FP7-SMARTCITIES-2013
Project number: 609132
http://www.e-balance-project.eu/

e-balance

Deliverable D4.3

Detailed middleware specification and implementation

Editor:	Daniel Garrido
Dissemination level: (Confidentiality)	PU
Suggested readers:	Consortium/Experts
Version:	1.0
Total number of pages:	32
Keywords:	Implementation, Middleware, Interfaces, Communication Platform

Abstract

This deliverable describes the specification of the e-balance middleware together with the detailed information about the design and implementation. The document will be structured in two mains parts: middleware design and implementation. The middleware is consistent with the input from other tasks. More specifically, it follows the data interface presented in D3.2. A solution following this approach has been developed allowing the communication between different Management Units of the e-balance system. An API has been provided in such a way that the energy management platform can get and provide information from the communication platform. Security mechanisms have also been considered attending to D4.2. Finally, different network stacks as proposed in D4.1 can be used.

Impressum

[Full project title] Balancing energy production and consumption in energy efficient smart neighbourhoods

[Short project title] e-balance

[Number and title of work-package] WP4 Communication Platform
[Document title] Detailed middleware specification and implementation
[Editor: Name, company] Daniel Garrido, UMA

[Work-package leader: Name, company] Daniel Garrido, Universidad de Málaga

Executive Summary

The deliverable presents the specification, design and implementation of the middleware in charge of facilitating and making the exchange of data transparent to all the components (e-balance applications on the management units) involved in the e-balance system. It allows exchanging the data between the devices, hiding the communication between them and offers a homogeneous API for accessing the data.

The middleware has been implemented using the ServiceStack, which is an open source framework designed to create web services under the .NET environment. ServiceStack allows modularizing the functionality of the middleware in two levels: plugins and services. The basic functionality of the middleware has been implemented as services and the plugins act as modules able to host multiple services. The plugin concept has been used to implement all those components which are part of the middleware.

The middleware is based on four main modules: Data Persistence, Request Processor, Data Access Control and Maintenance and Group Management. The data persistence module is in charge of storing all the information generated in the system in a persistent form. The request processor module is the one in charge of attending the different requests coming either from the data interface or from the networking layer. All requests go through the submodule of the request processor – the data access control module that grants or denies permission to access the data. And finally, the maintenance and group management module manages the communication network and keeps track of the connected devices, like management units, sensors and actuators.

The data is represented by e-balance variables. A variable defines the meaning and location of the data while its instance represents a single sample of the data. The data structure allows addressing the data in several dimensions, i.e., according to time, space and owner.

The deliverable also presents the details of the data interface. It is an API, through which the data consumers can interact with the functionality offered by the middleware which is basically to exchange of data. On the one hand, the document presents this API based on RESTful Web Services, which make the middleware fully interoperable with remote devices as its operations can be triggered from any operating system and the data consumers can be implemented using any programming language. This API offers four simple functions:

- *Write()* – allows updating/modifying the value of an e-balance variable.
- *Query()* – allows reading the value of a specific e-balance variable in a given moment.
- *Event()*. Through this function an e-balance user can subscribe to the system to receive information of a particular e-balance when a defined condition is satisfied.
- *Periodic()*. It allows receiving information about an e-balance variable in a periodical way.

On the other hand, a functional API is also presented. It was designed to be used by the device local software applications that needs to interact with the middleware using functions. This API is built on top of the RESTful Web Service API and offers high level functions designed for the developers to simplify and accelerate the implementation of e-balance applications. For instance, the functional API offers the function *QueryFromChildMUs(ebVariable)* which is an extension to the basic function *Query()*.

List of authors

Company	Author
UMA	Eduardo Cañete Jaime Chen Daniel Garrido Manuel Díaz
IHP	Krzysztof Piotrowski
EFACEC	Alberto Jorge Bernardo Paulo Delfim Rodrigues Nuno Silva António Carrapatoso Alberto Rodrigues

Table of Contents

List of Tables

List of Figures

Abbreviations

API	Application Programming Interface
CMU	Customer Management Unit
DMS	Distribution Management System
DER	Distributed Energy Resources
HAN	Home Area Network
HV	High Voltage
LV	Low Voltage
LV-FAN	Low Voltage Field Area Network
LVGMU	Low Voltage Grid Management Unit
MDM	Metering and energy Data Management
MVGMU	Medium Voltage Grid Management Unit
MV	Medium Voltage
MV-FAN	Medium Voltage Field Area Network
PS-LAN	Primary Substation Local Area Network
REST	REpresentational State Transfer
SCADA	Supervisory Control And Data Acquisition
TLGMU	Top Level Grid Management Unit
TSO	Transmission System Operator

1 Introduction

The aim of the e-balance project is to design and implement a new infrastructure and efficient algorithms able to manage the smart grid system in a smart way. From the end-user point view, the whole system will be able to analyse and control the energy production and consumption of the prosumers in order to take smart decisions in order to positively change the behaviour of users and to achieve a better use of the energy. And from the infrastructure point of view, e-balance pursues to develop a system capable of efficiently exchanging information among all the devices that form this self-resilience system. The data exchange middleware is part of the communication platform, it is designed and developed in WP4 and connects the different devices in the smart grid. Task 4.1 surveys and proposes the networking layer that supports the e-balance energy balancing system. In Task 4.2 different security and privacy mechanisms have been studied in order to ensure that the communication platform is secure. Both tasks provide their inputs to the Task 4.3.

This document describes the results of the Task 4.3, where a common communication middleware platform for energy control and management has been designed and implemented. The developed system is highly distributed and heterogeneous as there are many different devices running in parallel that have to cooperate. It is also hierarchical due to the fact the e-balance infrastructure is organized in three levels: HAN, LV-FAN and MV-FAN, similar to the hierarchy levels in the energy grid.

The middleware uses the communication protocols developed in the Task 4.1, and the security and privacy mechanisms proposed in Task 4.2. Furthermore, the platform provides the distribution of the data required by the algorithms developed in WP5 for energy control and management.

1.1 Deliverable Position in the Project

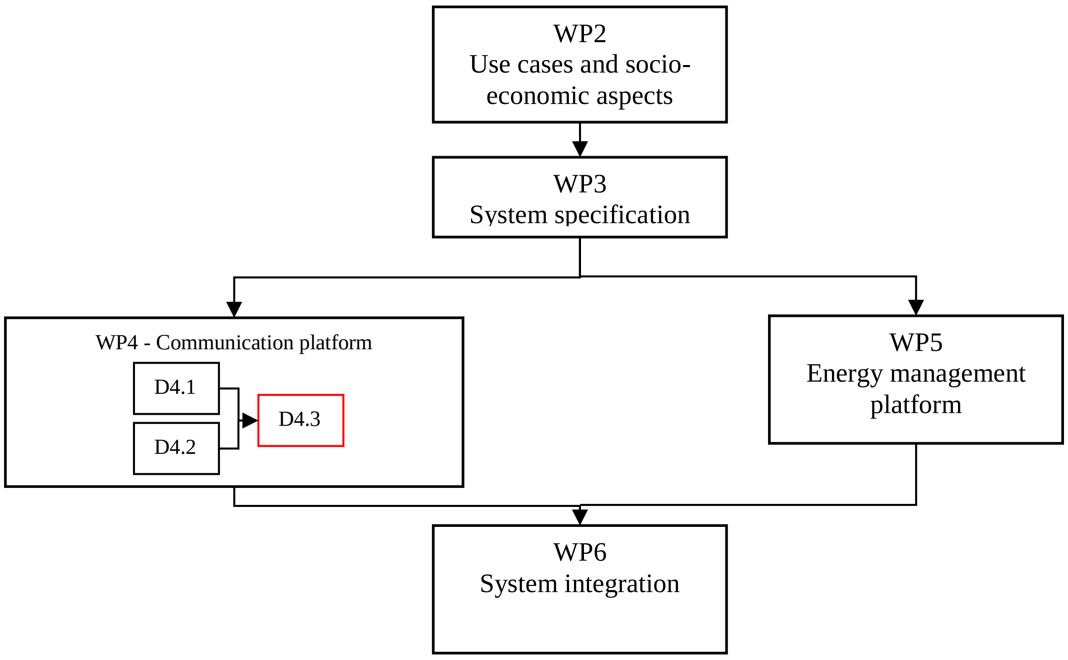
WP2
Use cases and socio-economic aspects
WP3
System specification
WP4 - Communication platform
D4.1
D4.2
D4.3
WP5
Energy management platform
WP6
System integration

Figure 1 – The position of deliverable D4.3 within the e-balance project

The deliverables D4.1 and D4.2 have studied the networking layer as well as the security and privacy mechanisms, respectively. The outputs of these two deliverables are the base to design and implement the data exchange middleware (Task 4.3) which is described in this document (D4.3). Once the middleware has been finished, it will be tested by means of in-lab tests taking into account the requirements defined in D2.4. This activity will be carried out in the Task 4.4 and described in the deliverable D4.4. Finally, once the data exchange middleware has been verified and tested a more through in-site evaluation will be carried out in WP6 together with the energy management platform developed in WP5.

2 Data exchange in the e-balance system

The e-balance system architecture (see Figure 2) shows the system components and their interactions as well as their relations with the grid. The e-balance system components are represented by dark blue coloured shapes. The light blue boxes represent the Bulk Generation and Transmission Level. The e-balance system involves several management units and the figure depicts the hierarchical tree of these management units with a single management unit level for each voltage level in the grid.

