UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

BACHELOR IN COMPUTING ENGINEERING

Specialization in computing

BACHELOR DISSERTATION

A multi-agent system for automation of configuration and simulation of photovoltaic gridconnected systems

Enrique Cepeda Villamayor

September, 2021

UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

Tecnologías y Sistemas de Información

Specialization in computing

BACHELOR DISSERTATION

A multi-agent system for automation of configuration and simulation of photovoltaic gridconnected systems

Author: Enrique Cepeda Villamayor

Supervisor: Luis Rodríguez Benítez

Supervisor: Luis Jiménez Linares

September, 2021

TFG Enrique Cepeda Villamayor - Smart Grid Simulator

Este texto ha sido preparado con la plantilla LaTeX de TFG para la UCLM publicada por Jesús Salido en GitHub[1] y Overleaf [2] como parte del curso *«LaTeX esencial para preparación de TFG, Tesis y otros documentos académicos»* impartido en la Escuela Superior de Informática de la Universidad de Castilla-La Mancha.

[1] https://github.com/JesusSalido/TFG_ESI_UCLM
[2] https://www.overleaf.com/latex/templates/plantilla-de-tfg-escuela-superior-de-informatica-uclm/phjgscmfqtsw

Tribunal:

Presidente: _____

Vocal: _____

Secretario: _____

Fecha de defensa: _____

Calificación: _____

Presidente	Vocal	Secretario
Fdo.:	Fdo.:	Fdo.:

To my family
For always trusting in me.

Abstract

Nowadays, business models based on self-generated Photovoltaic (PV) energy are flourishing, as they bring economic and environmental benefits. Moreover, the development of new Renewable Energy Sources (RESs) and the steady decrease of the PV panels production costs promote energy self-consumption and Distributed Generation (DG). Accordingly, government institutions and companies are betting on cleaner ways of generating energy as an investment for the present and the future. This Bachelor of Science (BSc) thesis is focused on the development of an information system to simulate the behavior of a **Smart Grid** (A smart electricity network) of buildings. The goal of the Smart Grids is the improvement of the electrical power infrastructure by integrating more and cleaner energy sources. In this sense, this BSc thesis proposes a grid-connected system in which the energy comes from RESs, and in particular, from PV systems installed on the building's roof-top. These type of networks are composed by *consumer buildings*, which request their energy needs, *producer buildings*, that share their surplus energy with the network, and *prosumer buildings*, that can take on both roles. The developed system makes possible to configure these grids in a myriad of ways, and subsequently, simulate and visualize the energy distribution among the buildings. In this context, multiagent systems are the right fit for the energy distribution, in which every agent represents a building in the grid. The objective of the system agents is maximizing *social-welfare* by sharing their surplus energy in a non-competitive environment. This tool allows to simulate the behavior of Smart Grids and simplify an analysis on their performance, lowering the entry barriers of collective self-consumption for the general public.

Resumen

Hoy en día, los modelos de negocio basados en la energía fotovoltaica autogenerada están en auge, ya que su uso tiene beneficios tanto económicos como medioambientales. Además, el desarrollo de nuevas fuentes de energías renovables y una disminución constante en los costes de producción de los paneles fotovoltaicos promueven el autoconsumo de energía y la Generación Distribuida (GD). Es por esto que tanto instituciones gubernamentales como compañías están apostando en formas más limpias de generar energía como una inversión de presente y de futuro. Este Trabajo Final de Grado (TFG) se focaliza en el desarrollo de un sistema de la información para simular el comportamiento de una **Red Eléctrica Inteligente (Smart Grid)** de edificios. El objetivo de estas redes inteligentes es mejorar la infraestructura eléctrica mediante la integración de más fuentes de energía y más limpias. En este sentido, este TFG propone un sistema conectado a la red en el que la energía proviene de fuentes de energías renovables, y concretamente de sistemas fotovoltaicos instalados en los tejados de los edificios. Este tipo de redes están compuestas por *edificios consumidores*, que solicitan sus necesidades energéticas, *edificios productores*, que comparten su excedente de energía con la red, y los *edificios prosumidores*, que pueden tomar ambos roles. El sistema desarrollado hace posible la configuración de estas redes en numerosas maneras y consecuentemente, la simulación y visualización de la distribución de energía entre estos edificios. En este contexto, los sistemas multiagente se ajustan perfectamente al modelo de distribución de energía, donde cada agente representa un edificio en la red. El objetivo de los agentes del sistema es maximizar el *bienestar social* mediante el reparto del excedente de energía en un entorno no competitivo. Esta herramienta permite simular el comportamiento de los Smart Grids y simplifica el análisis de su rendimiento, reduciendo las barreras de entradas del autoconsumo colectivo para el público general.

Acknowledgements

En primer lugar quisiera agradecer a Luis Jiménez Linares y a Luis Rodríguez Benítez por la oportunidad que me han dado al proponer y supervisarme este TFG, vuestra ayuda ha hecho que este proyecto haya sido posible. También a Rafa, Iñaki y Miguel, que apostasteis por mí y me habéis ayudado en todo lo posible.

A mis padres, porque siempre me habéis apoyado en todo lo que me he propuesto y me habéis ayudado en los momentos difíciles, habéis sido un pilar fundamental para mí durante todos estos años. A mi hermano, Jaime, mi mentor y mi apoyo durante toda la carrera, porque has sabido ayudarme y escucharme. Me has hecho ver todo lo que se puede conseguir con esfuerzo y trabajo, espero algún día ser tan bueno como tú. Por supuesto al resto de mi familia, mi abuela, mis tíos, primos, por depositar vuestra confianza en mí.

A mis compañeros y amigos de la universidad, porque me habéis aportado muchísimas cosas. Sin los momentos vividos y sin vuestro apoyo seguro que no estaría escribiendo estas líneas.

Y a mis amigos de siempre, a Adrián, Alberto, Mateo, José Antonio, y a todos los demás que habéis estado y estáis ahí.

Enrique Cepeda Villamayor
Ciudad Real, 2021

Contents

List of Figures

List of Tables

Listings

List of Acronyms

FOSS Free and Open Source Software

DVCS Distributed Version Control System

BSc Bachelor of Science

RAM Random Access Memory

SSD Solid State Drive

OS Operating System

DBMS Relational Database Management Systems

JVM Java virtual Machine

UI User Interface

DOM Document Object Model

DB Database

OS Open Source

PV Photovoltaic

IEA International Energy Agency

FIPA Foundation for Intelligent Physical Agents

TFG Trabajo Final de Grado

ORM Object Relational Mapper

IPCC Intergovernmental Panel on Climate Change

GHG Greenhouse Gasses

EC Energy Community

ROI Return Of Investment

GUI Graphical User Interface

MAS Multiagent System

OO Object Oriented

ACL Agent Communication Language

AMS Agent Management System

AUML Agent Unified Modeling Language

UML Unified Modeling Language

CFP Call For Proposal

JADE Java Agent DEvelopment Framework

DF Directory Facilitator

AP Agent Platform

AID Agent Identifier

MTS Message Transport Service

AP Agent Platform

AOP Agent Oriented Programming

RR Round Robin

FiT Feed-in Tariff

PPA Power Purchase Agreement

EU European Union

DSM Demand Side Management

EMS Energy Management System

DG Distributed Generation

DER Distributed Energy Resource

RES Renewable Energy Source

AC Alternating Current

UCLM Universidad de Castilla-La Mancha

POC Proof Of Concept

WYSIWYG What You See Is What You Get

OSM OpenStreetMaps

PoC Proof Of Concept

SPA Single Page Application

MPA Multiple Page Application

API Application Programming Interface

UX User Experience

URL Uniform Resource Locator

DRY Don't Repeat Yourself

DC Direct Current

Voc Voltage Open Circuit

STC Standard Test Conditions

GFS Global Forecast System

GHI Global Horizontal Irradiance

DNI Direct Normal Irradiance

DHI Direct Horizontal Irradiance

WGS World Geodesic System

REST Representational State Transfer

CFP Call For Proposal

SQL Structured Query Language

ASGI Asynchronous Server Gateway Interface

CLI Command Line Interface

DSL Domain Specific Language

CRUD Create Read Update Delete

CHAPTER 1

Introduction

Nowadays, electricity is the heart of modern economy. The growing consumption of electrical energy at a global level is a reality, a result of its use for heating, cooling, transportation, digital services or the industry sector, among a myriad of causes. All these reasons have a great impact on the environment, as anthropogenic climate change is detrimental to humanity, wildlife, and ecosystems. The Intergovernmental Panel on Climate Change (IPCC) highlights the need for even more severe cuts in global Greenhouse Gasses (GHG) emissions, local atmospheric pollutants and the consumption of natural resources [27]. Moreover, the dependence of current energy systems on fossil fuels is indubitably high. In 2019, an 81% of global energy generation came from fossil fuels [19]. These resources are finite, so to reduce this enormous dependence, the use of more sustainable energy sources such as solar energy or wind energy is necessary.

In this sense, Photovoltaic (PV) electricity generation via building attached/integrated PV systems has become increasingly popular in recent years. Remarkably, on-site PV electricity generation in buildings accelerates the transition from a centralized energy system to a sustainable, decentralized, and local one. These buildings can form the so-called Energy Communities (ECs). On the one hand, the European Union (EU) acknowledges their great potential [6]. On the other hand, most countries currently lack of a legal framework for this type of communities, however, Spain is not the case. Since 2019, Spain allows building self-consumption networks [41]. The International Energy Agency (IEA) in its *2019 energy report* argues that for a sustainable growth in the use of alternative energies, a reduction in the generated energy from fossil fuels should happen. The percentage of energy from these fuels should be reduced to 59% by 2040. [19]. For these, among other reasons, solutions need to be developed and various types of initiatives are emerging, which facilitate cleaner and more efficient energy consumption.

In this context, the Smart Grid is one of these initiatives. Smart Grids are bidirectional electrical networks that integrate the behavior of its components to ensure a sustainable energy system for both energy consumers and producers. Its goal is to improve the electric power infrastructure by integrating more energy sources, such as Distributed Energy Resources (DERs) and Renewable Energy Sources (RESs). These grids are formed by these components: energy producers, consumers or those who fulfill both roles simultaneously, the prosumers. Figure 1.1 shows the energy flow between the different grid components.

This BSc thesis focuses its attention on the configuration and simulation of these grids, which particularly trade their PV generation and demand profiles. The main goal is to build a system to

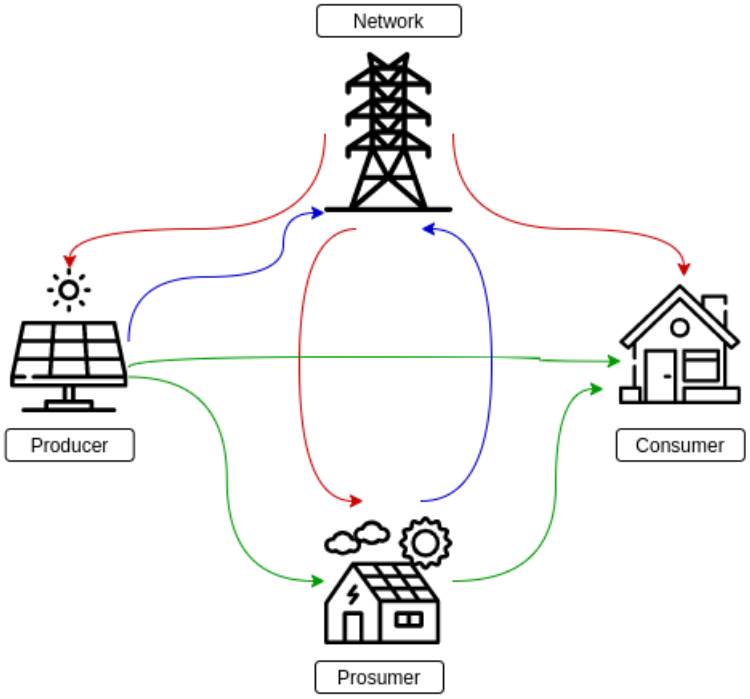


Figure 1.1: Energy distribution among smart grid components

design and get an optimized configuration for a Smart Grid. The users will take advantage of the benefits of the PV energy at practically zero cost, in exchange for an initial investment. However, it is important to note that this thesis is not focused on the Return Of Investment (ROI) of the grid configuration. For the design of the Smart Grids, a web application has been developed, whereas a multiagent system simulates its behavior once the design is finished. Other key points such as the PV panel installation on a building's roof-top, and the inference of the energy production of these installations are addressed as part of this BSc thesis.

1.1. DOCUMENT STRUCTURE

This section describes how the rest of the document is organized. To this end, each one of the subsequent chapters is briefly presented.

1.1.1. Chapter 3: Goals

In this chapter, the general and specific objectives that are addressed in this project are defined.

1.1.2. Chapter 4: Background

In this chapter, an introduction to Multiagent Systems (MASs) is done, explaining its core principles, its capabilities and Java Agent DEvelopment Framework (JADE), a framework to build these kinds of systems. Moreover, Smart Grids and concepts such as collective self-consumption are explained in

detail. Additionally, key points such as architectures, optimization methods and financial incentives in relation with Energy Management Systems (EMSs) are exposed.

1.1.3. Chapter 5: Methodology

In this chapter, the methodologies used to develop this BSc thesis are explained. For this purpose, Scrum has been used as the project management methodology. In addition, an iterative and incremental software development methodology has been applied. To end with, the different physical and software resources required to perform this project are defined.

1.1.4. Chapter 6: Results

In this chapter, the different artifacts and results obtained from the development plan are detailed. The chapter is divided into the different Sprints and the User Stories are discussed here.

1.1.5. Chapter 7: Conclusions

In this chapter, the achieved goals during the execution of the project are exposed. Furthermore, a set of improvements and future work proposals are described.
Finally, a user manual have been included to provide complementary information on the installation and usage of the tool

CHAPTER 2
Goals

2.1. MAIN GOAL

The main objective of this BSc thesis is the configuration and simulation of a Smart Grid of buildings, which aims to achieve a self-sustained distribution of energy by using PV panels installations. These buildings are interconnected and communicated, playing each of them the role of a rational agent. The main objective of this BSc thesis is expected to be achieved by accomplishing the following sub-goals:

2.2. SPECIFIC GOALS

- **SG1** Use map libraries and services to select the buildings which compose the Smart Grid and other necessary information such as the dimensions of the area in which the PV panels are placed.
- **SG2** Simulate the performance of PV energy systems depending on the GPS coordinates of the smart grid, climatic conditions, and other factors.
- **SG3** Implement an algorithmic procedure based in multi-agent theory to simulate the energy sharing among the buildings.
- **SG4** Implementation of a web interface module to visualize the energy distribution, generation, and consumption of the Smart Grid components.

The next section explains important concepts used as a basis for the thesis development.

CHAPTER 3
Background

This section reviews some relevant concepts used as a basis for the development of this BSc thesis.

3.1. MULTIAGENT SYSTEMS

The history of computing to date has been marked by five important and continuing trends [44]:

- Ubiquity
- Interconnection
- Intelligence
- Delegation
- Human-orientation

Ubiquity is understood as the cost reduction of computation systems which has made possible to introduce information processing on places and devices where, so far, it has not been possible nor cost-effective. Whereas the earliest computers only interacted with humans operators, computers today are normally **interconnected**. In fact, it is especially complex to find computers with no internet connection. Moreover, there is a trend towards the construction of computers with deeper **intelligence**, unthinkable only a short time ago, as there are more complex tasks than ever which can be automated.
Moreover, the possibility of work-load **delegation** over multiple machines has grown steadily over the years. Delegation implies yielding the control of certain task to computation systems. These tasks are becoming increasingly critical such as the air-flight control systems, the car self-driving systems and more. Sometimes, these decisions are more influenced by software engineer decisions than by human experience. At other times, system decisions are based on experts opinions, however, systems always have the last word. Human interaction with machines has evolved from simple interrupters into decisions based on internal knowledge and Graphical User Interface (GUI) interaction. At first, you needed to have a deep knowledge of the architecture to work with it via machine instructions. Later, *abstractions* appeared, such as the procedure programming paradigm, the abstract data types and more recently ones, such as the Object Oriented (OO) paradigm. These abstractions have created a more **user-friendly environment** for *humans* to interact with *machines*.

From the mentioned tendencies, a series of challenges for software developers emerge. With respect to **ubiquity** and **interconnection**, no techniques permit taking advantage of the distributed processors

potential. If these challenges are to be met, systems that can act on our behalf are needed. But for that, various capabilities need to be fulfilled. The first involves the system representing our best interests when interacting with other systems or people. The second implies that systems should act in an independent way, with no external interactions. As a result of these characteristics, a new paradigm in computer science known by the name of Agent Oriented Programming (AOP) appears. The result of applying this new paradigm is the so-called *multiagent systems*.
A Multiagent System (MAS) is composed by two or more agents exchanging messages with each other through some computer network infrastructure. An agent is a tuple **(Environment, Agent)** placed on a certain environment, capable of making an action to interact with the environment and to satisfy a series of goals on an autonomous way. Agents have two main characteristics: The first, as previously mentioned, is the autonomy. The ability to decide for themselves their next action. The second is the ability to interact with other agents by *interchanging data*, *coopering* for a common goal, *coordinating* its actions and *negotiating*. In the following points, the intelligent agents and their more relevant properties are going to be explained.

Intelligent Agent

An intelligent agent is a computation system placed in an environment in which it takes actions in a flexible and autonomous way to accomplish the objectives it was built for. Furthermore, an agent must have the following properties [44]:

- **Autonomy.** The agent is capable of acting without direct human or agent intervention.
- **Sociability.** The agents are capable of interacting with each other by using a specific communication language.
- **Mobility.** The agent is on an environment from which is capable of perceiving and responding to the stimulus and changes in consequence
- **Proactivity.** The agent not only has to react, but also take the lead to act in accordance with its goals and towards them.
- **Rationality.** The agent is capable of modifying its behavior through experience and intelligence to take the best possible decision based on its performance measure.

Figure 3.1 shows the behavior of an agent in a schematic way. The agent perceives the changes through the sensors in the environment, and it produces an action which could affect the environment. In this sense, an agent is composed by an architecture and a program. The architecture enables the agent to have perceptions, to process them internally and to execute the actuators as a result of the processing, whereas the program is in charge of receiving perceptions from the environment and generating a series of actions as a consequence.
It is important to note that in high complexity domains, the agent cannot have an absolute control over the environment. If an agent executes an action twice under the same circumstances, it can have different effects, that is why they must be prepared to fail. The environments in which the agents interact have the following classification [36]:

- **Observable vs Partially Observable.** In an observable environment, the agents can obtain complete, precise and up-date information from it, completely sufficient to make the optimal

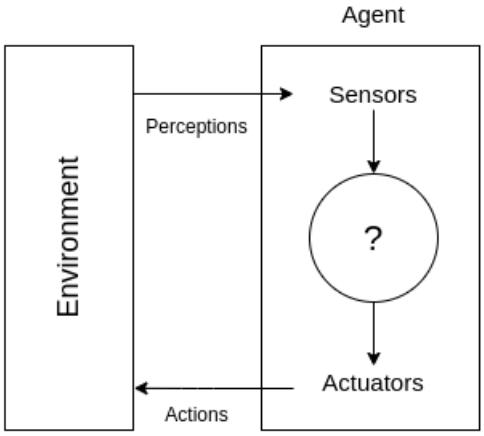


Figure 3.1: Agent interaction with the environment.

decision, otherwise, the environment is partially observable.

- **Deterministic vs Stochastic.** In a deterministic environment, the agent's actions uniquely determine the outcome, however, in a stochastic environment there is a certain amount of randomness.
- **Episodic vs Sequential.** In episodic environments, the agent action occurs in atomic episodes. Each episode consists of the perception of the agent and the performance of a unique action. The next episodes do not depend on the actions taken in the previous episodes. In sequential environments, on the other hand, the current decision could affect all future decisions.
- **Static vs Dynamic.** Whereas a static environment remains unaltered except by the agent actions effects, in a dynamic environment other agents can also interact.
- **Discrete vs Continuous.** An environment is static only if a fixed number of perceptions can be obtained from it.

Once defined what intelligent agents are and how they are composed to form a MAS, the way in which they communicate is going to be explained. The main property of a group of agents is the communication. In this BSc thesis, a Foundation for Intelligent Physical Agents (FIPA) standard for message exchange has been used, the FIPA—Agent Communication Language (ACL), which deals with messages, message exchange interaction protocols, speech act theory-based communicative acts and content language representations. The following section details the advantages of FIPA standards:

3.1.1. FIPA standards

FIPA is a body for developing and setting computer software standards for interacting with agents and agent-based systems. Originally, it was founded with the aim of defining a whole set of standards for implementing software standard specifications within which agents could execute. It also helps to specify how agents themselves should interoperate and communicate in a standard manner. The most widely adopted of the FIPA standards are the Agent Management Specification and (FIPA-ACL)

specification, explained hereafter.

Agent Management Specification

The Agent Management Specification provides the normative framework within which FIPA agents exist and operate. It establishes the logical reference model for the creation, registration, location, communication, migration, and retirement of the agents. The agent management reference model (Figure 3.2) consists of the following logical components, each representing a capability set [12]:

- An **agent**, which is the fundamental actor of an **Agent Platform (AP)**, and it may be registered at a number of transport addresses at which it can be contacted. Each agent must be uniquely identified by an **Agent Identifier (AID)**.
- A **Directory Facilitator (DF)**, which provides a yellow pages service to the AP agents. Agents may register their services with the DF or query the DF to find out what services are offered by other agents. Multiple DFs may exist within an AP.
- An **Agent Management System (AMS)**, a mandatory component of the Agent Platform. The AMS exerts supervisory control over access to and use of the AP. Only one AMS will exist in a single AP. The AMS maintains a directory of AIDs, which contains transport addresses (among other things) for agents registered with the AP. The AMS offers white pages services to other agents. Each agent must register with an AMS in order to get a valid AID.
- An **Message Transport Service (MTS)**, which is the default communication method between agents on different APs
- An **Agent Platform (AP)** provides the physical infrastructure in which agents can be deployed. The AP consists of the machine(s), operating system(s), agent support software, FIPA agent management components (a DF, an AMS and a MTS), and the agents.

