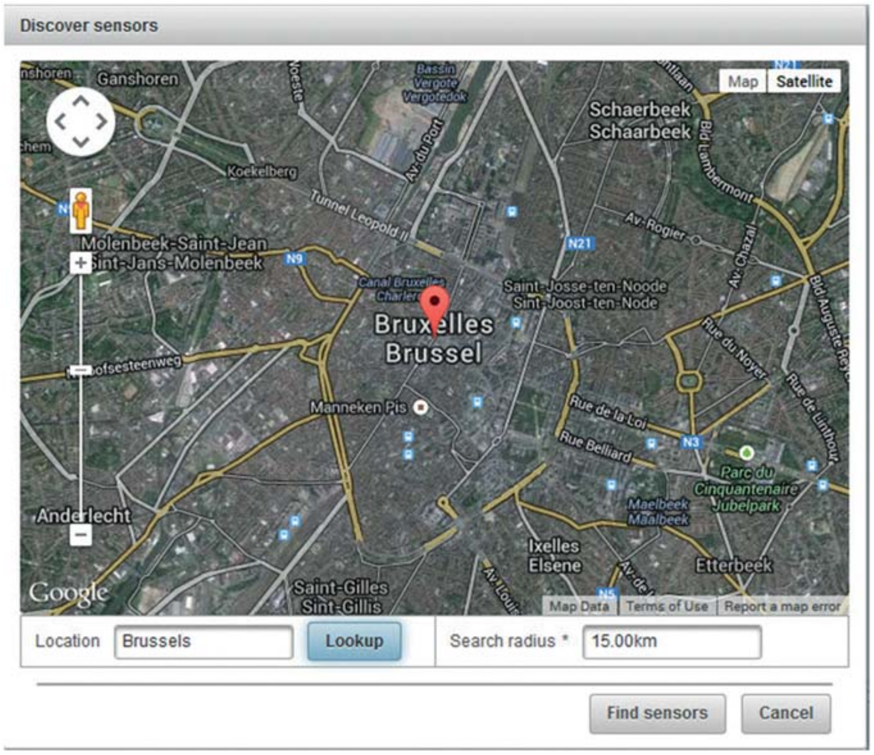

New application

Enter a unique name for new application * WeatherInBrussels

Application description: Weather In Brussels from central Brussels weather stations

Figure 5.12 New application creation.

This request is send to the Scheduler that in its turn queries LSM for available sensors in this area. The reported, from LSM, sensor types are sent to the Scheduler that in its turn sends to the Request Definition so as to fill the available “Data sources” toolbox (Figure 5.14). As we can see two sensor



Location: Brussels
Lookup
Search radius *: 15.00km

Figure 5.13 Sensor discovery in Brussels area.

types are deployed in that area (weather sensors and Integra Traceability Kiosk sensors).[2] By dragging and dropping the blocks from our toolbox we start to build our request. We drag and drop the "weather" sensor type and as we can see all the sensor type observations (outputs) are available to interact with (wind chill temperature, atmospheric pressure, air temperature, atmosphere humidity and wind speed).

A "Selection filter" from the "Filters & Groupers" toolbox is required. The one side of it is connected with the node and the other one with a "Between" comparator that has already been dropped to the workspace from the "Comparators" toolbox. We set up the "Between" comparator between "01/07/2014" and "01/28/2014" (three weeks) which are the dates of interest

[2]ITK is a multi-sensor device for track & trace applications in manufacturing and used in the scope of other OpenIoT applications.

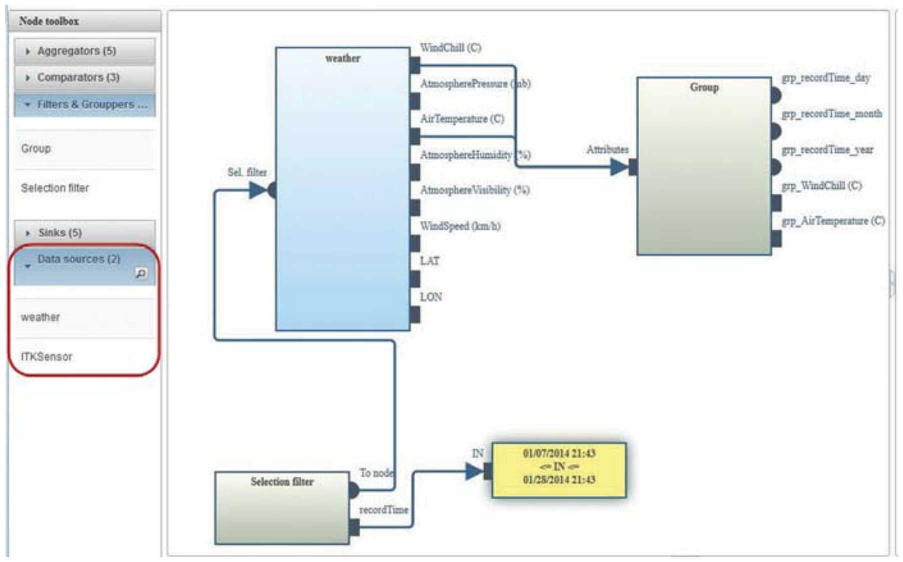


Figure 5.14 Comparator (between) properties.