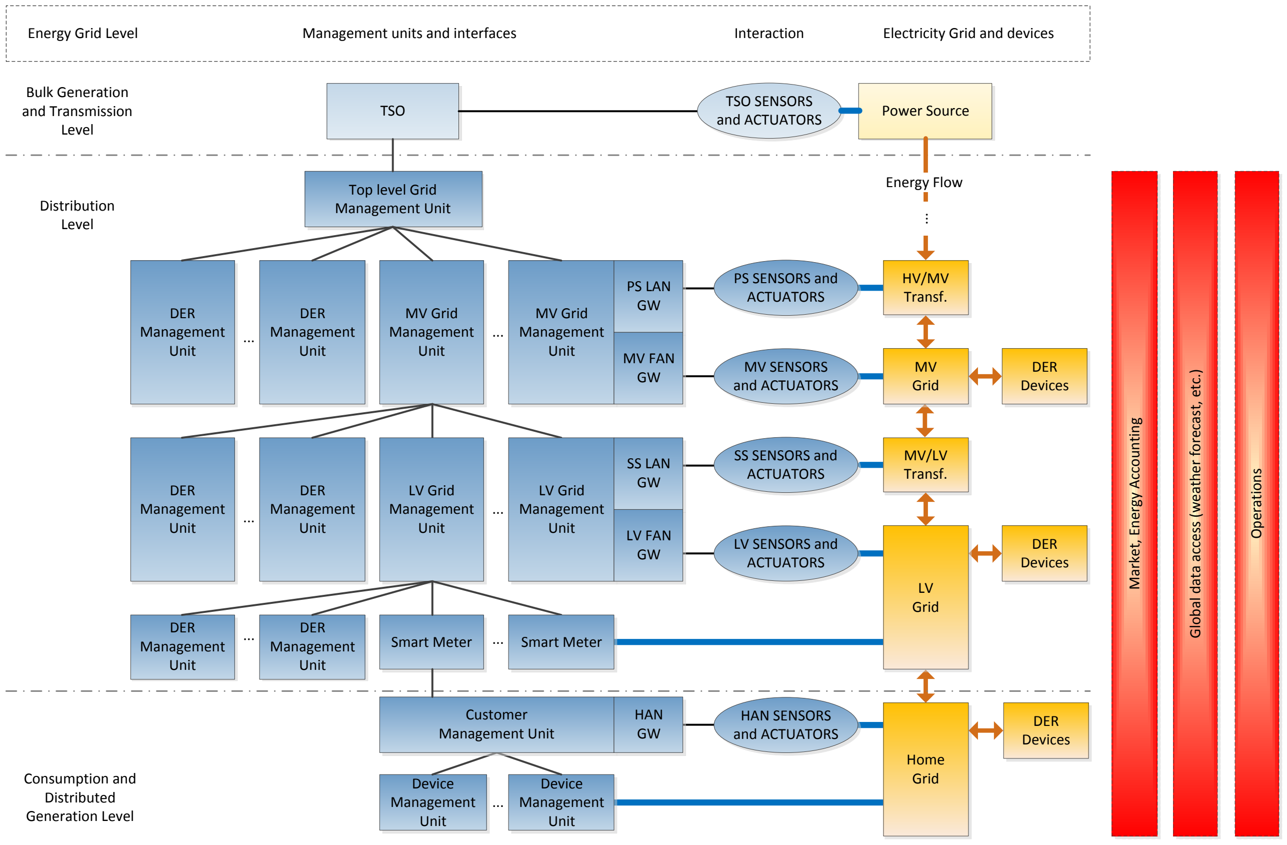

Figure 2 – The e-balance high-level system architecture within the grid

The yellow coloured shapes represent the grid and the devices within the grid. Finally, the red coloured boxes represent virtual layers like the Market, Energy Accounting, Global Data Access and the Operations. The different lines in the figure represent different kinds of interaction between the modules they connect. For instance, relevant for this deliverable, the black lines represent the network, i.e., the data exchange that involves the management units, as well as the sensors and actuators.

All the management units have a similar architecture, which is described in Section 2.3 of D3.1 as well as in D3.2. However, depending on the level, the management units may have different roles and duties. The processes executed on them may operate on behalf of different stakeholders and process data from different stakeholders. But, since at every level the concept of data collection and processing is similar and the e-balance system architecture is fractal-like, the management algorithms applied on different management levels share the same conceptual base, what improves the scalability of the approach.

The device level is the lowest level represented in the architecture. A device may be of any kind, including home appliance that only consumes energy, but it may also be an energy generation or storage unit. The Device Management Unit (DMU) is a central unit of the device that is aware of the current state of all the components that the device consists of and controls these components. The device management unit is also equipped with a communication module or gateway that allows upward communication with the higher level management unit, i.e., the Customer Management Unit (CMU) that controls all the customer devices at the customer premises.

The customer management unit is also equipped with several communication gateways. It communicates downwards with its underlying device management units, but it also communicates with Home Area

Network (HAN) sensors and actuators that interact with the home grid providing grid monitoring and control, but also support the home automation functionality.

The level above the customer management units consists of low voltage grid management units (LVGMU). These management units are located at the secondary substations and each of them controls the sensors, actuators, customer management units and DER management units located in the area of the grid supplied with energy by this secondary substation. A LV grid management unit is equipped with communication gateways for the upward and downward communication within the e-balance management hierarchy. It is also equipped with communication gateways for communication with sensors and actuators located at the MV/LV transformer (Secondary Substation Local Area Network – SS-LAN) and also in the LV grid feeders related to the secondary substation (Low Voltage Field Area Network – LV-FAN). All these communication gateways may be different, depending on the technologies used in each part of the network.

A medium voltage (MV) grid management unit (MVGMU) is similar to its counterpart for the low voltage. A MV grid management unit resides at a primary substation. It is equipped with upward and downward communication gateways and controls all the sensors, actuators and LV management units located at secondary substations related to this primary substation as well as the DER management units in the area for DER devices connected directly to the LV grid (in contrast to those at customer premises). In order to interact with the sensors and actuators at the HV/MV transformer the MV grid management unit is equipped with Primary Substation Local Area Network (PS-LAN) gateway. Similar, for communicating with the sensors and actuators in the MV grid related to the primary substation the Medium Voltage Field Area Network (MV-FAN) gateway is available at the MV grid management unit.

The top level grid management unit (TLGMU) controls all MV management units as well as all the DER management units for DER connected directly to the MV grid, i.e., it collects all the status data and sends control signals to all the lower level management units. The top level grid management unit may be considered as a control centre that provides also interfaces for management tools, like supervisory control and data acquisition (SCADA), market management, outage management, Distribution Management System (DMS), and Metering and energy Data Management (MDM). The top level grid management system communicates also with the Transmission Service Operator (TSO).

Previous paragraphs give us an idea about the complexity of the e-balance system from the communication point of view – it is composed of different hardware devices (management units, sensors, smart meters, etc.) with different resources, different operating systems and made by different manufacturers. Furthermore, all these devices have to communicate within a hierarchical structure in a secure way and under different roles.

In order to facilitate the development of applications on the top of e-balance system a middleware has been designed and implemented to provide an abstraction layer that hides the communication related details. This middleware provides developers a simple way of programming applications for the e-balance system and at the same time it manages the complex underlying communication network. This middleware is presented in detail in the following sections.

3 Middleware Design

The concept middleware in the context of this project refers to a piece of software that runs on the different hardware and operating systems on the different management units and provides high level services to the applications and users. The advantages of the middleware are especially evident in this kind of heterogeneous system, where different devices with different capabilities cooperate as one system. The communication middleware within the e-balance system handles this heterogeneity, automatically manages the network and the relationship between different devices and provides a homogeneous programming API that simplifies the task of using and developing applications.

In our system the middleware is thought to run on the different management units, namely, the TLGMU, the MVGMU, the LVGMU, the DERMU and CMU. The middleware three main goals are:

1. Provide means for exchanging information: the middleware will automatically handle different network stacks and will provide a simple way of communication between different devices.
2. Provide ways for storing/getting information: by using the API provided by the middleware, devices can obtain and store persistent information either from/on remote devices or from/on the local one.
3. Raise the level of abstraction with which these devices are programmed: the middleware provides a simple data access API based on a combination of publish/subscribe and querying.

3.1 Middleware Architecture

Figure 3 shows the main architecture of the communication platform and its relationship to the underlying energy infrastructure. The communication platform sits above the network stacks and below the e-balance applications. The communication platform makes use of the adaptation layer called the communication manager module that adapts different networking stack solutions to the same middleware communication language. The data persistence module holds all the information in the system and provides means for getting and storing it. The maintenance and group management module handles the network and the relationship between devices. The request processor module is the one in charge of handling all the requests received by the communication platform. This module is also responsible for correct processing of the requests depending on the security and privacy policies of the data owners. This functionality is implemented in the data access control submodule.

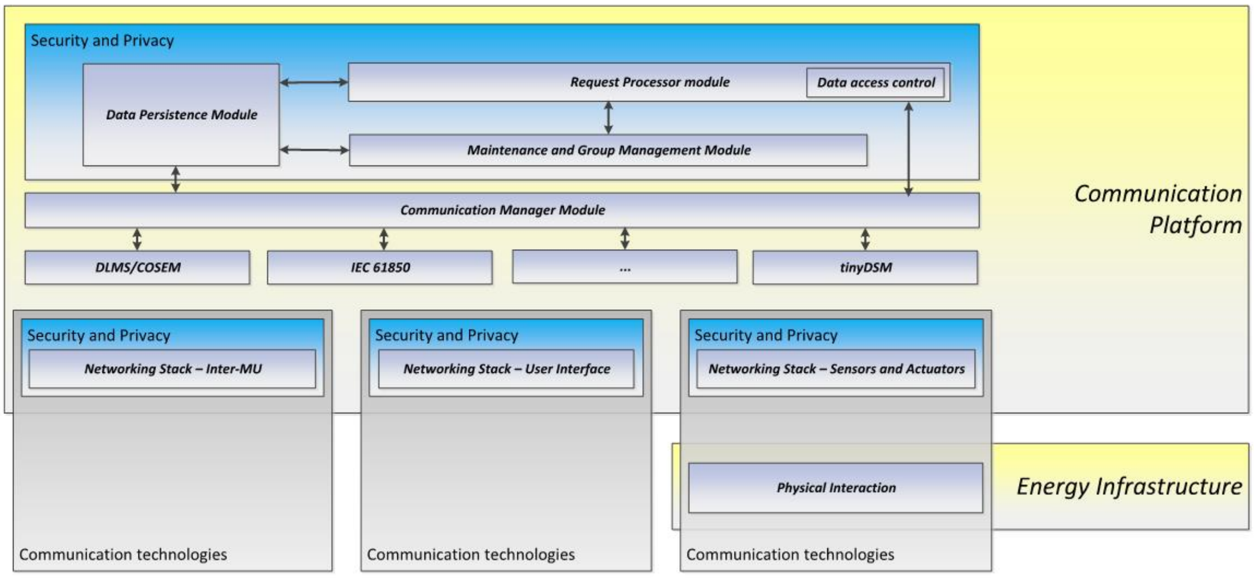
Security and Privacy
Data Persistence Module
Request Processor module
Data access control
Maintenance and Group Management Module
Communication Manager Module
DLMS/COSEM
IEC 61850
...
tinyDSM
Communication Platform
Security and Privacy
Networking Stack – Inter-MU
Security and Privacy
Networking Stack – User Interface
Security and Privacy
Networking Stack – Sensors and Actuators
Physical Interaction
Energy Infrastructure
Communication technologies
Communication technologies
Communication technologies

Figure 3 – e-balance communication middleware architecture

3.1.1 UML Diagram

Figure 4 shows the general view on the e-balance middleware from the software point of view. The diagram shows the different modules that build-up the system (data persistence, request processor, maintenance group management, data access control and user management modules) and also the services provided by each of them. The services represent the functionality offered by the modules. The figure also shows a library called *EBCommon* which is an e-balance library used at internal level to facilitate the middleware development. It has helper functions and other functionality such as time handling functions, functional interface to other modules, etc.

