Universidade de Aveiro
2022

Eduardo Luís Fonseca Coelho

GESTÃO DE ENERGIA EM CARREGADOR ELÉCTRICO

ENERGY MANAGEMENT IN ELECTRIC CHARGER

Universidade de Aveiro
2022

Eduardo Luís Fonseca Coelho

GESTÃO DE ENERGIA EM CARREGADOR ELÉCTRICO

ENERGY MANAGEMENT IN ELECTRIC CHARGER

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Diogo Nuno Pereira Gomes, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Daniel Nunes Corujo, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho aos meus colegas da matrícula 2017/2018 que me acompanharam e à minha família que tornou esta jornada possível.

o júri / the jury

presidente / president — Prof. Doutor Rui Manuel Escadas Ramos Martins
Professor Auxiliar da Universidade de Aveiro

vogais / examiners committee — Prof. Doutor Filipe Manuel Simões Caldeira
Professor Adjunto do Instituto Politécnico de Viseu - Escola Superior de Tecnologia e Gestão

Prof. Doutor Diogo Nuno Pereira Gomes
Professor Auxiliar da Universidade de Aveiro

agradecimentos / acknowledgements

Quero agradecer a todos os meus colegas que fizeram parte deste percurso de 5 anos, onde toda a matrícula 2017/2018 sempre teve um espírito de entre-ajuda incrível. Quero também agradecer à minha família que me deu todo o apoio e condições que necessitava para conseguir iniciar e concluir os meus estudos, principalmente nos momentos de trabalho e estudo mais intenso.
Agradeço ao Professor Daniel Corujo pela sua prestabilidade, e um agradecimento especial ao Professor Diogo Gomes pela sua orientação e pela sua disponibilidade e rapidez com que me prestou auxilio ao longo deste ano.
Por fim, resta-me agradecer aos colegas Alexandre Abreu, João Génio, João Trindade e Rúben Menino que me ajudaram nos piores momentos e contribuíram para os melhores, acompanhando-me de forma mais chegada neste percurso que sem dúvida teria sido pior com a ausência deles. Agradeço também ao Bruno Lopes, Eurico Dias, Daniel Correia e Joaquim Ramos com quem tive mais interações perto do fim deste percurso, mas com os quais tive também experiências positivas que estimo bastante.
Finalizo com um agradecimento ao corpo docente do departamento e às associações/grupos como GLUA, NEECT-AAUAv, AETTUA e CF de MIECT, pois todas contribuíram de certa forma para a minha integração ou aprendizagem.

Palavras Chave

Veículos Elétricos, Estação de Carregamento, OCPP, Microsserviços, Carregamento AC, Carregamento DC.

Resumo

Veículos Elétricos têm vindo gradualmente a ganhar mais tração, aparecendo como uma possível medida para reduzir o impacto ambiental causado pelo setor dos transportes. De modo a melhorar a qualidade da infraestrutura de carregamento elétrico, esta dissertação aborda o desenvolvimento de uma estação de carregamento com foco na modularidade e facilidade de atualização. A estação de carregamento deverá implementar uma arquitetura orientada a serviços com um ambiente distribuído de múltiplos carregadores em rede capazes de balancear cargas em tempo real. A estação de carregamento comunica com um sistema central utilizando o protocolo OCPP na versão mais recente (OCPP 2.0.1), utilizado para controlar uma rede de carregadores.

Keywords

Electric Vehicles, Charging Station, OCPP, Microservices, AC Charging, DC Charging.

Abstract

Electric Vehicles have been gradually gaining more traction, appearing as a possible measure to reduce the environment impact caused by the transportation sector. To increase the quality of the electric charging infrastructure, this dissertation addresses the development of a charging station with a focus on modularity and updatability. The charging station should implement a service-oriented architecture with a distributed environment of multiple chargers in grid capable of balancing loads in real time. The charging station communicates with a central system by using the most recent version of the OCPP protocol (OCPP 2.0.1), used to control a charger network.

