energies

Article

Development of Middleware Applied to Microgrids by Means of an Open Source Enterprise Service Bus

Jesús Rodríguez-Molina *, José-Fernán Martínez, Pedro Castillejo and Gregorio Rubio

Research Center on Software Technologies and Multimedia Systems for Sustainability (CITSEM) Campus Sur UPM, Ctra. Valencia, Km 7, 28031 Madrid, Spain; jf.martinez@upm.es (J.-F.M.); pedro.castillejo@upm.es (P.C.); gregorio.rubio@upm.es (G.R.)

* Correspondence: jesus.rodriguezm@upm.es; Tel.: +34-914-524-900 (ext. 20794)

Academic Editor: Paras Mandal
Received: 30 September 2016; Accepted: 29 January 2017; Published: 10 February 2017

Abstract: The success of the smart grid relies heavily on the integration of Distributed Energy Resources (DERs) and interoperability among the hardware elements that are present as part of either the smart grid itself or in a smaller size deployment, such as a microgrid. Therefore, establishing an accurate design for software architectures that guarantee interoperability and are able to abstract hardware heterogeneity in this application domain, along with a clearly defined procedure on how to implement and test a solution like this, becomes a desirable objective. This paper describes the requirements needed to design a secure, decentralized and semantic middleware architecture for microgrids and the procedures used to develop it, so that the mandatory software components that have to be encased by the solution, as well as the steps that should be followed to make it happen, become clear for any designer, software architect or programmer that has to tackle similar challenges. In order to demonstrate the usability of the ideas put forward here, two successful pilots where middleware solutions were created according to these principles have been described.

Keywords: distributed system; Enterprise Service Bus (ESB); microgrid; middleware

1. Introduction

The smart grid adoption is growing in a large number of countries, with the role of the different actors involved changing at a fast pace [1]. In a few years, the infrastructures bringing power both to industrial and residential customers will be enhanced with a plethora of new services providing intelligence to the network. The benefits of these new services will be used both by operators (in order to balance the generation and demand) and customers (so as to foresee the variation of power prices and configure the consumption). Moreover, consumers will be able to play a dual role where they will not only consume power from the grid but also, and due to the adoption of equipment required to use Renewable Energy Sources (solar panels, micro-wind generators, etc.), consumers will be able to generate, store and sell energy to the grid operator that can use this incoming power to balance the power flow in their network. In order to provide those services, operators need to develop a large number of solutions (in form of software components) that must communicate and coordinate among them. These components can be installed centrally or distributed among different operator servers. Distributing those components among hardware geographically separated enhances not only security (some countries consider the power network as a critical infrastructure to be protected against spurious attacks) but also leads to a balanced load for the server (as the smart grid operators could potentially serve millions of customers).

To coordinate the functionalities of the services, a distributed software integration solution appears as one of the possibilities. This solution can be referred to as middleware. The need for middleware architectures was established years ago when the rising demand of interconnectivity solutions started to snowball beyond the possibilities that could be offered by the existing software architectures. It is by enabling interoperability among hardware devices and software modules of different nature that distributed systems—as the smart grid or a microgrid de facto are—and networks of equipment can be extended and improved to their greatest capabilities. The reasons for having middleware as an advisable addition to the smart grid, how to develop it and the required installation procedures are the main topics of this manuscript.

1.1. Background of Middleware

Several decades ago it became clear that a solution for device interoperability was required to have different pieces of equipment cooperating in a holistic, seamless manner in a distributed fashion rather than having them depending on the computational capabilities that only centralized mainframes could offer. By having a system with these features it was possible to decentralize the different decisions and resources that could be used by the available equipment so that each of the parts could have more autonomy and, in the end, the deployment of several machines could be easier to hierarchize and extend in the future. However, in order to have such a system, it was necessary to use some kind of interoperating technology that included network interconnectivity, resource assignment or operations management. Basically, the need for a distributed system, conceived as "one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages" [2] was established. Thus, several developments were made with the aim of tackling the issues and challenges related to device interconnectivity, scalability or interoperability.

Whereas the progress done in networks communication, aided both by the constant improvement of physical transmission technologies and the standardization works done in this area of knowledge (either by means of the concepts put forward by the Open Systems Interconnection (OSI) architecture [3], or the applications carried out by the 4-layered, widely extended Transmission Control Protocol/Internet Protocol (TCP/IP) architecture [4] solved most of the issues that had been presented, significant challenges still exist when data formatting or higher level information transfers have to be made. These challenges involve four different ideas:

- *Interoperability among pieces of equipment.* In a distributed system-based deployment, it is likely that there will be different devices working with each other, such as new appliances sending and receiving data from legacy ones, high-capability pieces of equipment subscribing to services published by low capability ones, etc.
- *State-of-the-art in distributed systems*. While there are still many distributed systems that, overall, have not significantly changed during the last years (being the World Wide Web the most prominent one [5]), some others that include devices that were not manufactured expecting them to be enhanced with Information and Communication Technologies (for example, there are some concepts related to the deployment of power grids that date back to a hundred years [6]) have to be taken into account now.
- *Data complexity*. For the good and the bad, the information that can be transferred from one part of the system to another is likely to be of a more complex nature than before. Data can be refined by means of semantic capabilities (ontologies) to include them as a part of a bigger collection of concepts that relate among them.
- *Offered services*. The services that can be obtained from the system have increased in number and complexity at roughly the same pace as the capabilities of the involved equipment and communication infrastructure have. This is a case especially intense in the smart grid, that is capable of offering facilities specific to energy-related characteristics (demand side management, demand response, optimal power flow), as well as applications closely linked to its distributed nature (devices registered in a microgrid, data fetched from requests).

Fortunately, most of these issues can be solved by developing a software intermediation layer placed between the application level and the components linked to the hardware equipment, such as firmware or operating systems. This layer is the middleware, and it is used to abstract the heterogeneity of the hardware present in a distributed system while offering a set of facilities with homogeneous appearance (usually as Application Programming Interfaces (APIs)) to the application layer. As it can be seen in Figure 1, middleware gives the illusion of having a single machine operating with a single kind of device, even though reality might be very different.

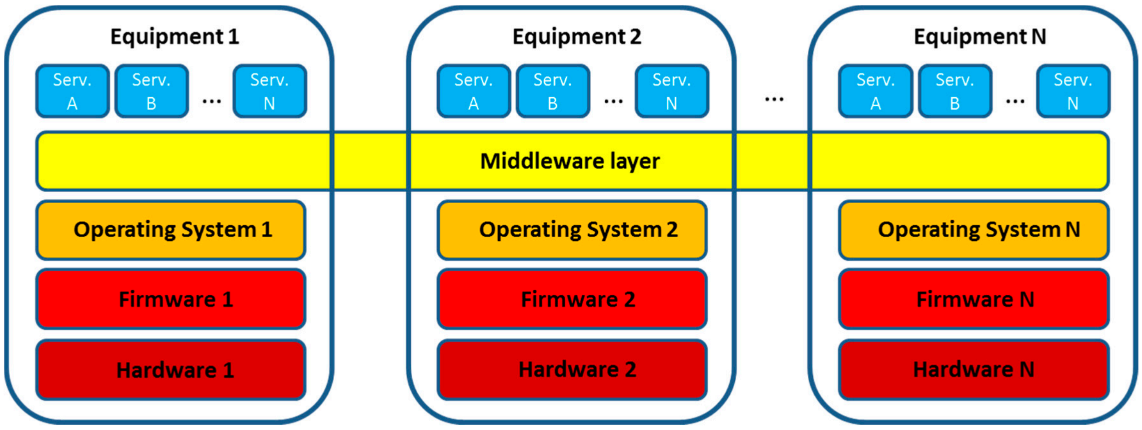

Figure 1. Existing middleware architecture among hardware devices.

As far as the smart grid or micrograds are concerned, the needs of interoperability, service delivery and hardware abstraction become obvious when having to face entities such as Advanced Metering Infrastructure (AMI), Supervisory Control and Data Acquisitions (SCADAs), Remote Terminal Units (RTUs) or Phasor Measurement Units (PMUs). What is more, the potential services to be enjoyed must be offered in a secure way, so that they will be utilized with the least amount possible of issues related to security. Arguably, if a system that is going to be used is not secure, then its usage will become discouraged and will be discarded. The services that are going to be offered are also determined by the features of the pieces of equipment installed in a microgrid that is part of a larger smart grid. A display of these ideas can be seen in Table 1 as a way to see how generic challenges are dealt with.

Table 1. Specific challenges for middleware in the smart grid.

Generic Challenges	Smart Grid Challenges
Devices registration	Registration of PMUs, RTUs, AMIs, etc.
Hardware abstraction	Abstraction of PMUs, RTUs, AMIs, etc.
Distributed service delivery	Delivery of demand response, demand side management, optimal power flow, etc.
System scalability	Usage of ontologies, semantic features.

It must be mentioned at this point that, by microgrid, we mean a set of distributed networks of resources generating electricity consumed by electric loads encased in a larger smart grid, with the latter covering a wider area and supervising the microgrid. In this way, a micogrid uses a point of common coupling to connect itself to the larger macrogrid, regarded as the distributed infrastructure interconnecting the microgrids [7]. Plus, microgrids are deemed as limited in power and size when compared to macrogrids, due to the fact that the latter typically mount distributed energy resources (DERs) as small photovoltaic panels (1–10 kW) and microturbines (25–100 kW) [8]. However, a microgrid is still larger than a nanogrid, which can be described as "a single domain for voltage, reliability, and administration. It must have at least one load (sink of power, which could be storage) and at least one gateway to the outside. Electricity storage may or may not be present" [9]. When all is said and done, it is up to the middleware architecture to accomplish the integration of new devices in order to have them working in a seamless manner with the legacy ones, as well as deliver the expected services and guarantee the scalability of the system with specific mechanisms.

1.2. Overall Framework

There are several roles that are assumed by the agents cooperating in power production that usually remain the same despite having a power grid with a higher or lower degree of intelligence. Likewise, there are also other kinds of facilities strongly related to perform trading operations with the electricity that is present in the power grid. For instance, the European Power Exchange SE (EPEXSPOT, [10]) that is made up of power markets such as the European Energy Exchange (EEX, [11]) are conglomerates of European energy exchanges where power and other raw materials used to generate electricity are interchanged. Their impact in building the smart grid must not be neglected, as electricity trade will become an option for a significant amount of small electricity producers (that is to say, the ones that have DERs available) that were previously excluded from those markets [12]. Furthermore, entities that require large amounts of investments in the power grid (such as DSOs or TSOs) may choose to become incorporated and issue shares to be bought and sold in the stock markets. All these entities can be regarded as the traditional main stakeholders of the system that is described here. When the power grid becomes enhanced with contributions by the ICTs that are included, and they provide a positive impact on the services that can be obtained from the overall development, it can be said that the power grid became a smart grid. The most typical actors holding a relevant position in the latter are:

- *Power generator*. This is the place where the electric power is generated. Typically, it will be a large facility that, by processing raw materials (coal, oil, uranium, potential energy from waterfalls, etc.) offers electricity as output.
- *Transmission System Operator (TSO)*. This is the agent that adapts the produced electricity to the suitable parameters of voltage and intensity required for its transport. TSOs usually deal with the cabling and infrastructure required by the power grid to transmit the produced electricity from one location to another.
- *Distribution System Operator (DSO)*. This is the actor responsible for the management and delivery of the energy that is produced in the system. However, in practice, it is not unusual that the DSO owns the power generation means, except when end users produce their own energy (and therefore become "prosumers", as explained below).
- *Aggregator*. This agent has information regarding the energy that has been generated by the end users (that will be turned into "prosumers"), the energy that is consumed by them and the very electricity they generate. The aim of the aggregator is using both the electricity and the data to manage groups of clients (aggregator clusters) to sell them the energy, requesting the one provided by the DSO whenever it is needed and managing the clusters as if they were part of a microgrid.
- "*Prosumers*". Traditionally, end users were limited to consume the electricity that they demanded according to the terms they had signed their power supply contracts. This procedure was unidirectional and could not be changed. However, since Renewable Energy Sources have become increasingly easy to process and handle, and the means used for their exploitation have dramatically decreased their acquisition and maintenance costs, former users have started adding them to their own homes and facilities, thus being able to produce and consume electricity (hence the term "prosumer"). The integration of the power that is produced by them is one of the main features of the smart grid, which has to guarantee that a bidirectional power flow can be offered for prosumers, as they must be able to receive power supply from the infrastructure provided by the TSO and used by the Aggregator too, as well as provide their own produced energy to the power grid, with major implications in arranged tariffs and the business of electricity production and transfer under a more general perspective [13]. Prosumers can be expected to work different ways depending on the role they wish to adopt. For instance, they may choose to use either part or their whole infrastructure to generate electricity, even including Vehicle-to-Grid capabilities [14] or be included as any other electricity generation source in Virtual Power Plants [15]. Prosumers

and aggregators are the new prominent stakeholders that will play an important role in any kind of deployment done over a microgrid or the smart grid when they can connect and interoperate.

In spite of having all these different agents in the electricity production, distribution and consumption value chain, from a purely software perspective, middleware is oblivious to the differences between the different actors present in the power grid and the roles they play. However, the variations among the pieces of equipment involved in each of the stages of electricity production and consumption have as a consequence that middleware adaptations will be very different from one location to the other, so implementation works will usually be impacted by the heterogeneity and complexity of the equipment that is going to be abstracted. Middleware architectures are expected to work as they would do in any other distributed system and performing the same already foreseen functionalities (hardware abstraction, device interoperability, distributed service delivery, etc.), as well as the ones expected to be included in a microgrid.

1.3. Contributions

This paper includes a description of the recommended procedures to be performed in order to design a middleware solution tailored for microgrids, providing a secure framework to develop the forthcoming applications. Furthermore, the following contributions are offered:

- Description of the tools used for an open source middleware implementation for the smart grid. Its usage guarantees that any development can be easily replicated in different environments.
- Depiction of securitization works. The steps taken to make the middleware architecture a secure one are displayed as well.
- Description of all the procedures used for middleware implementation for the smart grid. They are shown with a high level of detail, paying attention to all the troubleshooting that might be required during the installation and configuration processes for middleware implementation.
- Description of all the messages used for data interchange within the middleware architecture, as well as the ones used when transferring data from one hardware device to the other one implied in bidirectional communications.
- Thorough explanation of the testbed used to validate the middleware solution developed from the middleware architecture as an actual deployment capable of offering services to third parties implied in a microgrid.
- A design for a middleware architecture for the smart grid is put forward adding elements (distribution, semantics, etc.) of great innovative value.
- The implementation of that middleware architecture has led to a middleware solution that has been tested with success both in a laboratory and in a living lab environment as a way to guarantee the usability of a middleware solution for the smart grid.
- Context Awareness, Encryption, Hardware Abstraction and smart grid services are explained and offered as software components embedded in the architecture. In this way, they can be located in an intermediation layer that negates the differences present in the different devices deployed at the physical level. In addition to that, the elements that they should be made of are described as well.

All these contributions have been done taking into account what is expected from a middleware architecture, that is to say, abstracting heterogeneity and offering a development-friendly solution to the application, high level-based layers. Therefore, as it was shown in Figure 1, middleware is solely located between the local, hardware-based components of a smart grid-based deployment (regardless of whether they are located in a TSO, DSO or an Aggregator, since what matters in this case is how data are managed from lower levels rather than what entity is used to encase the middleware facilities) and the applications and interfaces accessed by human beings playing a role in the deployment: home dwellers, maintenance staff, application users, etc. Hardware abstraction is performed in the equipment where middleware is deployed. This solution is based on the development works done in (a) the e-GOTHAM [16] project, where the middleware solution is located in a Smart Meter and a Local Controller that could be regarded as part of the aggregator equipment and a Central Controller managing the local ones; and (b) I3RES [17], where middleware is installed as part of the software components present in a DSO or a third party that may be an aggregation company or a legislator. If those hardware elements are present in deployments of different dimensions like nano, micro or a full smart grid, its development procedure and the proposal displayed here would have little differences. When compared to other reviewed proposals, it becomes clear that, while there are many developments that successfully deal with most of the issues found the particular scenarios where they were deployed, they barely describe any information about several topics of major importance:

(1) *How the middleware solution was designed.* The procedures or the requirement analysis made in order to include the required services in the middleware architecture are usually not clear. Without that reasoning, it is hard to understand which software components should be included or what their functionalities should be.
(2) *Tools used for middleware implementation.* Whenever there are implementation works implied in a middleware solution for the smart grid or a microgrid, there are usually scarce data of how it has been carried out. While some proposals provide performance data or details about what programming languages were used, most of the information is missing.

In addition to that, the procedures that have been described here are heavily based on the directives that are put forward by the ISO/IEC/IEEE 42010 standard [18] regarding a software Architecture Description (AD). Specifically, the following aspects have been taken into account:

(1) The procedure of architecting, regarded by this standard as a process involving conceiving, defining, expressing, documenting and communicating tasks, is profusely described in the manuscript.
(2) A software architecture, an AD and an architecture framework have been provided as well, in the sense that fundamental concepts and properties of a system and a work product has been used to express an architecture.
(3) The architecture framework (considered as the conventions, principles and practices for the description of the architecture), view (the output resulting from expressing the architecture from the perspective of specific system concerns) and viewpoint establishing the conventions for construction, interpretation and usage of the architecture views) are thoroughly depicted in the manuscript.
(4) Since the context of the middleware solution, design and implementation have been defined, along with several figures illustrating software components and the relationships among them, it can be said that the environment and the model kind have been defined.

Furthermore, the solution that is put forward in this manuscript also takes into account features related to specific standards, such as OpenADR 2.0 (focused on Automated Demand Response, [19]) and ISO 50001 (bent on requirements for Energy Management Systems or EMSs, [20]). Among the features that have been taken into account regarding OpenADR 2.0, the following ones are the most important ones:

(1) Compatibility with high-level protocols. Hypertext Transfer Protocol (HTTP) and Extensible Messaging and Presence Protocol (XMPP) are mentioned as the protocols used for application-like layers in Demand Response. Uniform Resource Identifiers (URIs) working as service endpoints are described in the standard.
(2) The scope of the standard includes services like registration, reporting or availability that have been included in the proposal described by this manuscript. These services are also influenced by the OASIS EI Version 1.0 standard [21].