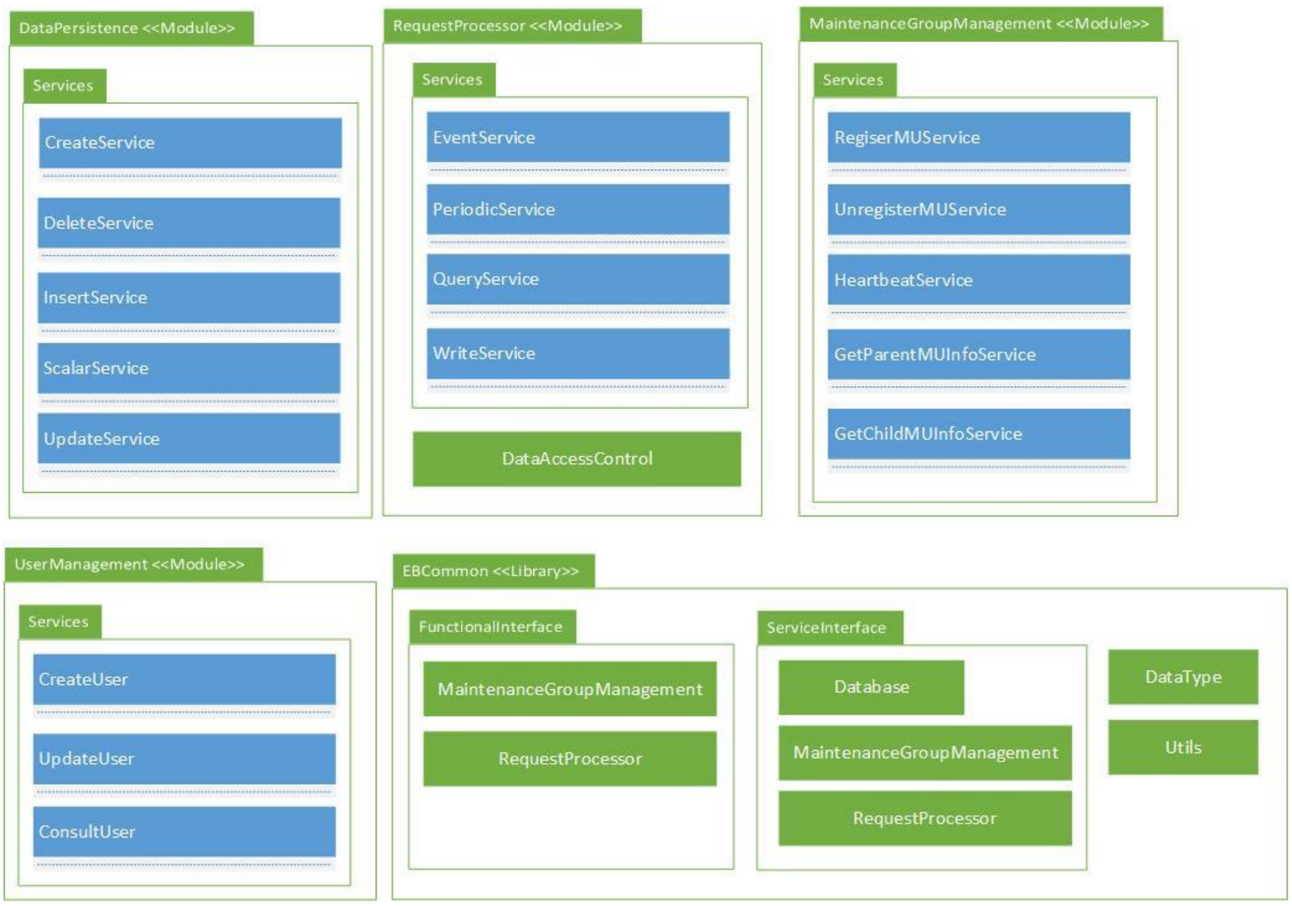

Figure 4 – Package Diagram

3.2 The Middleware Modules

The middleware consists of a set of independent modules that collaborate together by means of interfaces (services offered by each module). This design approach allows the different modules to be replaced without affecting the other modules. For example, if a new kind of database needs to be used, then only the data persistence module needs to be updated. The other modules remain the same. It is the task of the developer to adapt the new database to the API provided by the data persistence module to make it compatible with the rest of the modules. This section presents the different modules the e-balance data exchange middleware is composed of. A general overview of these different modules is shown in Figure 3.

3.2.1 Data Persistence

This module, depicted in Figure 5, is in charge of storing all the information generated in the system in a persistent form in a non-volatile storage, e.g., on a hard disk. The information is organized and structured in two entity groups: e-balance variables and e-balance configuration. The former store all those information related to sensor read from sensors and the latter stores all the information necessary to establish the proper working environment for each management unit and also the user defined settings that are MU specific. E.g. an e-balance variable stores the information on the energy consumption and an e-balance configuration variable stores the Id of the device.

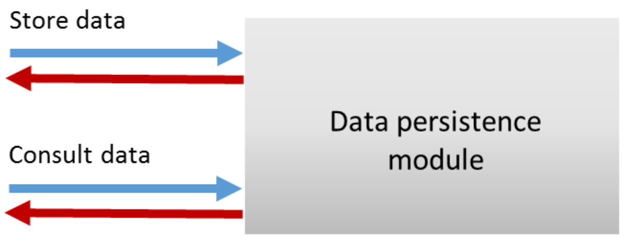

Figure 5 – Data Persistence Module

This module offers an interface for the rest of the data exchange middleware modules to achieve the independence of the applied database implementation. This approach allows different database and storage system to be used without affecting the e-balance system which relies on the API provided by this module. For example, on the CMU, which is a resource constrained device, a simple SQLite [1] database has been used. On more powerful devices such as the TLGMU a more complex database solution is to be applied.

Figure 6 depicts the diagram of the database structure used by the data persistence module. In general terms, the database tables and their relationships are designed to store the collected measurements, but also information about the subscriptions to defined events and periodics, as well as the relationships between the stakeholders and the e-balance variables, data related permissions, the failures detected in the system, etc.

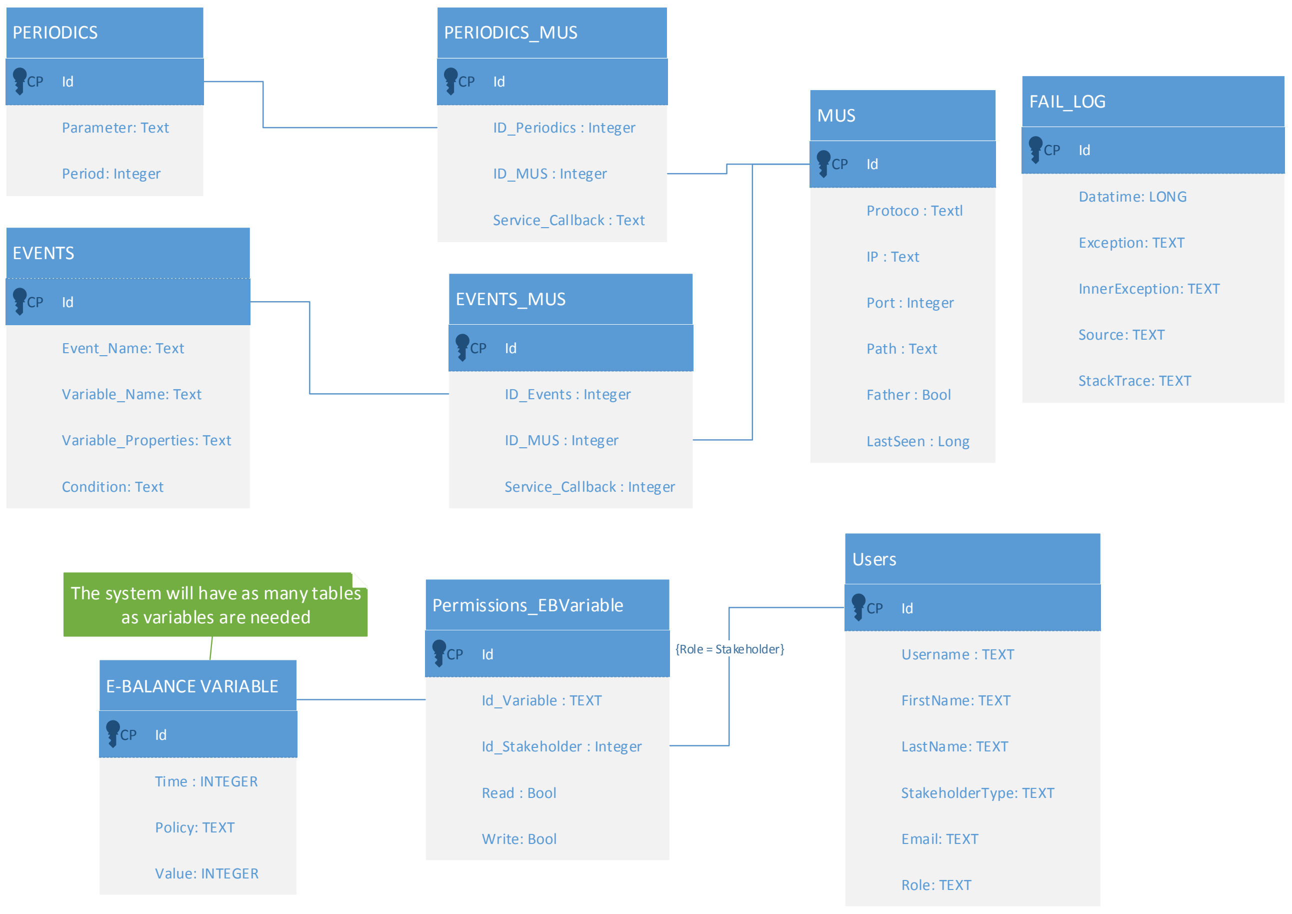

Figure 6 – Database diagram

3.2.2 User Management

All services offered in the e-balance middleware are protected against unauthorized access. Each user must have the credentials (username and password) to access a set of services or functionalities offered by the system. The set of services that can be accessed from each account is determined by its role. For example only the admin role has full access to all the services provided by the e-balance middleware. This functionality is managed through the user management module depicted in Figure 7. It allows the super-user (e-balance administrator) that is already pre-established in the system to create or remove different users and assign them different roles. For example, our system allows different users within the same stakeholder, which means that they have the same data permissions, but may have different functions.

