POLITECNICO DI TORINO

Master of Science Degree in Mechatronic Engineering

Master Degree Thesis

Supervisory Control and Data Acquisition for Distributed Smart Systems

Supervisors
Prof. Marcello Chiaberge
Prof. Srdjan Lukic

Candidate
Danny Crescimone
247050

Internship Tutor
FREEDM Systems Center
Dr. Srdjan Lukic

Academic year 2018-2019

Abstract

The today Energy system is characterized by the more and more growth and diffusion of the various distributed energy renewable sources in the generation and distribution level, such as wind and solar, are marked by the intermittent and variable generation. Moreover, now the customer can become a *prosumer*, it is no longer the last in the hierarchical order of the energy system, it can produce energy for itself and also feed the grid, so now need to be more integrated and in touch with the grid system. These aspects bring the necessity to evolve the actual power grid system that can't handle those changes. The emergence of the Smart Grids proposed as the electrical system of the future seems to be the technology that can provide the possibility to address those problems, creating an Internet of things application. However, this new power system actually presents the important lack of standardized communication and control protocols and a suitable platform to host it and simplify its diffusion. This thesis aims to provide a solution to those problems developing a Supervisory Control and Data Acquisition (SCADA) system integrating a Resilient Information Architecture Platform for Decentralized Smart Systems (RIAPS). In particular, that SCADA relies on Modbus and MQTT system interaction based on C++ and Python codes launched through the innovative middleware RIAPS. It has been examined the effects of the developed work directly in the target environment of a Smart Grid; everything has been tested in the Green Energy Hub testbed of the FREEDM Systems Center, that reproduce a possible Smart Grid environment made up of three microgrids. Therefore it was possible to use and test the real and innovative power electronic devices proper to a futuristic system including Low-Voltage Solid State Transformer. Moreover, it is necessary to underline the importance of this project, which represents the first real application of a power system running on RIAPS. The obtained results demonstrate the strengths of this novel Smart Grid platform and that this thesis constitutes a valid way to follow towards standardization.

Acknowledgements

During the development of this thesis work, I haven't ever been alone. It would not have been possible to carry out this project without the help and sustain of several people.
Foremost, I would like to express my gratitude to Polytechnic of Turin that during these five years of emphodi et amo gave me the possibility to grow not only as a student but also as a man. Thanks to North Carolina State University for giving me the opportunity of this thesis experience abroad. Thanks to FREEDM Systems Center and Dr. Iqbal Husain for allowing me to work on this project with all its cutting edge technologies. A special Thanks to my supervisors Dr. Lukic and Prof. Chiaberge for helping me with their wise advice during the development of my thesis. Thanks also to Dr. Mary Metelko e Dr. Peter Volgyesi of Vanderbilt University for their support and help with RIAPS.
I would also like to thank Dr. Salina Zabin and Dr.Mehnaz Khan for their tips and support for my project. An immense thanks to Karen Autry, without whom my stay in the USA would not have been the same. Thanks to all FREEDM Systems Center faculty and its Staff: Hulgize Kassa, Dr. David Lubkeman, Ken Dulaney, Ken Dulaney, Terri Kallal, Rebecca Mclennan.
I would like to take this opportunity to thank my friends of FREEDM Systems Center: Oscar Montes, Siavash Nakhaee, Ahmed Abdelsamad, Sayan Acharya, Erick Aponte, M A Awal, Kristen Booth, Rahul Chakraborty, Alireza Dayerizadeh, Thomas Dotson, Siddharth Mehta, Mohamed Zubair, Valliappan Muthukaruppan, Adam Stevens, and Hao Tu for always being there for me and for all the beautiful moments passed together. A special thanks also to the lovely friends met in the E.S King Village, Jessie Hoffstetter, Jens Vertongen, Julie Guiguitant, and Alice Michaelis.
Thank God for making me complete this American adventure very well. Thank all my family, my sisters, my parents and grandparents for sustaining me both economically and morally. An immense thank to my Girlfriend Maria Lucia for putting up with me during these tough months and for her

love. Lastly and most importantly, I would like to dedicate my thesis to my parents Sandro and Elisa for having enormous faith and belief in me at every point of my life and for having taught me to be what I am today.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

The today energy scenario is characterized by the influence and integration of new renewable energy resources such as wind, solar, water, and biomass. These green sources of energy are becoming more and more exploited by the power system of all the world. The reason for this change in energy production is due to multiple factors. The continuous population growth and the more consume of electrical energy combined with the limited petrol resource are the major reason for this migration to the renewable energy resource. Moreover, the more and more strict normative against the carbon dioxide emissions lead to a consequent run for innovative green solutions.
Another factor that influenced this migration is the global climate change, the increase of fossil fuel exploitation leads to a augment of the CO2 in the atmosphere and to a consequent increase of the global temperature. This brings to a melting of the polar cap that causes an increase of the sea level (from 1997 to 2017 the level has increased by 20 cm [1]). The increasing of carbon dioxide which continuously rises will lead to a further augment of the temperatures (other 2°C will cause catastrophic consequences to several ecosystems [2]. Unfortunately, the Intergovernmental Panel on Climate Change (IPCC) estimates that continuing to exploit the fossil fuels in this way will lead to an augment of the temperature of about 5°C in fifty years. All these reasons bring as to the conclusion that we need to change the actual energy system integrating more and more production of zero-carbon energy and apply a new system with the possibility to insert efficient strategy of energy management to reduce the waste and to properly host these new resources.

1.2 Current Power Grid

The actual power system with its three levels generation, transmission, and distribution should change a lot because actually, it works properly only for huge producers while the small prosumers are not able to interact directly with the energy market and can't be determined by energy costs. Concerning the generation level, it is actually improved by the integration of the Eco-Friendly source, instead, the other two levels have still remained too obsolete and not ready to host the new technologies. In today's scenario, the energy retrieved by those sources isn't smooth and constant but intermittent and variable. The actual power grid encounters many difficulties with such kind of power generation because it needs different algorithms and technologies to establish control and management of the system, so it needs to become "smart". In fact, this traditional energy systems result to be purely mechanical in which electronics devices, transmission line, meter, and transformers are very old and cannot be compatible with the new technologies for handle and control all the systems stakeholders.
Moreover, there are injections of green energy also in the distribution layer due for example to the diffusion of distributed renewable energy generation which needs a more flexible control of the distribution grid. Therefore also at this level, it is needed to bear the challenges of that new system, in which now the consumer became also a producer that not only have the energy to feed its own loads but also can sell the surplus of energy to the grid. In this way, it is no longer a mare user but also an important subject that can monitor and manage his energy consumption.
For these reasons now the grid systems should change to give the possibility to the new stakeholders to obtain several improvements and innovations, creating a system in which it is possible to integrate the new energy sources, with their difference from the traditional ones, and in which the user can play a more central role.

A Possible Solution

Those concepts bring to the inevitable move from the actual grid toward a new power system characterized by the "Energy Internet" in which everything is connected and available on the network: the Smart Grid. In the FREEDM

Systems Center, it is trying to give a solution to all the above-mentioned necessities, in particular, a project called Green Energy Hub (GEH) concerning microgrid systems, represent a futuristic home grid in which is present an innovative power electronic device: the Solid-State Transformers (SST). However, the GEH presents some issues:

- The communication between the nodes of the application and the control machines has several problems and limitations. It present a leak in the Supervisory Control and data Acquisition (SCADA) . The control machines should be able to control and monitor the GEH system by sending command and receiving data from the connected devices. Instead actually it is possible only to receive data even in a not stable and proper way.

- The complete absence of a suitable software platform which can host that system and make the life easier for the developer to implement algorithms, to manage the target devices connected into the network, to simplify the plug and play capability, and many other features.

- The implementation of a better control strategy to switch between the two microgrids mode (islanded and grid-tied)

These limitations make the simulated Smart Grid a fragile system not ready yet to be applied to our homes and in which the developer find several obstacles and difficulty to add new improvement and update.

1.3 Motivation

the previously explained emissions constraints and the consequently green energy introduction are forcing this migration to a new power grid system. This is one of the reason for the choice of the Smart Grid as the principal topic of this thesis work relay on the fact that it is becoming an important concept to develop and improve to reach the standardization and the final spread in all the world.
As it is seen, the actual GEH testbed represents a system with some imperfection and leak in which the developers found some difficulties to upgrade it. Thanks to its characteristic it could constitute an optimum starting point for future Smart Grid application. So trying to improve and reduce these

imperfections can help the FREEDM Systems Center and so the power systems community to speed up the Smart Grid injection and diffusion on the global market.

1.4 Objectives

The objectives that this thesis work carries out are above all focused on the upgrade and resolution of the first two issues of the GEH previously presented:

- To implant the GEH testbed in the RIAPS platform (a new Smart System middleware), in which the microgrid system obtain all the advantage of a suitable resilient architecture platform for Smart Grid.
- To establish a full SCADA system for the GEH inside the new Smart Grid platform, in which will be possible to send command to the devices and receive data from them
- To provide a hybrid communications paradigms made up of Modbus and MQTT, making the GEH a perfect IoT application fully connected on the Internet.

1.5 Thesis Outline

A brief outline of the following chapters are presented next. Chapter 2: Smart Grid State-of-the-Art, starting with an explanation of the reason of the needs for a new grid system, going to the history and definition of the Smart grid system concluding with architectural analysis and relatives today issues.
Chapter 3: FREEDM Systems Center, a fundamental introduction of the work environment in which this project has been developed, the Solid State Transformer and the GEH testbed are presented here.
Chapter 4: RIAPS, talk about the revolutionary middleware used for the smart system, in these pages it is analyzed in detail.
Chapter 5: Methods, all the methods and tools used in this project are analyzed here, to explain better how the work was carried out and how it was possible to obtain the wanted results. Starting with a list and description of the fundamental components of this thesis work, useful to provide an overview of the methods chosen. Then an explanation of some RIAPS used

commands is present. Concluding with the examination of the developed codes.
Chapter 6: Result and Conclusion, in which all the results and the outcomes of the project are showed and analyzed. Concluding the thesis with a discussion of what has been achieved and an analysis of the possible future works.

Chapter 2

Smart Grid State-of-the-Art

Global warming, limited availability of fossil fuels, growing demand for energy, and augmenting of energy price are all factors that have led to an exponential increase of renewable energy exploitation. However, if on the one hand, this new trend helps the fighting against the aspects mentioned above; on the other hand, it greatly affects the increase in complexity of the already sophisticated power system. This together with the necessity of integrating the power system with the new smart devices creating more straightforward and intelligent management of the resources, raise the need for a new standardized system which provides the right instruments to handle these new actors of the power system. A novel and revised power system, known as "Smart Grid," has been introduced to reach these new goals.
Taking the core role of this research, in this chapter the Smart Grid is analyzed in detail, starting with a brief introduction of its history, passing through a review of its main characteristics and the new features implemented in the power system, ending with an analysis of the problems and challenges to be overcome.

2.1 The Need for a New Power System

In now day in which everything is reachable on the web, merely reachable by anyone, and with the more and more attention placed on our and earth health, born the necessity to change and upgrade everything that surrounds us. And what surrounds us more than electric energy?

This can, therefore, be considered the mainspring that brought us to try to update the actual grid system, which results too complicated and too clumsy to deploy on the internet and to plug and play these renewable energy resources. The main reason that makes it impossible, lay on the behavior of those new energy resources, which could be split into two main groups:

- Those resources that have characteristics very close to the traditional energy source and so are easy to predict and control. This group comprises hydroelectric and biomass energy.
- The energy resources characterized by variable capacity factor and discontinuous power generation, so the control and management of them result very complexly. Examples of this group are Wind and Solar energy.

This letter branch of renewable energy resources causes a big problem for the actual grid system. The system should be able to exploit those resource when available (when the wind moves the eolic turbine, or the sunlight penetrates the PV panel) to reduce the energy price and CO2 emissions. However, a side effect, in this case, is the hazardous balance of an unaware and fixed demand with a variable and unpredictable supply. The solution to this problem is the use of control algorithms that makes the system a smart system able to overcome situations like this.
However, the traditional energy systems result to be, profoundly mechanical system in which control center, transmission and distribution lines, made up of electronics devices, breakers, and transformers that were built in most case more than one hundred years ago. Therefore integrate the new technologies to monitor, manage, and control all the power system nodes, in other word create an "Energy Internet" it is almost impossible whit this old system.

2.2 Smart Grid History and Definitions

The aim of control the power demand of users is not really new. The idea of making different strategies for handling loads instead of supply was born in the seventies, the period of the significant energy growing demand characterized by many blackouts and energy crisis [3]. In particular, the principal objectives were to reduce and modify their energy consumption when necessary, something very similar to the actual goal of integrating renewable

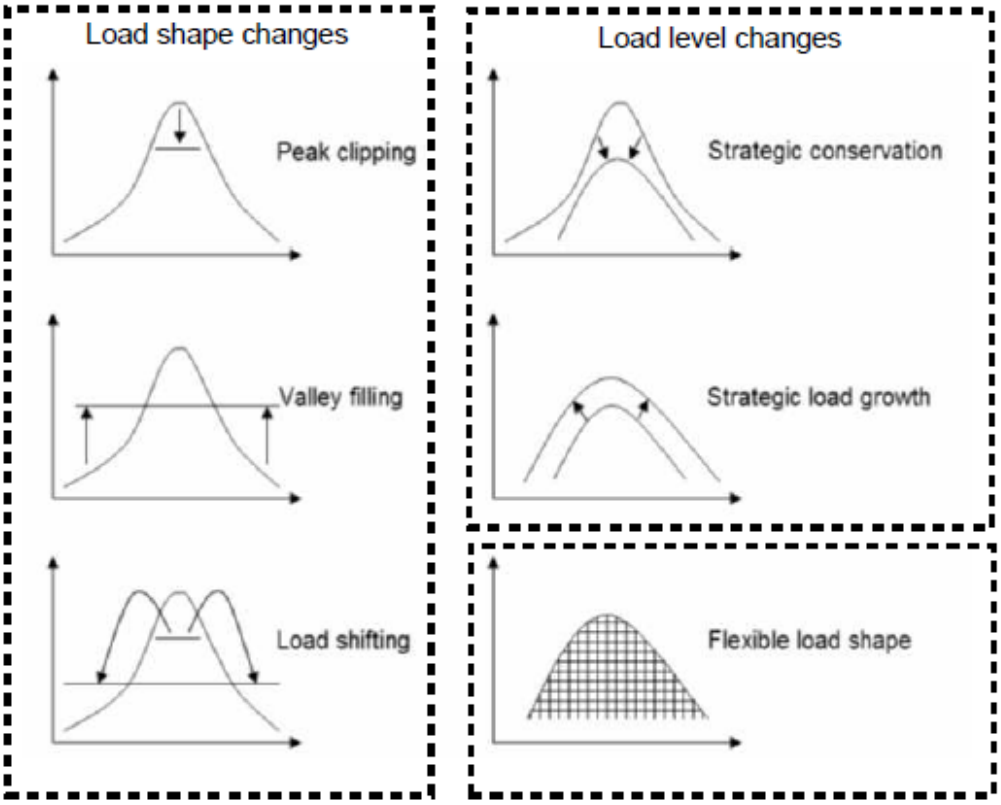


Figure 2.1: Load control strategies (source: [4])

energy resources.
An overall picture of what were the different load control tactics can be explored in [4], where these strategies are split into the modification of load shape and modification of load level as presented in Figure 2.1. And is precisely this ancient control vision which includes adjustable load shape that later resulted in the Smart Grid idea.

The modern idea of Smart Grid and so "Energy Internet" is very similar to the concepts of the internet for the computer industry of the eighties and nineties. In this sector, the development of internet caused a significant change from the dependence on centralized computers to frameworks based on distributed clients connected on the widespread internet.
In a similar criterion, the Smart Grid system shift from a centralized energy

generation and distribution system in which the user represents a mare endpoint; to a decentralized system characterized of many variegate microgrids (explained better in 2.3.3) in which now the customers became also a producer.

2.3 System Structure and Characteristics

The Smart Grid presented as the new technology which has the critical duty of moving the actual power system from a centralized and passive structure to a dynamic, interactive and distributed one (centered on the multiple customers).

Following the generic Smart Grid architectural model provided by the SGAM (Smart Grid Reference Architecture), the studied system is constituted of different layers (see Fig 2.2), each one with a fundamental and specific target; some of these are outside the range of this thesis, so they are not investigated but to know more they are noted with reference below:

- Component Layer made up of all the physical devices needed for sense and measure the variables of the system. Moreover a component of the new Grid should also be able to establish an internet connection, to communicate these data to all the network; moreover, they should allow a fast and straightforward system upgrade and rescale.

- Communication Layer, the central scope of this project. The development of adaptive communication strategies to maximize the efficiency of the Smart Grid in which each device connected in this new energy internet can interact one each other through different protocols (explained better in section 2.3.1).

- Information Layer, fundamental level of the Smart Grid in which all the data coming from the measurement components thanks to the communication layer are first collected and sorted, after applying data modeling, data integration and data analytic methods, they are then delivered to the control station.[6]

- Function Layer, also known as Control Layer, in which all the data collected in the information layer are used as input variables for the

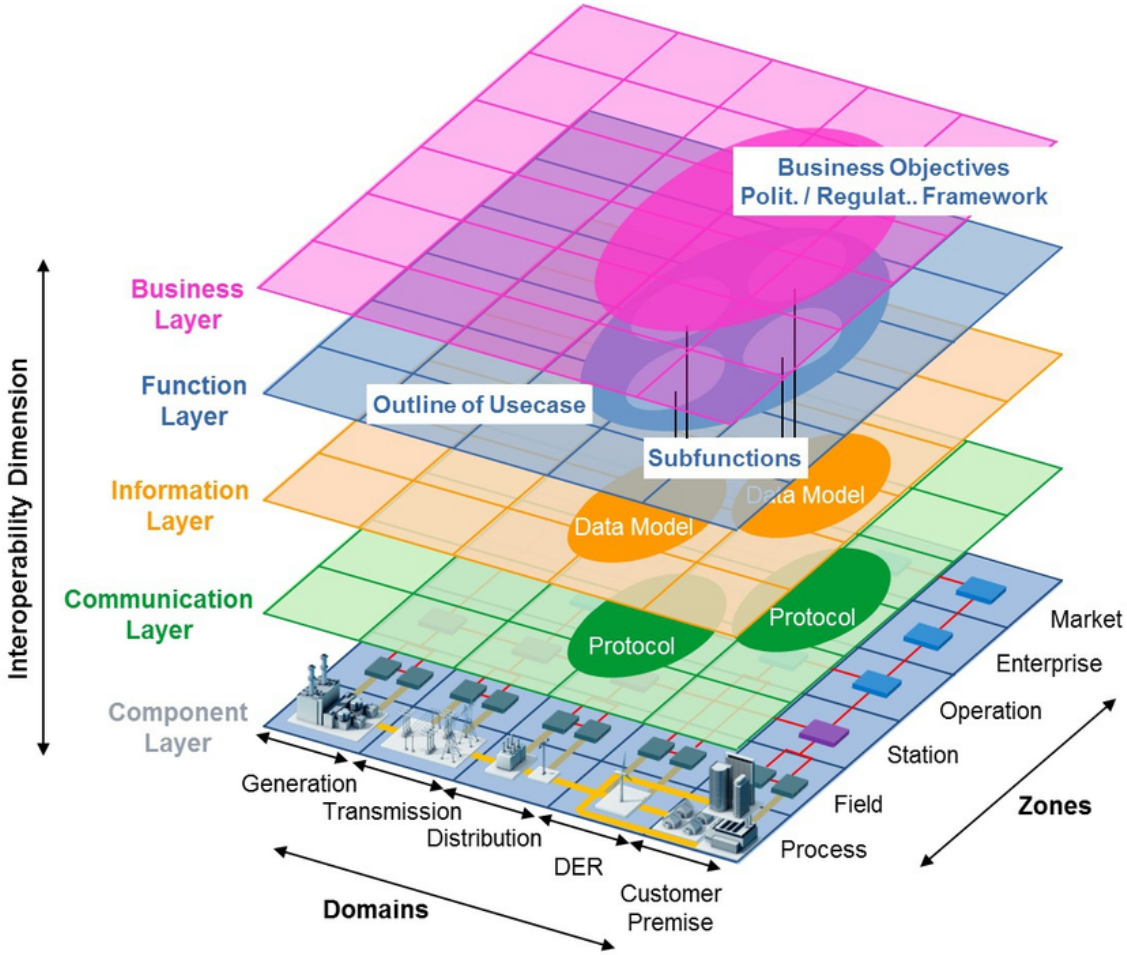


Figure 2.2: Smart Grid Layered Architecture [5])

various control algorithms. As a Smart application, the Smart Grid should deploy the "intelligence" (control algorithm) through all its nodes (see the following pages)

- Business Layer, as one of the benefits of the Smart Grid the power/load manipulation, it represents business models and regulatory requirements, and provides a more efficient system and the possibility to manages the different energy resource choosing the most economical and efficient exploitation of them, thanks to the control algorithms [7].

Another essential characteristic that marks this studied new system is reliability. As a critical component of our life, the power system should be safe and stable with various situations, like a sudden increase in the demand peak or outage, for these reasons the reliability is an essential skill that the Smart Grid owns (see [8] for more detail).
Moreover, an additional characteristic of the Smart Grid is the "hidden security layer." Because of one of the strongholds of the Smart Grid is the use of Internet in which each component connected in the systems, the cybersecurity became a priority in this new idea of power system. The various information detected by all the instrument should be private and accessible only to the authorized personnel; the same should be done for the command sent by the user (explained in detail in [9]).

2.3.1 Communication Layer, an IoT Example

To guarantee the communication layer functionality, Smart Grid distinguish three hierarchical sub-networks, as shown in Fig. 2.3. The coverage range of the network and the data rate,i.e., number of bits exchanged per second, define the Smart Grid distribution system communication area, so it is possible to distinguish [10]:

- Industrial Area Network (IAN), Home Area Network (HAN) or Building Area Network (BAN), restricted to a single home, and so including only the load energy resource and storage device of a single habitation, in fact, they don't need a particular massive communication link (100 Kbps at maximum).
- Neighborhood Area Network (NAN) or Field Area Network (FAN), a branch of houses as the HAN is related only to the power generation and consumption.
- Metropolitan Area Network (MAN) or Wide Area Network (WAN) correlated to the power transmission and distribution level; The Smart City could be an example. For this kind of application an very high bit rate is needed.

The communication layer represents all the protocol and mechanisms utilized to share information and commands between all the components connected to the WAN, and also communication between WAN and function

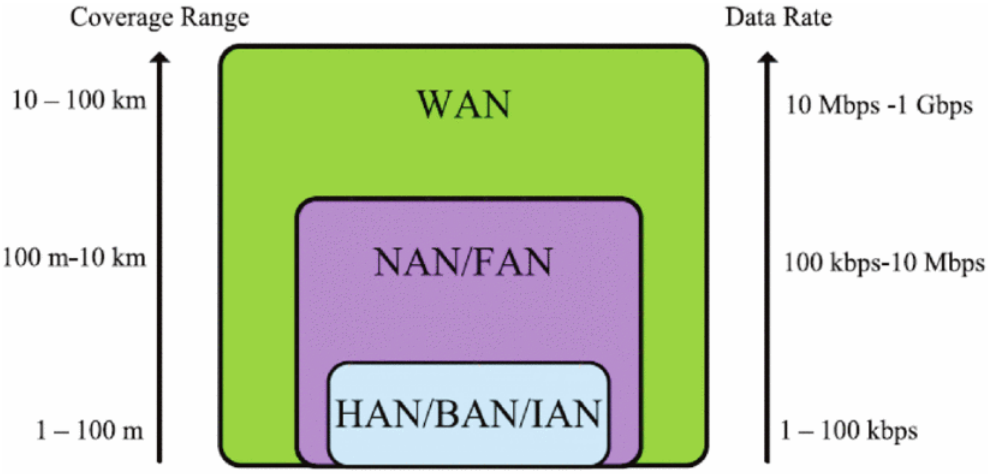


Figure 2.3: Smart Grid hierarchical sub-networks [11])

layer (power management station).
After presenting the various components of a Smart Grid application underlying also their role and functionality in the system, so now is essential to analyze how they can interact and communicate. In particular, Smart Grid communication can be wired, for example, power line communication (PLC) and telephone line, or wireless like GPS or WiFi.

For each application and situation, it is essential to choose the right communication type and protocol based on the distance and data rate requirement of each specific project. In the Table 2.1 all the communication technologies used in the Smart Grid application are present. To better analyze in detail the different protocols and standard of that table items, see [12], [13]. From that Table, it is possible to see that communications in optical fiber, 2G (GPRS), 3G, and 4G are most comfortable for a wide range and high bit rate, instead, applications like PLC, Ethernet or ZigBee are mostly used for more restricted distances.
In the nowadays, the most common communication approach is the use of PLC or Ethernet for interaction between the smart meter and data concentrator, and GPRS or satellite for the interaction between data concentrator and DCS [14].

Communication Technologies	**Subnetworks**		
	HAN/BAN/IAN	**NAN/FAN**	**WAN**
Wired			
Optic Fiber	-	-	X
DSL	-	X	-
PLC	X	X	-
Ethernet	-	X	-
Wireless			
Z-Wave	X	-	
Bluetooth	X	-	-
ZigBee	X	X	-
Wireless Mesh	X	X	-
WiMAX	-	X	X
GPRS	-	X	X
4G	-	X	X
GPS	-	-	X

Table 2.1: Communication Technologies for Smart Grid (source: [12])

Power Internet of Things

The presence of system nodes all connected to the internet to share information make the Smart Grid a real example of the Internet of Think application. The idea of energy internet generates a revolution on the field of power system, in which now every information is running on the internet thanks to smart and embedded devices like the smart meter (explained later) with is embedded gateway can obtain an IP address becoming internet object. The advantages of this new system are several, both for consumer and producer,

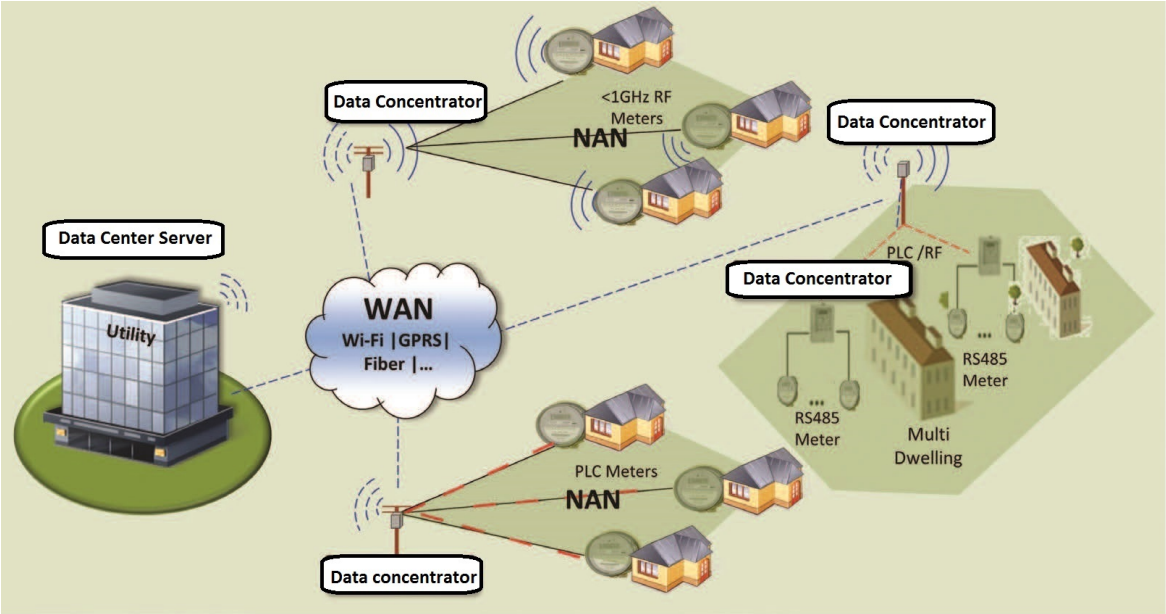


Figure 2.4: Smart Grid component overview

the first now with the easy access to information can know his power consumption and manage his smart home energy; instead, the control station now can receive/send information and command in a faster and safer way.

2.3.2 Component Layer

The component layer lies at the base of all mentioned layers. It consists of all those devices that make this new grid system a smart application. As it is shown in Figure 2.4, the architectural structure of a Smart Grid is made up of critical instruments such as smart meter, data concentrator, gateway, and data-center server [14].

Smart Meter

The smart meter has to deal with multiple energy vectors and plays a central role in this new environment. In particular, it provides several fundamental services to the prosumer such as integration of variegate components and devices, collecting recording and storing data, easy displaying and monitoring

of the information for the user and load control function [15].
The Smart meter is able to manage different data sources and provide real-time data processing by using a multi-services approach, information coming from water, electric, heating, and gas meters to allow ample purpose services.

Gateway

The scattered smart meters once retrieved and collected the data they share those through the uses the gateways. A gateway is networking hardware, i.e., a device which is embedded in the smart meter to allow network communication. Like a router, it is able to route all the incoming and outcoming data flow of smart meter, because it not only receive data from meter but also from the other data concentrators, and these could be data coming from other smart meters (for microgrid control) or command sent by the data center server.

Data Concentrator

The data concentrator is another crucial element of the component layer of the Smart Grid system. It has the job to collect information coming from the many smart meters and the data-center and also it has the critical tasks to analyze those data. Examining the information coming from the smart meter, the data collector could immediately offer a first defense line for the outage and overvoltage. Moreover, it provides routing processes and data encryption, ensuring privacy and safety [16].

Data Center Server

Data center server (DCS), also know as demand-response server, it can be considered as a bridge between smart meters and supplies. The DCS receives all the information of the WAN and interacts with consumer and producers communicating them all the changes of the interested network.
The principal task of the DCS is to allow efficient distribution of energy and provide immediate reaction to the time-varying consumer demand of power, optimizing the algorithm for power distribution.
The data-center server moreover makes accurate user demand forecast, that allows it to foretell the times of peak power consumption. Furthermore, it

has the role to manage and control the whole WAN in which it is applied and provides also restoring tools in case of fault on the network [17].

2.3.3 Function Layer, Microgrid Control Strategy

Another benefit provided by the Smart Grid system seen at HAN or NAN level is the possibility to create the so-called microgrids systems. The microgrid is a localized power system group that could operate in both grid-tied mode, i.e., attached to the WAN grid, and In the islanded mode detached from the WAN and producing in complete autonomy the energy for itself thanks to the distributed renewable energy resource integrated into the HAN or NAN system. A real example of a microgrid is described in chapter 3.1.
The Smart Grid function layer not only plays the fundamental role of monitoring and managing the power flow but also allows the switch between the Grid-tied and islanded mode (or vice-versa) it is necessary to use one of the most representative items of the control layer, the grid synchronization control strategy.
The control procedures ranging from centralized to entirely distributed and decentralized with a hierarchical control structure (see Fig. 2.5 TU7). The distributed energy resources (DERs) like wind turbines and photovoltaic panel, and the distributive energy storage devices (DESDs) produce DC power, by using inverters they can be linked with the AC power grid and form a DC or AC microgrid. This inverter ensures not only DC/AC conversion but also voltage balance, grid synchronization, and system load sharing. Another essential component of this distributed approach is the use of a transceiver that like an embedded device make the DRER able to interact with all the network. However, the cores of the hierarchical distributive strategy are the three control level, *Primary Control*, *Secondary Control*, and *Tertiary Control*.