(3) Security is also a major feature of this standard. The solutions that have been put forward in this case are closely related to already tested and well-proven ones, such as Transport Layer Service (TLS) and Public Key Infrastructure (PKI) certificates. X.509v3 certificates are issued for security solutions when transferring information.

At the same time, the ISO 50001 standard deals with topics like optimizing the usage of energy-consuming assets, energy management best practices and improvements or prioritizing the development of new, energy-efficient technologies. Although the standard does not target performance assessment of energy usage ("This International Standard does not prescribe specific performance criteria with respect to energy", [22]), it still aims at energy management requirements, thus following a procedure (starting with the requirements) resembling the one described here. The authors of this manuscript have taken into account the contributions offered by this standard, and offer a lower level point of view regarding how energy management can be improved as a consequence of the integration of all the components present in a distributed system such as the smart grid.

1.4. Sections

This paper is divided in several parts structured with the aim of readability and understanding of the exposed concepts. An introduction has just been presented providing the context and contents of the paper. Section 2 includes a collection of relevant related works that have been included as a way to compare the principles proposed in this manuscript to the ones shown in other proposals. Section 3 presents the elements required in order to deploy a secure open source middleware architecture for the smart grid. Section 4 describes how the implementation works can be carried out. Section 5 reflects the results of the validation performed to test the framework previously presented. Finally, Section 6 includes the conclusions and future works.

2. Related Works

There are a number of works that to an extent deal with the challenges that developing a semantic middleware architecture presents for the smart grid. They have been included as a way to consider the approaches that have been taken by several authors regarding what middleware for microgrids should look like and how they have implemented functional solutions.

2.1. Device-Level Service-Oriented Middleware Platform

Sučić et al. [23] put forward their own ideas for middleware applied to microgrid applications using Distributed Energy Sources with sematic capabilities. According to their point of view, the vendor-independent International Standard IEC 61850 [24] is expected to offer a significant step towards integration of devices in the smart grid, while stating that there are still issues regarding support for dynamic and self-managed microgrid features. Nevertheless, one key innovation that is offered by this standard is the inclusion of semantic capabilities; indeed, standardized data semantics are used for field device control and management purposes. At the same time, a communication interface is also offered in a technology-neutral way, in order to make possible the separation between data semantics and data exchange services. IEC 61850 is especially useful for microgrid integration, as it respects the fact that the latter can work either inter-connected or in an island configuration. The authors strengthen their proposal by adding some other features; for example, Open Platforms Communications Unified Architecture (OPC UA) is used as a way to provide interoperability. It must be taken into account, though, that OPC UA is yet to be fully compliant with IEC 61850 due to the event-driven data exchange mechanism implemented, as data exchanges are done by using periodic request/response queries rather than by means of actual events. Other technology pivotal to the work presented by the authors is Devices Profile for Web Services (DPWSs), which according to the authors can be deemed as a specific profile for Web service protocols specifically conceived for resource-constrained devices. In the proposal, both OPC UA and DPWS are coexisting together in the microgrid middleware architecture. Finally, Abstract Communication Service Interfaces (ACSIs) are used as the set of rules and state machines provided by IEC 61850 to define data interchange characteristics. Manufacturing Message Specification (MMS) is the way that a standardized middleware mapping has been enabled by the authors. The overall architecture relies on mapping the ACSI service models that are provided by the standard to the services that are implemented in the hosting device that is making use of the Service Oriented Architecture. For example, the event management facilities that provide functionalities for reporting and logging, which are mapped to subscription and monitoring as the services hosted in the hosting device. Furthermore, OPC UA AddressPace components are used as a way to model information based on IEC 61850. Overall, this proposal acknowledges the importance of a collection of facilities, such as semantics, security or web services for the end users that are of major importance for any power grid that becomes enhanced with Information and Communication Technologies. These features have been born in mind for the design and procedures shown in this manuscript so that semantic features and security will be added in the design.

2.2. Smart Microgrid Monitoring Using Data Distribution Service over Internet Protocol Networks

Shi et al. [25] offered a middleware-based development of monitoring capabilities for a microgrid by means of an implementation of Data Distribution Service (DDS) running over the Internet Protocol. The authors stress the importance of integrating the distributed generation (DG) of energy, while at the same time mention how middleware is expected to provide APIs for developers to access both middleware and the underlying facilities. DDS is a software standard created by the Object Management Group (OMG) that focuses on providing services of data level transfers between entities of a distributed system [26]. Implementations are available either via open source (OpenSpliceDDS [27] or OpenDDS [28]) or under license (for instance, CoreDX is offered by Twin Oaks [29] and there is another DDS implementation from eProsima [30]). Among the advantages mentioned by the authors about using DDS, rapid prototyping for each of the required features (by means of defining data of interest via Interface Definition Language or IDL, or even other languages depending on the iteration, such as Data Definition Language or DDL for CoreDX) or providing middleware in an explicit manner are present. The authors describe how DDS can be used to build a communication architecture that will use a middleware solution to interconnect different facilities typical of a microgrid (solar panels, wind turbines, web servers, etc.). Other features worth mentioning are the existence of (a) a real-time database used to store all the data that is produced under the same pattern and is aided by a real-time service (Real-Time Connect or RTS); (b) a publish/subscribe software infrastructure used as the main communication mode and (c) a monitoring system working through a web server and a database. Whereas this proposal does not include the security and semantic capabilities that the former one has (in the end, it depends on what is decided to be programmed by means of either a the available implementations of DDS, such as OpenSpliceDDS, OpenDDS, the one offered by eProsima or CoreDX, so it is up to the staff responsible for one development adding the required components or not), the idea of having an API to access the middleware solution has been taken into account for the design and procedures shown in this manuscript.

2.3. Intelligent Agent Based Micro Grid Control

Kouluri and Pandey [31] offered a middleware solution bent on delivering microgrid control by means of intelligent agents. This system has been implemented with an open source tool based on a Java Agent Development framework (JADE) and is conceived as a way to complement the control capabilities that can be provided by a SCADA system. There are three different kinds of agents that have been defined in the proposal: control agent (used to monitor the system voltage and frequency with the purpose of detecting emergency conditions that will trigger corresponding actions), DERs agent (used to store DER information linked to the agent) and load agent (contains data about users and their loads). As far as the actual middleware is concerned, a communication middleware has been developed to cope with the inputs that are used in the model. The proposal has been tested in a testbed in MATLAB/Simulink, where a simplified distribution circuit has been modelled. Rather than the performance of the middleware solution that was implemented, the main purpose of this testbed is checking the performance of the microgrid during faulty conditions, as the Multi Agent System is supposed to isolate the required parts during an event of this nature. Results show that by having a user agent removing the non-critical loads, stability on the network can be maintained when a microgrid is disconnected from the main power grid. Overall, although the proposal presented by the authors of the manuscript comprehends more topics than a middleware solution for a microgrid, there are several features of this proposal that should be considered. One of the most interesting ones is the usage of open source solutions for development. From the point of view of the authors of this manuscript, that is an inexpensive way to solve many issues regarding design and development, so it has been included in the proposal described here.

2.4. Interpreted Domain-Specific Models to Manage Smart Microgrids

Allison et al. [32] put forward their own perspective of middleware solutions by using a design for adaptive middleware in domain-specific models. The authors make use of Domain-Specific Modelling Languages (DSMLs) as a way to guarantee the interoperability not only of the deployed hardware, but also among different deployments that may or may not belong to the same application domain. Another element of critical importance in the proposal is the existence of interpreted DSMLs or i-DSMLs, as they allow the direct execution of models using a specialized execution engine. It is mentioned in the proposal that a Domain-Specific Virtual Machine is one of the actual i-DSML execution engines, and how they can be used for the application domain of the microgrids. As a consequence of this closeness, a microgrid modelling language (MGridML) and a DSVM prototype for MGridML interpretation, referred to as microgrid virtual machine (MGridVM) have been developed. The proposal is strongly based on the microgrid concept that is put forward by CERTS that establishes different levels of priority on the loads, in case they have to be disconnected (or reconnected) from the distribution grid. On the one hand, MGridML can be described as a metamodel with schemas for control and data grid. It uses three different notations to represent models: one is XML-based (X-MGridML), another one is graphical (G-MGridML) and the third one is User Interface-based (UI-MGridML). On the other hand, MGridVM implies using a synthesis process that requires using a user-defined model or an event generated by the middleware layer included in the MGridVM as inputs; the expected outputs will be the control scripts to be executed by the middleware. The authors describe how DSVMs are divided into four different sublayers called user interface (user can specify models through this layer), synthesis engine (with the already described functionality of using current runtime and new models as inputs to generate control scripts), middleware (as mentioned before, responsible for executing the control scripts generated by the previous layer and coordinating the delivery of domain services) and broker (provides an API to interact with the middleware and other interfaces that communicate with the underlying platform). In addition to this, the authors describe in [33] a Communications Virtual Machine that makes use of a middleware (User-Centric Communication Middleware or UCM), along with a Microgrid Control Middleware (MCM). The latter interprets control scripts related to energy management functionalities related to energy services for microgrids, whereas the former one makes use of the previously mentioned four layers to communicate remote and local user applications. The idea of the proposal is that a generic middleware solution should readapt to every new environment where it is deployed, and the fact that it has been divided into four different layers reinforces the idea, as they tend to isolate specific functionalities in each of those levels for a better re-adaptation. Thus, the objective of having a middleware architecture that can be changed according to the needs of a location has also been considered in the proposal of this manuscript.

2.5. GridStat

This and the next solution have been regarded as one of the most appealing ones according to what was researched in [34]. Rather than providing a way to interchange messages throughout the different hardware elements present in a microgrid (servers, AMI, equipment belonging to the aggregator, etc.), a middleware architecture comprising several services that are used outside hardware, networking and the application layers has been developed. As far as GridStat is concerned [35], there are several key features presented by the authors of this proposal that are of major important for the implementation of a middleware solution. One is the existence of entities used to offer Quality of Service capabilities (QoS brokers). Another one is the existence of a middleware architecture divided in several layers (a common theme in the presented solutions), which are called planes. There are two planes in the proposal, where one is used for high-level management network adaptation and resource allocation (the management plane) whereas the other involves forwarding information from each source to the destinations that have been chosen by the management plane components (the data plane). The proposal works under a publish/subscribe paradigm, where publishers and subscribers may be located in different clusters of routers with the same parameters for resource management and cyber security (named clouds in the proposal). Tests have been performed to have an accurate idea of how the proposal behaves when latency and load scalability are studied; in both cases the results show that for baseline forwarding latency, increasing backbone status routers inside a cloud in a linear way will result in a linear-like latency increase. Multicast forwarding latency, on the other hand, registers a higher increase in latency for 31,000 and 41,000 update forwarding messages per second. According to the authors, this is due to a limitation in the java library class used for the experiments, where the garbage collector was extensively used. Overall, while this proposal lacks information about semantic capabilities and how they should be implemented, security is regarded as a service of major importance, so it enforces the impression of the authors of this manuscript that it must be provided as a service, either when transferring data or when accessing the middleware solution.

2.6. Self-Organizing Smart Grid Services

Awad and German described their own concepts for smart grid services that can be self-organized [36]. As happened previously, a multi-agent approach is mentioned in this proposal as a way to provide design, supervise and control features to the infrastructure level. The authors mention that a centralized approach is not suitable for the smart grid, as a deployment like that would present major issues regarding scalability and high latency resulting from having to communicate with a central entity. Interconnecting heterogeneous devices is also mentioned as an important task to carry out, hence the importance of having a middleware solution capable of performing that task. The proposal shown in this case, as it happened with GridStat, is divided in two main levels called Decision level and Infrastructure level. The Decision level provides the Infrastructure level with directives, semantic information-related data used for design, supervision and control purposes, whereas the Infrastructure level offers feedback data resulting from what has received previously from the Decision level. The middleware solution to be used in this proposal is encased in the infrastructure level, and includes processes like data routing, aggregation, replication and filtering. The key concept that can be inferred from the proposal that has been summarized here is that a middleware solution for a microgrid not only is a good way to interconnect devices of different capabilities, but also must be distributed in order to manage failures that may happen in any side of the system. In order to do so, context awareness can be included as part of the middleware solution, as it will be used to react to adverse environment conditions or underlying flaws of a deployment.

2.7. A Customer Energy Management Platform

This proposal [37] presents the particularity of having been mostly developed within a private company with a commercial interest in exploiting a distributed electricity-based user platform (KT

Corporation, Seongnam, Korea). Apart from that, it shows most of the features that have been depicted for all the other proposals that have been included in this study: the infrastructure built by KT is divided in several layers with different functionalities that go as follows: to begin with, a low-level platform is used to include several systems that will perform operations related to the most common functionalities of the smart grid (business supporting system, metering data management system, demand response management system and renewable energy management system). Secondly, an Integrated Database stores information related to several features of the deployment where this middleware proposal is included (usage data, customer info, metadata, etc.). Besides, there is a group of Customer Energy Management Systems that include, among others, a system for Home Energy Management. Lastly, there is a service interface used by the KT infrastructure to interact with the appliances that end customers may have at their disposal. Most of the ideas that are described in this proposal have already been introduced in the other ones, but the concept of having a middleware solution that can be used for profitability is also considered by the authors of this manuscript, as a middleware solution must be developed, tested and maintained with reasonable amounts of resources (namely, time and budget). Thus, our procedure will be focused on providing an accurate description of the necessary steps to have a middleware solution covering all the required issues at a reasonable pace and low cost.

2.8. Smart Middleware Device for Smart Grid Integration

Oliveira et al. [38] introduced in their proposal the idea of encasing middleware in a device specifically designed with that purpose. They claim that even though some efforts have been done to include some devices to legacy protocols such as the Modbus protocol, which can be regarded as a standard for industrial networks, specific gateways are still needed for communications. Their proposal focuses on providing a design and implementation of one of those gateways as the location for the middleware architecture, the so-called Smart Middleware Device (referred in the proposal as its Portuguese acronym, that is, DMI). This device is responsible for both managing data flow features as well as the software components that will be used for protocol translation. This proposal can be regarded as a middleware architecture, due to the fact that it has a collection of services as part of the software implemented. However, the fact that it is strongly interwoven with a hardware component has to be taken into account since, according to the vision of the authors of the proposal, middleware will be located only within the device. The services available can be classified in two different groups: the core components and the translators. The core components are the ones used for all the requirements that information has in a smart grid system: communication between elements and units used to gather information (for example, SCADAs), a repository to store answers done to queries and a management component used for operations like ordering queries, send answers that require translation or pushing a query to a SCADA. On the other hand, the components used for translation are focused Modbus, Distributed Network Protocol, version 3 (DNP3, [39]) and IEC61850. Overall, the proposal is very aware of the need to allow different protocols to operate in the same deployment with ease (hence providing software components used for translation among them when required), which is also considered in the procedures described in this manuscript. The main constrain of the proposal is that it has been conceived to be encased in one single device with very specific capabilities, so installing it in any other piece of hardware in a microgrid or in an alike deployment can be challenging.

2.9. Cyber-Secure Data and Control Cloud for Power Grids

Hoefling et al. [40] offered their own ideas regarding a secure middleware for the smart grid as they have presented in the project called Cyber-secure Data and Control Cloud for power grids (C-DAX, [41]). As in previous cases, their proposal focuses on solving the security issues present when exchanging data in the smart grid or, as the authors often refer to, Active Distribution Networks (ADNs). Aside from PMUs, Phasor Data Concentrators (PDCs) and Real-Time State Estimation (RTSE), facilities are also mentioned as beneficiaries of the proposal. PDCs are units that receive timestamped information with an up to 50 Hz data refreshment frequency that are time-aligned and aggregated from different PMUs. RTSE applications, on the hand, receive the aggregated data from the PDCs and feed the data in a mathematical model that estimates the current state of the grid. The information that is transferred is organized as topics, which are described as abstract representations of unidirectional information channels with a specific storage capability (hence being somewhat close to the ideas that are offered by Data Distribution Service-based implementations). NASPInet is also mentioned here, claiming that PMU measurement is enabled with this infrastructure. This proposal has essentially been conceived as a Message-Oriented Middleware, due to the fact that the main emphasis of the works done is put into guaranteeing that there will be a secure procedure to interchange messages in a distributed infrastructure, rather than encasing services related to semantic capabilities or offering an API to the higher levels. The data plane included in this middleware solution makes use of two elements called the Designated Nodes (DNs), used to provide access to the cloud used to implement the C-DAX proposal, and Data Brokers (DBs) used to either forward or store topic information. In addition to that, there is a Monitoring and Management System used to monitor and control the underlying system that is deployed. Among the different scenarios that are considered for the tests, failures in Data Brokers are taken into account. In addition to that, the proposal takes into account the resilience of the smart grid as a Cyber-Physical System, which is an important feature regarding services like context awareness. However, while the proposal puts a strong stress in deploying security in a distributed system as the smart grid, there are few other services that are not included in the proposal (such as semantic capabilities, context awareness, etc.

2.10. Building as a Service (BaaS)

Martin et al. [42] focused their proposal in the data interchanged in the application domain of the smart grids used for energy efficiency in buildings. The authors believe that middleware can be used as a software element to link several entities in a distributed system where building energy consumption is at the core of its interests. The proposal relies on concepts common in distributed systems (Service-Oriented Architecture, Event-Driven Architecture) and aims to enable interoperability among several networked entities: already existing Building Management Systems (BMSs), Data Warehouses where information is stored, existing Building Information Models (BIMs), existing Information and Communication Technologies infrastructure and other services related to assessment, prediction and optimal control services. The provided information shows that the proposal can be regarded as a middleware architecture due to the fact that is divided in layers that provide different functionalities. It must be taken into account, though, that only the Communication Logic Layer (CLL) is explicitly cited as the middleware in the proposal. As in other middleware architectures, there are three levels that have been defined for the proposal from a high-level architecture point of view: the Application Layer, the Communication Logic Layer and the Data Layer. The Application Layer contains: (a) the models required to build the thermal model; (b) the modules devoted to the required automation and control capabilities for a specific building and the technical building management and (c) the services kernel with the main functionalities of the level (event, service and signal handlers, configuration and simulation manager, module registry, time control and permissions). The communication logic layer is further subdivided in two sublayers: the Core Communication sublayer (consisting of the Domain Controllers or DCs and the Data Acquisition and Control Manager or DACM) and the Data Access Object sublayer (containing the Data Access Objects for both the DC and the DACM). The lowest level of the architecture contains services related to the building (such as the ICT infrastructure for weather and access control), the Building Information Modelling Server, the Data Warehouse infrastructure and the external services used to gather data for the system, like the weather forecast. Overall, the usage of an API or the usage of standardized software technologies, like JavaScript Object Notation (JSON) for the Building Information Model, Java Database Connectivity (JDBC) for the Data Warehouse or Simple Object Access Protocol (SOAP) is consistent with what has been defined in the manuscript. Unfortunately, the proposal does not include services that have become almost mandatory to have in a middleware architecture, such as semantic capabilities or securitization.