Contents

Contents i

List of Figures v

List of Tables ix

Acronyms xi

1 Introduction 1

1.1 Context 1

1.1.1 Impact of the transportation sector in CO2 emissions 1

1.1.2 Electric Vehicles (EV) as a solution 2

1.1.3 Motivation 3

1.2 Objectives 3

1.3 Document structure 4

2 State of the Art 5

2.1 Charging Infrastructure 5

2.2 Understanding electric current 7

2.3 Type 2 connector in Europe 8

2.4 IEC 61581 9

2.4.1 Low-Level Communication (LLC) 9

2.4.2 Charging Modes 10

2.5 ISO 15118 13

2.5.1 Power Line Communication 13

2.5.2 IEC 61581 Annex D - extended pilot function 15

2.6 Open Charge Point Protocol 15

2.6.1 OCPP 2.0.1 16

2.7 EV Roaming 17

2.8 Architecture 18

2.8.1 Monolithic Architecture . . . 18
2.8.2 Microservices Architecture . . . 20
2.9 Message-Oriented Middleware . . . 21
2.9.1 Apache Kafka . . . 22
2.9.2 RabbitMQ . . . 23
2.10 Summary . . . 25

3 Implementation **27**
3.1 Architecture Overview . . . 27
3.2 Previous Development . . . 29
3.3 Message Broker . . . 31
3.4 Charge Point . . . 33
3.5 Decision Point . . . 35
3.5.1 Orchestrator . . . 36
3.5.2 Handling Requests & Responses . . . 36
3.5.3 Transactions . . . 37
3.5.4 Reservations . . . 40
3.5.5 Local Database . . . 40
3.6 Mode3 Protocol Driver . . . 41
3.6.1 Connector Hardware & EV Driver Simulator . . . 44
3.7 Energy Driver & PowerModule . . . 45
3.7.1 PowerModule Hardware Simulator . . . 47
3.8 Parking Sensor . . . 48
3.9 Human-Machine Interface . . . 49
3.10 RFID Card Reader . . . 51

4 Results **53**
4.1 Analyzing an Entire Charging Session . . . 53
4.1.1 Before Arriving at Charging Station . . . 53
4.1.2 Arriving at Charging Station . . . 54
4.1.3 Authorization at Charging Station . . . 55
4.1.4 Start Charging Session . . . 57
4.1.5 During Charging Session . . . 58
4.1.6 Stop Charging Session . . . 60
4.1.7 After Charging . . . 62
4.2 Mode3Driver Finite State Machine - Unit Testing . . . 63
4.3 Mode3Driver Time Requirements Analysis . . . 66
4.4 HTTP Server Testing . . . 67

4.4.1 Smoke Testing 67
4.4.2 Load Testing 68
4.4.3 Stress Testing 68
4.4.4 Considerations 69

5 Conclusion & Future Work **71**
5.1 Conclusion 71
5.2 Future Work 72

Referências **73**

Appendix A **77**
Mode3 Hardware Board Documentation 77

List of Figures

1.1 CO2 emissions (metric tons per capita) - EU between 1960 to 2018 2

2.1 3-tier model as used in OCPP 6

2.2 Electric vehicle interface standards 6

2.3 Difference on how AC and DC chargers deliver power to the battery 7

2.4 Pinouts for Type 2 female (charging station outlet/vehicle connector) and male (vehicle inlet/outlet side plug) electric vehicle charging plugs 8

2.5 Combo 2 (left), compared to IEC Type 2 (right): two large direct current (DC) pins are added below, and the four alternating current (AC) pins for neutral and three-phase are removed 9

2.6 Mode 1: Household socket and extension cord 11

2.7 Mode 2: Domestic socket and cable with a protection device 12

2.8 Mode 3: Specific socket on a dedicated circuit 12

2.9 Mode 4: Direct current (DC) connection for fast charging 13

2.10 EVSE connects with a Plug-In Electric Vehicle (PEV) 14

2.11 System compliant with HomePlug Green PHY 14

2.12 Reflection of current roaming landscape with the difference of OCPI shown as it is not forced to a single company's roaming hub 18