Primary Control

Because of an in-depth analysis of the *Primary Control* is beyond the goal of this thesis, a short overview is presented. This first control strategy also knew as *Droop Control* has the objective to stabilize the power systems and to obtain an equal load sharing between the various DERs and DESDs. In

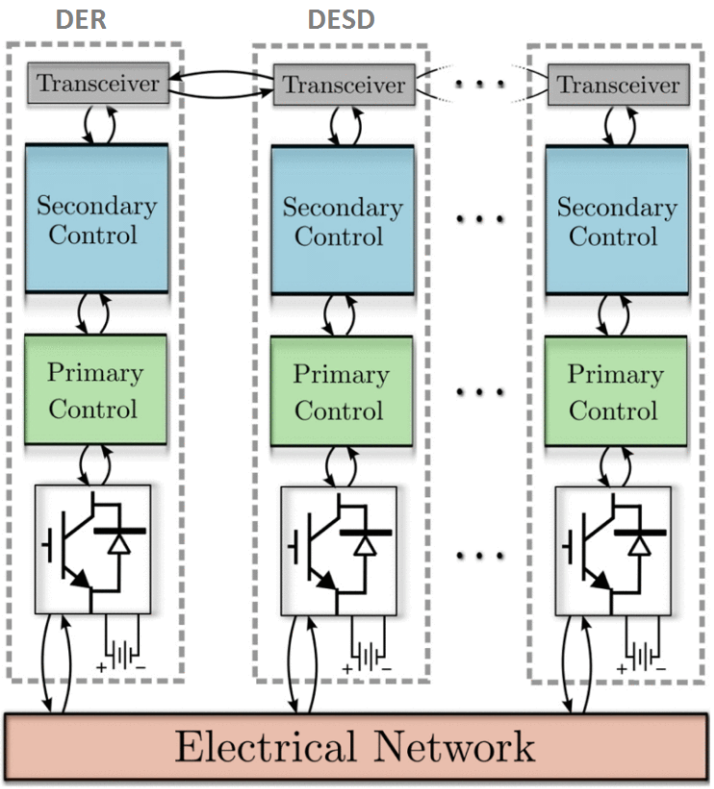


Figure 2.5: Smart Grid control layer

islanded-mode, inverters are managed as voltage source inverters, controlling voltage magnitude and frequency of the microgrid [18].

$$E_i = E^* - n_i \cdot Q_i, \tag{2.1}$$

$$\omega_i = \omega^* - m_i \cdot P_i, \tag{2.2}$$

Where E_i and ω_i are respectively the voltage magnitude and frequency of the i-th microgrid; instead E^* and ω^* are the voltage and frequency of the grid network; Q_i and P_i are the reactive and active power of the i-th microgrid and the n and i are the *Primary Control* coefficient. For more detail about the *Primary Control* and about the 2.1,2.2 see [18],[19].

Secondary and *Tertiary Control*

The *Primary Control*, while is effective for stabilization, it makes the steady-state voltage and frequency deviation from their nominal values [20],[18]. So the equation 2.1 e 2.2 must be integrated with the *Secondary Control* to resolve the aforementioned voltage and frequency deviation.
In the literature, there are several *Secondary Control* approach varying from centralizing and decentralize algorithm, each one with its own pros and cons. For example, one centralized strategy is to apply the *Secondary Control* directly in the bulk energy system using only one centralized integral controller and "one-to-all communication" [18], [21]. The side effect of this concept is that it contrasts with the Smart Grid goal of autonomous handling of all the various microgrid.
A decentralized strategy could be instead to utilize "slower integral controller" on each inverter [19]. However, the different timescales among primary and *Secondary Control* remove the power-sharing capability provided by *Droop Control*.Another distributed *secondary Control* strategy proposed is characterized by the continuous communication between all the DERs and DESDs that share frequency voltage and reactive power. However, the drawback is that each inverter must interact with all the other inverter, creating a dense and busy communication structure. A resolution to these problems is needed to for a smoother diffusion of the Smart Grid concept; this is one of the goals of this project, in fact, a solution can be seen in chapter 4, in which a novel Smart Grid Platform is analyzed.
For what concern the *Tertiary Control*, it is strictly related to the business layer, in fact, it deals with the network energy dispatch considering the actual energy prices and market (for more detail see [22]).

2.4 Issues

The introduction of the analyzed Smart Grid in the actual power grid system is a big challenge because it needs an essential factor that till now is missing: the standardization. The presence of so many different control strategy and communication paradigms, together with the lack of a uniform platform in which the Smart Grid system could rely on are all obstacle for the creation and diffusion of a standard that each microgrid and WAN can follow to obtain a smooth diffusion of this revolutionary system.
The widely used control strategy is based on centralized control, in which each information coming from all the nodes of the systems are collected in a command station than analyzed; finally, answers containing the control variables are sent back. It is easy to understand that this approach has a critical problem characterized by the fact that if a fault happens in the control station, everything will be blocked.
Instead, the newer distributed control algorithm, which is something closer to the Smart Grid concept of autonomous handling of the actual power system devices, has the already mentioned problems related to the distribution of the algorithms, and the time difference between the various control algorithms.
All these Issues underline the necessity of utilizing a more comfortable network platform for the Smart Grid concept that provides the possibility to deploy decentralized distributed control algorithms, simple and fixed communication paradigms between the connected nodes, reliability, fault tolerance, and scalability.

Chapter 3

FREEDM Systems Center

This thesis project has been carried out in the department of the North Carolina State University specialized in Power Electronics and Power System, whose name is FREEDM Systems Center. Therefore it is indispensable to introduce and describe this cutting edge pole also defining the structure and components of which it is made. Above all, it will be explained in detail the Solid State Transformer, whose utilize has been fundamental for this project. In particular, after an in-depth introduction of the FREEDM Systems Center and an accurate analysis of the Solid State Transformer based on state of the art, there will also be a description of the Green Energy Hub testbed. It is precisely this latter the core of the hardware part of the project in which all the experiments took place, thanks to the use of the Low-Voltage Solid State Transformer together with the High-Voltage one make the FREEDM Systems Center one of the most innovative centers of the power systems.

3.1 FREEDM System a revolutionary Smart Grid concept

According to the trends of the energy consumption of the entire world that is growing year after year, only in 2013, the International Energy Agency (IEA) has estimated that the word energy consumption was about 158 TWh. And the majority of this energy consumption comes from non-renewal and so non-environmentally friendly energy resource, such as petroleum, coal and natural gas. This actual situation could be hazardous and unstable, because of the non-renewable nature of these sources and the fact that the majority of these products comes from politically volatile states; also considering the

increase of global warming produced by the emission of carbon dioxide.
One solution to reduce these risks is to increase the exploitation of renewable energy. Two obstacles thwart the integration of this Eco-friendly energy sources one the actual grid system, the necessity to store that energy and the need for stable control action. Therefore to allows a smoother and more stable spread of the renewal energy, trying to provide at least the 50% of consumed energy, the future grid must overcome these storage and complex control problems.
There exist two different fields of application of the renewal system, both important to reach those goals: large-scale centralized installations, like solar or wind farms; and wide-scale Distributed Renewals Energy Resources (DRERs), i.e., the use of DRER at the industrial or residential level, shifting the power production from centralizing to the distributed concept [23].
Therefore one visionary idea which matches these needs is FREEDM or the Future renewable electric energy delivery and management power distribution [24]. It proposes a system for residential purpose made of several microgrids, allowing the users to control their energy needs with modular and reduced DRERs (photo-voltaic panel or wind turbine) by using a uniformed plug and play interface, helpful also for creating an easily scalable application.
Moreover, to obtain a utilization as efficient as possible, Distributed Energy Storage Devices (DESDs) to sustain the DRERs that is not able to provide a stable and constant power supply that depends on the environmental factors. So the customer could also sell the surplus of energy produced by is microgrid pushing it in the primary grid thanks to the *Intelligent Energy Management* software present in the system by checking the price information and the DRERs and DESDs state and availability.
An example of the ideal new FREEDM System Smart Grid idea can be seen in Fig. 3.1 in which are depicted the grid interfaces of a futuristic home or other distribution level loads (microgrids).As we can see, the crucial role of the envisioned FREEDM System is taken by the Solid State Transformer (SST), which act as an internet router creating a bi-directional link between the microgrid and the main grid (explained in detail in the section 3.2).

3.1.1 An *Energy Internet* Architecture

As a perfect Smart Grid, the FREEDM System proposes to be an *Energy Internet* architecture; in fact, it provides three essential skills proper of the

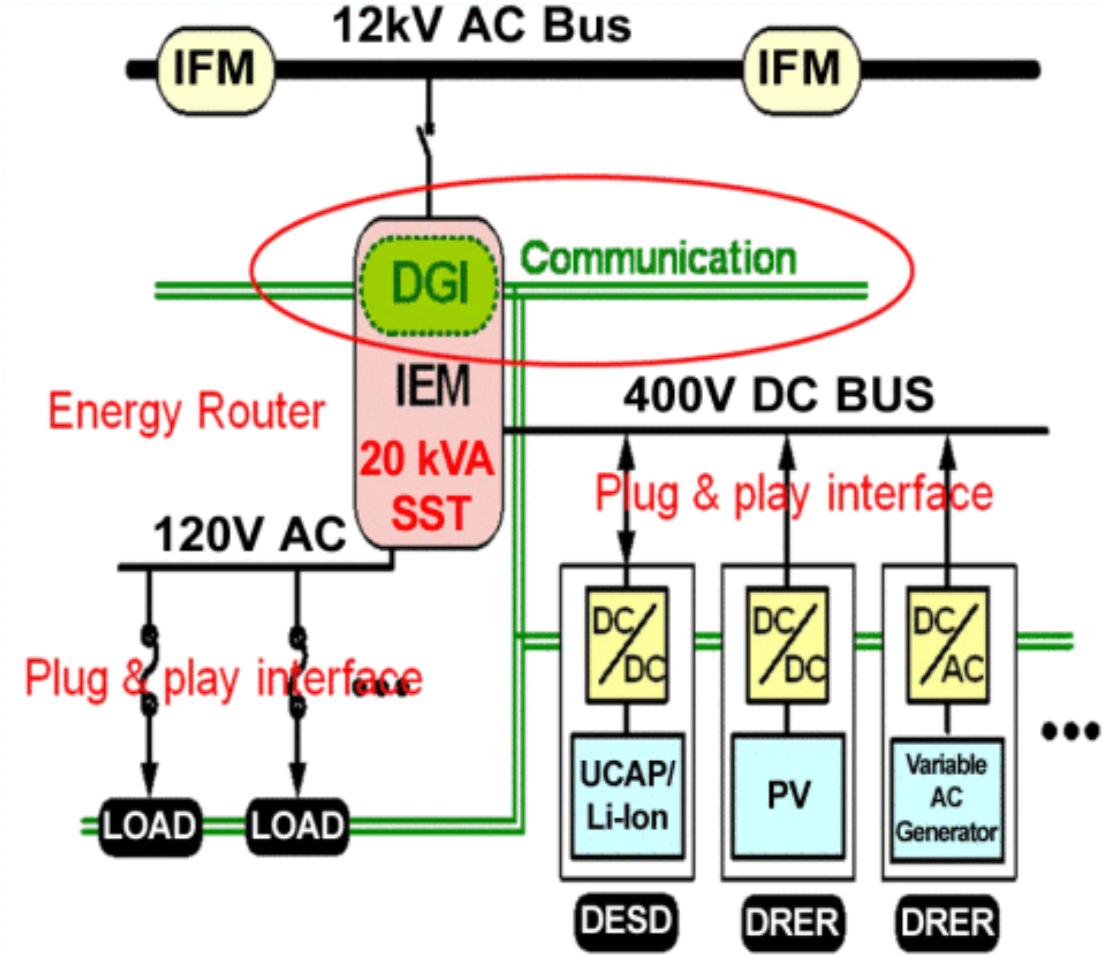


Figure 3.1: One node of FREEDM System model [23]

information internet:

- Plug and play characteristic, like a computer that could reach the internet via Ethernet cable the FREEDM System exploits both the DC and AC bus. This characteristic also includes a communication interface, so each device connected to the system is recognized and can communicate and describe the storage device, loads and generating source. Therefore the realization of this skill allows as to exploit the system as a "USB port of our computer" [23], that automatically detect a new USB device, and linking it with the appropriate drivers. In the analyzed system, the driver is represented by the power electronics devices (such as converter)

which provide stability, adaptation, and control of the power flow and quality of the various device plugged to the system.

- Router capability, like the internet router route information to the right place the FREEDM *Intelligent Energy Management* (IEM), is an energy router that connects the various device plugged in the microgrids to the distribution grid (explained better in the subsection 3.1.3). all this is possible thanks to the Solid State Transform as it is showed in Fig 3.1. Once plugged in the FREEDM System the device is managed by the IEM that link it to the main grid, monitored provided of control reference after collected its data. In particular, a control reference could be an on/off command or an adjustable variable command like power or voltage.

- An open-standard operating system, like in information internet characterized by the TCP and HTML to exchange info through the web, the FREEDM System has the *Distributed Grid Intelligence* (DGI) embedded into the IEM device (in this case into an SST) which exploit the communication network to handle and interact with the single internal part of a microgrid and with the other external microgrid system.
  Beside the IEM device, there is also the *Intelligent Fault Management* (see subsection 3.1.3) very useful to avoid the propagation of the fault in the 12kV AC bus as shown in Fig. 3.2. Moreover, it provides also reconfiguration capability and seamless power flow.

Following the FREEDM System concept, the new power system also includes the possibility to exploit both the DESDs and a common level energy storage devices (ESDs). In fact, as we can see in Fig 3.2, the system is also connected to the 69kV sub-transmission circuit through an IEM. And even if needed for example in some industrial application could also be possible to have a 480V three-phase current thank to another IEM.
The proposed system once successfully implemented and integrated into the actual grid could be considered as a standard laptop, in which a communication bus handles each power management device. More or less in the same way but of course with several orders of magnitude of difference that a laptop can be smoothly connected and disconnected to a power source, the FREEDM System could be attached and detached from the primary grid if it there are faults or if it is unavailable. Moreover, in this revolutionary system like in a laptop, the user can choose among to the different mode of power

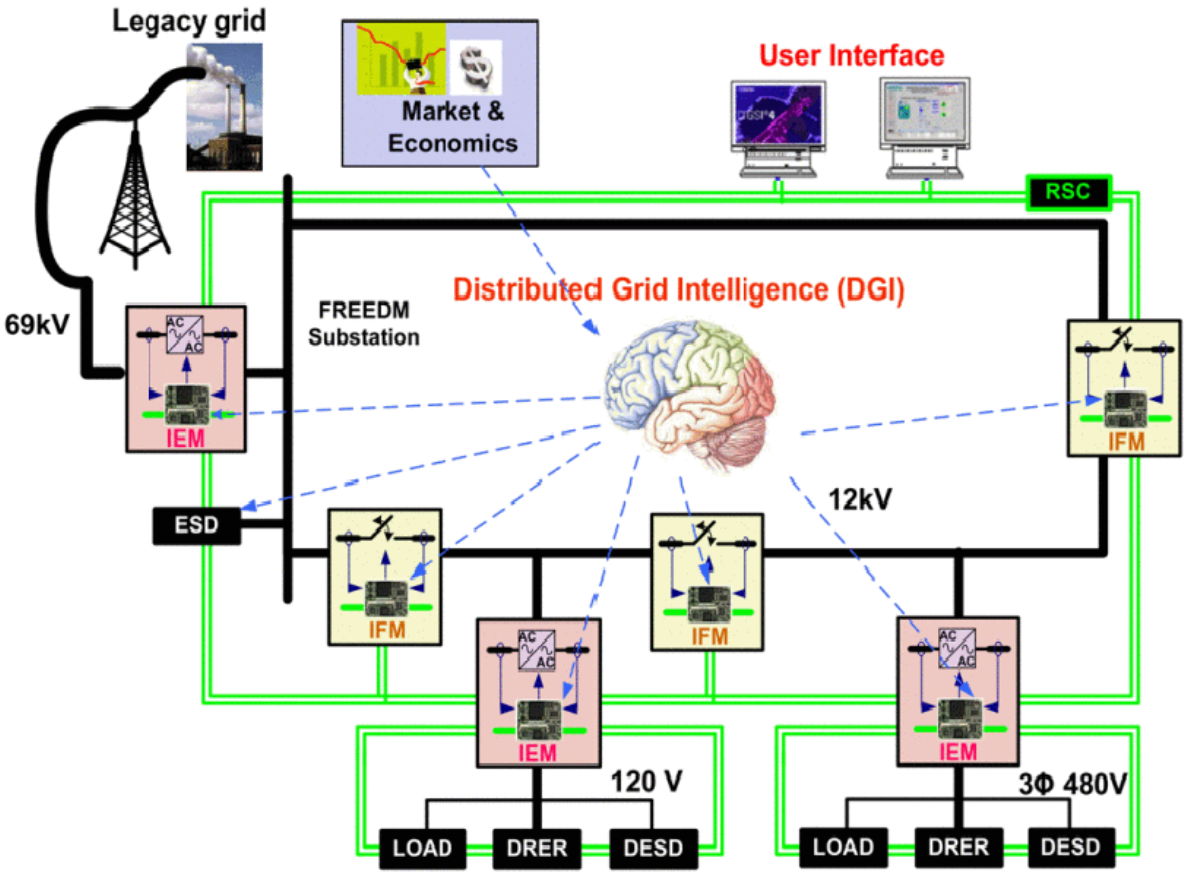


Figure 3.2: Key elements of the FREEDM System [23]

consumption (high performance, balanced or energy saving).

3.1.2 *Fault Isolation Device*

Because of the FREEDM System acts as an *Energy Internet*, a precise and fast protection a reconfiguration is required. To satisfy this requirement, in the System is utilized another new device, the Solid State *Fault Isolation Device* (SS-FID) that provides super-fast fault isolation. This device is provided with *Intelligent Fault Management* (IFM) functions, becoming an IFM node able to communicate with the other IFM and IEM nodes. But the most important IFM function remain the fault detection, isolation and then

reconnection as possible.
In the standard AC power system, the more common *Fault Isolation Devices* are the mechanical circuit breaker that isolates the fault in around 10ms [23]. However in the FREEDM System short power interruption caused by a fault like a short circuit, for example, could generate dangerous disturbance at high load. So In this revolutionary system the use of both one of the many skills of the SST, the continuous load control, and the IFM that provide fast isolation, the safety and fault rejection is not a problem.
Therefore the FID is developed with semiconductor switches that replace the inadequate slow mechanical circuit breakers. Also if this change adds some losses and due to the silicon device, an augment of the cost, and a more accurate galvanic isolation the semiconductors are preferred because of they are able to provide a super fast turn-off, around 50 µs [23]

3.1.3 FREEDM System Control

One of the side effects of this *Energy Internet* system is that the control algorithm becomes much more complicated than the traditional microgrids-based system seen in 2.3.3. Figure 3.3 demonstrate it, in fact, as it is possible to see the FREEDM Systems control is made up of several layers. Level 1 is the standard user hardware level in which the primary goal is to achieve a smooth plug and play of the various device (DRER, DESD, and loads). In Level 2 the control is focused in the SST which the objective to manage the LV-AC and LV-DC bus and allows active and reactive power or frequency control for the ports of the grid side.
Moreover, at this control level, the power quality and the ac side harmonics should meet the requirement imposed by the system. In Level 3, the control takes care of some very fundamental topics such as energy dispatching, grid protection, islanding from the main grid and re-tieing in the main grid. In Level 4 instead, take place the multiple FREEDM Systems control that deals with the possible interaction among other systems.
All these controls are integrated into the System through the DGI embedded in each IFM and IEM nodes. So these control could also be classified into two branches: *Intelligent Energy Management* and *Intelligent Fault Management* nodes.

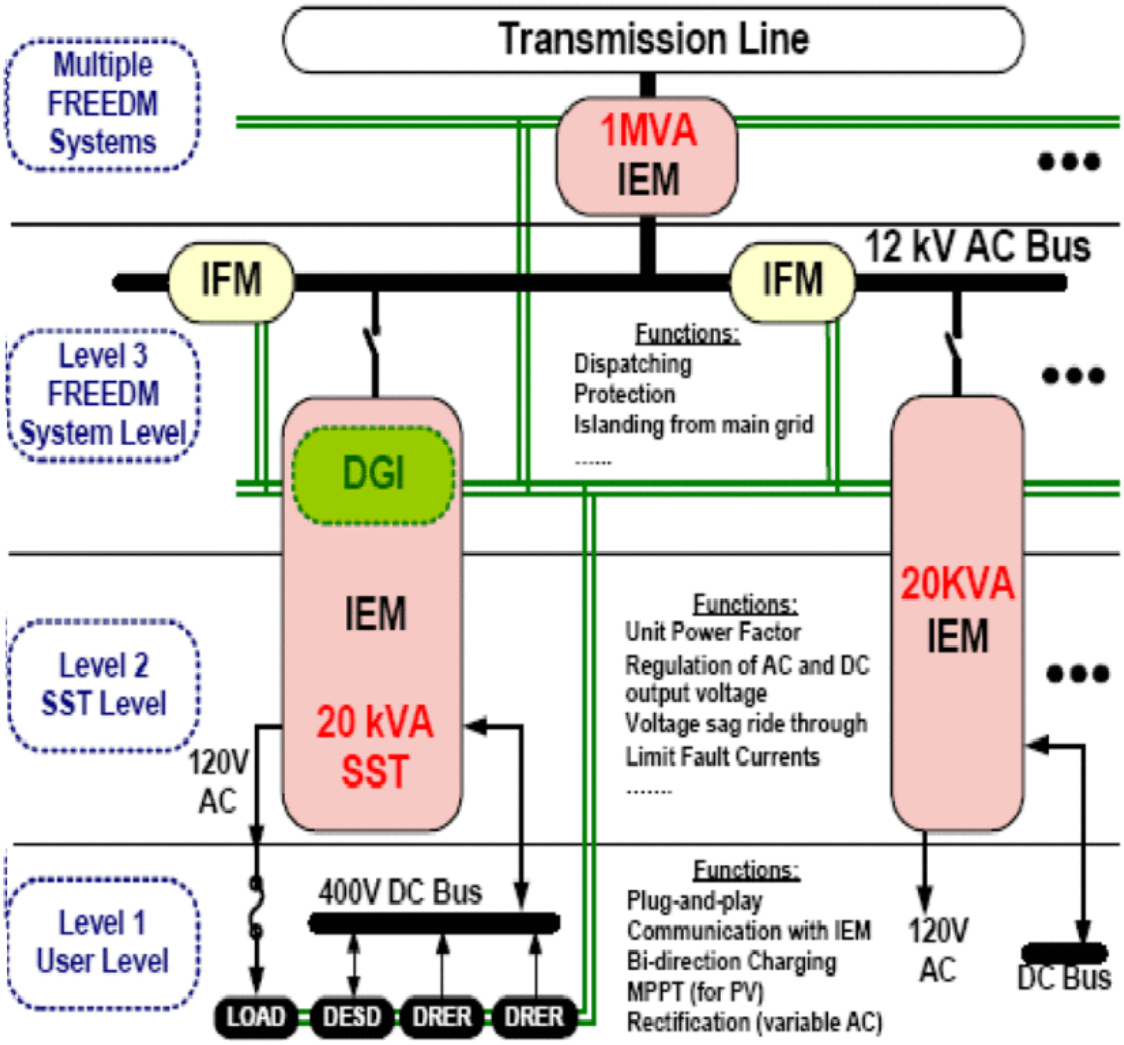


Figure 3.3: FREEDM System Control levels

Intelligent Energy Management

The *Intelligent Energy Management* has the critical job to detect and identify the various hardware devices plugged to the System. The FREEDM System provides bi-directional interaction and power flow with the uninterrupted operation when the devices are plugged or unplugged.
The IEM must deal with two types of ports, active and passive. An active ports host devices which are able to communicate its status so the IEM could quickly identify the device typology. In particular, they report what their

needs are, like power or voltage, and also what are their synchronization activity to reach the energy and power dispatch objective.
In the passive ports instead the devices don't communicate their status, so the recognition of this kind of hardware is done by IEM through their behavior analysis, so it takes more time to identify the device typology.
Another fundamental task of the IEM is to implement energy management control goals as fast as possible. As already said, the integration of the IEM is realized by the DGI devices lying on each IEM nodes. Therefore this kind of architecture made up of DGI both for IEM and IFM nodes allows to easily scaled and re-scaled whatever is needed, is it need only to change the number of IEM and IFM nodes without intact the behavior of the whole System.
One of the most likely events that an IEM must struggle is the transition between islanding mode and grid-tie mode, and vice-versa. When the System is going to connect to the 69kV distribution grid, everything must be balanced (frequency, voltage magnitude, and phase) in order to have a smooth link; this is achieved thanks to an IEM node placed in between the grid and the FREEDM System (see Figure 3.2). In the same way, when System is islanded from the 69kV grid, it must achieve an instantaneous power balance to connect to the 12kV bus, and again this is possible by using the substation IEM that can regulate that primary circuit bus. And is precisely this transition from grid-tie to islanding that causes the most substantial operation for the IEM in terms of time-sensitive, because of the power balance must be as fast as possible and so it needs that each node interacts with each other. A further islanding operation could be that one of the single IEM which control one microgrid, so once this is done, the detached microgrid must be supplied only by its DRERs or DESDs.
Another task of the IEM is power optimization, but this is a lighter time-sensitive application. In the FREEDM System, the optimization could be a long and a short term, and include several optimization goals:

- Minimum carbon cost
- Minimum operation cost
- Optimum voltage regulation
- Minimum circuit losses
- Maximization of the utilization of DRERs
- optimum use in combination of DRERs and DESDs

Because of some of these optimization goals could be in contradiction one each other, the IEM apply the Pareto method to give weight to all the objective. According to this technique, an objective function Z is composed of a weight variable W and X which is the value to be optimized.

$$Z = \sum_{i=1}^{N} (W_i \cdot X_i)$$

The optimization of Z over the system controls variable is implemented across different scales of weights. Next, the Pareto method is to handle the multiple optimal objects achieved and conclusively choose the weights which provide the most robust solution (see [25] for applicative examples).
The IEM also deal with power dispatch; in fact, it has to be able to modify the load power supply modality according to the various operational situation (max grid exploitation, "green energy" from only DRERs, battery plus DRERs). For example, if the Systems switch to islanded mode because of a failure to the primary grid, the IEM automatically change the load power supply from grid to DESD (or DRER, or both together), and reduce the power availability to those loads that have low priority, focused the majority of the battery power to the higher priority loads. The IEM can create a sort of non-increasing priority list of all the nodes attached to the system, thanks to the DGIs, in fact, each node communicate its information, and then an algorithm takes the right priority level, allowing the IEM to dispatch the DESDs and DRERs power accurately.

Intelligent Fault Management

The *Intelligent Fault Management* has the hard task to ensure reliability and stability to the FREEDM System through DGIs. The first action that it is supposed to do is the recognition and localization of the fault. The fault current of this revolutionary System due to the use of the SST will be slower than the standards. This fact makes the job of the IFM more complicated in terms of time behavior and exactly fault recognition and location.
The IFM algorithm of fault detection is not based on the magnitude of current like in the most common system, but instead, consist of a current rise-based fault identification method. The device current crosses a digital filter to analyze the wideband-phase shifted component. So the original device current and the output current of the filter are then together analyzed to detect irregular rates of rising or unusual high current peak. In the FREEDM System,

the fault current is usually limited to 200% of the rated current of the circuit [23]; so thanks to this new method that provides faster fault detection, these common faults will be easily handled by the System.
Regarding the system protection again fault propagation, a quite complex example of the solution used in the FREEDM System is showed in Figure 3.4. Whether the sum of the current in a loop is not null means that there is a fault in that loop, the IFM detect the failure and send an error signal to the FID which reacts appropriately; on the contrary, if the current is zero this indicates that there is no fault in that zone. The FID are all positioned in the proximity of the terminal of the multiple loops and never laying between the SST and the loads because the SST itself acts as a switch blacking the current when a fault is identified. SST switching time is very insignificant. In fact, the fault recognition speed depends above all on interactions and hardware time delay. According to [23], the average time for detection and reporting of the fault to the FID is 2ms, which is remarkably shorter than the traditional protection action.

Another work provided by the IFM is intelligent fault coordination and re-closing. Because of the FREEDM System looped structure is characterized by short distances among the various nodes, so the fault current usually result in very similar values generating fault coordination problems among ring bus security and generation security. Since the traditional fault coordination is based on detection of diverse fault current values to create a trigger level for defending the devices closing section of the system. Therefore in the FREEDM System, the *Intelligent Fault Management* apply a fault coordination and a re-closing method based on detecting the different characteristics of the various loops of the System for intelligent protection.

3.1.4 *Distributed Grid Intelligence*

One of the most characterizing skill of the FREEDM System is the distribution of the control algorithm, possible thanks to the embedded devices DGI scattered in all the IFM and IEM nodes. So the development of the basic software deployed in the DGI helps the System to reach this wanted intelligent management. Therefore the *Distributed Grid Intelligence* can be considered ad the mind of the System contains the "piece of intelligence" able to handle all the operation situation and the management of this *Energy Internet* system.
One of the DGI privileges is the possibility to apply a software based on:

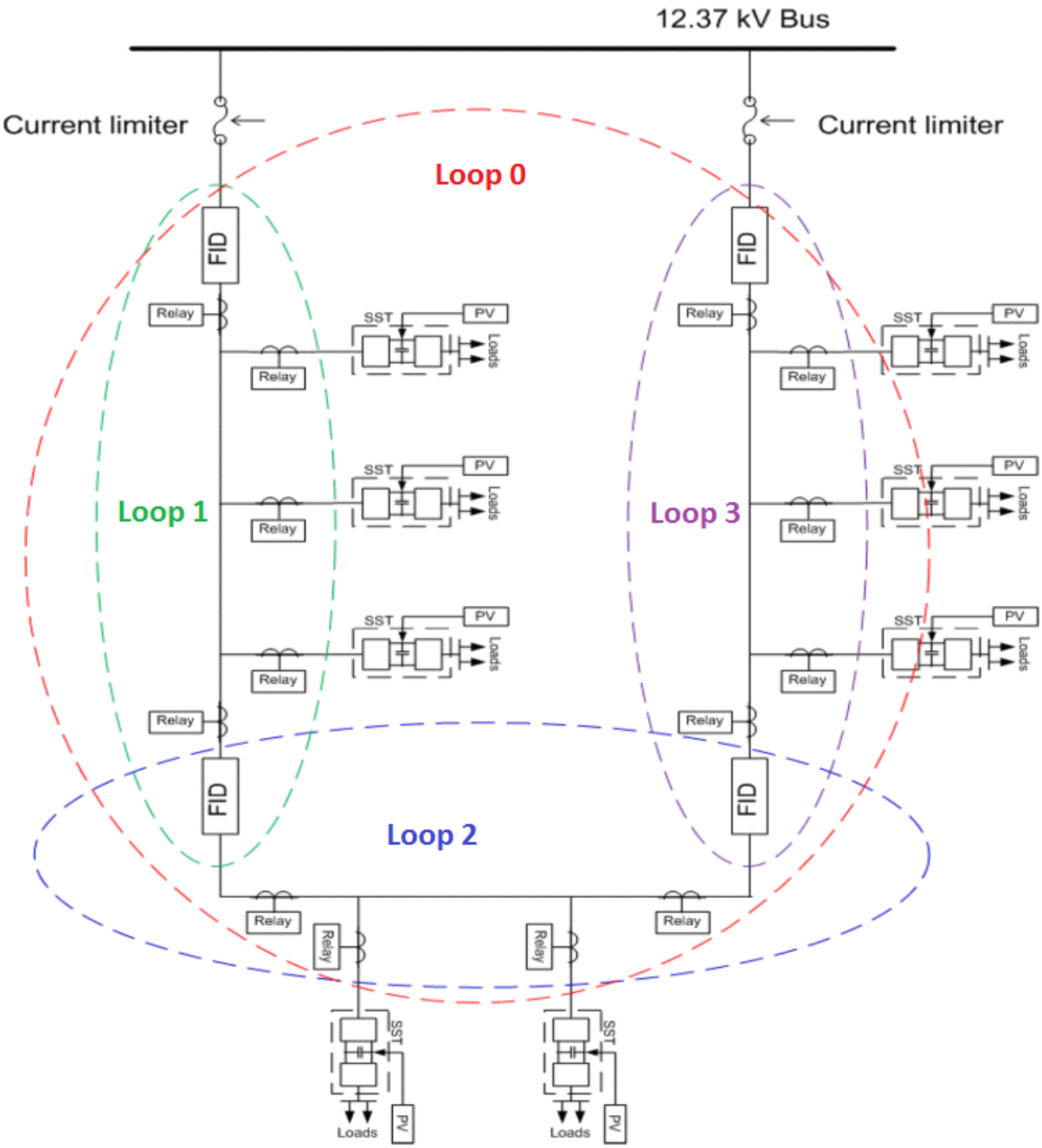


Figure 3.4: Example of a looped FREEDM System Protection

diverse control levels deployed as collaborating agents on multiple processors inside each IEM and IFM; and also on different timing behavior. The IEM

and IFM can work act both fast in the local domain and in a long period of time when working on power optimization.
Another DGI characteristic is the data security, in which each DGI command and information is never refused by the nodes, and also never diffused outside the System to avoid dangerous leakage of private information.
An example of the usual energy management job performed by the DGI is the distributed leveling of the load, considering the associated demand/supply. All the DGI communicate one each other trying to provide in an efficient manner the energy demanded.
Instead to reach the economic optimization and profitable energy transfers among the various microgrids, in the DGI is used a cost function is integrated in addition to the power leveling algorithm. According to the Test performed in [23] the System is able to give autonomously optimum answers also to very extreme demand for power.