2.11. OpenNode Smart Grid Architecture

Leménager et al. [43] offered their own novel concepts for a middleware solution focused on the smart grid. They described the implementation work done as part of the OpenNode project [44]. In the proposal, it is shown that there is a collection of key concepts (modularity, extensibility, distribution of intelligence, an open common reference architecture, usage of open standards and cost effectiveness) directed to developing an open development for Secondary Substation Nodes (SSNs) that are located within secondary substations of the power grid, and a middleware solution used to interconnected the SSN with the utility systems related to the electrical facilities and infrastructures. It is mentioned by the authors that the middleware, always regarded as something separated from the SSNs and running above them, is used to deal with the stakeholder diversification and the flexibility needed to interoperate among the power grid, the network and the management methods. The middleware is solely focused on hardware abstraction, as there is no information about the kind of services that could be encased in the middleware proposal or any format of the messages that is interchanged between the data-based levels and the lower ones. There are two separated roles for the new software components developed both at the middleware and the SSN level: the SSN is used for interacting with any smart meter or local Intelligent Electronic Device (IED), whereas the middleware is conceived as a way to interchange metering data, along with grid automation information. The information is transferred to an Enterprise Service Bus (ESB) in case it might come in handy to be transferred to other utility management systems. The proposal has been tested in several realistic scenarios where two physical and one virtual test were carried out, but there is very scarce information about some crucial aspects of the middleware, like how data messages are interchanged or if semantics or security services are implemented somewhere inside or outside the middleware.

2.12. Open Issues and Challenges

The proposals that have been presented here cover most of the issues that can be found while developing middleware for a microgrid, or a similar development with a larger scope if a wider smart grid is to be included or even if any other kind of Cyber-Physical System is involved. They are useful contributions to get an accurate idea of what is needed when a development of this nature has to be done, but in most of the cases they provide a proposal for a solution rather than a design that aims to be as generalist as possible or the steps to follow to complete it, as it is the intention of this manuscript. The most relevant open issues and challenges that have been found are as follows:

(1) *Lack of a clear description of what a middleware solution is.* Each of the proposals relies on their own background to put forward a middleware solution. While the presented ideas may come in handy to prove their usefulness in a specific context, they are likely to fall short when tried to be applied in other environment that is still related to the application domain of middleware for microgrids.

(2) *Lack of a defined set of services and functionalities.* As it is shown in the proposals, most of the middleware architectures are capable of handling different services depending on the needs of the system. However, there is not a clear consensus on the services that should be specified in a solution to be regarded as usable for a similarity of requirements in other location, in a way that can be seamlessly ported without next to no edition.

(3) *Lack of instructions on how to develop a suitable middleware solution.* Development of middleware is not as straightforward as if it was a non-distributed system, as it involves different hardware elements that may have different capabilities and may be scattered in a specific area. What is more, depending on where the system is deployed, elements used to manage information or infer knowledge from the system may be rendered useless if there is no need for them. The smart grid is a clear example of this feature, in the sense that there are distributed pieces of equipment

of different computational power and functionalities that are expected to offer different kinds of services (energy consumption metering, power flow control, aggregation of similar profiles, etc.), so the development of middleware is likely to be a convoluted process that may present unforeseen issues during implementation.

Therefore, having a way to optimize the way middleware is designed and developed seems a very convenient concept, as it can save a lot of time and costs, especially if open source tools are used by the programming team. However, in order to do so, a tight control on what is going to be designed, implemented and tested must be established. Consequently, this manuscript tackles these open issues in the following way:

(1) A description of the services and functionalities that should be provided by a middleware solution has been listed in the following section. Among others, security, semantic capabilities or context awareness are presented and justified in the application domain of middleware architectures for microgrids.
(2) Rather than talking about middleware messages or communication paradigms (Client/Server, Publish/Subscribe, etc.), this manuscript puts forward the idea of a software architecture as the best way to contain the services required. In addition to that, an implementation and justification with an ESB is shown as well.
(3) A complete set of instructions on how to develop the middleware solution is further shown in this manuscript. Due to the complexity that middleware development can have, this is a suitable way to offer a contribution to the community of the smart grid and software engineering. The principles that are described here have been used successfully for middleware implementation in the e-GOTHAM and I3RES research projects.
(4) The messages that are used to transfer information from the lower levels have also been included here for more accurate specification, in a way not dissimilar to the ones shown in other standards (OpenADR for Demand Response, ISO/IEC 42010 [18] or software architectures) that are not explicitly described as middleware solutions for the smart grid.
(5) The same procedure has been used for higher-level access points to the middleware: instructions about what solutions to use and how to enable access to the middleware layer have been described too.

3. Elements of a Secure, Open Source Middleware Architecture for the Smart Grid

When designing a middleware architecture for the smart grid and implementing its resulting solution, it will be done as if it was any other kind of distributed system using a software architecture. One of the software methodologies that can be used for this task is the waterfall model [45], where one task goes after the following one and can use the immediately previous activities as a starting ground for their own. There are five stages that are used under this model:

(1) *Analysis.* The functionalities that are going to be offered by the middleware architecture are considered, commonly under the joined contributions of other elements that are not part of the middleware (network layer communications, applications to be deployed by the end users, etc.), but have a strong influence on it.
(2) *Design.* Based on the analysis, a collection of requirements is completed. These requirements are pivotal to decide which kind of services will be provided by the architecture, as they offer the functionalities that were described by the requirements that have been established.
(3) *Implementation.* The development of the software components needed to fulfil the design done in the previous stage is provided. It is very likely that this will be the most time-consuming and challenging stage of them all, and unforeseen mismatching with the designed procedures can take place here easier than anywhere else.

(4) *Testing.* The implemented functionalities are deployed in a testbed (or the location of the actual system that is going to be used for its future exploitation) so that it can be evaluated if they are delivering the expected services, and if there is true seamless integration of all the hardware components and the applications developed for the users.
(5) *Maintenance.* Once the prototype, with all its ancillary services, has been proven worthy of its use, it will be maintained for its continued exploitation. This stage can involve formal contracts to replace or add new hardware or software components to the overall system.

Despite the initial conception of this software development model, it is advisable to use some feedback in each of the stages in order to guarantee that the obtained results are matching the expectations that were previously formulated, thus resulting in the waterfall model with feedback among stages. Furthermore, there are other software development models that can be used as a way to support the ongoing work. For example, incremental prototyping can be used to repeatedly iterate the previous stages if a prototype or demonstrator is too complex to be tackled with all its features to begin with, as the waterfall model and incremental prototyping offer complementary advantages [46]. Taking into account the directions provided by ISO/IEC 42010 [18], there are several concerns that must be taken into account in the proposal made in this manuscript, namely:

(1) The purpose of the system of interest: developing a middleware architecture according to a set of procedures that ensure that will be implemented with as little trouble as possible.
(2) The suitability of the architecture for achieving the system-of interest's purpose(s): it has been attempted to go further the boundaries of what has been established as middleware for a microgrid by inferring the weaknesses of the existing proposals, as well as from the experience developing middleware solutions for demonstrators.
(3) The feasibility to construct and deploy the system-of-interest: the different middleware iterations that have been tested in two different locations prove that the system-of-interest is feasible enough to be implemented in a real-world scenario.
(4) Potential risks and impacts of the system-of-interest to its stakeholders throughout its life cycle: most of the potential risks come from the chance of a failure in the hardware components where it is installed, rather than the middleware itself. Should it fail, its distributed nature and the existence of software components such as context awareness would make possible that it still worked partly.
(5) How the system-of-interest is going to be maintained and evolved: since an open source ESB has been used for the bulk of the system, new services can be added (even without stopping the ESB and interfaces from the API can be developed the same way as the ones that already exist. The registration of new devices just requires an XML file with their capabilities and an ontology update.

In a similar manner, OpenADR 2.0 has a collection of key ideas that, even though are more oriented to Demand Response than software interoperability and middleware architectures, are noteworthy of mentioning:

(1) The standard puts forward the idea of offering access points for applications. As it was mentioned in the introduction, high level applications are offered HTTP and XMPP endpoints that will enable their access to lower levels for Demand Response use cases.
(2) They idea of having effective security measures plays also a major role in the standard. This concept has also been incorporated in the proposal and procedures that are described in this manuscript.
(3) A collection of services that has to be made available is also very consistent with the idea that is presented in this manuscript about having a middleware solution that is containing all the required functionalities for the correct performance of the smart grid as a distributed system.

(4) Message payloads present in the standard make use of XML signatures. Making use of established standards for data representation has also been used in the presented manuscript as a way to make device interoperability and application access easier.

(5) Several requirements are described as a way to assess conformance to the standard. Requirement analysis is a concept of pivotal importance in this manuscript, as it determines the kind of services that are going to be implemented in the middleware solution.

As far as the ISO 50001 standard is concerned, it aims to extend the management characteristics present in other standards from the same organization (quality management in ISO 9001 [47], environmental management in ISO 14001 [48]) to the application domain of the Energy Management Systems by means of establishing energy policies for organizations. While this purpose is out of the scope of this manuscript, energy policies are implicitly enabled by the integration of the devices present in a microgrid by means of a middleware solution.

There are several elements that must be taken into account regarding middleware architectures for the smart grid: since a middleware architecture is expected to do functionalities regarding interoperability, scalability or information treatment and transfer, there will be some software and hardware elements that are almost of mandatory nature in order to be able to deal with those functionalities. Those elements are:

(1) *A framework capable of wrapping all the components that will be developed for the middleware architecture.* This is like having a container that can be transferred from one place to another whenever the middleware architecture has to be ported from one hardware location to a different one. Additionally, architecture designs that rely on some kind of software architecture paradigm can be troubleshot in an easier manner and methods used to access their facilities (interfaces, communications) are well known and documented. For the set of procedures that is going to be offered in this manuscript, an ESB architecture is put forward. Furthermore, the availability of information regarding the framework that is going to be used must also be considered. From the perspective of available data, open source developments are more advisable that having proprietary solutions. The reasons for making this choice are several: to begin with, open source solutions are usually cost free and operate under different kind of licenses that do not require the pay of any major fee when using them. In addition to that, there is plenty of information available for open source solutions, such as available code in online repositories as GitHub [49].

(2) *Security procedures have to be used as well, in a way that they will not affect the overall performance of the system*. The purpose of using security in the middleware architecture is that the system is deemed as secure by the end users. There are several solutions regarding cryptography and algorithms able to provide security. In addition to that, access to the developed middleware solution must also be granted in a secure manner, so that it will guarantee that data does not get tarnished by hostile actors for the system. Security procedures must also be taken into account in two other scenarios: on the one hand, it is required for all the operations required to charge the end customer with the expenses of consuming electricity from the power grid (for example, in the messages that are interchanged in the communication links, security can be provided by means of message encryption or using firewalls in the system [50]); without information privacy or data integrity, the regular operation of the grid can be easily altered and it will result unreliable. On the other hand, there are comparable issues in other distributed systems, such as Wireless Sensor Networks, where privacy regarding the data collected, as well as other parameters like information transfer through network hops [51] are taken into account as well.

(3) Since the middleware architecture is located between the more hardware-based layers of the system and the upper ones, *technologies that are going to be used to interface hardware and communications on the one hand, and applications on the other hand, are necessary to be taken into account.* The existing solutions have been studied with the same criteria that were chosen for frameworks or information availability for the system: proposals that require the least economic

cost or had available the most significant amount of information can be put forward before other solutions without these features.

(4) *Services to be offered are one of the most important characteristics to be taken into account,* as the amount and the available services configure almost any other aspect of the middleware architecture for the smart grid. Commonly, those services will be related to accessing to lower and upper layers, device registration, context awareness, demand side management, demand response, optimal power flow, etc.

Taking all these aspects into account, a justification has been elaborated for each of the presented topics so that it will become clear what has been chosen for each of the needs of the middleware architecture.

3.1. Justification of the Enterprise Service Bus (ESB) Architecture

As far as the framework (or in a more loose manner, the software wrapper for all the services that the middleware is made of) is concerned, there were several architectures that could be considered. Among all of them, ESB was chosen as the most suitable software architecture. This architecture can be used as a way to contain services of very different nature in a distributed manner [52]. As it has been portrayed in Figure 2, ESBs are the intermediate entities used to interconnect the providers of a set of features with the consumers of those features, regardless of them being human beings (end users) or other entities that are related to another part of the distributed deployment, while decoupling the different times when messages are sent and received (thus, allowing the usage of publish/subscribe paradigms that will not require sending and receiving messages in real time) while sharing a common environment for all the applications to be deployed and interconnected by the bus (such as the smart grid). The software components that will be used to satisfy the services are provided as bundles, that is to say, software packages that contain the executable files, ancillary files and any software code that has been built for a service. Those bundles will communicate one with the other by means of internal messaging protocols, such as Java Message Service (JMS). Outside of the ESB, communication protocols can be enabled and used as in any other location: HTTP, Representation State Transfer (REST), Service Oriented Architecture Protocol (SOAP), Java EE Connector Architecture (JCA) or SPARQL Protocol and RDF Query Language (SPARQL) are usable in an environment with an ESB implementation.

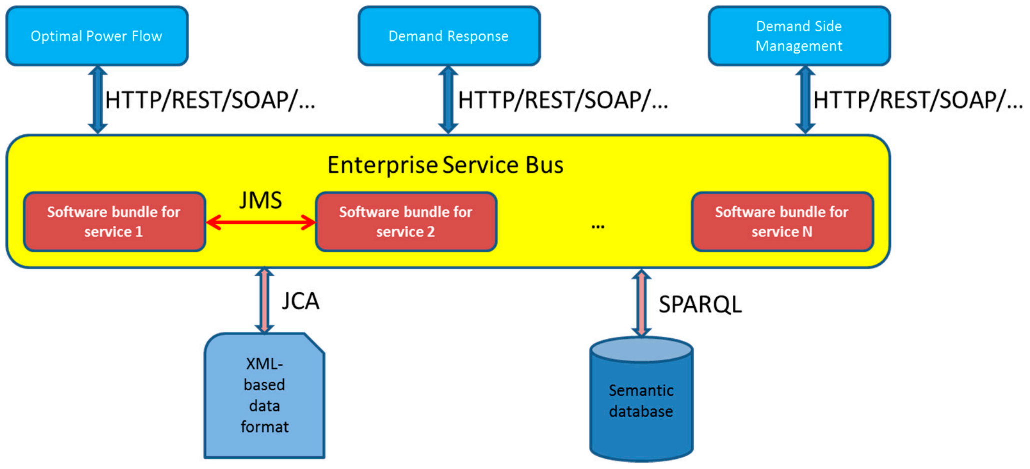

Figure 2. Enterprise Service Bus and interconnection capabilities.

An ESB can be used to interconnect both applications developed with different features (programming languages, data formats for information transfers) and lower level pieces of equipment that also have a significant degree of heterogeneity (legacy pieces of equipment not meant to be used as something integrated in a software-reliant system, new applications that have novel interfaces that are still not easy to access, etc.). While there are other options that could be used as frameworks for a middleware architecture (that is, different software architectures that will contain all the modules that will be implemented), using an ESB seems the most suitable of them all for middleware due to the following reasons:

(1) An ESB offers a place to store all the services that will be used by the middleware architecture. Rather than being used as a way to interchange messages or perform mere data transfers, services and their functionalities can be kept in a non-volatile location to access them whenever it is required. Implementations regarding ESB will invariantly make possible the storage of the services or store services themselves, like in MuleESB [53], Petals ESB [54] and other ESB solutions.
(2) As explained before, an ESB has been conceived from scratch to adapt different applications built in different software languages with different functionalities. Therefore, it matches with a high degree of accuracy what is expected from the smart grid, since the latter are expected to include services related to security, semantic capabilities or context awareness, as it is shown in [55].
(3) The concept of the bus as a way to use it for data transfer and service deployment is very consistent with the idea of having an intermediation architecture that is including a collection of services accessed by the applications. This layer of middleware architecture accurately matches the bus of an ESB, as it has already been used in other distributed systems with a strong need for decentralized data storage and management [56].
(4) By having a bus to transfer information from one part of the system to the other, it is made sure that there are no centralized (or at least especially prominent) components used to manage interconnectivity. Consequently, an ESB can be defined as an implementation or an embodiment of a distributed system from a Service-Oriented Architecture point of view [52], which is a better approach for a middleware architecture in a smart grid, since it will contain several DERs and different pieces of equipment regarding the power grid.

It is important noting that by choosing an ESB Architecture there are other software solutions and styles that can be used in a complimentary way as well. For example, REST was developed by Roy Thomas Fielding as an architectural style in his PhD thesis [57]. While one of the objectives of the web services that become enabled by REST is the interconnectivity of different facilities present in the Internet, thus becoming somewhat close to the objectives of an ESB architecture, its area of interest not only does not collide with the one represented by the ESB, but also has a symbiotic-like relationship with it because REST interfaces behave as an access point to the underlying middleware architecture represented by the ESB. As far as the licensing paradigm is concerned, using open source information is something that must be considered favourably, as there are many developments that use open source to accomplish their functionalities. Thus, it is the authors' opinion that a Free Open Source Software (FOSS) development is the best possible choice for a middleware architecture.

3.2. Justification of the Security Procedures

The need to have security in a middleware architecture for the smart grid should come as no surprise, as it is a distributed system that must be provided with some kind of mechanism ensuring that the data being transferred during a regular use of the system will not jeopardize the performance of the services offered by the system. Taking this into account, there are several ways to provide security that might even be complementary. As a first introduction on the actions taken regarding the implementation of security procedures, it can be mentioned that:

- *Hypertext Transfer Protocol Secure* (HTTPS [58]) can be used for the interfaces of the system that has been implemented and described. While it forces the use of web certifications and some implementation works, it offers security features as authentication, confidentiality and data integrity to make up for them. It has to be taken into account that not only additions and new features are added at the application layer to the HTTP protocol, but also there are underlying functionalities that have to be implemented as a result of the needs of lower layers. These are tackled by the usage of TLS, which is an update of SSL (Secure Sockets Layer) that uses sockets to establish communications. TLS/SSL can be regarded as a protocol that enhances communications by providing security to a network where interconnected computers of different kinds are transferring data [58]. Despite being explicitly mentioned as a transport layer (due to the fact that it still uses sockets, as SSL does), it needs a degree of control over upper, session-based functionalities, since TLS is in charge of being aware of re-transmission and segment loss.