2.13 Monolithic architecture base example 19

2.14 Microservices architecture base example 21

2.15 Basic structure of message-oriented middleware 22

2.16 Apache Kafka example 23

2.17 RabbitMQ example 24

2.18 RabbitMQ exchange types 25

3.1 Architecture Overview 28

3.2 Mobile application mock-up: Part 1 29

3.3 Mobile application mock-up: Part 2 30

3.4 Topic Exchange handling a "status" message in the Charging Station 32

3.5 Charge Point module highlighted in the Architecture 33

3.6 CSMS to Charging Station Request 33
3.7 Decision Point module highlighted in the Architecture 35
3.8 Decision Point Threads 35
3.9 Sequence Diagram:: Cold Boot Charging Station 36
3.10 Sequence Diagram: Reserve a specified Connector at a Charging Station 37
3.11 Sequence Diagram: Authorization using a start button 39
3.12 Mode3 Protocol Driver module highlighted in the Architecture 41
3.13 Mode3 Driver Finite State Machine: States and Transitions 42
3.14 Sequence Diagram: EVPlug 43
3.15 Role of Simulator in the system 44
3.16 Simulator Graphical User Interface 45
3.17 Energy Driver module highlighted in the Architecture 45
3.18 Sequence Diagram: Request PowerModule 46
3.19 Parking Sensor module highlighted in the Architecture 48
3.20 Sequence Diagram: Start Transaction options - ParkingBayOccupancy 48
3.21 Interface Layout of the Parking Sensor module 49
3.22 HMI module highlighted in the Architecture 49
3.23 Sequence Diagram: Authorization using PIN-code 50
3.24 HMI Graphical User Interface 51
3.25 RFID CardReader module highlighted in the Architecture 51
3.26 Sequence Diagram: Authorization using RFID 52
3.27 RFID Card Reader Graphical User Interface 52

4.1 Before Arriving: Create/Cancel Reservation 53
4.2 Simulator and HMI: Before Arriving 54
4.3 At arrival: ParkingSensor detects EV 54
4.4 At arrival: EV Plug 54
4.5 Simulator and HMI: EVPlug 55
4.6 Authorization: RFID Card 55
4.7 Authorization: PIN-code 55
4.8 Simulator and HMI: Authorization 56
4.9 After Authorization: Displaying tariff 56
4.10 Start Charging: Start Button 57
4.11 Start Charging: Remote Start 57
4.12 After PWM: Requesting Current decided by PWM 57
4.13 Simulator and HMI: PWM 58
4.14 During Charging: Connector Status 58
4.15 Simulator and HMI: Charging Ongoing 59

4.16 During Charging: Reduce Energy Consumption 59
4.17 During Charging: Enable/Disable Power Relay 59
4.18 During Charging: PowerModule Overheating (for CCS) 60
4.19 Simulator and EnergyDriver: Updating Current 60
4.20 Stop Charging: Vehicle Fully Charged 61
4.21 Stop Charging: Stop Button 61
4.22 Simulator and HMI: Waiting for cable unplug 61
4.23 After Charging: EV Unplug 62
4.24 After Charging: Display Charging Cost 62
4.25 Mode3Driver Finite State Machine 63
4.26 Time Requirement Analysis: normal Mode3 66
4.27 Time Requirement Analysis: with optional EnergyDriver 67

1 Frame byte structure 77
2 Mode3 Change State CMD 77
3 Mode3 Status Request CMD 78

List of Tables

2.1 Commonly used terms 5

2.2 Commonly used terms 6

2.3 Pulse Width Modulation signal: voltage possibilities 10

2.4 Control Pilot (CP) duty cycle definitions 10

2.5 Charging Modes in IEC 61851 10

2.6 Operations specified in OCPP 1.5 16

3.1 Description of Database.db tables (Part 1) 40

3.1 Description of Database.db tables (Part 2) 41

3.2 Waiting states' transition conditions 42

3.3 PowerModule Internal States 47

4.1 Finite State Machine: Unit Testing (Part 1) 64

4.1 Finite State Machine: Unit Testing (Part 2) 65

Acronyms

AC Alternating Current
AMQP Advanced Message Queuing Protocol
API Application Programming Interface
CAN Controller Area Network
CCS Combined Charging System
CSMS Charging Station Management System
CSO Charging Station Operator
DC Direct Current
EV Electric Vehicle
EVSE Electric Vehicle Supply Equipment
GUI Graphical User Interface
HLC High Level Communication
HMI Human-Machine Interface
HTTP Hypertext Transfer Protocol
IEC International Electrotechnical Commission
ISO International Standards Organisation
LLC Low-Level Communication
MOM Message-Oriented Middleware
MSP Mobility Service Providers
OCPP Open Charge Point Protocol
PLC Powerline Communication
PWM Pulse Width Modulation
RFID Radio-Frequency Identification
V2G Vehicle-to-grid
XML Extensible Markup Language

CHAPTER 1

Introduction

1.1 Context

With the increasing concern about the impact of human growth in climate change, multiple new technologies have been emerging. New technologies that allow the use of renewable energies or simply energies that have a less pollution footprint. One of these areas is electric vehicles, where we see not only growth in terms of the electric vehicles themselves (more affordable price, more charging options and bigger battery lifespan) but also efforts in trying to fix the issue of lack of interoperability in the electric vehicle charging infrastructure with the creation of standards.