3.2 Solid State Transformer State Of The Art

Before arriving in our homes, electric power should overcome three fundamental steps:

1. Generation phase, in which the MV energy produced by the various existing sources is transformed in HV by a transformer to reduce the transportation loss.

2. Transmission phase, once changed in HV the energy is branched through the HV cables to travel long distances.

3. Distribution phase, in which the HV energy is first transformed by an HV/MV transformer and delivered to be delivered to industrial and commercial users, and then by an MV/LV transformer is brought to LV for the residential use.

As we can see the power electronics, including above all the transformer, play a very critical role in the actual power system. For this reason, it is considered very important to try to improve as much as possible the actual technological component that makes up the whole system.
However, with the advent of new discontinuous energy generation resources like wind and solar energy, and the developing of the Smart Grid, raise the need of not only guaranteeing supply-demand balance but also in considering the dynamical behavior of this new power grid environment. So power electronic improvements are becoming more and more necessary to provide to the Smart Grid a smoother injection in the actual electric system [26].
In fact, in the past, power converters have found their complete application in transmission and distribution phases of the electric systems, for example in flexible AC transmission (FACT) devices, HV-DC device, static synchronous compensator (STATCOM) and static Var compensator [27]. However, the field that in the now day they are most widely used and needed is the increasingly diffusion world of the renewable energy source.
Another power converter that can be an advantageous element for the Smart Grid applications is the Solid State Transformer, awarded by the Massachusetts Institute of technology as one of the ten most emerging technologies in the power electronic [28].
The evolving trends of "normal" transformers have been essentially concentrated on the development of innovative magnetic material, better insulation elements, a more efficient building process, and other business factors.

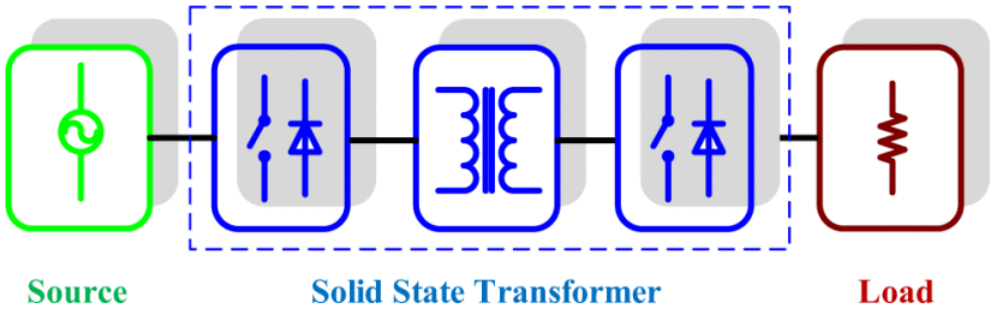


Figure 3.5: Solid State Transformer [27]

Therefore, for this reason, it has been preferred to add in the power electronics world a novel application in the distribution phase that could not only really enforce the diffusion of the Smart Grid but also revolutionize many other fields of application (as we will see in the following pages).

3.2.1 Characteristics

The birth of this revolutionary instrument is due to the efforts of the researchers, which have been centered their attention on improving the efficiency, power density, reduction of weight, volume and cost [29].
The SST is a controllable transformer based on power electronics which is intended to become the hart of the tomorrow distribution level of the power system, and thanks to the power electronics that the SST is able to perform multiple efficient operations with a volume and weight decreased a lot compared to the conventional iron-core 50 Hz transformer. In fact, the massive amount of iron and copper utilized in a conventional transformer make expensive the cost of transportation, bulky, and take up much space; so, reducing the volume and weight can affect economic benefits.

The SST move the voltage transformation from 50/60 Hz (medium frequency) to 3/10 kHz (high frequency). The essential configuration of the Solid State Transformer can be seen in Fig. 3.5, in which this high-frequency voltage is stepped up/down thanks a high-frequency transformer, and finally, molded back into the desired 50/60 Hz voltage to supply the desired load [27]. Thanks to the use of solid state semiconductor devices, it is able to provide important power quality features and other useful functions which are not

provided by the old transformer such as regulation of voltage and power factor that allows power flow control, voltage and load disturbance rejection, and low-frequency low-voltage ride-through features, fault current limitation and voltage sag compensation [30]. However, the key characteristic that differentiates the SST from a standard transformer is its capacity to buffer and uncouple the LV feeder sections from both MV and LV distribution grids. The SST can connect in a bidirectional manner the distribution system of the grid with the AC and DC loads, and the other energy source; for these reasons the SST can be considered a perfect instrument for the microgrid application characterized by a lot o different DRERs and DESDs.
The Solid State Transformer is also known as Intelligent Transformer because applying its integrated communication, and intelligence features became a perfect power management unit or a "power router" enabling plug-and-play capability, improving grid reliability and giving the highest priority to the green energy [24].

3.2.2 Topology

Unlike the functionality, the SST design and configuration are not so simple. The power devices and high-frequency transformer of SST can introduce stability problems in the system. Counting also the fact that the SST includes many ancillary circuits like heat sinks, cooling system, supplemental power, gate drivers and control circuits [27].

To better explain the general architecture of an SST, a baseline topology of a 20 kVA SST used in the FREEDM Systems Center was chosen (see Figure 3.6). That SST which is applied in the MV side of the power grid, for some application is required to add a transformer before the SST to decrease the injected voltage. Table 3.1 provides all the electrical specification of the chosen SST system in which it is possible to distinguish three different stages (see also Fig. 3.7):

- first off all in the AC/DC stage a phase-shift pulse width modulation rectifier, useful to obtain a high voltage and power system, converts the input single-phase AC voltage from 7.2 kV to 11.4 kV DC voltage (high-voltage DC side).

- Successively in the bi-directional DC/DC stage, the voltage is converted with a high switching frequency (10 kHz) from 11.4 kV DC voltage to 400V DC (low-voltage side) thanks to the dual active bridge (DAB) with

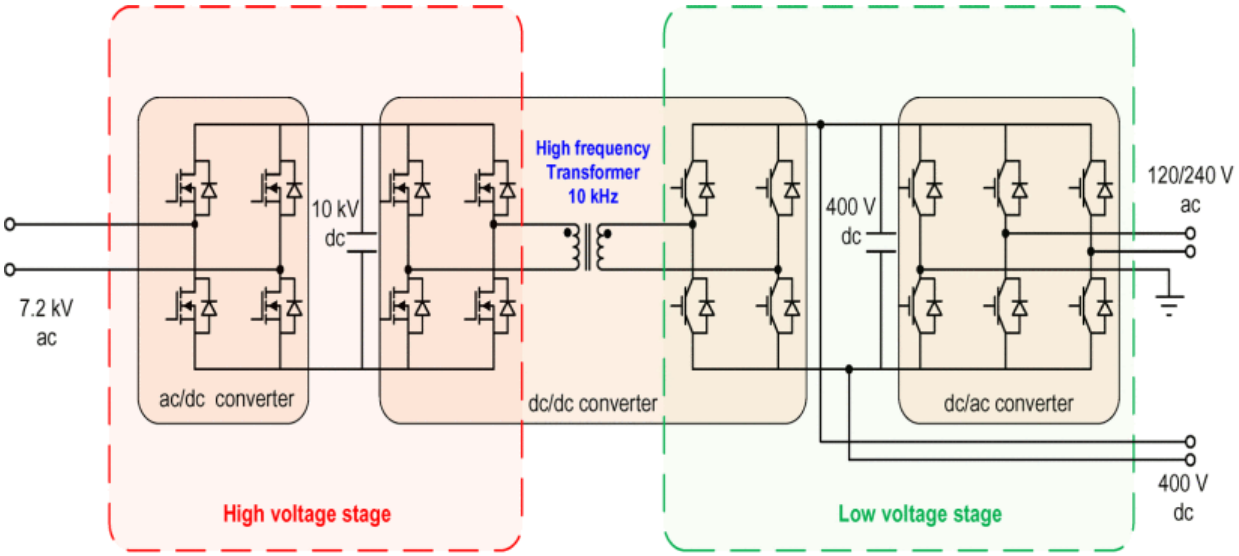


Figure 3.6: SST baseline topology [23]

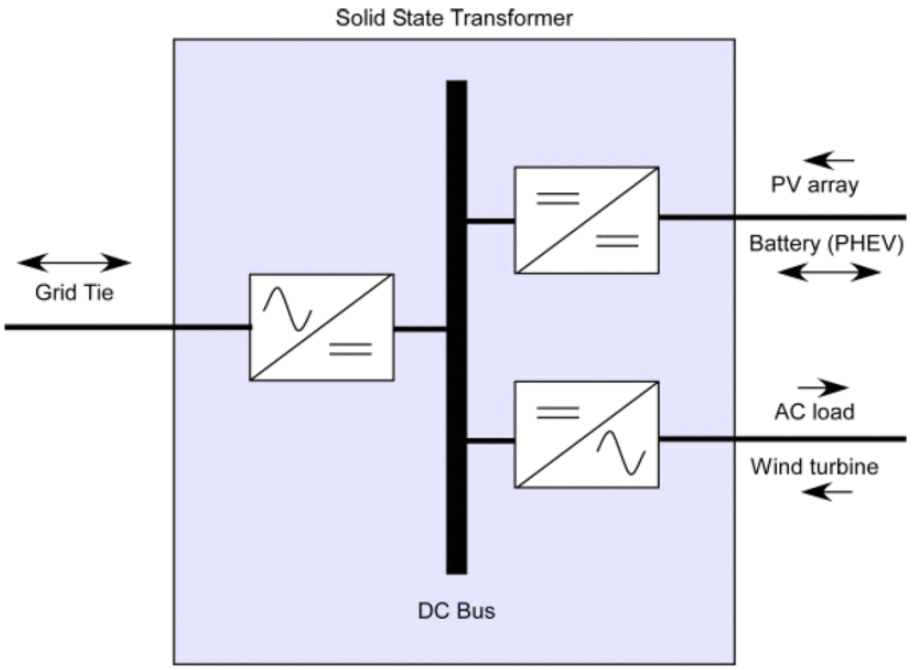


Figure 3.7: SST functional configuration [23]

a switching frequency of 10 kHz. This LV side gives us the possibility to include DESDs and DRERs in the systems.

- At the last stage, a DC/AC voltage source inverter transforms the voltage from 380V DC to 120/240V AC, providing the opportunity to integrate AC loads [31].

Input Voltage	3.6kV	Output DC voltage	200V
Input Current	2.78A	Output DC current	20A
Power Rating	10kVA	Output AC voltage	120V
High Voltage DC voltage	5.7kV	Output AC current	50A

Table 3.1: FREEDM SST System specifications

3.2.3 Efficiency

The switching devices in the high and low-voltage sides are silicon insulated gate bipolar transformer (SiC-IGBT) able to switch at very high frequency. If on the one hand they permit to reduce a lot the dimension of the passive components, on the other hand, they contribute significantly to an increase of the switching losses; therefore a trade-off between the size and the efficiency must be taken into consideration by the SST designer.
The actual efficiency of the standard transformers is around 96%-97% [27], exceeding this value is already a tough challenge for a single power converter, so considering the fact that the SST is made up of a cascade configuration it results in slightly lower efficiency than the traditional one.
The efficiency of the Solid State Transformer can be computed by estimating each components losses of every stage. Following the computation done by [32], the overall estimated SST efficiency should lie in a range between 91% and 96%, considering the high power rating dependency.
All the data analyzed up now are regarding the efficiency of the technological structure of the single device, so it is more important to extend the range of view, looking at all the system in which an SST should be applied. In

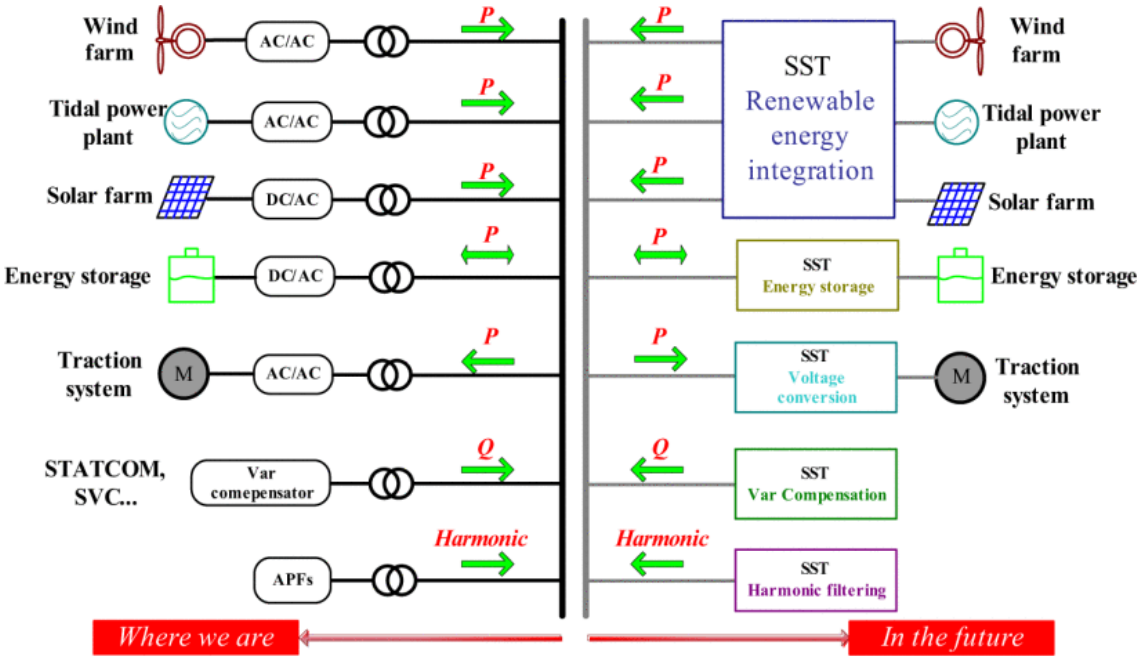


Figure 3.8: Overview of the possible application fields of a SST [27]

fact, ad already said integrating an SST on the Grid system allow a massive increase of the degree of the exploitability of all the renewable distributed energy resource, augmenting so the use of green energy and therefore reducing the cost of the electric energy gaining a higher system efficiency.

3.2.4 Other Applications

The application fields of the SST are not limited to the Smart Grid systems, but also can be applied to many other areas. Fig 3.8 depict in one side, the various field in which the actual transformer is applied, such as distribution system, transportation system, interfacing the power electronic device (FACTs, STATCOMs and so on) such as active power filter and reactive power compensator. On the other side, Fig. 3.8 shows the possible scenario in which the SST replaces that transformer; the results are undoubtedly a higher power density, a more compact, efficient and integrated system.

Traction Systems

In the areas of transportation systems as we can see in Fig. 3.9a the SST take the place of both the old transformer and the DC/AC converter, improving the power density bringing it from 0.2-0.35 kVA/kg to 0.5-0.75 kVA/kg [27], reducing the size and the weight of the systems gaining more space for the passengers. However, the most significant improvement that the Solid State Transformer could bring to the actual transportation field is the substantial augment of efficiency; in fact, according to [33] the efficiency of this system moves from 88%-90% with the old transformers to 95%-97% with the SST.

Renewable Energy Resources

Because of the more and more diffusion of the distributed renewable energy resources (wind, solar, tidal power), the researches focused their attention in maximizing as much as possible these resources. The SST offers a solid solution to integrate the DRERs in the actual distribution system of the grid, substituting the old power electronic and the tradition transformer. In fact, taking into consideration the example shown in Fig. 3.9b, in which in depicted a wind energy system that necessitates a more compact and simple architecture and a reactive power compensation to make stable the point of common coupling improving its efficiency. These necessities can be satisfied substituting the actual system made up of two transformers, a capacitor and a STATCOM [27] with a single Solid State Transformer as we can see in Fig. 3.9b.

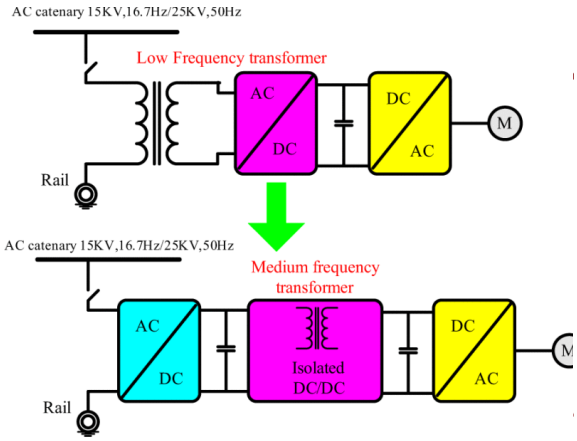
Energy Storage Devices

As already mentioned in the sub-section[topology], the SST provides the possibility to connect the DC and the novel AC battery in the distribution system of the grid thanks to its DC and AC link at low and medium voltage. One revolutionary concept that is becoming a trend is the electric vehicle diffusion, that place the problem of the home recharge of the batteries. In particular, the new technology of fast charging (see Fig. 3.9c) analyzed by the electric power research institute (EPRI), states the advantage of using an SST on a 45 kVA DC fast charging station .
The typical architecture of the recharge station is made up of one transformer, one rectifier, and one DC/DC converter. Substituting this actual

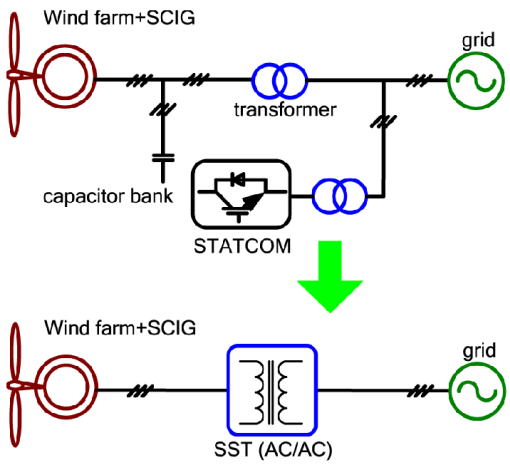
configuration with the Solid State Transformer like in the previous analyzed case this change allows to reduce the cost heavily and make the system simpler halving the component but above all bring the efficiency from about 90% to beyond 95% [34].

Unified Power Quality Conditioner

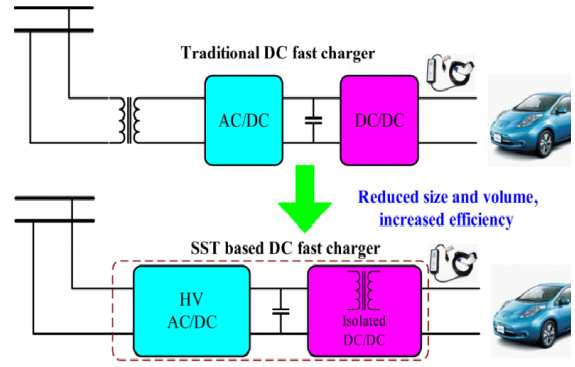
As already presented between the SST capability there are the reactive power compensation and the current harmonic filtering whose capability is handled by the controller bandwidth which in turn in managed by the Solid State Transformer switching frequency [27]. Together these SST skills are very fundamental to permit to use the SST as a unified power quality conditioner (UPQC) applied in the distribution stage of the system (see Fig. 3.9d). This novel UPQC that from an isolation and voltage conversion point of view is considered as a Solid State Transformer discard the series and the shunt transformer that now are working at line frequency, so again the system became lighter and smaller.



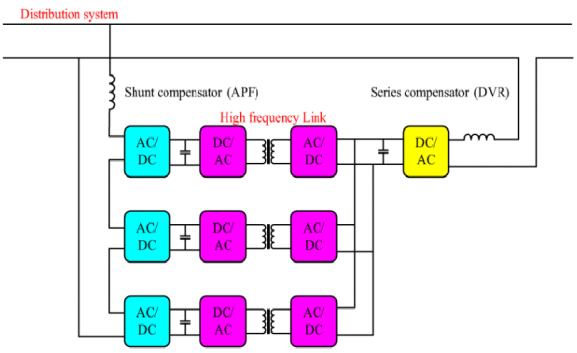
(a) SST-based traction system



(b) Wind energy system with SST



(c) SST-based fast charger



(d) SST-based UPQC structure

Figure 3.9: SST Applications [27]

3.3 Microgrid Application

Thanks to its router capability and the availability provided of both DC and AC output (see Fig. 3.6), the SST result to be perfect for integrating microgrids systems, in which all the three power flow ports (grid, DC and AC side) are achieved.
The various DRERs and DC distributed energy storage device (DCDESD) are connected to the distribution grid through the LV-DC link of the SST, and thanks to this latter that this DC-devices need only the use of a single stage conversion (DC/DC converter) as we can see in Figure 3.1. Therefore they do not necessitate to use other power electronic devices like in the classical distribution grid without the use of the SST, in which to host this DC sources and loads needed at a minimum a two-stage conversion (DC/DC converter in cascade with the DC/AC inverter).
Instead, regarding the AC loads, they are connected to the distribution grid thanks to the low-voltage AC output port of the Solid State Transformer and intermediated by a simple DC/DC converter.
Therefore the system that comes out by applying the SST in between the distribution grid and the microgrid is a more straightforward, more compact, reliable, integrated, and lighter microgrid system.

3.3.1 Green Energy Hub Testbed

As already mentioned, the FREEDM Systems Center has both the High-Voltage SST and Low-Voltage SST. So far, there has been talk of the SST state of the art analyzing in detail the most widespread HVSST. Now instead, it is useful to introduce the LVSST, which take a fundamental place in this project.
Exploiting this necessity to describe also a real application of an SST, considering the fact that there are no essential differences between the HV and the LV one beside the voltage magnitude of the system.
One real applicative example of the SST that integrates microgrid systems can be seen in the Green Energy Hub (GEH) Testbed of the FREEDM Systems Center, which the LVSST represent its breathing hart and it could be considered as a residential transformer (see Fig. 3.10). The operational scale of the GEH testbed, make the functional demonstration of the fundamental devices of a microgrid easier and simple to test and validate the various operational situation (like grid-tied and islanded mode), interactions and

management functionality among all the node of the microgrid, and also to evaluate new software and hardware before applying them into the FREEDM structure.
The particularity of the system is the presence of three 10kVA Low-Voltage Solid State Transformers that together are very useful to testify the possibility to work and interact with a system made up of multiple transformers connected to the main grid simulating and neighborhood application of Smart Grid. As explained before there exist the possibility to use the traditional transformer together with the SST, like in this case, in which an MV/LV transformer brings the current from 7.2kV single phase to 240V single phase.Moreover three line impedance (each one of 0.5 mΩ) are added to provide the voltage drop,

3.3.2 GEH Communication Architecture

To obtain all the characteristics that made it a perfect "intelligent transformer" such us monitoring, distributed control algorithm and communication functionality the distributed Grid Intelligence (DGI) units are implemented into the SSTs and in the various DC-AC load to match the requirements of the Smart Grid. In particularly before this project work, they were able to communicate through the North Carolina State University network using the computer boards (in particular BeagleBone Black explained in 5.1.2). So the computer board receive the command and then sent this command to a digital controller (DSP) that is linked to all the hardware of the SST.

Moreover, the DSP is able to retrieve data from the real hardware and communicate that data to the Board, which spread these data around the network. This is only to make an overview of the LVSST systems of the GEH testbed. For the rest, each LVSST structure is exactly the same as the one explained in the previous section, it has a rectifier which interface with the grid current, an inverter to work with AC-load and a DC/DC converter for DCESD.
Regarding the various load, distributed energy storage device and distributed renewal energy resource attached on the SST (which are also depicted in Fig. 3.10) it is not very fundamental going in the details, because they could change easily and continuously thanks to the SST skills, so each researcher could plug and play each device, that with the Solid State Transformer they can test and verify its working capability.

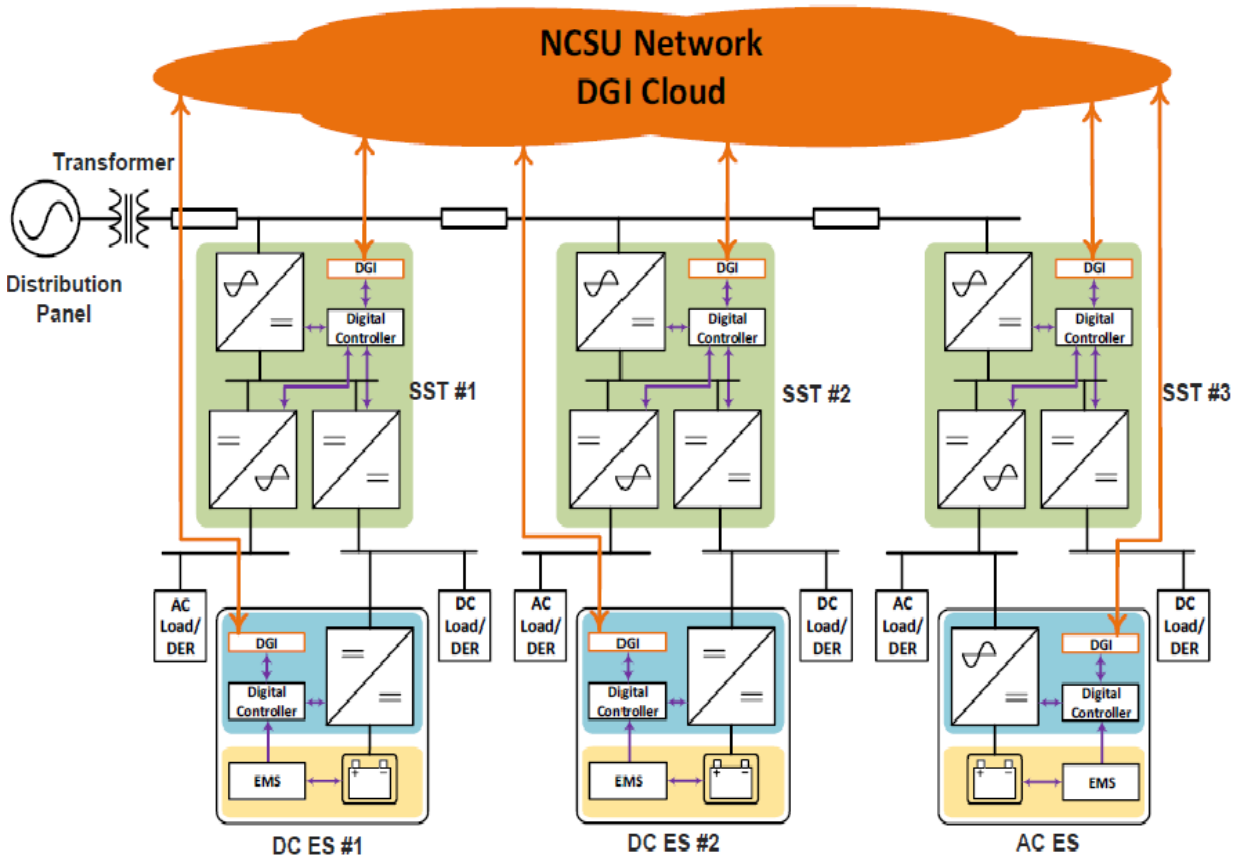


Figure 3.10: GEH testbed architecture

Communication Issues

However, this actual communication architecture has several problems and limitations. The operator that wants to communicate with one device plugged in the GEH system has to use an ssh terminal (remote control) to send command and receive information; this means that the operator should open and handle as ssh terminals as the number of the devices connected to the system. In The FREEDM GEH actually, we are working on eight DSPs, each one connected on a real piece of hardware (explained in detail in chapter 5); consequently, with this uncomfortable old communication structure the operator should menage eight different ssh terminals. Therefore this represents a big problem in terms of lack of scalability and limits of the number of devices

that can be connected to the studied system . Considering that the high number of terminals increases a lot the complexity and the human mistake probability, the communication system also result to be far from simple and challenging to use.
Another critical problem is the lack of security and privacy. The actual GEH testbed relies on the NCSU network DGI cloud (as shown in Fig. 3.10), and this means that everything is connected on to a WAN that doesn't fit the microgrid system simulated by the SST and so these could lead to a shallow capability of making private an secure all the info of the HAN.
All these issues testify the necessity of replacing this actual communication architecture with a smarter, simpler, safer, more reliable and scalable communication architecture for Smart Grid architecture.

3.3.3 An implementation Example

The goals of the GEH testbed are fundamentally two: to use a multi-SSTs system which could put in practice the essential FREEDM strongholds; and to demonstrate the possibility to apply that systems to residential households connected to an SST.
Fig 3.11 shows an example of real test implementation of the LVSST of the FREEDM Systems Center made up of: Three residential 10 kVA LVSSTs which provide three AC-DC hybrid microgrids with 5kV, 380V DC port, and 5kVA, 120V AC split phase port; two DC energy storage which integrates battery storage to each SST enabled microgrid; Distributed energy resources (Photo-voltaic Panel) that can be applied on both DC and AC side; A Distributed Grid Intelligence node embedded on each SST and DESD.
The demonstration tested the LVSST system in four fundamental use cases:

- Grid-Connected mode with a single SST, to exploit both Grid and DC energy storage source for AC and DC loads, and provide an active and reactive power dispatch capability thanks to the use of DGI.
- Grid-connected mode with multiple SSTs, providing support for distributions control functions such as var or voltage control; and to simplify the energy and power-sharing capability among the various SSTs.
- Islanded mode with one SST, this is one of the main strongholds of the Smart Grid systems in which the LVSST allows the system to operate as a single microgrid which is auto-sustained by its own DRERs and DESDs.

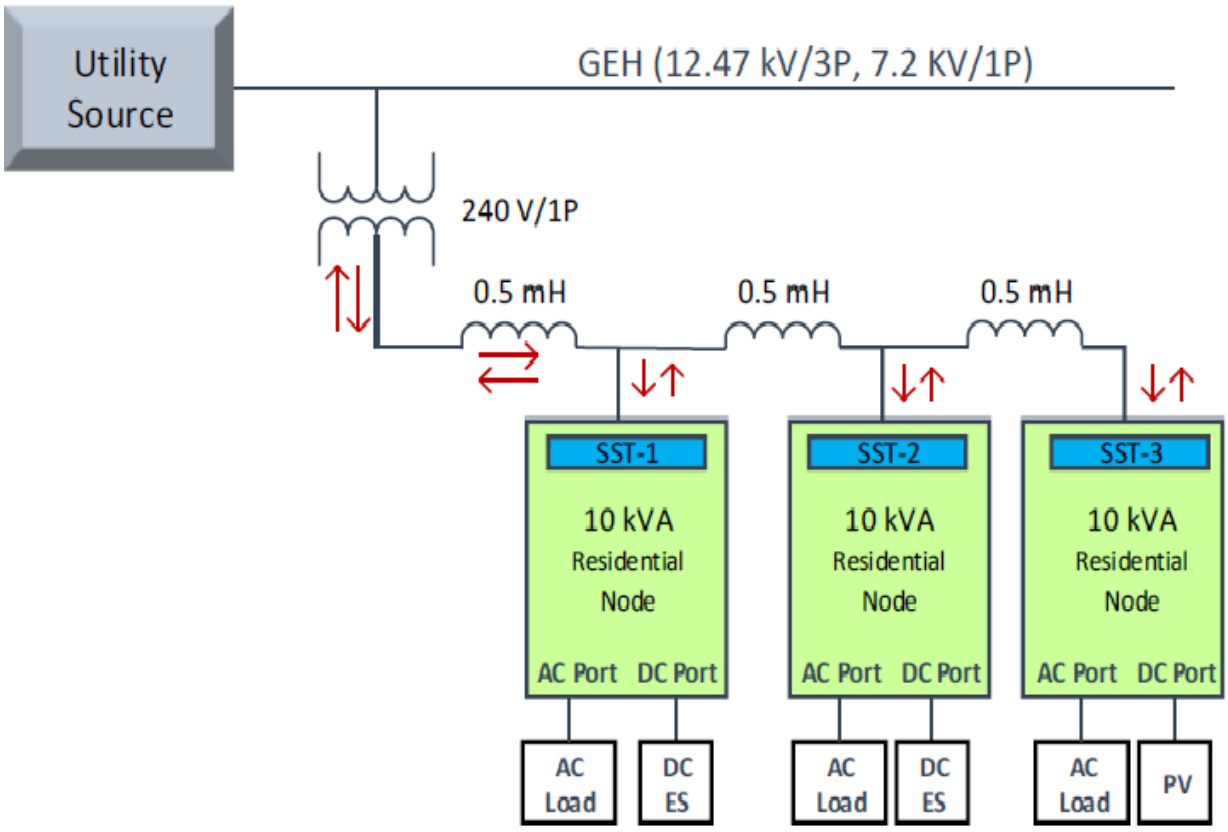


Figure 3.11: Multi-LVSSTs with bi-directional capability

- Islanded mode with multiple SSTs, which simplify smooth islanding and grid connecting capability of the multi-SST system; to simplify energy and power-sharing capability among different SSTs during islanded conditions.

