ebalanceplus

Data exchange middleware specification

Deliverable D5.3

AUTHORS : KRZYSZTOF PIOTROWSKI
IGOR KOROPIECKI
JAIME CHEN

DATE : 28.01.2022

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grand agreement N°864283

Technical References

Project Acronym	ebalance-plus
Project Title	Energy balancing and resilience solutions to unlock the flexibility and increase market options for distribution grid
Project Coordinator	CEMOSA
Project Duration	42 months

Deliverable No.	D5.3
Dissemination level [1]	PU
Work Package	WP5 Communication Platform and system integration
Task	T5.3 Data exchange middleware
Lead beneficiary	SOF
Contributing beneficiary(ies)	IHP
Due date of deliverable	31 January 2022
Actual submission date	28 January 2022

[1] PU = Public
PP = Restricted to other programme participants (including the Commission Services)
RE = Restricted to a group specified by the consortium (including the Commission Services)
CO = Confidential, only for members of the consortium (including the Commission Services)

Document history

V	Date	Beneficiary	Author
0	21/07/2021	SOF	Jaime Chen
1	26/11/2021	IHP	Krzysztof Piotrowski, Igor Koropiecki

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

Summary

1.1 Summary of Deliverable

This document describes the data exchange middleware, which oversees distributing the information generated in the system to make it available to the different algorithms. It is also the middleware's responsibility to adapt different communication protocols so that information from a diverse number of external devices can be stored. The document presents the architecture of the system and provides implementation details of two important concepts: the middleware, which offers a data-centric API to access the data and the adapters, which convert information from external devices into data that is understandable by the middleware, and consequently by the algorithms.

Disclaimer

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

Table of Contents

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

Table of tables

Table of figures

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

List of Abbreviations

Abbreviations	Definitions
CMU	Customer Management Unit
DER	Distributed Energy Resources
DERMU	Distributed Energy Resources Management Unit
HVAC	Heating, Ventilation and Air Conditioning
LVGMU	Low Voltage Grid Management Unit
MU	Management Unit
MVGMU	Medium Voltage Grid Management Unit
PS	Primary substation
SCADA	Supervisory Control And Data Acquisition
SS	Secondary substation
TLGMU	Top Level Grid Management Unit
VM	Virtual Machine

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

1 Introduction

The main goal of the project is to increase energy flexibility of distribution grids, predict available flexibility, increase distribution grid resilience and design and test new ancillary models to promote new markets based on energy flexibility. These objectives allow unlocking the energy flexibility market in distribution grids to support energy prosumers and electric operators. To achieve that goal, a crucial requirement is that the different entities that form the system need to be able to exchange information and to store it. The following sections present the software responsible to provide such functionalities which is called the data exchange middleware or the ebalance-plus middleware interchangeably throughout the document.
Section 2 presents and overview of the ebalance-plus architecture from a communication point of view. Section 3 details the design of the data exchange middleware. Section 4 introduces the adapter and module abstraction used to connect the middleware to external data providers. Sections 5 and 6 provide insights on the implementation of the middleware and adapters respectively. Finally, Section 7 presents the conclusions.

2 The ebalance-plus architecture

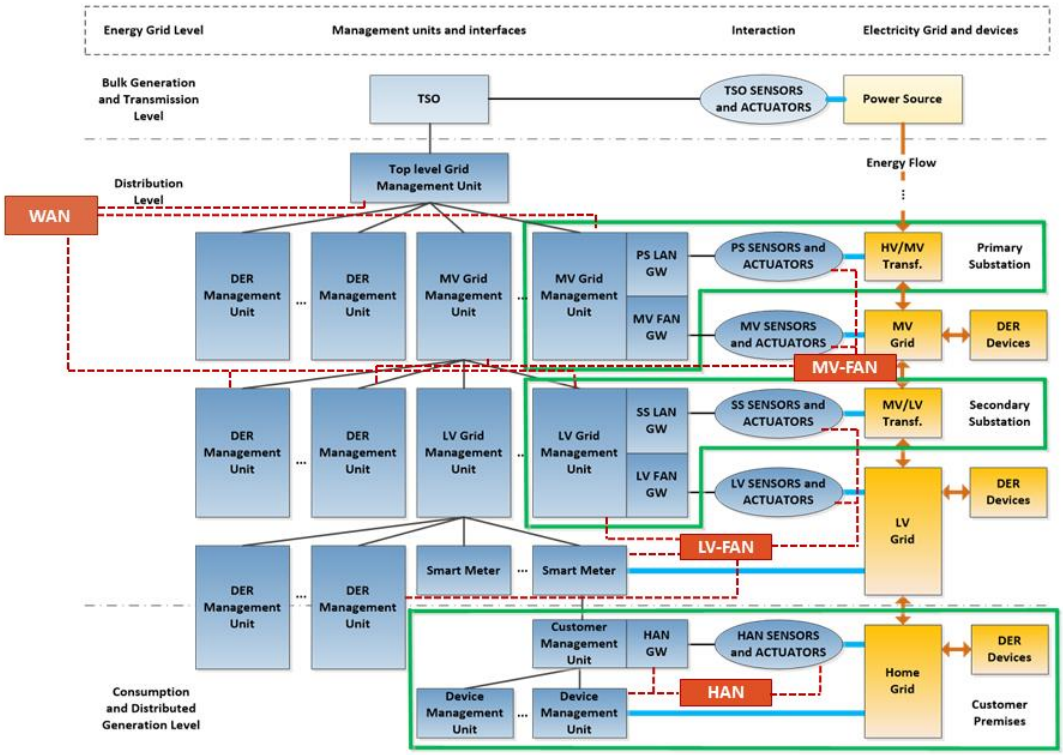

Figure 1 Example of the ebalance-plus architecture

The ebalance-plus system is composed of units, called management units or MUs, that implement algorithms to forecast and manage the available flexibility to incentivise demand response programmes and increase the distribution grid capacity to avoid congestions and advise optimization strategies.
In the proposed architecture each MU is considered an autonomous system, such as an embedded PC, although other options are possible, such as virtualizing the MUs in containers or VMs. The units are organized following a tree hierarchy where each MU has a parent MU and possibly multiple children MUs except for the top-level grid management unit (TLGMU)

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

which is the MU located at the root of the tree. Figure 1 illustrates the relationship between the different MUs (depicted as rectangular dark blue boxes) in a scenario that has four main levels (enumerated from the top of the tree to bottom):

- **TLGMU:** top level grid management unit located, for example, in the cloud
- **MVGMU:** medium voltage grid management unit located in the primary substation (PS)
- **LVGMU:** low voltage grid management unit located in the secondary substation (SS)
- **CMU:** customer management unit located in the customer premises, buildings, etc.
- **DMU:** device management unit located inside external devices that communicate with the system

In addition, distributed energy resource management units (DERMUs) can be deployed at any level to management DER devices located at different levels of the energy grid.
Each unit must be able to exchange information with external devices and with each other to distribute the information generated in the system and transport it to the data consumers which are most often the ebalance-plus algorithms although it can also be GUI applications, SCADA/monitoring applications, etc. Also, data must be stored and consulted at any time prior to its generation. The software that provides such functionality is the **data exchange middleware**, which is a key concept in the architecture. Another key concept is the **adapter module** which assists the middleware in contacting external devices and retrieving the information from them or modifying available setpoints. For example, an adapter for a smart battery might be used to get information from the battery state and to change the charging schedule. This functionality is used by algorithms through the API provided by the data exchange middleware. From the point of view of the algorithm there is not external device, only pieces of data that can be written or set. See Chapter 3.2 to get more details about the data interface abstraction provided by the middleware.
Although Figure 1, that has been used to present the reference architecture of the system, shows a scenario with four levels, the hierarchical nature of ebalance-plus makes its flexible and the architecture can be adapted to each specific scenario. Some of the variables that can be modified based on the scenario requirements are specified in Table 1.