1.1.1 Impact of the transportation sector in CO2 emissions

In the European Union (EU) a growth in CO2 emissions can be seen between 1960 to 2018, as shown in Figure 1.1. Even though a steadily decrease can be seen since 1980 due to environmental policies and measures, the data is per capita and the population keeps increasing along side the number of vehicles which is expected to more than double by 2050 [1].

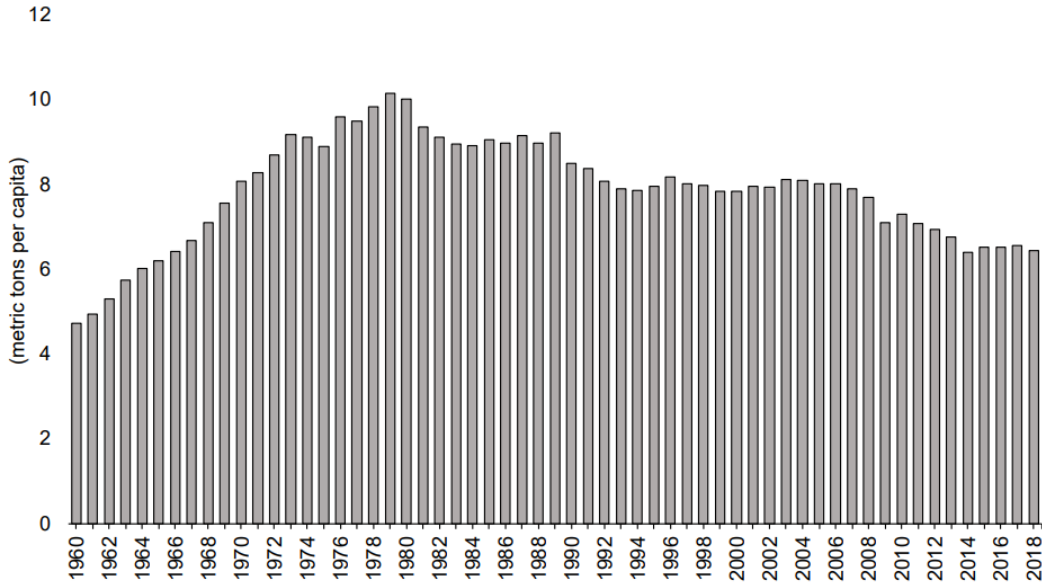


Figure 1.1: CO2 emissions (metric tons per capita) - EU between 1960 to 2018 [1]

Consequently, even though a downwards trend in CO2 emissions can be seen, the share of the transport sector is actually increasing. When compared to 1990, the share of total CO2 emissions of transportation increased from 14.8% in 1990 to 24.6% in 2018, with the actual transportation CO2 emissions increasing by 29% in that time span[1].

Given that the transportation sector is the most significant contributor to CO2 emissions in the EU [1], taking measures to reduce CO2 emissions in this sector is crucial to tackling environmental issues [2].

The European Commission has made energy transition one of its main goals and it did so by creating the "European Green Deal"[2]. The Green Deal is a roadmap of key policies with the EU's climate agenda in mind [3].

One of the steps in the roadmap is accelerating the shift to sustainable and smart mobility, with visions set out for 2030, 2035 and 2050. EU's environment ministers have come to an agreement to introduce a 100% CO2 emissions reduction target by 2035, with an effective ban on the sale of new petrol and diesel cars from 2035 onward. The vision for 2050 consists of zero-emissions technology for all modes of transport [4].

1.1.2 Electric Vehicles (EV) as a solution

Electric Vehicles come as an alternative for fossil-fuel powered vehicles in order to achieve the wanted zero-emissions transportation. But in order to ease Electric Vehicle (EV) adoption by the population, some quality of life issues must be solved.

Range anxiety is a term used to describe EV users that suffer anxiety caused by:

- the distance their electric vehicle can handle, which is limited by the batteries capacity which is smaller than the normal fossil-fuel vehicles;

[1] https://www.eea.europa.eu/ims/greenhouse-gas-emissions-from-transport

[2] https://ec.europa.eu/info/strategy/priorities-2019-2024/european-green-deal_en

- the lack of charging stations compatible with the purchased vehicle or just lack of charging locations in general;
- the time it takes to charge the vehicle, since it is a longer process when compared to a gas station, and how that affects planning a trip or the daily routine;

The sooner we can mitigate the issue of range anxiety, the sooner we will see a vast increase in EV penetration [5].