As it is possible to see on Figure 3.11 One of the most important capability that is possible to see using the GEH testbed above described is the bi-directional capability of each branch of this system which provide the possibility to integrate and utilize the various bi-directional distributed energy storage devices.

Chapter 4

RIAPS

In a Smart Grid system made up of multiple and different subjects, like distributed generations (DGs) and loads managed by different communities, companies, and prosumers (customers that are able to consume and produce energy on their own), the need arises to use a platform able to distribute the "intelligence" throughout all these smart grid's agents [35]. So what a Smart Grid needs is to replace the old centralize control and management system characterized by a complex and chaotic communication network in which the exchange of control or measurement data could be very slow (see Figure 4.1). Example of this platform are Paradrop [36] and SCALE [37] which in a dynamic condition they do not provide a stable environment to host heterogeneous systems. A solution to this problem is a universal computing platform implementing the decentralized control, which implements core services essential for a stable deployment environment. Services like robust communication between the various nodes of the Smart Grid, distributed coordination, and management of information (sensors and actuators local data), support for time synchronization, and service to deploy and remotely handle the distributed applications (see Figure 4.2).

Challenges

However, to obtain the wanted Smart Grid platform there are some challenges to overcome like [38]:

- The analyzed control algorithm utilizes a multi-agent control method where the number of participating agents can be significant and changeable. Therefore, the controller and the hardware implementation must

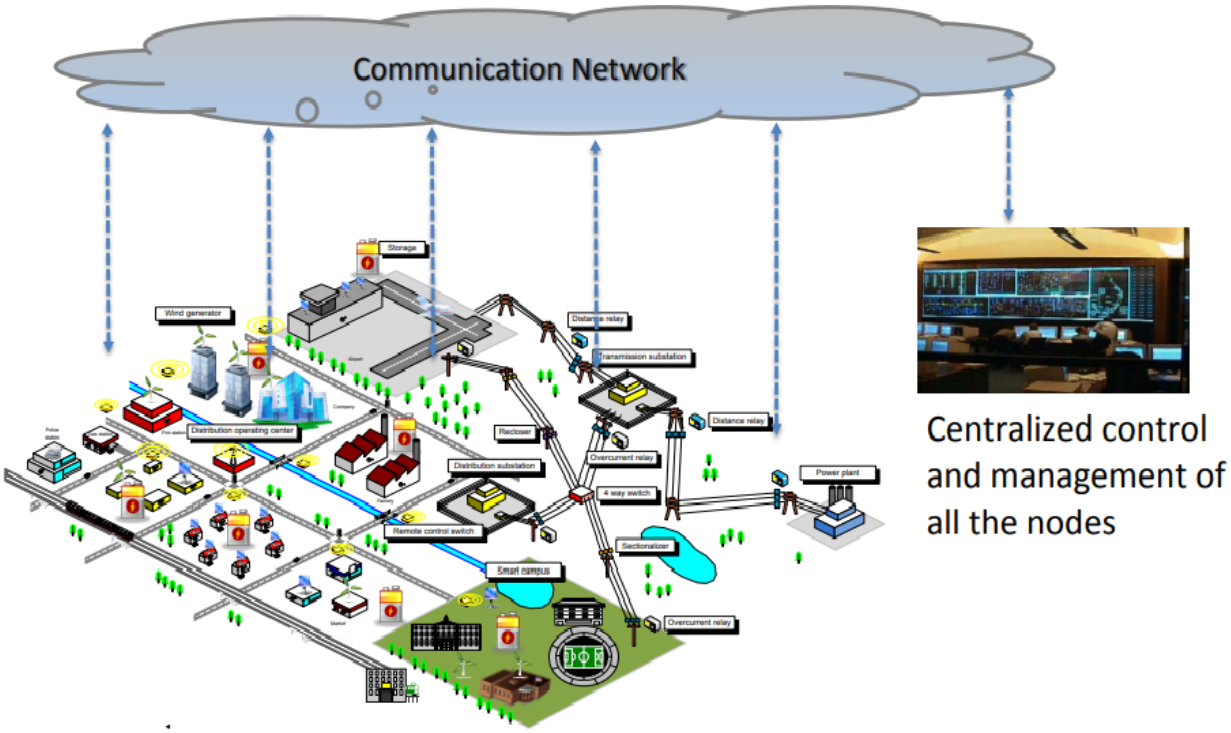


Figure 4.1: Power Grid with centralize control

be scalable and should support plug-and-play capacity.

- The hardware controllers need to have adequate computational capability so that the chosen sampling time step can be made small enough to reduce the quantization error produced by transforming from continuous mode to discrete mode.
- Although distributed control algorithms do not need refined communication network compared to centralized ones, reliable and precise data exchange are still significant towards system stability because of the small system inertia [38]. So Information exchange among distributed hardware controllers needs complement communication system.
- Each agent has to be able to establish a link with physical devices like the phasor measurement unit (PMU) [39].

Without these characteristics, the control systems described in the previous chapter cannot operate properly [40].

RIAPS is the platform that satisfies all these requirements. In fact, it provides all these services creating a very reliable platform for distributive intelligence. Hence for this reason in the FREEDM Systems Center, we have been used the RIAPS platform.
In the following sections, it is explained better what is RIAPS, its structure,how it works, and finally, a practical implementation is analyzed.

4.1 What is RIAPS

RIAPS stand for "Resilient Information Platform for Smart Grid" [41], it is an embedded distributed real-time computing platform developed by the North Carolina State, Vanderbilt University, and Washington State University.
One of the purposes of RIAPS is to provide a design-time and run-time software environment for building Smart Grid applications [42].
RIAPS was developed to act like computer software that provides functions to software applications in addition to those given by the operating system (in our case Linux) . So it appears like a "software glue" and for this reason is it considered as a "middleware," i.e., a software layer that stands between the application and the operating system (see Figure 4.6).

4.1.1 Characteristics

Here to prove the uniqueness of RIAPS, its most important features are reported:

- Discovery and deployment mechanism to host new node and run simultaneously multiple applications (Multi-Tasking) [40].
- To distribute the intelligence (software applications) and control capability to a multitude of computing local endpoint (nodes) reducing the traffic of the communication network so that each node can access local actuators and measurements augmenting the geological range (see Fig. 4.2).

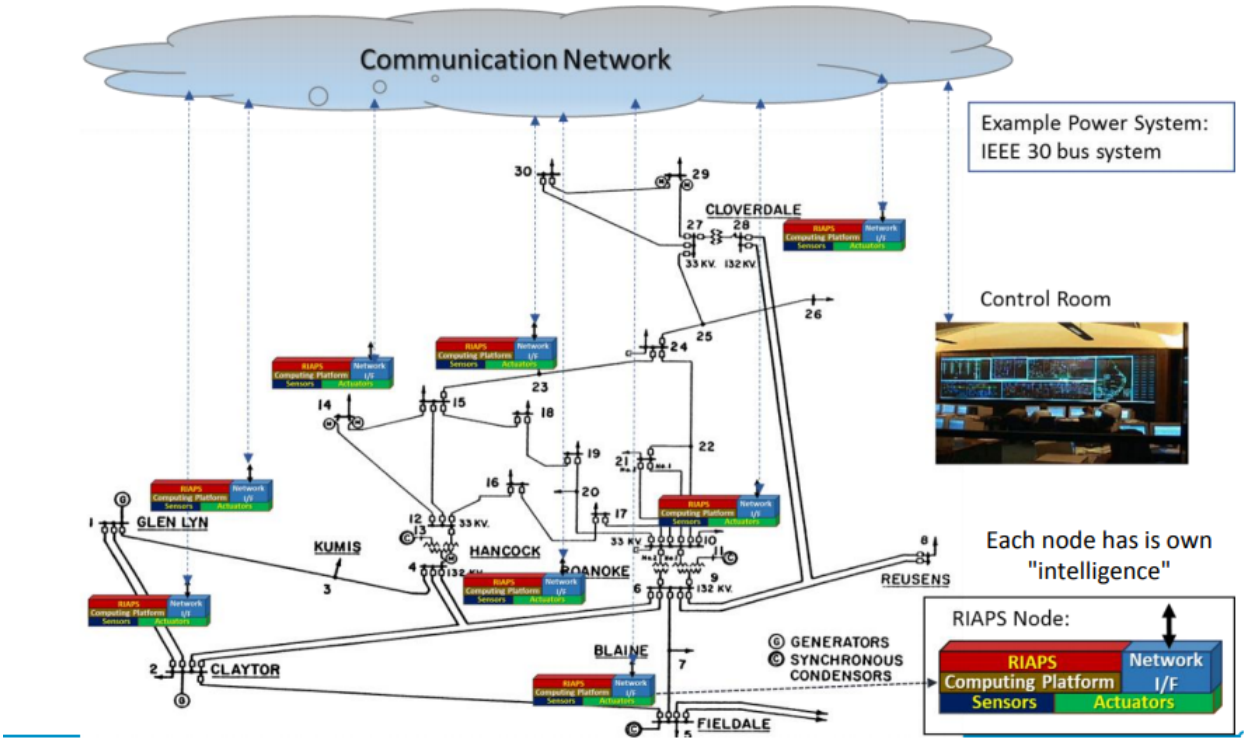


Figure 4.2: Smart Grid system with RIAPS [43]

- Time synchronization to allow a better control action and to coordinate all the mechanism.

- To improve the speed of local action by avoiding latency. This could be very important for example when a fault occurs to a connected part of the Smart Grid, and this part became island-mode unexpectedly; therefore this information must propagate very quickly to the controller and hence to the rest of the grid to ensure a safe service and a stable and smooth operation.

- To improve reliability, by decreasing dependencies on multiple devices and complex message interface by dodging or blocking possible malfunctions, failure, overloads, and data missing.

- To allow the Smart Grid to become a real Internet of things application [44] thanks to the characteristics provided like:

- Evolve over time by sustaining rapid update and enhancement with low cost and small structural impacts.
- Scalability to manage a high number of users sensors actuators and devices (DGs, loads, breakers , transformers, relays) [42]
- Interoperability among heterogeneous systems
- Modularity by creating the system as a combination of inter-operable elements that communicate through lightweight mechanisms.
- Flexibility in developing heterogeneous services with diverse features and requirements.
- Multiple communication paradigms (like synchronous and asynchronous, it is explained better in the 4.5 paragraph)
- Security to ensure authentication, data access, and privacy.

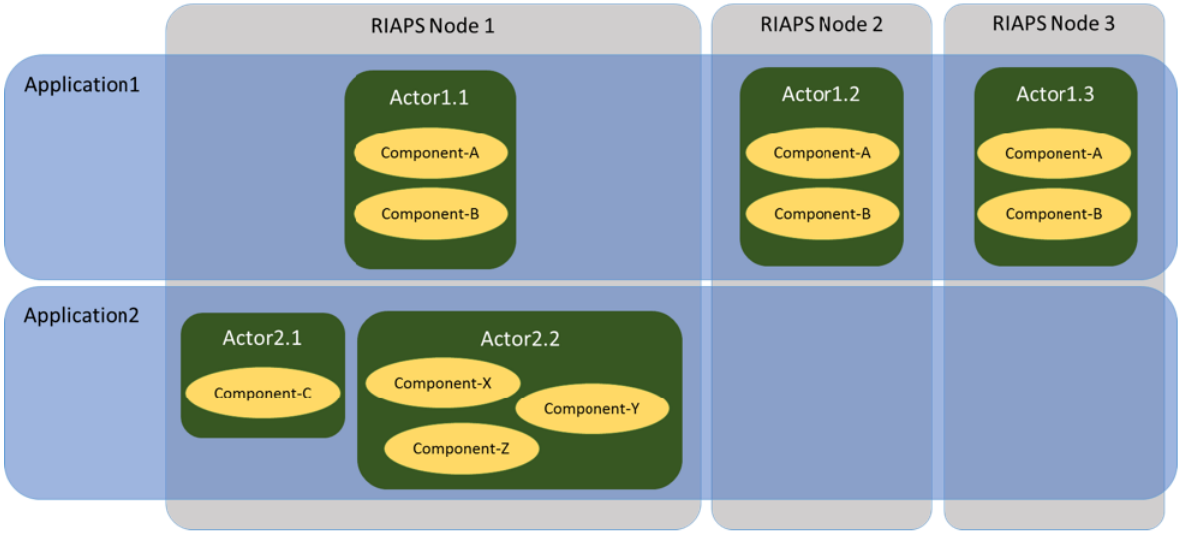


Figure 4.3: RIAPS component overview

4.2 Architecture Overview

To understand how RIAPS works it is fundamental to know its frameworks and all the architectural aspect.

4.2.1 Component Overview

The aforementioned distributed intelligence in RIAPS platform is called applications which occupies the top layer of the RIAPS component architecture and is made up of two main models (files), the .riaps and .deplo file. The applications are deployed on the platform by a single node called *Control Node* that allows a smooth deployment of the control algorithm to the RIAPS *Target Nodes* (see Fig. 4.5) through the RIAPS deployment mechanism (the .deplo file, explained better in 4.3.2). One application is made up of one or more nodes containing actors that are OS processes which are used to realize an abstract function, like state estimation or data logging, and run simultaneously on one node or in parallel on multiple nodes [41] (see Figure 4.3).
Instead, the job of the .riaps model is to describe the function and interaction of each node inserted on the RIAPS platform.

Actors act as application manager since they are made up of one or more

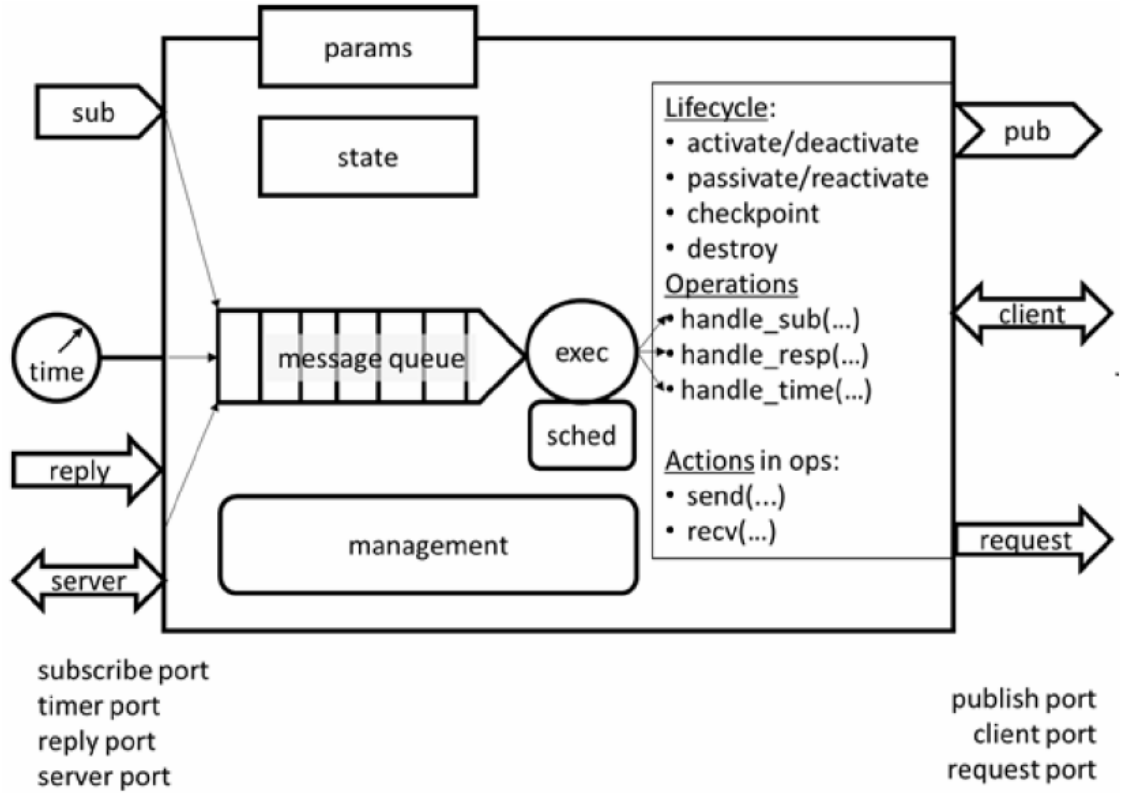


Figure 4.4: RIAPS component structure [43]

components to which it provides the function to control and handle them remotely.
The component which contains the functions to be executed, it is a reusable architecture block in RIAPS platform that implements distinct physical functionality, like measurement sensing and might encapsulate complex python or c++ functions. Each component can possess different kinds of ports used to supports its function like interacting and communicate with other components (see Figure 4.4) . A particular type of port is a timer port which wakes up every time step, it is usually implemented to achieve discrete calculation and acts as a programmable time-based triggering or as a periodic reference of messages (including the current time-stamp) for the component. Other ports include subscribe and publish port to subscribe and publish messages respectively (covered in section 4.5).

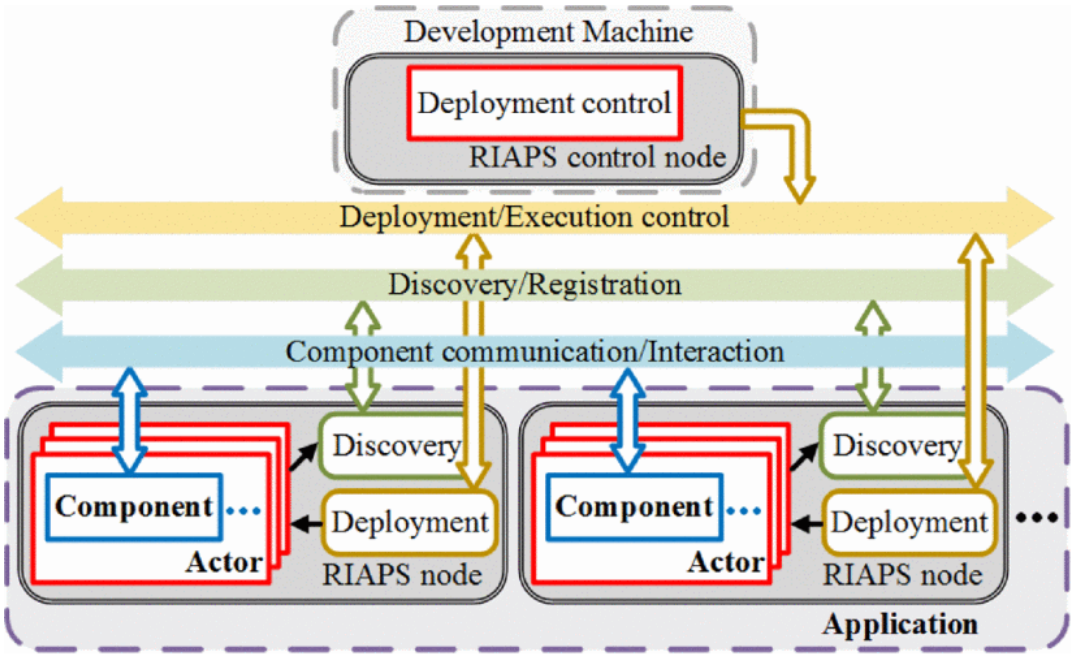


Figure 4.5: RIAPS Architecture [38]

4.2.2 Component Execution

Before starting to speak of the execution of a component, it is necessary to explain how a component is built and managed.
Originally a RIAPS node is like a shell without a soul (i.e., application code) when the RIAPS developer wants to use that node with a defined function, he utilizes the *Control Node* (see Fig. 4.5) that contains the *Control Application* to provide to the *Target Node* that function; so now the node has got its soul which permit it to be executed.

The control application provides three critical services:

1. To elaborate a download pack of the RIAPS application
2. Deploying the download pack to the chosen RIAPS nodes
3. Managing the deployment service operating on the RIAPS nodes to

launch, stop or remove the applications.

The RIAPS developer can download and control the execution of RIAPS applications whit a Linux script or better with an already implemented GUI that makes the control app much more intuitive (RIAPS CTRL showed in 5.2.1). The standard application components that now contain the pieces of code (the distributed functions) are defined as single threaded. An actor could hosts more than one component that could run in the same actor as separate threads, so an actor could be considered multi-threaded. It means that that the RIAPS platform allows to a thread to execute the codes of the components when the component is triggered. Therefore a component runs in its own execution thread, and the component execution engine issues the thread when a component is triggered, i.e., when a message arrives for which the component was waiting. The trigger method that releases the thread can be event- or time-triggered, more precisely when [45]:

- The time expires
- The stare of the port change
- An operation is accomplished

These three trigger methods provide the application logic of the component. Whether a component is triggered, both because information has appeared or because of a timer has expired, a service named *Discovery Service* is issued (covered in section 4.4). Each communication port has a correlated method whose name is determined from the type of the port (see Figure 4.4). The requested *Discovery Service* is expected to read the triggering message from the port (note that this message could be a time-stamp if a timer produced it), execute some actions and possibly send messages out through ports, and finally return.

4.3 Run-Time Application

As it has already mentioned the purpose of the RIAPS run-time system is to provide a software framework for developing distributed applications that lie in the top layer. It acts as a middleware between these applications and the operating system, and it is divided into two sections (see Figure 4.6): *Component Framework*, and *Platform Managers*. The first is more close to the application, instead the latter is strictly dependent on the operating system.
These two layers implement all the services that will be utilized by a developer in supporting the build of the distributed applications.

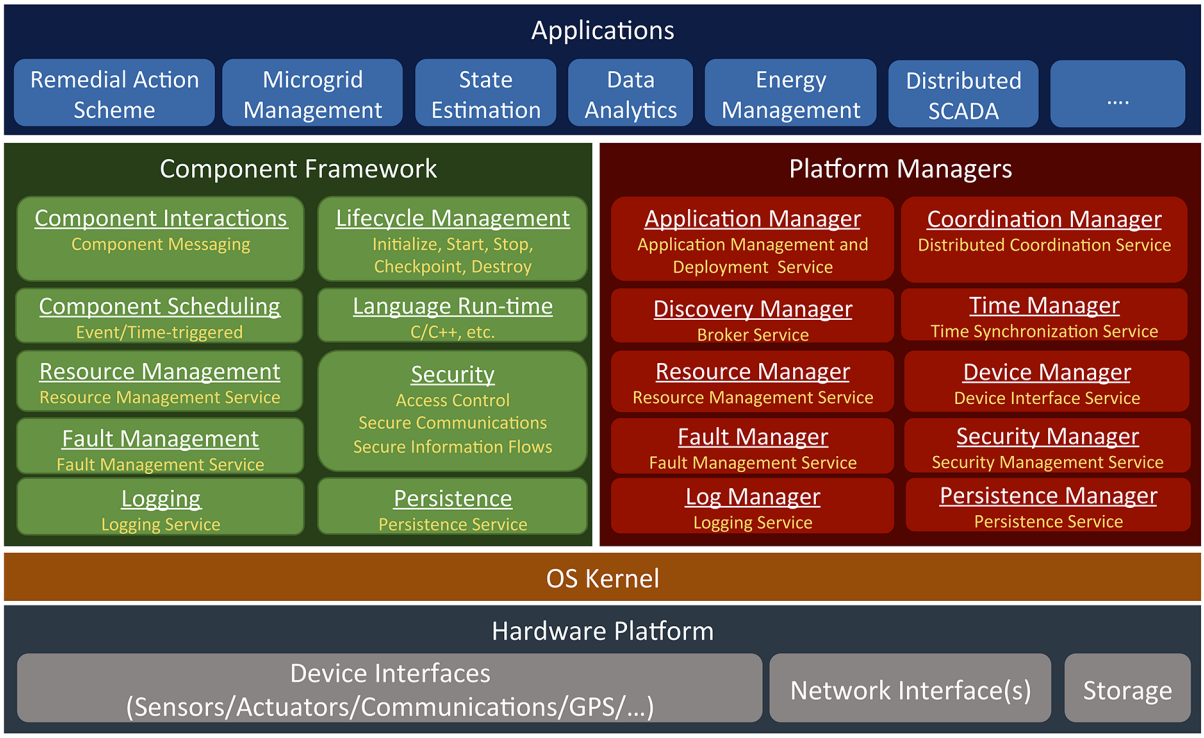


Figure 4.6: Middleware: RIAPS run-time system (Source: [35])

4.3.1 Component Framework

The *Component Framework* is made up of a collection of software libraries that are connected dynamically with the application components. Therefore the *Component Framework* section is the place in which the implementation

of the several middleware libraries resides. The purpose of the *Component Framework* is to implement an as high as possible level of abstraction to allow to create resilient, elaborate,and distributed applications on the RIAPS platform.
The middleware libraries comprise:

- The *Component Scheduler* that performs the *Component Handler*, to manage the various component executions.
- The *Component Interaction* library which deals with the interaction between component it contains the library of Publisher/subscribe request/reply and other communication patterns.
- The *Life-cycle Management* services which remotely sustain and manage the application components.
- The *Language Run-time* libraries that preserve all the used programming language libraries like C++ and Python.
- The *Resource Management* support very useful to check computing platform resource availability/utilization.
- The *Fault Management* support that identifies and moderates irregularities in application components.
- The *Security* library that establishes secure node interactions.
- The *Logging* library to save component events.
- The *Persistence* library that permits to save data in a persistent manner

All these libraries are connected with the components employed to build an application.

4.3.2 Platform Managers

Platform Managers are specific operating system processes, executed as daemons in the Linux operating systems [35]. This layer is made up of different platform services that operate as autonomous processes and implement system-level control capabilities. The services provided by the platform managers layer are:

- **Deployment Service**

  The *Application Manager* which remotely allows to deploy and manage the applications, contain the *Deployment Service*, an application that runs on each target nodes, and it is responsible for launching all RIAPS processes, i.e. the application actors and the *Discovery Service* (see Figure 4.5).
  The first action that the *Deployment Service* makes is trying to connect to the RIAPS control app. After that, it activates the *Discovery Service* process and waits for instructions from the control app. The Control Application can then deploy the files of the RIAPS applications and then command to the *Deployment Service* to start them. After that, *Deployment Service* starts the applications actors, and then it:

  1. Creates an interaction channel to them so that they can communicate to the *Deployment Service* different problems (irregularities in component code implementation) and they can receive information from the *Deployment Service*, like when a new application actor is added on the network and is available to interact.
  2. Controls the actor and recognizes if it has stopped unexpectedly (in this case it restarts the actor automatically).

- The *Discovery Manager* that defines available links between components on the same node or of different nodes; and contain the *Discovery Service* (explained in detail in paragraph 4.4)

- **Distributed Coordination Services**

  The *Distributed Coordination Manager* that provides the *Distributed Coordination Service* which allows group membership, an application component can dynamically create, joint, leave a nodes group which promotes fast communication among the members. A group can also select a "leader", i.e., a component which makes global decisions, the "election" is fault tolerant and automatic, and the members of the group can communicate directly with their leader. Moreover, group members can "vote" in a consensus process which reaches an agreement above a value.
  Group members use a combination of the above features to agree on a control action performed at a scheduled synchronized instant of time.

In a Smart Grid application, for example, the group is made of microgrids the consensus could be on voltage and frequency value and a time-coordinated control action could be when the microgrid pass from islanded to grid-tied mode.

- **Synchronization Service**

  The *Time Manager* which implements extremely high precision timing and time *synchronization Services* that keep synchronized the system time of all the nodes deployed on the RIAPS network.
  Thanks to this service application can query the global actual time, query the status of the service and above all sleep until a specific time. the *Synchronization Service* is based on the PTP IEEE-1588 protocol for time distribution [42]. The source used by this service to retrieve the absolute time can be a network time protocol server (NTP) or a GPS receiver connected to the target nodes. Once the clock time source is chosen, thanks to the PTP all the clocks in the RIAPS system will follow the primary clock source with an accuracy of 10 µsec [46].

- **Resource Manager**

  The *Resource Manager* check the computing capability to guarantee that the Platform Managers and the components can work concurrently.
  Moreover thanks to the resource manager the application actors can be modified to limit their:

  – Memory utilization

  – CPU utilization

  – Network bandwidth utilization

  – Disk space utilization

  For such limited actors, the deployment service modifies the operating system configuration accordingly and keeps track of the actor performances. When the application actor is about to surpass the limits, it will receive a proper message from the deployment service, and the application actor behaves accordingly to this message.

- **Fault Management Service**

  The *Fault Manager* implements for the node the *Fault Management Services*, that works together with the *Deployment Service* to avoid the faults. The faults can arise at any level: in an actor, component, communication network, or system service [43].RIAPS developers have to be able to develop applications that can recover from faults everywhere in the system. The deployment service is in charge of detects such errors, and when it finds them two cases could be manifested:

  1. The deployment service simply notifies the application components through a special message, and they behave accordingly to this message trying to resolve on their own the faults.
  2. The deployment service reacts to these faults independently relaunching the application actor if crashed.

- The *Device Manager* which allow the interaction and the management of the connected input/output devices and also provide the Device interface services (explained in the paragraph 4.6).

- The *Security Manager* which controls all the authentication, digital signatures, and controls the keys. Application packages are first encrypted, compressed, and signed cryptographically before the deployment. The receiver node verifies cryptographic signs, decrypt it, and install the applications.
  All the levels of the application communications are protected by the elliptic curve encryption CurveCP on the messaging layer. All the interactions are protected via private/public key-pairs that are produced dynamically when the application is deployed. Keys are introduced whenever a network connection is established, and they are part of the deployment package, collected in a certificate repository on target nodes..

- The *Log Manager* provides to all the log activity a single gate point on the nodes.

- The *Persistence Manager* which provides a sort of repository to store the data in a non-volatile way.

4.4 Discovery Service

The *Discovery Service* is considered one of the most important functions implemented in RIAPS platform because it carries out many essential tasks that allow RIAPS to be a revolutionary embedded distributed real-time computing platform.

4.4.1 Needs for the *Discovery Service*

As it was said, the RIAPS platform matches perfectly the needs of the decentralized control, that provides the systems a fundamental characteristic, the scalability.
However, if on the one hand, this platform provides great flexibility, guaranteeing all connected nodes to change over time; on the other hand, it generates some addition necessity.
For example, in one of the application fields of RIAPS platform the Smart Grid systems, a homeowner can decide to buy a photovoltaic plant and disconnect himself from the main grid using only his local photovoltaic grid or vice-versa. Another example could be when the power demands change, or faults happen, sections of a connected grid might detach from the main grid and so become islands. Furthermore, if the islanded section of grid wants to reconnect the main grid, it is required to obtain information about its voltage frequency and phase, data that it can obtain only if the actors of the island and all those of the grid discover each other.
Therefore it results essential that this information diffuses very fast to all the other nodes to guarantee stable performance and uninterrupted service.
Given these examples, it is simple to understand because of the RIAPS platform and so the entire Smart Grid needs to know:

- When a new node joins the network
- When an already registered node leaves the network
- When applications and their components come and leave; because of the *Discovery Service* is in charge to monitor the status of the services and the message type of the applications

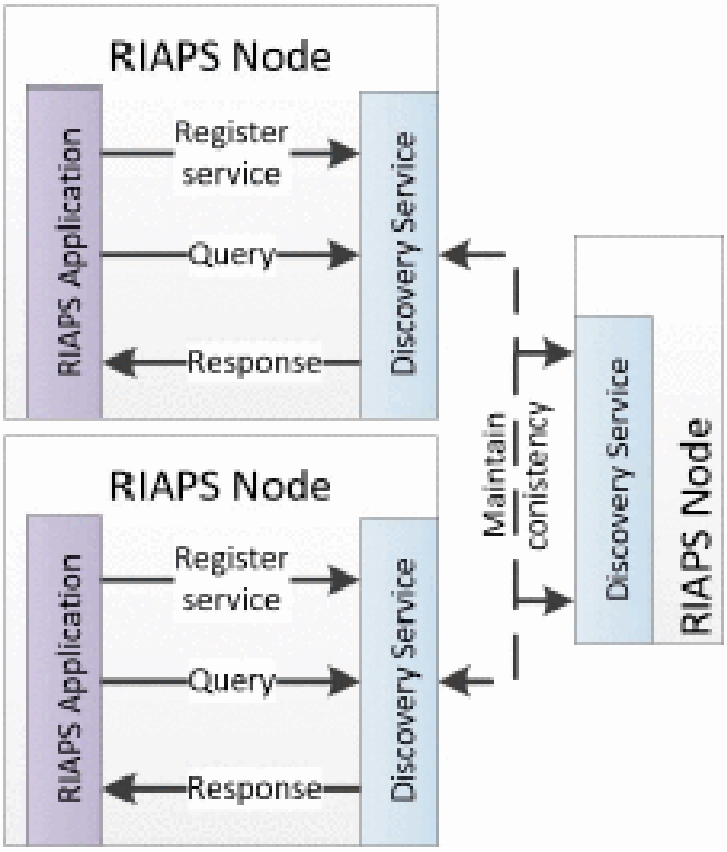


Figure 4.7: *Discovery Service* infrastructure [35]

Another crucial characteristic provided by the decentralized implementation is fault tolerance. Centralized control is inadequate because it does not provide scalability and it can result in a single point of failure.
Fault tolerance is very important because any node or communication link could fail accidentally and unpredictably. So it is needed that the *Discovery Service* bounds the local failure avoiding to spread it for all the grid creating a huge collapse.
So for all these reasons, the *Discovery Service* result to be very essential and need to be deployed on each node across the network (see Figure 4.8).