to us to collect our data (Figure 5.14). The next step is to add a “Group” node from the “Filters & Groupers” toolbox which we are going to use so as to group the Wind Chill and Air Temperature by Year/Month/Day (see Figure 5.15 [9]) which is selected through the node’s options. The Wind chill and Air temperature outputs of the “weather” node are connected to the “Group” node attributes and automatically. As shown in Figure 5.16, these outputs are generated also to the “Group” node.

Since we need the average values for every day, we drag and drop two “Average” nodes from the “Aggregators” toolbox to the workspace and we connect the Wind Chill and Air Temperature outputs to them respectively (see Figure 5.16). The next step required in order to visualise the output (i.e. two average values for every day) to a line chart, is to drag and drop a “Line Chart” from the “Sinks” toolbox. The X axis presents the time and the Y axis presents/compares the temperature values. At the line chart properties, two series count are presented (in order to visualize two inputs) and for the X axis we select date observation as type. Hence, all the day/month/year outputs of the “Group” node are connected to “x1” and “x2” inputs of the “Line Chart” node respectively and the Wind Chill and Air Temperature outputs to “y1” and “y2” inputs respectively (Figure 5.16 [9]).

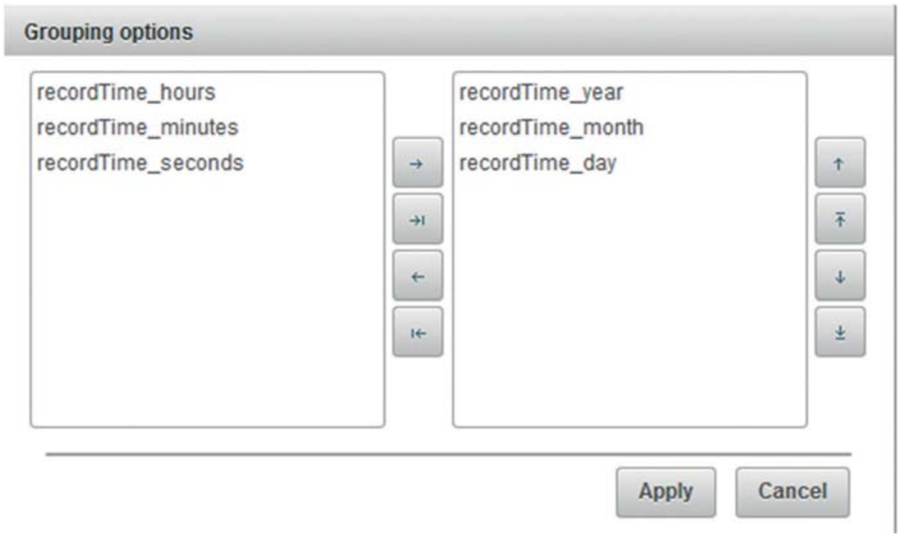


Figure 5.15 Grouping options.

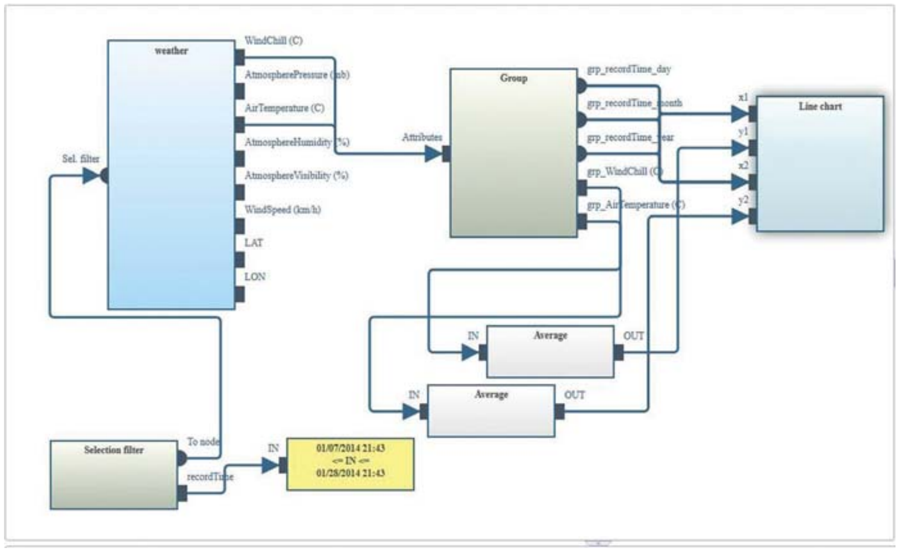


Figure 5.16 Line chart properties.