Table 1 ebalance-plus deployment variables

Variable	Description
Type of MUs	DMU, CMU, DERMU, etc
Number of levels in the hierarchy	Depth of the tree
Number of devices per level	Customizable per parent MU. Each MU can have as many children MU as needed
MU hardware	Virtualized (container, VM), deployed in a PC or embedded PC
MU connectivity	Through a proxy server, direct connection, socket-based connection, http-based connection, etc
Number and types of adapters per MU	Customizable per MU

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

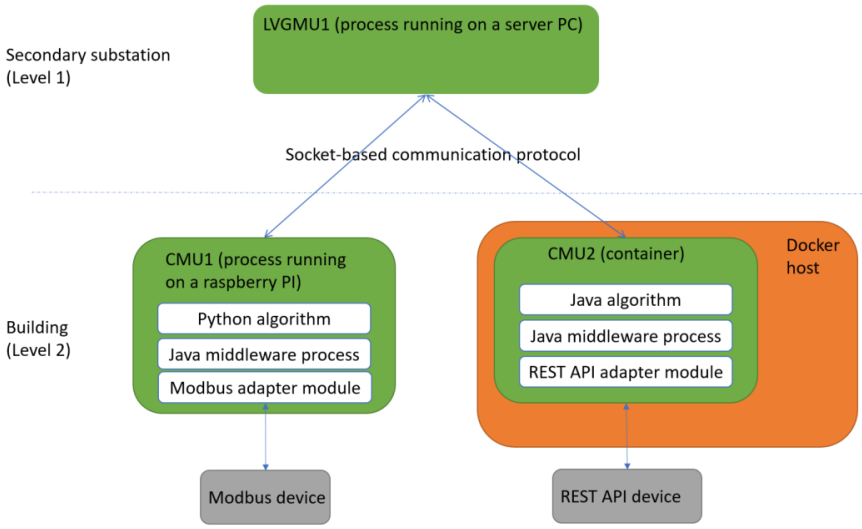

Figure 2 Simplified view of a test ebalance-plus deployment

Figure 2 shows a simplified view of an example architecture from a communication point of view with the summarized features shown in Table 2. In this scenario there are 3 different MUs but deployed in different ways. LVGMU1 is deployed in a conventional server where as CMU1 and CMU2 are deployed in a raspberry PI and a Docker host respectively. LVGMU1 is the parent of CMUs 1 and 2 and can communicate with them using a direct socket-based connection (automatically handled by the middleware). Moreover, since the middleware provides middleware client libraries in different programming languages, the algorithm can run on Python (in CMU1) or in Java (in CMU2 and LVGMU1). Finally, external devices can be integrated in the middleware via adapter modules. CMU1 has a Modbus adapter module and CMU2 has a REST API device module. There is no restriction about the number of adapter modules that can be connected to a MU.
This scenario shows the flexibility of the proposed architecture and highlights the customizable nature of ebalanceplus. For example, if the deployment takes place in an outdoor environment with little physical space or in a customer premise an embedded device is the most sensible choice.

Table 2 Deployment specification of the example depicted in Figure 2

Variable	Description
Type of MUs	2 CMUs (CMU1 and CMU2) and 1 LVGMU (LVGMU1)
Number of levels in the hierarchy	2
Number of devices per level	2 at customer level and 1 at low voltage grid level
MU hardware	CMU1 is deployed in an embedded PC, CMU2 is virtualized in a container and LVGMU1 is deployed in a conventional server
MU connectivity	direct socket-based communication between MUs
Number and types of adapters per MU	customizable per device

Due to the high number and the autonomous nature of heterogeneous devices that compose the system, the data exchange middleware needs to handle different protocols, APIs, programming abstractions and data formats. On the other hand, from the point of view of the algorithms it is ideal that all the data is obtained, modified using a single programming abstraction that represents the data in a uniform way. The data exchange middleware fills the

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

gap between data providers (external devices, sensors, etc) and data consumers (algorithms, GUI apps, mobile apps) and provides storage and historical access to the data via queries.

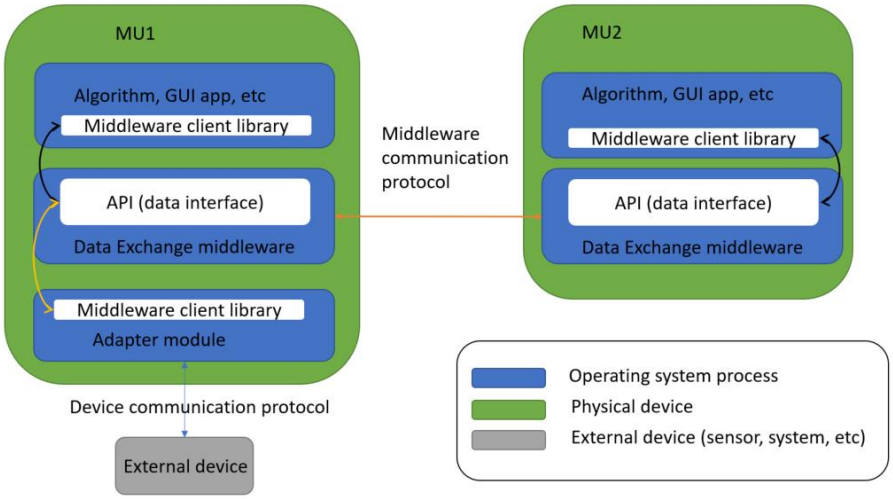

Figure 3 Information exchange in ebalance-plus

A simplified view of two MUs exchanging information is depicted in Figure 3. Two algorithms (running in MU1 and MU2) can exchange data (orange and black arrows) using the middleware and more specifically using its API (data interface). The use of the API is simplified by using a middleware client library which internally uses the data interface and offers high levels calls in a specific programming language. Currently, two client libraries are provided together with the middleware, one in Python and the other one in Java. In the same way, external devices share information with the middleware using the data interface. This means that all communication either between different units (orange arrows), between an algorithm and the middleware (black arrows) and between an external device and the middleware (yellow arrows), is performed using the same API. This approach simplifies the development of applications and make them portable between different units. Two important concepts allow information to be available in the whole system:

1. **The middleware:** Allows algorithms to read/write data from MUs (see Chapters 3 and 5)
2. **Adapter module:** Allows external devices to be integrated in the middleware see Chapters 4 and 6)