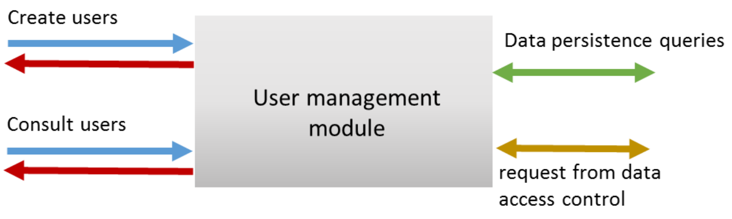

Figure 7 – User Management Module

This module is already related to the security and privacy and is also part of the work done in Task 4.2, but is presented here as an important part of the middleware. Further, the stakeholder identifiers assigned to the different users created with the module are also used by the other security and privacy related module, the data access control module (part of the request processor module presented in the following section) to check the access permissions for the e-balance variables.

3.2.3 Request Processor

The request processor module is the one in charge of processing the different requests coming either from the local energy management logic via the data interface or from remote devices via the networking layer, as presented in Figure 8. The request processor module handles the following messages:

- Requests to perform an operation as defined by the data interface API in Section 3.3 or coming from a remote management unit via the communication manager.

  For example, if a read request is received by the request processor, it will use the data persistence module (described in Section 3.2.1) to query for the information and will return the results back to the caller.

- Requests, whose destination is the maintenance and group management module.

  The goal of this module is described in Section 3.2.5. The maintenance and group management packets are first received in the request processor module to centralize the data access control in one single module.

- Additional requests coming for example from different sensors deployed in the system.

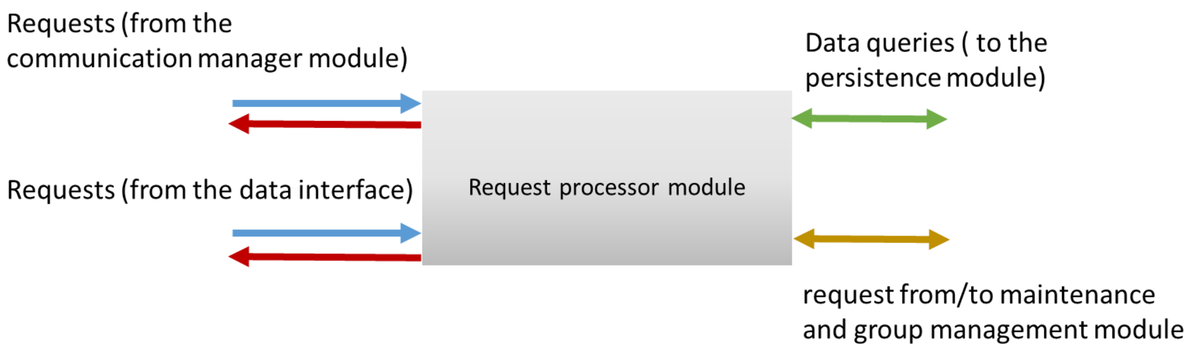
Requests (from the communication manager module)
Requests (from the data interface)
Request processor module
Data queries (to the persistence module)
request from/to maintenance and group management module

Figure 8 – Request processor module

3.2.4 Data Access Control

All requests go through a submodule of the request processor – the data access control module that grants or denies permission to access the data or the rest of modules in the system, depending on the access policies defined by the data owners.

The data access control module, depicted in Figure 9, is a submodule of the request processor module. It is in charge of granting access to authorised stakeholders and denying access to the others. All requests are routed through the request processor and thus, also through the data access control module. This module queries the permissions for the given data variable from the data persistence module and either allows or denies the access to the data. In case the access request is denied the caller will be notified about the access restriction.

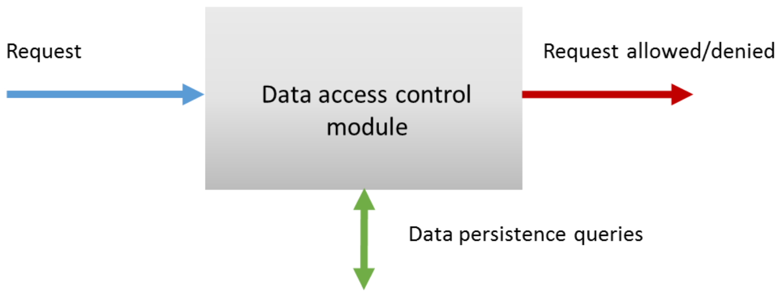

Figure 9 – Data access control module

The details on the security mechanisms used in the implementation of the module are given in deliverable D4.2, while the overall security and privacy protocol is described in deliverable D5.4.

3.2.5 Maintenance and Group Management

The maintenance and group management module, depicted in Figure 10, manages the MUs network and keeps track of the connected devices and sensors. For example, it periodically checks the status of remote management units and informs the request processor module about the state and events, like a node disconnection, a new management unit entering the system, etc. The module saves or queries information by means of the data persistence module.

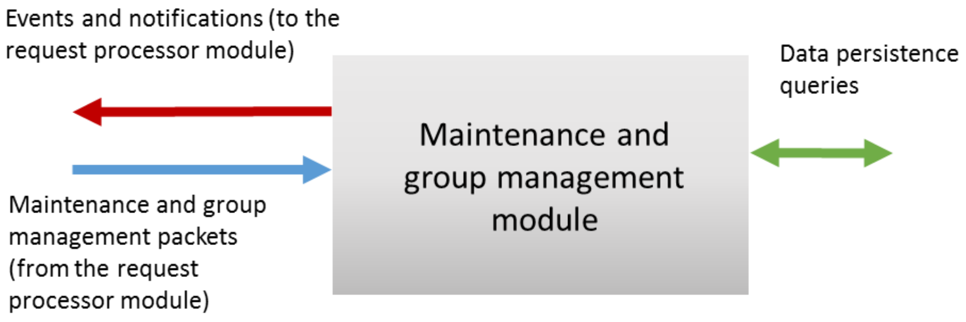
Events and notifications (to the request processor module)
Maintenance and group management packets (from the request processor module)
Maintenance and group management module
Data persistence queries

Figure 10 – Maintenance and group management module

The information received by this module must come from the request processor module. Additionally, all the events generated by this module that want to be sent out must be notified to the request processor module.

All the MUs in an e-balance system deployment have a parent-child relationship. For example, a CMU will be associated (as child) to the respective LVGMU (parent). Similar, a LVGMU will be associated (as child) to the respective MVGMU (parent).

Each MU offers three operations that allow associating a child MU to a parent MU. The association process is also used to exchange credentials to be able to access the services offered by both the child and parent MU. The operations related to the MU registration and maintenance process are as follows.

- **RegisterMU** – this service allows to a user with the operator role to register a MU child at a MU parent or vice versa. It is used to exchange credentials between the child and the parent MU.
- **UnregisterMU** – this service allows a MU to unregister from a parent or child MU.
- **Heartbeat** – this service allows a child MU to indicate to its parent MU that it is connected and running (alive). This service detects temporal disconnections or disruption in the path between the parent and the child MUs. It can be used to detect MU failures.

All operations in the registration module can only be executed with an account (user) with the operator privileges (operator role). The registration is usually only carried out once while the MU is initially configured.

Figure 11 shows a sequence diagram where the process to associate a child MU to a parent MU is described. In this particular example, it is shown how an operator called John registers a CMU into a LVGMU. The needed steps to carry out this particular registration are as follows.

- An operator whose name is John goes to the house where a CMU (E.g. CMU_ID = C1) has to be installed. He connects to the CMU and opens the GUI through which he can associate the CMU with the respective LVGMU. He fills out the form with the following information: its username, password, LVGMU_URL and LVGMU_ID, and clicks on the RegisterCMU button.
- LVGMU receives the request and it stores the CMU information (URL and ID) in its database.
- The CMU credentials are generated locally at LVGMU receiving the request.
- LVGMU sent to the CMU the credentials created for the CMU in the previous step and its own credentials. This way, the LVGMU has the credentials to access to the CMU and the CMU has the credentials to access to its parent LVGMU.
- CMU receives the credentials generated for it by the LVGMU and uses them to create a local user. This user will be used by the LVGMU to talk to CMU.
- Once John has linked the CMU with the LVGMU, the CMU calls the heartbeat service offered by the LVGMU in a periodical way to indicate it is alive and working.