As for the middleware architecture that is advised to be used in the smart grid, it will still be located between the application-based layers and the hardware-based ones, so it will be acting as a go-between among the security services that are used at the transport layer and the needs required for the application one.

- *Cryptography* is usually associated to upper layers in the OSI and TCP/IP models, as these are the ones capable of performing the required operations to cypher and decipher messages. Although these operations are done locally, the messages are transferred from one machine to another one within a computer network, so the content of the operations will travel in a secure manner from one piece of equipment to the other. There is a plethora of algorithms that can be used for securitization, such as Rivest, Shamir and Adleman (RSA, [59]), Advanced Encryption Standard (AES) [60] or Diffie-Hellman [55,61]. In addition to that, care must be put in whether security algorithms will be using symmetric (private-key cryptography) or asymmetric (public-key cryptography) cryptography. Asymmetric cryptography is regarded as the safest one, as it does not require having a separate channel to transmit the key that is going to be used (and there is no way that this key can be compromised). The period of time required to complete the cyphering is usually longer than the one used with symmetric cryptography, though, and may be a reason strong enough to use symmetric cryptography instead.

3.3. Justification of Low and High Level Technologies

Since middleware is placed in the natural place where data are being transferred from hardware to end user interfaces, a way to access the data that are being transmitted from below must be figured out in order to have the applications receiving the data that has been sent from the hardware devices. In addition to that, the services located at the middleware will also be generating their own information when they are requested (or as part of their usual operations, such as context awareness), so they will also need to interact with end users (via application layer interfaces) and appliances located in the grid (via network layer interfaces). Due to these reasons, interconnectivity technologies are required to maintain the seamless integration of all the subsystems that make up for a smart grid (or at least, the microgrids that make up the whole smart grid). The technologies used will make a huge difference depending on whether they are used for low level or high level interconnectivity. Regarding low level technologies, the protocols that are used here must be able to adapt themselves to the paradigms most used regarding data transmission in communications from the network layer upwards, that is to say, Client/Server and Publish/Subscribe. There are several tools to manage this issue. Two of the most popular ones are JMS and Advanced Message Queuing Protocol (AMQP).

- *JMS* is a Java-made Message Oriented Middleware (MOM) API [16] focused on sending messages among two or more clients. As it can be inferred from its name, it is heavily conditioned by the Java as its programming language of implementation, since it is regarded as part of the enterprise edition of the Java platform, and will therefore be able to be used as a tool for sending, receiving, reading and creating messages, although just for components based on Java 2 Enterprise Edition.
- *AMQP* can be regarded as a protocol that "provides a platform-agnostic method for ensuring information is safely transported between applications, among organizations, within mobile infrastructures, and across the Cloud" [62]. Its flexibility allows its usage among all the stack layers, ranging from connecting hardware devices to applications. As it happened with JMS, AMQP can be used for point-to-point and publish/subscribe communications. Another feature that makes AMQP suitable for data transmissions is its capability of incorporating security elements, as encryption can be based on either TLS or Simple Authentication and Security Layer (SASL) [63].

As for the protocols used in higher level interfaces, there is also a plethora of solutions that can be used to seamless interconnect the developed middleware with high-level interfaces that will be used for applications. These are usually regarded as application layer protocols. Their functionalities are usually very specific and oriented to data transmissions. Among these, the already described HTTP protocol is one of the most popular ones. Other solutions to take into account are File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP) or Secure Shell (SSH) for securitization of data transfers. Finally, there are some other high level solutions that will be shown in the manuscript.

- *Constrained Application Protocol (CoAP)* is bent on providing interconnectivity for devices with very low computational capabilities, such as the ones that can be found in the Internet of Things or, to an extent, as hardware subsystems in the smart grid. Its main functionality is translating the HTTP requests that are done by end users (or actors in a more general way) for a simplified integration of those in the web. Additionally, it also offers some other services such as multicast support, low overhead and overall simplicity. User Datagram Protocol (UDP) is used as the most suitable transport layer protocol, as it is more advisable to have it used for resource-constrained application domains due to the fact that does not implement a procedure for data retransmission as the one done by TCP, which is more reliable but demands higher network resources.
- *Representational State Transfer (REST)* is an architectural style (rather than a protocol, as the former solutions) that has become widespread as a result of having it used in the World Wide Web. As such, REST is oblivious to the underlying implementation details that are used. It aims to provide several features that are common among middleware architectures: scalability, modifiability, portability, etc. Operations (or verbs) that are defined to be used when implementing REST interfaces are very closely related to the ones used in HTTP: GET (in order to receive messages from the upper layers), POST (acceptation of an incoming message as a subordinate of the resource targeted by the URI), PUT (storage of the enclosed entity in the operation under the URI used to access the system) and DELETE (so that a resource will be deleted) are defined.

From the two previously described solutions, REST interfaces have been included in the proposal of a middleware for the smart grid, since they offer a more versatile usage regarding the capabilities of the devices using those interfaces as access points at any kind of service that is located below them.

- *Message Queuing Telemetry Transport (MQTT)* is regarded as a connectivity protocol for machine-to-machine or Internet of Things communications [64]. It aims to be a lightweight solution that allows transferring information via messages that work under a Publish/Subscribe paradigm. MQTT v3.1.1 became an OASIS standard on 7 November 2014. Contrary to REST, which requires a low number of significant operations, MQTT works by exchanging fourteen different Protocol Data Units that are referred to as *MQTT Control Packets*. The first one is called CONNECT (used when a client requests a connection to a Server), CONNACK (utilized for acknowledge connection requests), PUBLISH (for message publication), PUBACK (for publish acknowledgement), PUBREC (response to a PUBLIC packet with Quality of Service 2 according to the standard), PUBREL (response to a PUBREC packet), PUBCOMP (response to a PUBREL packet), SUBSCRIBE (used to subscribe to topics), SUBACK (subscribe acknowledgement), UNSUBSCRIBE (in order to unsubscribe from topics), UNSUBACK (unsubscribe acknowledgement), PINGRED (ping request), PINGRESP (ping response) and DISCONNECT (that is, a disconnect notification). It has a purpose that resembles the ones found in the other protocols, since MQTT targets application layer, light/constrained communications among a distributed system. According to the Website devoted to its development, MQTT was formerly known with other names, like SCADA protocol, MQ Integrator SCADA Device Protocol (MQIsdp) or WebSphere MQTT.

- *Supervisory control and data acquisition (SCADA)* is a control system architecture that can also be regarded as a way to interact with several different elements deployed in an area. Although the features of a SCADA system are quite different from the ones that were shown before (due to the fact that rather than being located at a level in a layered software model, a SCADA system implies devices collection information and transmitting it via networked communications and displaying it to an end user via Graphical User Interface), there are protocols in this domain that can be used for data transmission in the domain of a power grid. For example, the Distributed Network Protocol (DNP3) can be described as a collection of communication protocols that offer features such as Secure Authentication and is responsible for data transmission between master stations to outstations [65].

3.4. Services Expected to Be Offered

One of the main reasons for having a collection of services that differs from one kind of development to the other is the use cases that will be used. Under the perspective of software modelling activities using Unified Modelling Language (UML, a general-purpose language used for modelling in software engineering conceived with the aim of providing an standardized view of a software system [66]) use cases can be determined by the usage of use case diagrams.

When developing a middleware solution, the services expected by the end users (or in a different manner, by the hardware appliances that have been installed in the power grid) must be installed somewhere in the system. That is to say, the software packages and modules required to deliver those services, in conjunction with the hardware appliances, must be located somewhere so that they can perform the functionalities that are expected from them. This may cause challenges in the whole system, as they must be located in a place that can be reached by the software functionalities, and the distributed nature of the smart grid makes it difficult choosing one hardware device over another one. Fortunately, many of the services that can be used in the smart grid can be located in the middleware architecture. Despite this is not an inevitable or mandatory procedure to follow, it is advisable due to the fact that, by having those software components in a distributed environment located between the network and hardware capabilities of the system and the applications layer, they can be accessed by any application and any network element from the distributed system. What is more, due to the construction features of the equipment that is used in the smart grid, devices participating of it might be even proprietary solutions that cannot be altered (which would be equivalent of accessing one device without the manufacturer's authorization) or have too little power to have all the required software capabilities installed (such as end user AMI). Overall, there are four different kinds of services that be found in a middleware architecture for the smart grid, while there are three kinds of them that can be extended to almost any other application domain:

(1) *High level services.* These are the services that are in closest cooperation with the applications that are built immediately above the middleware architecture. They will be either providing a specific functionality for the system or will be used as an access point for the functionalities that are placed in lower layers of the middleware, or even outside it. Nevertheless, the services will always be accessed the same way, namely, via high level interfaces that are cooperating with the application layer as if they were common Service Access Points (SAPs) in a layered architecture. In the previous section it was explained how REST software architectures or CoAP protocol could be used for that purpose; as long as there is a way to send and receive information from/to the applications that are being used, high level services can be utilized to interact with them. Should the high level service be developed as something with more functionalities than just being an intermediary between the applications and the middleware, those functionalities will be accessed in the same way, with the difference that there will be no other messages sent to any other location of the middleware architecture or lower levels. These latter services can be referred to as Autonomous Services: services that are used to access other parts of the system can be regarded as Middleware Access Points.

(2) *General purpose services.* These services are to be expected in almost all the middleware architectures that are based on building software components within them so that additional services will be provided from the middleware architecture. When considering that a middleware architecture is going to be included in the system there are some functionalities that will have to be provided:

    (a) It is of major importance knowing the hardware devices that are present in a particular area of the smart grid (either a microgrid, or something wider, but always measureable and significant for the smart grid), so they must be registered in the middleware. By registered it is meant that information regarding their characteristics must be kept in the middleware architecture by any kind of mean that makes it easily accessible (databases, files, etc.) in case it is requested.

    (b) Additionally, having the middleware architecture capable of grasping the meaning of the information that is either transferred or stored in it is a desirable feature with a major impact in the whole system, as it will be able to have many decisions automated and knowledge will be inferred from the system. This can be achieved by the usage of semantic capabilities, which are offered by the implementation of ontologies. In a very generic way, an ontology can be defined as a dictionary that contains data about all the terms and definitions that appear in the system, while at the same time keeping the relationships between those terms [67]. Ontologies and semantic capabilities are a major feature in any system with machine learning capabilities with a wide range of applications [68].

    (c) Standardization processes are consistent with middleware architecture's in the sense that they are the ones capable of abstracting the heterogeneity of the existing equipment and communications. Common Information Model (CIM) easily comes to mind when these features are taken into account [69]. Its conception as a UML-like model for standardization of power grids entities can be ported to a more general point of view where middleware is used both in smart grids and any other application domain. Whatever is developed, a way to standardize the information that is being retrieved as a result of either application requests or publish/subscribe paradigm actions should be implemented in the middleware architecture.

(d) Context awareness is increasingly becoming a relevant subject in middleware architectures as well. By this feature it is meant how the system is capable of perceiving the parameters that make the environment where it is confined, as well as the capacity of reacting to those parameters [70]. In this way, a software component gathering information of several features of the deployment is required as well.

(e) Last but not least, security functionalities must also be taken into account. This is a service that will be required on a general basis, so it makes sense that it has to be provided as part of the middleware. In any case, it does not collide with other security measures that might be adopted somewhere else in the system.

(3) *Distributed energy services.* These are the services most specific to the smart grid, so unlike the former ones or low level services, they are not easy to port to a different area of knowledge. These services gather the most typical functionalities of this application domain, so they can be expected to cover the most usual functionalities of the smart grid: Demand Side Management (modification of the end users' behaviour about energy consumption with the idea of making it more suitable to the needs of the power grid in the long term), Demand Response (punctual modifications of clients' behaviour in order to obtain a short-term benefit in their energy consumption expenses, such as during daily peaks) or Optimal Power Flow (description regarding the flow of electric power so that it will match the best fitting possible quantity during the regular usage of the smart grid).

(4) *Low level services.* These are services devoted to the interconnection of lower level pieces of information that are sent from/to the hardware devices that obtain the key measurements from the distributed electrical system. More often than not, they will consist of hardware abstractors using the technologies that were presented before (JMS, AMQP) in order to interface the communications that interconnect the pieces of equipment at the network layer level.

The expected inputs and outputs for each of the services, along with what type of service can be expected from each of the main features, have been placed in Table 2. This chart will be used to describe the procedures used while developing the middleware solution for the smart grid.

When all is said and done, the middleware architecture offered as an example in this manuscript takes into account all the features that have been included before. Rather than having one number of software modules or another, the underlying idea is that all the previously mentioned functionalities must be included in the middleware architecture. An example of an architecture that would fulfil all the previously shown requirements would be is in Figure 3.

The architecture design that can be seen in the previous figure is a typical result from the design stage. General purpose services, Low level services, High level services and Distributed energy services can be regarded as four subsystems that will be further divided into several components within each of them that will be used to tackle how the services are being offered and which classes, methods and functions have to be codified.

Table 2. Inputs and outputs for each of the services.

Name	Service Type	Description	Input	Output
Autonomous service	High level	Service that is requested from an application that requires no other one in the whole system to deliver the required information	Request made from an application	Data expected to satisfy the former request. It is obtained from the very service that was requested to obtain it.
Middleware Access Point	High level	Service that is used to access more complex functionalities that lie either in the middleware architecture or somewhere else (but always at a lower level than high level services)	Request made from an application	Data expected to satisfy the former request. It is obtained from devices or services located in lower levels.
Context Awareness	General purpose	Service that is capable of providing information about the surroundings of the system	High level requests regarding system surrounding parameters	Information about context parameters (temperature, power flow values, etc.). Context awareness can also be triggered without previous user requests if an action has to be taken when taking into account parameter values.
Security	General purpose	Service that provides security functionalities (confidentiality, data integrity, authentication)	Data transferred through the middleware architecture	Securitized data; either encrypted information or authorized access to the middleware architecture
Semantic capabilities	General purpose	Service that enriches the received information by means of ontologies and semantic capabilities	Data received from middleware actions (device registration, device discovery requests, etc.)	Semantically enriched information that might be used to complete registration, information storage, etc.
Standardization	General purpose	Service used to adapt the different information formats received in the middleware architecture	Data formatted according to the proprietary solutions existing in the hardware devices	Data formatted according to the choice that has been done regarding middleware format
Registration	General purpose	Service used to include information of the available devices and services in a deployment	Device registration request	Either the request is send to the semantic capabilities module or an Acknowledgement is sent to the device that sent the request in the first place
Demand Side Management	Distributed energy	Plan to influence the user into adopting long term habits regarding energy consumption	Data regarding energy usage during a comparatively long period of time	Data regarding a plan for energy consumption optimisation in the long term
Demand Response	Distributed energy	Plan to influence user to adopt temporary and/or short term measurements for energy saving	Data regarding energy usage during a comparatively short period of time	Data regarding a plan for energy usage in the short term (peak shaving, etc.)
Optimal Power Flow	Distributed energy	Description of the best possible way to transmit electricity from one part of the smart grid to another	Data regarding energy consumption in a very short period of time (in a real-time fashion)	Data regarding a plan to optimise the usage of energy
Hardware abstractor	Low level	Service used to adapt low level communications (hardware, network layer protocols)	Data received from the network layer	Data sent to the middleware architecture

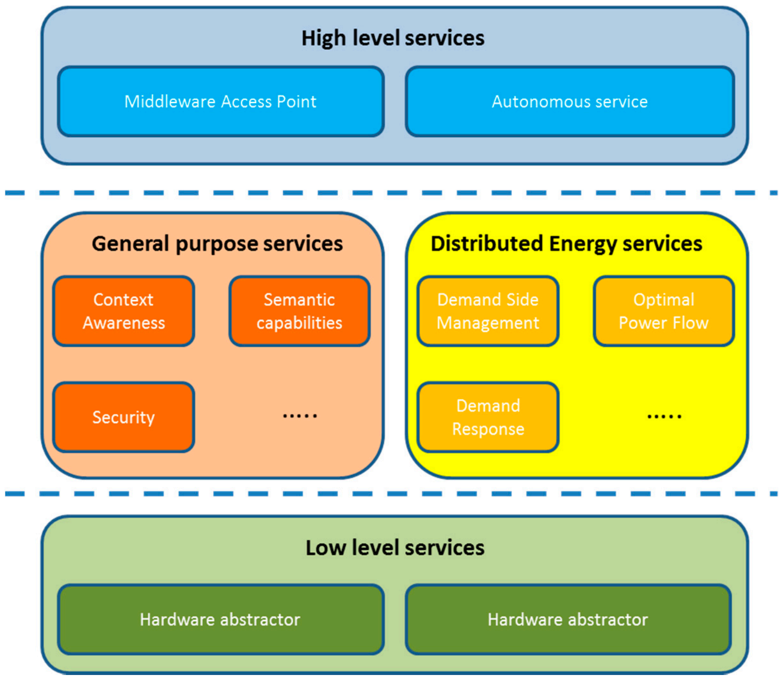

Figure 3. Middleware architecture for the smart grid.

4. Development of the Middleware Solution

As already mentioned, there are two steps that must be taken before implementing a middleware solution for the smart grid, that is to say, analysis and design. The output that will result after the execution of these two stages is: (a) the establishment of a collection of functional and non-functional requirements that will be used in order to design the architecture and (b) the very architecture that has been designed, where all the development works will be carried out. In this case, the architecture that has been conceived for the smart grid comes in very handy, as there is a collection of requirements unlikely to change from one development to the other. Fixing that group of services and functionalities that are common in all the imaginable middleware deployments is of great help while undertaking middleware standardization works into account. It has to be noted that by requirements is meant functional requirements, which are determining how the system does what it is supposed to tackle. These are the requirements that are taken into account to design the bulk of the functionalities that will become implemented during the next stage of the waterfall cycle. Non-functional requirements, on the other hand, are the ones that not only define what the system really is (features of its capabilities, its most prominent features, etc.) but also establish boundaries regarding what the system is able to offer, so they will be the ones offering very specific features that must be fulfilled by the system, whatever way is used to implement its inner functionalities.