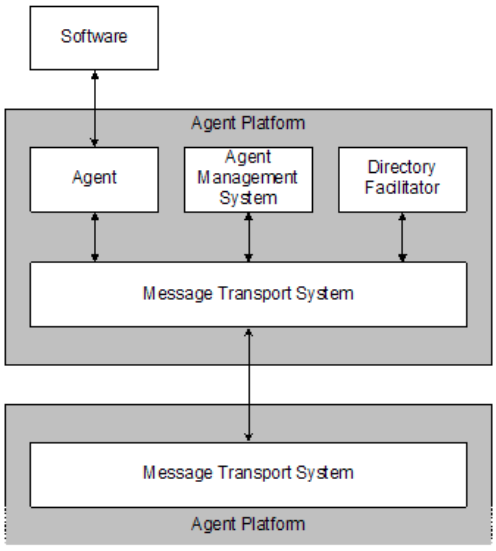


Figure 3.2: Agent Management Reference Model
Source: *Agent Management Reference Model* [12].

FIPA-ACL Message Structure Specification

FIPA defines different types of messages and communication protocols for a MAS. A FIPA-ACL message (Figure 3.3) contains a set from one to many parameters. The compulsory parameter on any ACL message is the performative, however, parameters such as the sender, the receiver, and the content is expected. Table 3.1 shows all possible parameters a message can have along with their meaning. The conversations between agents normally follow certain patterns. For that, FIPA has defined certain interaction protocols, explained in the following section.

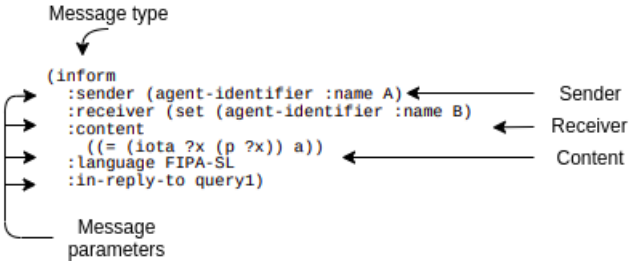


Figure 3.3: Example of an inform ACL message.

Parameter	Meaning
:content	Message content
:sender	Message sender identity
:receiver	Message receiver identity (From one to many receivers)
:language	Representation language employed on attribute :content
:ontology	Ontology name used on attribute :content
:reply-with	Response tag
:in-reply-to	Expected response tag
:protocol	Protocol identifier of used interaction
:conversation-id	Identifier of a series of communicative acts which form part of a certain conversation
:reply-to	Agent to send the responses
:reply-by	Maximum time in which the message must be received

Table 3.1: ACL Message parameters.

FIPA Interaction Protocols

All these protocols can be understood and expressed by means of Agent Unified Modeling Language (AUML) diagrams [18]. These diagrams are designed to model interaction protocols for MASs and they specify the communication flow between various agents. They are similar to Unified Modeling Language (UML) interaction diagrams and have two dimensions, the vertical represents the time, and the horizontal one represents the role of the involved agents in the interaction. Several of the main interaction protocols are briefly described in the following part:

- **FIPA Request.** This protocol is used when an agent asks another agent to perform an action. The message receiver can accept or reject the request, but if it accepts it, it must perform the

action and reply to the sender back when the action is finished.

- **FIPA Query.** This protocol is used when an agent asks another one about some type of information. Upon receipt of the message, the receiver message can respond with the information, a fail response, with a not-understood message, or directly rejecting the request. The agent must always include the reason the request has been rejected.
- **FIPA Contract Net.** In this protocol, the initiator agent wants one to many agents to perform an action. For that, the agent asks other agents for offers by using Call For Proposal (CFP) messages which contains the action to perform and some other preconditions. The agents to which the offer request was sent message back the sender with the offer conditions. The initiator agent then study the offers and chooses the best suitable ones while rejects the rest. In figure 3.4 we can observe a AUML diagram of the protocol.

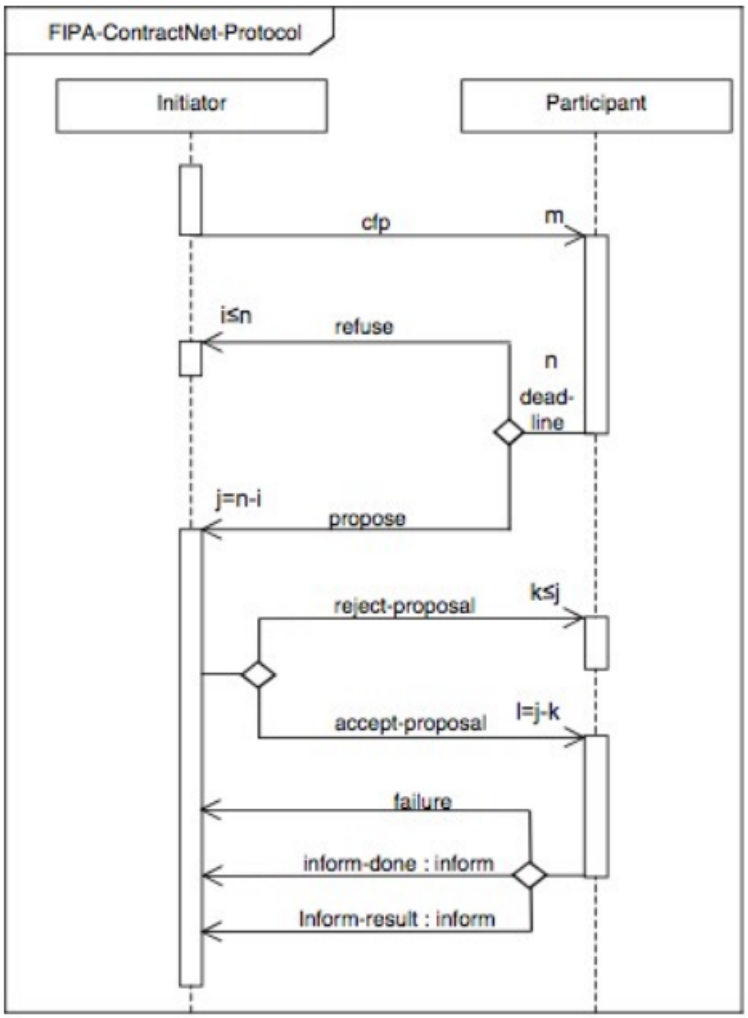


Figure 3.4: Contract Net Interaction Protocol AUML

Source: *Contract-Net Specification* [13].

Once defined FIPA standard, the following section describes JADE, a FIPA compliant platform directly related with the BSc thesis development.

3.1.2. Java Agent Development Framework (JADE)

JADE is a software framework fully implemented in the Java language. It simplifies the implementation of multi-agent systems through a middleware that complies with the FIPA specifications and through a set of graphical tools which support all life-cycle phases. One of the main characteristics of JADE-based multiagent systems is the ability of its agents to be **distributed** across different machines. The configuration of these agents can be controlled via a remote GUI (Figure 3.5). In addition, the configuration can be changed at run-time by moving agents from one machine to another when it is required. JADE can also be executed in many Operating Systems, as it is constructed in multi-platform language, Java.

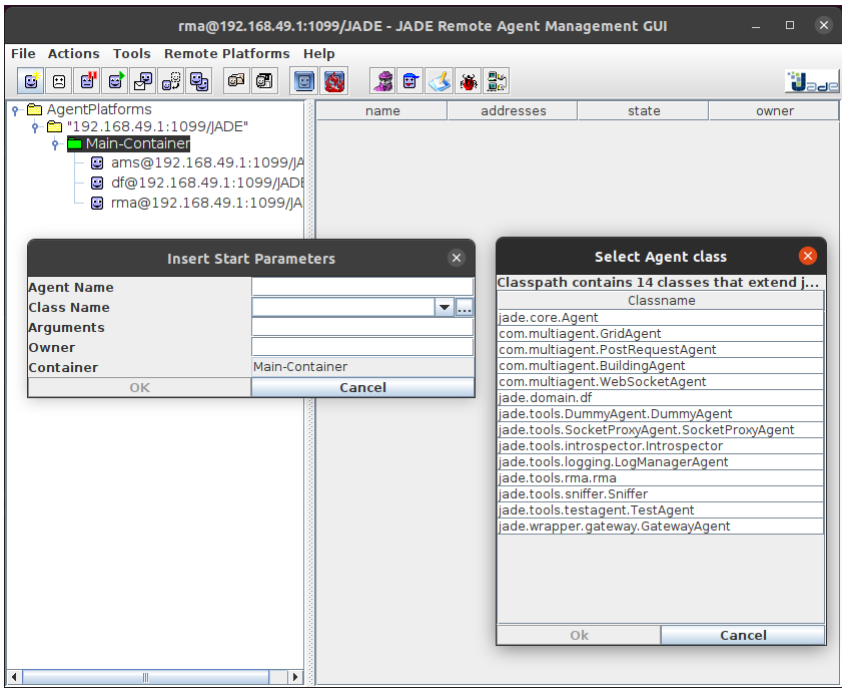


Figure 3.5: JADE Graphical User Interface.

The communication architecture of JADE offers flexible and efficient messaging, where it creates and manages a queue of incoming ACL messages, private to each agent. Agents can access their queue via a combination of several modes: blocking, polling, timeout and pattern matching based. JADE is FIPA complaint, and it implements the specifications previously mentioned in sections 3.1.1, 3.1.1. Agents live on top of a **platform** that provides them with basic services, such as the message delivery service, the Directory Facilitator (DF) and the Agent Management System (AMS). A platform is composed of one or more **containers**. These containers can be scattered among different machines, which allows the distribution and the scaling of multiagent systems. In this sense, in the platform there exists a special container called **Main Container**. This container can also contain agents, but differs from other containers as:

- It must be the first container to start on the platform, and all other containers register to it at bootstrap time.
- It includes two special agents:
    - The AMS that represents the authority in the platform and is the only agent able to

perform platform management actions such as starting and killing agents or shutting down the whole platform (normal agents can request such actions to the AMS).

- The DF that provides the Yellow Pages service where agents can publish the services they provide and find other agents providing the services they need.

Furthermore, JADE allows agent mobility between containers. Before starting the process, agents check the availability of the container, save their state and resume their execution on the destination container. In JADE, an agent can be in five different states as observed in the Figure 3.6: active, waiting, initiated, suspended and in transit.

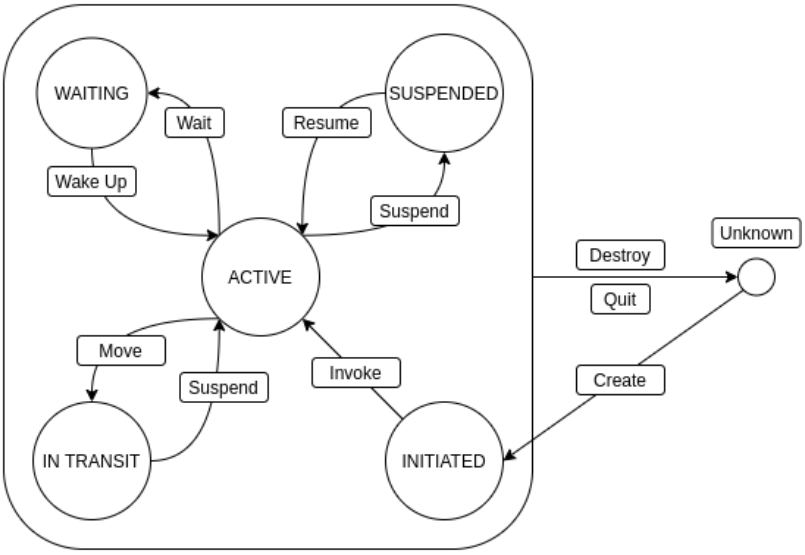


Figure 3.6: JADE Agent State Machine.

JADE offers a new way of programming based on behaviors, in which a goal is defined by a series of tasks or behaviors. JADE only allows the execution of a behavior in each moment of time, and each agent has two behavior queues, the queue of active behaviors and the queue of the blocked ones. For behavior scheduling, JADE uses a Round Robin (RR) algorithm, handling all behaviors with the same priority. An agent finishes its execution process when there is no behavior left in none of its queues or when the agent forced to finish. Figure 3.7 shows the life-cycle of a JADE agent. Once multiagent systems have been addressed, the following section is going to describe Smart Grids, a new way of energy distribution in which Renewable Energy Sources (RESs) take the lead.

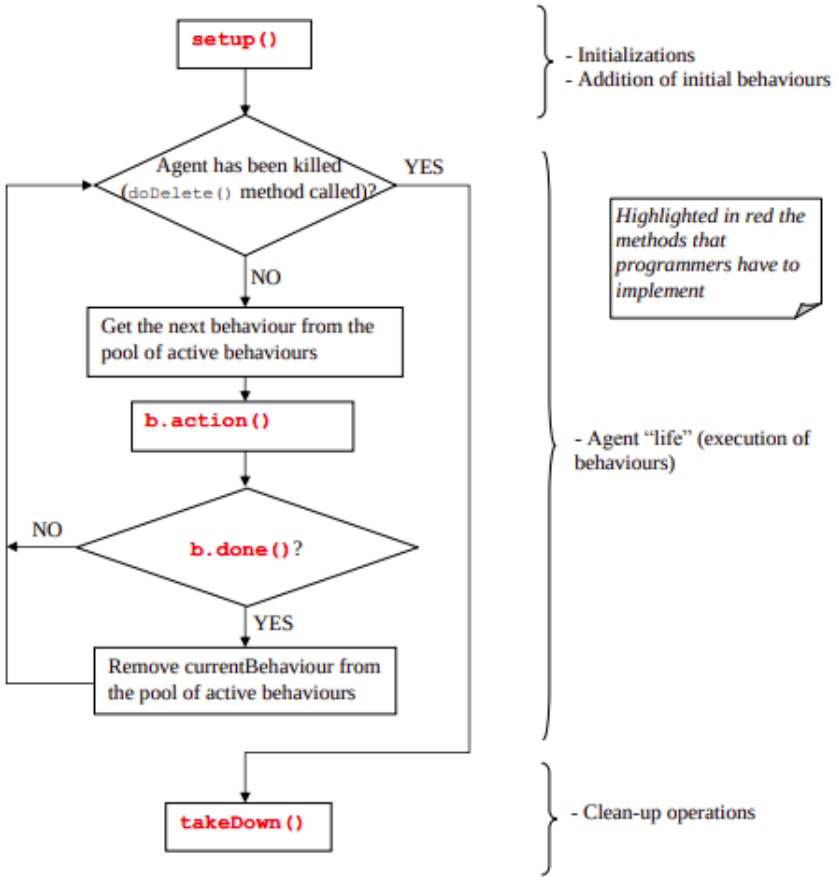


Figure 3.7: JADE Agent Flow Control

Source: *Jade Programming Tutorial [21].*

3.2. SMART GRIDS AND DISTRIBUTED ENERGY MANAGEMENT SYSTEMS

The energy scene is experiencing accelerating change. Centralized Energy Management Systems (EMSs) are being replaced by distributed ones, facilitated by advances in power system management and information and communication technologies. This is due to the growth of concepts such as Smart Grids, Distributed Energy Resources (DERs), Renewable Energy Sources (RESs) and Demand Side Management (DSM). Over the last decade, research advances in DERs and RESs as well as financial incentives (Feed-in Tariffs (FiTs), Power Purchase Agreements (PPAs), etc.) have lead to a large deployment of PV generators into the electric grid, with a worldwide capacity growing from 1.3 to 139 GW between 2000 and 2014 [43]. PV generators are usually placed on low and medium voltage networks. With respect to the voltage of the grid, there are three main types:

- **Low-voltage grid.** The low-voltage network is the part of power distribution which carries the energy from distribution transformers to electricity meters of end customers. Most modern low-voltage grids are operated at Alternating Current (AC) rated voltage of 100–110 or 220–240 volts.
- **Medium-voltage grid.** The medium-voltage network is the part of power distribution in charge of connecting high-voltage and low-voltage networks and operate from 1000 volts to

35 kilovolts (kV) using AC. Only large consumers are fed directly from distribution voltages, as most utility customers are connected to low-voltage grids.

- **High-voltage grid.** The high-voltage network is the part which connects generation stations with the mid-voltage network. It is used to transport energy over long distances and to feed a few customers. Its voltage is above 35kV.

In the United Kingdom (2007), units with a maximum capacity of 4 kW represented 21% of the total PV capacity [29]. Moreover, the steady decrease of production costs promotes PV self-consumption. Smart Grids, as an umbrella for all these new concepts and more, aim to improve the electricity grid performance and democratize electric production by embracing energy producers across every grid type.

Smart Grids are bidirectional electrical networks that can integrate the behavior of its components to ensure a sustainable EMS for both the energy consumers and the producers. Its goal is to improve the electric power infrastructure by integrating more energy sources in the context of Distributed Generation (DG). The grid has three main components: energy producers, consumers or those who fulfill both roles simultaneously, the prosumers. In this sense, DERs and RESs (hydrogen, PV, wind energy, etc.) are concepts of paramount importance to increase the penetration of these grids. Other objective of Smart Grids is increasing self-consumption and self-sufficiency rates. Self-consumption refers to the process, as a producer, of consuming directly your own energy production. In Equation 3.1, B represents the PV production that is not locally consumed, but sent to the grid, and C is the production locally consumed. The self-consumption rate (SC) is defined as the quotient between the locally consumed production (C) over the total production (B+C) [39].

$$SC = \frac{C}{B + C} \tag{3.1}$$

The self-consumed part relative to the total load is also a commonly used metric. In Equation 3.2, A represents the net electricity demand and C represents the PV power locally consumed. The self-sufficiency rate (SS) is defined as the quotient between the production locally consumed (C) over the daily net load (A+C) [23].

$$SS = \frac{C}{A + C} \tag{3.2}$$

Figure 3.8 shows a schematic outline of the power profiles of on-site PV generation and power consumption. As previously mentioned, A and B are the total net electricity demand and generation respectively, whereas the overlapping part in area C is the PV power that is utilized directly within the building. It also points out when load shifting and energy storage for a higher self-consumption rate (Figure 3.1) could be applied.

A strategy to improve the self-consumption rate is to use an energy storage system to adapt the production curve to the consumption needs, as seen in Figure 3.8, however, this alternative has the disadvantage of the initial cost, as the storage can cost from 150 to €1200/kWh depending on the technology [35]. Another strategy is **collective self-consumption**, where several consumers share the production of their own RESs production, most commonly PV production, on a low-voltage grid. They sign mini-PPAs in which the consumer pays a lower price for the energy and the provider

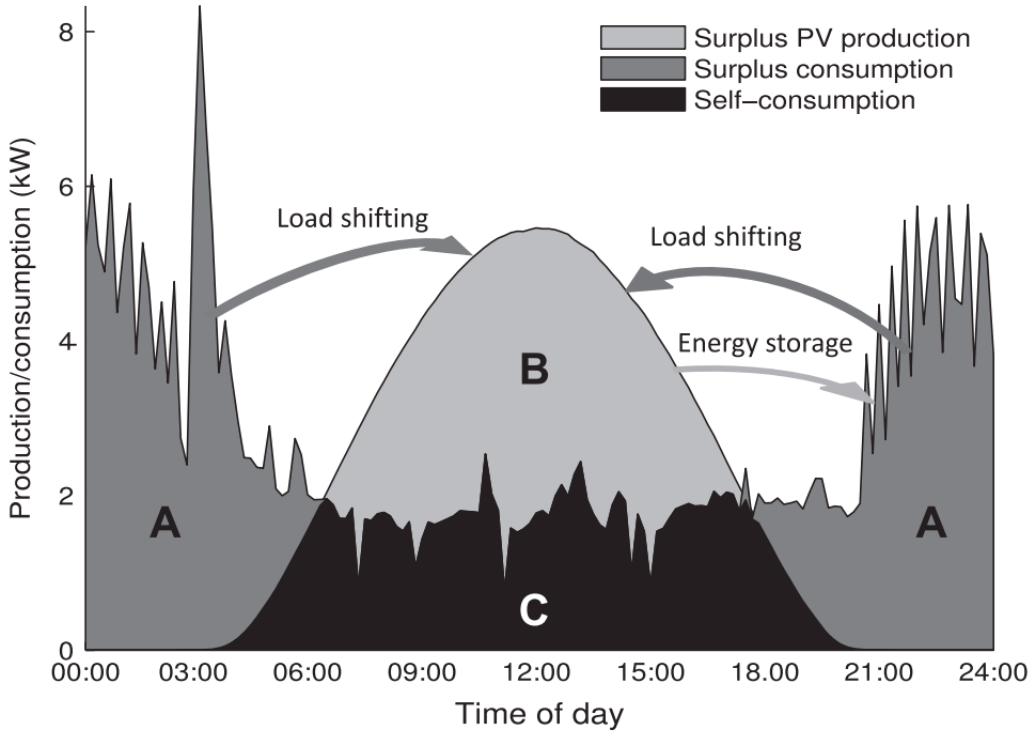


Figure 3.8: Schematic outline of daily net load (A + C), net generation (B + C) and absolute self-consumption (C) in a building with on-site PV. It also indicates the function of the two main options (load shifting and energy storage) for increasing the self-consumption.

Source: *Photovoltaic self-consumption in buildings: A review* [23].

receives a return on his/her investment. Projects of collective self-consumption within one building have also been developed [11]. A slightly different approach is **crowdfunding**, in which people come together and crowdfund a production system on their own buildings instead of being billed by a concrete producer. For this strategy, buildings need smart meters in every apartment, which can accurately measure and bill for the consumed electricity. The cooperatives/organizations which these people form are called Energy Communities (ECs). An EC can be understood as a way to "organize" collective energy actions around open, democratic participation and the provision of benefits for the members and/or the local community [6]. In the context of collective self-consumption, an important concept is the architecture of an Energy Management System.