3 Data exchange middleware design

3.1 Middleware architecture

The ebalance-plus system uses a middleware framework that allows the participants to communicate, exchange and store information. The framework stores data in tuple space structures which allows to implement a variety of logical structures that can contain all the information necessary to identify a value, its description, source, and time of creation. The tuple space is accessed by creating variables that can be written, read, or removed. Each variable is further divided into owner spaces. Operations on variables are checked against defined access control policies. By default, each owner can control only their own data. It is possible to grant or revoke permissions to change the default permissions. The variables concept implemented by the framework varies slightly from the general understanding of a variable. The first difference is that by default, each value written to the variable is stored as a separate entry. The second is that data stored in a single variable can have multiple owners. The third difference is that a variable can have multiple fields that are defined when creating the variable. For example, it is possible to create a weather variable that contains fields such

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

as temperature, humidity, or wind. Then, owners can store and share their weather information as they please. The architecture of the framework follows a distributed approach. It allows the participants to store data close to the locations where the data is produced and/or where it might be consumed in order to be processed. The communication is secured through public key infrastructure (PKI).
The framework defines the Data Interface channel that is responsible for establishing a secure and transparent channel for communication between clients and a middleware server instance. The Data Interface channel is available as a library either in Java or Python. In the context of the middleware framework, clients that use the library to communicate and implement functionalities, are referred to as services (see Figure 4). The services use the Data Interface to perform actions on data (variables), permissions and use other system functionalities.

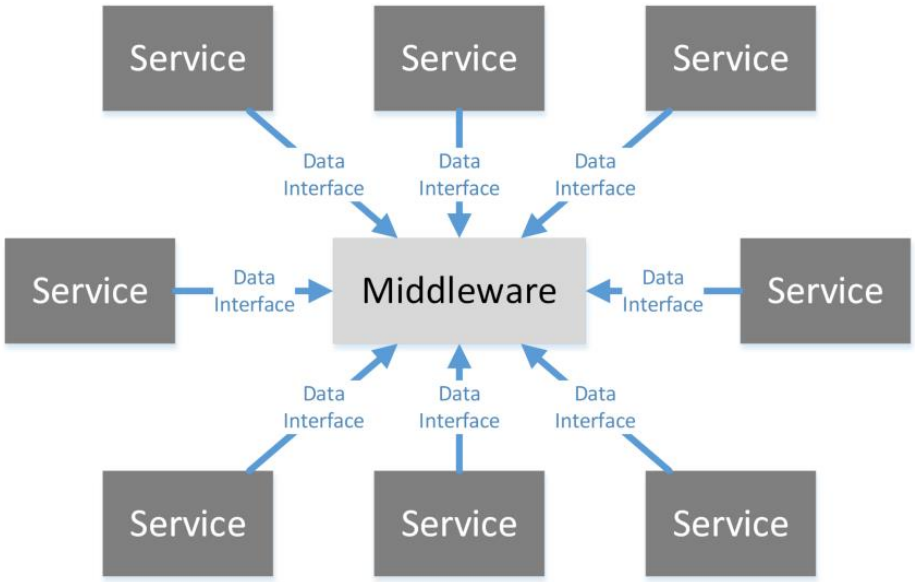

Figure 4 Data exchange middleware services

The services are generic applications and by default, there are no limitations (other than limitations enforced by the operating system) in actions they may perform. In cases where a single machine runs a middleware instance and services that belong to a single user or services that communicate with the middleware server are run on physically separated hardware from the middleware and each other, the threat is minimal. However, in cases where a single machine hosts the middleware and services that belong to different users, there is a risk of malicious services that have the possibility to obtain confidential information such as private keys, databases, passwords, or source code of services that belong to competing users.
To protect against malicious services, the middleware platform implements a utility that is responsible for:

- **bootstrapping secure environments for services.**
- **running services with adequate privileges.**

The middleware servers can communicate directly or through a framework-provided proxy server (see Figure 5). If both servers are visible to each other, the communication can be direct. If the destination server is not visible to the source server, the communication must be executed through a proxy server.

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

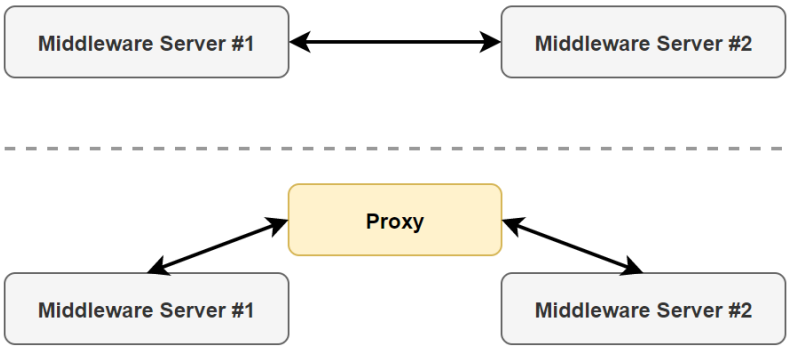

Figure 5 Communication between different instances of the middleware

When two middleware servers communicate, a single channel is opened. In this case, only the destination server has to be visible. The response is sent through the same channel as the request.

3.2 Middleware data interface and API

The middleware framework includes a library for the services to integrate with the platform. The library contains an API called the Data Interface (see Figure 6) which is responsible for providing a transparent and secure communication channel.

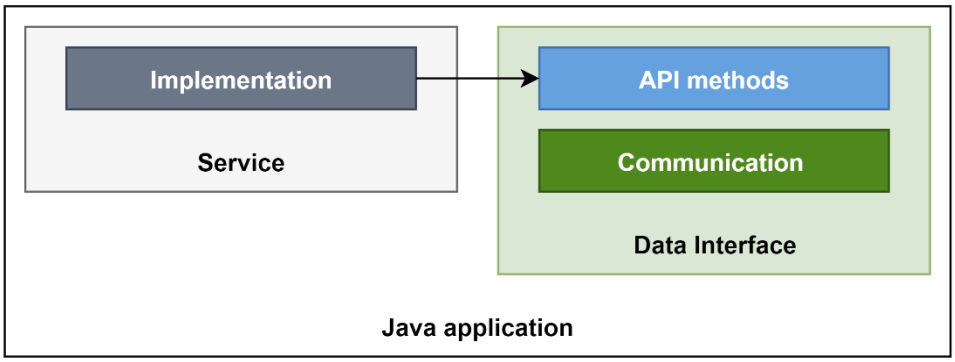

Figure 6 Data Interface in the context of a Java application

The Data Interface allows a service to:

- **Create variables** – when creating a variable, it is possible to specify the name, type, additional fields, and quota limits.
- **Write, read, update, clear values to/in a variable** – services can perform any operation that is necessary for data management in the permitted owner spaces.
- **Remove variables** – when no other owners have data in the variable, the variable can be removed.
- **Create and revoke permissions** – it is possible to subscribe to changes in the permitted owner spaces. Conditional notifications are also possible.
- **Create and remove subscriptions** – owners can subscribe to changes in other owner's data spaces.
- **System information** – services can obtain information about the middleware server that they are connected to.