Following the visual definition of the service, the overall design can be validated using the “Validate design” option of the “Current application” menu (see Figure 5.17). This generates automatically the SPARQL scripts

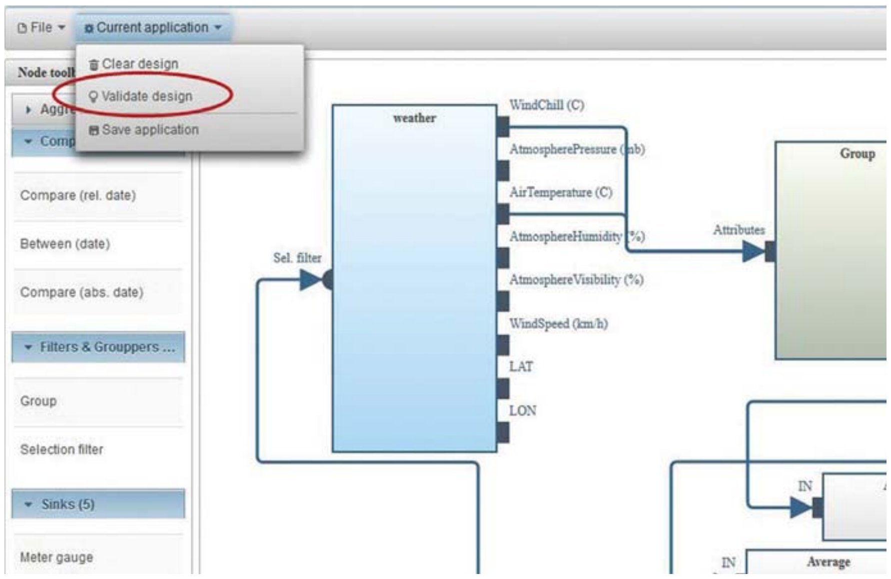


Figure 5.17 Validation of the service design.

that describe the graphical representation in our workspace. For every group of data that provides its output to a widget a different script is generated. In this specific example there is a need to visualize two different outputs (Wind Chill and Air Temperature) in one line chart and hence two scripts are generated (Figure 5.18 [9]).

For testing purposes these scripts could be taken and executed directly against the SPARQL interface of LSM (e.g., Figure 5.19).

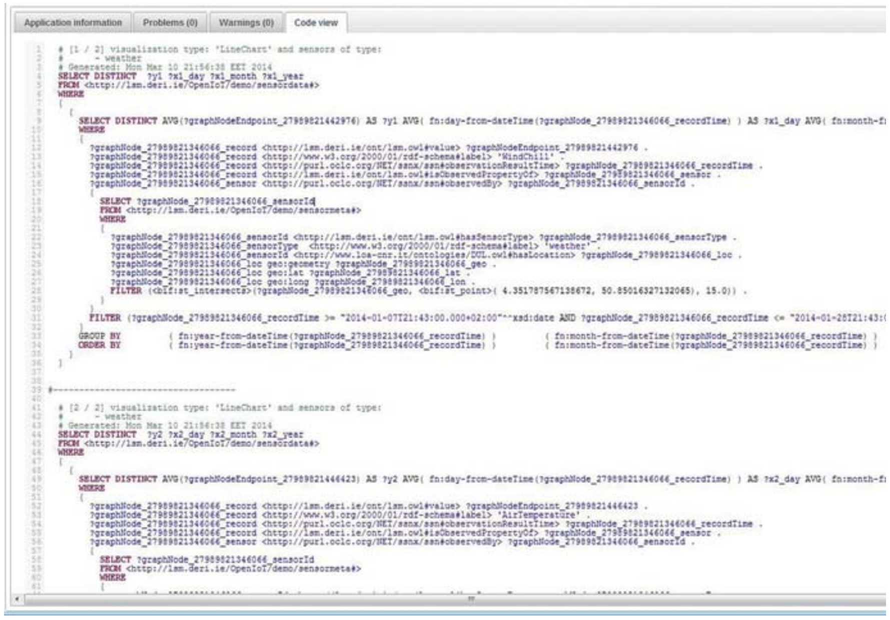
The Request Definition UI can also be used to save (register) the newly described Sensing-as-a-Service application to the Scheduler (Figure 5.20).