On a **decentralized architecture**, there is no central supervision, as each element optimizes its local and individual behavior. Moreover, power flows bidirectionally and self-sustainable micro-grids can coexist along with the high voltage generators, as Figure 3.9 shows. These methods principally use MAS, in which agents act autonomously and independently of each other. This approach empowers energy communities to reduce costs and electric systems to scale.

On a **centralized architecture**, a few power plants generate high voltage power, which is converted by transformers to produce medium and low voltage power as shown in Figure 3.10. In this architecture, a single supervision system gathers the data to solve an optimization problem and afterwards sends clear instructions to each participant on the grid. Moreover, power flows unidirectionally and producers need to predict the consumption in case of load spikes.

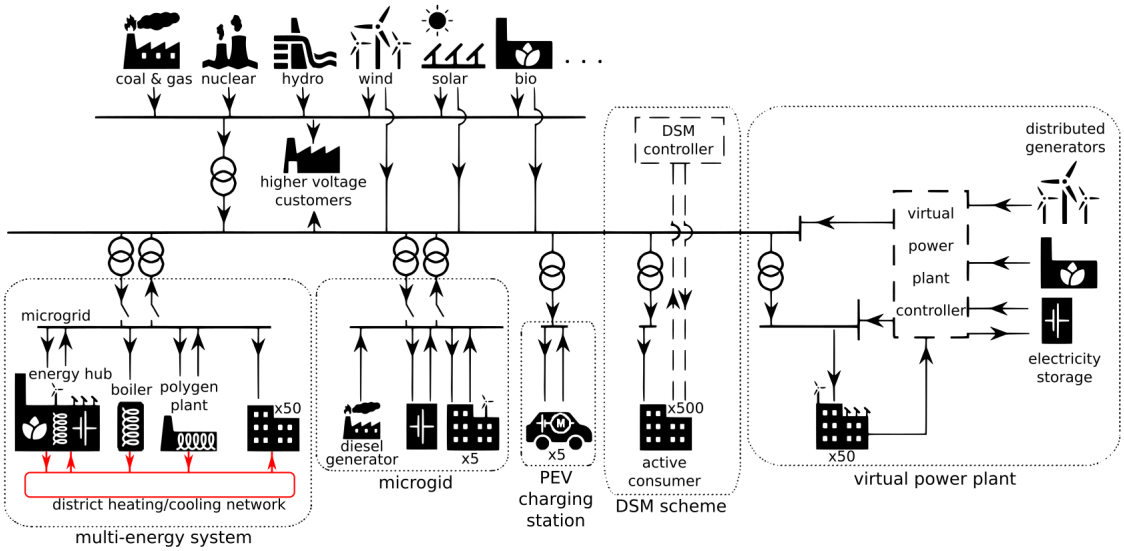


Figure 3.9: Simplified conventional decentralized architecture.

Source: *Towards the next generation of smart grids* [17].

Both in centralized and decentralized architectures, optimization on the energy distribution process is needed and sought. There are a myriad of approaches to this topic, some of them are the following:

- **Meta-heuristics.** Swarm intelligence algorithms and evolutionary algorithms have been used for DSM, for example, to reduce energy costs in a house [45] or for electric vehicles charge scheduling [40].
- **Fuzzy-logic.** In energy management systems, fuzzy logic has been useful to deal with the uncertainty of renewable source production [34].
- **Dynamic programming.** In energy management, dynamic programming has been particularly used for storage system management and charge scheduling [10].
- **Game theory.** Game theory has been used along with MAS in energy management. Each participant defines a strategy to maximize a concrete criterion. The participants can be *cooperative*, exchanging information, or not. In general, Game Theory is applied by defining games through a pricing system, and thus it is easily applicable to real micro-grids. Other approaches have emerged, such as dynamic pricing systems to reduce electricity costs [31] or non-cooperative game frameworks for a solar micro-grid including consumers, PV producers, and a battery, which reaches a Nash equilibrium [26]. This situation represents collective self-consumption.

To encourage Renewable Energy Sources and Distributed Generation, financial formulas have been developed. The most used ones are the following:

- **Distributed Power Purchase Agreements (PPAs)**. In this formula, ECs, businesses, governments or other type of entities can purchase energy directly from the generator. E.g, in France, PV generators can sell energy to each others as long as they are part of the same legal entity [11]. The advantage of PPAs is that consumers pay lower price for the energy.
- **Net metering**, where the excess solar electricity is remunerated via reverse metering or credits

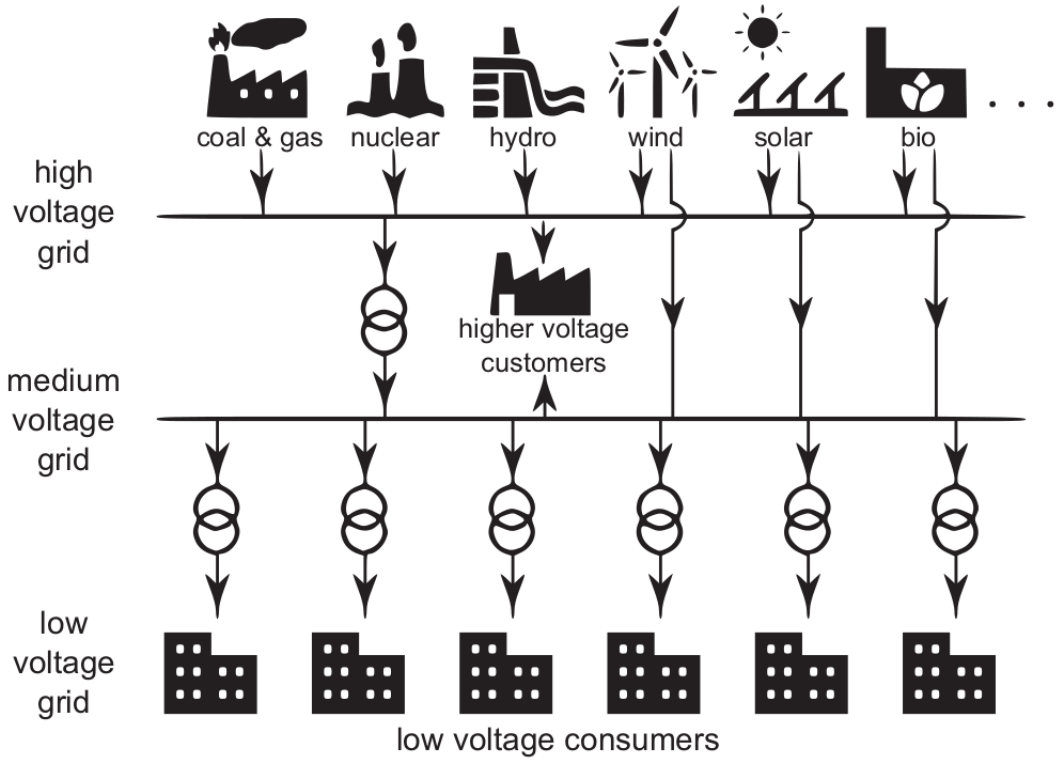


Figure 3.10: Simplified conventional centralized architecture.

Source: *Towards the next generation of smart grids* [17].

for future consumption. The advantage of net metering is that it boosts high levels of PV installations deployment. Some EU countries have net metering schemes in place.

- **Feed-in Tariffs (FiTs)**, economic incentives designed to accelerate investment in renewable energies. They offer cost-based compensation to renewable energy producers, providing price certainty and long-term contracts to finance renewable energy investments. FiTs often include a price *regression* in order to encourage a technological cost reduction [9].

This section has presented an overview on multiagent systems, intelligent agents and in JADE, a framework to build MASs. On the other part, self-consumption, Demand Side Management using different optimization methods and architectures, and RESs financial formulas have been explained. These concepts are the basis for the development of this BSc thesis. The next section describes the used resources and the methodological followed throughout the thesis.

CHAPTER 4

Methodology

In this chapter, the methodologies employed throughout the entire BSc thesis are introduced and described in detail. In particular, Scrum has been applied as the project management methodology. Along with it, an iterative and incremental software development methodology has been carried out, where the product is developed in several increments and each increment is executed in iterative cycles. Moreover, the employed software and hardware tools are going to be described.

4.1. AGILE METHODOLOGIES

Developing software is not a simple task and usually involves changing requirements and project specifications, which could lead to failure if no procedure is used to structure the project. To tackle this topic, agile methodologies emerged, with the goal of simplifying the intrinsic complexity of project development in this ever-growing complex world. During the last few years, the number of people, teams and organizations using them have increased considerably [20]. These methodologies try to satisfy the stakeholders by means of the twelve principles defined in the Agile Manifesto [16], which are condensed into four core values. These values are:

1. Individuals and interactions over processes and tools.
2. Working software over comprehensive documentation.
3. Customer collaboration over contract negotiation.
4. Responding to change over following a plan.

Agile practices centers on teamwork, by allowing teams to work on a project and to make modifications and changes in product development, in order to achieve the project's goal efficiently. The agile projects are usually divided in short duration increments. Each increment produces an operational prototype, which is revised with the client. In this context and according to the *14th annual state of Agile report* [1], Scrum [38] is the most applied agile methodology with a 58% of the total number of organizations using it, as shown in Figure 4.1.

From our perspective, Scrum is a good fit for this BSc thesis, as it consists of clearly differentiated parts which can be improved throughout the increments and finally integrated. Moreover, there is a lot of literature about it. In our case, Scrum has been adapted to a one-person development team. Hereafter, Scrum is going to be explained briefly.

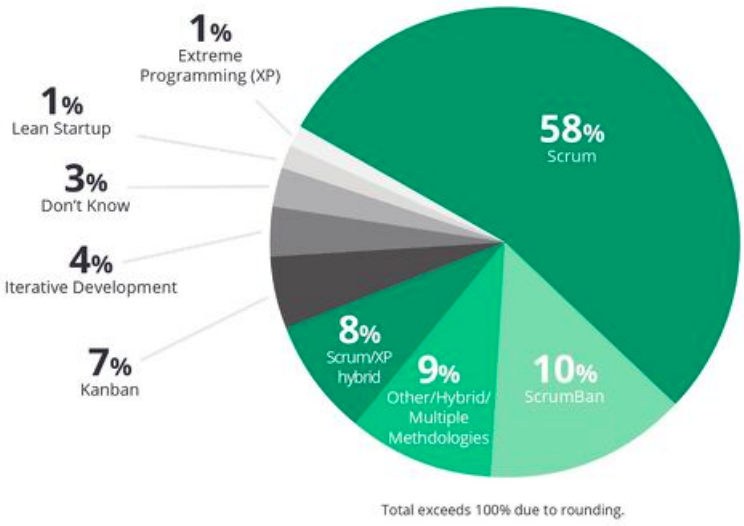


Figure 4.1: Most applied agile methodologies
Source: *The 14th annual state of agile survey* [1].

4.2. SCRUM

Scrum is an agile methodology for software development created by Jeff Sutherland along with Ken Schwaber in the early 1990s. It was created with the goal of simplicity in mind and creates opportunities from improvement, with the final purpose of being able to produce better quality products, or software in this case. Scrum, as an agile methodology, employs an iterative and incremental approach to optimize predictability and control risk. Concisely, Scrum requires a Scrum Master to facilitate an environment where:

1. A *Product Owner* orders the work for a complex problem into a *Product Backlog*.
2. The *Scrum Team* turns a selection of the work into an Increment of value during a *Sprint* (A small period of time between 1 and 4 weeks).
3. The Scrum Team and the clients inspect the results and adjust the plan for the next Sprint.
4. The three previous stages are repeated.

This life-cycle is shown with a higher level of detail in Figure 4.2. Each element of the methodology serves a specific purpose which is essential to the overall value. Subsequently, these elements are going to be explained in detail.

4.2.1. Scrum Artifacts

This section explains the Scrum artifacts which form part of the Scrum events, the Scrum Team and its composition and the User Stories and its format.

Product Backlog

The Product Backlog is a list of prioritized user stories pending to be included on a Sprint. This list is created by the Product Owner, but its content can be also negotiated between the Product Owner

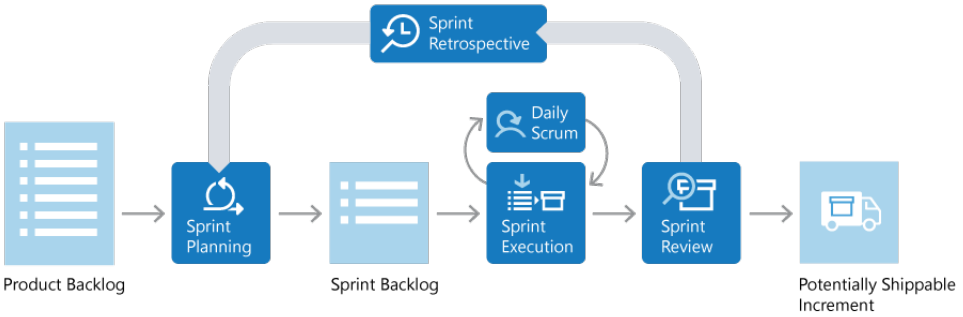


Figure 4.2: Scrum life cycle

Source: https://www.visualstudio.com/es/learn/what-is-scrum

and the development team. It is important to highlight that the list is always open to change.

Sprint Backlog

The Sprint Backlog is a subset of items contained in the Product Backlog. It is composed by the user stories to be developed during the Sprint period, and its content cannot be modified except for critical changes in the Product Goal. In such cases, the Sprint is cancelled.

Increment

It is a concrete stepping stone toward the Product Goal. Each increment is additive to the previous ones and verified altogether, so they must be usable. Multiple increments may be created within a Sprint and must meet the definition of done established by the Scrum Team.

4.2.2. Scrum Events

The Sprint is a container for all the events. Each event is designed to inspect and adapt Scrum artifacts. These events are the following.

Sprint

The Sprint is the heartbeat of Scrum, where ideas are transformed into value. These are fixed length events of one month or less, where one Sprint starts immediately after the conclusion of the previous one. During the Sprint, the Product Backlog can be refined and scope modifications may arise as the knowledge increases. Only the Product Owner has the authority to cancel the Sprint, but that is not possible unless the Sprint Goal becomes obsolete. In this project, each Sprint took a month.

Sprint Planning

The Sprint Planning is a meeting dedicated to plan the work to be performed during the Sprint span of time. The Product Backlog serves as a sack from which the most priority items are chosen. These items, in agile methodologies, are defined by the requirements, and they are known by the name of user stories. A user story [8] is a brief description of the functionality the stakeholder demands, concretely in this project, a software functionality. All these items form a Sprint Goal, which must

be valuable for the stakeholders. All Scrum Team must attend this meeting, but other people may also be invited. In this project, the Sprint planning took place every month.

Sprint Review

The Sprint review is an informal meeting held at the end of the Sprint to analyze the increment and adapt the Product Backlog if needed. During the meeting, the work done can be discussed between the different members of the team. In this project, the Sprint Review occurred just before the monthly Sprint Planning.

Daily Scrum

It consists of daily meetings with a short duration where the team can synchronize their activities for the next 24 hours. This is done by inspecting the work done in the previous day. In this project, this meeting was substituted by at-demand meetings with the team as well as biweekly meetings.

4.2.3. Scrum Team

The core unit of Scrum is a minimal team of people, the Scrum Team. This team usually is self-organized, cross-functional and must have an optimal size, typically ten people or fewer. The team model in Scrum is designed to deliver value and to optimize flexibility, creativity, and productivity. In this team, there are clearly differentiated roles, which have been assumed by the members of the project.

Roles assumed by the student

- **Development team.** It is a team of professionals in charge of creating value and increments throughout the Sprint. Moreover, they are also in charge of adding features to the Backlog and adapting their plan towards the Sprint Goal.

Roles assumed by the BSc thesis directors

- **Product owner.** The Product Owner is responsible for maximizing the value of the product resulting from the work of the Scrum Team. He must be able to delegate the responsibility to others and represents the needs of the stakeholders. This person is also in charge of establishing the user stories, assigning them a priority and adding them into the product backlog. The Product Owner must also have a clear perspective on the product which will be developed, and must transmit it to the development team. This role is assumed by Luis Jiménez Linares.
- **Scrum Master.** The Scrum Master is accountable for the Scrum Team's effectiveness, enabling the team to produce value. This person is also responsible for ensuring the Scrum events take place at the right time and helps the Product Owner to manage the Product Backlog. Moreover, the Scrum Master also serves as an intermediary between the stakeholders and the team to remove barriers and comprehend and implement the methodology. The role is assumed by Luis Rodríguez Benítez.

4.2.4. User Story

User stories are short descriptions which express client needs and requirements, and their use is common when working with agile methodologies such as Scrum. When working with user stories, the acceptance criteria needs to be defined, to verify that they are really completed. User Stories are prioritized by the customer (or the Product Owner in Scrum) to indicate which are most important for the system and will be broken down into tasks and estimated by the developers. Technical-only stories are also valid, but they are more unusual. In Table 4.1 the User Story format used on this BSc thesis is shown.

User Story		
Name		
Sprint	**Priority**	**Effort**
1	High	4 days
Description	A brief description of the user story	
Acceptance criteria	Requirement/s to mark a user story complete	
Tasks	Task list to achieve the user story goal	

Table 4.1: User Story format.

Once defined Scrum and user stories, the next section is going to explain in detail the development methodology applied on the project development.

4.3. DEVELOPMENT METHODOLOGY

The development methodology is used by software developers and engineers to write code, helping them in the process and giving some guidelines and standards when developing information systems in order to maintain a high quality code by reaching agreements. In this thesis, the iterative and incremental software development methodology is used. In this type of methodology, the project is divided into several matching blocks, with the temporal blocks defined in Scrum. Each iteration of the iterative and incremental methodology can be considered as a short project which adds value to the final product. For each of these iterations, all life cycle phases must be executed, incorporating analysis, design, coding and testing phases. This is the iterative part of the methodology. Furthermore, in each iteration the team must include new goals, requirements, and new knowledge from previous iterations. This is the incremental part of the methodology. The result is a new slice of functionality ready to ship to the client. Figure 4.3 represents the iterative and incremental methodology flow.

Some advantages of using these methodologies over other canonical ones are:

- The client/stakeholder is involved in each iteration of the development.
- Functional results are obtained from the first iterations.
- The tolerance to change is higher, as the methodology encourages it.
- It is easier to make architectural changes, especially in early development phases.

Some disadvantages of using these methodologies are the next ones:

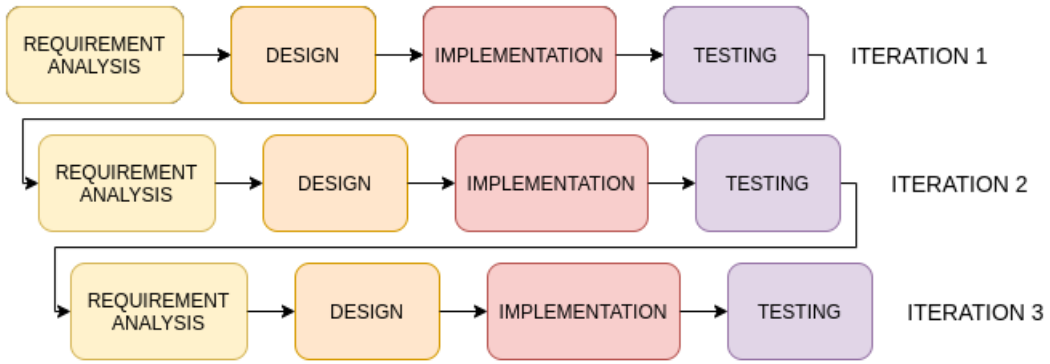


Figure 4.3: Iterative and incremental methodology flow
Adapted from *http://www.technologyuk.net*.

- The result of each iteration must be a functional product and must satisfy the needs of the stakeholders and the clients.
- The iteration's duration is fixed, so an accurate estimation of the time an increment requires is needed.
- An increment might be complex and might not fit on a single iteration.

Taking all this into account, this type of methodology was chosen for this project.
Subsequently, the different resources used in the development of this project are going to be detailed.

4.4. REQUIRED RESOURCES

In this section, the different hardware and software resources that were employed during the development of the BSc thesis are exposed.

4.4.1. Hardware resources

The development of this project has been done using a Lenovo ThinkPad® L380. Its main features are:

- 16 GB of Random Access Memory (RAM)
- Intel® Core™ i7-8550U CPU @ 1.80GHz
- 256 GB of Solid State Drive (SSD)

4.4.2. Software resources

This section includes the used Operating System, the programming languages, libraries, frameworks, and different development and documentation tools used along the project development.

Operating System

Ubuntu 20.10 [1]**.** Ubuntu is a popular Open Source (OS) that has been used in the development computer. It is based on GNU/Linux, and it is well-known for its simple configuration and user-friendly interface, nonetheless, it is also used on servers, microcomputers, etc.