Using an electric vehicle on a regular basis requires safe and easy-to-use charging installations [6]. To get to the point of wide adoption, the EV landscape also needs to be standardized, in order to give more options to EV users.

With this in mind, several protocols and standards are in constant development. In some occasions, the European Commission has decided which standard should be used within Europe in order to standardize the industry, like the Type 2 charging plug[3] in 2013.

1.1.3 Motivation

The main motivation of this project is knowing there are currently Monolithic implementations in the EV landscape facing difficulties when wanting to implement new code or updates, which on a monolithic architecture require an arduous planning or rewriting of existent code. Therefore, there is a need for developing integrated electric chargers with a service-oriented architecture.

Also, even though the newer version of Open Charge Point Protocol (OCPP) brings some advantages to charging stations, it is not backwards compatible, which requires companies to reimplement their OCPP logic.

This project's idea comes as a solution to both of these problems, by structuring and building the Charging Station software from scratch with scalability and easier updatability in mind, while also using this opportunity to already develop the backend OCPP with the newest version.

1.2 Objectives

The goal of this Dissertation is to conceptualize and develop the Charging Station with a service-oriented architecture with backend OCPP and as a distributed environment, allowing multiple chargers in a grid, capable of balancing energy in real-time.

The Charging Station must communicate via OCPP with a Central System that was developed in another dissertation called "Electric Vehicle Charging Control System via OCPP 2.0" done by Alexandre Machado [7]. Further explanation of the starting point of the project is done in Section 3.2.

Considering that there are time requirements to be met in Charging Station to EV communication, it is important to choose fast and reliable technologies when developing these components.

[3] `http://www.mennek.es/index.php?id=latest0&L=1&tx_ttnews%5btt_news%5d=929&cHash=1fd716bc2fa538f0f516aac1b4d8d8ba`

1.3 Document structure

In the "State of the Art" chapter the main standards for plugs and charging, and the main protocols used for communication within the EV infrastructure are analyzed, along with possible design architectures and different message brokers, weighing their advantages and disadvantages so the best options for our end goal can be chosen.

The "Implementation" chapter presents the whole system architecture and specifies the starting point of this dissertation in terms of previously implemented parts of the project. Then it explains how the communication between modules was implemented with a message broker, followed by a section for each module of the Charging Station where their logic, functionalities and role within the system are explained.

The "Results" chapter shows the functionalities achieved by the collaboration of all previously explained modules. It also discusses the testing of the system in regards to guaranteeing functionality, meeting time requirements and verifying that previously implemented tests were not compromised.

Finally, the "Conclusion & Future Work" chapter presents the final considerations of the dissertation and provides suggestions for future improvements and developments.

CHAPTER 2

State of the Art

2.1 Charging Infrastructure

Typically, the Charging Infrastructure consists of: an EV that connects to an Electric Vehicle Supply Equipment (EVSE) in order to charge, a Charging Station that has one or more EVSEs, and the Charging Station Management System (CSMS). A more in depth description of these concepts is described in Table 2.2.

EV Driver [Actor]	The Driver of an EV that wants to charge his EV at a Charging Station. The Driver parks close to an EVSE and plugs the charging cable into one of the available Connectors.
Charging Station [Device]	The Charging Station is the physical system where an EV can be charged. A Charging Station has one or more EVSEs.
Electric Vehicle Supply Equipment (EVSE) [Device]	An EVSE is considered as an independently operated and managed part of the Charging Station that can have multiple connectors but can only deliver energy to one EV at a time.
Connector [Device]	An independently controlled and managed electrical outlet on an EVSE. EVSEs may have multiple Connectors in order to provide multiple physical socket types, for example, they can have a Connector for Mode3, another for CCS and another for CHAdeMO, therefore covering a bigger pool of electrical vehicles.
Charging Station Management System (CSMS) [System]	Contains the information of multiple Charging Stations and has the information for authorizing Users for using the Charging Stations it controls.

Table 2.1: Commonly used terms (Part 1) [8]

Charging Station Operator (CSO) [Actor]	A party that manages a CSMS and therefore can control and manage multiple Charging Stations via the CSMS.
Energy Management System (EMS) [Actor]	An external system that may impose charging limits on a single Charging Station or on a CSMS.

Table 2.2: Commonly used terms (Part 2) [8]

To better understand the relationship between Charging Stations, EVSEs and Connectors, the following Figure 2.1 illustrates well these previously explained concepts.