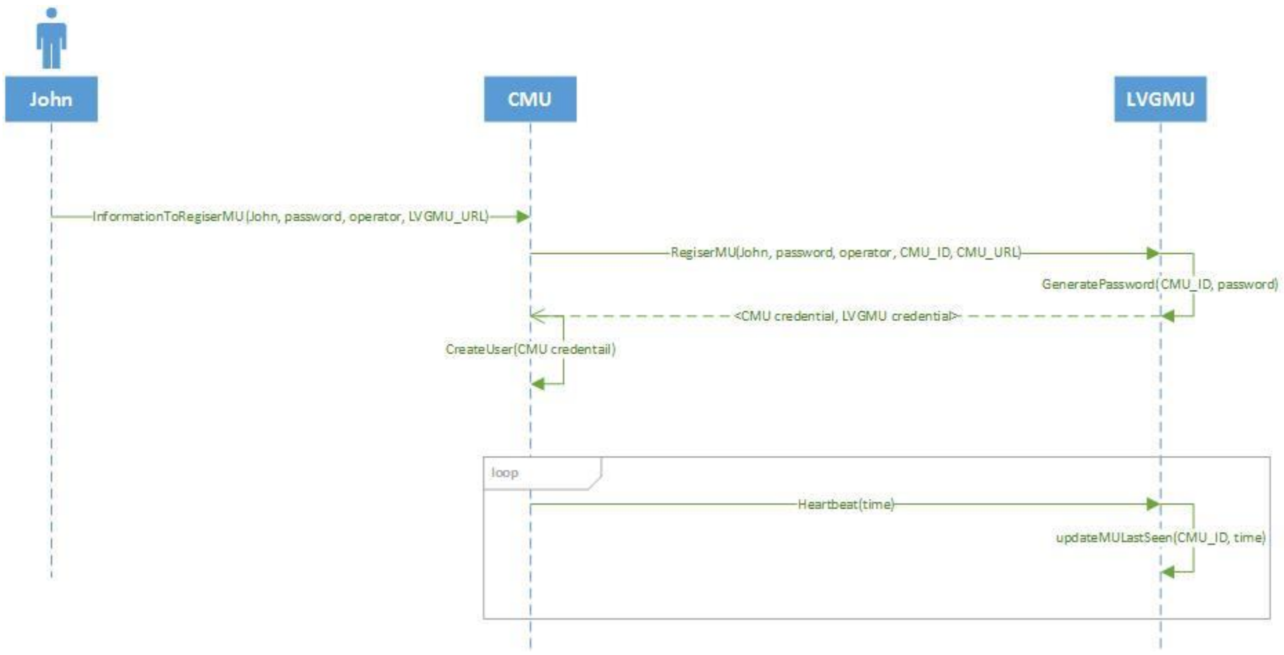
John
CMU
LVGMU
InformationToRegiserMU(John, password, operator, LVGMU_URL)
RegiserMU(John, password, operator, CMU_ID, CMU_URL)
GeneratePassword(CMU_ID, password)
<CMU credential, LVGMU credential>
CreateUser(CMU credentail)
loop
Heartbeat(time)
updateMULastSeen(CMU_ID, time)

Figure 11 – Process to associate two MUs

3.3 Management Units Data Interface

The data interface defines the functions through which software modules (Energy Management Platform) on the local MU or remote MUs can interact with a particular MU to read and/or write data to/from it. The data interface makes it easier to develop an e-balance application.

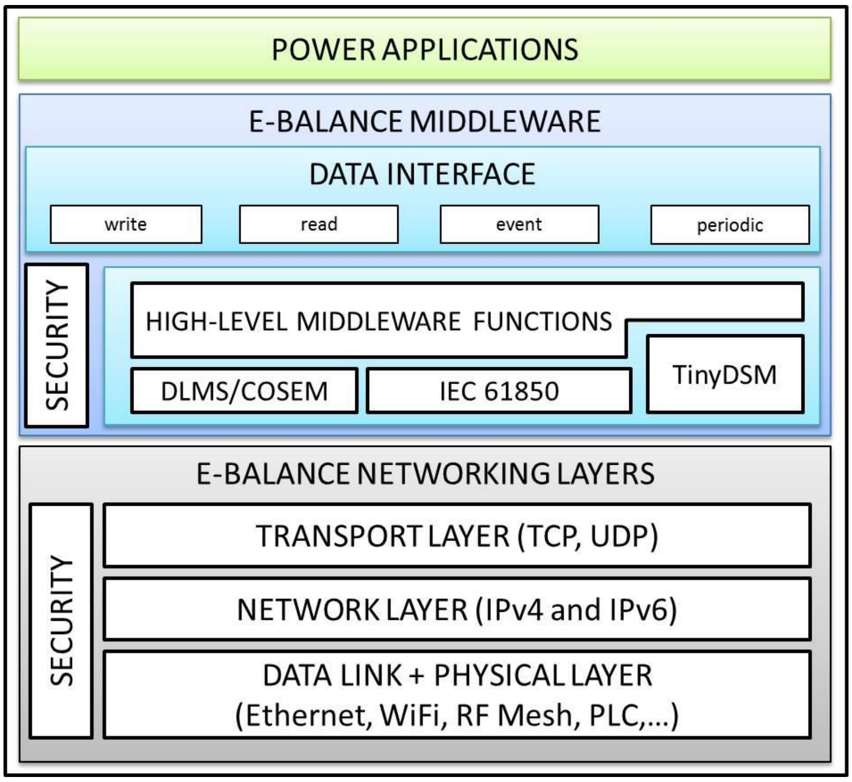
POWER APPLICATIONS
E-BALANCE MIDDLEWARE
DATA INTERFACE
write
read
event
periodic
SECURITY
HIGH-LEVEL MIDDLEWARE FUNCTIONS
DLMS/COSEM
IEC 61850
TinyDSM
E-BALANCE NETWORKING LAYERS
SECURITY
TRANSPORT LAYER (TCP, UDP)
NETWORK LAYER (IPv4 and IPv6)
DATA LINK + PHYSICAL LAYER
(Ethernet, WiFi, RF Mesh, PLC,...)

Figure 12 – Data Interface Context

The data interface (Figure 12) was designed with simplicity as primary goal. For this reason, it is only composed of four basic operations:

- **Query –** this operation allows reading the value of a specific e-balance variable in a given moment.
- **Write –** it is an operation designed to update/modify the value of a variable.
- **Event –** this operation allows monitoring an e-balance variable and to receive notifications in case this particular e-balance variable satisfies a defined condition (e.g.: e-balance variable >= threshold).
- **Periodic** – this operation allows subscribing to periodical reception of the value of an e-balance variable. An example here can be a periodic reading of the consumption variable in order to refresh the value on the customer GUI. The difference to the event subscription is that there is no condition based on the value of the variable, instead the period between notifications is defined. It is possible to have multiple events defined for a single variable for the same data owner, but it is possible to define only a single periodic subscription for one variable from a single data owner.

An operation can only be called on existing variables in the system and only if the caller has the required permissions, in order to be successful. The list of existing variables is predefined in the e-balance system.

All the communication carried out in the system must be done using these simple API. The middleware automatically translates this API calls to the corresponding messages understandable by the underlying network stack. The event and periodic operations follow the well-known publish/subscribe communication paradigm. The application subscribes to a parameter and events are sent whenever the value is updated or a condition is met. On the other hand, the application can also carry out on-demand requests using the read and write operations.

4 Middleware Implementation

This section describes the implementation details, i.e., it explains how the design explained in Section 3 has been implemented.

4.1 ServiceStack

The e-balance middleware has been implemented using ServiceStack [2] which is an open source framework designed to create web services under the .NET environment. ServiceStack allows modularizing the functionality of the middleware and also structure these at two levels: plugins and services. Through the services the basic functionality of the middleware has been implemented and the plugins act as modules able to contain as many services as needed. The plugin concept has been used to implement all the components presented in Section 3.2.

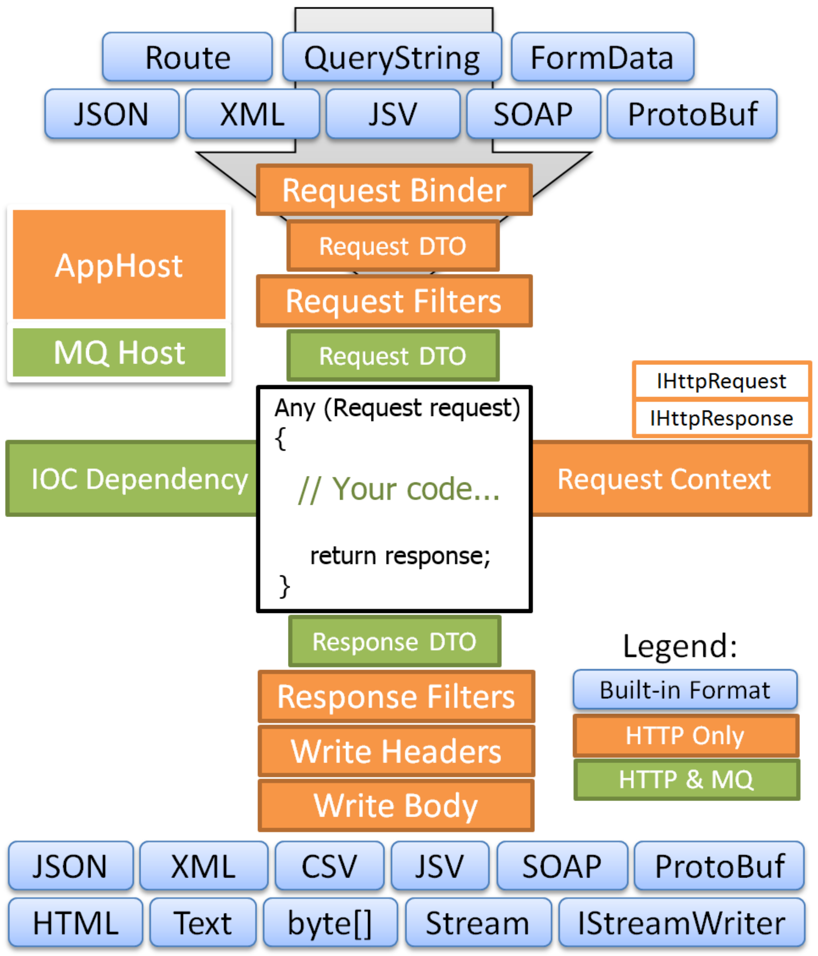

Figure 13 – ServiceStak Server Architecture

Each ServiceStack web service is modelled through a Request DTO (data transfer object) and a Response DTO. That is, the input of a service is a request DTO and the output is a response DTO. Both these DTOs define the interfaces of the service. Seeing the service as a function, the request DTO would be the set of input arguments and the response DTO would be the set of output arguments. Figure 13 shows the server architecture of service stack and where the DTOs are placed within this architecture.

4.2 E-balance variables

All the e-balance data is modelled as database tables. Therefore, an e-balance variable as a piece of the data is modelled as a database table where the name of the table refers to the e-balance variable and the fields of the table refer to the properties of the variable. An e-balance variable can have as many properties as it is needed, but by default there are three mandatory properties, namely the Id, the Timestamp and the Value that will always be present.

Of course it is also possible to have all e-balance variables in one table, but the effectiveness of the approach depends on the architecture of the underlying database module. Both these approaches have advantages and disadvantages, e.g., having all variables in one table would allow adding new variables easier, but in the chosen approach a new table can also be added, if necessary. The main advantage of the chosen approach is that having one table per variable allows separating the data with different meaning/function.

Let us suppose we need a variable to store the energy consumption in the system. To achieve that, a table like the one shown in Table 1 can be created and used.

ENERGY_CONSUMPTION		
Id	**Timestamp**	**Value**
1	12/02/2015 14:10:00	30.4
2	12/02/2015 14:20:00	25.8

Table 1 – Example: Energy consumption table in the e-balance system

This table represents the e-balance variable called *energy_consumption* in the system. The variable has the following properties.