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

3.3 Middleware utilities

The middleware framework offers two utilities for management. The first utility – MWGUI is meant for the non-technical users and allows to browse and visualize the data that the user owns. Also, using this utility, the user can manage data access permissions for other stakeholders. It is also possible to subscribe to data changes in context of the current user. When subscribed, any data writes by other stakeholders in the name of the user, will update the interface. The utility supports browsing large amounts of historical data by using pagination. Users can specify the desired date range and the interface displays data grouped by the chosen range.

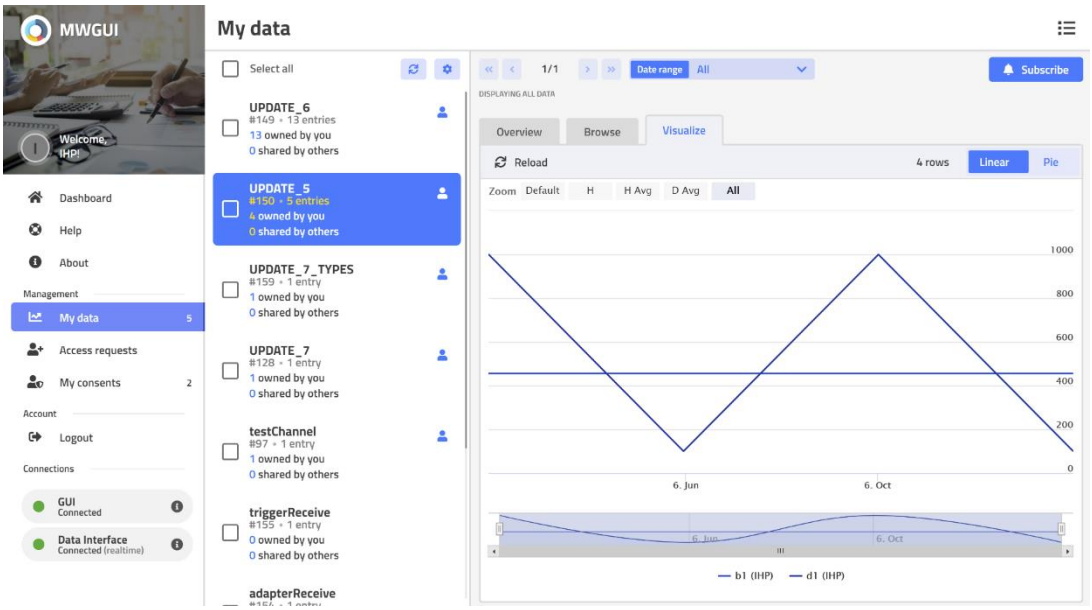

Figure 7 MWGUI

The second utility – AdminGUI is meant for the technical users and can be used to manage middleware server instances. The utility is hosted by the middleware server that the user wants to manage. It is possible to manually enter connection information to other middleware servers and manage multiple instances from a single instance of the AdminGUI.

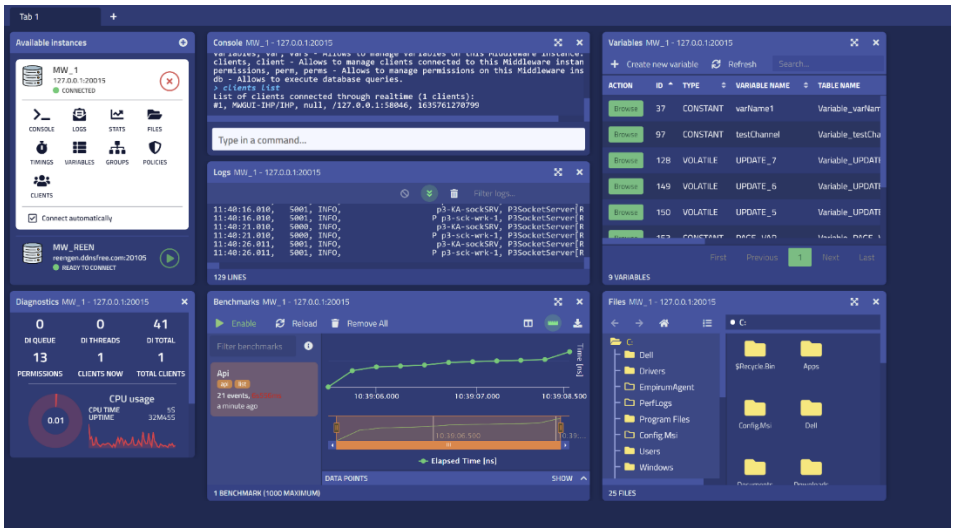

Figure 8 AdminGUI

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

The AdminGUI implements multiple widgets that make it possible to manage the middleware instance. The widgets include: Console, Logs, Diagnostics (processor, memory, disk), File Browser, Benchmarking, Data browsing (variables, permissions, routing table) and Clients. The interface allows to open widgets in tabs which allows to open as many widgets as necessary to manage the server. The user can, for example, open a new tab for each managed server. Access to the GUI is protected and each widget server-side logic implements measures to make the interface as secure as possible. For example, the file browser can be configured to constrain directory access up to a certain path.

4 Middleware adapter modules

4.1 Introduction

Smart grids require information from a high number of devices to be distributed from the place they are generated to where the algorithms will use it. Example of devices that provide useful information for the intelligent algorithms are:

- **Power inverters**
- **Energy storage systems**
- **Electrical vehicles and charging stations**
- **Smart meters**
- **HVAC systems**

The data generated by each of these devices might be needed in an algorithm running on one or multiple MUs. As a core design guideline, the algorithm should only get the data through the data interface of the middleware. To follow this guideline, an additional layer of abstraction is needed to convert information from all these communication protocols to data that the middleware can understand and process. In ebalance-plus, this layer of abstraction is provided by an adapter module. The adapter module makes it possible to connect a variety of external devices with the middleware. This architecture makes it possible to replace or update external devices without having to reprogram or modify the algorithms and/or the middleware.
Section 4.2 introduces the general adapter module architecture in the context of a MU whereas Section 4.3 details the internal abstractions used to implement it.

4.2 Adapter module architecture

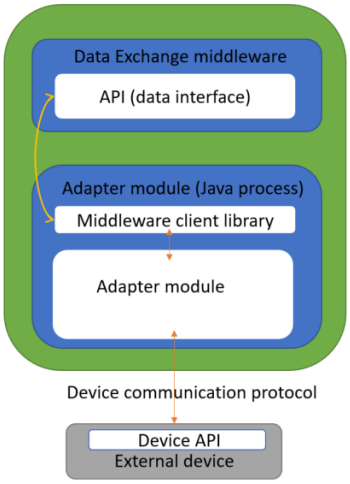

Figure 9 Adapter module architecture

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

The high-level module architecture of an adapter module is depicted in Figure 9. An adapter module is a middleware service, meaning a program that uses the middleware API with some given credentials, that runs as a Java (or Python) process and has a specific task: mapping information from an external device to data inside the middleware. The algorithm using the data exchange middleware will read/write this data but will never directly contact the external device. This indirect communication scheme simplifies the communication between all devices in the system since adapter modules can be unplugged without affecting algorithms. The communication between an external device and the middleware is bidirectional. Information from the external device is written inside the middleware (e.g., sensor data) and, in the other way around, information written in some specific variables of the middleware triggers calls in the external device to configure some setpoints (e.g., changing configuration in the external device). The communication in each direction is handled separately using different operations offered by the middleware:

- **From the external device to the middleware:** the adapter module is configured to periodically obtain information from the external device and store it in the middleware. Since many devices will probably use the same communication protocol, for example Modbus, a single generic adapter is developed for each communication protocol, and it is later adapted to each specific device.
- **From the middleware to the external device:** the communication in this direction is used to configure parameters in the external device when an algorithm requests to do so (e.g., a balancing algorithm). This communication is initiated when an algorithm writes a specific value in a middleware variable. This variable will trigger a subscription which is received by the adapter module. Then the adapter module transforms the subscription event into a request that modifies the configuration of the external device.