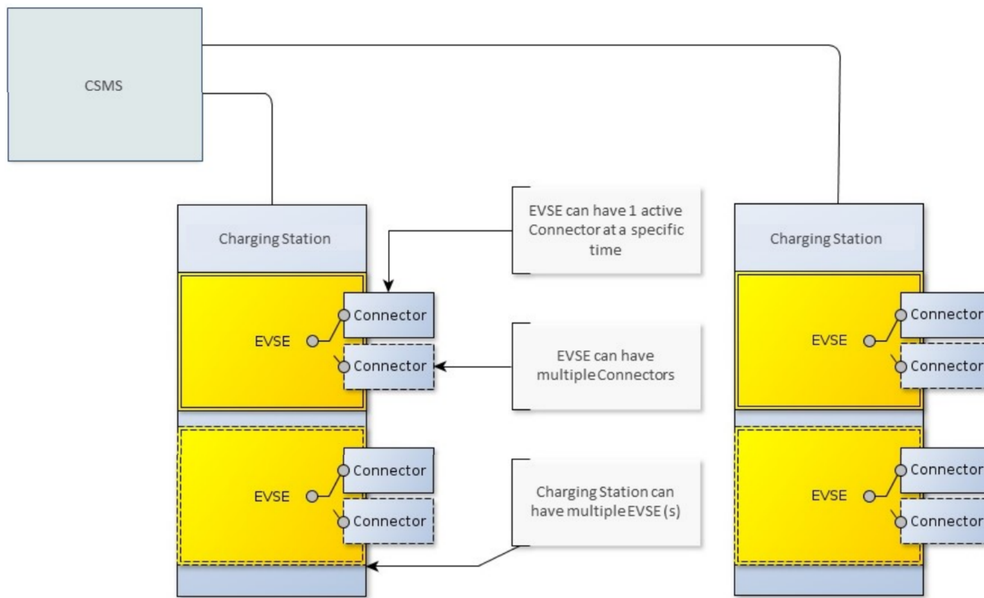


Figure 2.1: 3-tier model as used in OCPP [9]

In order to achieve interoperability inside the charging infrastructure, multiple standards and protocols have been continuously developed. As observed in Figure 2.2, the communication between an EV and an EVSE that is part of a charging station, is defined in IEC 61851 and ISO 15118. These standards are explained in Section 2.4 and 2.5, respectively. The communication between the Central System (or CSMS) and the Charging Stations is the Open Charge Point Protocol (OCPP), explained in Section 2.6.

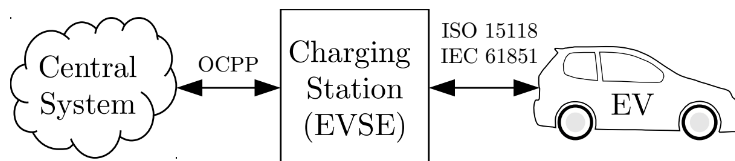


Figure 2.2: Electric vehicle interface standards [10]

2.2 Understanding electric current

Before discussing the different types of charging stations, we first need to understand the different types of electric current used in them.

Current is the movement of electrons through a conductor. When it comes to EV charging, there are two important methods of electric current: Alternating Current (AC) and Direct Current (DC). The main difference between AC and DC lies in the direction in which the electrons flow [11].

Alternating current (AC) is a method in which the positive and negative sides are switched periodically and thus the direction of the flow of electricity changes accordingly causing the current to alternate, while in Direct current (DC) the electrons flow steadily in a single direction and the voltage is always consistent.

The power that comes from the power grid or, for example, household domestic sockets, is always AC. This happens since AC can be transported over long distances efficiently. However, the type of electricity that is used in the power circuitry of electric devices and stored in electric vehicles' batteries is DC. As a result, in order to charge an electric vehicle, two possibilities are presented below.

Charging stations that provide AC charging and are typically found at home, workplace settings or in public locations. This form of charging is simple yet it is slower since the alternated current provided by the charging station needs to be converted to DC to be able to charge the EV. This conversion happens inside the vehicle, on the on-board charger, which limits the speed of the charging to the AC to DC converter's capabilities.

Another option is DC charging stations that are able to bypass the EV on-board battery charger with the energy transfer being accomplished via direct current directly to the vehicle's battery. This is possible due to the AC to DC converter being inside the charging station itself (outside of the vehicle), which may provide faster charging speeds if this converter has higher power than the on-board one. This method of charging is also known as fast charging.