- **Id** – identifier of a particular tuple – the data structure representing a single value of the variable.
- **Timestamp** – datetime of the stored energy_consumption value – the temporal identifier of the value.
- **Value** – the value of the energy_consumption variable for each point in time.

To work with these variables, the e-balance system implementation provides a data structure called EBVariable with the following fields.

- **Name** – refers to the name of a particular e-balance variable, e.g. energy_consumption.
- **Properties** – refers to the properties of a particular e-balance variable, e.g., [Time, Value]
- **Values** – contains the values that are going to be stored/read in/from an e-balance variable.
- **Condition** – allows defining a SQL-like condition to read a set of tuples that satisfy the condition, e.g., “Id >2 and Id<5” or “Timestamp = 12/02/2015 14:20:00”. Is based on the defined properties.

The data interface described in the following section has been designed and implemented taking into account the data model presented in this section.

4.3 Data interface

4.3.1 RESTful Web Service Interface

As it was described in the previous section, the data interface of the middleware is composed of four basic operations. Each of these operations (write, query, event and periodic) is implemented using a RESTful web service so that an application can easily interact with a MU through Internet.

Representational State Transfer (REST) defines a set of architectural principles by which one can design web services that focus on system’s resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages.

While REST stands for Representational State Transfer, which is an architectural style for networked hypermedia applications, it is primarily used to build web services that are lightweight, maintainable, and scalable. A service based on REST is called a RESTful service. REST does not dependent on any protocol, but almost every RESTful service uses HTTP as its underlying protocol.

An implementation of a REST Web service follows four basic design principles.

- **Use HTTP methods explicitly**. REST asks developers to use HTTP methods explicitly and in a way that is consistent with the protocol definition. This basic REST design principle establishes one-to-

one mapping between create, read, update and delete (CRUD) operations and HTTP methods. According to this mapping:
   - To create a resource on the server, use POST.
   - To retrieve a resource, use GET
   - To change the state of a resource or to update it, use PUT.
   - To remove or delete a resource, use DELETE.
- **Be stateless**. A REST Web service application (or client) includes all the parameters, context, and other data needed by the server-side component to generate a response within the HTTP headers and body of a request. Statelessness in this sense improves Web service performance and simplifies the design and implementation of server-side components because the absence of the storing of the state on the server removes the need to synchronize session data with an external application.
- Expose directory structure-like URIs
- Transfer XML, JavaScript Object Notation (JSON), or both.

Web services in general make it possible for the developers to interact with the MU from practically any kind of hardware and software (operating) system and programming language.

Each web service has a defined request (information passed to the service/server) and a defined response (information returned by the service/server). This definition is also known as data transfer object (DTO). In the following, the DTOs of each web service (write, query, event and periodic) are described in detail.

4.3.2 Data Web interface

This section describes the DTOs for the web services providing the data access operations defined by the data interface.

4.3.2.1 Write

<u>DTO Request:</u>

- **RequestSource –** identifier of the caller; both the device and the service/stakeholder. A web service URL is composed of the following fields.
   - **Protocol** – it can be http or https.
   - **IP** – IP address of the MU where the web services are located.
   - **Port** – the port through which the MU web services can be accessed.
   - **ServicePath** – path within the MU where the web services are located.

- **Variable** – variable that is going to be stored.
   - **Name –** name of the e-balance variable.
   - **Properties** – indicates the exact properties of the e-balance variable that are going to be stored or modified. If it is set to null, the values parameter must contains a value for each property.
   - **Condition** – this parameter must be set to null when a new value is going to be inserted. However, it will be useful when an existing tuple needs to be modified as it will allow us to identify this tuple.

- **Values** – defines the values of the variable that are going to be stored or modified.

<u>DTO response:</u>

- **OperationResults:**
   - **OpCode –** indicates a code to describe the result of the operation in a concise way.
   - **Info –** provides a descriptive message about the operation.
   - **Success –** indicates if the service was successfully executed.

EXAMPLE

Data Owner			
Protocol	**IP**	**Port**	**ServicePath**
http	192.168.43.98	2554	/

e-balance Variable		
Name	**Properties**	**Condition**
ENERGY_CONSUMPTION	Time, Value	Null

Values
12/04/2015 12:43:00, 23.2

4.3.2.2 Query

DTO Request:

- **RequestSource –** identifier of the caller; both the device and the service/stakeholder. A web service URL is composed of the following fields.
    - **Protocol** – it can be http or https.
    - **IP** – IP address of the MU where the web services are located.
    - **Port** – the port through which the MU web services can be accessed.
    - **ServicePath** – path within the MU where the web services are located.

- **Variable** – variable that is going to be read.
    - **Name –** name of the e-balance variable.
    - **Properties** – indicates the exact properties of the e-balance variable that are going to be read. If it is set to null, all the properties will be returned in the response.
    - **Condition** – allows retrieving tuples related to the e-balance variable that satisfy this condition. If this parameter is set to null the system will try to bring the last tuple.

DTO response:

- **DataResults** – contains the requested information.
- **OperationResults.**
    - **OpCode –** indicates a code to describe the result of the operation in a concise way.
    - **Info –** provides a descriptive message about the operation.
    - **Success –** indicates if the service was successfully executed.

EXAMPLE: DTO request

Data Owner			
Protocol	**IP**	**Port**	**ServicePath**
http	192.168.43.98	2554	/

E-balance Variable		
Name	**Properties**	**Condition**
ENERGY_CONSUMPTION	Null	Null

EXAMPLE: DTO response

Data Result
{12/04/2015 12:43:00, 23.2}

Operation Results		
OpCode	**Info**	**Success**
1	Data successfully read	True

4.3.2.3 Event

DTO Request:

- **RequestSource –** identifier of the caller; both the device and the service/stakeholder. A web service URL is composed of the following fields.
    - **Protocol** – it can be http or https.
    - **IP** – IP address of the MU where the web services are located.
    - **Port** – the port through which the MU web services can be accessed.
    - **ServicePath** – path within the MU where the web services are located.

- **Event** – the event subscription that is going to be activated or deactivated.
    - **Name –** name of the event.
    - **Variable –** the variable to be monitored by the event.
        - **Name –** name of the e-balance variable.
        - **Properties** – indicates the exact properties of the e-balance variable that are going to be included in the notification. If it is set to null, all the properties will be included.
        - **Condition** – defines the condition that is checked on the monitored variable. If the condition is satisfied then the subscriber receives a notification. It cannot be null.

- **EventCallBack** – name of the service through which the notifications about the event are provided.

- **OperationType** – indicates if the event is going to be activated or deactivated (subscribing or unsubscribing to/from the event indicated in the second argument).

DTO response:

- **OperationResults.**
    - **OpCode –** indicates a code to describe the result of the operation in a concise way.
    - **Info –** provides a descriptive message about the operation.
    - **Success –** indicates if the service was successfully executed.

EXAMPLE:

Data Owner			
Protocol	**IP**	**Port**	**ServicePath**
http	192.168.43.98	2554	/

Event			
Name	**E-balance Variable**		
ENERGY CONSUMPTION LAST VALUE	**Name**	**Properties**	**Condition**
	ENERGY_CONSUMPTION	Null	Null

EventCallBack
notifyEnergyConsumption

Operation Type
subscribe

4.3.2.4 Periodic

DTO Request:

- **RequestSource –** identifier of the caller; both the device and the service/stakeholder. A web service URL is composed of the following fields.
    - **Protocol** – it can be http or https.
    - **IP** – IP address of the MU where the web services are located.
    - **Port** – the port through which the MU web services can be accessed.
    - **ServicePath** – path within the MU where the web services are located.

- **Periodic** – the periodic subscription that is going to be activated or deactivated.
    - **Name –** name of the periodic.
    - **Period –** indicates the frequency for the accesses (reading of the variable).
    - **Variable –** the variable to be covered by the periodic subscription.
        - **Name –** name of the e-balance variable.
        - **Properties** – indicates the exact properties of the e-balance variable that are going to be included in the notification. If it is set to null, all the properties will be included.
        - **Condition** – allows restricting the delivered tuples only to ones satisfying the condition.

- **EventCallBack** – name of the service through which the notifications are provided.

- **OperationType** – indicates if the subscription is going to be activated or deactivated (subscribing or unsubscribing to/from the periodic indicated in the second argument).

DTO response:

- **OperationResults.**
    - **OpCode –** indicates a code to describe the result of the operation in a concise way.
    - **Info –** provides a descriptive message about the operation.
    - **Success –** indicates if the service was successfully executed.

EXAMPLE:

Data Owner			
Protocol	**IP**	**Port**	**ServicePath**
http	192.168.43.98	2554	/

Periodic				
Name	**Period**	**E-balance Variable**		
		Name	**Properties**	**Condition**
ENERGY CONSUMPTION 5000S VALUE	5000	ENERGY_CONSUMPTION	Null	Null

EventCallBack
notifyEnergyConsumption

Operation Type
subscribe

4.3.3 Data Functional Interface

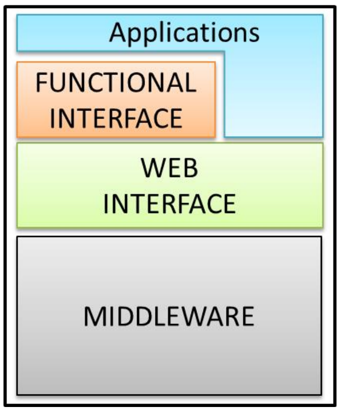

Figure 14 – Web and Functional Data Interface

All the functions provided by the data interface are based on RESTful web services so that two management units (MUs) will be able to interact with each other to exchange or store data through web services executed over IP. This kind of interface is quite restrictive in the sense that it only allows users to carry out unicast addressing. To solve this issue a functional interface on the top of the web interface has been created (see Figure 14). It does not only allow to do broadcast (for example, a LVGMU could request information from several CMUs) but it also improves the usability of the data interface when it is used from internal e-balance software modules (e-balance applications).