5.5.6 Visualizing the Request

In order to visualize the captured data, one has to log-in to the Request Presentation UI. Following this log-in the user profile is loaded and the user is able to view all the services registered under his account. The registered services are fetched from the SD&UM, which also builds the appropriate scripts to query this information from LSM (Figure 5.21).

Then we choose the application of interest to us (i.e. “WeatherInBrussels”) (Figure 5.22).

Accordingly, an empty widget associated with the selected application is presented. By using the “force dashboard refresh” option from the



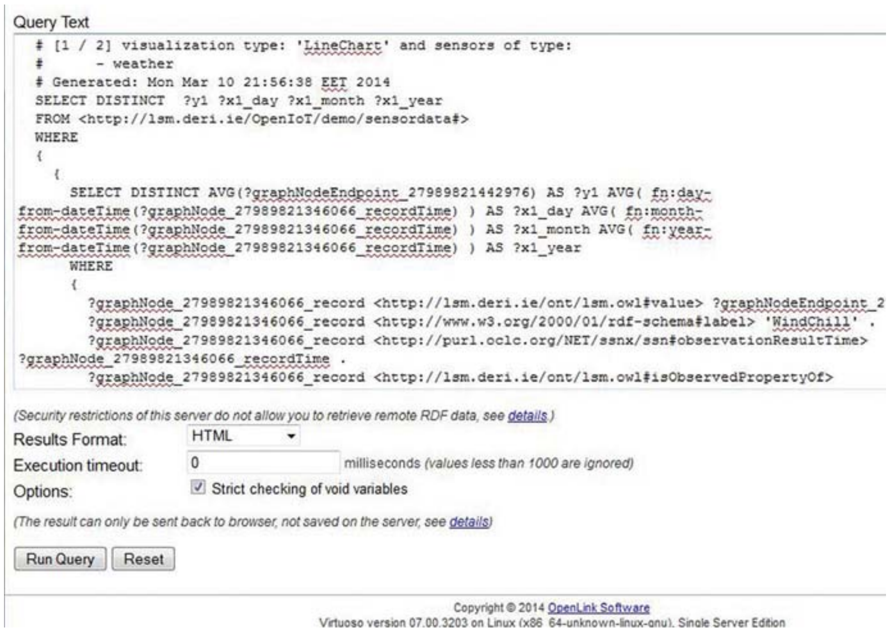
Application information | Problems (0) | Warnings (0) | Code view

```
# [1 / 2] visualisation type: 'LineChart' and sensors of type:
#       - weather
# Generated: Mon Mar 10 21:56:38 EET 2014
SELECT DISTINCT  ?y1 ?x1_day ?x1_month ?x1_year
FROM <http://lsm.deri.ie/OpenIoT/demo/sensordata#>
WHERE
{
  {
    SELECT DISTINCT AVG(?graphNodeEndpoint_27989821442976) AS ?y1 AVG( fn:day-from-dateTime(?graphNode_27989821346066_recordTime) ) AS ?x1_day AVG( fn:month-f
    WHERE
    {
      ?graphNode_27989821346066_record <http://lsm.deri.ie/ont/lsm.owl#value> ?graphNodeEndpoint_27989821442976 .
      ?graphNode_27989821346066_record <http://www.w3.org/2000/01/rdf-schema#label> 'WindChill' .
      ?graphNode_27989821346066_record <http://purl.oclc.org/NET/ssnx/ssn#observationResultTime> ?graphNode_27989821346066_recordTime .
      ?graphNode_27989821346066_record <http://lsm.deri.ie/ont/lsm.owl#isObservedPropertyOf> ?graphNode_27989821346066_sensor .
      ?graphNode_27989821346066_sensor <http://purl.oclc.org/NET/ssnx/ssn#observedBy> ?graphNode_27989821346066_sensorId .
      {
        SELECT ?graphNode_27989821346066_sensorId
        FROM <http://lsm.deri.ie/OpenIoT/demo/sensormeta#>
        WHERE
        {
          ?graphNode_27989821346066_sensorId <http://lsm.deri.ie/ont/lsm.owl#hasSensorType> ?graphNode_27989821346066_sensorType .
          ?graphNode_27989821346066_sensorType  <http://www.w3.org/2000/01/rdf-schema#label> 'weather' .
          ?graphNode_27989821346066_sensorId <http://www.loa-cnr.it/ontologies/DUL.owl#hasLocation> ?graphNode_27989821346066_loc .
          ?graphNode_27989821346066_loc geo:geometry ?graphNode_27989821346066_geo .
          ?graphNode_27989821346066_loc geo:lat ?graphNode_27989821346066_lat .
          ?graphNode_27989821346066_loc geo:long ?graphNode_27989821346066_lon .
         FILTER (<bif:st_intersects>(?graphNode_27989821346066_geo, <bif:st_point>( 4.351787567138672, 50.85016327132065), 15.0)) .
        }
      }
      FILTER (?graphNode_27989821346066_recordTime >= "2014-01-07T21:43:00.000+02:00"^^xsd:date AND ?graphNode_27989821346066_recordTime <= "2014-01-28T21:43:(
    }
    GROUP BY         ( fn:year-from-dateTime(?graphNode_27989821346066_recordTime) )         ( fn:month-from-dateTime(?graphNode_27989821346066_recordTime) )
    ORDER BY         ( fn:year-from-dateTime(?graphNode_27989821346066_recordTime) )         ( fn:month-from-dateTime(?graphNode_27989821346066_recordTime) )
  }
}


#---------------------------------

# [2 / 2] visualization type: 'LineChart' and sensors of type:
#       - weather
# Generated: Mon Mar 10 21:56:38 EET 2014
SELECT DISTINCT  ?y2 ?x2_day ?x2_month ?x2_year
FROM <http://lsm.deri.ie/OpenIoT/demo/sensordata#>
WHERE
{
  {
    SELECT DISTINCT AVG(?graphNodeEndpoint_27989821446423) AS ?y2 AVG( fn:day-from-dateTime(?graphNode_27989821346066_recordTime) ) AS ?x2_day AVG( fn:month-f
    WHERE
    {
      ?graphNode_27989821346066_record <http://lsm.deri.ie/ont/lsm.owl#value> ?graphNodeEndpoint_27989821446423 .
      ?graphNode_27989821346066_record <http://www.w3.org/2000/01/rdf-schema#label> 'AirTemperature' .
      ?graphNode_27989821346066_record <http://purl.oclc.org/NET/ssnx/ssn#observationResultTime> ?graphNode_27989821346066_recordTime .
      ?graphNode_27989821346066_record <http://lsm.deri.ie/ont/lsm.owl#isObservedPropertyOf> ?graphNode_27989821346066_sensor .
      ?graphNode_27989821346066_sensor <http://purl.oclc.org/NET/ssnx/ssn#observedBy> ?graphNode_27989821346066_sensorId .
      {
        SELECT ?graphNode_27989821346066_sensorId
        FROM <http://lsm.deri.ie/OpenIoT/demo/sensormeta#>
        WHERE
        {
```