Programming Languages

- **Python 3 [24].** Python is an interpreted high-level programming language which is designed to be simple on its syntax and its interfaces. Moreover, it is Object-Oriented (OO) and dynamically typed language. In the last few years, its user base has grown a lot, thus becoming the fourth most popular programming language in the *2020 Stack Overflow developer survey* [33]. It is multi-platform, and it has myriad applications, from scripting to data science. Furthermore, it has an enormous OS community, which has helped to build web frameworks such as Flask, Django or FastAPI.
- **Javascript [14].** Javascript is an interpreted high-level programming language firstly designed to add dynamism on the client side of Web Navigators. It is Object-Oriented, dynamically typed, and it supports prototype, functional, imperative and event-driven based programming. Nowadays, it is the most popular programming language [33] and it is also used on the server side and other types of services. Analogous to Python, it has a vast amount of frameworks. React, a popular web framework, is built on top of it and it has been used for this thesis.
- **HTML [32].** HyperText Markup Language or HTML is the most popular Markup Language for web development. Web browsers engines have the ability to transform HTML documents into visual representations for the users to interact with.
- **SASS [30].** Syntactically Awesome Style Sheets or SASS is a style sheet language normally used along with HTML that creates rules for how a page element should look on screen. It is a compatible superset of CSS, which has been used as well in this project.
- **SQL.** Structure Query Language, or SQL, is a declarative Domain Specific Language (DSL) designed to manage data held in a relational Relational Database Management Systems (DBMS). It is widely used in commercial applications, as it is especially useful to handle structured data.
- **Java [3].** Java is a high-level programming language designed to be general-purpose and multi-platform. It is OO, however, the last versions deeply support other types of programming paradigms such as the functional. Java applications are compiled to Byte-code that can run on the Java virtual Machine (JVM), an interpreter which can run on countless computer architectures.

Main libraries and frameworks

- **Pvlib** [2]**.** Pvlib is a community-supported tool that provides a set of functions and classes for simulating the performance and production of photovoltaic energy systems. It is used to simulate the solar panel's energy production.

[1] https://ubuntu.com/
[2] https://github.com/pvlib/pvlib-python

- **JADE [5].** JADE is a software framework fully implemented in the Java language, which simplifies the implementation of multiagent architectures and also complies with the FIPA specification (Foundation for Intelligent Physical Agents) by using an ad-hoc designed middleware. It also provides peer to peer agent communication based on asynchronous messaging and yellow pages services, among other things. A JADE system can be distributed across machines.
- **React [4].** React is a declarative JavaScript library for building user interfaces, which makes painless the creation of an interactive User Interface (UI). It is a Component-Based library in which each component manages its own state, therefore the state can be out of the Document Object Model (DOM).
- **Redux [4].** Redux is a library to manage the state shared between the application components. Redux construction was inspired by a unidirectional data flow pattern, the Flux Pattern.
- **FastAPI** [3]**.** FastAPI is a web framework to construct APIs (Application Programming Interfaces) based on Python type annotations. Its simplicity and its robustness make it adequate for developing scalable APIs. Furthermore, it is based on open standards such as OpenAPI and the JSON Schema, and it automatically generates documentation.
- **SQLAlchemy** [4]**.** SQLAlchemy is an Object Relational Mapper (ORM), a component that provides the **data mapper pattern**, where classes can be mapped to the database in open-ended, multiple ways. It is designed for efficient and high-performing database access. A database migration tool has been used along with SQLAlchemy, **Alembic** [5].
- **Spring** [6]. Spring Boot is a Java framework that makes it easy to create stand-alone, production-grade Spring based Applications.

Development tools

- **VSCode** [7]**.** VSCode is a multi-platform source-code editor made by Microsoft. It has features such as debugging, syntax highlighting, intelligent code completion, code refactoring, and embedded Git commands. One of its advantages is its high degree of versatility thanks to the extensions developed by the community.
- **TMux [28].** TMux is a terminal multiplexer for Unix-type systems, which allows the user to divide its terminal into multiple sections and generate multiple independent sessions.
- **Git [7].** Git is a Free and Open Source Software (FOSS) Distributed Version Control System (DVCS) designed to track file changes and coordinate work between teams. It is mainly used to keep track of code changes.
- **GitHub** [8]**.** GitHub is a web-based hosting for Git repositories which provides access and control to them and adds extra features on top of that, such as bug tracking, pull requests, task management, and wikis.

[3] https://fastapi.tiangolo.com/
[4] https://www.sqlalchemy.org/
[5] https://alembic.sqlalchemy.org/en/latest/
[6] https://spring.io/projects/spring-boot#overview
[7] https://code.visualstudio.com/
[8] https://github.com/

- **Virtualenvwrapper.** [9] Virtualenvwrapper is a Python virtual environment tool that allows to create virtual environments easily and quickly to isolate project dependencies.
- **Database Browser for SQLite** [10]**.** Database Browser for SQLite is a visual and open source tool to create, design, and edit database files compatible with SQLite. It has been used to visualize the Database (DB) records.

Documentation tools

This section presents the software tools used to generate the documentation of this project:

- **LaTeX [22].** LaTeX is a document preparation system for high-quality typesetting. It is most often used for medium-to-large technical or scientific documents, but it can be used for almost any form of publishing. The entire project is written in LaTeX.
- **Overleaf** [11]**.** Overleaf is a collaborative cloud-based LaTeX editor used for editing and writing scientific documents. It includes internally Git as a DVCS, being compatible with GitHub to control the documentation process. It has been used to write this BSc thesis.
- **Draw.io** [12]**.** Draw.io is a highly versatile online software for making diagrams. It has been used to design most of the project diagrams and graphics.
- **Tables generator.** [13] Tables generator is a web page which allows generating LaTeX tables in a What You See Is What You Get (WYSIWYG) editor.
- **Universidad de Castilla-La Mancha (UCLM) TFG Template [37].** This template has been used as a guide to prepare this BSc thesis.

Figure 4.4 shows the relevant software resource used for the development of this BSc thesis.

4.4.3. BSc thesis costs

The final cost of this project is presented in Table 4.2. The estimation of human costs was calculated based on the UCLM remuneration of employment contracts for research projects [42]. Taking the *Tercera-O-I* category as a reference, which supposes an annual cost of €20.415.57; 7 working hours per day and 225 working days per year, the cost per hour is €12.96.

Concept	Hours	Cost (€)
Lenovo ThinkPad L380	—	1059
Project's development	205	2656.8
Project's documentation	53	686.88
TOTAL	**267**	**4428.6**

Table 4.2: Project total cost.

Once identified the different methodologies and tools used through the whole process of development, the next chapter presents the results of this project, detailing each one of the Scrum iterations.

[9] https://virtualenvwrapper.readthedocs.io/en/latest/
[10] https://sqlitebrowser.org/
[11] https://www.overleaf.com/
[12] https://app.diagrams.net/
[13] https://www.tablesgenerator.com/

Figure 4.4: Software resources used in the project development.

CHAPTER 5
Results

All the work done over this project is shown in this chapter. In the first place, the initial Sprint is presented, where the Scrum Team is set up and the initial planning of the project is designed. Throughout this section, all the Sprints and User Stories involved in the project are reviewed within the chronological order followed in the development.

5.1. SPRINT 0: INITIAL PLANNING

In this first phase, the main objective was to define the Scrum Team, the initial Product Backlog, the project plan and the temporal and cost planning. Once this Sprint was completed, the project started as settled in the temporal planning. The tasks associated with this initial Sprint are shown in Table 5.1.

Sprint 0 Tasks	
Task	**Estimation (h)**
Define the Scrum Team	0.5
Research about similar proposals	8
Proof Of Concept (POC) of key libraries for the project	7.5
Define the first Product Backlog	3
Generate the project plan	2
Generate the temporal plan	2
Generate the cost plan	2

Table 5.1: Sprint 0 Tasks.

Once the tasks were defined, and following the Scrum roles defined in Section 4.2.3, the Scrum Team was set up with the following structure:

- **Scrum Master:** Luis Rodríguez Benítez
- **Product Owner:** Luis Jiménez Linares
- **Development Team:** Enrique Cepeda Villamayor

After defining the Scrum Team, a Proof Of Concept (PoC) of the project key libraries was made. A PoC is a demonstration that a certain method or idea works, in order to demonstrate its feasibility. In software development, it is usually a small piece of software which shows the capabilities and the limits of a tool. In this concrete case, pvlib and OpenLayers libraries were tested before the start of the project.

Afterwards, the Scrum Team had several meetings to define the initial and main requirements. The requirements were converted into User Stories, which needed to be accomplished during the execution of the BSc thesis. The Scrum Team defined different User Stories and their estimated completion time using the **planning poker technique**. In this technique, the product owner explains a User Story and each member of the development team writes his estimation into a card. Nonetheless, in this case, all the members of the Scrum Team have participated in the estimation. It is important to note that the Product Backlog is a dynamic entity, and therefore, it evolves throughout the Sprints. The initial list of User Stories is shown in Table 5.2.

Initial Product Backlog			
ID	**User Story**	**Priority**	**Estimation (h)**
1	Design a component on a web application to define the Smart Grid members.	High	30
2	Develop a module to infer the energy production of a PV roof installation.	High	25
3	Design and develop a component to indicate the hourly building consumption of a building.	Medium	20
4	Develop an algorithm to fit the maximum PV panels on the roof-top area of a building.	Medium	15
5	Develop a multiagent system to simulate the energy distribution between the Smart Grid components.	High	30
6	Design of a dashboard component to visualize the energy distribution between the buildings.	Medium	15
7	Integrate the web dashboard and the multiagent system components.	Medium	15
8	Integrate the real address of a building in the application via its geographical coordinates.	Low	10
9	Development of a page to select the maximum amount of panels of a producer building	Low	10

Table 5.2: Initial Product Backlog.

The format and the details of a User Story are explained in Section 4.2.4. The evolution of the Product Backlog and the accomplished User Stories are defined throughout the following sections.
After defining the initial Product Backlog, the next step was to define the different Sprints that must be carried out and their associated User Stories (Table 5.3). Figure 5.1 shows the temporal planning of the project. This was the initial plan, therefore, additional tasks could be created to fix errors or to add functionalities not taken into account in the initial phases. Once defined key aspects of the development of the project, the development began with the first Sprint, detailed in the following section.

Sprint	**User stories**	**Estimation (h)**
0	—	25
1	1	30
2	2, 3	45
3	4, 5	45
4	6, 7	30
5	8, 9	20
—	—	**195**

Table 5.3: Sprints along with its associated user stories.

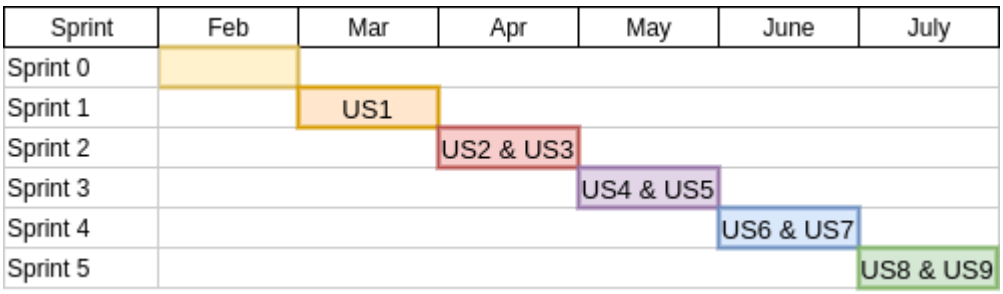


Figure 5.1: Project time planning

5.1.1. Sprint 1: Development of a component as a part of a web application to define the Smart Grid members

According to the project plan, the tasks associated with User Story 1 were executed. This Sprint aimed to obtain a web interface in which any building can be selected from a map to form part of a Smart Grid. For that, OpenLayers, OpenStreetMaps (OSM) and Overpass Application Programming Interface (API) were used, a set of tools that help to obtain the building geographical and geometrical features. Other issue related to the development of the web application was the framework to build it. When the POC for these parts was completed, a classical approach was followed, in which a basic web page was built using HTML and JS along with JQuery, but there exists a myriad of alternatives. Each framework has benefits and drawbacks, however, the application architecture and the used tools depend substantially on the chosen framework. After the Sprint planning meeting, Table 5.4 shows the first User Story.

User Story 1		
Design a component on the web interface to define the Smart Grid members.		
Sprint	**Priority**	**Estimated time**
1	High	30 hours
Description	Design an application component to specify the buildings members of the Smart Grid and define its role (consumer, producer and prosumer)	
Acceptance criteria	▪ Design the component visuals and develop it. ▪ Choose the web framework in which the web application is going to be built.	
Tasks	▪ The buildings can be selected and removed by clicking on the map. ▪ A list of the selected buildings must also be displayed. ▪ The building features such as the location or the role must be displayed. ▪ The user must be able change the role of the building in the grid.	

Table 5.4: User Story 1.

5.1.2. Selection of a web framework and a component library to build the web application

For this task, the decision was between building a Single Page Application (SPA) or a Multiple Page Application (MPA). As Figure 5.2 shows, when a user makes a request, a single-page application only reloads the necessary data. However, in the case of a multi-page application, the entire web page content is refreshed. For the first alternative, there were modern web frameworks such as React, VueJS or AngularJS to construct the web application, whereas any required external data could be

requested to an API. On the other hand, to construct a MPA, there exists frameworks such as Django, ASP.Net or Spring in which the Backend is the protagonist, and *HTML* templates work along with *Javascript* and *JQuery* to provide reactivity. Several factors resulted in the election of React as the Frontend framework:

- SPAs help to decouple the Frontend and the Backend.
- In React, the Frontend application logic is not managed on the DOM, which results in a cleaner code, as each component has its own state shared only if needed.
- It is built around the concept of a component, which leads to a higher degree of modularization.
- It performs better than HTML template approaches thanks to its Virtual DOM, which handles the state and DOM changes internally instead of being brought to the browser.
- It is a SPA framework with one of the highest community support.

Figure 5.2: Single Page Apps vs Multiple Page Apps lifecycle
Source: *Lvivity* [25]

Once React was chosen, several decisions were made:

- **Material UI**[1] was chosen as a component library. Material UI is a component library built to work with React, which facilitates the development of web application and personalized components. The use of component and style libraries is highly widespread, as they provide essential components with a common color palette, a common *Look and Feel*, and they are based on design guidelines (Material UI uses Material Design guidelines[2]). Moreover, its API is very intuitive, and its components are highly customizable, which allows to extend its behavior and its look to build a custom theme.
- A **component-based design** approach was going to be followed, where the functionality is divided by components that can be used in different parts of the application.

[1] https://material-ui.com/
[2] https://material.io/design

- **Create React App**[3] was chosen to build the project. This project uses *Babel*, *Webpack* and *ESLint* among other libraries to make the development of the React apps as transparent as possible.

5.1.3. Design and development of the building selection component

Once the framework and the component library were chosen, the first component of the web application could be built, the building selection component. For that, a map was going to be used, in which the user could select the grid buildings. For this task, OpenLayers was the chosen library to build it. OpenLayers is an Open Source JavaScript library which makes easy dynamic map building. Moreover, it is highly composable, as it has a system of compound layers to obtain the data shown on the map. Matthew Brown's project [4] was used as a base on how OpenLayers could be used along with React. The building selection map was mainly composed by two of these layers:

- A **Tile Layer**, which fetches multiple tile images to compose the whole map background. The source of the tile is **OpenStreetMaps**, a collaborative project to create an editable map of the world and whose geographical data is the main contribution. Its database is created by the users and multiple services such as Facebook, Apple, Microsoft, among others have used it.
- A **Vector Layer**, which allows rendering polygons on a map. These layers can obtain the polygon data from static sources (files, GeoJSONs, etc.) or dynamic ones in order to fetch building geometrical and geographical information. In this sense, **Overpass API** has been used. Overpass API is a read-only service that serves up custom selected parts of the OSM map data, acting as a database Backend. It has its own query language (OverpassQL) to optimize the response time, but it also supports XML format requests.

To support OpenStreetMaps main services, Overpass API has several third party services open to requests and available to the general public with different specifications [5]. Initially, one of the main server endpoints was used, however, several problems emerged. The first problem was the response time, as each request took 2 seconds, which consequently affected the **User Experience (UX)** when loading the buildings on the map. The second problem was that the server returned a 429 HTTP code when the number of request per hour rose, although the request limit was not exceeded. To solve this issue, a study on the latency and the response time of these requests took place. The goal was determining which was the best server to make the requests with the minimum latency and response time. Table 5.5 shows the three best server instances by their hardware specifications and by its geographical location, from which the best instance was selected.

[3] https://create-react-app.dev/
[4] https://github.com/mbrown3321/openlayers-react-map
[5] https://wiki.openstreetmap.org/wiki/Overpass_API#Public_Overpass_API_instances

Servers			
Name	**Endpoint**	**Hardware**	**Usage Policy**
Main Overpass API Instance	https://lz4.overpass-api.de/api/interpreter	4 physical cores 64 GB RAM SSD	< 10000 per day
French Overpass API Instance	https://overpass.openstreetmap.fr/api/interpreter	8 physical cores 16 GB RAM SSD	< 1000 per day per project
Kumi Systems Overpass API Instance	https://overpass.kumi.systems/api/interpreter	4 servers with 20 cores 256 GB RAM SSD each	No rate limit

Table 5.5: Overpass best server instances.

For the study, the average latency was obtained using *ping* command, whereas the average response time was obtained by simulating the original browser request using *curl* [6], repeating the request 100 times with *repeat* command, and finally obtaining the average response time and the standard deviation by using *st* [7] command. Table 5.6 shows the results, in which the French instance has the lower latency, whereas Kumi Systems Instance has by far the lowest response time. Therefore, the Kumi Instance was chosen to request the building data.

Servers		
Name	**Average latency (RTT) / Standard deviation**	**Average response time / Standard deviation**
Main Overpass API Instance	50.465ms / 19.279ms	2.318s / 1.601s
French Overpass API Instance	35.911ms / 20.158ms	1.345s / 0.129s
Kumi Systems Overpass API Instance	67.263ms / 20.091 ms	0.138s / 0.007s

Table 5.6: Latency and response time from Overpass servers.

The geographical and geometric data is given by Overpass, including data such as the area of the building. Figure 5.3 shows the map with the contour of the selected buildings drawn in purple and the rest in yellow. The streets, park icons, and more come from the Tile Layer which fetches the data from OSM.

[6] https://curl.se/
[7] https://github.com/nferraz/st

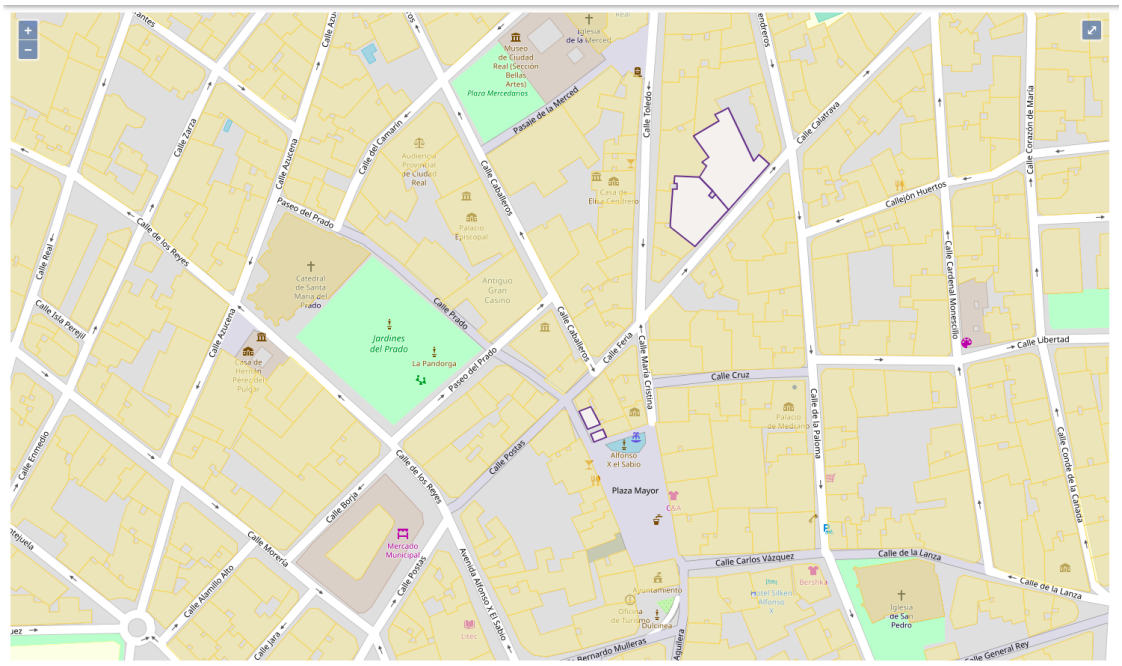


Figure 5.3: Map interface to select the building for the smart grid.

Regarding the list of selected buildings from the map, the elements were designed to focus the pertinent building on the map when clicked. They can also change their role and show their more relevant features: Their location, building area and role along with their unique identifier. Figure 5.4 shows an example of a building list component, composed by several building cards.

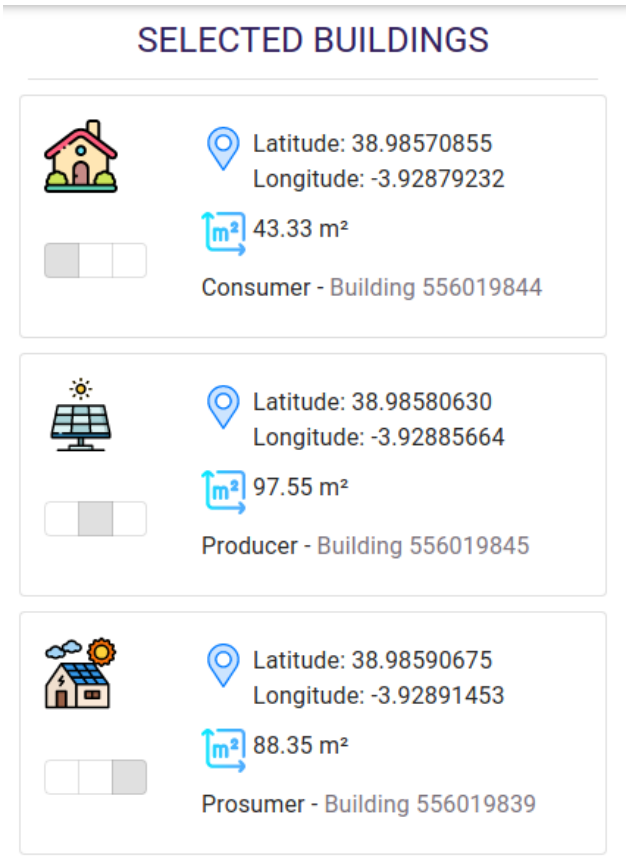


Figure 5.4: Example of a building list component

5.2. SPRINT 2: DEVELOPMENT OF AN ENERGY INFERENCE MODULE AND A COMPONENT TO INDICATE THE ENERGY CONSUMPTION OF A BUILDING

In this Sprint, two User Stories were accomplished, the development of a component inside the web application to indicate the consumption of a building, and the development of a module to infer the energy production of a certain PV panel roof-top installation. For the building consumption component, the goal involves the design and the build of the interface components, the introduction of a way of sharing logical state between all the components across the application, the addition of unique routes to transit to other pages easily and the creation of a color palette to create homogeneity between the components. On the energy production module, the goal involves to obtain real meteorological data, to get the altitude of a concrete geographical position and the inference of a certain amount of energy for a panel configuration. Tables 5.7 and 5.8 present the corresponding User Stories.