4.3.3.1 Configuration

Any application that is going to use the e-balance middleware must setup the following parameters: application credential and address of the management unit where e-balance middleware will be running. It means that an application cannot use the middleware if an administrator of the system has not created a user for this application.

```
void SetCredentials(string user, string pass)
```

```
void SetMUAddress(EBUrl localAddress)
```

4.3.3.2 Write

Unicast call to store the value of an e-balance variable on a particular MU.

```
WriteResponse Write(EBUrl destinationMU, EBVariable ebVariable)
```

Broadcast call to store the value of an e-balance variable on several MUs specified by the destinationMUs parameter.

```
WriteResponse[] Write(EBUrl[] destinationMUs, EBVariable ebVariable)
```

Broadcast call to store the value of an e-balance variable on all the child MUs of the MU that makes the call.

```
WriteResponse[] WriteInChildMUs(EBVariable ebVariable)
```

4.3.3.3 Query

Unicast call to read the latest instance of an e-balance variable from a particular MU.

```
QueryResponse Query(EBUrl destinationMU, EBVariable ebVariable)
```

Broadcast call to read the latest instance of an e-balance variable from several MUs

```
QueryResponse[] Query(EBUrl[] destinationMUs, EBVariable ebVariable)
```

Broadcast call to read an e-balance variable from all the child MUs of the MU that makes the call.

```
QueryResponse[] QueryFromChildMUs(EBVariable ebVariable)
```

4.3.3.4 Event

Unicast call to subscribe to an event on a particular MU.

```
EventResponse SubscribeToEvent(EBUrl destinationMU, EBEvent ev, string serviceCallBack)
```

Broadcast call to subscribe to an event in several MUs

```
EventResponse[] SubscribeToEvent(EBUrl[] destinationMUs, EBEvent ev, string serviceCallBack)
```

Broadcast call to subscribe to an event on all the child MUs of the caller MU.

```
EventResponse[] SubscribeToEventInChildMUs(EBEvent ev, string serviceCallBack)
```

Unicast call to unsubscribe of an event from a particular MU.

```
EventResponse UnsubscribeFromEvent(EBUrl destinationMU, EBEvent ev)
```

Broadcast call to unsubscribe of an event from several MUs.

```
EventResponse[] UnsubscribeFromEvent(EBUrl[] destinationMUs, EBEvent ev)
```

Broadcast call to unsubscribe of an event from all the child MUs of the MU that makes the call.

```
EventResponse[] UnsubscribeFromEventInChildMUs(EBEvent ev)
```

4.3.3.5 Periodic

Unicast call to subscribe to a periodic event in a particular MU.

```
PeriodicResponse SubscribeToPeriodic(EBUrl destinationMU, EBPeriodic periodic, string serviceCallBack)
```

Broadcast call to subscribe to a periodic event on several MUs

```
PeriodicResponse[] SubscribeToPeriodic(EBUrl[] destinationMUs, EBPeriodic periodic, string serviceCallBack)
```

Broadcast call to subscribe to a periodic event on all the child MUs of the caller MU.

```
PeriodicResponse[] SubscribeToPeriodicInChildMUs(EBPeriodic periodic, string serviceCallBack)
```

Unicast call to unsubscribe of a periodic event from a particular MU.

```
PeriodicResponse UnsubscribeFromPeriodic(EBUrl destinationMU, EBPeriodic periodic)
```

Broadcast call to unsubscribe of a periodic event from several MUs

```
PeriodicResponse[] UnsubscribeFromPeriodic(EBUrl[] destinationMUs, EBPeriodic periodic)
```

Broadcast call to unsubscribe of a periodic event from all the child MUs of the MU that makes the call.

```
EventResponse[] UnsubscribeFromPeriodicInChildMUs(EBPeriodic periodic)
```

4.4 Module implementation

This section provides a high level description of the way how each of the modules that compose the middleware has been implemented.

4.4.1 Request processor module

Through this module the users/stakeholders registered in the system and their applications will be able to interact with the e-balance variables – data stored in the database. To do that, the request processor module provides four services: write, query, event and periodic services. In the following, it is explained how each one of them is implemented and working.

4.4.1.1 Query Service

When the middleware receives a query request DTO, it checks if the user issuing the request has permissions to interact with the e-balance variable indicated in the DTO. If the user has permissions, the middleware retrieves from the database module the data related to the query specified (variable, properties and condition) in the request DTO.

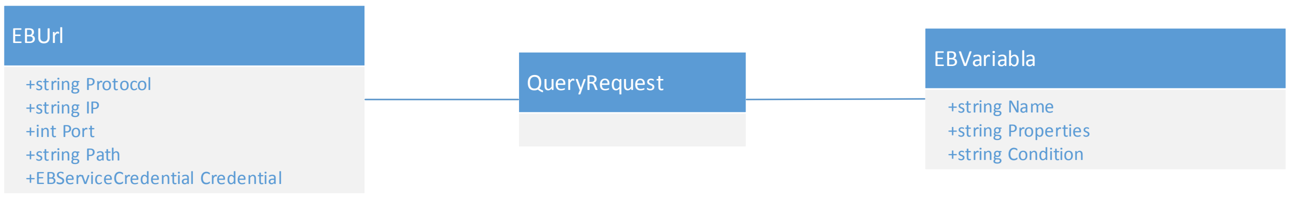

Figure 15 – Query request DTO

4.4.1.2 Write Service

When the middleware receives a write request DTO, it checks if the user issuing the request has permissions to modify the e-balance variable indicated in the DTO. If the user has permissions, the middleware will store in the database module a new value for the e-balance variable.

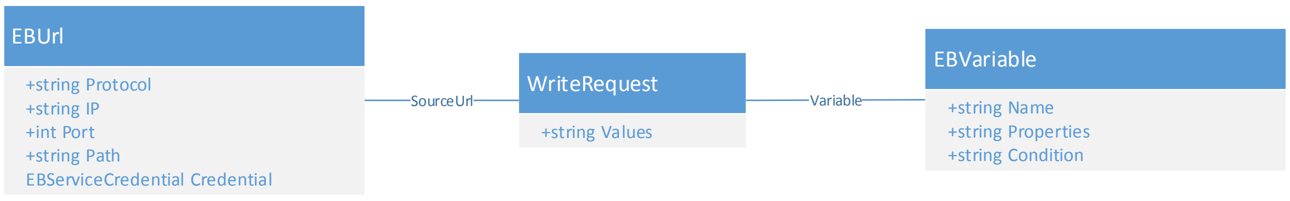

Figure 16 – Write request DTO

4.4.1.3 Periodic Service

When the middleware receives a periodic request DTO (see Figure 17), it checks if the requester has permissions to interact with the e-balance variable as specified in the DTO (read access with a given frequency). If the user has permissions, the system analyses if the requester wants to subscribe to or unsubscribe from already defined subscription for periodic reading of an e-balance variable. Once the middleware checks that it is the subscription, it stores in the database module the new subscription data and a timer is created to send to the user the data as requested in the DTO in a periodical way. To send back the information periodically (notifications), the middleware uses a web service address that was specified in the DTO.

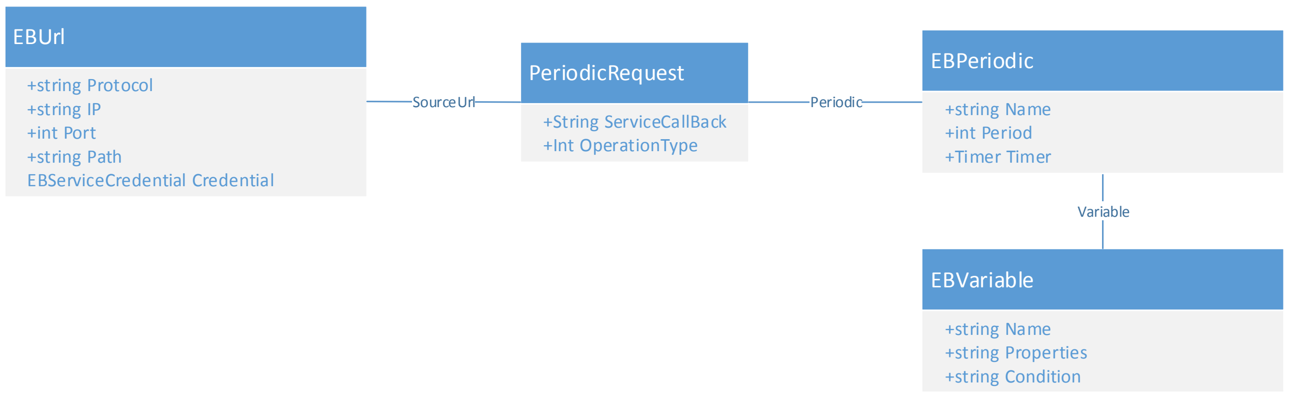

Figure 17 – Periodic request DTO

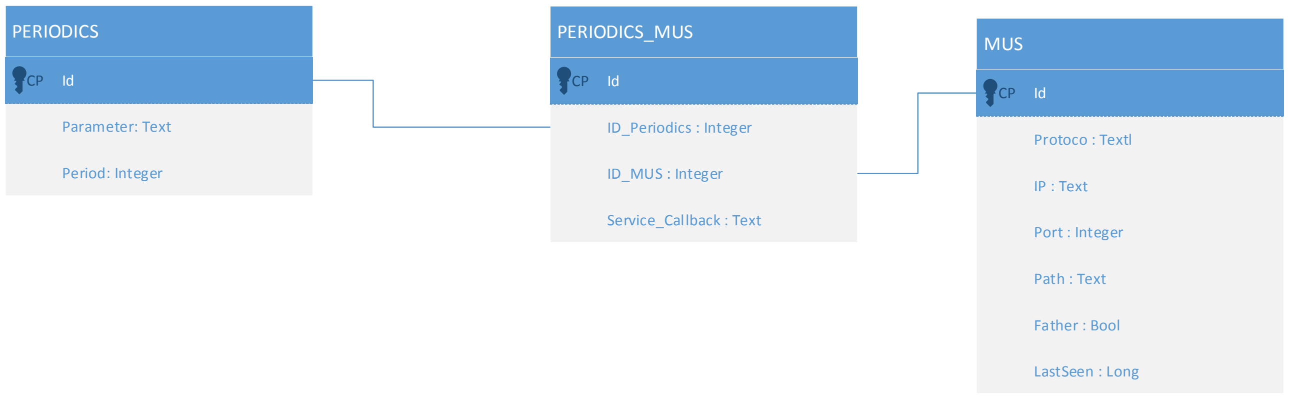

Figure 18 – Database structure to store the periodics

Figure 18 shows the tables used in the database to store the *periodics* registered in a particular management unit.