Figure 5.18 SPARQL script generation.



Query Text

```
  # [1 / 2] visualization type: 'LineChart' and sensors of type:
  #      - weather
  # Generated: Mon Mar 10 21:56:38 EET 2014
  SELECT DISTINCT  ?y1 ?x1_day ?x1_month ?x1_year
  FROM <http://lsm.deri.ie/OpenIoT/demo/sensordata#>
  WHERE
  {
    {
      SELECT DISTINCT AVG(?graphNodeEndpoint_27989821442976) AS ?y1 AVG( fn:day-
from-dateTime(?graphNode_27989821346066_recordTime) ) AS ?x1_day AVG( fn:month-
from-dateTime(?graphNode_27989821346066_recordTime) ) AS ?x1_month AVG( fn:year-
from-dateTime(?graphNode_27989821346066_recordTime) ) AS ?x1_year
      WHERE
      {
        ?graphNode_27989821346066_record <http://lsm.deri.ie/ont/lsm.owl#value> ?graphNodeEndpoint_2
        ?graphNode_27989821346066_record <http://www.w3.org/2000/01/rdf-schema#label> 'WindChill' .
        ?graphNode_27989821346066_record <http://purl.oclc.org/NET/ssnx/ssn#observationResultTime>
?graphNode_27989821346066_recordTime .
        ?graphNode_27989821346066_record <http://lsm.deri.ie/ont/lsm.owl#isObservedPropertyOf>
```

(Security restrictions of this server do not allow you to retrieve remote RDF data, see details.)

Results Format: HTML

Execution timeout: 0 milliseconds (values less than 1000 are ignored)

Options: ☑ Strict checking of void variables

(The result can only be sent back to browser, not saved on the server, see details)

Copyright © 2014 OpenLink Software
Virtuoso version 07.00.3203 on Linux (x86_64-unknown-linux-gnu), Single Server Edition

Figure 5.19 LSM SPARQL endpoint (2 weeks wind chill in Brussels).