User Story 2		
Develop a module to infer the energy production of a PV roof-top installation		
Sprint	**Priority**	**Effort**
2	High	25 hours
Description	It consists on developing a module to infer the energy production of a PV installation on a certain period of time taking into account the total number of panels which a building can fit and the chosen PV panel technology among other variables.	
Acceptance criteria	▪ The module is capable of obtaining real meteorological data to infer the production. ▪ The module is prepared to admit any type of configuration adapted to a building roof-top. ▪ The module returns consistent results.	
Tasks	▪ Obtain real meteorological data for the inference. ▪ Obtain the real altitude of a certain geographical position for the inference. ▪ Infer the production of a certain amount of energy for a certain panel configuration.	

Table 5.7: User Story 2.

User Story 3		
Design and develop a component to indicate the hourly building consumption of a building.		
Sprint	**Priority**	**Effort**
2	Medium	20 hours
Description	Development of a component in which the average hourly consumption of a prosumer or consumer building can be detailed.	
Acceptance criteria	▪ The component should read the building data from the building selection component. ▪ The component should only show prosumer/consumer buildings.	
Tasks	▪ Find a way to share the state between the building selection component and the consumption one. ▪ Create a color palette to design the new components in a homogeneous and reusable manner. ▪ Add page routes to the project to add new pages to the project application. ▪ Design and build the UI.	

Table 5.8: User Story 3.

5.2.1. Development of a module to infer the energy produced by a PV panel installation

One of the essential parts of the application is the PV energy inference module. The module is in charge of inferring the energy production for a certain PV panel system on the roof-top of a building. For this reason, pvlib has been used (Section 4.4.2), whose core mission is to provide open, reliable, and benchmarked implementations of PV system models. A PV system employs solar modules, each of them composed by a certain number of solar cells, which generate electrical power in Direct Current (DC). These PV installations may go from ground-mounted to rooftop-mounted, and the mount may be fixed, or it may track the sun across the sky. Furthermore, there exists other components that come into play, such as the power inverter, which takes the DC generated by the PV panels and converts it into AC. In essence, there are a lot of factors that influence the system's output.

One key factor when designing a PV system is determining the panel orientation and inclination. In this project, PV systems were designed to always have a south orientation and a tilt equal to the latitude of the building. These type of installations are a good fit for PV systems on roof-tops. On the one hand, the latitude angle provides optimum energy power throughout the year. An increased tilt angle above the latitude increases power output production in wintertime; however, it decreases in the summer. On the other hand, the optimal orientation angle (or azimuth angle) is around 0 degrees, which may vary in case a greater amount of energy is needed in the morning or in the evening [15]. Figure 5.5 shows the selected configuration. It is important to note that these variables affected the way the module which extracts the maximum PV panels on a building roof-top area works.

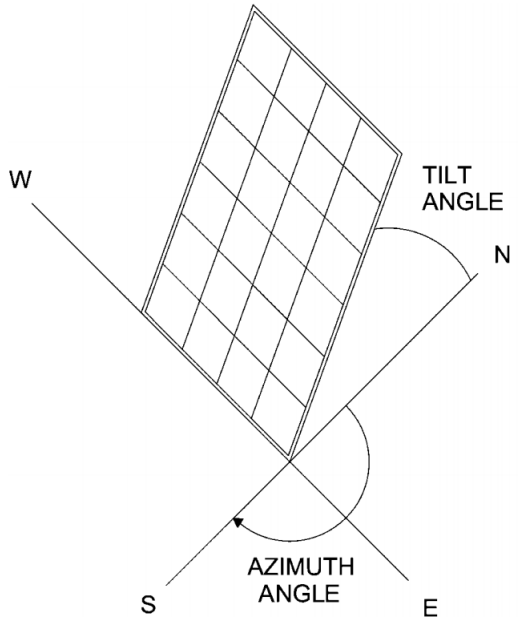


Figure 5.5: PV panel facing south at a fixed tilt

Source: *Solar Power in Building Design* [15].

Another important point is the panel selection. There exists a myriad of alternatives with different characteristics, but the most important ones are the efficiency (%), the voltage (V), and the wattage output (W). Taking these as the most relevant factors into account, *Silevo Triex U300* panel was

chosen to be part of every building PV system. This panel has a 17.9% of efficiency, its Voltage Open Circuit (Voc) is 69V and its Standard Test Conditions (STC) wattage is 300W. Figure 5.9 presents its more relevant characteristics.

Solar module	Triex U300	**Short circuit current (*Isc*)**	5.57A
Maximum power (*Pmax*)	300W	**Number of cells in each module**	96
Optimum voltage(*Vmp*)	57.5V	**Height**	1.586 m
Optimum current(*Imp*)	5.23A	**Width**	1.056 m
Open circuit voltage (*Voc*)	69.0V	**Area**	$1.68m^2$

Table 5.9: Parameters of the Silevo Triex U300 PV module.

Another element that plays an important role in the energy production of a PV panel is the solar irradiance to its surface, which can be divided into three variables:

- The **Global Horizontal Irradiance (GHI)**.
- The **Direct Normal Irradiance (DNI)**.
- The **Direct Horizontal Irradiance (DHI)**.

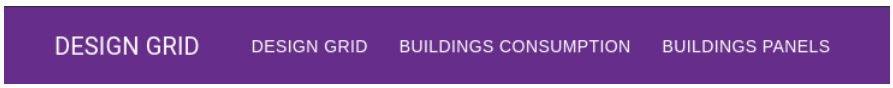
The module calculates these variables using real meteorological data obtained from the Global Forecast System (GFS)[8] in a concrete moment of time. Moreover, the altitude of a building has been obtained through **Google Elevation API** [9], as it provides the exact altitude of a geographical location, which affects pressure among other parameters. Other variables such as the solar position or the angle of incidence are inferred from the geographical position. The set of models which have been used to infer the energy production are from the Sandia National Laboratories [10]. These models are widely known and highly tested, and result in the DC energy that an installation produces in one hour. Finally, this energy is converted to AC energy via an ad-hoc inverter.
This was done because the process of selecting an inverter for a PV system is a difficult task, as there are endless options for numberless configurations. The inverter outputs the energy that a PV system can produce, and it is used as an estimation of a real PV system performance. However, the inverter does not have a 100% efficiency, the usual efficiency is 96%. Furthermore, *inverter clipping* can limit the amount of DC energy produced by the inverter. This problem occurs when the inverter reaches its maximum watt capacity. Additionally, it is important to make sure that a PV installation does not exceed the inverter maximum voltage, as the equipment can then be damaged. For these reasons, the maximum DC power of the PV system inverter is equal to the maximum DC power output of the PV system. Also, the maximum voltage of an inverter cannot be exceeded, and the PV system has its voltage limit set by the PV panel. It is for these reasons that the maximum voltage (and therefore the number of panels connected in series) is set to 1000V, the limit of the PV panel, as the inverter could be hypothetically substituted by other one with higher voltage tolerance.

[8] https://nomads.ncep.noaa.gov/txt_descriptions/GFS_doc.shtml
[9] https://developers.google.com/maps/documentation/elevation/overview
[10] https://www.sandia.gov/

Finally, the module was exposed by means of an endpoint on an API built using FastAPI framework. In addition to its forementioned virtues (Section 4.4.2), it provides the possibility of creating asynchronous endpoints and typed endpoint parameters, which helps to create a better test suite for it. After presenting the energy inference module, the next section describes in detail how, the component to indicate the consumption of consumers and prosumers, was built.

5.2.2. Development of the building consumption component

Two of the main roles on a Smart Grid are the roles of the consumer and the prosumer. Consumers and prosumers express their needs by means of their behavior, which goes from turning on the TV to feed industrial machinery. Therefore, to better simulate the real behavior of the buildings, it is actually useful to build a page to indicate the building mean consumption of a building by hours. As part of having multiple and well differentiated components on the application, *React Router* [11] was used. The library makes possible, in a composable and easy way, to navigate through the application by associating different Uniform Resource Locators (URLs) to distinct components defined at the core of the application. Additionally, to navigate through these components, a responsive page header was designed, which contains all the routes except for the route to launch the multiagent system, which it can only be accessed at the final stage of the grid design. Figure 5.6 shows the header when the viewport width is large and when the page width decreases to 600 pixels or fewer. After the development of the building consumption page was designed.



(a) Large Page Header

(b) Small Page Header

Figure 5.6: Responsive page header

Figure 5.7 shows the first version of the consumption selector component, in which there were several performance and lagging problems. An issue when designing a React component is the handling of too many elements mounted at the same time. React only mounts an element when it is shown on the screen and therefore, it saves resources when the element is not needed. In this case, the problem was caused by the internal state of each slider which React had to maintain and caused lagging on the sliders and component interactivity problems. In fact, the page performed poorer when the number of buildings scaled up. To solve the issue, several improvements were made:

- A design simplification. By removing the input component below each slider, the number of total components per building element was reduced considerably. However, the problem was still latent, as the number of states React had to manage was practically the same. That is why, the slider and the input shared the same internal state.
- The application of component pagination. Pagination is a common approach when solving performance issues on component lists. The division of the elements on multiple pages enables the browser and the library engine to manage less data and perform better.

[11] https://reactrouter.com/

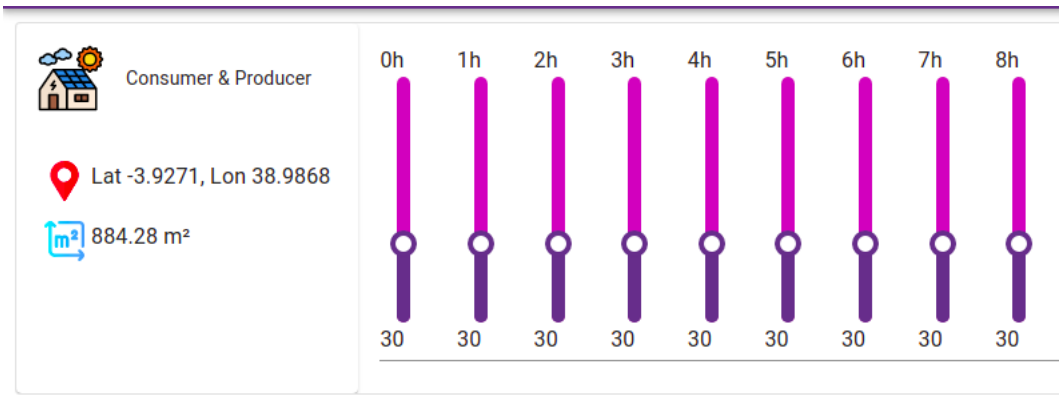


Figure 5.7: First version of the building consumption component

At the same time the component was modified, a **color palette** was designed to seek consistency with past and future component design. For this task, the color wheel theory was used, as a way to make a good choice. This theory is based on the fact that you can use a color wheel to find color harmonies by using the rules of color combinations. In this concrete case, a complementary combination was chosen, a violet tone *(#5F468A in hexadecimal)* was selected along with a yellow one *(#FFF2AF)*. To complement these tones, a darker violet *(#321E5C)* and a darker yellow *(#F5CF65)* were selected. Figure 5.8 shows the aforementioned palette built with Coolors [12]. Coolors is a color scheme generator which helps to generate color palettes and is widely used in the industry. After the building consumption card and the slider redesign and palette design, the component gained coherence with the whole application and the application started to have the same *Look and Feel*. Figure 5.9 shows the component after the redesign.

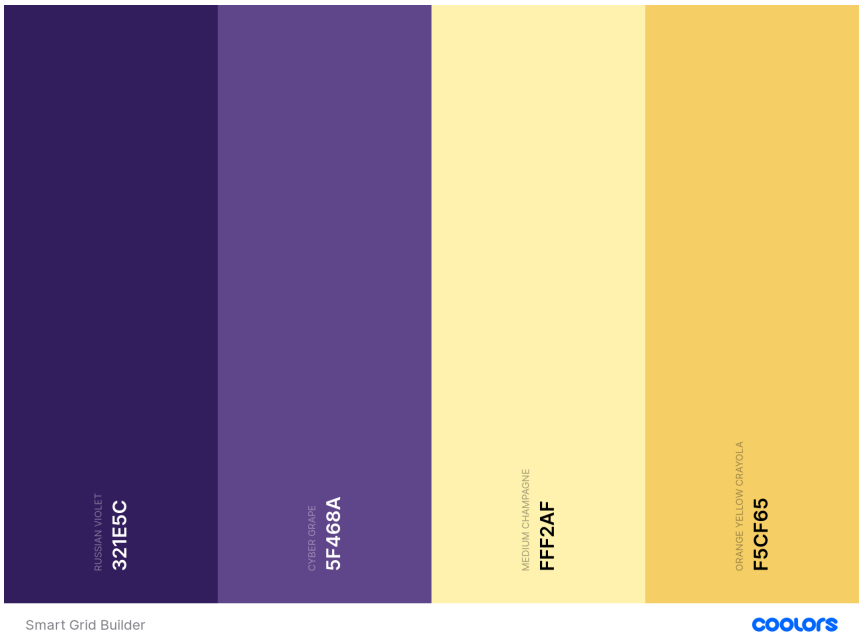
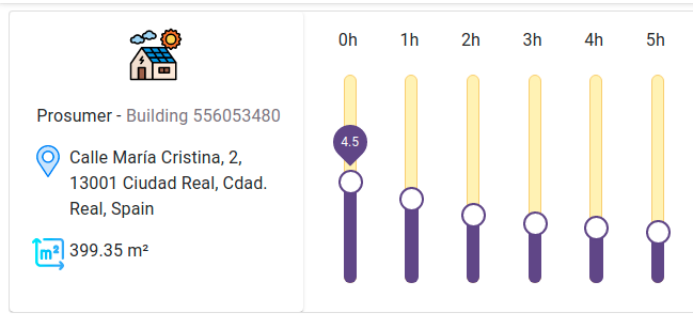


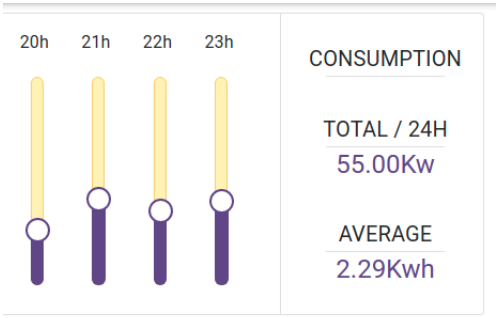
Figure 5.8: Frontend application color palette

Another important point when constructing single page applications with multiple routes is handling the component's internal state. In this sense, it is quite important to maintain the state of a component

[12] https://coolors.co/



(a) Left Part



(b) Right Part

Figure 5.9: Example of a building card once redesigned

when it is unmounted and to share some common state for usability reasons. However, React Router forces the dismount of the components associated with the current URL path, which totally deletes any change made to the Smart Grid configuration. One way to handle this situation is by *"lifting the state up"* [13] to parent components, but it makes the application harder to maintain. On the other hand, by extracting the shared state into a centralized location, any component can access the state or trigger actions to change it. This solution structures the code and helps to maintain independence between views and state. This is the idea behind Redux (Section 4.4.2), a predictable and centralized container designed to save the global state of JavaScript and React applications. For these reasons, Redux has been used to manage most of the application state. Redux is a framework-agnostic library which helps to write applications consistently. It makes predictable state mutations and restricts how and when these mutations can happen, and it is based on the idea of a one-way data flow, as Figure 5.10 shows:

- The **state**, the source of truth that drives the application.
- The **view**, a declarative description of the UI based on the current state.
- The **actions**, the events that occur in the app based on user input, and trigger updates in the state.

In Redux terminology, the concepts are the following:

- **Actions**. An action is a JavaScript object that describes a change that has happened in the application.
- **Reducers**. A reducer is a function which receives the action object and the state and decides

[13] https://reactjs.org/docs/lifting-state-up.html

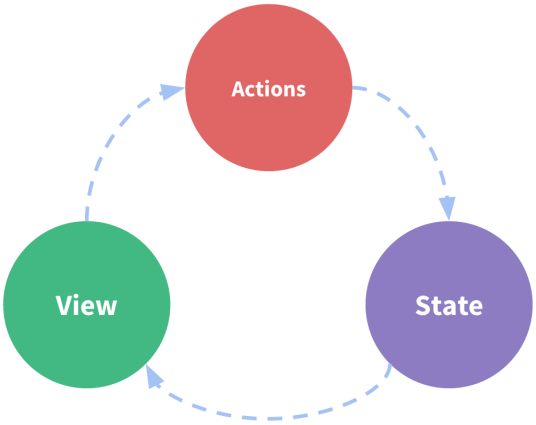


Figure 5.10: One-way data flow diagram

how to update it, if needed.

- **The Store**. The store is an object where the application state lives.
- **The Dispatcher**. The dispatcher is the way in which actions and reducers communicate.
- **Selectors**. A selector is a function that knows exactly how to extract specific pieces of information from a store state and avoids repeating logic (Don't Repeat Yourself (DRY) principle).

A Redux store only dispatches synchronous actions, however, most real applications need to work with data from APIs or to manage other kinds of asynchronous logic. The way to manage these asynchronous logic is by adding a *middleware*, an intermediary layer between the reducer and the asynchronous action. Figure 5.11 shows the Redux data flow pattern when asynchronous logic is needed. After the development of the building consumption component, the PV panel fitting algorithm and the multiagent system were ready to be built. The next section explains in detail how it was done.

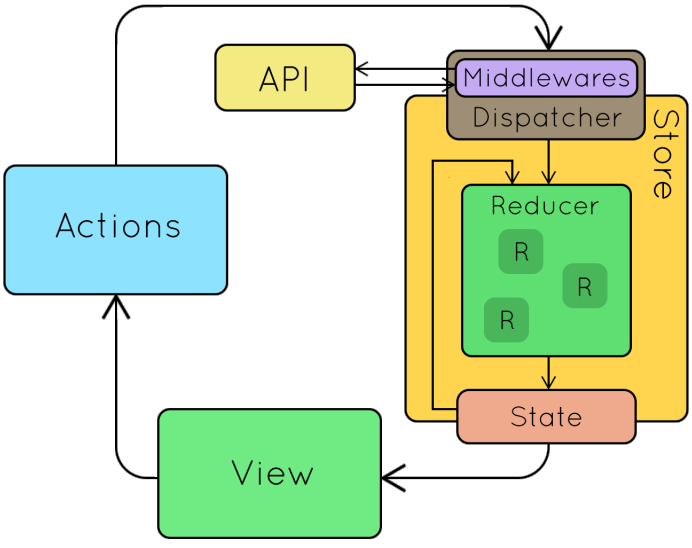


Figure 5.11: Redux data flow pattern

Source: *Redux js* [2].

5.3. SPRINT 3: PV PANEL FITTING ALGORITHM AND MULTIAGENT SYSTEM DEVELOPMENT

According to the project plan, the tasks associated with User Stories 4 and 5 are now described. This Sprint aimed to develop an algorithm to fit PV panels on a building roof-top and encompasses the development of the multiagent system which simulates the energy distribution between the buildings. Table 5.10 and 5.11 show both User Stories along with their sub-tasks and requirements. The next section details the development of both stories.

User Story 4		
Develop an algorithm to fit the maximum PV panels on the roof-top area of a building		
Sprint	**Priority**	**Effort**
3	Medium	15 hours
Description	Development of an algorithm to fit the maximum PV panels on the roof-top area of a building, knowing the panel dimensions and the building geometry.	
Acceptance criteria	▪ It must return the maximum number of panels which fit on a building roof-top. ▪ The algorithm complexity must not be high and the time to obtain the result must be feasible. ▪ There must be a visual tool to check the algorithm correctness.	
Tasks	▪ Design and develop the algorithm. ▪ Develop a way to show on screen the panel distribution on the building area.	

Table 5.10: User Story 4

User Story 5		
Develop a multiagent system to simulate the energy distribution between the Smart Grid components		
Sprint	**Priority**	**Effort**
3	High	30 hours
Description	Development of a multiagent system to simulate the energy distribution between the Smart Grid buildings which is connected with the PV energy inference module.	
Acceptance criteria	▪ The multiagent system must respond to FIPA standards. ▪ The producer agents must get the PV generation for each our from the API. ▪ The agents should communicate their energy needs and meet the energy needs of others if possible	
Tasks	▪ Model the different types of agents on thee multiagent system. ▪ Choose or design an interaction protocol for the energy negotiation. ▪ Integrate the energy generation endpoint on the API with producer agents.	