Figure 2.3 illustrates the two previously explained charging stations.

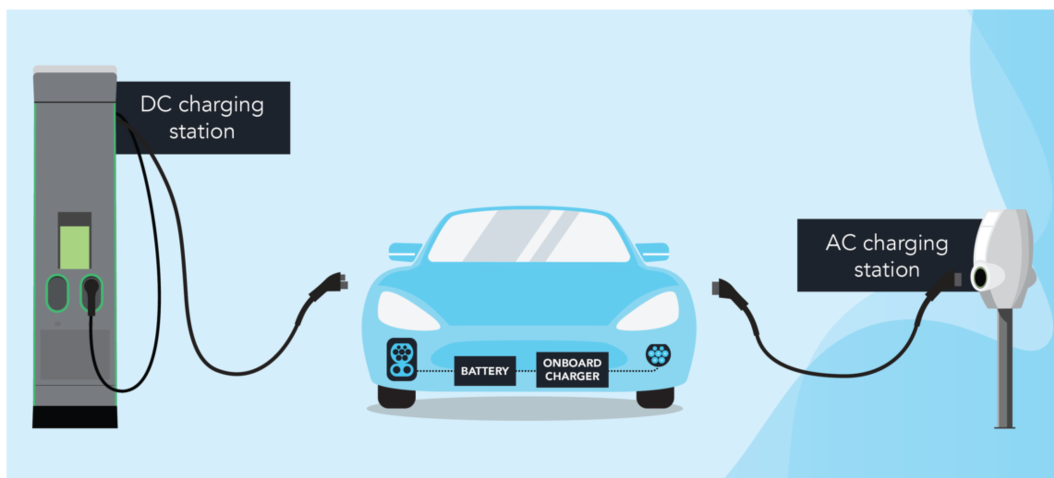


Figure 2.3: Difference on how AC and DC chargers deliver power to the battery[1]

Fast charging comes as a possible solution to range anxiety due to the smaller amount of time it takes to fully charge the vehicle.

Some real-life studies such as [12] try to determine the actual required daily vehicle range. The study was done in Atlanta, Georgia in 2004, one of the areas with the highest daily vehicle distance traveled per capita in the US, and the results showed an average daily distance of 72km. Most importantly, the study concluded that for a daily routine, a charging time of around 2-4 hours would be enough since it is the average time the studied individuals stayed at a certain destination, which included the workplace, shopping malls or other public spaces.

However, around 10% of the users in the study had at least a day during the year where they drove more than 1000km. For those specific occasions, a charging time below 30-60 minutes was considered convenient. Such a low charging time can be achieved using DC/fast charging.

A more in-depth look on charging modes is done at Section 2.4.2, where the charging modes defined by the international standard 61581 are described.

2.3 Type 2 connector in Europe

The International Electrotechnical Commission (IEC)[2] type 2 connector, part of the IEC 62196 standard series, is used for electric vehicles conductive charging mainly within Europe [13], ever since it was declared standard by the EU Commission[3] in January of 2013.

This type of plug is in accordance with the IEC 61851 standard, which specifies the lowest communication layer between the EV and the charging station. For the communication, the control pilot (CP) and proximity pin (PP) shown in Figure 2.4 are utilized [10]. The control pilot enables bidirectional communication by using a Pulse Width Modulation (PWM) signal, generated by the EVSE. Further explanation of the communication and the IEC 61851 standard can be found in Section 2.4.1.

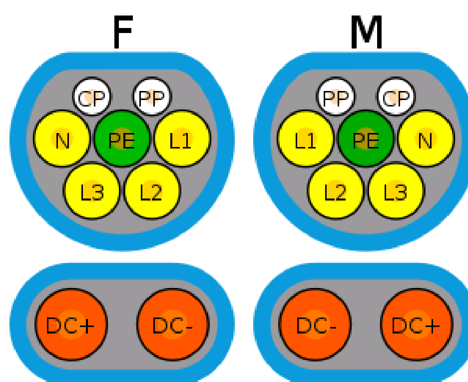


Figure 2.4: Pinouts for Type 2 female (charging station outlet/vehicle connector) and male (vehicle inlet/outlet side plug) electric vehicle charging plugs [14]

[1]https://blog.evbox.com/uk-en/ev-charging-levels

[2]https://www.iec.ch/

[3]http://www.mennek.es/index.php?id=latest0&L=1&tx_ttnews%5btt_news%5d=929&cHash=1fd716bc2fa538f0f516aac1b4d8d8ba

The Protective Earth (PE) pin acts as a full-current protective earthing system, while the Neutral (N), Line 1 (L1), Line 2 (L2) and Line 3 (L3) pins are all used in three-phase AC charging, and only the Neutral (N) and Line 1 (L1) in single-phase AC charging. Further explanation of the types of AC charging can be found in Section 2.4.2.