Following these two approaches, communication in both ways (to read/write from the device) can be achieved using the simple operations offered by the middleware API (write/subscribe operations in the middleware respectively).

4.3 The adapter and module abstractions

As previously highlighted, the number and nature of the devices that can be connected to the smart grid system is high. Software applications that need to exchange data between heterogeneous systems are particularly hard to develop. This, among others, is due to the efforts required to combine technologies and protocols with very different characteristics (e.g., database API, Web services, Modbus, MQTT), translating their semantics, while guaranteeing a certain modularity and maintainability in the solution.
To simplify information exchange between different devices the concept of adapter has been developed. An adapter describes the pieces of software aimed to abstract those external devices that produce and consume data in applications. By means of adapters, every participating device or external system can be accessed through the same minimalist set of high-level functions, regardless of the device type and the underlying technology. This is key to enhance productivity and minimize errors in applications.
Adapters are data-centric components as represented by a Java interface called IAdapter, which defines simple data functions with the following signatures:

- **IDataTime getData(String variable);**
- **void setData(String variable, IDataTime data);**
- **void setDataChangeListener(String variable, IDataChange listener);**

The getData() and setData() methods allow a variable to be read or written, respectively. The setDatachangeListener() is used to define the callback (listener) that will be called every time the specified variable changes its value. Data in adapters, in the same way as it is done in the middleware is organized in variables. Each variable is identified by a unique string and contains multiple values or columns together with a timestamp (IDataTime interface).

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

The notion of adapter allows any external system to be accessed using the same interface. It is the task of the implementation of each type of generic adapter to deal with the semantics of the communication protocol used by the external device.
It is worth noting that an adapter is a passive abstraction, which means that it only specifies a way to manipulate data in an external system but does not actually dictate how this interaction takes place. To model the dynamic part of the communication between adapters different approaches are available. On the one hand the concept of "interaction" is proposed. An "interaction" encapsulates inter-adapter communication patterns that can be reused in applications, enabling data exchange to be described in a higher-level way. On the other hand, the subscription capability of an adapter can be used to model more specific interactions between adapters. Figure 10 shows how these two approaches are used to communicate adapters.

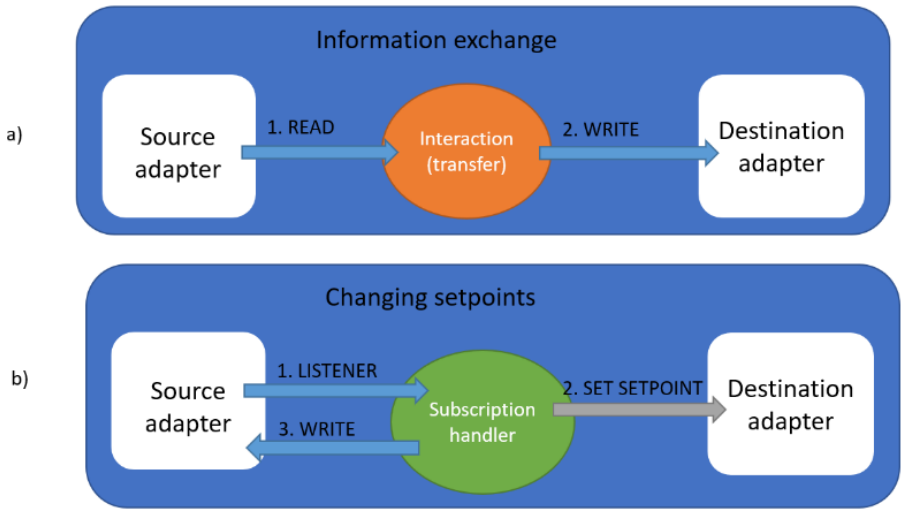

Figure 10 Adapter module architecture

The first approach, shown in Figure 10 a, is to use the concept of "interaction" to model a data exchange between the source and destination adapters. Since operations on both ends (a read and a write) are part of the interface IAdapter this type of communication can be performed with any type of adapter in a direct manner without having to implement scenario-specific code. This is usually used to get monitoring information from the source adapter and store it in the destination adapter. For example, data from a sensor can be periodically received in the middleware using a transfer interaction.
The second approach, shown in Figure 10 b, allows complex interactions, such as changing setpoints in an external device, to be modelled at the expense of having to manually handle part of the code. The interaction starts when a request signal, via a listener, is detected in the source adapter (step 1). This is notified using the subscription mechanism provided by the adapter abstraction. This listener notification is handled by a custom program, called the subscription handler, which transforms the request into something that the destination adapter understands (step 2). This does not have to be one of the operations in the IAdapter interface. It is usually device-specific code that changes the configuration of the destination device. Finally, once the operation has been carried out the result it is written back in the source adapter (step 3). This final step is used to store the result of the operation back in the source adapter.

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

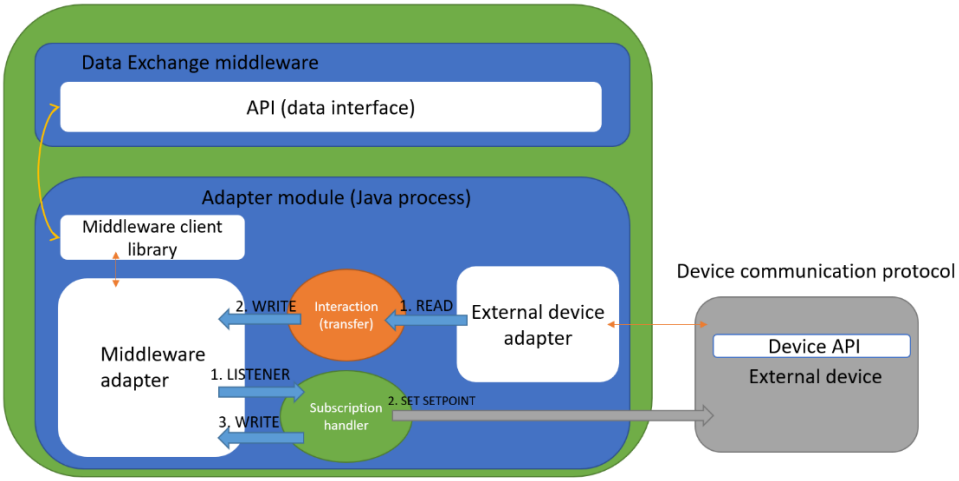

Figure 11 Adapter module in ebalance-plus

In ebalance-plus, the term “adapter module” is used to refer to the combination of adapters, subscription handlers and interactions with the goal of integrating information from an external device into the data exchange middleware. In terms of deployment, the adapter module is what it is ultimately deployed as a .jar file to handle the communication between an external system and the middleware. Figure 11 illustrates the use of all these concepts in a test scenario.
An adapter for the ebalance-plus data exchange middleware has been implemented (middleware adapter). This approach allows to treat the data exchange middleware as another adapter in the same way as an external device is and simplifies the communication between both sides. Also, an adapter for the specific external device needs to be implemented (external device adapter). These two adapters are used to model two types of data exchange:

- **Get monitoring information from the external device and store it in the middleware (approach in Figure 10 a):** A transfer interaction is used to periodically transfer information from the external device to the database of the middleware. This is used to get monitoring information from the external device.
- **Configure setpoints in the external device when a request is received in the middleware (approach in Figure 10 b):** The request in the middleware is detected in the adapter module by means of a data listener configured in the middleware adapter. The listener handler transforms the request in a call to the specific communication protocol of the external device. Finally, the result of this operation is written back in the middleware.