Table 5.11: User Story 5

5.3.1. Development of an algorithm to find the maximum PV panels on a building roof-top installation

This section explains in detail the algorithm designed to find the maximum amount of panels on the roof-top of a producer building. For this task, a coordinate system was needed, which permits to represent the roof-top and the panel area on a Cartesian Plane. The chosen coordinate system was the World Geodesic System (WGS), a standard in cartography, geodesy, and satellite navigation. Each building provided by Overpass API (Section 5.1.1) has a polygonal shape, and each polygon vertex has a unique latitude and a longitude, with which the polygon can be represented in two dimensions. **Latitude** measures the angular distance from the Earth's Equator to a point north or south of the equator, whereas **longitude** is the angular measure of east/west from the Prime Meridian. Therefore, *latitude* values increase or decrease along the vertical axis, while *longitude* values increase or decrease along the horizontal axis. Figure 5.13 shows the building visual representation in the web application and in the module using Matplotlib [14]. The first step of the algorithm was to obtain the dimensions of the PV panel. Equation 5.1 details the real width of a PV panel.

$$Panel\_width = real\_width * \cos(panel\_tilt\_angle) \tag{5.1}$$

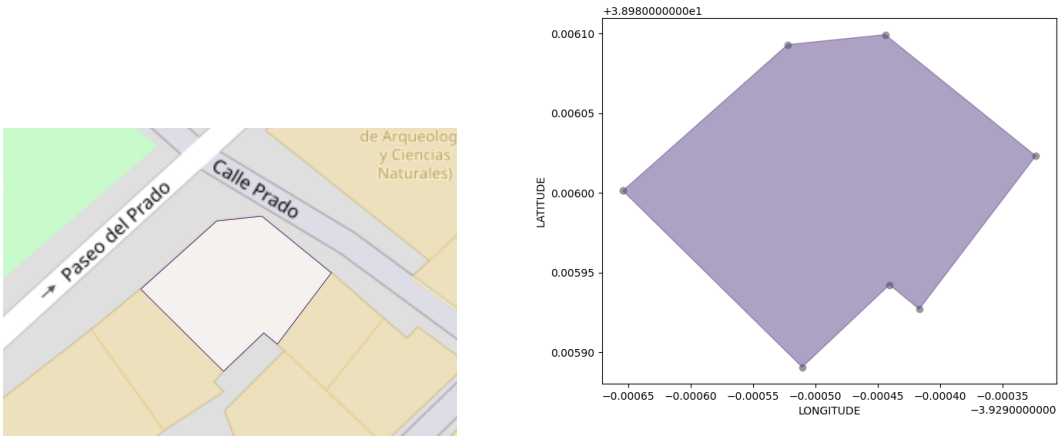
To facilitate the development of the algorithm, *Shapely* [15] was chosen. Shapely is a Python package that provides classes and functions for planar geometric objects analysis and manipulation. Listing 5.1 shows the developed algorithm based on the use of the package. First, the coordinates of the building polygon are converted from the form *(latitude, longitude)* to the form *(longitude, latitude)*, as the longitude is represented in the *horizontal axis (X)* on a Cartesian plane and the latitude is the *vertical axis (Y)*. After that, the panel width and height on this plane is calculated, by taking into account the polygon bounds and the real distance between them. In this context, the **Haversine distance** was used in order to determine the real distance between two points on a sphere (the earth), given their longitudes and latitudes. Equation 5.2 details the formula, where:

- r is the radius of the sphere, in this case 6371 kilometers.
- φ_1 and φ_2 are the latitude of point 1 and the latitude of point 2.
- λ_1 and λ_2 are the longitude of point 1 and the longitude of point 2.

$$haversine\_distance = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos\varphi_1 \cos\varphi_2 \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \tag{5.2}$$

Finally, panels are placed from the *(minimum x, minimum y)* point to the *(maximum x, maximum y)* vertex, always checking if they fit in. If a panel fits in, the maximum number of panels on the installation is incremented by one. Finally, the algorithm returns the maximum number of panels on the roof-top of a building. When placing the panels, those which are in the same row are contiguous, whereas the distance between the panels on different rows is equal to the length of the panel itself. This is done to avoid a phenomena called **panel shading**, in which one panel shadows one behind it, and it provokes a decrease in the energy production of the shaded panel. One shaded cell can

[14] https://matplotlib.org/
[15] https://github.com/Toblerity/Shapely



(a) Building in the web application

(b) Matplotlib building

Figure 5.13: Building represented in the web application vs Building represented in matplotlib

reduce the output of a whole string of cells or modules. Figure 5.14 shows a building installation result plotted on a Cartesian plane.

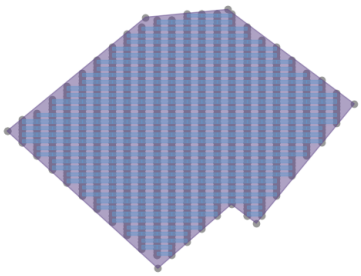


Figure 5.14: Building panel configuration

Listing 5.1: PV Panel fitting algorithm

```
from haversine import haversine as haversine_distance, Unit
from shapely.geometry import Polygon, MultiPolygon
from descartes.patch import PolygonPatch
from matplotlib import pyplot

BLUE = '#6699cc'
PURPLE = '#5f468a'

class PanelPlacer:

  @classmethod
  def __get_cartesian_panel_height_width(cls, panel_length, panel_width, building_bounds):
    '''
    It converts the panel width and height on meters into distance in the cartesian system
    '''
```

```
16      minx, miny, maxx, maxy = building_bounds
17      #Calculates the distance between de x/y min point and the x/y max point
18      width_distance = haversine_distance((minx, miny), (maxx, ↩
            ↪ miny), Unit.METERS)
19      height_distance = haversine_distance((minx, miny), (minx, ↩
            ↪ maxy), Unit.METERS)
20      #It divides the distances previously calculated between the panel width and height
21      #to obtain the maximum rows and columns of PV panels
22      max_rows = height_distance / panel_width
23      max_columns = width_distance / panel_length
24      #It calculates finally the height and the width of a panel
25      cartesian_panel_height = (maxy - miny) / max_rows
26      cartesian_panel_width = (maxx - minx) / max_columns
27      return cartesian_panel_height, cartesian_panel_width
28
29
30    @classmethod
31    def __coordinates_to_lonlat(cls, coordinates):
32      '''
33      It transforms the list of coordinates from the form
34      [(lat,lon),(lat,lon),...] to the form [(lon,lat),(lon,lat),...]
35
36      This process is needed because in a cartesian system the ↩
            ↪ longitude is the X axis
37      and the latitude the Y axis, but the coordinates are ↩
            ↪ normally represented
38      by the form (latitude, longitude)
39      '''
40      lonlat_list = []
41      for vertex in coordinates:
42        latitude, longitude = vertex
43        lonlat_list.append([longitude, latitude])
44      return lonlat_list
45
46    @classmethod
47    def run(cls, lanlon_coordinates, panel_length, panel_width):
48      lonlat_coordinates = ↩
            ↪ cls.__coordinates_to_lonlat(lanlon_coordinates)
49      building_polygon = Polygon(lonlat_coordinates)
50      building_bounds = building_polygon.bounds
51      #The panel height and width in the cartesian coordinate system are gotten
52      cartesian_panel_height, cartesian_panel_width = ↩
            ↪ cls.__get_cartesian_panel_height_width(panel_length, ↩
            ↪ panel_width, building_bounds)
53      #The bounds are the maximum and minimum points of the building polygon on the y
            and x axis
54      minx, miny, maxx, maxy = building_bounds
55      correctly_placed_panels = []
56      max_panels = 0
57      x = minx
58      y = miny
59
60      #An iteration from the bottom of the polygon to top and from the left part to the right
```

```
            one is done
61      while(y <= maxy):
62        y2 = y + cartesian_panel_height
63        while(x <= maxx):
64          x2 = x + cartesian_panel_width
65          panel_coordinates = [[x,y], [x,y2], [x2,y2], [x2, y]]
66          #A polygon representing the panel vertices is constructed.
67          #If it is within the building polygon, it means that it fits inside,
68          #the number of maximum panels on the roof-top of a building is incremented by
                one
69          panel = Polygon(panel_coordinates)
70          if panel.within(building_polygon):
71            correctly_placed_panels.append(panel)
72            max_panels += 1
73          x = x2
74
75        x = minx
76        #Whereas there is no distance between panels on different columns, the distance
              between panel rows is one panel
77        #This distance help to avoid panel shading, in which one panel shadows the one
              behind it
78        y = y2 + cartesian_panel_height
79
80      return max_panels
```

After the design of the panel fitting algorithm, the multiagent system was developed using the data provided by the previously developed modules. The following section explains in detail how it was done and how the system works.

5.3.2. Development of a multiagent system to simulate the energy distribution between the Smart Grid components

The development of the multiagent system is one of the most important parts in this BSc thesis. For its development, JADE was used, a framework which allows developing MAS in Java easily and also is compliant with FIPA standards. The first task when the multiagent system was developed was the modeling of the building roles into agents. Two kinds of agents were defined, the **Building Agent** and the **Grid Agent**. Each hour, the energy distribution happens, and the Building Agent can take the role of either a **consumer** or a **producer**. The prosumer building can be either a consumer or a producer, depending on its needs and the production on a concrete hour of the day. So, if a prosumer has energy deficit (it consumes more energy than it produces), then the building acts as a *consumer*, otherwise, it acts as a *producer*. On the other hand, the Grid Agent represents the electric supplier, which yields energy to the Smart Grid buildings if needed.
Afterwards, the right FIPA Interaction Protocol was needed to be found. The chosen Interaction Protocol was the **Contract-Net** one with several variations, the **Energy Distribution Contract-Net**. Figure 5.15 shows the *Energy Distribution Contract-Net*, in which the initiator role is taken either by a consumer or a prosumer with energy deficit, and the responder role is taken either by a producer, a consumer with energy surplus or the Grid Agent.

At first, there is a synchronization process in which the consumers wait for the producers to be ready for the energy distribution. In that time, the producers and prosumers have to obtain, from the API

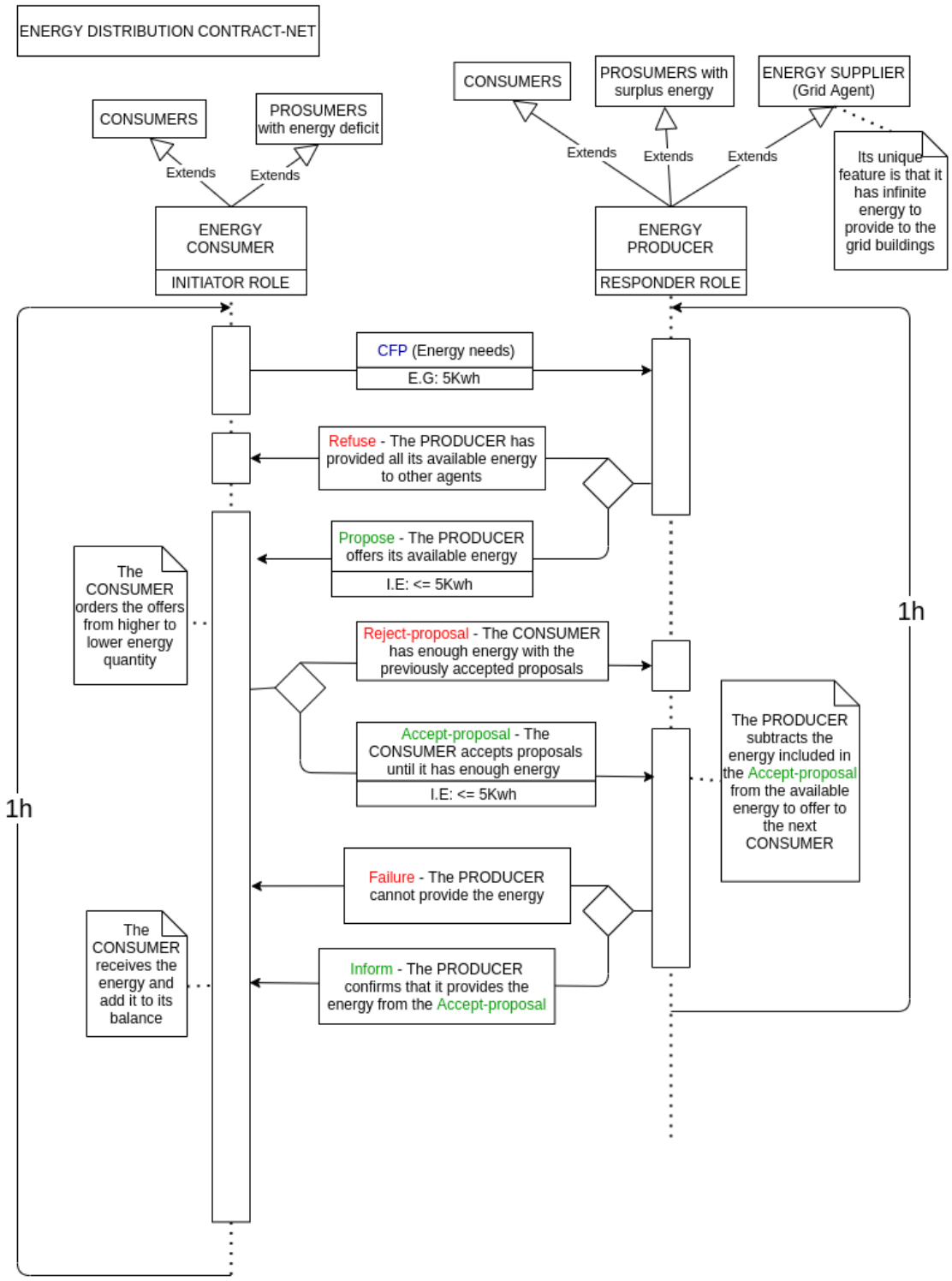


Figure 5.15: Energy Contract Net Interaction Protocol AUML

energy inference endpoint, the amount of energy generated by a certain PV panel installation. The request is made using Apache HttpComponents[16] library. After that, the producers annotate their needs and take the role of a producer. On the other part, the prosumers subtract their energy needs from the energy generated in that hour and in case of profit they act as producers, otherwise, they act as consumers. When this phased has finished, the consumers are ready to request their energy needs to the energy producers by means of a *CFP* message. In response, the producers can either:

- Send a proposal message with their available amount of energy to share.
- Reject the proposal, as they have provided all its available energy to other agents which previously asked for it.

If the response is a proposal, the consumers choose the minimum amount of offers for the demand is satisfied, and they send an *accept proposal* message along with the energy required from each corresponding agent. However, if the process finishes and the consumer agent still needs energy, that is to say, there are no producers offering energy or the total offer does not reach the demand, the consumers can request energy to the Grid Agent, which represents the **energy supplier** in the grid. The consumer rejects the proposals once its energy demand is fulfilled.

Once the producer receives the *accept proposal* message, it subtracts the energy which comes in the message from its available amount to share, and finally it sends an *inform* message if the process has been successful. However, if the process does not work properly, the consumer receives a *failure* message, and therefore it is able to revert the transaction and ask the Grid Agent, *representing the energy supplier*, for the energy that it needs. When a consumer fulfil its needs, its process finishes until the next hour. When all the consumers finish, producer agent processes also finish until the next hour, when the next energy distribution iteration starts again.

For synchronization purposes, each building agent has the whole grid data and takes into account the possible role of a building. Agents use the DF to identify themselves as a consumer, as a producer, or as a Grid Agent. In this Interaction Protocol, prosumers and producers share their remaining energy in order to improve the collective self-consumption rate and all agents live in a non-competitive environment, where social-welfare is desirable. After describing how the energy inference module and the multiagent system works, the next section address the integration of these parts into the web application.

5.4. SPRINT 4: DEVELOPMENT OF A DASHBOARD COMPONENT TO BE INTEGRATED WITH THE MULTIAGENT SYSTEM

As planned, the tasks associated with User Stories 6 and 7 are now accomplished. This Sprint aimed to develop a dashboard to visualize the multiagent system behavior along with the energy transactions and aimed to integrate the dashboard with the multiagent system. Tables 5.12 and 5.13 show both User Stories along with their sub-tasks and requirements. The next section details the development of these stories.

[16] https://hc.apache.org/

User Story 6		
Design of a web dashboard to visualize the energy distribution between the buildings.		
Sprint	**Priority**	**Effort**
4	Medium	15 hours
Description	Development of a web dashboard to visualize the energy distribution between the Smart Grid components and the Grid itself.	
Acceptance criteria	▪ The dashboard must show the energy transactions between the buildings. ▪ The dashboard must show a summary of the behavior of the Smart Grid with a certain configuration.	
Tasks	▪ Design and develop the UI of the page. ▪ Find and include statistics and metrics to check how the multiagent system behaves.	

Table 5.12: User Story 6.

User Story 7		
Integrate the web dashboard and the multiagent system components.		
Sprint	**Priority**	**Effort**
4	Medium	15 hours
Description	Development of a method to integrate the web dashboard and the multiagent system in which the behavior of the system shown in real time.	
Acceptance criteria	▪ The dashboard must be aware of the multiagent system behavior showing it in consequence. ▪ When the dashboard is opened, the multiagent system must be launched.	
Tasks	▪ Explore alternatives for the integration. ▪ Develop a way of connecting both components. ▪ Add persistence to the application by creating Representational State Transfer (REST) API endpoints.	

Table 5.13: User Story 7.

5.4.1. Development of the dashboard component

To develop the dashboard page, the design was divided in several components, which are consequently explained. The first component is placed at the top of the page and let the user visualize the overall statistics of the Smart Grid during the execution of the multiagent system. Figure 5.16 shows these statistics in the following order:

- **Last hour production:** It shows the energy production, in the last hour, of the buildings which conform the grid, without taking into account the energy shared by the energy supplier (the grid agent).
- **Last hour consumption:** It shows the energy consumption, in the last hour, of the buildings which conform the grid. It does not take into account the energy shared with the energy supplier (the grid agent) where there is an energy surplus).
- **Overall grid production:** It displays the energy production of the buildings which conform

the grid, without taking into account the energy shared by the energy supplier (the grid agent).

- **Overall grid consumption:** It displays the energy consumption of the buildings which conform the grid. It does not take into account the energy shared with the energy supplier (the grid agent) where there is an energy surplus).
- **Energy from the supplier:** It indicates the extra energy distributed by the energy supplier when the producer buildings do not have enough energy to share with the consumers.
- **Shared energy with the supplier:** It indicates the energy given to the energy supplier when the producer buildings have an energy surplus at the end of the energy distribution.

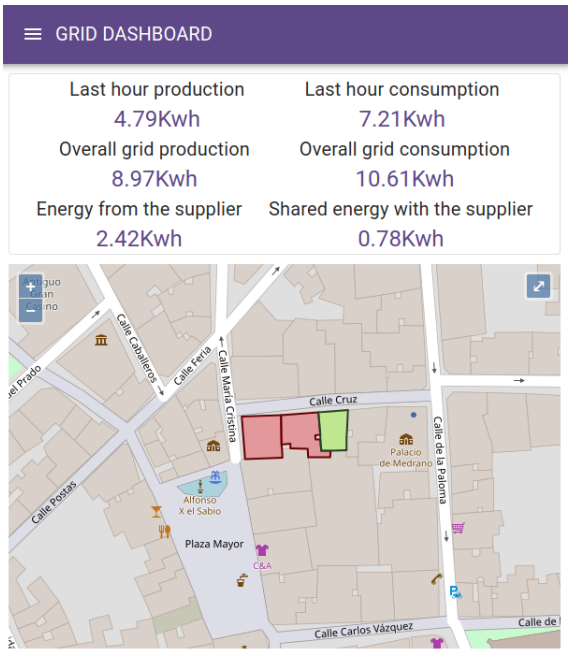


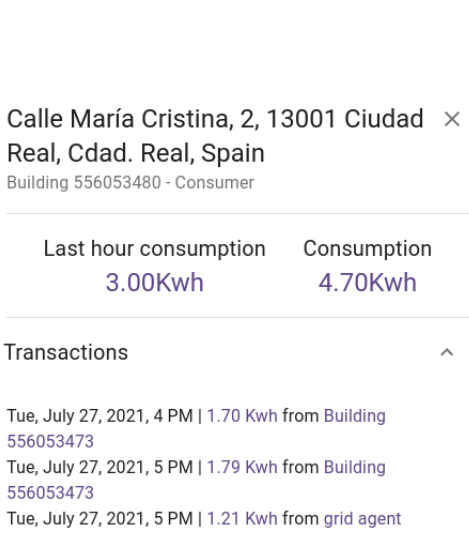
Figure 5.16: Smart Grid overview on an example simulation

The last two metrics can be used for *net metering* purposes, while the other metrics can be used to check how balanced is the energy production with respect to the consumption. Another component which makes up the dashboard page is a map, similar to the building selection one, where the buildings which conform the Smart Grid are shown. When any of these buildings is selected, a Pop-up component is shown, which contains the statistics and transactions the building has been involved in. These statistics are the following:

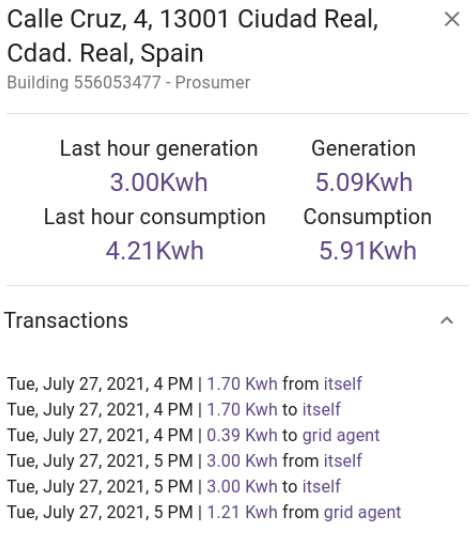
- **Last hour generation:** The energy generation of the building in the last hour.
- **Last hour consumption:** The energy consumption of the building in the last hour.
- **Overall generation:** The total energy generation during the execution of the multiagent system.
- **Overall consumption:** The total energy consumption during the execution of the multiagent system.