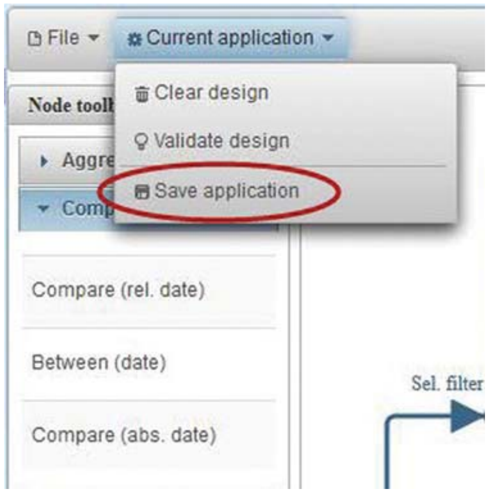
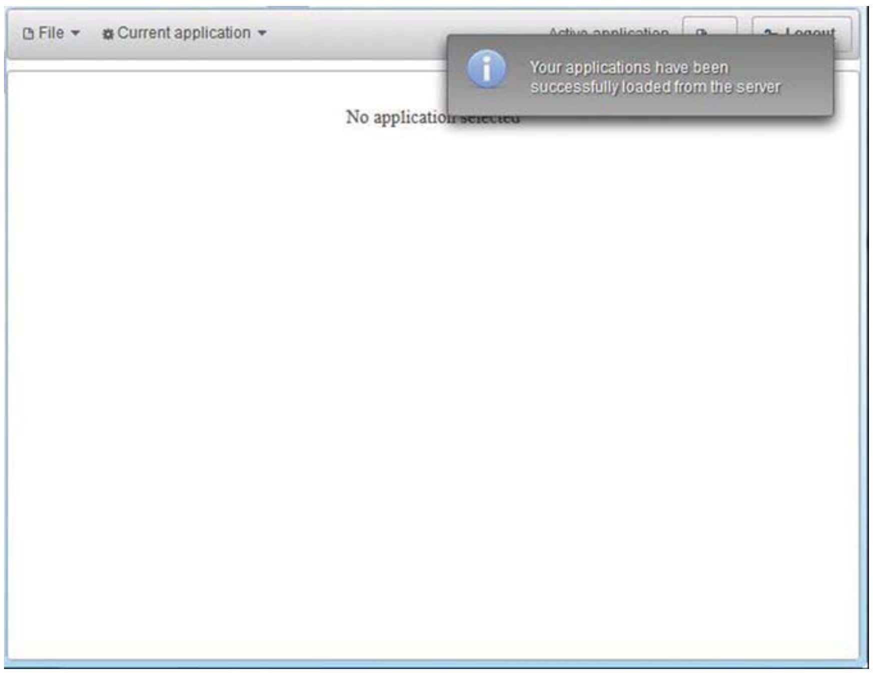


Figure 5.20 Save application button.



Your applications have been successfully loaded from the server

Figure 5.21 Request presentation loaded profile.

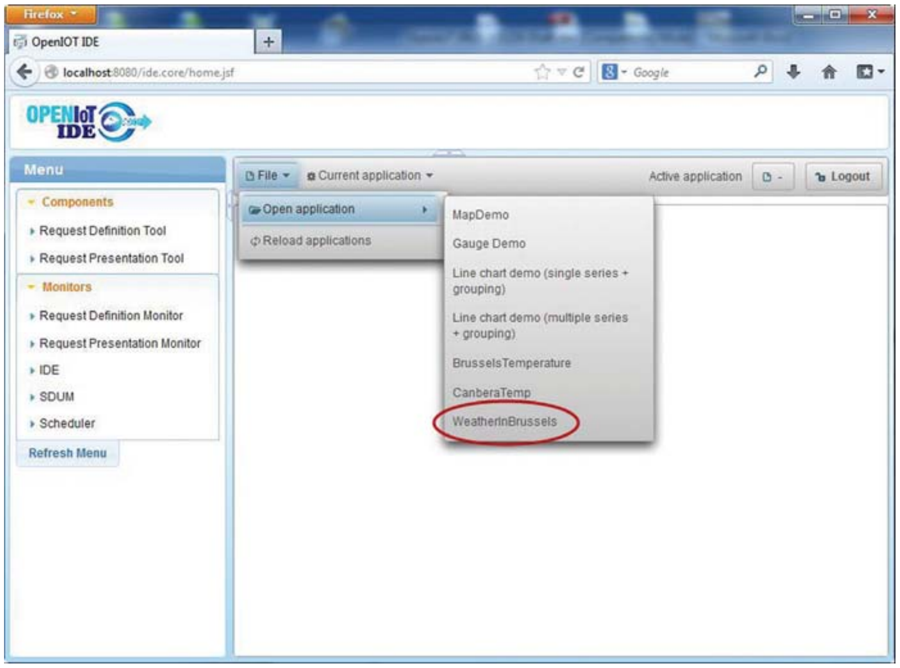


Figure 5.22 Load "WeatherInBrussels" scenario.