In the proposed architecture, each external device is controlled by an adapter module which runs in its own process. This allows external devices to be connected/disconnected at runtime from a running instance of the middleware.

5 Implementation of the data exchange middleware

5.1 Middleware server

The middleware framework implements a custom socket-based protocol optimized for performance and support for real-time and polling communication approaches. The protocol (and its security) is based on the Java implementation of the socket communication. No custom security code has been written and only the proven and reliable industry standards are used.

This project has received funding from the European Union’s Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

The middleware server also embeds the AdminGUI that can be disabled through configuration options. The Figure 12 presents a detailed overview of the middleware framework architecture.

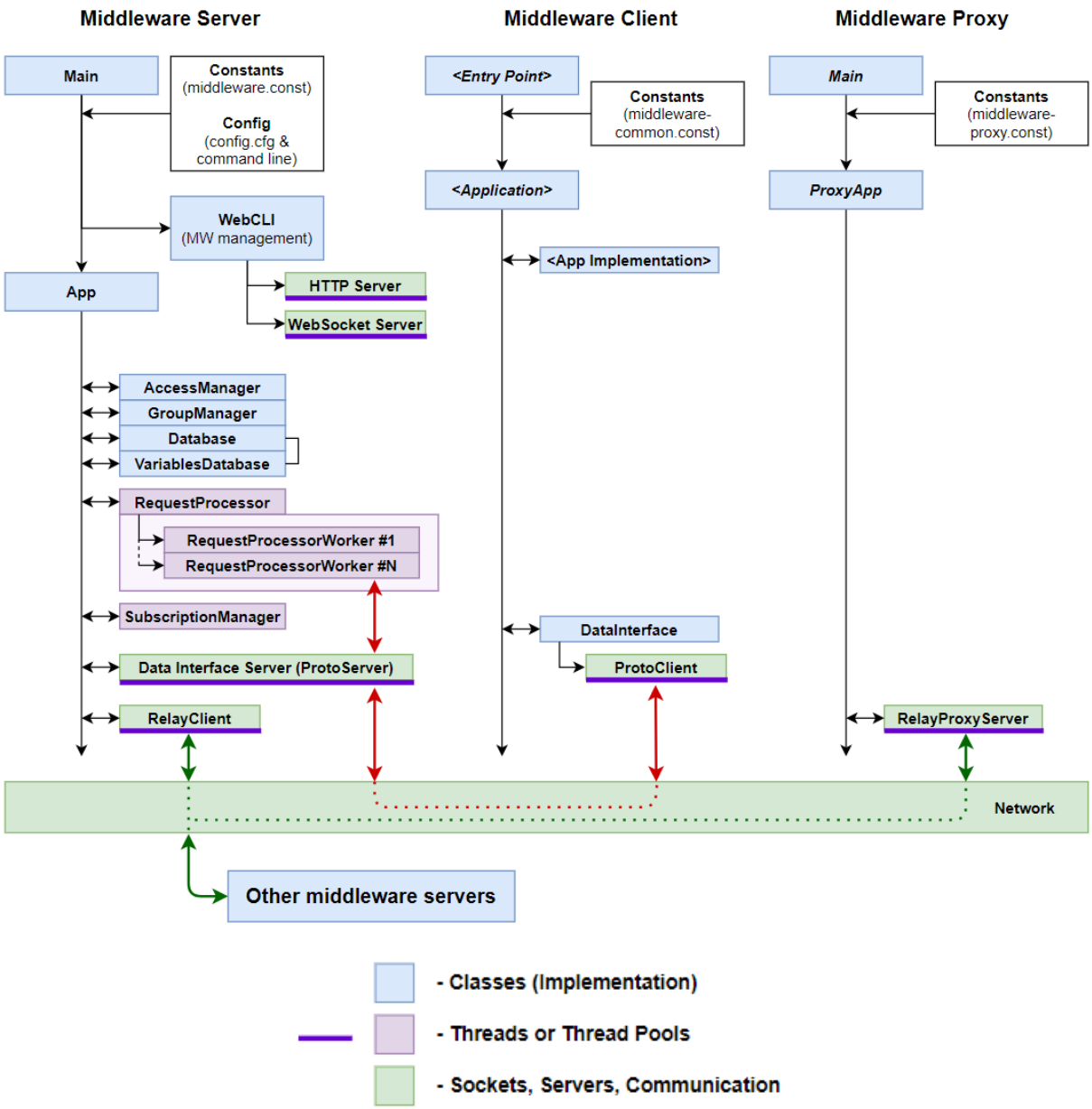

Figure 12 Middleware framework detailed architecture

The middleware server instance contains modules that communicate with each other to handle requests sent by the clients and issues requests to other middleware instances or the proxy server to handle inter-middleware requests. The modules located on the middleware server:

- **Access Manager** – the module provides functionalities related to access control of incoming requests. The request processor communicates with this module to check whether the incoming request can be processed.
- **Group Manager** – the module keeps track of hierarchy in context of the current middleware instance – who, is the parent, who are the children, what proxy servers are available. If the system based on the middleware platform does not have a hierarchy, it is possible to simply define peers without implying relations.
- **Request Processor** – the module is responsible for handling incoming requests. The middleware server can submit requests to this module. The requests are queued and

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

processed in the order that they've arrived. The request processor spawns workers to handle requests.

- **Request Processor Worker** – is spawned by the Request Processor and is responsible for handling the incoming request, validating access through Access Manager, communication with module used to redirect messages to remote middleware instances.

- **Subscription Manager** – the module is responsible for variable subscriptions and periodic notifications.
- **Database** – The module is responsible for accessing the database. Allows to issue SQL queries or manage data through defining models and issuing operations on them.
- **Variables Database** – The module communicates with the database module and is responsible for any actions related to the variables (write/read/update/etc.).
- **Data Interface Server** – responsible for receiving requests from clients. The requests are then forwarded to the Request Processor.
- **RelayClient** – responsible for communication with other middleware instances and middleware proxy server.

The diagram presented in Figure 13 shows an example transmission between a service and a remote middleware (the most complicated scenario on the platform).