Figure 5.17 shows different examples of the statistics Pop-up when the multiagent system is working and registering the energy transactions. Depending on the energy balance of a building, its color and

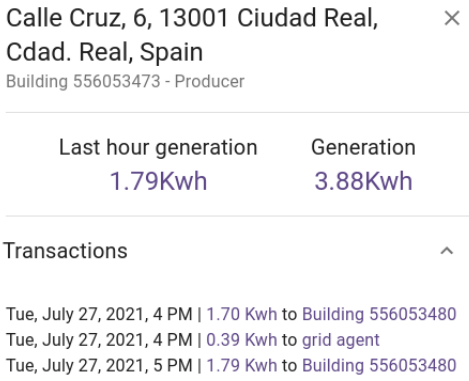
contour changes. The color is purple when the energy balance is break-even, it is garnet when the balance is negative, and it is green when the overall balance is positive. Figure 5.18 shows the color palette design used to highlight a building. Figure 5.19 shows the whole component look.



(a) Example of a consumer building Pop-up



(b) Example of a prosumer building Pop-up



(c) Example of a producer building Pop-up

Figure 5.17: Example of building Pop-up components based on its role

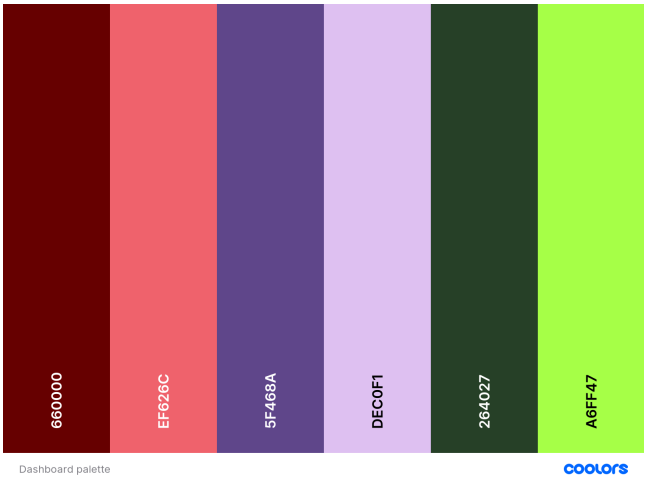


Figure 5.18: Dashboard color palette

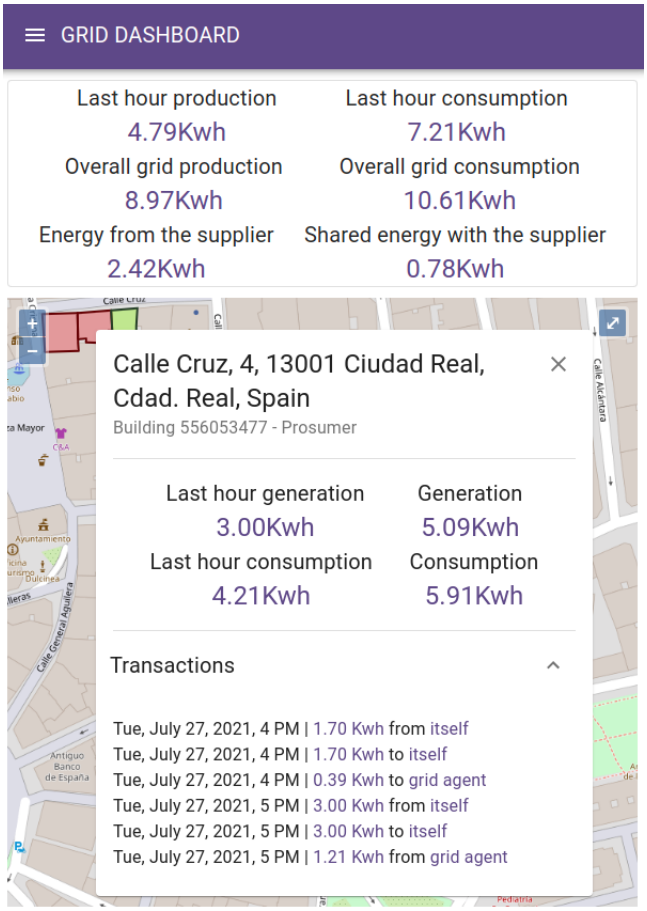


Figure 5.19: Dashboard page on an example simulation

5.4.2. Integration between the dashboard and the multiagent system

To integrate the dashboard component with the multiagent system, a connection layer had to be developed. For that, the previously mentioned REST API was built, that has specific endpoints to register the completed energy transactions which occur in the multiagent system. For that, the addition of a persistence layer is needed. For that, SQLAlchemy (Section 4.4.2) and alembic were used. SQLAlchemy is an ORM which simplifies the task of managing persistence on Python applications by mapping classes (**models**) to database tables. In the backend, this ORM can use different database engines, without any need for the user to learn the respective Structured Query Language (SQL) dialect. In this case, SQLite has been used as the **database engine**, as it is a lightweight engine, it is reliable, and the database file can be stored locally. On the other hand, alembic was used to generate and apply the *database migrations*. Alembic is a migration engine created for SQLAlchemy, which creates SQL sentences in the dialect of the SQL engine, to modify old database schemas. After some iterations of design, the resulting database **models** were the following ones:

Listing 5.2: Database models

```
1
2 from sqlalchemy import Float, Column, ForeignKey, Integer, ↩
      ↪ String, TIMESTAMP, DateTime
3 from sqlalchemy import func
4 from sqlalchemy.orm import relationship
```

```
from sqlalchemy.ext.declarative import declarative_base
import pandas as pd

Base = declarative_base()


#Grid model
class Grid(Base):
    __tablename__ = "grids"

    id = Column(Integer, primary_key=True, index=True)
    buildings = relationship("Building", back_populates="grid")

#Energy transaction model
class EnergyTransaction(Base):
    __tablename__ = "energytransactions"

    id = Column(Integer, primary_key=True, index=True)
    sender_id = Column(Integer, ForeignKey('buildings.id'))
    sender = relationship("Building", foreign_keys=[sender_id], ↩
        ↪ back_populates="sent_transactions")
    receiver_id = Column(Integer, ForeignKey('buildings.id'))
    receiver = relationship("Building", ↩
        ↪ foreign_keys=[receiver_id], ↩
        ↪ back_populates="received_transactions")
    energy = Column(Float)
    #exact timestamp when the transaction occurs
    timestamp = Column(TIMESTAMP, nullable=False, ↩
        ↪ default=func.now())
    #hour in which the transaction occurs in the smart grid
    grid_timestamp = Column(TIMESTAMP, nullable=True)


    def to_dict(self):
        '''
        Returns a dictionary which represents the energy ↩
            ↪ transaction
        '''

        return {
            'sender': self.sender.name,
            'receiver': self.receiver.name,
            'energy': self.energy,
            'timestamp': self.timestamp,
            'grid_timestamp': self.grid_timestamp
        }

#Building model
class Building(Base):
    __tablename__ = "buildings"

    id = Column(Integer, primary_key=True, index=True)
```

```
52      name = Column(String, index=True, nullable=False)
53      direction = Column(String, nullable=True)
54      type = Column(String, nullable=False)
55      grid_id = Column(Integer, ForeignKey('grids.id'))
56      grid = relationship("Grid", back_populates="buildings")
57      sent_transactions = relationship("EnergyTransaction", ↩
            ↪ foreign_keys=[EnergyTransaction.sender_id], ↩
            ↪ back_populates="sender")
58      received_transactions = relationship("EnergyTransaction", ↩
            ↪ foreign_keys=[EnergyTransaction.receiver_id], ↩
            ↪ back_populates="receiver")
59
60
61      def to_dict(self):
62          '''
63          Returns a dictionary which represents the building model
64          '''
65          return {
66              'name': self.name,
67              'direction': self.direction,
68              'type': self.type,
69          }
```

Once the models were created, several API endpoints were developed to perform Create Read Update Delete (CRUD) operations, such as the registration of new transactions and the obtainment of the non-fetched transactions of a concrete grid. The transaction creation endpoint is called by the multiagent system to register a new transaction once it is completed. On the other way, the web application calls the endpoint to get the non-fetched transactions periodically and to stay updated on the Multiagent System behavior. On a separate issue, the multiagent system is launched by an API request from the Frontend application in which the data of the designed grid is sent. The multiagent system is launched by means of a background process, which creates a new container in the JADE Agent Platform, in which the grid agents communicate autonomously.

This Sprint has shown the development of the dashboard page along with its integration with the user interface and the addition of a persistence layer. Next sections expose in detail the development of the tasks associated with the fifth Sprint.

5.5. SPRINT 5: DEVELOPMENT OF A PAGE TO CHOOSE THE QUANTITY OF PANELS AND THE INTEGRATION OF THE REAL ADDRESS OF A BUILDING

In this Sprint, the tasks associated with User Stories 8 and 9 are now described. This Sprint aimed to integrate the real address of a building in the web application via its geographical coordinates, and it also aimed to develop a component to select the number of PV panels of a producer building roof-top installation. However, due to the need for greater control on the lifecycle of the agents that make up the grids, an additional User Story was developed, the User Story 10. This story aimed to build an API to tear down and launch the JADE containers in which the agents of a same Smart Grid are contained. Tables 5.14 to 5.16 show the developed User Stories along with their subtasks and

requirements. The next section details the development of these stories.

User Story 8		
Integration of the real address of a building in the application via its geographical coordinates.		
Sprint	**Priority**	**Effort**
5	Low	10 hours
Description	A study and development of a solution to integrate the real address of a building in the application by its geographical coordinates.	
Acceptance criteria	▪ The obtained address must be accurate. ▪ The service must be reliable. ▪ The service must be free of charge.	
Tasks	▪ Find an API which returns the building address of a geographical location accurately. ▪ Develop the integration between the API and the application.	

Table 5.14: User Story 8.

User Story 10		
Development of a container manager API to launch and teardown the agent's containers of an Agent Platform		
Sprint	Priority	Effort
5	Medium	10 hours
Description	Development of an API to launch and teardown the JADE containers in which the agents, which conforms a Smart Grid, are placed.	
Acceptance criteria	▪ The API must use the JADE API to construct, launch and kill agents and containers. ▪ The API must be built in a Java Framework that allows an easy integration with the multiagent systems. ▪ The API must be able to manage multiple requests and save the active containers.	
Tasks	▪ The API must be able to manage multiple requests and save the active containers. ▪ Select a Java web Framework to develop the API. ▪ Create a container manager to manage the containers on a certain JADE Agent Platform. ▪ Integrate the container manager with the API to work together and launch the agents. ▪ Connect the FastAPI with the Grid Manager API.	

Table 5.15: User Story 10.

User Story 9		
Development of a page to select the maximum amount of panels of a producer building		
Sprint	**Priority**	**Effort**
5	Low	10 hours
Description	Development of a page to select the number of panels on the roof-top of a producer building	
Acceptance criteria	▪ The component must be intuitive and the number of panels for each producer building should be indicated by a number or by percentages. ▪ The components must be in accordance with the previously developed.	
Tasks	▪ Design the UI by making mockups and tests. ▪ Integrate the PV panel fitting API endpoint with the component.	

Table 5.16: User Story 9.

5.5.1. Integration of the real address of a building via its geographical coordinates

To integrate the real address of a building in the web application, several options and services were studied:

- **Nominatim** service is a FOSS tool to search OSM data by name and address (geocoding) and to generate synthetic addresses of OSM points (reverse geocoding). The problem was its general lack of building addresses information, therefore, the option was discarded.
- **Bing Maps** service can be also used for reverse geocoding. It did not have the problems of the previous service, but it did not provide free credits.
- **Google Maps API** has a reverse geolocation service which provides accurate addresses for most geolocation points, and it is the same service it uses on its well-known Google Maps application. Additionally, Google provides free credits to test its service, therefore, this was the selected option.

After the service was chosen, it was integrated in the building selection component. For that, an endpoint was set up on the REST API, as it is preferable to hide these kinds of keys from the Frontend users, despite the fact that usually the Frontend code is minified on page build time. When a building is selected on the building selection component, an asynchronous request is made through a *Redux Thunk*. The Thunk function makes a request to the API endpoint by making use of *axios* library. Finally, the address is shown on the component. If the service is not available, the latitude and the longitude of the building are shown on the component instead. Figure 5.20 shows an example of the building selection page once the integration was completed.

5.5.2. Development of a page to indicate the maximum number of panels of a PV installation

To develop this task, a mock-up component was made, which maintained homogeneity with the previously developed components. In the left part of the page there is a list of prosumer and producer buildings which make part of the Smart Grid. For all these buildings, the number of PV panels on its

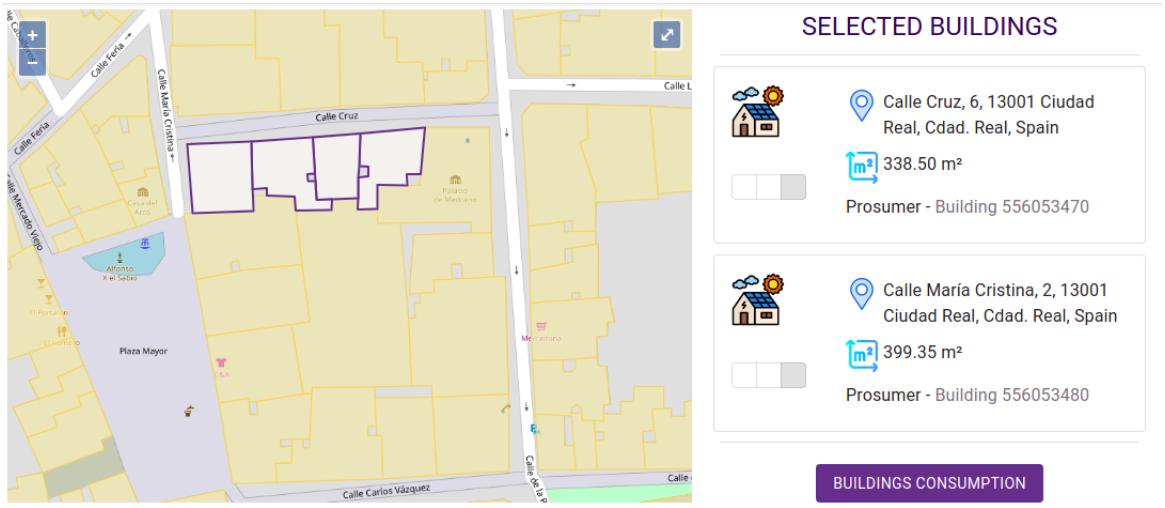


Figure 5.20: Builiding selection page

roof-top installation needs to be specified. When a building is selected, a Pop-up selector component appears in the middle of the page to indicate the possible number of panels, which goes from one panel to the maximum number of panels on the building roof-top. This number is limited by the area, the orientation and the inclination of the panels, and it is given by the PV panel fitting module. For that, an API request to the PV panel fitting module (Section 5.3.1) is made just when the building is selected in the building selection component (Section 5.1.3). Figure 5.21 shows the page once it was integrated with the rest of the components.

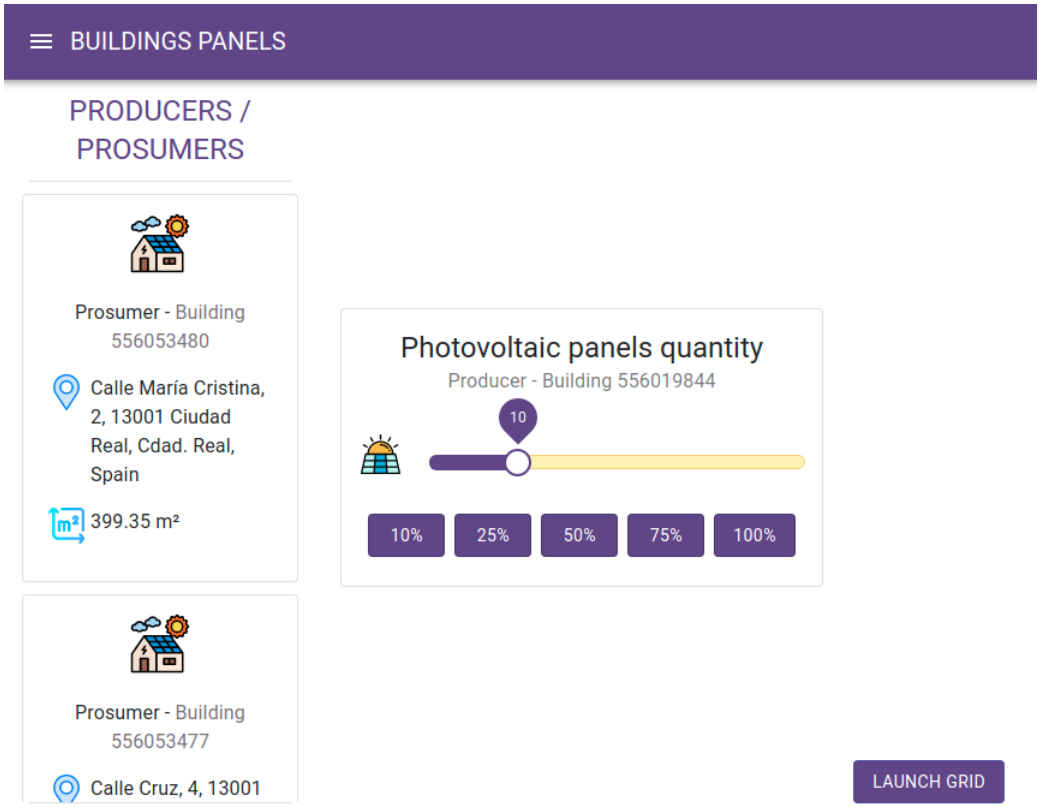


Figure 5.21: Page component to select the number of panels of a PV installation

5.5.3. Development of a container manager API to control the Smart Grid's state

This task emerged from the need of managing the agent's lifecycle, especially once the dashboard page is unmounted from the interface. As previously explained (Section 5.4.2), the multiagent system which represents a grid was launched by running a background process through the use of JADE Command Line Interface (CLI), however, it caused several problems:

- The first problem was that once an agent started, it could not be killed. This situation came from the fact that in JADE, each agent has its own thread. Furthermore, the containers and the platform processes are not exposed by the CLI, so there is no way of killing the agents of a concrete container, except for killing the process of the Agent Platform (AP).
- The second problem is that the JADE CLI is very limited in functionality. All the parameters have to be strings, to later be parsed to its correspondent type.
- Lastly, there was no centralized way of controlling the active containers.

For all these reasons, JADE provides a wrapper API which connects to an active AP and helps to manage containers and agents. This API is the only way, apart from the JADE GUI, of managing containers and agents lifecycle. By using it, a container manager was developed, which was capable of creating a container with certain agents, killing a container, and maintaining the state in memory of the active ones. On top of that, a REST API using Spring Framework [17] was developed, the **container manager API**, which wraps the container manager interface and exposes two endpoints, one to launch a container and another to kill an active container. Moreover, two additional endpoints in the FastAPI API were created, that were in charge of making the requests to the container manager API. In this way, the container manager API is transparent to the web application. Figure 5.23 shows the FastAPI API documentation, which breakdown all the previosly mentioned and developed endpoints, and it is powered by Swagger [18]. All these endpoints have been tested using *pytest* library, to ensure a correct behavior. Furthermore, Figure 5.22 shows the project final architecture. The architecture is composed by:

- The **Frontend application**, that only communicates with the FastAPI API.
- The **FastAPI API**, which manages data persistence, and communicates with the Spring API.
- The **Spring API**, which manage the Smart Grid containers.
- The **JADE AP**, that hosts the multiagent systems launched by the Spring API, and communicates with the FastAPI API to make energy inference requests.

Once the endpoints were developed, they were integrated with the web application, in such a way that when the dashboard page component is unmounted, the container killer endpoint is used. On the other hand, when the dashboard component is mounted, the container creation endpoint from FastAPI is used. This way of managing the containers brings greater scalability and robustness to the system by killing the non-used containers and managing better their state. Finally, Table 5.17 shows the backlog of the accomplished User Stories and their duration.