4.4.2 How it works

RIAPS intends to grant decentralized and modular solutions for all the services of the platform so that they can be utilized in various applications. Hence, the *Discovery Service* works as an independent process on each RIAPS target node that listens for messages coming from different components. So essentially it acts like a broker managing the routing of messages.
In Figure 4.7 is shown an example of how the *Discovery Service* operates: It serves as a registry storing the app service information and actors associated details like communication protocols, message types, ports, and IP addresses; then it forwards this new information to the neighboring nodes. Therefore now the ports of the message producers, in this example a query (but can also be a client, request, and publish port, explained better in Sec. 4.5) and the ports of message users (server, reply, answer, and subscriber) can be now connected. So when the RIAPS applications require app services, first, they query the *Discovery Service* included in their own node, and then that establish a connection with the RIAPS nodes containing the required application services; finally, the responses are asynchronously delivered to the nodes that had required the service.
The *Discovery Service* provides two different types of "registers" to store data to be chosen a priori by the developer, Redis and OpenDHT [43].
The first is used as a centralized database, running on the RIAPS control node to keep track of all the registrations. Redis (Remote Dictionary Server) [47] is an open source key-value database.
Instead, OpenDHT is an openly accessible lightweight DHT (distributed hash table) implementation in which the hash tables are distributed across all the network containing the registrations.
Therefore the *Discovery Service* relies on one of these two methods to store nodes details, and distribute the service information through the network. The distribution does not mean complete data replication on all the nodes; for example, OpenDHT collects the registered data locally and delivers it to a maximum of eight neighbors [48]. Moreover, for the *Discovery Service* that is using the DHT, there is no difference between “server” or “client” nodes, but they all run with the same rules acting as peers nodes downloading and uploading data.

Two distinct significant cases of network configurations can take place in the RIAPS platform:

1. When all the nodes are placed in the same local subnet. To detect the available *Discovery Service* handlers on the same local subnet, the RIAPS platform utilizes UDP beacons, peer to peer service of discovery for the local network [49], whereas a UDP message (user data protocol) is a protocol which allows sending messages as datagram (header + payload) [50].
   Periodically, every *Discovery Service* deployed in a node announces, through UDP IPv4 broadcast, its subnet address and attends for incoming beacons. Those UDP packets are asynchronously sent and received, and the *Discovery Service* handlers keep track for the list of received addresses. Before the *Discovery Service* elaborates the UDP packets, they are filtered to exclude UDP messages that are non-RIAPS specific. These UDP messages act as a heartbeat, i.e., when no packets are received from an already registered node throughout two-time intervals, then the *Discovery Service* excludes this sleeping node from the list of the peers. Instead, if it appears a UDP beacon from a new node, the *Discovery Service* logs this new node address in OpenDHT, which later appends it to the list of recognized nodes.

2. When the nodes are placed on distinct sub-networks. In this case, the UDP beacons cannot discover the nodes that are in different network areas; the foreign addresses must be transferred explicitly to the *Discovery Service*. Therefore, in this case, assuming that routing between the subnets are available, predefined gateways are used which implement the *Discovery Service* for the IP of the these external sub-networks.

Managing the ingress and egress scenarios

In the preceding pages, we have remarked that the *Discovery Service* based on OpenDHT works on batches of maximum eight nodes to share registration data and application service; however, however, we have not yet examined how to use the data collected in RIAPS. If a RIAPS application starts running, it announces its services in the *Discovery Service* of the application node. The *Discovery Service* saves these data in the DHT, and the DHT disseminates this newly obtained information through the batch of nodes.
If a node leaves the network, the *Discovery Service* contained on other nodes can still enroll new services or make new subscriptions. Instead, when a new node enters the batch, all the data collected in the network are available to the new node. At the starting phase, the new RIAPS application not

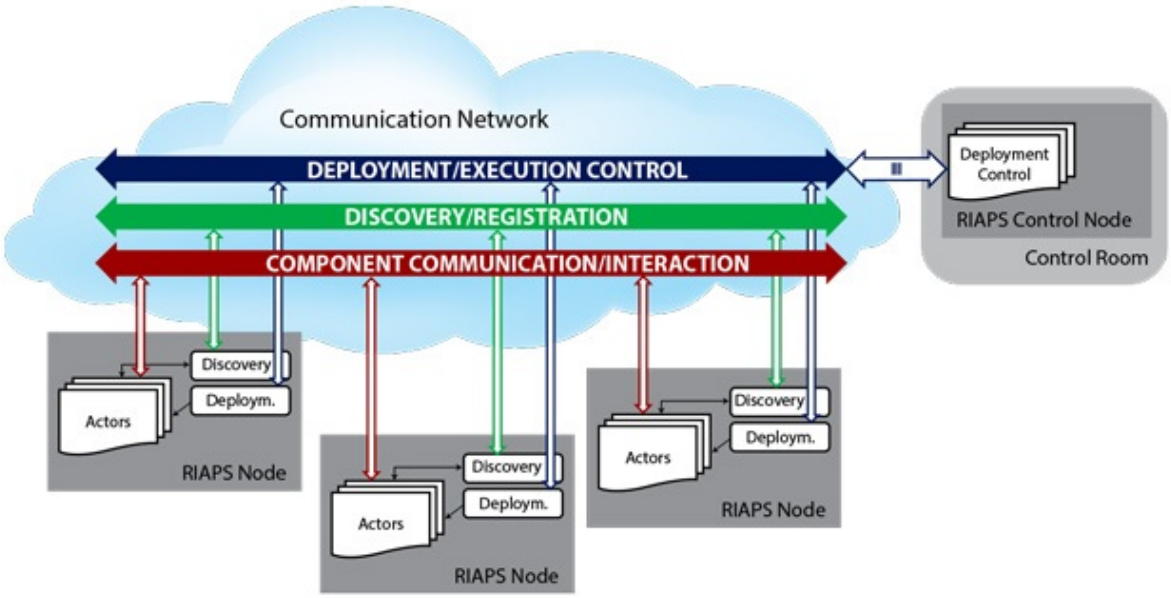


Figure 4.8: RIAPS Communication Network [42]

solely announces the provided functionality but also demands the required services. When the needed services are already in the OpenDHT, the *Discovery Service* sends a notice to the demanding RIAPS application. So then the application processes this new notice and connects to that service. Instead, if the requested service is not yet available, the *Discovery Service* will send a callback to the demanding application if it becomes available. Because of this attitude to the versatile management of the node egress and ingress, it is necessary that the *Discovery Service* find each other.
OpenDHT does not implement an API to eliminate a no longer useful service from the DHT [35], but it is possible to remove the service establishing an expiration value (10 minutes by default value). After that time the service is erased from the DHT. It is important to underline that setting due date also means that the registrations of the various application nodes must be refreshed periodically by the *Discovery Service* if the applications are still working. Consequently, when a service stops answering, but its latest registration has not expired yet, it is not eliminated from the DHT.

4.4.3 Fault Tolerance

As it is mentioned, the *Discovery Service* play another critical role within the RIAPS platform, the fault tolerance. It is responsible for refreshing the registration of application services inside OpenDHT. But, before refreshing, the *Discovery Service* have to verify that the RIAPS component function to be renovated is still running and accessible. This signifies that the *Discovery Service* have to handle the situation when an application service suddenly leaves the batch, i.e., it is disconnected from the network without sending any notification message.
Another capability of the *Discovery Service* in terms of fault tolerance is that it provides the exclusivity of the information in the sense that each service data is coupled with the unique actor process ID (PID). Particularly, when an application component registers a service, in the *Discovery Service* is also registered the PID of the principal actor with a time-stamp. The list of PIDs/services combinations is checked periodically by verifying if the actor of the associated PID is still running. If there is a fault and the process is arrested, the *Discovery Service* removes the PID/service pair from the openDHT and will not renovate the registration of the application actor.
The *Discovery Service* could stop surprisingly, so the application components have to be resilient. In fact if there is some fault to the *Discovery Service*, the actor, and the components continue to run, but unfortunately, they cannot acquire information about other services, and until the *Discovery Service* is restarted, the new actor cannot be registered, and so it cannot enter the network.
The components do not perform any discovery control algorithm, but thanks to the actor that manages the components contained within it, provides to those components a stable link with the *Discovery Service*. The approach of the actors monitoring the health of the *Discovery Service* is the equivalent of the *Discovery Service* that monitors components. In fact, the actors know the *Discovery Service* PID and keep the time-stamp. When the *Discovery Service* PID is no longer in the running processes list, the actor starts a process of rebooting. Rebooting signify that the actor registers the RIAPS running services again, but this time in a completely new *Discovery Service* instance, and asks again for the needed services. So the actor detects the eventual absence of the *Discovery Service* and reacts reconnecting the components to a new *Discovery Service*, which re-register the component.

4.5 Component interactions

The communication network provided by RIAPS allows node-node (component-component) and node-external device interaction and communication. In this section, it is explained how the various RIAPS applications components interact one each other.

RIAPS applications are formed from interacting components hosted in actors that run on different nodes of the network. The benefit of encasing many components inside one actor is that the communication's cost existing among components in one actor is remarkably smaller than the cost of communication between actors implemented on the same node. Moreover, the interaction's cost among actors placed on separate nodes is even more expensive, because the information has to travel within a complicated protocol stack and through a probably busier network.

4.5.1 Component Communication Patterns

The components can have various communication patterns that define how the messages are managed. The message in RIAPS environment is considered as structured information exchanged within RIAPS target nodes.
The various possible communication patterns are:

- Subscribe/publish model: is an asynchronous communication paradigm, made up of three actors: publisher, subscribe and message broker. The publishers provide messages of particular topics that are delivered by a broker to subscribers to that topic. Publishers and subscribers are unknown, removing the dependencies of information between clients. Examples of this communication pattern are ZeroMQ and MQTT.
- Request/reply model: is constituted by a client which request for some data and by a server which has to reply to the request. The communication paradigm is asynchronous in the sense that the sender does not explicitly wait for a response when the reply comes the sender component is activated again. However, a new request cannot be sent until the response to a previous request has been acquired. In other words, the sent request and the received response must progress hand in hand.
- Client/server model: equal to the request/reply pattern except for the type of communication paradigm that is synchronous: the client must

explicitly wait till the reply message has not arrived. As for the request/replay, the request sent by the client and the server response must progress in lockstep.

- Query/answer model: identical to the request/reply model, except for the lockstep rule that is not required, i.e. an arbitrary amount of requests can also be sent without receiving a direct response.

All component interactions in RIAPS applications must be executed using these models. Hence if it is required to create an interaction between two components or nodes (see Fig. 4.9, first of all, it is needed to choose a port to use for a specific component related to the models listed above. Than ports select a message type, and when an application is deployed, the message types represent the services requested or provided by the corresponding port [43], whilst the routing of the message is provided by the RIAPS *Discovery Service*.

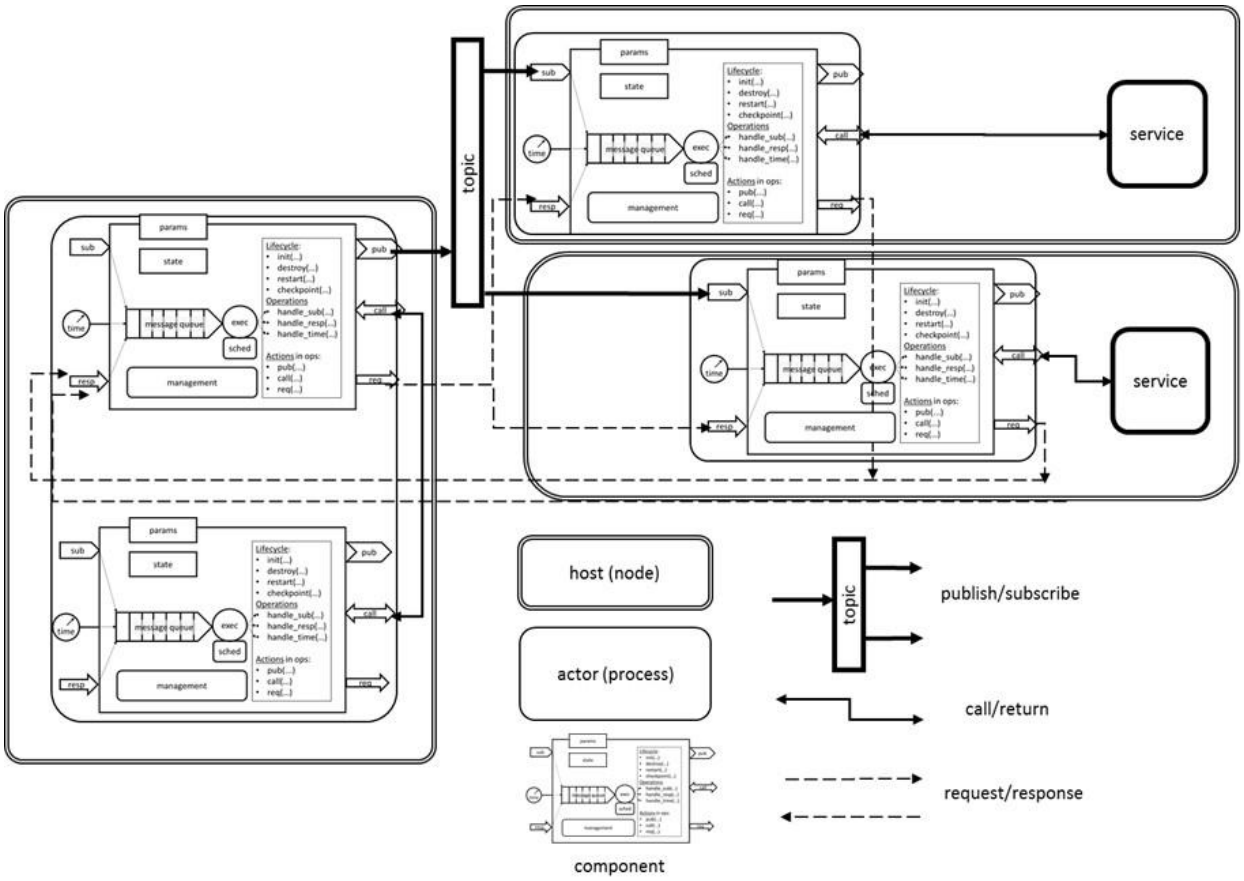


Figure 4.9: Example of a possible RIAPS component interactions [43]

4.6 External Device Interactions

When application actors needed to retrieve some information like command actuation or data measured by a real sensor or power system device, it is required to link the external device with the RIAPS applications.
These links could change over time as systems are upgraded, new devices could be added or removed or moved to other nodes to reach the requirements of the specific area where they are placed. Therefore, RIAPS applications never communicate directly with the physical devices; instead, they communicate via the *Device Interface Service* [45], that provide the *Device Components* which are contained inside the *Device Communication Actors* (DCA).
In other words, the *Device Interface Service* virtualizes the physical devices into RIAPS *Device Components*, allowing the latter to communicate with the application components using the internal communication patterns explained before (see Figure 4.10).
The *Device Component* provided by RIAPS to overcome this kind of situations is a variant of the application components previously described. But, unlike the standard component, a *Device Component* has two particularity:

1. It could be "multi-threaded," i.e., like the standard component, it possesses one "executor thread" that executes the component functions. However, unlike the standard component, the *Device Component* can start one or more threads that run simultaneously with the main thread and can communicate with I/O devices.

2. It has a special port named the *Inside Port* for interacting between the threads; it is useful for sending and receiving messages from the component threads.

4.6.1 How It Works

Each unique combination of a device I/O connections (DIOCs) and software protocol is a *Device Connection Point* (DCP). The physical I/O interface can support different industrial standards such as RS-485, RS-232, TCP/IP on Ethernet, GPIO, and I2C. Moreover, RIAPS *Device Interface Service* implements specific application programming interfaces (APIs) needed to establish direct communication between the physical device and *Device Components* to maintain the real-time data access, such as Modbus, DNP3 and IEEE

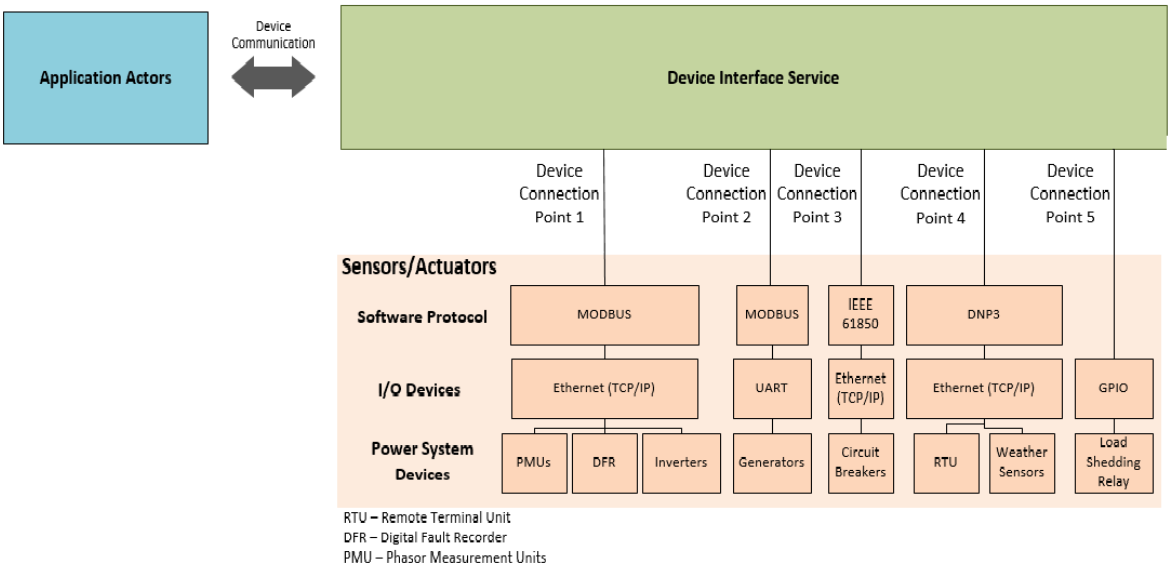


Figure 4.10: How *Device Interface Service* works

61850, and IEEE C37.118.2 synchrophasor data transfer protocol [38]. Various power devices can be materially connected to the same physical I/O hardware interface, for example, an Ethernet link working with a Modbus protocol can communicate with a Phasor Measurement Unit (PMU) (as is shown in Fig. 4.10), and a relay connected to a GPIO pin can sense the voltage of a branch of the grid [40][45].
Therefore, the job of the *Device Component* is to build a "bridge" between the application actors and physical devices, so it has to translate to and from the proper physical and software communication protocol for the application actors and, if required, add time-stamps to the external device data, based on the global time given by the *Time Synchronization Service*. For periodical functions or programmed outputs, each *Device Component* handles the real-time scheduling needs for the devices linked with their specific *Device Connection Point* by demanding global time data and programming sleep times when required.
To achieve these goals, the DCAs are created for each DCP available inside the RIAPS target nodes and provided with all the necessary resource, drivers, arbitration methods, and a real-time scheduler for timed control operations. When the developers want to add a new physical device to the RIAPS platform, they first must describe how the distinct hardware node is configured

creating the *Device Configuration Metadata* (DCM) that contain: what types of physical devices are connected to this nod and how they are configured. Using the DCM, each *Device Communication Actor* will register itself with the device management actor to notify it which physical devices are accessible through its DCP. Upon registration, each *Device Communication Actor* is initialized on request.
Moreover, the developers have to deploy within each application actor the *Device Interface Metadata* (DIM), a piece of data useful to recognize the particular physical hardware that the application will utilize. Making the assumption that a RIAPS app is created and developed considering the target power system device, once deployed it must connect to distinct instances of those physical devices [45]. This link between the virtual and the physical devices is implemented in the DIM and, it is a portion of the configuration files deployed together with the application. It is important to underline that the implementation of the DIOCs is not contained inside of the application, but they lie in the DCAs, and the application components are connected to those DIOCs at run-time.
When an Application actor sends to the Discovery Service a connection request to establish a bond with a power system device, and the requested device is available, the Discovery Service can start to provide to the application actor the needed connection data so that it can begin to communicate directly with the proper DCA. The building of the "bridges" between the DIOCs of the DCAs and the application components are promoted through the *RIAPS Broker Service* a service of the Discovery Manager application contained in the already examined *Platform Manager* (see sec. 4.3.2).
Any DCA can detect the hardware errors and allows the *Discovery Service* to check the condition of the device connections. When a physical device is no longer available for example because of a hardware malfunction or failure, the *Discovery Service* notifies the associated application actor that the device has disconnected.
Moreover, when no actor remains connected with the device because no longer needed, the *Discovery Service* will demand to the DCA to stop all related communication with that device.

4.6.2 External Device Interaction Patterns

As it was done for internal communication, a list of the RIAPS interaction patterns between application actors and physical device is explained [45] :

- Periodic input: When a physical sensor can periodically provide sensed

data than the *Device Interface Service* activate the stream of information through the correct application components. However, these data are first collected by the *Device Interface Services* that time-stamps it (with global time from the Time Synchronization Service) and then distribute this data to the interested applications

- Sporadic input: When the I/O sensor device produces a new sample, it is time-stamped, and this sample is transferred to the interested application through published messages. Another possibility is that the data is saved in a queue of samples or as an individual sample and the application will ask this from the service.

- Sporadic output: An applications components demands the device interface service to produce an output for an actuator. The transfer takes places as quickly as feasible. The demander application might or might not bide for the transfer to finish. In the second case, the application actor must register a request to get the notification regarding the conclusion of the transfer in an asynchronous way.

- Scheduled output: An application requires to execute actuation commands at a precise physical time (or at a periodical scheduled precise moment). The application component calls the device interface service to program the actuation action and produces the information to be transferred. The service schedules this request, that is executed only when comes the physical time.

- Periodic output: When an application needs to build a periodic actuator output activity, it has to provide a starting value to send. As a consequence, the service created produces a cyclic thread that first reads the value to be sent from a sample position, and then delivers that value to the actuator, repeating this with a definite frequency. The application can asynchronously refresh the sample position with a new value used as a starting point for the next transfer.

Chapter 5

Methods

In this chapter, all the methods and tools used in this project are analyzed to explain better how the work was carried out and how it was possible to obtain the wanted results.
Starting with a list and description of the fundamental components of this thesis work, useful to provide an overview of the methods chosen. Then an explanation of some RIAPS used commands is present. Concluding with the examination of the developed codes.

5.1 Outline and Description of the Utilized Tools

The are various components that ware used to achieve the expected result. To better organize the section, it was preferred to subdivide the analysis of the tools based on the role token in each of the three phases of the project: code development, simulation, and testing phase.

5.1.1 Development Phase

The starting point of the project is the development phase, the stage in which the software was built. This section is limited to list and describe only the various tools utilized during the building of the project, so an accurate analysis of the developed codes can be found in section 5.2). The created codes are launched in the RIAPS environment, which represents the principal and fundamental tool utilized not only for this phase but for all /the project. This

revolutionary middleware, already explained in detail in the chapter 4 leads us to discover another tool used for the design: GNU/Linux (in particular Ubuntu 18.04), the operating system in which RIAPS lies. In fact, all the development stage was conducted inside this open-source operating system that thanks to the use of the middleware mentioned above, it was possible to develop the wanted codes.
The developed applications are implemented in Python 3.7 and C++, the only two programming languages allowed in RIAPS and supported by the Gnu Compiler collection 7 (GCC 7).

MQTT

One fundamental concept of this thesis is the communication. Indeed, it is needed to distinguish two different communication interfaces, the interaction between devices (target nodes) running in the RIAPS network and communication between all the target nodes and the command system (SCADA system or a control node). The first use ZeroMQ as communication paradigm, for the latter instead is used MQTT.
They are booth asynchronous communication paradigm, based on the publisher-subscriber messaging patterns, however as the ZeroMQ is a RIAPS predefined communication architecture, in which all the libraries are already integrated on the middleware. MQTT for RIAPS instead is an external communication protocol appropriately chosen for this project and depends on the Eclipse Paho Python Client library, so it needs to install the additional library Paho-MQTT. Another difference is that ZeroMQ does not use a central server; each Client could be a server just using the library [51]. Instead, MQTT exploits a broker that is a central server which the task to route all the message coming from all the connected Client. In particular, for this work was used first a free testing broker like iot.eclipse.org and test.mosquitto.org, and once the application is tested, it was used a private broker created using Eclipse Mosquitto, in this way the communication structure is more reliable and safe.
The MQTT communication architecture is made up of publisher/subscriber (clients) and broker (server). The data are contained inside a Topic which publishes it to the broker, it has a buffer to save the data and acts like a repository that routes the message to the subscriber that subscribed for that specific topic (see fig 5.1).
One important parameter present in the MQTT communication pattern is the Quality of Service (QoS), which value 0,1 and 2 make a different type of

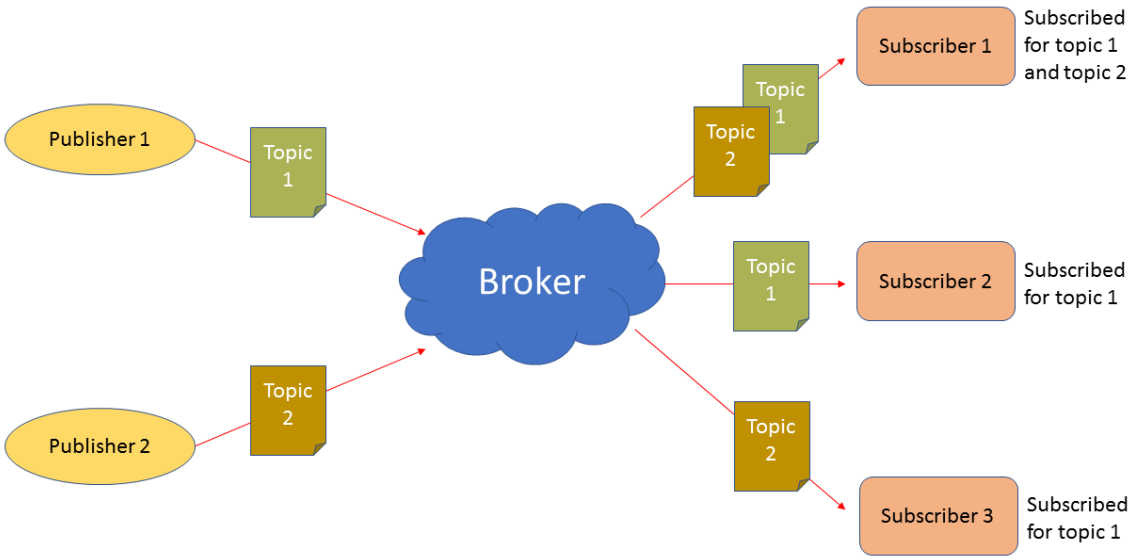


Figure 5.1: MQTT communication overview

service and can be chosen for each client. For example, QoS = 0 means that we do not need to know if the information was sent to the message could be sent once or never; QoS = 1 means instead the message could be delivered once or more time; QoS = 2 means that the message is delivered exactly once. For this project, the QoS of each client was set to 2 to avoid a lack of information or duplicate message. The choice of MQTT as a messaging pattern was made because it is a lightweight communication protocol, and can be considered the machine to machine IoT interaction protocol for excellence, so perfect for the Smart Grid system. Other characteristics make MQTT the most suitable communication pattern for this project such as the simple interoperability with a wide range of devices, above all the presence of the SCADA protocol, allows a SCADA system to send and receive data from all the device connected to the broker. Another important skill is the minimization network bandwidth thanks to the very lightweight overhead, that allows the application in several embedded device with low power consumption.

5.1.2 Simulation Phase

Once developed the software, the second step is the simulation phase. In this stage, the working capability and functionality are tested using simulated hardware that replaces the real devices, to avoid the possible hazard of malfunction caused by faults in the code.

BeagleBone Black

IT is exactly this the phase in which it is started to use the BeagleBone Black Rev C (BBB) (see Figure 5.2). It is an embedded computer board running Linux as default, simple to connect on the internet through Ethernet cable o WiFi, and also able to interact with other devices through cable connection like USB, GPIO, and UART. The BBBs have the essential task to represent the real hardware devices in the RIAPS network, using a flashed microSD card containing the RIAPS image it was possible to install the RIAPS middleware inside each BBBs. So because of the real implementation of this project provides the simultaneous use of eight devices, in this simulation step have been used eight BeagleBone Black. The choice of BBB as a development platform was forced by the RIAPS developer that created the image only for that computer board. Therefore the BBB booted in RIAPS environment is able to be automatically recognized by all the other target and control nodes of RIAPS, so the control node can detect its presence, communicate with it and deploy the wanted piece of code (more detail in section 5.2).

Communication Simulation Tools

In particular, the goals of this phase is to test the two communication patterns:

- The first is Modbus, a serial communication protocol made up of Master, who need information, and Slave who provide data which are stored in registers. This communication pattern instead allows the exchange of information between the DSP (digital signal processor, explained in 5.2.2) of the hardware device attached in the network, and the BBBs used to interact with the external environment.
- The second is MQTT already explained in the previous page; it is the protocol used to create an interaction between the BBB and the control station or any device connected on the internet. In this phase, the

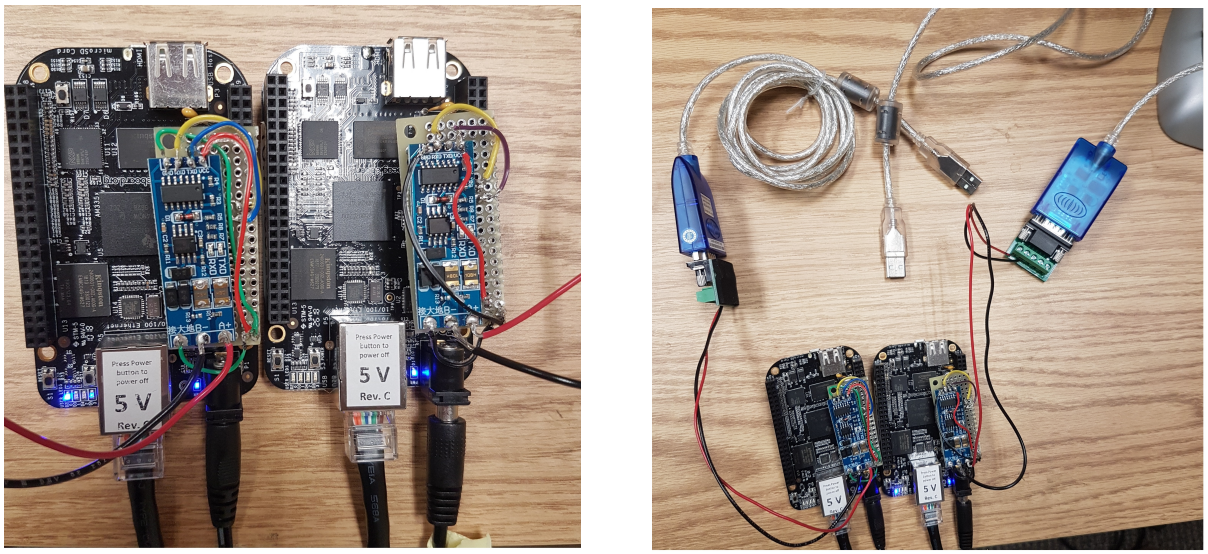


Figure 5.2: Tools For Simulation

MQTT communication of eight BBBs running contemporary in RIAPS environment have been tested.