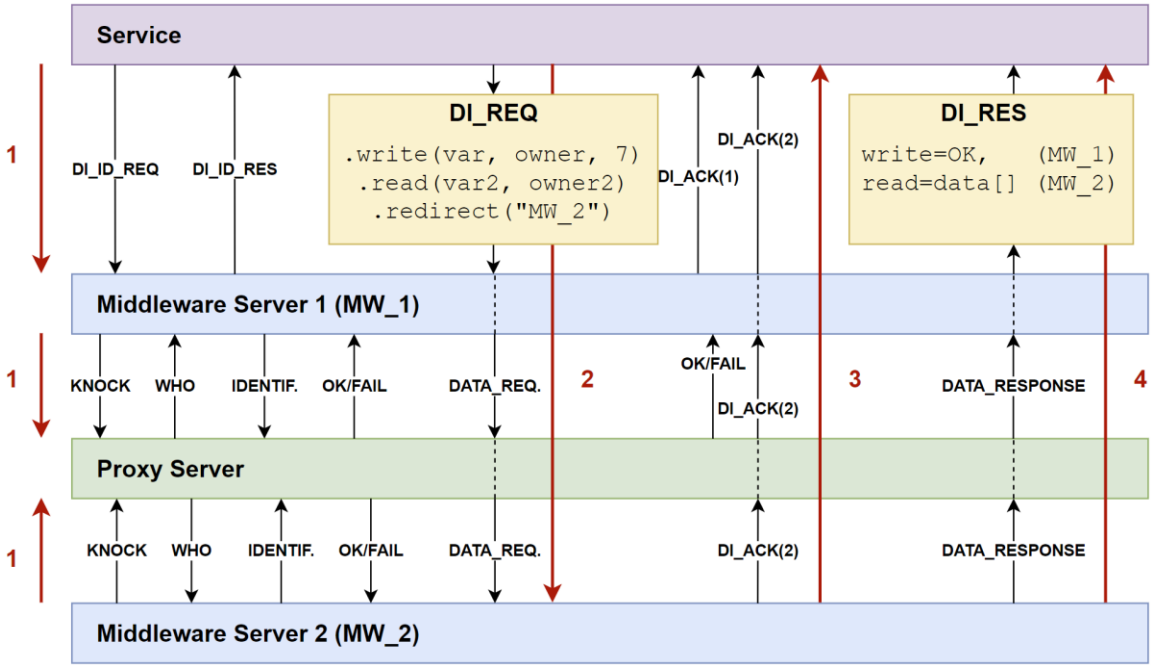

Figure 13 Example transmission with a remote middleware

At the very beginning (1), the service connects and identifies with its local middleware and all middleware server instances connect and identify with the proxy server. Successful identification creates a persistent communication channel between two participants. After some time, the service sends a request (2) to its local middleware, containing a single local write request and a single remote read request. The local middleware server sends a DI_ACK packet (3) to confirm that the whole request is queued. The local sub-request is handled by the local middleware (MW_1), while the second (remote) sub-request is sent to MW_2 through the proxy server. The MW_2 instance sends a DI_ACK packet to confirm that the request was received. After that, it handles the request and sends the response back to the source middleware (MW_1) through the proxy server. Once the MW_1 receives the response and all partial responses are completed, the final response is sent to the service that issued the request.

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

5.2 Data Interface

The Data Interface defines available methods (for actions to be performed on the platform) and ensures that required parameters are passed when sending requests to the middleware server. The programmer can register listeners to receive notifications about incoming subscriptions or the connection state of the interface. The interface allows to execute requests synchronously or asynchronously to support multiple approaches to application development. The interface allows issuing partial requests which allows to perform multiple actions on the server through a single request (example communication and partial request shown in Figure 14).

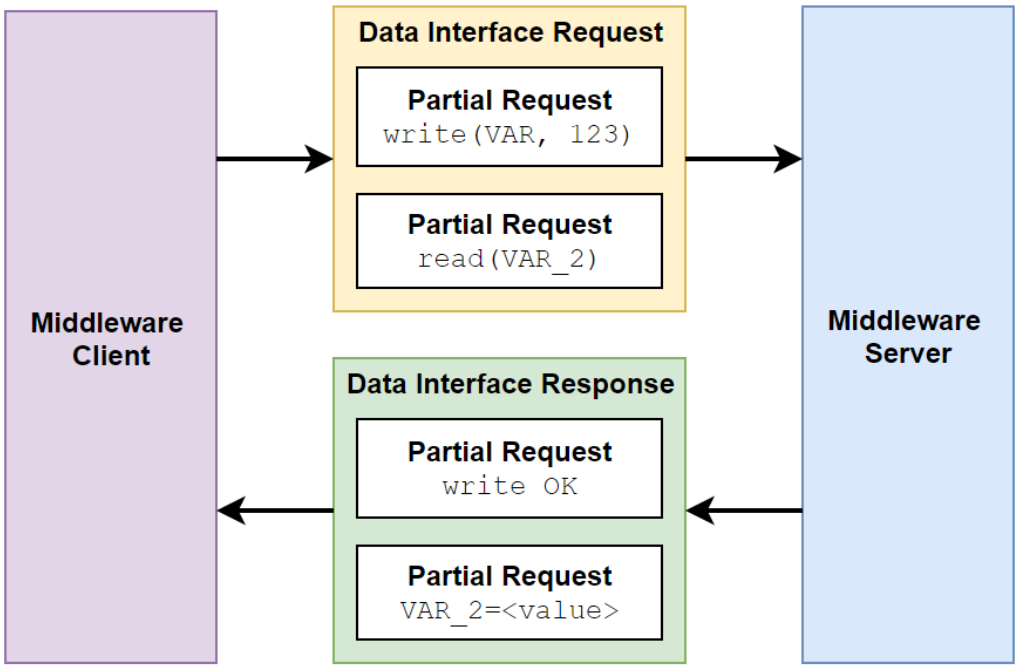

Figure 14 Communication between client and server through Data Interface

Figure 14 shows an example request sent by the client that consists of two partial requests. Each partial request represents a separate action to be performed. The sub-requests are processed consecutively, so when one is writing to a variable and then reading from it (in the same request), the just-written value will be taken into account when reading. The partial requests can be named. When creating the partial response objects, the middleware will automatically name the responses with the names given in the Data Interface request.

An example usage of the Data Interface client has been shown in Figure 15. First, the transport configuration is created. It is possible to specify the target middleware server information and provide security credentials. After the transport is configured, it can be passed further to create an instance of the Data Interface. The user can specify the service/stakeholder combination and register listeners. After the listeners are registered, the authorize() method is called that initializes the connection and conducts the authorization with the target middleware server. After the authorization is complete, it is possible to issue requests. The Data Interface allows to create a single request and state multiple actions (sub-requests) that will be executed sequentially on the destination server. The response contains multiple sub-responses that allow to observe the result of the corresponding sub-requests. Each sub-response that carries more functionality than simple status information, has a corresponding class that allows to extract the carried information. For example, when reading data from the middleware, the sub-response is paginated. Through obtaining a VariablePageResponse object, it is possible to view data on the current page and obtain more information about pagination such as current page, total rows and more.