"Current application" menu, the Request Presentation exploits the "pollForReport (serviceID: String): SdumServiceResultSet" rest service of the SD&UM. This SD&UM service retrieves the previously registered application from the LSM module, retrieves the involved SPARQL scripts, executes them against the LSM SPARQL interface, analyses the results, builds a list of the results and how to present them to the widget and finally sends these data to the Request Presentation module where the result is visualized (Figure 5.23 [9]). The result is a filtered result set from the initially raw data stored to the database every 4 hours of the average Wind Chill temperature versus average Air temperature in Brussels for the specified time interval.

5.6 From Sensing-as-a-Service to IoT-Analytics-as-a-Service

Earlier paragraphs have illustrated the Sensing-as-a-Service paradigm, along with its practical implementation based on the OpenIoT open source project

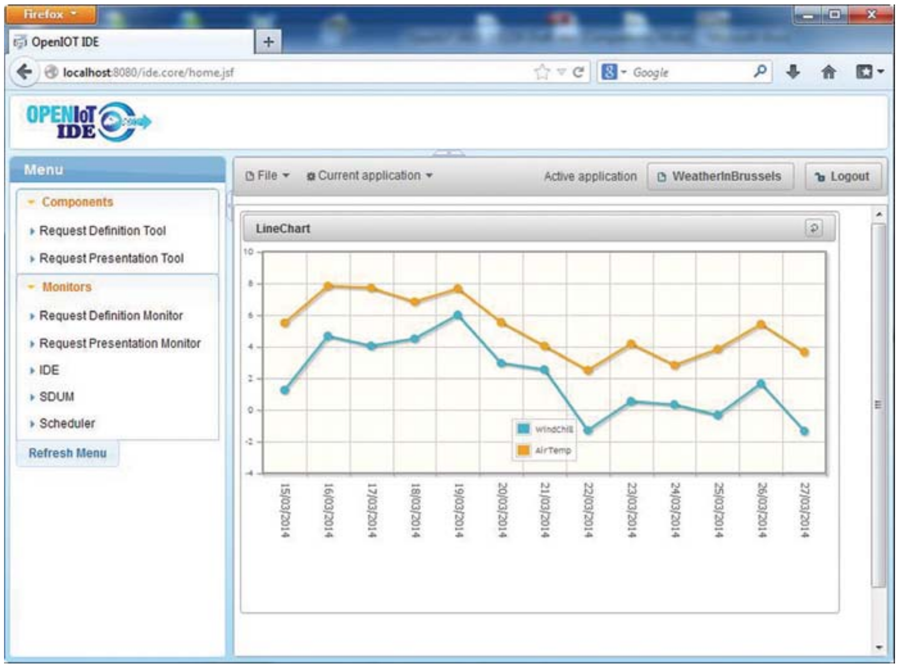


Figure 5.23 Wind chill vs. air temperature in Brussels line chart.

and the tools that it provides. The Sensing-as-a-Service paradigm as implemented by OpenIoT involves:

- Dynamic selection of (virtual) sensors from the set of sensors that are registered with the (RDF-based) directory services. This selection of sensors is empowered by the semantic unification of diverse data streams, which is supported by OpenIoT on the basis of the semantic annotation of virtual sensors and their observations.
- Definition of IoT data processing functions over the selected IoT data sources based on functionalities that can be expressed as SPARQL queries. Note that SPARQL does not enable the definition and execution of sophisticated data analytics functions. Rather, it is limited to supporting simple statistical processing functionalities such as the calculation of sums, averages and variances over observations provided by the selected virtual sensors and/or groups of virtual sensors.

Hence, the introduced Sensing-as-a-Service functionalities do not provide the means for non-trivial data analytics based on data mining and machine learning schemes. Nevertheless, the extension of infrastructures like OpenIoT with analytics functionalities is straightforward. In particular, such an extension entails the following steps:

- Integrating an analytics framework (such as the R project) in order to support the execution of machine learning functionalities.
- Implementing a data pre-processing (i.e. data preparation) layer, aiming at transforming the IoT data streams from the OpenIoT cloud to a format compatible with the analytics framework (e.g., R).
- Enhancing the concepts of the ontology in order to support additional devices, data streams and data analytics properties in a way that ensures the semantic unification of the various data streams to be produced prior to their integration to the analytics framework.