Table 3 displays how the obtained requirements from the design stage (that, at the same time, used the analysis stage to determine which facilities would be offered by the middleware architecture) are used as an input for the design of the software components that will be codified.

Codifying a middleware solution can be a complex, time-consuming task. In order to develop it in an ordered manner, there are some actions that must be taken in a specific order so that everything will be developed, installed and tested minimizing the waste of time and the dead ends that may occur during the period of time used for development works. The order that is provided for the implementation activities to be carried out after the requirement analysis and design are finished is as follows:

(1) *Framework installation.* The overall software location where bundles are going to be used must be installed. Since it was decided to use an ESB architecture to install all the developed software packages, and the open source paradigm was found to be the most advantageous one, this step implies finding, downloading and installing an open source ESB.

(2) *Software development.* The software packages that are going to be included in the architecture as the services provided to the whole smart grid are developed. Following the terminology that will be used afterwards, these software packages will be called *bundles*. Besides, how bundles are interconnected with each other and how they are used to transfer data from one service to another must be made clear and implemented in a way that is neither convoluted, nor represents a potential threat to the normal performance of the installed software.

(3) *Interconnection of high level interfaces.* When a significant number of middleware components has been developed, interconnections among the lower and upper level facilities must be provided; should there be no connectivity, information cannot be transferred and the whole system will fail, which is a common concern about the implementation of middleware architectures.

(4) *Interconnection of low level interfaces.* The interconnection works are extended to the low level interfaces to include network communications and hardware. Messages that are used to interchange data in the deployment must be formalized to ensure that the architecture is capable of interacting with the available devices. At the same time that the previous actions are taken care of, interfaces and access can be made secure so that only authorized personnel or end users will access the middleware.

(5) *Securitization procedures.* It is important to perform securitization as a parallel work related to the ones involving interconnectivity, as interfaces are the elements that will be used to access the middleware solution and they should be secure by the time testing is done.

Table 3. Usage of requirements for implementation duties.

Requirement (Design)	Implementation (Action)
Semantic capabilities are required for machine learning and knowledge inference	Development of a semantic module
Available hardware components must be registered	Development of a registration module
Applications must be able to access the middleware architecture	Development of Application Access Points
Hardware components must be able to access the middleware architecture	Development of modules with low level interfaces
Data confidentiality, integrity, and authentication must be provided in the deployed system	Development of security capabilities

4.1. Framework Installation

The first action that must be taken is installing the framework to be used in order to contain the software components that will be developed in further steps. Among the collection of ESBs available, JBoss ESB, which is the one developed by Red Hat Inc. (Raleigh, NC, USA), seems to be a suitable one, as the framework is already built and can be downloaded with ease [71], there is a considerable amount of resources available that come in handy during the development stages and tests can be done in a fast manner (as software bundles can be deployed while the ESB is up and working, thus allowing the real-time addition of additional services in an industrial environment, rather than having to stop the framework, cancelling the services for a while and restarting the whole framework again). The JBoss ESB is provided as a community effort and does not offer any kind of Software Level Agreement, though.

In addition to that, the programming language that will be used has also to be taken into account. In the example that is put forward in this manuscript, Java was used as the programming language, as: (a) it is one of the available options for the development of software packages in the particular

ESB that has been chosen as a framework; (b) it is the programming language used for the bundles that are deployed without having to stop and re-start the ESB, and overall; (c) Java is one of the most popular programming languages currently used, with a significant amount of resources that can be consulted [72]. Therefore, it is very easy to find documentation about what procedures should be followed in case a bug is found or there is any issue while writing the software required for each of the services include in the ESB. By downloading the development kit for the programming language of choice, development activities can be carried out. In case of Java, the development kit is named Java Development Kit or JDK.

When the Java Development Kit is installed implementation works can be done regarding the middleware architecture. The actions required to install JBoss ESB once it has been downloaded are simple enough for a Unix-like machine:

(1) The compressed file that contains the JBoss ESB is unzipped.
(2) If required, the resulting directory can be transferred to another location. There are two directories within the larger one that are of major importance: the one named deploy and the one named bin. The former one is used to store all the executable software bundles that are developed once they have been codified, whereas the latter is used to access the executable file containing the script used to launch the JBoss ESB. This file might be different depending on the JBoss ESB version that is downloaded.
(3) Once it is launched, access to the default installed bundles can be obtained by executing the command list.

Another available solution in order to deploy an open source ESB for devices with lower hardware capabilities, such as a Raspberry Pi [73] can be present in obtaining an Apache ServiceMix Minimal Assembly version [74]. This light ESB does not provide any default bundle, so a small set of them might have to be installed.

One last feature that has to be taken into account is the usage of an Integrated Development Environment for the development tasks that will be carried out during the following stage of the project. Eclipse and NetBeans are the two most popular options, as they both are capable of having plugins installed for the expansion of their activities and can be used for multiple development languages. They can be downloaded either by means of an installer with several different options (Eclipse, [75]) or by choosing an option among the one with the suitable capabilities (NetBeans, [76]). For the example that is shown in this manuscript, Eclipse was used as the IDE of choice for the development works.

4.2. Software Development

The result that was obtained in the design stage (the design for the middleware architecture to be deployed in the smart grid) will be used for the implementation of software bundles. However, in order to do the wrapping process that will be needed to have software bundles in the middleware solution, a maven-related project must be created. A maven project relies in the usage of Apache Maven, which is a software project management and comprehension tool used to coordinate a project build, reporting and documenting it [77] under the Project Object Model or POM [78]. Therefore, there will be a number of ancillary files that will be created and/or will have to be modified, as it will be described in the next section. The four subsystems that were present (High level services, Low level services, General purpose services and Distributed energy services) can be further subdivided in components that send and receive data among them and other components from other subsystems:

(1) *High level services*: each of the services can be regarded as a component from the point of view of UML design. Typically, there will be two different kinds of components: the ones that are regarded as Middleware Access Points (offering an interface to interact with the application layer and another one to deal with other components of the middleware architecture) and Autonomous Services (services that do not require interaction from other services in the architecture). While

communications between these two kinds of components are rare (Autonomous services are unlikely to interact with others, hence the term autonomous), both components will be providing interfaces used for interconnectivity between services and, above all, applications located in a higher level, which will be accessing to the middleware only via High level services.

(2) *General purpose services*: there will be some fixed patterns for communications between these services. For example, whenever there is a communication and security has to be enabled, the messages that are transmitted throughout the middleware architecture will include the security component in order to have their content ciphered and deciphered. In addition to that, information regarding context will be used whenever there is a service that requires data from the location of the appliance that has the solution installed, so messages will be using the context awareness component for their tasks. Finally, semantic capabilities are likely to be used whenever there is a request that involves semantically enhanced information (for example, a SPARQL-formatted request from an Application Access Point) so they will be used regularly when data transfers take place from one subsystem from another one.

(3) *Distributed energy services*: this subsystem will be using a plethora of changing services regarding details that go beyond the scope of practical implementation works (Service Level Agreements, needs of the end users of the microgrid where the middleware is being installed, etc.). Nevertheless, it can be assumed that there will be three services that will be implemented: Demand side management, demand response and optimal power flow. Each of these will be interconnecting itself with the required components of the middleware architecture for data transfer. While it is very likely that they will be using functionalities from general purpose software services (semantic information from the registered devices, encrypted sensitive information, etc.) they can work in a differentiated way one from the other, as they perform functionalities that do not require capabilities from each other. That is why they are related to high level services and general purpose services more than with each other.

(4) *Low level services:* these services will mainly be involved in data transfers from lower levels and will send data to either energy generation services or general purpose services. Therefore, their interactions are quite simple, as they only imply sending and receiving data from/to higher middleware levels, and each of the hardware abstractors has been tailored for the specific needs of each of the hardware devices they are related to, if necessary.

Thus, implementation works will be made by establishing what the components of each subsystem are made of. Regarding Object-Oriented Programming, the smallest, self-contained entity (that is to say, that has a meaning by itself) is the class. Classes can be defined as templates that offer a set of features (attributes) and actions that can be carried out with them (methods or functions) [79] used to develop objects and entities that will be implemented as software components. In order to determine the classes that are going to be required, sequence diagrams can be used. Once the design stage has made clear the classes (with their corresponding attributes and methods) that are required, implementation works can be done by using the framework that was defined in the previous step of the implementation procedures. These actions to be taken will differ from one service (that is to say, component from the subsystems from the UML point of view) and the other. The steps that can be taken in each case are as follows:

(1) *Middleware Access Point*: they will use the requests that are formulated by the end users via applications with any kind of mean they have as a front end (typically, a Graphical User Interface). As previously stated, REST interfaces are shown as one of the most optimized procedures to transfer data requests from one upper level layer to a lower one. When a Java implementation is used, there are several features that will have to be included, namely, the packages that are going to be used to import the REST capabilities and the nature of the operations that will be used to interact with the applications (which can be, among other options, GET and POST ones).

```
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;
 ...
@POST
@Path("/request_type/{serviceID}")
@Produces("application/xml")
@Consumes("application/xml")
<privacy_level> <return_type> <method_name> (<Parameter_type> <parameter>){
 ...
}
@GET
@Path("/getCurrent/{serviceID}")
@Produces("application/xml")
<privacy_level> <return_type> <method_name> (<Parameter_type> <parameter>){
 ...
}
```

As it is hinted from the code that has to be added in a Java implementation, there are two ways that these interfaces can be used: on the one hand, they can use (a) a GET operation to just push the query without any content; it will be up to the Middleware Access Point to add any information required to have it transferred to lower layers (usually, it will be able to add the suitable data by inferring what kind of request is being made from the URI or URI parameters used in the enquiry) or (b) a POST operation that is self-containing the information that is required to be sent through the middleware (the Middleware Access Point does not have to include anything in this case). Depending on the implementation there will be a class or a collection of class responsible for receiving the application request via REST interfaces (REST interfaces connector) that will send the request for its processing (a procedure that implies becoming aware of the request that was done, the services that must be provided, etc.). Once the request has been specified, data will be sent to the class or set of classes connecting the Application Access Point to the lower level services of the middleware architecture. Figure 4 shows which parts the Middleware Access Point is made of. This middleware service has as its main functional requirement the transfer of requests to lower levels where the information of the query can be dealt with.

The resulting output of this service will be a data request using an inner format that will be understandable by the middleware architecture. However, this internal data format is prone to be changed again until it matches the one defined by the middleware ontology defined as part of the semantic module.

(2) *Autonomous service*: these services are self-contained and, due to a collection of reasons (they are very light, do not use context awareness, security is provided in the application, etc.) do not require, or have not been conceived to use the other services of the middleware architecture. This usually implies that the services that are behaving like this do not require direct information from the devices or the semantically annotated information, as they are not performing requests to those components. In this case, there are two different kinds of classes that must be taken into account. The expected class or classes that could be as a connector with the REST interfaces will

remain largely the same, as they will be using the information sent from the URIs and will send the collected data to another object where it will be processed. However, that processing will be quite different because no other component will be used; this will have as consequence that the operations that are performed within the autonomous service are way more complex, as the logic that is used to solve them is contained in one single component. On the other hand, there will be no data transfers to other services, so no data interchange will be made with lower level middleware services. These facts have been reflected in Figure 5. The only interaction is done against the application layer, rather than any other middleware service, so the main functional requirement of this service is processing the request as a self-contained set of functionalities capable of satisfying the latter.

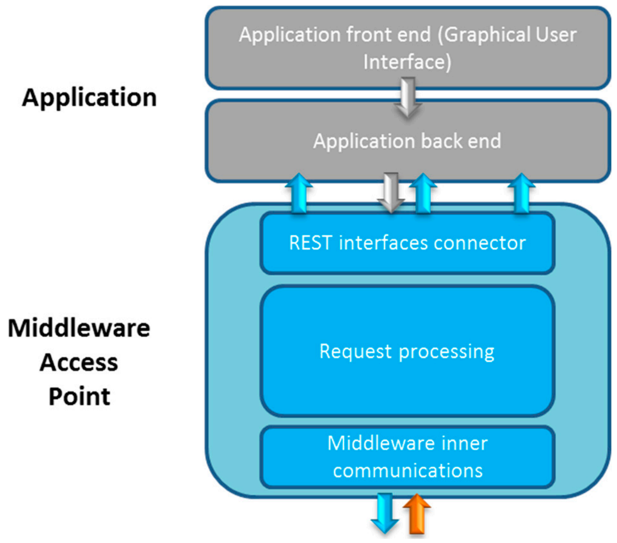
Application
Application front end (Graphical User Interface)
Application back end
Middleware Access Point
REST interfaces connector
Request processing
Middleware inner communications

Figure 4. Data request processing done in a Middleware Access Point.

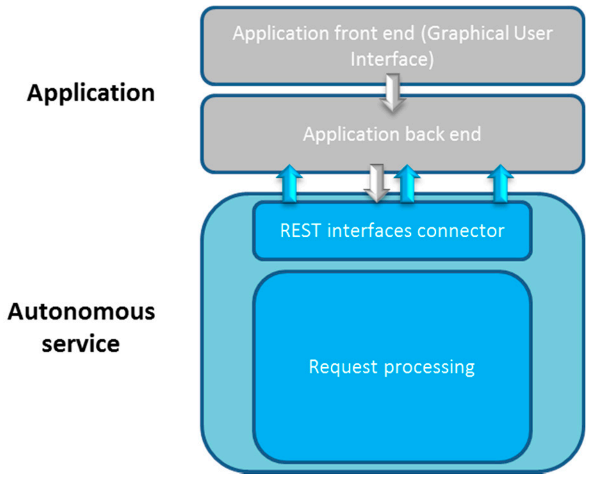
Application
Application front end (Graphical User Interface)
Application back end
Autonomous service
REST interfaces connector
Request processing

Figure 5. Data request processing done in an Autonomous Service.

The resulting output for this kind of services will be some data that will be sent to the application layer and will not further interact with the middleware services.

(3) *Context Awareness*: since this service is using inputs to provide its functionalities (application domain information, device availability, etc.) it will often interact with other components of the middleware. Consequently, the core of classes that is used to perform its duties will involve information gathering from all the features that context is made of. Data requests usually involve context awareness information to an extent as well. For example, in the e-GOTHAM project [16] context awareness information was used to determine whether a device that was going to be registered had already been registered before (so that redundancies in registration or masquerade attacks could be prevented). Furthermore, devices that were functional (along with their services) were checked in order to know if they had to be substituted by other device with similar functionalities. Another functionality that can be provided by context awareness is checking whether data requests are done to existing devices by enquiring device information to the semantic capabilities module. Again, in order to update the information regarding devices and services available, the context awareness module will be closely linked with the semantic capabilities one, since it will be sending information received from the present hardware of a smart grid deployment. In order to do so there will be a Lower level information collector that will be receiving the PDUs sent by the Hardware adaptors present in the lower layers of the architecture. In addition to that, there will be several connectors that will be used to send the information to other middleware components: one will be used to send it to high level Application Access Points (High level connector), to the Security component (Security connector) and to the semantic capabilities module (Semantic capabilities connector). The overall behaviour of the Context Awareness component regarding the other components, and a proposal of the classes that could be used to implement its functionalities, have been depicted in Figure 6. As the main functional requirement of this service, acting as the component that will be providing information of the environment where the solution is deployed can be cited as the most prominent one. The output obtained by this service will be information to be sent to the high level Middleware Access Points whenever there is an information request regarding Context Awareness, or any information received from lower layers regarding hardware devices or application domain parameters in order to have it stored (within the semantic capabilities connector) or to have it cyphered (as a task done by the Security component connector).

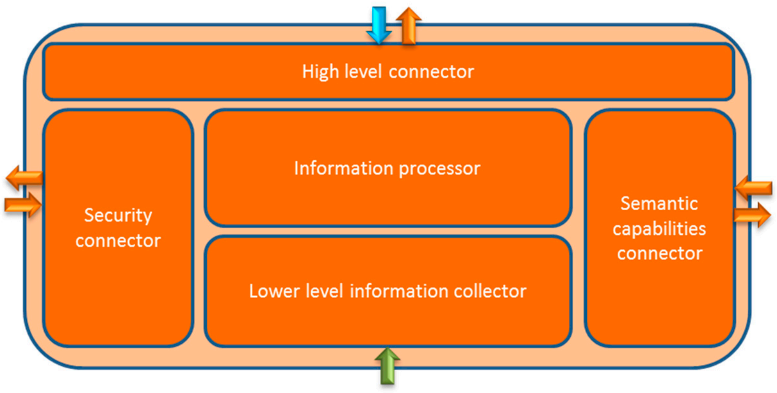
High level connector
Security connector
Information processor
Lower level information collector
Semantic capabilities connector

Figure 6. Data processing done in the Context Awareness module.

(4) *Security*: the functional requirement of this module is the encryption activities that will be done in the solution installed in the smart grid. Whenever cryptography activities are carried out, they will be done so in order to provide a service to either Context Awareness or the Semantic Capabilities module. In the illustration that is displayed in Figure 7, it is shown how both symmetric and asymmetric cryptography are available. It must be taken into account here what was mentioned previously regarding security: while asymmetric cryptography is generally better than symmetric cryptography, the latter is way faster, so advantages and disadvantages of each

methodology must be weighed before using one or the other depending on the services that are committed (security agreements, Quality of Service with a fixed degree of delay, etc.). Security implemented as a software module working within the middleware architecture was done with success in the I3RES project [17].

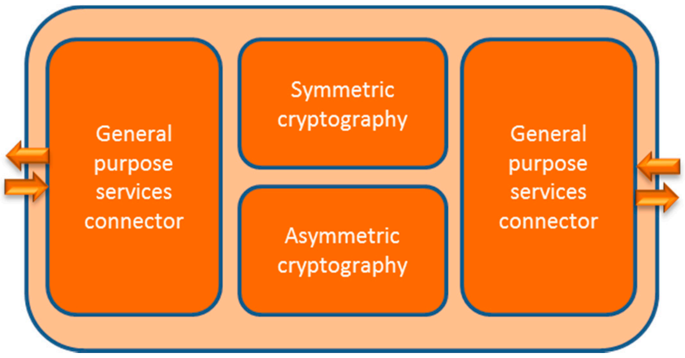
General purpose services connector
Symmetric cryptography
Asymmetric cryptography
General purpose services connector

Figure 7. Data processing done in the Security module.