To simulate the working scenario of the first protocol, it is used the Modbus Slave simulator *Modbus Poll*, an important software that allows to set up the registers to provide data to the Modbus master represented by the RIAPS node, in which all the library is already present in the middleware itself. More detail on the Modbus master code will be provided in section 5.2.2.
For what concerns the MQTT part, it is utilized the application MQTT.FX, an MQTT client simulator based on JavaFX, and thanks to the possibility to send and receive data through MQTT broker, it can interact with all the BBBs, which in turn have the *Eclipse paho-mqtt* library that allows them to exploit the MQTT protocol.
Because of the Modbus slave simulator and the BBB should be linked for Modbus communication to start to interact. *Modbus Poll* is a simulation software installed in a computer exploit the USB port to interact with the external devices. Instead, in this project, it was used BBB UART ports to communicate through Modbus, these are perfectly suitable for this kind of protocol because they are divided in UART TX (transmission port to send messages) and UART RX (reception port to receive messages). So to link the USB port of the computer and the UARTs port of the BBB it

Figure 5.3: Testing phase on GEH testbed

was needed to apply two tools able to create the wanted connection, the UART to Modbus converter: SMAKN® SCM TTL to RS485 Adapter 485 to Serial Port UART Level Converter Module 3.3V 5V; and the Modbus to USB converter: Gearmo Pro 5ft USB to RS485-RS422 FTDI Chip.
To simulate the complete communication environment (from Modbus Slave to MQTT client) have been used two BBBs equipped with the converter tools. It would have been expensive as well as useless to test the complete interaction pattern for all the eight BBBs. The final built wanted connection is shown in Figure 5.2, it was connected the 3.3V power port, the GRD, the UART TX and RX port with the corresponding Modbus converter side port of the SMAKN® SCM TTL. In the second conversion step the Modbus-USB

converter TX and RX ports are connected respectively with the A+ and B- port of SMAKN® SCM TTL.

5.1.3 Testing and Validation Phase

After the simulation phase, the last step is the test with real hardware in which the developed code and the settled BBBs are deployed in the GEH testbed connected to the grid voltage to demonstrate the working capability of this project. That test environment already explained in 3.3.1 try to emulate a Smart Grid system made up of three microgrids each one formed by one LVSST DC load and AC loads. In Figure 5.3 it is possible to see all the testbed of this project: the first microgrid, that one with LVSST1 has one AC and one DC load, instead, the second with LVSST2 has only one DC load, and third with LVSST3 for the moment is without any load. Moreover, to monitor and check the system operation two oscilloscopes are used and connected to two monitors to visualize the details better.

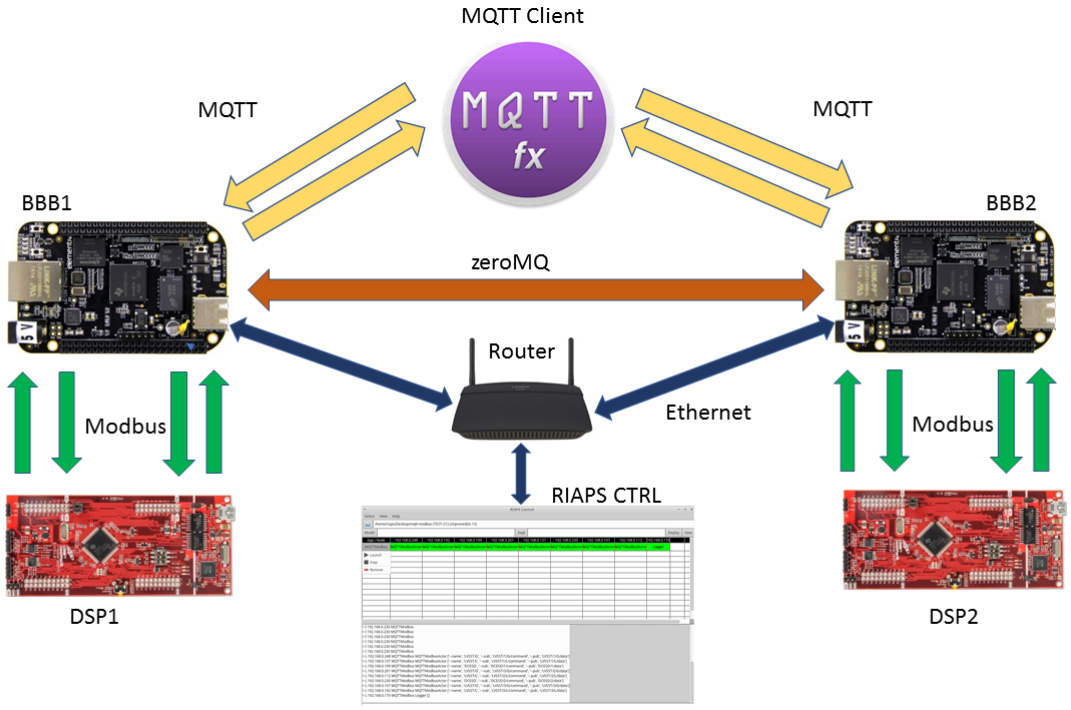


Figure 5.4: New Communication Architecture in the GEH testbed

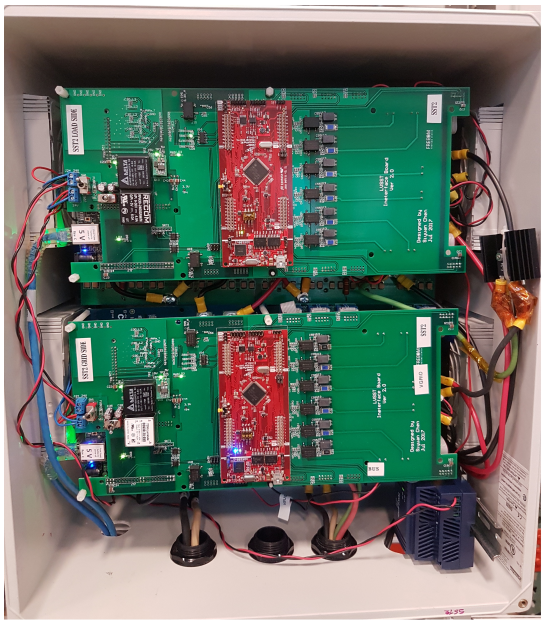


Figure 5.5: LVSST module

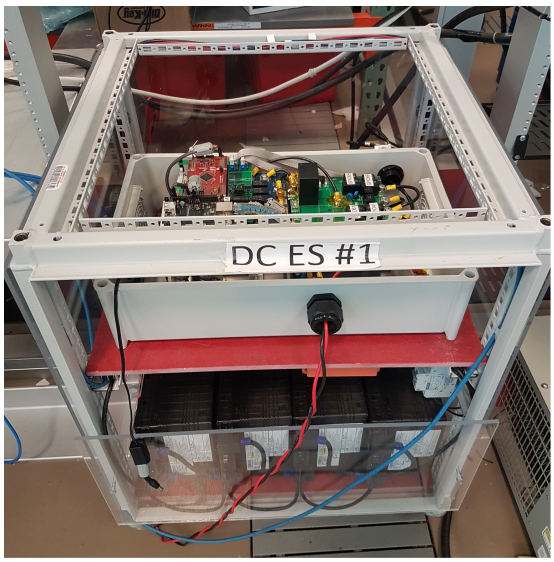


Figure 5.6: DCDESD module

LVSST

The core of each microgrid is the LVSST (see Fig. 5.5) which is made up of two sides: the Grid side and the Load side. Both have the interfacing board and containing all the converter explained in 3.2, but the first deal with the voltage coming from the grid and the second has the task to route that voltage to the right load. Each side provided with one BBB with RIAPS microSD image to allow communications and control (is quite difficult to see in 5.5 because hidden behind the first layer); and one TI F28377S digital signal processor (DSP), the red board in the Fig. 5.5). The latter has the critical task to receive and perform all the command received from the BBB and contain the LVSST algorithm based on the functionality explained in chapter 3.2.1.

So the DSP can be seen as a mediator between the BBB (and so the user that send and receive data through MQTT) and the actual hardware,Figure 5 show the complete communication architecture between the explained tools. Another important component of utilized during the test phase is the *DC Distributed Energy Storage Device*

DCDESD and AC/DC load

Another important component of utilized during the test phase is the *DC Distributed Energy Storage Device* (DCDESD) as already mentioned in 3.3.1 it constitutes the DC load module.
Each DCDESD is made up of five Toshiba model 20Ah 2P12S module, high-reliability rechargeable lithium-Ion Battery each one with a capacity of 40 Ah a DC voltage of 27.6V and a power rate of 1104Wh for an overall capacity of 200 Ah. The DCDESD that is also possible to see in Fig. 5.6 is not only made up of batteries but like the LVSST module, it is provided with a BBB, DSP and DC/DC converter.
Finally, as can be seen in the lower right corner of Fig. 5.3 an Avtron model k490 AC load is used to simulate all the common home AC loads. For the test, the AC load is kept to 250 W and sometimes increased until 2 kW to test the reliability of the system with varying loads (more details are provided in section 6.1.4). Instead the DC load is represented by a resistor bank with a resistance set to 270Ωwhich could be connected to all the microgrids. Other electric and electronic devices and detail present in each module but are not examined because they go beyond the goal of this thesis work.

5.2 Codes Explanations

After an overview of the field in which this project has been developed, now an accurate analysis of the implementation of the RIAPS apps and the codes built is needed. Therefore in this paragraph first take place a brief description of RIAPS implementation and then some critical aspects of the codes are underlined.

5.2.1 RIAPS Implementation

After all the theoretical insight of the chapter 4, now it is shown a practical demonstration of the RIAPS power, also describing how to launch applications in this extraordinary Smart Grid platform.
A RIAPS application is made up of several elements such as run-time libraries, configuration file, and all the codes to apply on the *Target Nodes*. As already explained, one of the strengths of the middleware mentioned above is the capability to systematically pack and deploy all those files on the wanted *RIAPS Target Nodes* which are automatically added in the *RIAPS CTRL* app thanks the use of the *RIAPS Deploy* service contained on each node. All this is possible by using the *RIAPS CTRL* app, which is manually launched in the *Control Node*, and those nodes that receive the algorithms are now called *Actor Nodes*, each one with the predetermined role.
Figure 5.7 show the *RIAPS CTRL* app, the command window accessible for only the *Control Nodes* of the system. It has the fundamental task to manage all the *Target Nodes* attached to the platform; in particular, it allows us to start, stop, or remove the file packages on the *Target Nodes*. That brilliant app is also very useful to have an overview of the controlled system providing the IP address of the nodes and the type of algorithm deployed on it. Moreover, it provides also the possibility to obtain a graphics view of the just built interaction network, see Appendix A in which an example of Linux dot file shows the created *RIAPS Target Nodes* interaction (for this example is possible to see only three *Control Nodes* instead of the nine of the real project for order and clarity reason).
Another important capability of *RIAPS CTRL* is the possibility to monitor all the systems thanks to the user interface in which it is possible to visualize the eventual failure or bugs in the network.

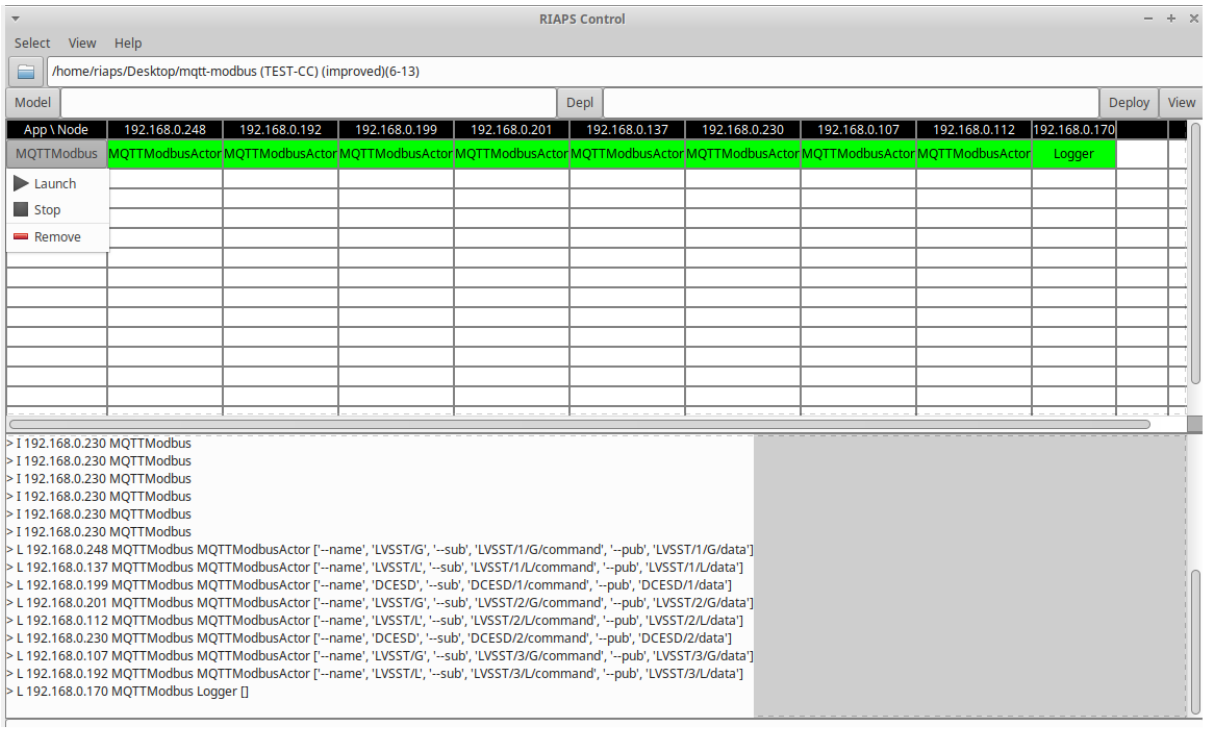


Figure 5.7: *RIAPS CTRL* app

5.2.2 Codes Developed

The *RIAPS CTRL* app so far explained, it is underlined the ability to download the developed codes directly inside the wanted elements. Now it necessary to talk about those piece of codes that are deployed, describing their functions.
Beside the paho-MQTT library, Modbus library, and the RIAPS configuration files, the other elements packed and downloaded into the nodes are the developed codes. They are divided into six different files, each one with a distinct fundamental role:

- *MQTTModbus.riaps*
- *MQTTModbus.deplo*
- *MQTT.py*
- *ComputationalComponent.py*
- *ModbusUartReqRepDevice.py*

- *Logger.py*

In Figure 5.7 is it possible to see two blank boxes, "Model" and "Depl," should be filled respectively with the *MQTTModbus.riaps* file and *MQTTModbus.deplo* file.

RIAPS Model

The .riaps file written in C++, present in Appendix B.1, took the role of the main file in which all the piece of codes are handled. In particular, it contains all the file and function calls, the *RIAPS Actor* which dynamically loads and launches *RIAPS Components*, and the *RIAPS Device* that is a particular variant of *RIAPS Actor* which has only *Device Component*. The *RIAPS Actor* of the *MQTTModbus.riaps* are the *MQTTModbusActor* and the *Logger*. The first holds all the files that provide the communications between DSP and BBB and between BBB and MQTT client, but also contain the algorithms to deploy on the BBBs (in particular eight BBBs obtain this Actor as shown in Fig 5.7).

ComputationalComponent

Included in the *MQTTModbusActor*, there is the *RIAPS Component* that play the fundamental roles of algorithms container for the various device and above all to acts as a bridge between the DSP and the MQTT Client, see Figure 5.8 for an overview of the codes interactions in which the ComputationalComponent play the central role.
As it is possible to see in Appendix B.4, the *ComputationalComponent.py* file contains the initialization of the input registers (the registers those contain the data retrieved from the actual Hardware) and the holding register (those that contain the command value) of the DPS, more detail about the command and data send and received are provided in chapter 6.1. Instead to in Appendix C is possible to see the MQTT.fx windows for publish the commands (written inside a python dictionary) and for subscribe and receive the messages from the distributed smart devices.
The *ComputationalComponent.py* file also includes the "clock" method that takes care of the Modbus status request and to the Modbus request for receiving data sent to the *ModbusUartReqRepDevice.py* every eight seconds (the time clock is set inside the RIAPS model). Another fundamental method is the "RecCommand" to handle the messages received (commands) from

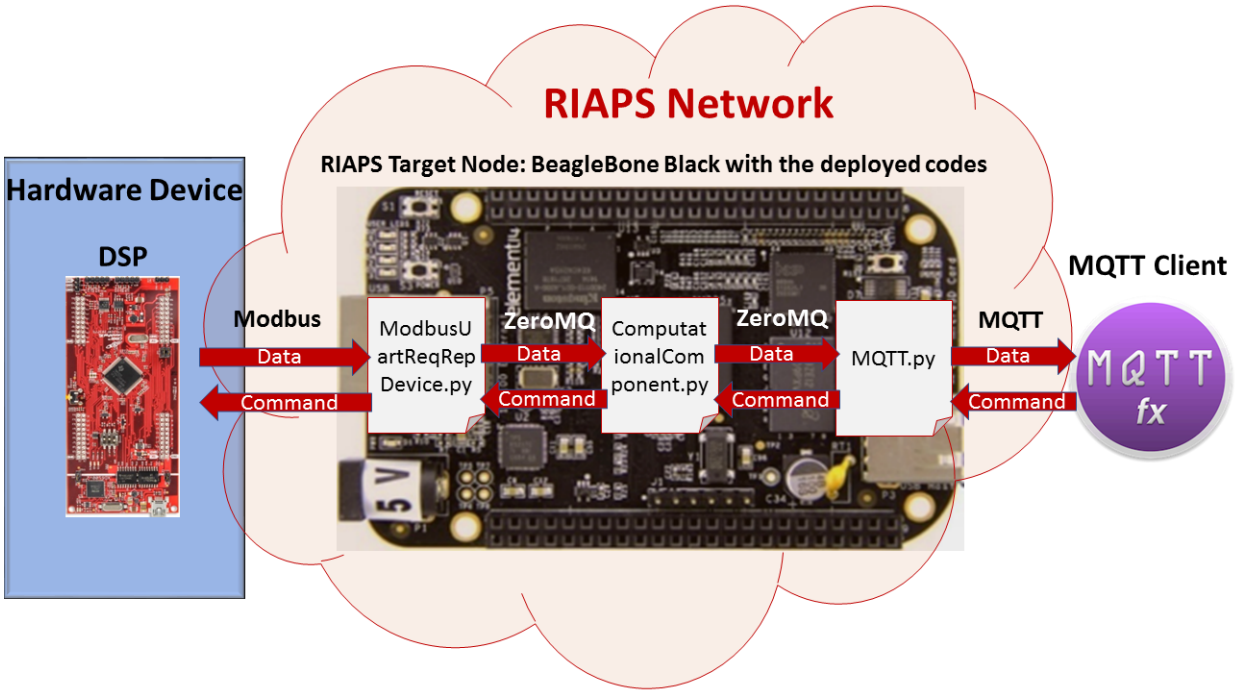


Figure 5.8: Interaction overview of a single *RIAPS Target Node*

MQTT.py, it is first converted from analogue to digital and than in base on the type of command a Modbus request is sent through ZeroMQ to change the value of the wanted holding registers of the DSP. The "Conversions" method instead converts the digital data coming from the input register of the DSP to analog human-readable data and packed to a Python dictionary. After the conversion phase, the dictionary goes to the "modbusCommandReqPort" method in which it is sent to the *MQTT.py* through ZeroMQ. It has been chosen as a dictionary to simplify the MQTT message and allow a JSON conversion.

Logger

Instead, the *Logger* actor has the role in taking a register of all the event and message exchanged during the run phase of the software, so it is a handy tool to monitor the execution of the codes and also to provide a simple way for debugging. The *Logger.py* file contained in the *Logger* Actor, showed in Appendix B.6, is downloaded in only one *Target Node*. Usually, it is the same control machine that in this case, is both Control and Target Node.

MQTT Device and Code

Regarding the *RIAPS Devices*, contained inside the *MQTTModbus.riaps*, one of these is *MQTT Device* which contain initialization of the used communication ports, timer clock of the loop functions, references and fundamental parameters for its Python code which needs the address of the chosen broker the QoS and the topic for which publish and subscribe. Now focusing on the Python code also included in the *MQTTModbusActor*, *MQTT.py*, which contain all the standard functions for an MQTT communication that can be seen in Appendix B.3.
The Python method "pubPort" has the important role in handling the message that comes from the DSP, containing the data of the devices, and after storing them in a variable has the important task to publish them in MQTT style (by topic).
Instead, the method "incoming" manage the commands that comes from the MQTT client, subscribing for them and once received, send them through ZeroMQ to the DSP passing from the *ComputationalComponent.py* file. For a better overview of the communication architecture, see Fig. 5.8 and Appendix A in which are shown all the interaction between nodes.

ModbusUartReqRepDevice Device and Code

The other *RIAPS Devices* present in the RIAPS model is the *ModbusReqRepDevice* that represents the Modbus-UART communication lines. In fact, inside this RIAPS function, there is the initialization of the two port utilized; one used for test the communication between the two devices and to notice the status ("modbusStatusRepPort"); the second is the real port utilized for the exchange of data ("modbusCommandRepPort"). Moreover, it contains also all the parameters useful for the *ModbusUartReqRepDevice.py* file, in particular, the slave address and baud rate (Modbus slave parameters) and the BBB UART port used, in this case, the first (UART1).
Once received those parameters the Python code (see Appendix B.5) which include the Modbus Library needed for the communication, answer to all the request received by the *ComputationalComponent.py*, such as status information command execution and need of data from DSP.

RIAPS Deplo

The .deplo is a proper RIAPS file written in C++. As we can see from Appendix B.2, It has the critical role to route the various *RIAPS Actors*

in the wanted devices attached in the network, and represented by their a priori fixed IP address. It is also possible to choose to work only with certain node instead of all together. Moreover, it provides the possibility to the developer to change and choose three important parameters for the *MQTTModbusActor*:

- The name of the actual device for which it will work, very useful for applying the right conversations and algorithm inside the *ComputationalComponent.py*.
- The topic for which the *MQTT.py* has to subscribe, in this case, the commands that come from the MQTT Client.
- The topic for which the *MQTT.py* has to publish, in this case, the data that comes from the DSP.

Chapter 6

Results and Conclusion

So far, they have been described all the components and the methods used to accomplish this thesis work, passing through the overall project architecture explanation. Now it is fundamental to analyze all the results and the outcomes of the project that thanks to the GEH testbed was possible to obtain. Concluding the thesis with a discussion of what has been achieved and an analysis of the possible future works.

6.1 Test Results

Before showing the obtained results, it is needed to explain how they are grouped and organized. The test outcomes are separated on the base of the working principles. Moreover, they will be shown two types of result: that one that comes out from the oscilloscope for graphic monitoring of the system; and the outcomes retrieved directly from the DSP attached on the various components of the GEH testbed. These last results which provided through the MQTT client (see appendix C in which an example of data subscribed by the client is present) are beneficial for a double check of the entire application because they should reflect the oscilloscope ones.

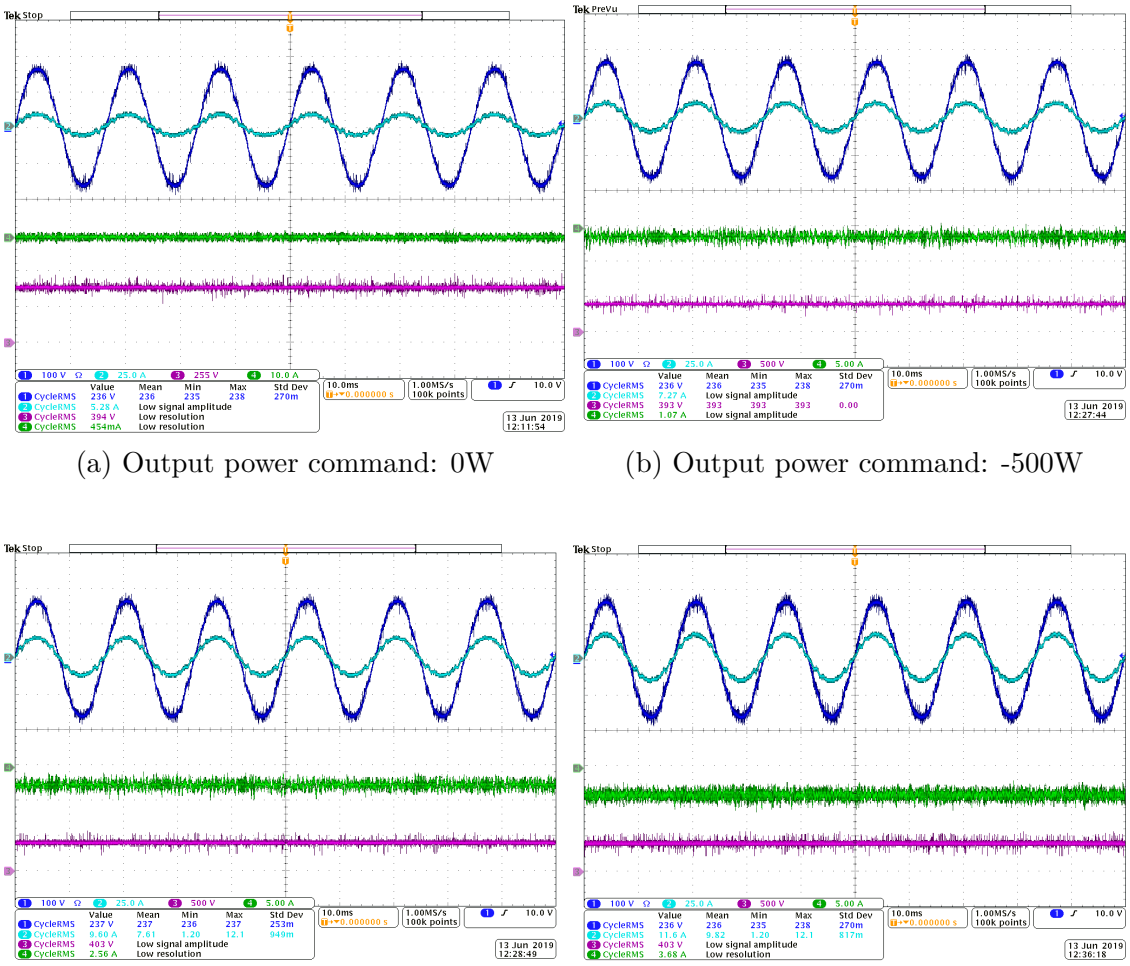
Premises

It is essential to define the four different oscilloscope signals as a premise valid for all the figures that will show in this chapter: Blue = AC Grid Voltage, Azure = Grid Current, Violet = DCDESD Voltage, Green = DCDESD Current. All the noise present in the outputs of the oscilloscope and which are

also propagated in the data present in all the tables is due to the high harmonic effect coming from the main grid. So for these reason we will see some small variation in data that during some change they should be constant, or even value of the table that doesn't match the theoretical ones. During the test phase, all the working capability of the GEH was checked, but for simplicity, the shown results are referred only to one single microgrid made up of LVSST1, DCDESD1, AC, and DC load, also because the results should have been exactly the same for all the three microgrids. For all the duration of the test it was used an AC load of 500W (unless otherwise indicated) and an DC load of around 500W represented by a resistance of 270Ωand a voltage of 380V, for a total power needed for the loads of around 1000W

6.1.1 Grid-Tied Mode, DCDESD Charging

In this phase of the test, the goal is to charge the batteries of the DCDESD blocks exploiting the grid current. This is an example of the power of the LVSST that can simply and smoothly converted the AC current to a DC current for the five batteries. Moreover, as it is possible to see from Fig. 6.1, the LVSST and all the supervisory control and data acquisition system developed, give the wanted results at varying of the power flow from 0W to -1500W.
To better analyze the outcomes of this phase, from Fig. 6.1a in which the DCDESD is OFF, till to Fig. 6.1d in which the control command sent is -1500W (and turn ON the DCDESD) the differences are: - It is possible to see a more and more increase of the Grid current (azure signal) because of the request of power is increasing from -500W to -1500W, so the LVSST need to absorb more current from the main grid. - Increasing of DCDESD current (Green signal), in this case, the oscilloscope automatically put these signal positive, but actually, it is negative because the commands inserted are negative output power (charging) that means that the direction of the power flow is incoming to the DCDESD.
Instead, from table 6.1 it is possible to see all the outcomes regarding the DCDESD coming directly from the DSP and visualized in the MQTT client (see Appendix C to for an example of MQTT received data). As can be seen, These values reflect those of the oscilloscope except for some small differences due to the high data fluctuation. Instead for the Battery Power it is refers to the simple multiplication between battery current and voltage ($P = VI$). Table 6.1 demonstrate also that the battery are charging, in fact, as it is possible to notice the SoC that start from 30% when the DCDESD is OFF,



(a) Output power command: 0W

(b) Output power command: -500W

(c) Output power command: -1000W

(d) Output power command: -1500W

Figure 6.1: DCDESD charging phases with variable output power flows

goes to 32% after after a few minutes.

6.1.2 Islanded Mode, DCDESD Discharging

the objectives of this testing phase are multiples:

1. the goal to use the battery together with the Grid current to answer the need of the power of the microgrid, in this way the LVSST inject grid current to fill the lack of power of the DCDESD, which in turn provide a limited power chosen by the operator.

2. to try to use the DCDESD as a unique power source for the microgrid

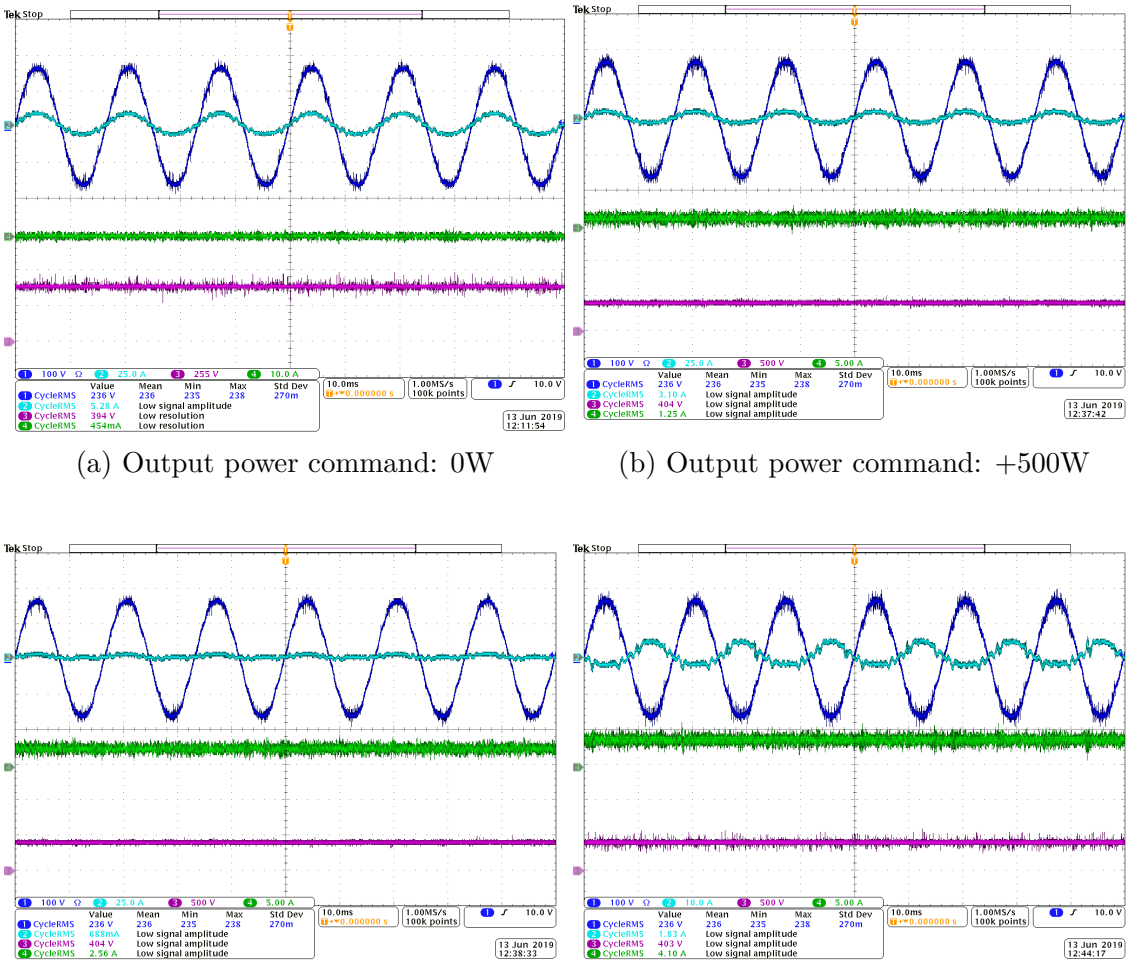
Command	Battery Power[W]	DC Voltage[V]	Battery Current[A]	SoC[%]
OFF	0	0	0	30
-500W	-394	388	-1.19	30
-1000W	-764	387	-2.11	32
-1500W	-1132	386	3	32

Table 6.1: DCDESD data from DSP during the charging phase with variable flows

sustaining alone the DC and AC load power request; in this way the grid current will be zero, providing the possibility to detached that subsystem from the main grid so activating the Islanded mode.