[17] https://spring.io/projects/spring-boot
[18] https://swagger.io/

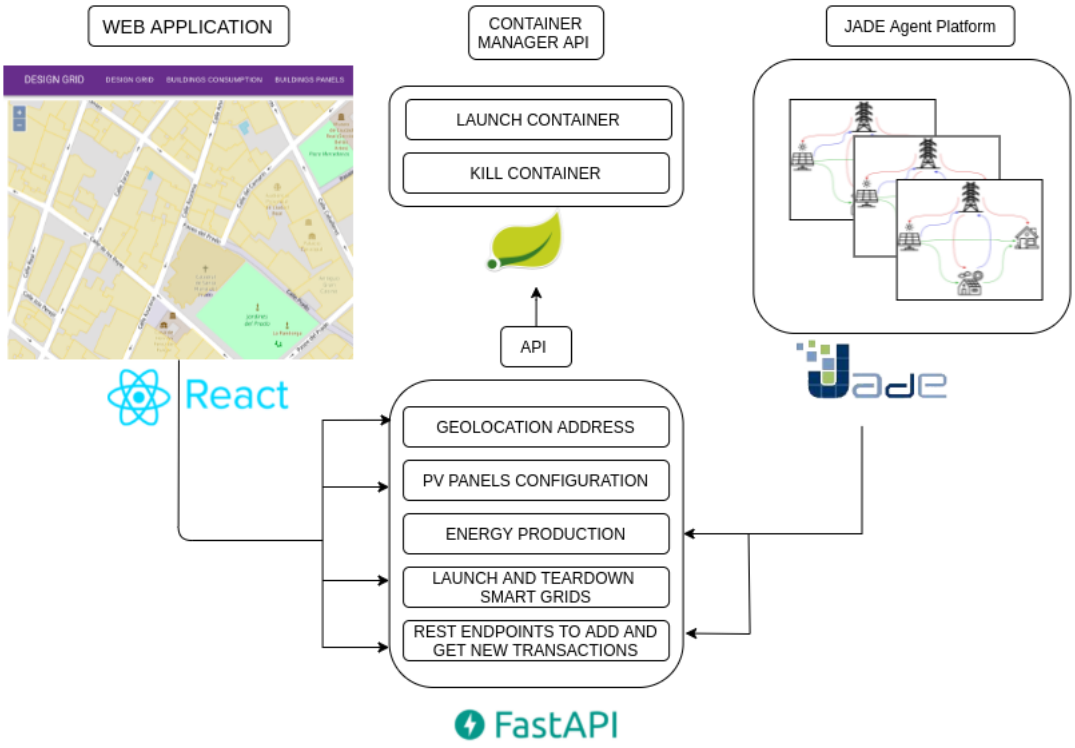


Figure 5.22: Project architecture

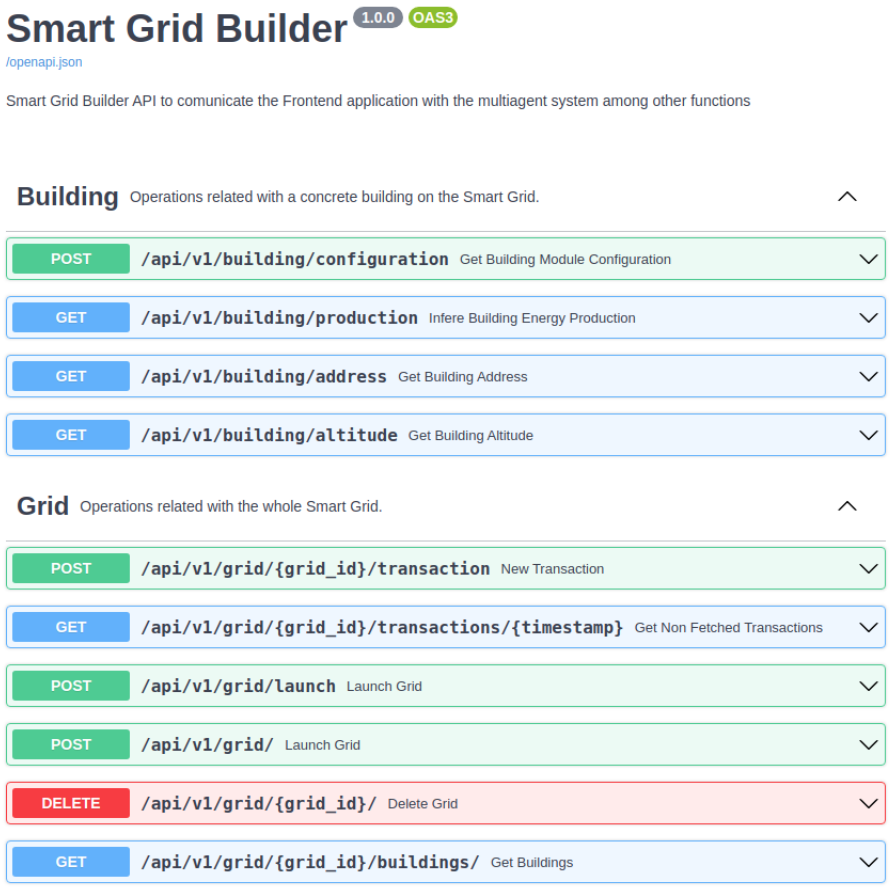


Figure 5.23: API built-in Swagger documentation

Backlog of completed User Stories			
ID	**User Story**	**Priority**	**Duration (h)**
1	Design a component on a web application to define the Smart Grid members.	High	30
2	Develop a module to infer the energy production of a PV roof installation.	High	25
3	Design and develop a component to indicate the hourly building consumption of a building.	Medium	20
4	Develop an algorithm to fit the maximum PV panels on the roof-top area of a building.	Medium	15
5	Develop a multiagent system to simulate the energy distribution between the Smart Grid components.	High	30
6	Design of a dashboard component to visualize the energy distribution between the buildings.	Medium	15
7	Integrate the web dashboard and the multiagent system components.	Medium	15
8	Integrate the real address of a building in the application via its geographical coordinates.	Low	10
9	Development of a page to select the maximum amount of panels of a producer building	Low	10
10	Development of a container manager API to launch and teardown the agent's containers of an Agent Platform	Medium	10

Table 5.17: Backlog of completed User Stories.

During this chapter, we have explained how the development of the project was followed, now the next section details the conclusions of this BSc thesis.

CHAPTER 6
Conclusions

This chapter presents a summary of the different goals achieved throughout the BSc thesis development, the competences acquired, and it proposes possible future lines of work and ideas to extend the project.

6.1. COMPLETENESS OF THE PROPOSED OBJECTIVES

The main objective of this BSc thesis was **the configuration and simulation of a Smart Grid of buildings, which aims to achieve a self-sustained distribution of energy by using PV panels installations**. The goal has been achieved through the completion of the four specific objectives planned at the initial stage of the project. Now the satisfaction of the specific objectives is analyzed.

- The first goal, **"Use map libraries and services to select the buildings which compose the Smart Grid and other necessary information such as the dimensions of the area in which the PV panels are placed"**, has been completed by means of a web application built with React, in which the user can select the Smart Grid buildings, detail the hourly mean consumption of each consumer building and the number of panels of each producer on its roof-top PV panel installation.
- The second goal, **"Simulate the performance of PV energy systems depending on the GPS coordinates of the smart grid, climatic conditions, and other factors"**, has also been achieved, concretely in the second Sprint. A module constructed with pvlib library is capable of inferring the energy production of a PV system in a certain time. To infer it, irradiance, meteorological and geographical data was used, and the PV system specifications for each building were defined, such as the PV panel model and the specific inverter. The module is exposed by means of an API endpoint.
- The third goal, **"Implement an algorithmic procedure based in multi-agent theory to simulate the energy sharing among the buildings"**, was achieved in the third Sprint. A multiagent system has been designed and implemented using JADE. The multiagent system uses a variation of the Contract-Net Interaction Protocol, the **Energy Distribution Contract-Net**, and it is intercommunicated with an API to register its own behavior. The system uses the web application data to simulate real energy transactions between the buildings which conform it.
- The fourth goal, **"Implementation of a web interface module to visualize the energy**

distribution, generation, and consumption of the Smart Grid components", has been completed in the fourth Sprint. A web interface has been constructed inside the web application which allows the user to visualize the general statistics of the Smart Grid, the statistics of each grid building, and all the energy transactions the building has been involved in.

This BSc thesis facilitates the creation and configuration of Smart Grid simulations using PV roof-top installations. On top of that, it is a very useful tool to analyze the performance of these types of installations. By simulating an energy distribution based on mutual benefit, the users can analyze if a system is disproportional and why. These types of grids are highly beneficial to the environment, and the project lowers the entry barriers of collective self-consumption for the general public. The project is divided in three differentiated parts which are easily extendable with new features. The next section exposes some future lines of work that could enhance the project.

6.2. COMPETENCES ACQUIRED

This work has led the author to fulfill the listed Computing specialization competencies [1] presented below:

- **CM1** *Ability to have a deep understanding of the fundamental principles and models of computation and know how to apply them to interpret, select, evaluate, model, and create new concepts, theories, uses and technological developments related to computing.*
  This BSc thesis consists of a system to design Smart Grids and to simulate the energy distribution between their buildings, by using a new MAS Interaction Protocol to distribute the energy which uses intelligent agent's theory as a basis.
- **CM3** *Ability to evaluate the computational complexity of a problem, to know algorithmic strategies that can lead to its resolution and to recommend, develop and implement the one that guarantees the best performance according to the established requirements.*
  The system uses an algorithm to design a PV system using the maximum amount of PV panels that fit on a building roof-top. Furthermore, it uses another algorithm to infer the energy production of this system for a concrete moment of time.
- **CM4** *Ability to know the fundamentals, paradigms and techniques of intelligent systems and to analyze, design and build systems, services and computer applications that use these techniques in any field of application.*
  An application to ease the design of a Smart Grid from a user's point of view, as well as an intelligent system responsible to distribute the energy between the grid buildings, have been designed. Furthermore, two REST APIs have been built to connect both parts and manage the active Smart Grids.
- **CM5** *Ability to acquire, obtain, formalize and represent human knowledge in a computable form for the resolution of problems by means of a computer system in any field of application, particularly those related to aspects of computation, perception and performance in intelligent environments.*
  Formalization of human knowledge for problem-solving is essential when designing multiagent

[1] https://pruebasaluuclm.sharepoint.com/sites/esicr/tfg/SiteAssets/SitePages/Inicio/CompetenciasIntensificaciones.pdf

systems. During this project, different approaches and its impact on the Smart Grid buildings were studied, examining its benefits and finally applying a mutual-benefit approach to add value to the energy supply landscape.

6.3. FUTURE WORK

In this section, some ideas to improve the prototype and to expand it are proposed:

- The addition of other types of Renewable Energy Sources (RESs) to the system, such as eolic energy. By using other types of renewable energy sources, the self-sufficiency and the self-consumption rate could increase, as these sources can smooth out the peaks and troughs of the PV energy production.
- The inclusion of facade integrated PV systems to exploit the vertical walls of a building. This novelty approach helps to reduce the energy consumption by decreasing the workload of air conditioning and lighting systems. They also provide efficient thermal insulation, good noise and weather protection, and they are used as a supplementary power source for buildings.
- The inclusion of other negotiation models based on mutual-benefit or in self-benefit, which could use other performance metrics and/or heuristics.
- The incorporation of a cost analysis on the designed Smart Grids, which could analyze the long-term viability of the PV installations and the ROI for the owners based on the performance of the system.

Bibliography

[1] *14th Annual State Of Agile Report*, May 2020. [Online]. Available: https://stateofagile.com/#ufh-i-615706098-14th-annual-state-of-agile-report/7027494.

[2] D. Abramov, *Redux Fundamentals, Part 2: Concepts and Data Flow*, 2015. [Online]. Available: https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow.

[3] B. Baesens, A. Backiel, and S. v. Broucke, *Beginning Java Programming: The Object-Oriented Approach*. John Wiley & Sons, 2015, ISBN: 9781118739495.

[4] A. Banks and E. Porcello, *Learning React: Functional Web Development with React and Redux*. "O'Reilly Media, Inc.", 2017, ISBN: 9781491954577.

[5] F. Bellifemine, A. Poggi, and G. Rimassa, "JADE — A FIPA-compliant agent framework", in *Proceedings of PAAM*, London, vol. 99, 1999, p. 33.

[6] E. Calvo-Gallardo, N. Arranz, and J. C. Fernández de Arroyabe, "Analysis of the European energy innovation system: Contribution of the Framework Programmes to the EU policy objectives", *Journal of Cleaner Production*, vol. 298, p. 126 690, 2021, ISSN: 0959-6526. DOI: https://doi.org/10.1016/j.jclepro.2021.126690. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0959652621009100.

[7] S. Chacon and B. Straub, *Pro Git*. Apress, 2014, ISBN: 9781484200766.

[8] M. Cohn, *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, 2004, ISBN: 0321205685.

[9] T. Couture *et al.*, "A Policymaker's Guide to Feed-in Tariff Policy Design", Oct. 2010. DOI: 10.2172/984987.

[10] X. Dong *et al.*, "Optimal Battery Energy Storage System Charge Scheduling for Peak Shaving Application Considering Battery Lifetime", in. Jan. 2011, vol. 133, pp. 211–218, ISBN: 978-3-642-25991-3. DOI: 10.1007/978-3-642-25992-0_30.

[11] S. Dunlop and A. Roesch, *EU-wide solar PV business models*. 2016. [Online]. Available: solarpowereurope.org.

[12] *FIPA Agent Management Specification*, 2002. [Online]. Available: http://www.fipa.org/specs/fipa00023/SC00023J.html.

[13] *FIPA Contract Net Interaction Protocol Specification*, 2002. [Online]. Available: http://www.fipa.org/specs/fipa00029/SC00029H.html.

[14] D. Flanagan, *JavaScript: The Definitive Guide*. "O'Reilly Media, Inc.", 2011, ISBN: 9780596805524.

[15] P. Gevorkian, *Solar power in building design (greensource): The engineer's complete project resource*. McGraw Hill Professional, 2007, pp. 109–101, ISBN: 9780071594448.

[16] A. Group. (Nov. 2001). "Principles behind the agile manifesto", [Online]. Available: https://agilemanifesto.org/principles.html.

[17] S. Howell *et al.*, "Towards the next generation of smart grids: Semantic and holonic multi-agent management of distributed energy resources", *Renewable and Sustainable Energy Reviews*, vol. 77, pp. 193–214, 2017, ISSN: 1364-0321. DOI: https://doi.org/10.1016/j.rser.2017.03.107. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1364032117304392.

[18] M.-P. Huget and J. Odell, "Representing Agent Interaction Protocols with Agent UML", Jan. 2004, pp. 1244–1245, ISBN: 978-3-540-24286-4. DOI: 10.1109/AAMAS.2004.230.

[19] IEA, *World Energy Outlook 2019*, 2019. [Online]. Available: https://www.iea.org/reports/world-energy-outlook-2019.

[20] P. M. Institute, *Pulse of the Profession 2017*, 2017. [Online]. Available: https://www.pmi.org/learning/thought-leadership/pulse/pulse-of-the-profession-2017.

[21] *JADE Programming Tutorial*, 2009. [Online]. Available: https://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf.

[22] L. Lamport, *LATEX: A document preparation system: User's guide and reference manual*. Addison-Wesley Professional, 1994.

[23] R. Luthander *et al.*, "Photovoltaic self-consumption in buildings: A review", *Applied Energy*, vol. 142, pp. 80–94, 2015, ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2014.12.028. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306261914012859.

[24] M. Lutz, *Learning Python: Powerful Object-Oriented Programming*. "O'Reilly Media, Inc.", 2013, ISBN: 9781449355692.

[25] Lvivity, 2020. [Online]. Available: https://lvivity.com/single-page-app-vs-multi-page-app.

[26] I. Maity and S. Rao, "Simulation and Pricing Mechanism Analysis of a Solar-Powered Electrical Microgrid", *IEEE Systems Journal*, vol. 4, no. 3, pp. 275–284, 2010. DOI: 10.1109/JSYST.2010.2059110.

[27] V. Masson-Delmotte *et al.*, *Global Warming of 1.5 oC*. 2018. [Online]. Available: https://www.ipcc.ch/sr15/download/.

[28] M. McDonnell, *tmux Taster*. Apress, 2014, ISBN: 9781484207758.

[29] E. McKenna, J. Pless, and S. J. Darby, "Solar photovoltaic self-consumption in the UK residential sector: New estimates from a smart grid demonstration project", *Energy Policy*, vol. 118, pp. 482–491, 2018, ISSN: 0301-4215. DOI: https://doi.org/10.1016/j.enpol.2018.04.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0301421518302222.

[30] E. A. Meyer, *CSS: The Definitive Guide*. O'Reilly Media, 2017, ISBN: 1449393195.

[31] A.-H. Mohsenian-Rad *et al.*, "Autonomous Demand-Side Management Based on Game-Theoretic Energy Consumption Scheduling for the Future Smart Grid", *IEEE Transactions on Smart Grid*, vol. 1, no. 3, pp. 320–331, 2010. DOI: 10.1109/TSG.2010.2089069.

[32] C. Musciano and B. Kennedy, *HTML & XHTML: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2006, ISBN: 0596527322.

[33] S. Overflow, 2020. [Online]. Available: https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents.

[34] B. Robyns, A. Davigny, and C. Saudemont, "Methodologies for supervision of Hybrid Energy Sources based on Storage Systems – A survey", *Mathematics and Computers in Simulation*, vol. 91, pp. 52–71, 2013, ISSN: 0378-4754. DOI: https://doi.org/10.1016/j.matcom.2012.06.014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378475412001504.

[35] B. Robyns *et al.*, *Energy Storage in Electric Power Grids*. John Wiley & Sons, 2015, pp. 17–52, ISBN: 9781119058687.

[36] S. J. Russell *et al.*, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010, ISBN: 9780136042594.

[37] J. Salido. (2019). "Plantilla guía de TFG para la ESI-UCLM". GitHub, Ed., Universidad de Castilla-La Mancha, [Online]. Available: https://github.com/JesusSalido/TFG_ESI_UCLM.

[38] K. Schwaber, *The Definitive Guide to Scrum: The Rules of the Game*. 2017.

[39] M. Stephant *et al.*, "A survey on energy management and blockchain for collective self-consumption", in *2018 7th International Conference on Systems and Control (ICSC)*, 2018, pp. 237–243. DOI: 10.1109/ICoSC.2018.8587812.

[40] W. Su and M.-Y. Chow, "Performance evaluation of a PHEV parking station using Particle Swarm Optimization", in *2011 IEEE Power and Energy Society General Meeting*, 2011, pp. 1–6. DOI: 10.1109/PES.2011.6038937.

[41] M. para la Transición Ecológica, *Real Decreto 244/2019*. 2019. [Online]. Available: https://www.boe.es/eli/es/rd/2019/04/05/244/con.

[42] UCLM, *UCLM Retributions*, 2021. [Online]. Available: https://www.uclm.es/-/media/Files/A01-Asistencia-Direccion/A01-023-Vicerrectorado-Politica-Cientifica/Formularios/Contratos/TABLA-RETRIBUCIONES-2021.ashx.

[43] C. H. Villar, D. Neves, and C. A. Silva, "Solar PV self-consumption: An analysis of influencing indicators in the Portuguese context", *Energy Strategy Reviews*, vol. 18, pp. 224–234, 2017, ISSN: 2211-467X. DOI: https://doi.org/10.1016/j.esr.2017.10.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2211467X17300627.

[44] M. Wooldridge, *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2009, ISBN: 9780470519462.

[45] Z. Zhao *et al.*, "An Optimal Power Scheduling Method for Demand Response in Home Energy Management System", *IEEE Transactions on Smart Grid*, vol. 4, no. 3, pp. 1391–1400, 2013. DOI: 10.1109/TSG.2013.2251018.

APPENDICES

APPENDIX A

User Manual

All the code developed is publicly hosted in GitHub under the next link:

https://github.com/EnriqueCepeda/TFG-EnriqueCepeda

The project is organized into the following directories:

- **Frontend.** Web application constructed with React, which lets the user design and visualize the Smart Grid behavior and the energy transactions between the grid buildings.
- **Backend.** Backend API responsible for the roof-top PV panel fitting algorithm, for the PV energy inference module and for the application data persistence, among other things.
- **Multiagent.** REST API which launches the multiagent systems that simulate the energy distribution between the buildings which conforms a Smart Grid.

A.1. SYSTEM REQUIREMENTS

- Python 3, 3.8.10
- Pip, 20.2.4
- Node, 12.18.2
- Npm, 6.14.8
- Java 8, 1.8 (OpenJDK 8 or JAVA SE 8)
- Apache Maven 3.6.3

A.2. INSTALLATION PROCESS

First, it is necessary to clone the repository.

```
$ git clone --recursive https://github.com/EnriqueCepeda/TFG-EnriqueCepeda.git
```

Afterwards, in the Backend directory, there is a requirements file containing all the dependencies with their version number. These are meant to be installed in an isolated virtual environment using Python v3.8.10, however, they can also be installed in the default Python interpreter. In order to install the dependencies, the user must navigate to the backend directory and run the following commands:

```
$ pip install -r requirements.txt
$ pip install -e lib/pvlib-python/.\[all\]
```

P.S: Python and pip must be installed first

Regarding the Frontend application, the dependencies are also isolated, but instead of installing them with *pip* package manager, *npm* must be employed. So, the user must navigate to the frontend directory and run:

```
$ npm install
```

P.S: Node Js and npm must be installed first

Regarding the multiagent system, the installation of certain packages and dependencies is needed. For that, *maven* is used. To install the dependencies, the user mustnavigate to the multiagent folder and run the following command:

```
$ mvn install
$ mvn dependency:copy-dependencies
```

P.S: Maven and Java must be installed first

Afterwards, set the GOOGLE_MAPS_APIKEY environment variable in your system using the provided value by e-mail. Also add the following environment variables for the multiagent system to work properly:

```
PROJECT_FOLDER="{path_to_repo}/TFG/multiagent"
CLASSPATH="$PROJECT_FOLDER/lib/*:$PROJECT_FOLDER/target/classes:$PROJECT_FOL
↪  DER/target/dependency/*"
```

P.S: The way in which the environment variables are introduced may vary depending on the system, in this case, it was done on an Ubuntu 21.10 machine using zsh shell. The dollar sign ($) references other variable value, and the colon divides one value from another.

A.3. RUN THE MULTIAGENT SYSTEM SERVER AND THE JADE AGENT PLATFORM

To use the application, first is necessary to activate the *JADE Agent Platform* and the *Spring-boot* server. To activate the Agent Platform, the user must navigate to the *multiagent* folder and run the following command:

```
$ java jade.Boot -name -SmartGrid -gui
```

To run the *Spring-boot* server, the user must navigate to the multiagent folder and execute the following command:

```
$ mvn spring-boot:run
```

A.4. RUN THE BACKEND SERVER

To use the application, first is necessary to run the backend server, which uses *uvicorn Asynchronous Server Gateway Interface (ASGI) server*. To run it, the user must navigate to the backend folder and execute the following command:

```
$ uvicorn api.main:app --port 8000
```

To run the API test suite, the user must execute the following command in the backend folder:

```
$ pytest api
```

Lastly, to watch the backend documentation, the user must open a web browser and navigate to http://localhost:8000/docs.

A.5. RUN THE FRONTEND SERVER

To run the application in the browser, the application needs to be constructed and minified. For that, the user must navigate to the frontend directory and run the following command:

```
$ npm run build
```

To serve the application build generated by the previous command, the user must run the following command:

```
$ sudo npm install -g serve
$ serve -s build
```

Lastly, to use the application, it is necessary to open a web browser and navigate to http://localhost:5000.