4.4.1.4 Event Service

This service works in a similar way as the periodic service. The only difference is that it only sends information back (notifications) when the condition – specified in the DTO (see Figure 19) – is satisfied. These conditions are internally checked by the middleware every 5 seconds. This parameter can be customized.

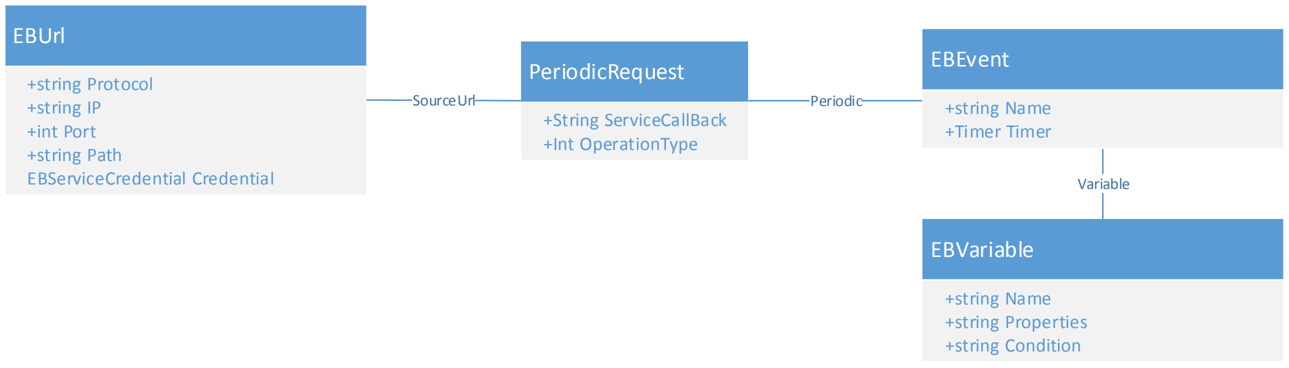

Figure 19 – Event request DTO

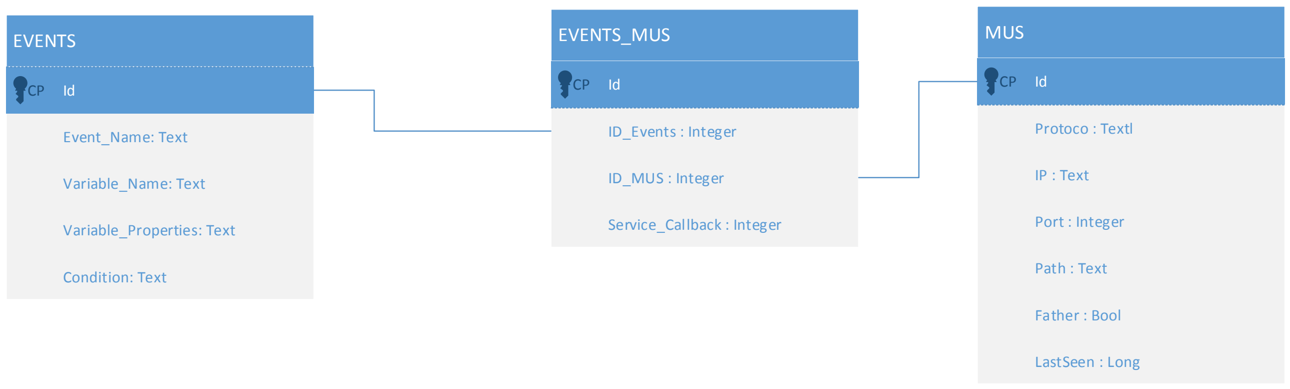

Figure 20 – Database structure to store the created events

Figure 20 shows the tables used in the database to store the *events* registered in a particular management unit.

4.4.2 Maintenance and group management module

As described in Section 3.2.5, the maintenance module manages the MUs network and keeps track of the connected devices and sensors. All the services that modify the maintenance tables are only available to operator and administrator users and are used during the configuration of the system. All the information related to the child and parent MUs are stored in the table shown in Figure 21. For example when a MU A is registered as child in the MU B, the MU A will store the information of the MU B in its MUS database table and the father field of the table will be set to true. On the other hand, the MU B will store the information of the MU A and the father field will bet set to false which means that the MU A is one of its children.

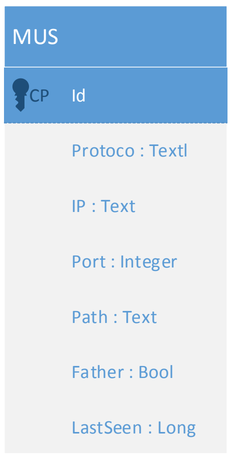

Figure 21 – Database entity to store information about child and parent MUs

The maintenance and group management module includes three main operations which are used to register, deregister and to send a hearbeat signal to a remote device.

The registration process starts by sending a registration request to a remote MU indicating the intention to become either a child or a parent of the remote MU. The request includes data about the MU initiating the registration process and also its credentials. The MU receiving the request accepts or denies the request based on the provided data. If the registration is granted then both MUs keep data about each other in the table shown in Figure 21 and communication between them is possible.

In addition deregistration functionality is provided to disable communication between two previously registered MUs. It removes the logical link between the MUs, describing the architecture and topology of the system. When deregistration takes place both involved MUs delete the data they have previously stored about each other in the database table and future communication between them is disabled.

Finally the optional heartbeat functionality is used to send notifications between two registered MUs stating that both of them are still available. These notifications are used to keep track of valid links between registered MUs. For example, heartbeat notifications can be used to detect nodes that have a failure and are no longer available. This parameter is controlled through the “LastSeen” field of the MUS database table (as depicted in Figure 21). Each time a MU receives a heartbeat signal, it updates the value of the “LastSeen” field.

4.4.3 Data persistence module

The data persistence module holds all the information about the e-balance system. This module is only expected to be accessed by other modules in the system and not by users through the GUI. Because of that the access to the services of the data persistence module is protected and only administrators can use them.

The data persistence module implemented in the prototype is based on relational databases. The database used on the CMU and LVGMU is implemented using an embedded database called SQLite. A database access API has been implemented over SQLite and is offered both with the functional data interface and the web interface. All modules requiring access to the database use this database access API. This allows decoupling the concrete database implementation chosen from the implementation of the rest of modules. For example, in this prototype SQLite has been chosen but if higher performance is needed a more complex solution such as MySQL can be used without modifying the implementation of other modules.

4.4.4 User Management module

A system to create and manage users is already available within the ServiceStack framework. It can be used to assign roles to the users, but it has also other interesting features. Depending on the roles the users have, they are able or not to execute the services offered by the system. The user management module has been developed on the top of this functionality. This module adds to the middleware four new services which can only be executed by the system administrator.

4.4.4.1 Create User Service

Each time the middleware receives a request (See Figure 22) to create a new user, it first checks in the user repository (table) of ServiceStack, if the new username is available in the system. If it does not exist, it is stored in the user repository and not in the database module. The middleware will not only store the username, but it will also store the full name, the email, the role and the type of stakeholder. That is, a user can be classified with two different roles: operator or stakeholder. If the former is assigned to a particular user the StakeholderType field must be empty, however if the latter is used, the StakeholderType can be one of the following types: DSO, customer, energy supplier, etc.

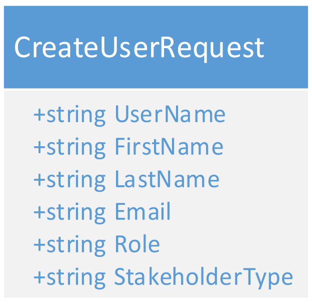

Figure 22 – CreateUser request DTO

4.4.4.2 Delete User Service

This service allows users with high privileges to delete a user previously created in the system by using the username. If the user repository contains a user whose name is equal to the name indicate in the request DTO, it will be deleted from the system.

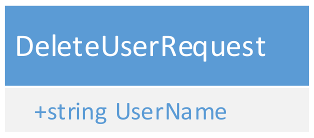

Figure 23 – DeleteUser request DTO

4.4.5 Data access control

This module control what users can read or write information into/from the e-balance variables. It is controlled through a mapping between users, permissions and variables. Figure 24 shows the database entities used to store this information.

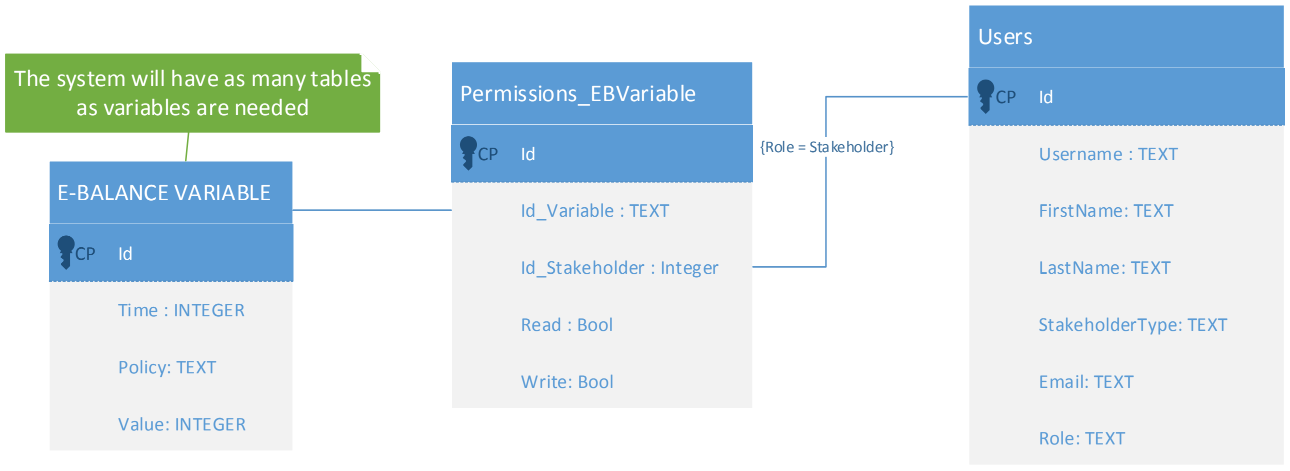

Figure 24 – Database to store the permissions users have on the e-balance variables

Thus, once a user tries to read or write an e-balance variable, the middleware checks in the database module if the user has permissions to do that. If user does not have permissions, she will be notified about that.

References

[1] SQLite. Self-contained, serverless, zero-configuration, transactional SQL database engine. https://www.sqlite.org/

[2] Service Stack LLC. ServiceStack framework. https://servicestack.net/