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

```
String mwHost = ...
int realtimePort = ...
int pollingPort = ...
P3TransportConfig trConfig = P3TransportConfig.fromArgs(mwHost, realtimePort, pollingPort,
P3Transport.SOCKET_REALTIME);
trConfig.setTruststorePath("path/to/truststore");
trConfig.setTruststorePassword("storepass");
trConfig.setKeystorePath("path/to/keystore");
trConfig.setKeystorePassword("storepass");
trConfig.setKeyPassword("keypass");

DataInterface dataIntf = DataInterface.create(trConfig)
  .setDefault(DataField.QUERY_SERVICE, SERVICE_NAME)
  .setDefault(DataField.QUERY_STAKEHOLDER, STAKEHOLDER_NAME)
  .registerConnectionListener(new ConnectionListener() {

    @Override
    public void onConnect(P3Transport transport, boolean isReconnect) { ... }

    @Override
    public void onDisconnect(P3Transport transport, boolean willReconn) { ... }
  })
  .registerShutdownListener(new ShutdownListener() {

    @Override
    public void onShutdown() { ... }
  })
  .registerErrorListener(new ErrorListener() {

    @Override
    public void onError(String errorMessage) { .. }
  })
  dataIntf.registerSubscriptionListener(new SubscriptionListener() {

    @Override
    public void onSubscription(SubscriptionInfo info) { ... }
  })
  .authorize();

if(dataIntf.isAuthorized()) {
  IDataInterfaceRequest req = dataIntf.createRequest();
  req.getSystemStatus(DataInterface.LOCAL).as("status");
  req.read("varName", "stakeholder", DataInterface.LOCAL, null).as("read");
  // Other statements of partial requests

  try {
    IDataInterfaceResponse resp = req.execute();
    SystemStatusResponse statsResp = resp.getFirstNamedAs("status", SystemStatusResponse.class);
    VariablePageResponse pageResp = resp.getFirstNamedAs("read", VariablePageResponse.class);
    // Further operations on partial responses
  } catch (Exception e) {
    e.printStackTrace();
  }
  dataIntf.shutdown();
} else {
  System.out.println("Failed to authorize");
}
```

Figure 15 Example usage of the Data Interface

5.3 Bootstrapper

The middleware framework also embeds a service bootstrapper that allows to securely run services in environments that consist of services coming from many sources. The bootstrapper has been already described with more detail in the deliverable 5.2.

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

6 Implementation of the adapter modules

Adapter modules are an abstraction that allows external devices to be connected to the data exchange middleware. Since most of the behaviour of these modules are the same regardless of the communication technology of the external device, there is a big part of classes that are reusable. In ebalance-plus the adapter modules have been implemented in Java and use the Java middleware client provided as a .jar file to access the middleware API. Typically, an adapter module is composed of two adapters as depicted in Figure 11.

6.1 Adapter and interactions implementation

As explained in Section 4.2, adapters are data exchange-centred components represented by the IAdapter interface. In a typical adapter module two adapters are needed:

- **The middleware adapter which abstracts the access to the middleware:** this adapter is implemented once and reused by all adapter modules
- **One additional adapter for the external device:** this is a specific adapter that understands the communication protocol of the external device.

In the end, an adapter is just a Java class that implements the IAdapter interface presented in Section 4.3. Although one specific adapter is needed for each external device that will be used, generic adapters for each communication protocol have been implemented so that specific adapters can extend the generic adapter with minimal added code. Table 3 summarizes the generic adapters that have been implemented.

Table 3 Generic adapters

Adapter	Communication protocol	Comments
Modbus adapter	Modbus TCP/IP	It maps a middleware variable to a set of Modbus registers (each column of a variable is a Modbus register)
JDBC adapter	Generic relational database using JDBC	It maps a middleware variable to a database table (each column of a variable is a column of a table)
REST API adapter	HTTP-based REST API	Each REST API endpoint is mapped to a middleware variable
Websocket adapter	Websocket	Handles the websocket connection and queries information but the mapping is left to the corresponding implementation (classes extending this generic class)

In general, the concept of adapter best suits communication protocols with non-persistent connections. The module connects momentarily to the external device, obtains the information, and disconnects. This process is repeated to update the information of the different variables. This is the default behaviour that is used even for connection-oriented communication protocols such as websockets.

As for the interaction between adapters, one main type of interaction is used by almost all adapter which is the “transfer interaction”. The transfer interaction is configured by specifying a set of variables and a period. The transfer will automatically read the variables from the source adapter and write them in the destination adapter with the periodicity specified.

This project has received funding from the European Union’s Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

6.2 Module abstraction implementation

The adapter module abstraction is implemented by a base class that specific adapter modules have to extend.

```
public void run() throws Exception {
    logger.info(String.format("Adapter module: %s v%s", properties.get("module.name"),
        properties.get("module.version")));

    configure();
    startMiddleware();
    startAdapter();
    initTransfer();
    executeWaitBlock();
    stop();

    logger.error("Adapter disconnected");
}
```

Figure 16 Run method of the generic adapter module

The base class specifies the generic behaviour of the typical adapter module as illustrated in Figure 16, which shows the actual code of the run method. The following steps are executed when an adapter module is started:

1. **Configure the module**
2. **Start the middleware adapter**
3. **Start the specific adapter**
4. **Configure the transfer interaction**
5. **Keep the module running until a stop signal is received**

The adapter module is ultimately exported to a runnable .jar file that will be executed in the management unit where the corresponding middleware instance is running. The configuration of the adapter module is kept in a separate properties file so that changes in configuration can be made without having to regenerate the jar file. Also, each adapter module produces its own configurable log files.
By keeping each adapter module in a separate .jar, adapters can be easily started/stopped at runtime. Also, failures from one adapter module do not cascade to each other. Finally, by following this approach, each adapter module can be generated and later installed in different MUs by only changing the configuration file.

6.3 Demo site adapters and modules

In ebalance-plus the data exchange middleware, together with the adapters, will be tested in different demo sites which are part of work package WP6. Depending on the requirements of these demo sites different specific adapters will need to be implemented and integrated. Table 4 shows the preliminary list of adapter modules that will be implemented and deployed. It is important to note that this list can be modified in the future once the list of devices in each demonstrator is finalized.

Table 4 Demo site adapter modules

Demo site	Adapter type	External device
UMA	REST API	Energy meter
UMA	REST API	PV panels
UMA	REST API	Smart batteries
UMA	MODBUS TCP/IP	Power inverter
UMA	REST API	EV charging station

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

Demo site	Adapter type	External device
UMA	Websocket	HVAC system
UMA	REST API	Weather station
UNC	Database	Smart meter, nano grid, power inverter and PV plant
UNC	REST API	Smart batteries
JUNIA	REST API	Complete system
DTU	REST API	Complete system

7 Conclusions

Smart grids are complex system in which a high number of heterogeneous devices need to exchange information. To simplify the way these devices, communicate a data exchange middleware has been presented. The middleware provides a data-centric high-level programming abstraction based on simple operations that is used by all actors in the system to hide the complexity of the underlying architecture and hardware devices. To integrate external devices/systems within the middleware an adapter module is used. An adapter module provides an abstraction over each specific external device and oversees transforming requests/responses from/to the middleware/device.

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283

References

There are no sources in the current document.

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement N°864283