3. To check the LVSST capability to feed the grid when the DCDESD provide more power than needed for the microgrid; this is the perfect example of bi-directional power flow skills of the LVSST which can give power both to the grid and to loads of its subsystem.

The Figures 6.2 show the oscilloscope outputs of this testing stage in which it is possible to notice the achievement of the objectives set.
The Fig. 6.2b testify the collaboration of LVSST (Grid power) and the DCDESD power; it is possible to see also a decrease of the Grid current and augment of the DCDESD current
Fig 6.2c shows the obtaining of the Islanded mode; in fact, the Grid current is flat (no current absorption from the Grid) because of the power requested by DC and AC load together is 1000W. Moreover, it is also evident an increase of DCDESD current due to the increase of the power flow
Instead, In Fig.6.2d is possible to see the Grid current that now is out of phase with the Grid Voltage, this demonstrates the achievement of the last goal, i.e., the batteries are feeding the Grid with their energy surplus produced. Again the grid current has resulted higher than the previous step.
Moreover, like the previous phase also here the analyzed outcomes of the oscilloscope match the result coming from the DSPs shown in table 6.2. So In this case, it is possible to notice an expected reduction of the SoC of the batteries starting from 32% to 30%, caused to the outgoing energy flow.



(a) Output power command: 0W

(b) Output power command: +500W

(c) Output power command: +1000W

(d) Output power command: +1500W

Figure 6.2: DCDESD discharging phases with variable output power flows

6.1.3 Reactive Power Injection

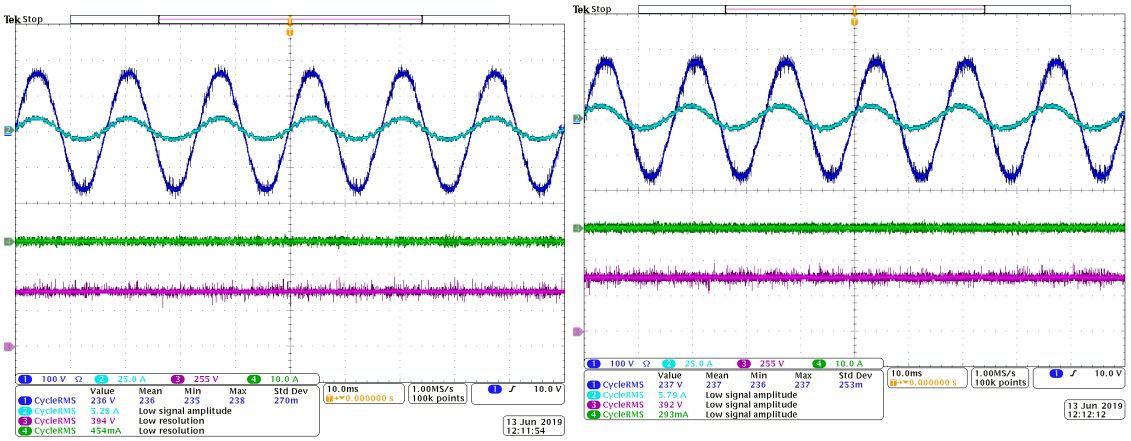
Concerning the reactive power injection phase, the objective is to try to modify the amount of reactive power in the microgrid. Unlike the previous cases, the commands are sent to the LVSST grid side BBB, because it is attached to the portion of LVSST block that is physically linked with the main grid. That test is essential to provide a demonstration of the reactive power compensation of the GEH testbed that can be very useful for a future installation of the photovoltaic panels or wind turbines in which is necessitating a stabilization of the point of common coupling voltage.
The oscilloscope results of Fig. 6.3 shown that as expected, the only change

Command	Battery Power[W]	DC Voltage[V]	Battery Current[A]	SoC[%]
OFF	0	0	0	32
+500W	386	389	1.09	32
+1000W	781	390	2.26	32
+1500W	1154	386	3.07	30

Table 6.2: DCDESD data from DSP during the discharging phase with variable flows

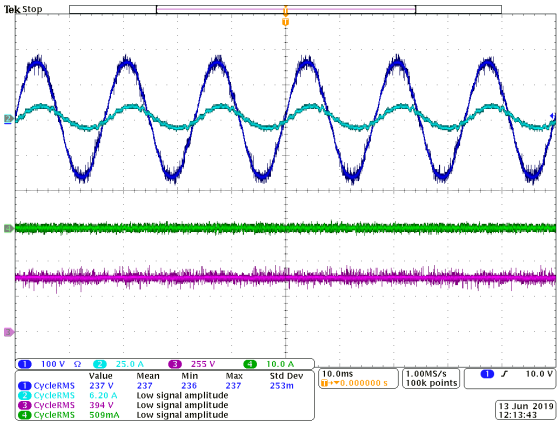
is on the Grid current phase, whereas all the other value remains unchanged, demonstrating the possibility to inject and compensate reactive power in the system. In particular, Fig. 6.3b represent an inductive system in which the current leads the voltage, so we are injecting reactive energy on the microgrid (following the used sign convention). Instead, Fig. 6.3b in which the command is -500 Var show the current lags the voltage like in a capacitive circuit. In table 6.3 is possible to see the outcomes coming from the DSPs of the LVSST grid and load side. In this case, the data retrieved are:

- AC side active power that refers to the active power absorbed from the grid to compensate the load;
- AC side reactive power which values correspond to the amount of reactive power present in the subsystem;
- AC RMS voltage, corresponding to the Grid Voltage;
- DC side power, which is the power the LVSST provide to the DC port;
- AC Voltage of port1 and 2, because the architecture of the AC side of the LVSST was divided into two part, and their values are always the same as is seen in the table;
- AC Active Power of port 1 and 2, in this case, the value are a bit different, but their sum corresponds to the magnitude of AC load that was chosen (500W).



(a) Output reactive power command: 0Var

(b) Output reactive power command: +500Var



(c) Output reactive power command: -500Var

Figure 6.3: LVSST reactive power compensation phase

As is possible to notice the reactive power commands match the respective reactive power output (see table 6.3 LVSST Grid side), and so testify the correct working principle of the test of this stage. Moreover, the data coming from that table which regards the LVSST load side result unchanged during that all the testing phase; and this demonstrate that the changing of reactive power doesn't not corrupt any operation of the loads.

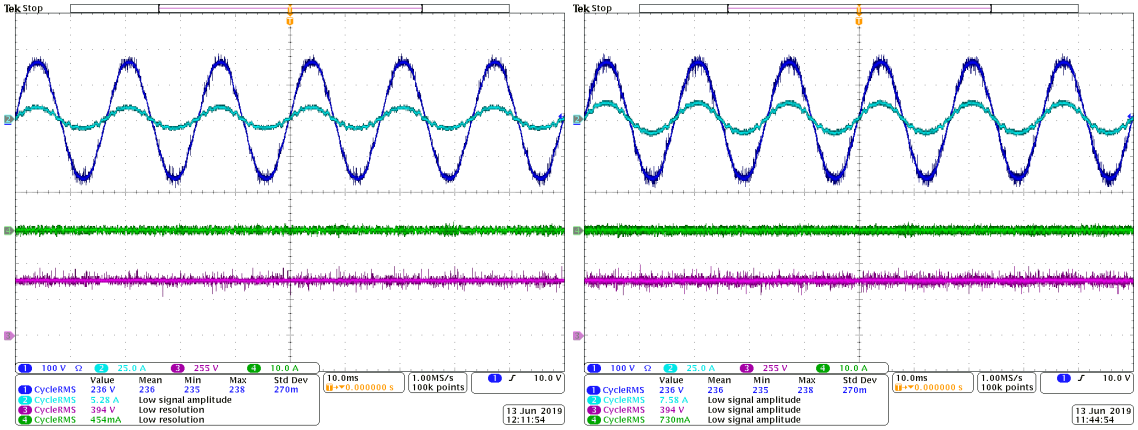
6.1.4 Variable Loads Test

The last test regards the possibility to vary the AC loads connected to the microgrid in order to testify the grid capability of a to bear a sudden change

LVSST Grid side				
Command	AC Pwr[W]	AC Pwr[Var]	RMS volt[V]	DC Pwr[W]
0Var	1044	0	233	364
-500Var	1056	520	235	366
+500Var	1036	-474	233	370
LVSST Load side				
Command	AC 1 Volt[V]	AC 1 Pwr[W]	AC 2 Volt[V]	AC 2 Pwr[W]
0Var	120	383	120	274
+500Var	120	384	120	275
-500Var	120	381	121	274

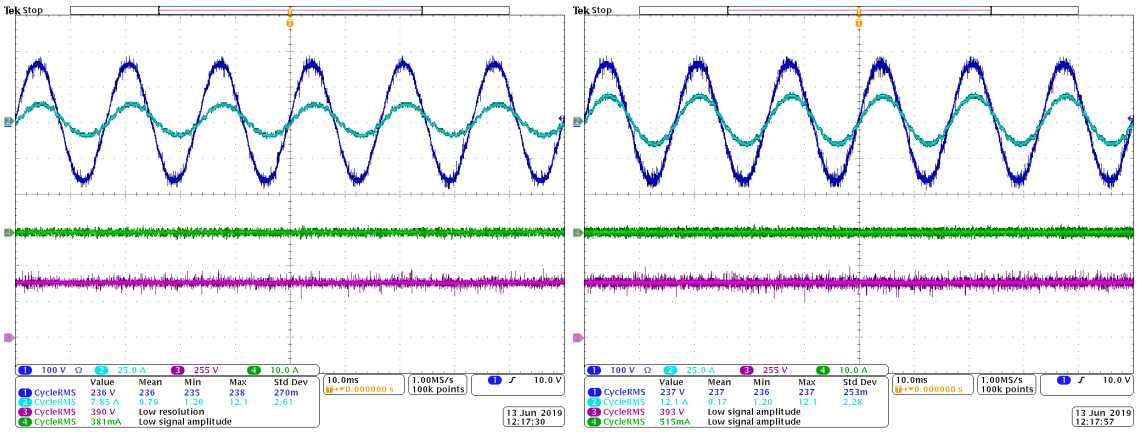
Table 6.3: LVSST grid and load side, data from DSP during the reactive power ejection phase

on the power demands.
Figure 6.4b shows the system with an augmented load of 1000W, so as expected the Grid current increase and as seen in Table 6.4 the AC side active power now is increased because have to feed the augmented AC power of the port 1 and 2 following the equation:
$ACactivePower = DCpower(DCload+batteries)+AC1Power+AC2Power$
In this case, the DCDESD is OFF, so the DC power is equal to the DC load that is around 500W. So was demonstrated that the microgrid system is able to work and react properly to a change of load.
In Fig. 6.4c and 6.4d are shown the oscilloscope outcomes of respectively a reactive power injection command of -500Var with an AC load of 1000W and reactive power injection command of -500Var with an AC load of 2000W. In both the case, the LVSST can satisfy the active power requirements and the reactive power commands.



(a) AC load: 500W

(b) AC load: 1000W



(c) AC load: 1000, Output power command: -500Var

(d) AC load: 2000, Output power command: -500Var

Figure 6.4: AC loads varying and AC reactive power injection

LVSST Grid side				
Command/load	AC Pwr[W]	AC Pwr[Var]	RMS volt[V]	DC Pwr
0Var/500W	1044	0	233	364
0Var/1000W	1593	0	235	369
-500Var/1000W	1597	-498	236	366
-500Var/2000W	2603	520	235	367
LVSST Load side				
Command/load	AC 1 Volt[V]	AC 1 Pwr[W]	AC 2 Volt[V]	AC 2 Pwr[W]
0Var/500W	120	383	120	274
0Var/1000W	119	562	119	534
-500Var/1000W	120	561	121	534
-500Var/2000W	120	1003	120	1000

Table 6.4: LVSST Grid and load side, data from DSP for loads varying and reactive power injection

6.2 Conclusion

The results obtained testify the complete achievement of the set objectives defined by this thesis work. It resulted in being the first real working example of a Smart Grid working on RIAPS, so this is a significant step forward for this platform to reach the consensus of the power institutions to apply it to the real world outside the university environment. Thanks to this project that exploits the GEH testbed which working processes reflect perfectly that one of a real Smart Grid System that now there are effective results about RIAPS very close to its target range. Therefore now the resulting system obtains all the RIAPS benefit such as reliability, security, simple control, and monitoring system, multitasking, distribution of intelligence, time synchronization, simple communications, IoT application. However, the most critical achievement obtained by this thesis results is the awareness of having provided a real and working model to follow for the development of the future Smart Grids. Therefore, it was addressed one of the issues of the diffusion of this innovative power grid, now is provided a novel standard in terms of a platform able to host real Smart system.
Moreover, the obtained results demonstrate also that the developed supervisory control and data acquisition system architecture,that is based on The combination of MQTT and Modbus communication protocols, work properly. In fact, the MQTT retrieved messages match the data coming from the Oscilloscope. Also, the control capability test of the project was successful, as demonstrated by the expected changes of the characteristics of the signals shown by the various oscilloscope outputs. With this new communications system now the operator can control and interact with the distributed smart devices in a simple way, and without opening several command windows as was done with the old communications paradigm of the GEH testbed. So Now it is also possible to plug and play as many as wanted devices without any limits and re-configuring the system, now the system is scalable and flexible if there is some failure with a device it can be easily removed from the system without crashing the entire system. The developed project also provides security and privacy, utterly absent in the former communication framework. Now all the connected devices enjoy the privilege of a LAN suitable for microgrids application instead of the NCSU network.

6.3 Future Works

The next step toward standardization could be the GUI improvement; in particular, the integration of a LabView could be the right way to improve the SCADA system intuition and simplicity. This National Instrument product has already the configuration for the MQTT protocol, so it could be very simple to integrate it into this project. For example, a LabView button could be applied to substitute the sending of a command actually done by MQTT.fx; or for example, it is possible to implement some graph or digital display to monitor the work of the devices.
Thanks to the developed project can be easier to integrate and deploy new pieces of algorithms; a possible future work can be therefore the development or the update of control algorithms to improve the working capability of the microgrids.
Another future work can be the upgrade of the actual cabled internet connection of the distributed devices, shifting to the WiFi, to obtain more flexibility and an extended range. This is the right solution to bring this project to the homes of all the cities.

Bibliography

[1] Wikipedia. *Sea$_l$evel$_r$ise*. Last accessed 10 July 2019. 2019.

[2] Vox. *global-warming-2-degrees-climate-change*. Last accessed 10 July 2019. 2018. URL: https://www.vox.com/energy-and-environment/2018/1/19/16908402/global-warming-2-degrees-climate-change.

[3] Clark W Gellings. "Then and now: The perspective of the man who coined the term 'DSM'". In: *Energy Policy* 24.4 (1996), pp. 285–288.

[4] Clark W Gellings and William M Smith. "Integrating demand-side management into utility planning". In: *Proceedings of the IEEE* 77.6 (1989), pp. 908–918.

[5] Jan Bruinenberg, L Colton, E Darmois, J Dorn, J Doyle, O Elloumi, H Englert, R Forbes, J Heiles, P Hermans, et al. "CEN-CENELEC-ETSI smart grid co-ordination group smart grid reference architecture". In: *CEN, CENELEC, ETSI, Tech. Rep* (2012), pp. 98–107.

[6] Dahai Han, Jie Zhang, Yongjun Zhang, and Wanyi Gu. "Convergence of sensor networks/internet of things and Power Grid Information Network at aggregation layer". In: *2010 International Conference on Power System Technology*. Oct. 2010, pp. 1–6. DOI: 10.1109/POWERCON.2010.5666553.

[7] I. L. Klavsuts, D. A. Klavsuts, G. L. Rusin, and I. S. Mezhov. "Perfecting business processes in electricity grids by the use of innovative technology of demand side management in the framework of the general conception of smart grids". In: *2014 49th International Universities Power Engineering Conference (UPEC)*. Sept. 2014, pp. 1–4. DOI: 10.1109/UPEC.2014.6934690.

[8] S. Chren. "Towards Multi-layered Reliability Analysis in Smart Grids". In: *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Oct. 2017, pp. 116–119. DOI: 10.1109/ISSREW.2017.67.

[9] A. Stefanov and C. Liu. "Cyber-power system security in a smart grid environment". In: *2012 IEEE PES Innovative Smart Grid Technologies (ISGT)*. Jan. 2012, pp. 1–3. DOI: 10.1109/ISGT.2012.6175560.

[10] "IEEE Vision for Smart Grid Controls: 2030 and Beyond Reference Model". In: *IEEE Vision for Smart Grid Control: 2030 and Beyond Reference Model* (Sept. 2013), pp. 1–10. DOI: 10.1109/IEEESTD.2013.6598993.

[11] H. Gözde, M. C. Taplamacıoğlu, M. Arı, and H. Shalaf. "4G/LTE technology for smart grid communication infrastructure". In: *2015 3rd International Istanbul Smart Grid Congress and Fair (ICSG)*. Apr. 2015, pp. 1–4. DOI: 10.1109/SGCF.2015.7354914.

[12] Murat Kuzlu, Manisa Pipattanasomporn, and Saifur Rahman. "Communication network requirements for major smart grid applications in HAN, NAN and WAN". In: *Computer Networks* 67 (2014), pp. 74–88.

[13] Wenye Wang, Yi Xu, and Mohit Khanna. "A survey on the communication architectures in smart grid". In: *Computer networks* 55.15 (2011), pp. 3604–3629.

[14] T. Kazičková and B. Buhnova. "ICT architecture for the Smart Grid: Concept overview". In: *2016 Smart Cities Symposium Prague (SCSP)*. May 2016, pp. 1–4. DOI: 10.1109/SCSP.2016.7501035.

[15] Jixuan Zheng, David Wenzhong Gao, and Li Lin. "Smart meters in smart grid: An overview". In: *2013 IEEE Green Technologies Conference (GreenTech)*. IEEE. 2013, pp. 57–64.

[16] B. Karimi, V. Namboodiri, and M. Jadliwala. "Scalable Meter Data Collection in Smart Grids Through Message Concatenation". In: *IEEE Transactions on Smart Grid* 6.4 (July 2015), pp. 1697–1706. ISSN: 1949-3053. DOI: 10.1109/TSG.2015.2426020.

[17] N. H. Tran, D. H. Tran, S. Ren, Z. Han, E. Huh, and C. S. Hong. "How Geo-Distributed Data Centers Do Demand Response: A Game-Theoretic Approach". In: *IEEE Transactions on Smart Grid* 7.2 (Mar. 2016), pp. 937–947. ISSN: 1949-3053. DOI: 10.1109/TSG.2015.2421286.

[18] J. W. Simpson-Porco, Q. Shafiee, F. Dörfler, J. C. Vasquez, J. M. Guerrero, and F. Bullo. "Secondary Frequency and Voltage Control of Islanded Microgrids via Distributed Averaging". In: *IEEE Transactions on Industrial Electronics* 62.11 (Nov. 2015), pp. 7025–7038. ISSN: 0278-0046. DOI: 10.1109/TIE.2015.2436879.

[19] H. Yu, H. Tu, and S. Lukic. "A passivity-based decentralized control strategy for current-controlled inverters in AC microgrids". In: *2018 IEEE Applied Power Electronics Conference and Exposition (APEC)*. Mar. 2018, pp. 1399–1406. DOI: 10.1109/APEC.2018.8341200.

[20] Y. Du, H. Tu, S. Lukic, D. Lubkeman, A. Dubey, and G. Karsai. "Development of a Controller Hardware-in-the-Loop Platform for Microgrid Distributed Control Applications". In: *2018 IEEE Electronic Power Grid (eGrid)*. Nov. 2018, pp. 1–6. DOI: 10.1109/eGRID.2018.8598696.

[21] D. Shi, X. Chen, Z. Wang, X. Zhang, Z. Yu, X. Wang, and D. Bian. "A Distributed Cooperative Control Framework for Synchronized Reconnection of a Multi-Bus Microgrid". In: *IEEE Transactions on Smart Grid* 9.6 (Nov. 2018), pp. 6646–6655. ISSN: 1949-3053. DOI: 10.1109/TSG.2017.2717806.

[22] L. Meng, T. Dragicevic, J. C. Vasquez, and J. M. Guerrero. "Tertiary and Secondary Control Levels for Efficiency Optimization and System Damping in Droop Controlled DC–DC Converters". In: *IEEE Transactions on Smart Grid* 6.6 (Nov. 2015), pp. 2615–2626. ISSN: 1949-3053. DOI: 10.1109/TSG.2015.2435055.

[23] A. Q. Huang, M. L. Crow, G. T. Heydt, J. P. Zheng, and S. J. Dale. "The Future Renewable Electric Energy Delivery and Management (FREEDM) System: The Energy Internet". In: *Proceedings of the IEEE* 99.1 (Jan. 2011), pp. 133–148. ISSN: 0018-9219. DOI: 10.1109/JPROC.2010.2081330.

[24] M. T. A. Khan, A. A. Milani, A. Chakrabortty, and I. Husain. "Dynamic Modeling and Feasibility Analysis of a Solid-State Transformer-Based Power Distribution System". In: *IEEE Transactions on Industry Applications* 54.1 (Jan. 2018), pp. 551–562. ISSN: 0093-9994. DOI: 10.1109/TIA.2017.2757450.

[25] Parviz Ghoddousi, Ehsan Eshtehardian, Shirin Jooybanpour, and Ashtad Javanmardi. "Multi-mode resource-constrained discrete time–cost-resource optimization in project scheduling using non-dominated sorting genetic algorithm". In: *Automation in construction* 30 (2013), pp. 216–227.

[26] J. M. Carrasco, L. G. Franquelo, J. T. Bialasiewicz, E. Galvan, R. C. PortilloGuisado, M. A. M. Prats, J. I. Leon, and N. Moreno-Alfonso. "Power-Electronic Systems for the Grid Integration of Renewable Energy Sources: A Survey". In: *IEEE Transactions on Industrial Electronics* 53.4 (June 2006), pp. 1002–1016. ISSN: 0278-0046. DOI: 10.1109/TIE.2006.878356.

[27] X. She, A. Q. Huang, and R. Burgos. "Review of Solid-State Transformer Technologies and Their Application in Power Distribution Systems". In: *IEEE Journal of Emerging and Selected Topics in Power Electronics* 1.3 (Sept. 2013), pp. 186–198. ISSN: 2168-6777. DOI: 10.1109/JESTPE.2013.2277917.

[28] Technology Review. *10-emerging-technologies-of-2011*. Last accessed 11 May 2019. 2012. URL: https://www.technologyreview.com/s/423776/10-emerging-technologies-of-2011/.

[29] X. She, X. Yu, F. Wang, and A. Q. Huang. "Design and Demonstration of a 3.6-kV–120-V/10-kVA Solid-State Transformer for Smart Grid Application". In: *IEEE Transactions on Power Electronics* 29.8 (Aug. 2014), pp. 3982–3996. ISSN: 0885-8993. DOI: 10.1109/TPEL.2013.2293471.

[30] A. A. Milani, M. T. A. Khan, A. Chakrabortty, and I. Husain. "Equilibrium Point Analysis and Power Sharing Methods for Distribution Systems Driven by Solid-State Transformers". In: *IEEE Transactions on Power Systems* 33.2 (Mar. 2018), pp. 1473–1483. ISSN: 0885-8950. DOI: 10.1109/TPWRS.2017.2720540.

[31] X. She, S. Lukic, A. Q. Huang, S. Bhattacharya, and M. Baran. "Performance evaluation of solid state transformer based microgrid in FREEDM systems". In: *2011 Twenty-Sixth Annual IEEE Applied Power Electronics Conference and Exposition (APEC)*. Mar. 2011, pp. 182–188. DOI: 10.1109/APEC.2011.5744594.

[32] S. Bhattacharya, T. Zhao, G. Wang, S. Dutta, S. Baek, Y. Du, B. Parkhideh, X. Zhou, and A. Q. Huang. "Design and development of Generation-I silicon based Solid State Transformer". In: *2010 Twenty-Fifth Annual IEEE Applied Power Electronics Conference and Exposition (APEC)*. Feb. 2010, pp. 1666–1673. DOI: 10.1109/APEC.2010.5433455.

[33] D. Dujic, C. Zhao, A. Mester, J. K. Steinke, M. Weiss, S. Lewdeni-Schmid, T. Chaudhuri, and P. Stefanutti. "Power Electronic Traction Transformer-Low Voltage Prototype". In: *IEEE Transactions on Power Electronics* 28.12 (Dec. 2013), pp. 5522–5534. ISSN: 0885-8993. DOI: 10.1109/TPEL.2013.2248756.

[34] D Peeples. *The next big thing? EPRI's fast, flexible (and cheaper) EV charging system*. Last accessed 15 May 2019. 2018. URL: https://www.ttnews.com/articles/strong-growth-expected-ev-charging-stations.

[35] S. Eisele, I. Mardari, A. Dubey, and G. Karsai. "RIAPS: Resilient Information Architecture Platform for Decentralized Smart Systems". In: *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*. May 2017, pp. 125–132. DOI: 10.1109/ISORC.2017.22.

[36] Dale Willis, Arkodeb Dasgupta, and Suman Banerjee. "ParaDrop: a multi-tenant platform to dynamically install third party services on wireless gateways". In: *Proceedings of the 9th ACM workshop on Mobility in the evolving internet architecture*. ACM. 2014, pp. 43–48.

[37] Kyle Benson, Charles Fracchia, Guoxi Wang, Qiuxi Zhu, Serene Almomen, John Cohn, Luke D'arcy, Daniel Hoffman, Matthew Makai, Julien Stamatakis, et al. "SCALE: Safe community awareness and alerting leveraging the internet of things". In: *IEEE Communications Magazine* 53.12 (2015), pp. 27–34.

[38] Y. Du, H. Tu, S. Lukic, A. Dubey, and G. Karsai. "Distributed Microgrid Synchronization Strategy Using a Novel Information Architecture Platform". In: *2018 IEEE Energy Conversion Congress and Exposition (ECCE)*. Sept. 2018, pp. 2060–2066. DOI: 10.1109/ECCE.2018.8557695.

[39] Y. Du, H. Tu, S. Lukic, D. Lubkeman, A. Dubey, and G. Karsai. "Resilient Information Architecture Platform for Smart Systems (RIAPS): Case Study for Distributed Apparent Power Control". In: *2018 IEEE/PES Transmission and Distribution Conference and Exposition (T D)*. Apr. 2018, pp. 1–5. DOI: 10.1109/TDC.2018.8440324.

[40] Y. Du, H. Tu, S. Lukic, D. Lubkeman, A. Dubey, and G. Karsai. "Implementation of a distributed microgrid controller on the Resilient Information Architecture Platform for Smart Systems (RIAPS)". In: *2017 North American Power Symposium (NAPS)*. Sept. 2017, pp. 1–6. DOI: 10.1109/NAPS.2017.8107305.

[41] Vanderbilt University. *RIAPS overview*. Last accessed 2 June 2019. 2019. URL: https://riaps.isis.vanderbilt.edu/.

[42] Github. *What is RIAPS*. Last accessed 24 May 2019. 2018. URL: https://riaps.github.io.

[43] Github. *RIAPS Architecture Overview*. Last accessed 27 May 2019. 2018. URL: https://riaps.github.io/arch.html.

[44] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys Tutorials* 17.4 (Fourthquarter 2015), pp. 2347–2376. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2444095.

[45] A. Dubey, G. Karsai, P. Volgyesi, M. Metelko, I. Madari, H. Tu, Y. Du, and S. Lukic. "Device Access Abstractions for Resilient Information Architecture Platform for Smart Grid". In: *IEEE Embedded Systems Letters* (2018), pp. 1–1. ISSN: 1943-0663. DOI: 10.1109/LES.2018.2845854.

[46] P. Volgyesi, A. Dubey, T. Krentz, I. Madari, M. Metelko, and G. Karsai. "Time Synchronization Services for Low-Cost Fog Computing Applications". In: *2017 International Symposium on Rapid System Prototyping (RSP)*. Oct. 2017, pp. 57–63.

[47] REDIS. *What is REDIS*. Last accessed 24 May 2019. 2018. URL: https://redis.io/topics/faq.

[48] github. *What is REDIS*. Last accessed 25 May 2019. 2019. URL: https://github.com/savoirfairelinux/opendht.

[49] NetMQ. *UDP beacon*. Last accessed 25 May 2019. 2019. URL: https://netmq.readthedocs.io/en/latest/beacon/.

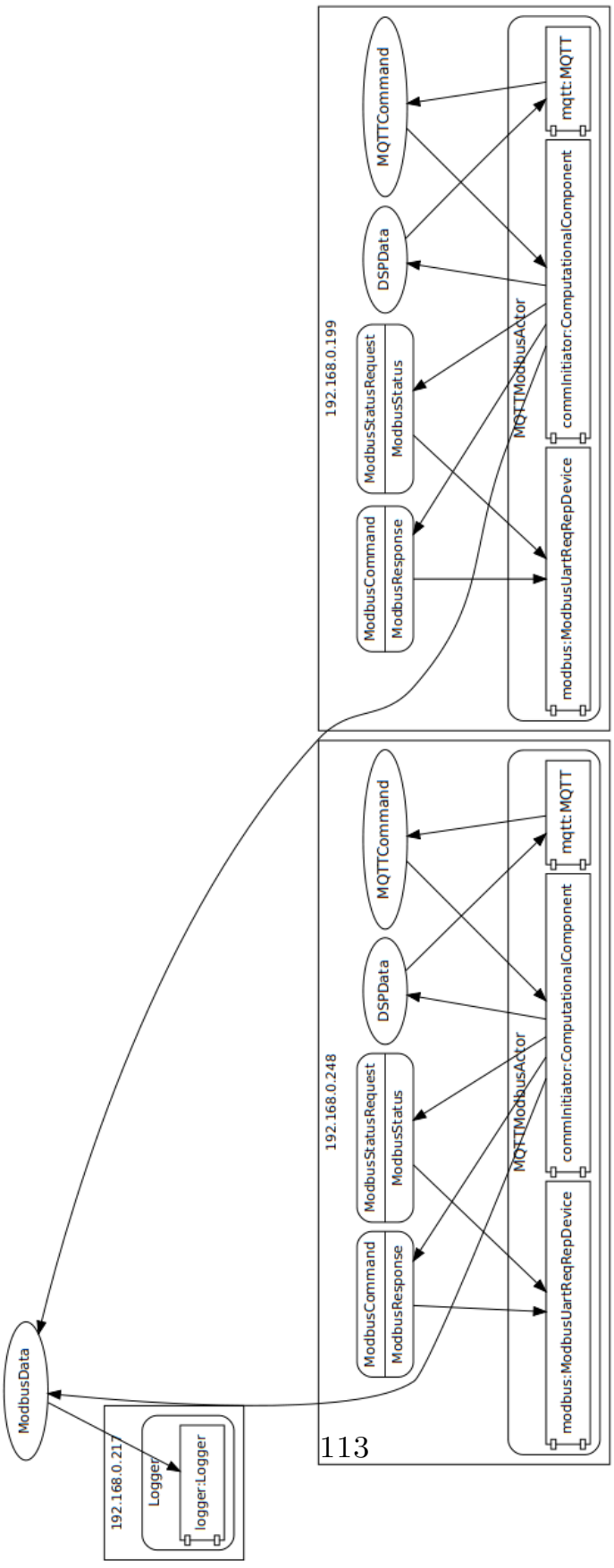
[50] Wikipedia. *UDP*. Last accessed 22 May 2019. 2019. URL: https://en.wikipedia.org/wiki/User_Datagram_Protocol.