The resulting output from this module will consist of a piece of information cyphered according to the algorithm used to cypher it or decipher it. If this information is to be finally collected by the application layer, either the application itself will have a functionality able to decrypt the message sent or it will be sent back to the Security module that will decrypt the data with its own set of keys.

(5) *Semantic capabilities*: this module can be used as the responsible for several major tasks or requirements: to begin with, it will collect the information received from lower layers so that it will be formatted to the CIM that is being used by the middleware architecture (which may or may not be the CIM used for power grids), as well as the registration functionalities required to store the information obtained not only from the devices (and what is more important from the SOA point of view, the services they are capable of offering) but also the information from the environment the different devices located as part of the smart grid provide, which will be essential for the Context Awareness module). Whatever the case, the information that is saved in this component is expected to be done so by (a) using a semantic database such as triple database or TDB [80] and (b) using the ontology that has been developed for the middleware solution as the way to interconnect with their own relationships all the elements that appear in the smart grid (which is an activity expected to be done by the Ontology-based Information Model formatter). In addition to that, the information to be stored here can make use of the security capabilities implemented in the middleware with the connector used for the security implementation module, as shown in Figure 8.

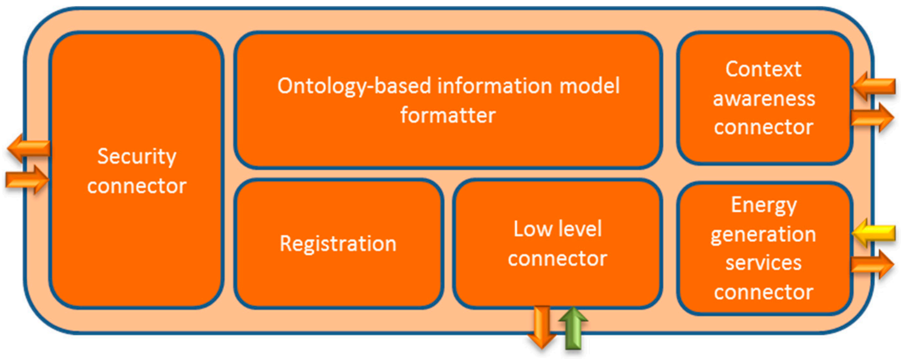
Security connector
Ontology-based information model formatter
Context awareness connector
Registration
Low level connector
Energy generation services connector

Figure 8. Data processing done in the Semantic Capabilities module.

The output resulting from this operation will be either stored information that will be used by the Context Awareness module, other higher level services and the Security module when information is sent to it.

(6) *Hardware abstraction layer*: this component will be interfacing the messages that are obtained from the lower level boundaries of the middleware architecture. The information that is sent throughout this layer will be done so according to a format defined previously as a set of protocol messages interchanged between the available infrastructure and the piece of equipment where the other components of the middleware architecture are running. Information received from the network layer will involve several different kinds of devices, but the information about them will be extracted in later stages by other middleware modules (basically, the Semantic Capabilities and the Context Awareness module). That is why, as portrayed in Figure 9, there are two more other functionalities that must be provided by the Hardware Abstraction layer: connectivity with the Context Awareness module (so that the information provided by the hardware devices and the application domain that is using those devices can be fed to the middleware architecture) and the Semantic Capabilities one (which will use the connector to send information back to the devices whenever it is necessary, such as a registration acknowledgement or an actuation message that has been triggered by the information that has been inferred by the semantics module). Overall, the main functional requirement of this service is the abstraction of all the hardware complexity that is underlying in the deployed system.

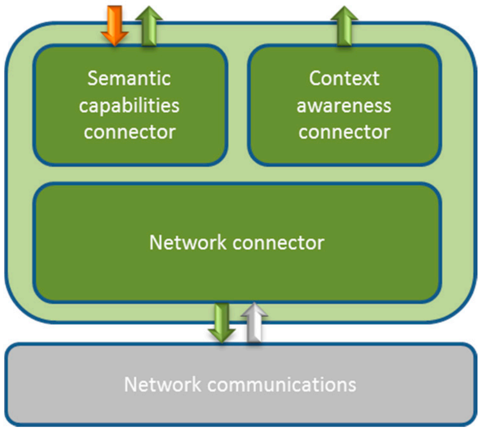

Figure 9. Data processing done in the Hardware adaptor module.

The output that will be used by this module will be the messages that are sent to higher layers of the middleware. If an actuation is required for the lower level devices, it will be done so with the instructions sent from the Semantic capabilities module.

(7) *Distributed energy generation services*: these services are more specific to the smart grid than any from the others that have been defined before. The structure that is used for each of the services is essentially the same one: a connector that will be used to interact with the high level middleware services will be used to send and receive information with the Middleware Access Points. The main difference among these services will be about the information that is sent; Demand side management will be dealing with parameters used for the long term change of power consumption customs among users (power consumption during a period of time, energy mix used during that period of time, etc.). Demand response, at the same time, will be doing the same in the short term. Last but not least, Optimal Power Flow information dealing with peak shavings or load balance will be sent and received throughout the middleware. The overall software structure of these services can be watched in Figure 10.

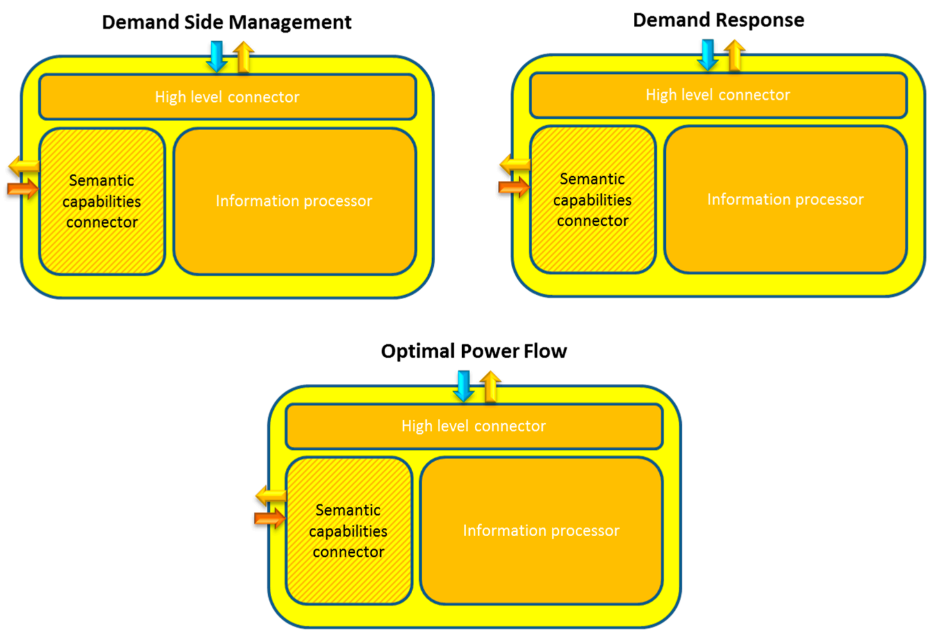

Figure 10. Data processing in distributed energy services.

In addition to the already presented facts to be taken into account for each of the software modules, there are some other aspects that must be taken into account:

(1) When developing a service, it could happen that there is already some existing code that can be reused in this case. While that is a plausible alternative, the modifications that have to be done to the inputs and outputs of the system may nullify the advantages of having already some development made, so it will have to be carefully weighed in case starting a development from scratch ends up saving more time that trying to tailor an existing solution to a set of requirements that impose severe differences with what was formerly used.

(2) Optimization must be heavily considered, especially since there are devices present in the smart grid that might not be that powerful and sending/receiving large amounts of data may have a negative impact in their overall performance. Smaller data types (for example, byte types) can take preference over other ones (for example, int or long) when codifying functionalities. Data representation formatting can be done with less verbose information representation languages (using a standard instead of a proprietary solution).

4.3. High Level Interfaces Procedures

In previous sections it was defined how REST can be used to grant access to the services that have been developed and deployed in the middleware solution. Furthermore, considering the low amount of software capabilities required (almost any kind of device is capable of using RESTful web services as long as they have either a web browser, a full OSI or TCP/IP stack implemented; smartphones, for instance, are able to run REST services with ease) and the simplicity of the required lines of code that have to be included, REST is a very appealing way to make visible the middleware services for an end user. Taking into account the service that is going to be accessed from the application layer, the REST-enabled method that will provide the key functionality (either using the high level software bundles as Middleware Access Points or Autonomous services) and the parameters that are required for its performance, it can be defined how services will be used for the benefit of the end users and, by proxy, what the URI that will be used to access the middleware will look like. In addition to that, it provides a very easy way to access the middleware development for the developers creating applications for the clients of the smart grid.

Under this scenario, each of the service requests that are done in the different application domains set for the functionalities that can receive queries according to a smart grid-based scenario will be answered with a collection of XML-formatted data that will be either visualized in a web browser or processed by an application. For example, in case of the query that is sent in order to get information from a device (that is to say, the URI that is remotely executed in a web browser) will be answered by offering a collection of data provided as an XML message, as shown below. To the best of the authors' knowledge, there is scarce information about how the XML messages used to access REST services in other middleware architectures, so the features that have been added here take into account the most relevant information required from a service and offer labels for information about readings, units of measurements and timestamps. These data will be offered as part of the information that is being gathered during the registration process, and can be based on the underlying semantics of the middleware architecture, as shown in the following message structure:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
   <head>
          <variable name="serviceId"/>
          <variable name="serviceType"/>
          <variable name="serviceFunctionality"/>
          <variable name="serviceState"/>
   </head>
   <results>
          <result>
                   <binding name="serviceId">
                   <uri>[SEMANTIC SERVICE ID DESCRIPTION]</uri>
                   </binding>
                   <binding name="serviceType">
                   <uri>[SEMANTIC SERVICE TYPE DESCRIPTION]</uri>
                   </binding>
                   <binding name="serviceFunctionality">
                   <uri>[SEMANTIC SERV. FUNCTIONALITY DESCR.]</uri>
                   </binding>
                   <binding name="serviceState">
                   <uri>[SEMANTIC SERVICE STATE DESCRIPTION]</uri>
                   </binding>
          </result>
  </results>
</sparql>
```

Some other data (and specifically, information related to real time capabilities) will be mainly provided as part of an XML message: for example, if the getConsumption operation is invoked, a message can be retrieved containing data about measurement units, figures and properties. Even a timestamp can be provided for its further processing (timeline-based graphs, etc.).

```
<?xml version="1.0" encoding="UTF-8"?>
<message xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <type>[DATA ANSWER IDENTIFIER]</type>
    <serviceID>[SERVCIE IDENTIFIER]</serviceID>
    <sensorReading>
            <timestamp>[TIMESTAMP VALUE]</timestamp>
            <property>
           <physicalQuality>[FEATURE MEA.]</physicalQuality>
            </property>
    <sensorOutput>
            <observationValue>
                  <quantityValue>[FEATURE FIGURE]</quantityValue>
                  <prefix>[MEASURE MULTIPLIER]</prefix>
                  <unitOfMeasure>[MEASURE UNITS]</unitOfMeasure>
            </observationValue>
    </sensorOutput>
    </sensorReading>
</message>
```

4.4. Low Level Interfaces Procedures

As far as the way to interact with low level interfaces is concerned, there must be a way used to exchange data with them that, at the same time, will leave enough resources available for any other kind of communications that are done between the network and the hardware devices. Considering the facilities that have to be provided by a middleware solution, low level interfaces will be responsible for hardware abstraction. Therefore, the XML message structure and components that are shown here are the ones used to abstract any hardware difference by means of implementing a data-centric way to transfer information, regardless of the nature of the devices. While other solutions could be used, such as tailored PDUs that would not necessarily be XML messages, this is regarded as the most effective procedure to deliver the information to the system ontology, as it has been implemented taking XML-friendly formats as the way to receive and display information. In order to do so, a protocol specifically used for data transfers among hardware components of the smart grid can be conceived. Solutions used with the aim of providing data connectivity at a level higher than the network one were designed and implemented in the e-GOTHAM and I3RES projects. Four different kinds of XML messages, which can be regarded as the PDUs of the protocol, have been defined to contain the required data for regular device registration, semantic device registration, registry response and context discovery. These PDUs are part of a protocol that can be named as a Smart Grid Data Interchange Protocol (SGDIP) and, from a layered point of view, will operate one level above the network layer and one below the application one, covering the data interchange functionalities required by the system. The different data fields in each of them are:

(1) *Regular device registration*: the PDU will require information regarding the kind of message that is being transferred, an identifier used to acknowledge some kind of unique piece of information used to determine the source of the data that is being sent (namely, the IP address of the device that is connected to the network) and a payload with the following information from the device: the manufacturer that is providing it, the model that has been included in the deployment, a serial number to uniquely identify it from the hardware point of view, and the device type that defines whether the device pre-existed the deployment where it has been included or has been developed for it (for example, as a result of a project) from scratch. Overall, the appearance of the PDU is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<message xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<type>REGULAR REGISTRATION REQUEST</type>
<transportId>[SOURCE IP ADDRESS]</transportId>
<payload>
        <deviceTypeId>
              <manufacturerId>[MANUFCT. NAME]</manufacturerId>
        <modelId>[MODEL NAME]</modelId>
              <serialNumber>[SERIAL NUMBER]</serialNumber>
              <deviceType>[EXISTING/SCRATCH]</deviceType>
        </deviceTypeId>
        <context>
              <location>
                      <latitude>[COORDINATE]</latitude>
              <longitude>[COORDINATE]</longitude>
        </location>
        </context>
</payload>
</message>
```

(2) *Semantic device registration*: the information contained in this case is more profuse than before, since more data are required to have a device registered according to the ontology that is implemented for the system requires a higher level of detail. Thus, new data have been added in this PDU: (a) its type refers to semantic rather than regular registration; (b) a new field identifying if the service that is going to be registered is already included in the middleware (middleware services were registered in e-GOTHAM and I3RES in a semantic manner) in order to use different kinds of data templates; (c) operation features of the service that becomes registered (name, type, input and output parameters, the numeric figure that is transferred as a result) and (d) the underlying protocol that is used for the registration (AMQP, JMS, etc.). The final appearance of the interchanged PDU is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<message xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<type>SEMANTIC REGISTRATION REQUEST</type>
   <templateId>[DEVICE OR SERVICE TYPE]</templateId>
   <serviceType>[DEVICE OR MIDDLEWARE BASED]</serviceType>
   <serviceFunctionality>[FUNCTIONALITY]</serviceFunctionality>
   <payload>
       <context>
           <location>
                <latitude>[COORDINATE]</latitude>
                <longitude>[COORDINATE]</longitude>
           </location>
           <deviceTypeId>
           <manufacturerId>[MAN.NAME]</manufacturerId>
                <modelId>[MODEL NAME]</modelId>
           <serialNumber>[SERIAL NUMBER]</serialNumber>
                 <deviceType>[EXIST./SCRATCH]</deviceType>
           </deviceTypeId>
       </context>
       <operations>
           <operation>
                  <name>[OPERATION NAME]</name>
                  <type>[OPERATION TYPE]</type>
                  <description>[OPER. DESCRP.]</description>
                  <parameters>
                  <parameter>
                  <inputParameter>[INPUT]</inputParameter>
                      <typeInput>[TYPE]</typeInput>
                  </parameter>
                  <parameter>
                  <outputParameter>[FIG.]</outputParameter>
                  <typeOutput>[TYPE]</typeOutput>
                  </parameter>
                  </parameters>
           </operation>
       </operations>
       <grounding>
           <groundingProtocol>[INF. PR.]</groundingProtocol>
       </grounding>
   </payload>
</message>
```

(3) *Registration response*: this is a PDU that will be sent to the device or service that requested to be registered in case it was successful. It contains the type of PDU, a token of the registration result (Y in case it worked as expected) and a service identifier to be used by the entity that started the communication. The same kind of PDU is sent for both kinds of registration procedures, as it is not necessary to send semantic data about the registration procedure back to the device or service that request it (that information would not be used for the registration procedure because it has already being consumed).

```
<?xml version="1.0" encoding="UTF-8"?>
<message xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<type>REGISTRATION RESPONSE</type>
<payload>
       <registered>Y</registered>
       <serviceId>[SERVICE ID CODE]</serviceId>
    </payload>
</message>
```

In case the registration procedure failed a slightly different PDU will be sent, containing "N" as the registration result, an error code and a description of the code that prevented the correct registration of the device.

```
<?xml version="1.0" encoding="UTF-8"?>
<message xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <type>REGISTRATION RESPONSE</type>
    <payload>
          <registered>N</registered>
          <errorCode>[ERROR CODE]</errorCode>
<errorDescription>[ERROR DESCRIPTION]</errorDescription>
    </payload>
</message>
```

(4) *Context discovery information*: this message will be used to check the current status of the devices and services that are expected to be used in the smart grid. Whenever a device becomes registered it will sent a message like this as a "keep alive" notification. It basically contains the kind of message that it is and the service identifier that was sent back to the device that got registered.