These steps provide a sound basis for advancing a Sensing-as-a-Service infrastructure to the IoT Analytics as a Service one. However, additional enhancements can be also implemented in order to ensure more scalable and high performance processing, through for example considering data storage, network latency and processing performance factors.

5.7 Conclusions

This chapter has focused on a special case of IoT/cloud integration, which entails the dynamic selection of sensors and the processing of their data towards a Sensing-as-a-Service paradigm. In addition to introducing the main principles of Sensing-as-a-Service, the chapter has also presented the practical aspects of this paradigm, based on a award-winning OpenIoT open source project. The latter provides technology and ease-to-use (visual) tools that enable the dynamic selection of virtual sensors from a cloud infrastructure and the subsequent processing of their data on the basis of functionalities that are provided by the SPARQL query language. The use of SPARQL as a data processing utility is enabled due to the semantic unification of the various IoT data streams, regardless of the (virtual) sensor that provides them. To this end, all IoT data streams are semantically annotated in order to comply with the same ontology. Overall, the OpenIoT project can be seen as a blueprint for implementing similar Sensing-as-a-Service systems.

The Sensing-as-a-Service paradigm can be also seen as a foundation for the implementation of IoT-Analytics-as-a-Service, through integrating more sophisticated data analytics capabilities over baseline Sensing-as-a-Service infrastructures. The latter provide a sound basis for IoT-Analytics-as-a-Service, since they offer the ever important data collection and semantic unification parts. We can expect a rise of IoT-Analytics-as-a-Service infrastructures in the near future, as enterprises are likely to seek opportunities for outsourcing complex the IoT analytics tasks to a cloud provider.

Acknowledgements

Part of this work has been carried out in the scope of the OpenIoT project (openiot.eu), which has been co-funded by the European Commission in the scope of the FP7 framework programme (contract No. 287305).

References

[1] Christian Vecchiola, Rajkumar Buyya, S. Thamarai Selvi. *Mastering Cloud Computing*. 1st Edition, Foundations and Applications Programming, Elsevier Print ISBN 9780124114548 Electronic ISBN 9780124095397.

[2] John Soldatos, M. Serrano and M. Hauswirth. *Convergence of Utility Computing with the Internet-of-Things*, International Workshop on Extending Seamlessly to the Internet of Things (esIoT), collocated at the IMIS-2012 International Conference, 4th–6th July, 2012, Palermo, Italy.

[3] John Soldatos, Nikos Kefalakis, et. al. *OpenIoT: Open Source Internet-of-Things in the Cloud*. Lecture Notes in Computer Science, invited paper, vol. 9001, (2015).

[4] Martin Serrano, John Soldatos. *IoT is More Than Just Connecting Devices: The OpenIoT Stack Explained*, IEEE Internet of Things Newsletter, September 8th, 2015.

[5] Martin Serrano, Hoan Nguyen Mau Quoc, Danh Le Phuoc, Manfred Hauswirth, John Soldatos, Nikos Kefalakis, Prem Prakash Jayaraman, Arkady B. Zaslavsky. *Defining the Stack for Service Delivery Models and Interoperability in the Internet of Things: A Practical Case With OpenIoT-VDK*. IEEE Journal on Selected Areas in Communications 33(4): 676–689 (2015).

[6] Karl Aberer, Manfred Hauswirth, Ali Salehi. *Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks*. MDM 2007: 198–205.

[7] Martin Serrano. *Applied Ontology Engineering in Cloud Services, Networks and Management Systems*. Springer publishers, March 2012. Hardcover, pp. 222 pages, ISBN-10: 1461422353, ISBN-13:978-1461422358.

[8] N. Kefalakis, S. Petris, C. Georgoulis, J. Soldatos. Open Source semantic web infrastructure for managing IoT resources in the Cloud. Book Chapter "Internet of Things: Principles and Paradigms", Elsevier Science, 2016, ISBN 978-0-12-809347-4.

[9] N. Kefalakis, J. Soldatos, A. Anagnostopoulos, and P. Dimitropoulos. A Visual Paradigm for IoT Solutions Development *in Interoperability and Open-Source Solutions for the Internet of Things*, Springer, 2015, pp. 26–45.

[10] J. Soldatos, N. Kefalakis, M. Serrano, and M. Hauswirth. Design principles for utility-driven services and cloud-based computing modelling for the Internet of Things. *Int. J. Web Grid Serv*., vol. 10, no. 2, pp. 139–167, 2014.