[51] eliteinformatiker. *Messaging Systems: An Overview over RabbitMQ, Kafka, ZeroMQ and Mosquitto - Part 1*. Last accessed 16 June 2019. 2017. URL: https://eliteinformatiker.de/2017/11/05/message-queues-brokers-overview-part-1.

Appendices

Appendix A

RIAPS Dot File


192.168.0.199
MQTTCommand
DSPData
ModbusStatusRequest
ModbusStatus
ModbusCommand
ModbusResponse
MQTTModbusActor
mqtt:MQTT
commInitiator:ComputationalComponent
modbus:ModbusUartReqRepDevice
192.168.0.248
MQTTCommand
DSPData
ModbusStatusRequest
ModbusStatus
ModbusCommand
ModbusResponse
MQTTModbusActor
mqtt:MQTT
commInitiator:ComputationalComponent
modbus:ModbusUartReqRepDevice
ModbusData
192.168.0.217
Logger
logger:Logger

Appendix B

Project Code

B.1 MQTTModbus.RIAPS file

```
/*
MQTT-Modbus

Created on May 18, 2019

@Author: Danny Crescimone
*/

app MQTTModbus

{
        message DSPData;     // Data coming from DSP
        message MQTTCommand; // commands sent by MQTT client

        message ModbusStatusRequest;  // send request to determine Modbus status
        message ModbusStatus;         // provides Modbus status (True for open/ready, False for not connected)
        message ModbusCommand;        // send Modbus action Request
        message ModbusResponse;       // get response from Modbus action
        message ModbusData;           // information for the logger to grab

        library serialModbusLib;

//-------------------------------------------------------------------------------------------------------------

        device MQTT(host="192.168.0.222", port=1883, qos=2, topic1="MQTTCommand",topic2="DSPdata")
        {
                // Define messaging ports
                pub subPort : MQTTCommand;
                sub pubPort : DSPData;
            inside _incoming;
            timer clock 1000;
        }
//-------------------------------------------------------------------------------------------------------------

        // Modbus (UART) device interface - UART1
        //     considered the server for the request/response interaction
        device ModbusUartReqRepDevice(slaveaddress=10,port='UART1',baudrate=38400,serialTimeout=1.0)
        {
                rep modbusStatusRepPort : (ModbusStatusRequest,ModbusStatus);
                rep modbusCommandRepPort : (ModbusCommand,ModbusResponse);
        }

//-------------------------------------------------------------------------------------------------------------

        // Example Component to show Modbus I/F usage
        component ComputationalComponent(dev="LVSST/G")
        {
                timer clock 8000;
                sub RecCommand: MQTTCommand;
                pub pubPort : DSPData;
                pub tx_modbusData : ModbusData;   // When data is ready
                req modbusStatusReqPort : (ModbusStatusRequest,ModbusStatus);

          // Port used to communicate with the ModbusUartDevice
                req modbusCommandReqPort : (ModbusCommand,ModbusResponse);
        }
```

```
        component Logger(db_host='127.0.0.1', db_port=8086, db_name='MOdbusIO',
                         db_user='riaps', db_password='riaps')
        {
                sub rx_modbusData : ModbusData;
        }

//-------------------------------------------------------------------------------------------------------------

        actor MQTTModbusActor(name, sub, pub)
        {       // Local message types
             local DSPData ,MQTTCommand, ModbusStatusRequest, ModbusStatus, ModbusCommand, ModbusResponse;
             {

               mqtt : MQTT(host="192.168.0.222", port=1883, qos=2, topic1=sub,topic2=pub);
               modbus : ModbusUartReqRepDevice(slaveaddress=10,port='UART1',baudrate=38400,serialTimeout=1);
               commInitiator : ComputationalComponent(dev = name);
             }
        }

//-------------------------------------------------------------------------------------------------------------

        actor Logger()
        {
          {
            logger : Logger(db_host='127.0.0.1', db_port=8086, db_name='ModbusIO',
                      db_user='riaps', db_password='riaps');
          }
        }
}
```

B.2 MQTTModbus.deplo file

```
app MQTTModbus {

//LVSST1
    on (192.168.0.248) MQTTModbusActor(name="LVSST/G", sub="LVSST/1/G/command", pub="LVSST/1/G/data");
    on (192.168.0.137) MQTTModbusActor(name="LVSST/L", sub="LVSST/1/L/command", pub="LVSST/1/L/data");
    on (192.168.0.199) MQTTModbusActor(name="DCESD"  , sub="DCESD/1/command"  , pub="DCESD/1/data");

//LVSST2
    on (192.168.0.201) MQTTModbusActor(name="LVSST/G", sub="LVSST/2/G/command", pub="LVSST/2/G/data");
    on (192.168.0.112) MQTTModbusActor(name="LVSST/L", sub="LVSST/2/L/command", pub="LVSST/2/L/data");
    on (192.168.0.230) MQTTModbusActor(name="DCESD"  , sub="DCESD/2/command"  , pub="DCESD/2/data");

// LVSST3
    on (192.168.0.107) MQTTModbusActor(name="LVSST/G", sub="LVSST/3/G/command", pub="LVSST/3/G/data");
    on (192.168.0.192) MQTTModbusActor(name="LVSST/L", sub="LVSST/3/L/command", pub="LVSST/3/L/data");

    on (192.168.0.217) Logger;
}
```

B.3 MQTT Python code

```
'''
MQTT Device Component.

Created on May 28, 2019

@Author: Danny Crescimone
'''

from riaps.run.comp import Component
import os
import logging
import json
import threading
import paho.mqtt.client as mqtt


class MQTT(Component):
    def __init__(self, host, port, qos, topic1, topic2):
        super(MQTT, self).__init__()
        self.client_id = "%s-%s" % (self.getAppName(), str(self.getLocalID()))
        self.host = host
        self.port = port
        self.qos = qos
        self.topic1 = topic1 # command
        self.topic2 = topic2 # data
        self.client = None
        self.logger.info("Init: clientId=%s, host=%s, port=%d, qos=%d, topic1=%s, topic2=%s" %
                         (self.client_id, self.host, self.port, self.qos, self.topic1, self.topic2))

    def on_connect(self, client, userdata, flags, rc):
        """MQTT connect callback handler.

        Runs in separate thread (see: loop_start()"""
        self.logger.info("Subscribe to %s" % (self.topic1,))
        client.subscribe(self.topic1) # subscribe only for the command

    def on_message(self, client, userdata, msg):
        """MQTT message callback handler.

        Runs in separate thread (see: loop_start()"""
        if self._incoming_plug is None:
            mqtt_thread = threading.current_thread()
            self.logger.info("Creating inside plug in thread %s" % (mqtt_thread.getName(),))
            self._incoming_plug = self._incoming.setupPlug(mqtt_thread)
        self._incoming_plug.send_pyobj(msg.payload)

    def start(self):
        """Start the device component, connect to broker and subscribe"""
        assert self.client is None

        self.logger.info("Start")

        self.logger.info("Activating inside port in thread %s" % (threading.current_thread().getName(),))
        self._incoming.activate()
        self._incoming_plug = None

        self.client = mqtt.Client(client_id=self.client_id,
                                  clean_session=True,
```

```
                                userdata=None,
                                protocol=mqtt.MQTTv311,
                                transport="tcp")

        self.client.on_message = self.on_message
        self.client.on_connect = self.on_connect

        self.logger.info("Connecting to broker: %s:%d" % (self.host, self.port))
        self.client.connect_async(host=self.host,
                                port=self.port,
                                keepalive=60)

        self.logger.info("Starting client loop")

        self.client.loop_start()

    def on_clock(self):
        """Delayed initialization / start"""
        now = self.clock.recv_pyobj()
        self.clock.halt()
        self.logger.info("On clock (delayed start)")

        if self.client is None:
            self.start()

    def on_pubPort(self):
        msg = self.pubPort.recv_pyobj() # here message from computationalComponent is received
        if self.client is None:
            self.start()
        payload = json.dumps(msg).encode("utf-8")
        self.logger.info("Publish %s" % (payload), )
        self.client.publish(topic=self.topic2, payload=payload, qos=self.qos) # publish only data

    def on_incoming(self):
        payload = self._incoming.recv_pyobj()
        self.logger.info("Subscribe %s" % (payload,))
        msg = json.loads(payload.decode("utf-8"))
        self.subPort.send_pyobj(msg)

    def __destroy__(self):
        self.logger.info("Destroy")
        self.logger.info("Stopping client loop")
        self.client.loop_stop(force=True)
```

B.4 ComputationalComponent Python code

```
'''
ComputationaComponent Modbus-MQTT

Created on Jun 14, 2019

@Author: Danny Crescimone
'''

import zmq
from riaps.run.comp import Component
from riaps.run.exc import PortError
import uuid
import os
from collections import namedtuple
from ModbusUartReqRepDevice import CommandFormat, ModbusCommands
import pydevd
import time
import logging

''' Enable debugging to gather timing information on the code execution'''
debugMode = True

RegSet = namedtuple('RegSet', ['idx', 'value'])
InputRegs = namedtuple('InputRegs', ['first', 'second', 'third', 'fourth'])

hReg1 = "start"  # to name the holding register
hReg2 = "power"

'''For Inverter control'''
HoldingRegs = namedtuple('HoldingRegs', [hReg1, hReg2])


class ComputationalComponent(Component):
    def __init__(self,dev):
        super().__init__()
        self.uuid = uuid.uuid4().int
        self.pid = os.getpid()
        '''Inizialize register'''
        self.inputRegs = InputRegs(RegSet(0,12), RegSet(1, 45), RegSet(2, 56), RegSet(3, 78))
        self.holdingRegs = HoldingRegs(RegSet(0, 0), RegSet(1, 0))

        self.message_sent = False

        '''Setup Commands'''
        self.type = dev  # should contain the Device typology name (for example "DCESD")
        self.numRegsToRead = len(self.inputRegs)
        self.defaultNumOfRegs = 1
        self.dummyValue = [0]
        self.defaultNumOfDecimals = 0
        self.signedDefault = False
        self.ModbusPending = 0
        self.ModbusReady = False
        self.logger.info("ComputationalComponent: %s - starting" % str(self.pid))

    def on_clock(self):
        now = self.clock.recv_pyobj()
        self.logger.info("on_clock()[%s]: %s" % (str(self.pid), str(now)))
```

```
        if not self.ModbusReady:
            # Halt clock while waiting for the status
            self.clock.halt()
            self.logger.info("clock halted")
            # If status=true, set flag and restart clock
            try:
                msg = "readytest"
                if not self.message_sent:
                    self.modbusStatusReqPort.send_pyobj(msg)
                    self.logger.info("Modbus status requested")
                    self.message_sent = True
                if self.message_sent:
                    modStatus = self.modbusStatusReqPort.recv_pyobj()
                    self.logger.info("Modbus status received")
                    self.message_sent = False
                    if modStatus:
                        self.ModbusReady = True
                        self.logger.info("Modbus is ready, clock is restarted")
            except PortError as e:
                self.logger.error("on_clock-modbusStatusReq:port exception = %d" % e.errno)
                if e.errno in (PortError.EAGAIN, PortError.EPROTO):
                    self.logger.error("on_clock-modbusStatusReq: port error received")
            self.clock.launch()
            self.logger.info("clock restarted")
        else:
            '''Read all input registers''' # on_clock
            self.command = CommandFormat(ModbusCommands.READMULTI_INPUTREGS, self.inputRegs.first.idx, self.numRegsToRead,
                                         self.dummyValue, self.defaultNumOfDecimals, self.signedDefault)
            self.SendReq(self.command)

    def on_RecCommand(self):  # to handle the commands received by MQTT
        if self.ModbusReady:
            cmd = self.RecCommand.recv_pyobj()  # Dictionary from MQTT (to test)
            self.logger.info("Command received from MQTT : %s" % str(cmd))
            numRegsToWrite = len(cmd)  # to count how many holding registers it has to write

            if numRegsToWrite == 1:
                '''Read all the holding register'''
                if "read" in cmd:
                    self.command = CommandFormat(ModbusCommands.READMULTI_HOLDINGREGS, self.holdingRegs.start.idx,
                                                 len(self.holdingRegs), self.dummyValue, self.defaultNumOfDecimals, self.signedDefault)
                    self.SendReq(self.command)

                '''Write a single holding register'''
                if hReg1 in cmd:
                    self.values = [cmd[hReg1]]
                    self.logger.info("command received from SCADA: %s" % str(self.values))
                    self.command = CommandFormat(ModbusCommands.WRITE_HOLDINGREG, self.holdingRegs.start.idx,
                                                 numRegsToWrite, self.values, self.defaultNumOfDecimals, self.signedDefault)
                    self.SendReq(self.command)
                elif hReg2 in cmd:
                    if self.type == "LVSST/G":
                       cmd[hReg2] = cmd[hReg2] + 1500 # to convert the command for DSP
                    elif self.type == "DCESD":
                       cmd[hReg2] = cmd[hReg2] + 2500 # to convert the command for DSP
                    self.values = [cmd[hReg2]]
                    self.command = CommandFormat(ModbusCommands.WRITE_HOLDINGREG, self.holdingRegs.power.idx, numRegsToWrite,
                                                 self.values, self.defaultNumOfDecimals, self.signedDefault)
```

```
                self.SendReq(self.command)

            elif numRegsToWrite == 2:
                '''Write all holding register'''
                if(hReg1 in cmd) and (hReg2 in cmd):
                    if self.type == "LVSST/G":
                        cmd[hReg2] = cmd[hReg2] + 1500  # to convert the command for DSP
                    elif self.type == "DCESD":
                        cmd[hReg2] = cmd[hReg2] + 2500  # to convert the command for DSP
                    self.values = [cmd[hReg1], cmd[hReg2]]
                    self.command = CommandFormat(ModbusCommands.WRITEMULTI_HOLDINGREGS, self.holdingRegs.start.idx,
                                    numRegsToWrite,self.values, self.defaultNumOfDecimals, self.signedDefault)
                    self.SendReq(self.command)
                else:
                    self.logger.info("wrong command keys sent, they should be: <%s> and <%s>" % (hReg1,hReg2))

    def SendReq(self, msg):  # method to send Modbus request
        '''Send Command'''
        if self.ModbusPending == 0:
            if debugMode:
                self.cmdSendStartTime = time.perf_counter()
                self.logger.debug(
                    "on_clock()[%s]: Send command to ModbusUartDevice at %f" % (str(self.pid), self.cmdSendStartTime))
            try:
                self.modbusCommandReqPort.send_pyobj(msg)
                self.ModbusPending += 1
                self.logger.info("Modbus command sent")
            except PortError as e:
                self.logger.error("on_clock-modbusCommandReqPort:send exception = %d" % e.errno)
                if e.errno in (PortError.EAGAIN, PortError.EPROTO):
                    self.logger.error("on_clock-modbusCommandReqPort: port error received")
        else:
            self.logger.info("Modbus is pending. Try to send the command later")


    def on_modbusCommandReqPort(self): # method to receive Modbus replay and handle message
        '''Receive Response'''
        try:
            msg = self.modbusCommandReqPort.recv_pyobj()
            self.ModbusPending -= 1
            self.logger.info("Modbus command response received")
        except PortError as e:
            self.logger.error("on_modbusCommandReqPort:receive exception = %d" % e.errno)
            if e.errno in (PortError.EAGAIN, PortError.EPROTO):
                self.logger.error("on_modbusCommandReqPort: port error received")

        if debugMode:
            self.cmdResultsRxTime = time.perf_counter()
            self.logger.debug(
              "on_modbusCommandReqPort()[%s]: Received Modbus data=%s from ModbusUartDev at %f, time from cmd to data is %f ms"
                  % (str(self.pid), repr(msg), self.cmdResultsRxTime,(self.cmdResultsRxTime - self.cmdSendStartTime) * 1000))

        if self.command.commandType == ModbusCommands.READMULTI_INPUTREGS:
            Cmsg = self.Convertions(msg)
            logMsg = "Register " + str(self.command.registerAddress) + " values are " + str(Cmsg)

            self.pubPort.send_pyobj(Cmsg) # message that go to MQTT
```

```
            self.logger.info("message send to MQTT: %s" % str(Cmsg))
        elif self.command.commandType == ModbusCommands.READMULTI_HOLDINGREGS:
            logMsg = "Register " + str(self.command.registerAddress) + " values are " + str(msg)
            #self.pubPort.send_pyobj(msg) # message that go to MQTT this is only for debugging fase
            self.logger.info("message send to MQTT: %s" % str(msg))
        elif self.command.commandType == ModbusCommands.WRITE_HOLDINGREG:
            logMsg = "Wrote Register " + str(self.command.registerAddress)
        elif self.command.commandType == ModbusCommands.WRITEMULTI_HOLDINGREGS:
            logMsg = "Wrote Registers " + str(self.command.registerAddress) + " to " + str(
                self.command.registerAddress + self.command.numberOfRegs - 1)
        self.tx_modbusData.send_pyobj(logMsg)  # Send log data"""

    def Convertions(self, msg):  # Register value convertions
        d = {}

        if self.type == "LVSST/G":
           msg[0] = round(10000-msg[0]*20000/65535)  # AC side active power[W]
           msg[1] = round(msg[1]*10000/65535-5000)   # AC side reactive power[Var]
           msg[2] = round(msg[2]*500/65535)          # AC RMS voltage[V]
           msg[3] = round(msg[3]*10000/65535-5000)   # DC side power[W]
           for i in range(len(msg)):
              if (msg[i] > 0 and msg[i] < 30) or (msg[i] < 0 and msg[i] > -30):
                 msg[i] = 0
           msg[2] = max(msg[2], 0)  # to saturate the data

           d = {"AC side Active Power[W]":msg[0],"AC side Reactive Power[Var]":msg[1],
                                 "AC RMS Voltage[V]":msg[2],"DC side Power[W]":msg[3]}

        elif self.type == "LVSST/L":
           msg[0] = round(msg[0]*200/65535)          # AC port1 voltage[V]
           msg[1] = round(msg[1]*6000/65535-3080)    # AC port1 active power[W]
           msg[2] = round(msg[2]*200/65535)          # AC port2 voltage[V]
           msg[3] = round(msg[3]*6000/65535-3050)    # AC port2 active power[W]
           for i in range(len(msg)):
              if (msg[i] > 0 and msg[i] < 30) or (msg[i] < 0 and msg[i] > -30):
                 msg[i] = 0
           for z in range(len(msg)):  # to saturate the data
               msg[z] = max(msg[z], 0)

           d = {"AC port1 Voltage[V]":msg[0],"AC port1 Active Power[W]":msg[1],
                       "AC port2 Voltage[V]":msg[2],"AC port2 Power[W]":msg[3]}

        elif self.type == "DCESD":
           msg[0] = round(msg[0]*10000/65535-5000)                    # Battery power[W]
           msg[1] = max(round(msg[1]*520/65535-20), 0)                # DC voltage[V]
           msg[2] = float("{0:.2f}".format((msg[2]*100/65535-50)/5))  # Battery current[A]
           for i in range(len(msg)-2):
              if (msg[i] > 0 and msg[i] < 35) or (msg[i] < 0 and msg[i] > -35):
                 msg[i] = 0
           msg[1] = max(msg[1], 0)  # to saturate the data
           msg[3] = max(msg[3], 0)

           d = {"Battery Power[W]":msg[0],"DC Voltage[V]":msg[1],"Battery Current[A]":msg[2],"SoC[%]":msg[3]} # Python Dict

        return d

    def __destroy__(self):
        self.logger.info("[%d] destroyed" % self.pid)
```

B.5 ModbusUartReqRepDevice Python code

```
'''
This module utilizes the MinimalModbus (which utilizes pySerial).
Both need to be installed in the development environment.
    $ sudo pip3 install minimalmodbus  (which should install pySerial)
'''

from riaps.run.comp import Component
#import logging
import os
import serial
from serialModbusLib.serialModbusComm import SerialModbusComm,PortConfig
from collections import namedtuple
from enum import Enum
#import pydevd
import time
import sys


''' Enable debugging to gather timing information on the code execution'''
debugMode = False

class ModbusCommands(Enum):
    READ_BIT = 1
    READ_INPUTREG = 2
    READ_HOLDINGREG = 3
    READMULTI_INPUTREGS = 4
    READMULTI_HOLDINGREGS = 5
    WRITE_BIT = 6
    WRITE_HOLDINGREG = 7
    WRITEMULTI_HOLDINGREGS = 8

CommandFormat = namedtuple('CommandFormat', ['commandType','registerAddress','numberOfRegs',
                        'values','numberOfDecimals','signedValue'])

class ModbusUartReqRepDevice(Component):
    def __init__(self,slaveaddress=0,port="UART2",baudrate=19200,bytesize=serial.EIGHTBITS,
          parity=serial.PARITY_NONE,stopbits=serial.STOPBITS_ONE,serialTimeout=0.05): # def for Modbus spec
        super().__init__()
        self.pid = os.getpid()
        self.statusRequestPending = 0
        self.commandRequestPending = 0

        if port == 'UART1':
            self.port = '/dev/ttyO1'
        elif port == 'UART2':
            self.port = '/dev/ttyO2'
        elif port == 'UART3':
            self.port = '/dev/ttyO3'
        elif port == 'UART4':
            self.port = '/dev/ttyO4'
        elif port == 'UART5':
            self.port = '/dev/ttyO5'
        else:
            self.logger.error("__init__[%s]: Invalid UART argument, use UART1..5" % self.pid)

        self.port_config = PortConfig(self.port, baudrate, bytesize, parity, stopbits, serialTimeout)
        self.slaveAddressDecimal = slaveaddress
        self.modbus = SerialModbusComm(self.slaveAddressDecimal,self.port_config)
```

```
        self.logger.info("Modbus settings %d @%s:%d %d%s%d [%d]" % (self.slaveAddressDecimal,
               self.port_config.portname,self.port_config.baudrate,self.port_config.bytesize,
               self.port_config.parity,self.port_config.stopbits,self.pid))
        self.modbus.startModbus()
        self.logger.info("Modbus started")

    def __destroy__(self):
        self.modbus.stopModbus()
        self.logger.info("__destroy__")

    '''
    Receive a Modbus status request.  Send back state of the modbusReady flag
    '''
    def on_modbusStatusRepPort(self):
        self.logger.info("start on_modbusStatusReqPort")
        try:
            statusRequest = self.modbusStatusRepPort.recv_pyobj()
            self.statusRequestPending += 1
            self.logger.info("Request for Modbus device status received")
        except PortError as e:
            self.logger.error("on_modbusStatusRepPort:receive exception = %d" % e.errno)
            if e.errno in (PortError.EAGAIN,PortError.EPROTO):
                self.logger.error("on_modbusStatusRepPort: port error received")

        if self.statusRequestPending == 1:
            try:
                msg = self.modbus.isModbusAvailable()
                if msg:
                    self.logger.info("Modbus is available")
                    self.modbusStatusRepPort.send_pyobj(str(msg))
                    self.statusRequestPending -= 1
                    self.logger.info("Response for Modbus device status sent")
                else:
                    self.logger.info("Modbus is not available")

            except PortError as e:
                self.logger.error("on_modbusStatusRepPort:send exception = %d" % e.errno)
                if e.errno in (PortError.EAGAIN,PortError.EPROTO):
                    self.logger.error("on_modbusStatusRepPort: port error received")
        elif self.statusRequestPending > 1:
    # This should not happen if the requesting component has the appropriate error handling for request/reply
            self.logger.info("A status request is pending, no additional request was made")


    '''
    Receive a Modbus command request.  Process command and send back response.
    '''
    def on_modbusCommandRepPort(self):
        '''Request Received'''
        try:
            commandRequest = self.modbusCommandRepPort.recv_pyobj()
            self.commandRequestPending += 1
            self.logger.debug("Request for Modbus device command received")
            if debugMode:
                self.modbusReqRxTime = time.perf_counter()
                self.logger.debug("modbusCommandRepPort()[%s]: Request=%s Received at %f" %
                            (str(self.pid),commandRequest,self.modbusReqRxTime))
```

```
        except PortError as e:
            self.logger.error("on_modbusCommandRepPort:receive exception = %d" % e.errno)
            if e.errno in (PortError.EAGAIN,PortError.EPROTO):
                self.logger.error("on_modbusCommandRepPort: port error received")

        if self.commandRequestPending == 1:
            self.unpackCommand(commandRequest)
            responseValue = -1  # invalid response
            if self.modbus.isModbusAvailable() == True:
                self.logger.debug("Sending command to Modbus interface")
                responseValue = self.sendModbusCommand()
                self.logger.debug("Modbus Response received by device")
                if debugMode:
                    t1 = time.perf_counter()
                    self.logger.debug("modbusCommandRepPort()[%s]: Send Modbus response=%s back to requester at %f"%
                     (str(self.pid),responseValue,t1))
            else:
                self.logger.info("Modbus is not available")

            '''Send Results'''
            try:
                self.modbusCommandRepPort.send_pyobj(responseValue)
                self.commandRequestPending -= 1
                self.logger.debug("Response for Modbus device command sent")
            except PortError as e:
                self.logger.error("on_modbusCommandRepPort:send exception = %d" % e.errno)
                if e.errno in (PortError.EAGAIN,PortError.EPROTO):
                    self.logger.error("on_modbusCommandRepPort: port error received")

        elif self.commandRequestPending > 1:
            # This should not happen if the requesting component has the appropriate error handling for request/reply
            self.logger.info("A command request is pending, no additional request was made")


    def unpackCommand(self,rxCommand):
        self.commmandRequested = rxCommand.commandType
        self.registerAddress = rxCommand.registerAddress
        self.numberOfRegs = rxCommand.numberOfRegs
        self.numberOfDecimals = rxCommand.numberOfDecimals
        self.signedValue = rxCommand.signedValue
        self.values = rxCommand.values


    def sendModbusCommand(self):
        value = 999  # large invalid value

        if debugMode:
            t0 = time.perf_counter()
            self.logger.debug("sendModbusCommand()[%s]: Sending command to Modbus library at %f" % (str(self.pid),t0))
        self.logger.debug("sendModbusCommand(): Sending command to Modbus library")

        try:
            if self.commmandRequested == ModbusCommands.READ_INPUTREG:
                value = self.modbus.readInputRegValue(self.registerAddress, self.numberOfDecimals, self.signedValue)
                self.logger.debug("ModbusUartDevice: sent command %s, register=%d, numOfDecimals=%d, signed=%s" %
                        (ModbusCommands.READ_INPUTREG.name,self.registerAddress,self.numberOfDecimals,str(self.signedValue)))
            elif self.commmandRequested == ModbusCommands.READ_HOLDINGREG:
```

```
        # This should not happen if the requesting component has the appropriate error handling for request/reply
        self.logger.info("A command request is pending, no additional request was made")



def unpackCommand(self,rxCommand):
    self.commmandRequested = rxCommand.commandType
    self.registerAddress = rxCommand.registerAddress
    self.numberOfRegs = rxCommand.numberOfRegs
    self.numberOfDecimals = rxCommand.numberOfDecimals
    self.signedValue = rxCommand.signedValue
    self.values = rxCommand.values


def sendModbusCommand(self):
    value = 999  # large invalid value

    if debugMode:
        t0 = time.perf_counter()
        self.logger.debug("sendModbusCommand()[%s]: Sending command to Modbus library at %f" % (str(self.pid),t0))
    self.logger.debug("sendModbusCommand(): Sending command to Modbus library")

    try:
        if self.commmandRequested == ModbusCommands.READ_INPUTREG:
            value = self.modbus.readInputRegValue(self.registerAddress, self.numberOfDecimals, self.signedValue)
            self.logger.debug("ModbusUartDevice: sent command %s, register=%d, numOfDecimals=%d, signed=%s" %
                    (ModbusCommands.READ_INPUTREG.name,self.registerAddress,self.numberOfDecimals,str(self.signedValue)))
        elif self.commmandRequested == ModbusCommands.READ_HOLDINGREG:
            value = self.modbus.readHoldingRegValue(self.registerAddress, self.numberOfDecimals, self.signedValue)
            self.logger.debug("ModbusUartDevice: sent command %s, register=%d, numOfDecimals=%d, signed=%s" %
                  (ModbusCommands.READ_HOLDINGREG.name,self.registerAddress,self.numberOfDecimals,str(self.signedValue)))
        elif self.commmandRequested == ModbusCommands.READMULTI_INPUTREGS:
            value = self.modbus.readMultiInputRegValues(self.registerAddress, self.numberOfRegs)
            self.logger.debug("ModbusUartDevice: sent command %s, register=%d, numOfRegs=%d" %
                  (ModbusCommands.READMULTI_INPUTREGS.name,self.registerAddress,self.numberOfRegs))
        elif self.commmandRequested == ModbusCommands.READMULTI_HOLDINGREGS:
            value = self.modbus.readMultiHoldingRegValues(self.registerAddress, self.numberOfRegs)
            self.logger.debug("ModbusUartDevice: sent command %s, register=%d, numOfRegs=%d" %
                   (ModbusCommands.READMULTI_HOLDINGREGS.name,self.registerAddress,self.numberOfRegs))
        elif self.commmandRequested == ModbusCommands.WRITE_HOLDINGREG:
            self.modbus.writeHoldingRegister(self.registerAddress, self.values[0], self.numberOfDecimals, self.signedValue)
            self.logger.debug("ModbusUartDevice: sent command %s, register=%d, value=%d, numberOfDecimals=%d, signed=%s" %
             (ModbusCommands.WRITE_HOLDINGREG.name,self.registerAddress,self.values[0],self.numberOfDecimals,str(self.signedValue)))
        elif self.commmandRequested == ModbusCommands.WRITEMULTI_HOLDINGREGS:
            self.modbus.writeHoldingRegisters(self.registerAddress, self.values)
            self.logger.debug("ModbusUartDevice: sent command %s, register=%d" %
                (ModbusCommands.WRITEMULTI_HOLDINGREGS.name,self.registerAddress))
            self.logger.debug("ModbusUartDevice: Values - %s" % str(self.values).strip('[]'))
    except Exception as e:
        self.logger.error("ModbusUartDevice: Exception thrown during Modbus command action - %s" % e)

    if debugMode:
        t1 = time.perf_counter()
        self.logger.debug("sendModbusCommand()[%s]: Modbus library cmd compl at %f, time to int with Modbus library is %f ms"
                % (str(self.pid),t1,(t1-t0)*1000))

    return value
```

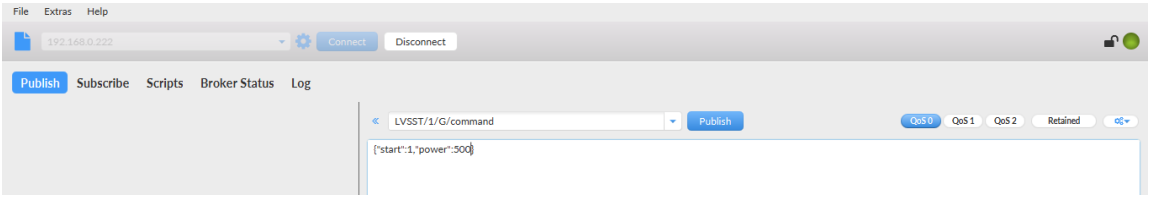
B.6 Logger

```
from riaps.run.comp import Component
#from influxdb import InfluxDBClient
#from influxdb.client import InfluxDBClientError
import json
import logging
from datetime import datetime
import os

BATCH_SIZE = 60

class Logger(Component):
    def __init__(self, db_host, db_port, db_name, db_user, db_password):
        super().__init__()
        self.pid = os.getpid()
        self.logger.info("%s - starting modbus logger" % str(self.pid))
        self.point_values = []
        #self.client = InfluxDBClient(host=db_host, port=db_port,
        #    database=db_name, username=db_user, password=db_password)

    def on_rx_modbusData(self):
        msg = self.rx_modbusData.recv_pyobj()
        self.logger.info("on_rx_modbusData()[%s]: %s" % (str(self.pid), repr(msg)))

        '''
        timestamp = int(1e9 * data['timestamp'])
        # alternative:
        # timestamp = datetime.utcfromtimestamp(data['timestamp'])
        self.point_values.append({
                "time": timestamp,
                "measurement": "pmu",
                "fields":  {
                    "VAGA": data["VAGA"],
                    "VASA": data["VASA"],
                    "VASM": data["VASM"],
                    "VAGM": data["VAGM"],
                },
                "tags": {
                    "Actor" : "ModbusIOActor"
                },
            })
        '''

        if len(self.point_values) >= BATCH_SIZE:
            self.client.write_points(self.point_values)
            self.point_values = []
```

Appendix C

MQTT.FX

Publish



Subscribe