```
<message>
<type>[CONTEXT DISCOVERY TYPE]</type>
<payload>
      <serviceId>[SERVICE ID CODE]</serviceId>
</payload>
</message>
```

Whenever there is any kind of information interchange regarding registration it will be done in the following fashion: when a device starts the registration process, it sends a registration message (which may or may not be semantically annotated, depending on the capabilities of the device). When that message is received by the middleware architecture, it is verified with an XMLSchema to ensure that no information has been altered or corrupted while it was sent. Afterwards, another XML document with all the required information for the semantic registration will be sent to the semantic repository (which will usually correspond to a semantic database) and the information about the registered device is stored. If the procedure was fine, a success confirmation message will be sent; failure messages will be sent back if an issue came up after registration. Keep alive messages will be transferred from the registered entities to the registration module so that the middleware will be aware of the available devices (and their services) that can be used at any moment. At this point, all the inner functionalities of the middleware have been codified and deployed. In addition to that, there is seamless connectivity between the application and the hardware-based layers, so middleware is effectively providing an API of available interfaces that withholds the complexity of the deployed hardware devices in a smart grid. The last step remaining is providing the necessary measures for the middleware deployment in order to make it usable under a secure environment. As far as the implementation and testing of this set of messages is concerned, it was done by means of XML messages codified as part of Java classes that contain not only the messages themselves, but also the required methods to send, receive and manage the information.

4.5. Securitization Procedures

The security procedures that have been enabled for the middleware architecture have been bent on two symbiotic actions: on the one hand, securitization procedures have been applied to the interfaces in order to guarantee that the middleware access is done in a secure manner by trusty users. On the other hand, cryptography has been used here in order to provide security services that are regarded as essential for data transfers that involve sensitive data. Consequently, it will become necessary for any client to have a certificate to use the application that becomes connected to the REST interfaces provided by the middleware architecture. The overall steps that have to be undertaken in order to provide security in this environment are as follows:

(1) When a piece of equipment where the middleware is installed is expected to be made secure regarding the access from the application layer, an HTTP secure server will have to be installed, as the piece of equipment will effectively become the server under a client/server paradigm. Apache HTTP server can be used for this purpose under a machine assuming the role of a server [81]. By enabling the Apache HTTP server, a SSL site will be created.
(2) Redirection for users accessing the middleware from the non-securitized to the securitized site must be done afterwards. In order to do so, a redirect directive within the non-securitized VirtualHost can be added to the default configuration file of the server.
(3) Once the piece of equipment with the middleware installed has been securitized, certificates will have to be created for the user. Also, depending on the pre-installed facilities of the piece of equipment or its current usage, it might be required to have it as a self-signed certificate authority. In order to create the required certificates for the remote end users, a particular location must be created in the certification authority to gather all the information about the legitimate users. At the same time, the elements that are going to take part will create a user key and a user certificate request will be created, which will be signed in order to have the Certificate Authority creating the certificate.
(4) The Certificate Signing Request (CSR) will be sent and it will be verified at the Certificate Authority. If everything was correct then the Certificate Signing Request will be signed with the Certificate Authority key.
(5) A signed server/client certificate will be sent to the server/client side of the communication (while client and server are interested parties in the data transfers, the Certificate Authority may not be part of it; its main role will be the verification of the CSR and its signing with its own key). It will be used during data transfers from that point on.

As for the securitization that can be done inside the middleware architecture, symmetric or asymmetric encryption facilities can be used, so both of them should be offered. One of the ways to develop these required functionalities is by means of the usage of the Bouncy Castle Crypto APIs libraries [82] which are cost-free and become updated with certain regularity. Therefore, they can be used as a way to provide security on the information transferred with an acceptable level of privacy without having to take budget limitations into account. In addition to that, Bouncy Castle API enjoys a strong online support, so it is easy to find support whenever there is an implementation issue. When all is said and done, the different stages that have been mentioned in this manuscript are built upon the contributions made by several steps. There are cases when one single step will not provide enough input, though. In that case, two previous contributions will be required, as stated in Table 4.

Table 4. Middleware architecture development progression.

Step	Input	Output
Obtaining an open source Enterprise Service Bus	Machines with a functional operating system	Open source Enterprise Service Bus installed and running with the default components
Development of middleware protocol or message collection	Collection of data and/or information that is going to be sent through the middleware architecture and the overall system	Messaging system that is going to be used for the common information model employed in the architecture, either taken from an already existing standard or tailored from scratch
Development of software bundles	Open source Enterprise Service Bus installed and running with the default components	Software services developed according to the reference architecture provided
Interconnectivity of software bundles	Software services developed according to the reference architecture provided	Software bundles cooperating with each other in order to deliver functionalities and services
	Messaging system that is going to be used for the common information model employed in the architecture	
High level interfaces	Software bundles cooperating with each other in order to deliver functionalities and services	Application-related interfaces capable of sending messages throughout tall the middleware inner layers and components
Low level interfaces	Application-related interfaces capable of sending messages throughout tall the middleware inner layers and components	Network and hardware-related interfaces able to send and receive messages from outer devices
Securitization procedures	Testing of all the functionalities is done so that their actual performance is checked	The final version of the middleware solution added to the smart grid is tested. Any minor bug is fixed

5. Validation Testbeds

The implementation works that have been mentioned in this set of procedures were used as the underlying procedures that enabled the implementation of semantic middleware solutions that, although were different in terms of the stressed areas of interest (as the e-GOTHAM project was more focused on Information and Communication Technologies and I3RES was mostly about electricity and load management), shared many common needs (interoperability hardware abstraction, etc.). Testing activities were a way to ensure that the future prototype to be installed could be used as a regular-working development in a smart grid environment. The access to the middleware services was remotely enabled in the laboratory where it was firstly implemented, and once the sets of tests that were run provided significant and positive feedback the developed middleware architecture was deployed in the city-living lab of Steinkjer, Norway [83], were further tests and exploitation were made in the context of an actual smart grid application domain. A similar procedure was carried out in the Finnish town of Ylivieska for the demonstration of the e-GOTHAM project [84]. By describing the validation procedures that have been carried out with the middleware that has been developed under the directives of the guidelines shown in this manuscript, the usability of the ideas put forward here can be proven right.

5.1. Testing Activities on the Middleware Solution: Laboratory Development

Testing activities varied depending on the project that was dealt with, due to the fact that the developed middleware had different objectives (for e-GOTHAM, middleware main purpose was about interconnecting hardware devices of a microgrid, whereas in I3RES was using the data collected to manage electricity consumption in a similar deployment). In e-GOTHAM, test benches were carried out in several locations of the middleware:

- High level services: there were several high level services (used for device registration or more frequently for support activities) that were tested regarding how they could be accessed by means of the implemented REST interfaces whenever they were requested, or if access was denied in case an unexpected request was done. Tests were considered as successful when either services could be accessed by previously registered simulated devices or could not be requested if the enquiry was sent by a non-registered simulated device. Registration of thirteen different devices and facilities (temperature and humidity and current meters, PMUs, real-time database, etc.) were made. These were simulated in the premises of the Technical University of Madrid by using the XML template files that should be sent by them for their registration.
- General Purpose services and Distributed Energy services: other tests that were carried out regarding services were about their registration. XML messages about each of the capabilities if the services that were located between the lower layer and the high level were sent.

All the required equipment that was involved in the middleware architecture (Apache server –Provided by the Apache Software Foundation, located in Forest Hill, MD, USA, ESB architecture, etc.) was installed in an Ubuntu-based virtual machine running the 14.04 Long Time Support (LTS) distribution of this Linux-based operating system. Tests were mimicked in facilities located in Ylivieska as well; the hardware capabilities required for the middleware to be installed were quite modest: the equipment where most of the middleware components were present had a 3 GHz CPU, 4 GB of RAM memory, 160 GB of Hard Disk and Gigabit Ethernet communications were used. This was the hardware that was used as the Central Controller in the Ylivieska deployment, too. As it will be described later, the Local Controllers were based on Raspberry Pi boards. As far as the I3RES testing activities are concerned, the first stage of the testing procedures was carried out in the same controlled environment in order to test the worst case scenarios and possible bugs. To do so, a machine enabled with remote access was provided. The machine was connected to the Internet and had an IP address given so that it could be accessed with the clients that already had the security certificates created with the procedure that has been formerly explained. The IP address that was given for regular connectivity was mapped to the one that was used for the masquerade configuration of the Technical University of Madrid, and finally to the one that was used as the one involved in the private network where the machine became connected. As it happened with the former project, the same kind of operating system was used to install the major part of the middleware components. The procedures that were used in order to know the performance of the middleware architecture in this environment involved testing the access to the REST interfaces that were mentioned in the previous section. The scenarios faced were as follows:

(1) When tests were successful, information would be retrieved without any problem throughout the requests done by means of the REST interfaces, which were used as Middleware Access Points.
(2) Should there be any kind of issue (commonly, exceptions disrupting data retrieval or inability for the end users to connect to the machine remotely) performance of the system would be reviewed. Most of the time the improvement works consisted of providing some XML-formatted information about unexpected usages of the middleware architecture that could be comprehended either by applications or end users.

While the middleware components that have been deployed were complex, the computer where they were deployed was of regular characteristics (it was a regular personal computer with 8 Gigabytes of RAM memory), so a middleware architecture can be deployed in any kind of regular environment rather than requiring any different or specialized one (usually, other kinds of hardware devices, such as PMUs or RTUs are way more limiting in a microgrid deployment rather than the software components of the middleware). Resources required to run middleware in a successful manner do not impose severe needs or strains for hardware equipment. Other tests that were performed as part of the I3RES project activities involved validating the connectivity between the elements that were supposed to be interconnected by the middleware solution (both hardware devices and software applications) and the middleware server where the software components were running. Applications and monitoring infrastructure were proven to be connected and data were retrieved from devices connected via middleware. The information obtained was done so with the same pattern that has been shown here: an XML file containing device names, type and a timestamp.

5.2. Scenarios for the Deployment

The principles that have been put forward in this manuscript have come as a result of the different software developments done for two scenarios. In case of the e-GOTHAM project, the Finish town of Ylivieska was used as a place to deploy a semantic middleware solution that was also made possible via open source ESB [40]. It was used as an intermediate entity between the application services—consisting of a web application or a consumption forecast component—and lower level components using an Apache Qpid broker (provided by the Apache Software Foundation) (common in AMQP works) to send and receive information from the hardware components (smart meters) that had been deployed in major public buildings of the town. The deployment was done with a mixture of centralized and decentralized features: on the one hand, there were several buildings in the city (concert hall, city school, elderly home, sports centre, a power plant, a terraced house, an elementary school and an industrial building), where each of them has been equipped with a Local Controller; these are pieces of equipment that collected the data harvested by Open Energy Monitor [85] with regards to power consumption. These Local Controllers were built by using Raspberry Pi model B modules that have enough capabilities (a CPU of 700 MHz, 512 MB of RAM memory, etc.) to collect and transfer the data that were gathered [86]. As shown in [84], eight of them were deployed; in some cases, such as the power plant that was monitored by the system, more than one was used. Some other buildings, such as the elementary school and the industry building, used analogic pressure sensors that had to be connected to Analog-to-Digital Converters so that their provided signals could be better used.

Apart from services required for the inner performance of the Central Controller, the ESB iteration that was used here contained two interfaces used to interact with all those devices connected to the physical facilities involved in the pilot. One interface was based on REST and was used to implement HTTP requests; a SQL database and an external consumption forecast interacted by means of this interface. The Qpid broker was used to interact with the Local Controllers installed in the other facilities. Rather than performing specific tests on the middleware solution, solutions that were running on top of it, but using the interoperability capabilities of the middleware: for instance, the consumption forecast service was validated in this pilot partially using an energy analysis technique [87]. As mentioned, Steinkjer was also the place of choice for the deployment of middleware in the I3RES project. The middleware architecture elements included in the deployed were matching the ones that have been shown here and, to a great extent, the ones used in the Ylivieska pilot:

(1) An ESB containing software services that were implemented as bundles that could be ported from one deployment or another, especially if they were autonomous services.
(2) Elements such as a semantically enabled registration of hardware by means by a Hardware Abstraction layer document and an Ontology.
(3) Higher level services that were connected to applications that were running outside the architecture but required their information, such as Graphical User Interfaces.
(4) Securitization was enabled in the interfaces as a way to guarantee that the implemented services were accessed only to individuals that had been previously been granted access.

All these elements where integrated in the living lab of Steinkjer and have been installed since the end of the I3RES project. Tests were carried out to measure the performance of the middleware solution, as it has been described in [12]. A total of 1000 measurements were taken in order to obtain accurate enough results. The output that was obtained has been displayed in Figure 11.

According to what was presented in [12], the performance results are as follows:

(1) Average value (blue line) measures the average time required to satisfy an inquiry done through the middleware. It was measured as 1602 ms.
(2) Deviation value (red line) measures the regularity of the obtained performance values, which can only be hinted considering the difference between the average and median figures. It was measured as 364 ms.
(3) Throughout (green line) is utilized to measure the requests that can be handled during a certain timespan. That figure was measured as 148.543 requests per minute.
(4) Median (purple line) measures were the second quartile of the taken measurements ends. It was measured as 1582 ms.

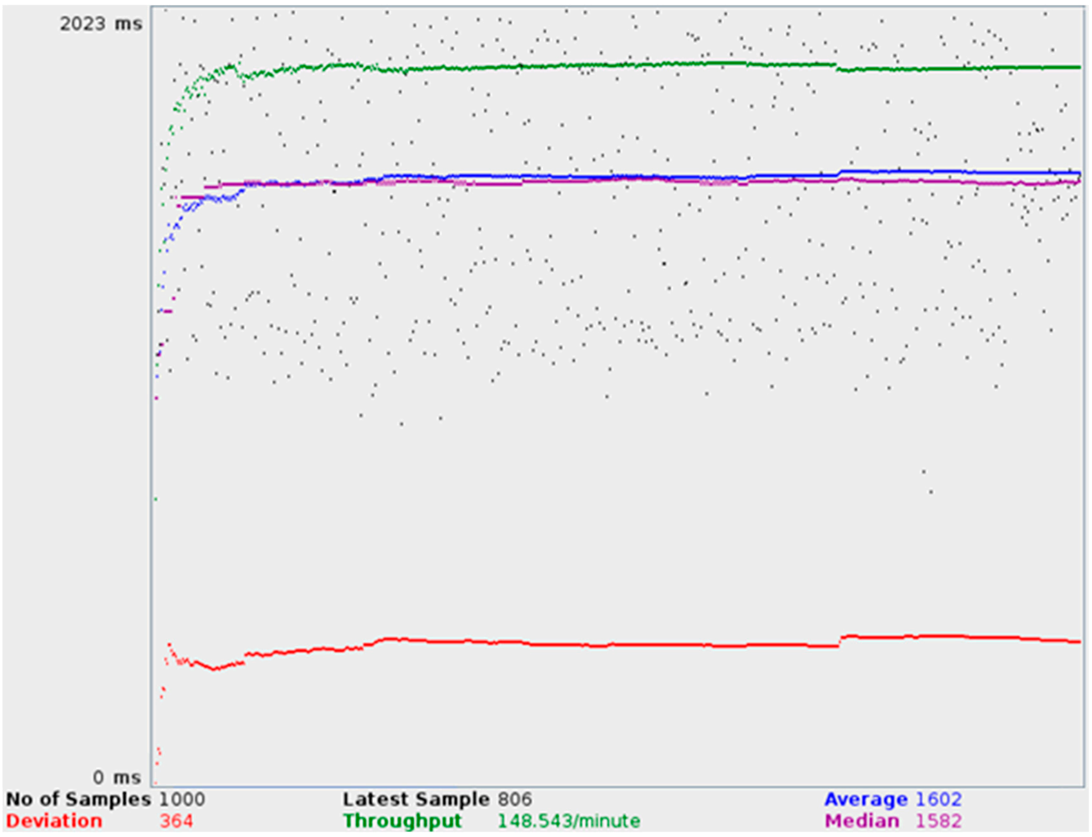

Figure 11. Performance results obtained in middleware tests, as shown in [12].

5.3. Lessons Learnt from the Deployments

From what has been described in this manuscript, it can be inferred that there are several actions to be taken in a specific order to have a middleware solution working in a smart grid-related environment. While the software development procedure can be tackled by following already known methodologies, the implementation works that have to be carried out have to be done in a very precise order that will avoid negative consequences, such as wasting time or economic resources:

(1) The framework has to be set according to the needs of the distributed system that is going to be implemented. As it has already been settled, open source ESB architectures are a good way to solve many issues regarding device interoperability and resource provisioning. The possibility of using open source or (if it is required due to industry agreements) privative licenses must be evaluated to find a solution that will guarantee the satisfaction of all the parties involved in the development or the software architecture.
(2) Software development will be focused on three objectives: codifying the necessary services, wrapping them as the kind of software project that is required for the architecture and interconnecting them so that they will be able to offer the expected facilities to the end user in a seamless manner.
(3) High level interfacing must be provided as a way to connect the applications to the middleware that is being developed.
(4) Low level interfacing must be performed with the idea of gathering the essential information from the services that are going to be used in the deployment. Therefore, protocols used to collect information, as well as internal protocols used to transfer the data from one component to another one depending on the needs of the service, must be designed or reused, in case there is already a solution that is matching the requirements of the system that has been designed.
(5) Securitization services must be provided as well so that the developed system will be trusted and can be used by as many parties as possible.

6. Conclusions and Recommendations for Future Work

This manuscript offers a plethora of contributions that can be summarized as the following ones: a justification has been offered on how to choose a framework to develop this middleware architecture with the aims of having it done at the lowest cost and the simplest way possible to have it developed. Furthermore, steps regarding implementation have also been offered. How to use an open source ESB solution (JBoss ESB), as well as how to codify software bundles that will be cooperating with each other has been explained as well. Moreover, in order to show the feasibility of the presented ideas, middleware implementations developed according to the principles are shown. Both the services that were implemented for them and the ones that were using the middleware as a wrapper for their own smart grid-based functionalities were successfully tested. The general model described here enforces the idea of having a collection of high level and low level middleware services, and a set of general purpose software services that will change very little from one development to a different one. These middleware components are a significant part of the normalization procedures shown in this manuscript. Taking all these contributions into account, the development of a middleware architecture for the smart grid should become easier and faster, as future implementations will be built up in the knowledge that is offered in this manuscript. Furthermore, there are some future works that can also be done to improve the solution presented:

(1) Standardization procedures can be extended with regards to the usage of a middleware inner protocol for the smart grid. The solution presented here can be expanded to deal with other application domains outside the smart grid.
(2) Hardware adaptors can be tailored for the specific kinds of appliances that are present in the smart grid. A higher level of detail could be used to define how hardware abstraction functionalities can be standardized for PMUs, RTUs, etc.

Security solutions used to encrypt and decrypt data can be added in order to have a wider variety of algorithms present in the middleware architecture.

Acknowledgments: The research works presented here have been done as part of I3RES (ICT-based intelligent management of integrated RES for the Smart Grid optimal operation, FP7, reference No. 318184)) and e-GOTHAM ("Sustainable—Smart Grid Open System for the Aggregated Control, Monitoring and Management of Energy", ART Call 2011 295378) projects. The research group integrated in the Technical University of Madrid (UPM) that has performed the tasks related to the development of this middleware architecture is called Grupo de Redes y Servicios de próxima generación (GRyS [88]), with its facilities located at CITSEM (Centro de Investigación en Tecnologías Software y Sistemas Multimedia para la Sostenibilidad [89]).

Author Contributions: Jesús Rodríguez-Molina developed the methodology and implementation procedures. José-Fernán Martínez assessed the content and provided contributions for the testing section. Pedro Castillejo focused on the justification of the components that are part of the solution. Gregorio Rubio created the introduction, followed the development of the sections and put forward conclusions and future works.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Atkeson, A.; Kehoe, P.J. *The Transition to a New Economy after the Second Industrial Revolution*; National Bureau of Economic Research: Cambridge, MA, USA, 2001.
2. Coulouris, G.F.; Dollimore, J.; Kindberg, T. *Distributed Systems Concepts and Design*; Addison-Wesley: Reading, MA, USA, 2012.
3. International Organization for Standardization/International Electrotechnical Commission (ISO/IEC). *ISO/IEC 7498-l: Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model*, 2nd ed.; ISO/IEC: Geneva, Switzerland, 1994.
4. Network Working Group Internet Engineering Task Force. *Requirements for Internet Hosts—Communication Layers, RFC 1122*; Internet Engineering Task Force (IETF): Fremont, CA, USA, 1989.
5. Berners-Lee, T. *Information Management: A Proposal*; European Organization for Nuclear Research (CERN): Geneva, Switzerland, 1989.
6. International Electrotechnical Commission. Smart Grid, Optimal Electricity Delivery. 2016. Available online: http://www.iec.ch/smartgrid/ (accessed on 30 September 2016).
7. Zachar, M.; Daoutidis, P. Microgrid/Macrogrid energy exchange: A novel market structure and stochastic scheduling. *IEEE Trans. Smart Grid* **2016**, *8*, 178–189. [CrossRef]
8. Wang, Y.; Mao, S.R.; Nelms, M. On Hierarchical power scheduling for the macrogrid and cooperative microgrids. *IEEE Trans. Ind. Inform.* **2015**, *11*, 1574–1584. [CrossRef]
9. Nordman, B. Nanogrids: Evolving our electricity systems from the bottom up. In Proceedings of the Darnell Green Building Power Forum, San José, CA, USA, 25–27 January 2010.
10. EPEX SPOT. European Power Exchange. 2016. Available online: https://www.epexspot.com/en/ (accessed on 7 December 2016).
11. European Energy Exchange AG. Available online: https://www.eex.com/en/ (accessed on 7 December 2016).
12. Rodríguez-Molina, J.; Martínez, J.-F.; Castillejo, P. A study on applicability of distributed energy generation, storage and consumption within small scale facilities. *Energies* **2016**, *9*, 745. [CrossRef]
13. Cai, Y.; Huang, T.; Bompard, E.; Cao, Y.; Li, Y. Self-sustainable community of electricity prosumers in the emerging distribution system. *IEEE Trans. Smart Grid* **2016**. [CrossRef]
14. Liu, H.; Ji, Y.; Zhuang, H.; Wu, H. Multi-objective dynamic economic dispatch of microgrid systems including vehicle-to-grid. *Energies* **2015**, *8*, 4476–4495. [CrossRef]
15. Rodríguez-Molina, J.; Martínez-Núñez, M.; Martínez, J.F.; Pérez-Aguiar, W. Business models in the smart grid: Challenges, opportunities and proposals for prosumer profitability. *Energies* **2014**, *7*, 6142–6171. [CrossRef]
16. e-GOTHAM Consortium. e-GOTHAM: Sustainable-Smart Grid Open System for the Aggregated Control, Monitoring and Management of Energy. 2012. Available online: http://www.e-gotham.eu/ (accessed on 4 May 2016).
17. CITSEM Web Manager I3RES—ICT Based Intelligent Management of Integrated RES for the Smart Grid Optimal Operation. 2012. Available online: https://www.citsem.upm.es/index.php/es/proyectos-es?view=project&task=show&id=37 (accessed on 4 May 2016).
18. ISO/IEC/IEEE 42010 Users Group. ISO/IEC/IEEE 42010 Website. Available online: http://www.iso-architecture.org/ieee-1471/interest.html (accessed on 7 December 2016).

19. OpenADR Alliance, OpenADR 2.0 Profile Specification B Profile. 2015. Available online: http://www.openadr.org/specification (accessed on 7 December 2016). (only for registered memebers).
20. International Organization for Standardization (ISO). *Win the Energy Challenge with ISO 50001*; ISO: Geneva, Switzerland, 2011.
21. Organization for the Advancement of Structured Information Standards (OASIS). *OASIS Energy Interoperation Version 1.0*; Considine, T., Ed.; OASIS: Burlington, MA, USA, 2014.
22. ISO. Energy Management Systems—Requirements with Guidance for Use. 2011. Available online: https://www.iso.org/obp/ui/#iso:std:iso:50001:ed-1:v1:en (accessed on 4 January 2017).
23. Sučić, S.; Havelka, J.G.; Dragičević, T. A device-level service-oriented middleware platform for self-manageable DC microgrid applications utilizing semantic-enabled distributed energy resources. *Int. J. Electr. Power Energy Syst.* **2014**, *54*, 576–588. [CrossRef]
24. Adamiak, M.; Baigent, D.; Mackiewicz, R. *IEC 61850 Communication Networks and Systems In Substations: An Overview for Users*. 2009. Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.698.1497&rep=rep1&type=pdf (accessed on 7 December 2016).
25. Shi, K.; Bi, Y.; Jiang, L. Middleware-based implementation of smart micro-grid monitoring using data distribution service over IP networks. In Proceedings of the 2014 49th International Universities Power Engineering Conference (UPEC), Cluj-Napoca, Romania, 2–5 September 2014.
26. Object Management Group. What's in the DDS Standard? 2016. Available online: http://portals.omg.org/dds/omg-dds-standard/ (accessed on 4 November 2016).
27. PrismTech. DDS Community. Available online: http://www.prismtech.com/dds-community (accessed on 4 November 2016).
28. Open DDS Community. OpenDDS. Available online: http://opendds.org/ (accessed on 4 November 2016).
29. Twin Oaks Computing, Inc. CoreDX DDS Data Distribution Service Middleware. Available online: http://www.twinoakscomputing.com/coredx (accessed on 4 November 2016).
30. Losa, J.M. Introduction to DDS. Available online: http://www.eprosima.com/index.php/resources-all/dds-all (accessed on 4 November 2016).
31. Kouluri, M.K.; Pandey, R.K. Intelligent agent based micro grid control. In Proceedings of the 2011 2nd International Conference on Intelligent Agent & Multi-Agent Systems, Chennai, India, 7–9 September 2011.
32. Allison, M.; Morris, K.A.; Costa, F.M.; Clarke, P.J. Synthesizing interpreted domain-specific models to manage smart microgrids. *J. Syst. Softw.* **2014**, *96*, 172–193. [CrossRef]
33. Morris, K.A.; Allison, M.; Costa, F.M.; Wei, J.; Clarke, P.J. An adaptive middleware design to support the dynamic interpretation of domain-specific models. *J. Syst. Softw.* **2015**, *62*, 21–41. [CrossRef]
34. Martínez, J.F.; Rodríguez-Molina, J.; Castillejo, P.; De Diego, R. Middleware Architectures for the Smart Grid: Survey and Challenges in the Foreseeable Future. *Energies* **2013**, *6*, 3593–3621. [CrossRef]
35. Gjermundrod, H.; Bakken, D.E.; Hauser, C.H.; Bose, A. GridStat: A flexible QoS-managed data dissemination framework for the power grid. *IEEE Trans. Power Deliv.* **2009**, *24*, 136–143. [CrossRef]
36. Awad, A.; German, R. Self-organizing smart grid services. In Proceedings of the 2012 Sixth International Conference on Next Generation Mobile Applications, Services and Technologies, Paris, France, 12–14 September 2012.
37. Lee, J.; Kim, Y.; Hahn, J.; Seo, H. Customer energy management platform in the smart grid. In Proceedings of the 2012 14th Asia-Pacific Network Operations and Management Symposium (APNOMS), Seoul, Korea, 25–27 September 2012.
38. Oliveira, J.P.C.; Rodrigues, A.W.D.O.; Sá, R.C.; Araújo, P.; de Araújo, A.L.C. Smart middleware device for smart grid integration. In Proceedings of the 2015 IEEE 24th International Symposium on Industrial Electronics (ISIE), Rio de Janeiro, Brazil, 3–5 June 2015.
39. Majdalawieh, M.; Parisi-Presicce, F.; Wijesekera, D. DNPSec: Distributed Network Protocol Version 3 (DNP3) Security Framework. In *Advances in Computer, Information, and Systems Sciences, and Engineering: Proceedings of IETA 2005, TeNe 2005, EIAE 2005*; Elleithy, K., Sobh, T., Mahmood, A., Iskander, M., Karim, M., Eds.; Springer: Dordrecht, The Netherlands, 2006; pp. 227–234.
40. Hoefling, M.; Heimgaertner, F.; Menth, M.; Katsaros, K.V.; Romano, P.; Zanni, L.; Kamel, G. Enabling resilient smart grid communication over the information-centric C-DAX middleware. In Proceedings of the 2015 International Conference and Workshops on Networked Systems (NetSys), Cottbus, Germany, 9–12 March 2015.

41. Consortium, C.-D. Cyber-Secure Data and Control Cloud for Power Grids. 2012. Available online: http://www.cdax.eu/ (accessed on 30 September 2016).
42. Martin, S.; Hernandez, J.; Valmaseda, C. A novel middleware for smart grid data exchange towards the energy efficiency in buildings. In Proceedings of the 2015 International Conference and Workshops on Networked Systems (NetSys), Cottbus, Germany, 9–12 March 2015.
43. Leménager, F.; Joannic, C.; Soriano, R.; Bachiller Prieto, R.; Monferrer, M.A.; Espejo Portero, N.; Mosshammer, R. Assessment and outlook of the OpenNode smart grid architecture. In Proceedings of the 22nd International Conference and Exhibition on Electricity Distribution (CIRED 2013), Stockholm, Sweden, 10–13 June 2013.
44. Consortium, O. The OpenNode Project. 2012. Available online: http://opennode.atosresearch.eu/ (accessed on 30 September 2016).
45. Iqbal, M.; Rizwan, M. Application of 80/20 rule in software engineering Waterfall Model. In Proceedings of the 2009 International Conference on Information and Communication Technologies, Doha, Qatar, 17–19 April 2009.
46. Trivedi, P.; Sharma, A. A comparative study between iterative waterfall and incremental software development life cycle model for optimizing the resources using computer simulation. In Proceedings of the 2013 2nd International Conference on Information Management in the Knowledge Economy, Patiala, India, 19–20 December 2013.
47. International Organization for Standardization (ISO). *ISO 9001:2015: Quality Management Systems—Requirements*; ISO: Geneva, Switzerland, 2015.
48. International Organization for Standardization (ISO). *ISO 14001:2015: Environmental Management Systems—Requirements with Guidance for Use*; ISO: Geneva, Switzerland, 2015.
49. GitHub, Inc. GitHub: How People Build Software. 2017. Available online: https://github.com/ (accessed on 30 September 2016).
50. Lembo, G.D.; Agnetta, V.; Fiorenza, G. Integration of DSO control systems and TSO automatic load shedding system to improve the security of the national grid. In Proceedings of the CIRED 2009—The 20th International Conference and Exhibition on Electricity Distribution—Part 2, Prague, Czech, 8–11 June 2009.
51. Gong, X.; Long, H.; Dong, F.; Yao, Q. Cooperative security communications design with imperfect channel state information in wireless sensor networks. *IET Wirel. Sens. Syst.* **2016**, *6*, 35–41. [CrossRef]
52. Psiuk, M.; Bujok, T.; Zieliński, K. Enterprise service bus monitoring framework for SOA systems. *IEEE Trans. Serv. Comput.* **2012**, *5*, 450–466. [CrossRef]
53. MuleSoft, Inc. What Is Mule ESB? 2016. Available online: https://www.mulesoft.com/resources/esb/what-mule-esb (accessed on 18 November 2016).
54. Petals Community. Petals Technical Overview. Available online: https://doc.petalslink.com/display/petalsesb/Petals+Technical+Overview (accessed on 18 November 2016).
55. Curry, E. Increasing MOM flexibility with portable rule bases. *IEEE Internet Comput.* **2006**, *10*, 26–32. [CrossRef]
56. Rodríguez-Molina, J.; Martínez, J.F.; Castillejo, P.; López, L. Combining wireless sensor networks and semantic middleware for an internet of things-based sportsman/woman monitoring application. *Sensors* **2013**, *13*, 1787–1835. [CrossRef] [PubMed]
57. Fielding, R.T. Architectural Styles and the Design of Network-based Software Architectures. Ph.D. Thesis, University of California, Irvine, CA, USA, 2000.
58. The Internet Society HTTP over TLS. 2000. Available online: https://tools.ietf.org/pdf/rfc2818.pdf (accessed on 8 December 2016).
59. Rivest, R.L.; Shamir, A.; Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* **1978**, *21*, 120–126. [CrossRef]
60. Miller, F.P.; Vandome, A.F.; McBrewster, J. *Advanced Encryption Standard*; Alpha Press: Orlando, FL, USA, 2009.
61. Diffie, W.; Hellman, M. New directions in cryptography. *IEEE Trans. Inf. Theory* **1976**, *22*, 644–654. [CrossRef]
62. AMQP Community Members. AMQP 1.0 Becomes OASIS Standard 2012. Available online: http://www.amqp.org/node/102 (accessed on 28 March 2016).
63. Network Working Group. Simple Authentication and Security Layer (SASL). 2006. Available online: https://tools.ietf.org/pdf/rfc4422.pdf (accessed on 28 March 2016).

64. MQTT Community. MQTT Version 3.1.1. 2014. Available online: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf (accessed on 8 December 2016).
65. Amoah, R.; Camtepe, S.; Foo, E. Securing DNP3 Broadcast Communications in SCADA Systems. *IEEE Trans. Ind. Inform.* **2016**, *12*, 1474–1485. [CrossRef]
66. Booch, G.; Rumbaugh, J.; Jacobson, I. *The Unified Modeling Language User Guide*, 2nd ed.; Addison-Wesley: Reading, MA, USA, 2005.
67. Diaz, H.D.; Ortega, J.F.M.; García, A.C.; Rodríguez-Molina, J.; Cifuentes, G.R.; Jara, A. Semantic as an Interoperability Enabler in Internet of Things. In *Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems*; River Publishers: Aalborg, Denmark, 2013.
68. Hussain, A.; Farooq, K.; Luo, B.; Slack, W. A Novel Ontology and Machine Learning Inspired Hybrid Cardiovascular Decision Support Framework. In Proceedings of the 2015 IEEE Symposium Series on Computational Intelligence, Cape Town, South Africa, 7–10 December 2015.
69. Simmins, J.J. The impact of PAP 8 on the Common Information Model (CIM). In Proceedings of the Power Systems Conference and Exposition (PSCE), 2011 IEEE/PES, Phoenix, AZ, USA, 20–23 March 2011.
70. De Diego, R.; Martínez, J.F.; Rodríguez-Molina, J.; Cuerva, A. A semantic middleware architecture focused on data and heterogeneity management within the smart grid. *Energies* **2014**, *7*, 5953–5994. [CrossRef]
71. Red Hat, Inc. Red Hat JBoss Fuse: Download for Development Use. 2016. Available online: http://www.jboss.org/products/fuse/download/ (accessed on 30 September 2016).
72. Redmonk Community. The RedMonk Programming Language Rankings: January 2015. Available online: http://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/ (accessed on 30 September 2016).
73. Moreno Rodríguez, L.M. Evaluación del dispositivo Raspberry Pi como elemento de despliegue de servicios en el marco de las Smart Grids. Bachelor's Thesis, Technical University of Madrid, Madrid, Spain, 2014. (In Spanish)
74. Apache Software Foundation. Downloads: Apache ServiceMix 4.5.3. 2011. Available online: http://servicemix.apache.org/downloads/servicemix-4.5.3 (accessed on 30 September 2016).
75. Eclipse Foundation. Eclipse Installer Download. 2016. Available online: https://eclipse.org/downloads/ (accessed on 30 September 2016).
76. NetBeans Website. NetBeans IDE 8.1 Download. 2016. Available online: https://netbeans.org/downloads/ (accessed on 30 September 2016).
77. Apache Software Foundation. Apache Maven Project. 2016. Available online: https://maven.apache.org/ (accessed on 30 September 2016).
78. Apache Software Foundation. Introduction to the POM. 2016. Available online: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html (accessed on 30 September 2016).
79. Vlissides, J.; Helm, R.; Johnson, R.; Gamma, E. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley: Reading, MA, USA, 1994.
80. Apache Software Foundation. Apache Jena, Documentation TDB. 2015. Available online: https://jena.apache.org/documentation/tdb/index.html (accessed on 30 September 2016).
81. Apache Software Foundation. Apache HTTP Server Project. 2016. Available online: https://httpd.apache.org/ (accessed on 30 September 2016).
82. The Legion of Bouncy Castle. Bouncy Castle Crypto APIs. 2013. Available online: https://www.bouncycastle.org/ (accessed on 30 September 2016).
83. Norwegian, Demosteinkjer Lab Web Site. Description of the Available Facilities in the Living Lab of Steinkjer. Available online: https://www.demosteinkjer.no/ (accessed on 30 September 2016).
84. e-GOTHAM Consortium. e-GOTHAM in Short. 2015. Available online: http://ylivieska.centria.fi/egotham/eGenglish.html (accessed on 30 September 2016).
85. OpenEnergyMonitor Project. OpenEnergyMonitor Project: About. 2016. Available online: https://openenergymonitor.org/emon/sustainable-energy (accessed on 30 September 2016).
86. RS Components. Raspberry Pi Model B. 2016. Available online: http://docs-europe.electrocomponents.com/webdocs/127d/0900766b8127da4b.pdf (accessed on 8 December 2016).
87. e-GOTHAM Consortium. Validation Report for Prototype 2 in the Three Pilots. 2015. Available online: http://www.e-gotham.eu/index.php/e-gothamproj/results2 (accessed on 5 December 2016).

88. CITSEM Web Manager. Next-Generation Networks and Services (GRYS). 2012. Available online: https://www.citsem.upm.es/index.php/en/research-en/groups-en/group-of-next-generation-networks-and-services-grys-en (accessed on 3 February 2017).
89. CITSEM Web Manager. Research Center on Software Technologies and Multimedia Systems for Sustainability (CITSEM—Centro de Investigación en Tecnologías Software y Sistemas Multimedia para la Sostenibilidad). 2011. Available online: https://www.citsem.upm.es/index.php/en/ (accessed on 3 February 2017).