The Combined Charging System (CCS) is a standard for charging electric vehicles which can use a Combo 2 connector, which is an extension of the Type 2 connectors but with two additional DC contacts to allow fast charging.

In Figure 2.4 we can see the male inlet found in most vehicles produced in Europe, which has the two additional DC contacts. This way, both connectors shown in Figure 2.5, Combo 2 for DC charging (left) and Type 2 for AC charging (right), can plug into the inlet, achieving the possibility of different types of charging while using the same socket.

Figure 2.5: Combo 2 (left), compared to IEC Type 2 (right): two large direct current (DC) pins are added below, and the four alternating current (AC) pins for neutral and three-phase are removed [14]

CCS has been required on electric vehicles sold in European Union since 2014, and it competes with Tesla's proprietary connector used by its Supercharger network, China's GB/T charging standard and the Japanese CHAdeMO [15].

2.4 IEC 61581

IEC 61581-1 [16] is a standard where the general requirements for an EV conductive charging system are described. Most importantly, it characterizes types of EV supply equipments, specifies an option on how the connection between the EVSE and EV can be done, defines what to take into consideration to achieve electrical safety and also describes a possible alternative to the communication previously specified, among various other subjects.

2.4.1 Low-Level Communication (LLC)

As briefly introduced in Section 2.3, IEC 61581 proposes a low-level communication using the control pilot (Figure 2.4) based on a PWM signal, which can indicate various EV connection states and therefore providing a way to handle time-critical state changes [17].

The PWM signal has an amplitude of 12 Volts and a frequency of 1kHz.

The EV indicates its state to the EVSE by changing the amplitude of the signal. Therefore, the EVSE can assume one of the states defined in IEC 61581 simply by measuring the peak voltage. These states are described in the following Table 2.3.

Measured Voltage	State	Description
+12 V	State A	No EV connected to the EVSE
+9 V	State B	EV connected to the EVSE, but not ready for charging
+6 V	State C	Connected and ready for charging, ventilation is not required
+3 V	State D	Connected, ready for charging and ventilation is required
+0 V	State E	Electrical short to earth on the controller of the EVSE, no power supply
-12 V	State F	EVSE is unavailable

Table 2.3: Pulse Width Modulation signal: voltage possibilities [16]

On the other hand, the control pilot pin can also inform the connected vehicle on how much current it is allowed to draw during charging, and it does so by varying the duty cycle of the 1kHz signal. The control pilot duty cycle definitions are shown in Table 2.4.

PWM duty cycle d	Current draw allowed
$d < 3\%$	No charging allowed
$3\% \leq d \leq 7\%$	Force high-level communication protocol according to ISO 15118 (Section 2.5)
$7\% \leq d < 8\%$	No charging allowed
$8\% \leq d < 10\%$	Max. current consumption for AC charging is 6 A
$10\% \leq d \leq 85\%$	Available current = duty cycle * 0.6 A
$85\% < d \leq 96\%$	Available current = (duty cycle - 64) * 2.5 A
$96\% < d \leq 97\%$	Max. current consumption for AC charging is 80 A
$d > 97\%$	No charging allowed

Table 2.4: Control Pilot (CP) duty cycle definitions [5]

2.4.2 Charging Modes

IEC 61851-1 describes four different charging modes and defines minimal requirements to ensure safety of charging systems [18]. These four modes are briefly described in Table 2.5, and further explained in the following subsections.

Charging Mode	Charging Setup
Mode 1	single-phase 250V, three-phase 480V, 16A
Mode 2	single-phase 250V, three-phase 480V, 32A
Mode 3	Charging with dedicated EVSE equipment
Mode 4	Charging with an external charger

Table 2.5: Charging Modes in IEC 61851 [18]

Mode 1

While Mode 2, 3 and 4 require, for safety reasons, the dedicated PWM signal provided over the Control Pilot, Mode 1 does not [18]. This is because Mode 1 describes charging the vehicle via a standard power outlet with a simple extension cord, as shown in Figure 2.6, and, consequently, with limited safety measures [19].

The connection to the AC supply network via the outlet must not exceed 16 A. It also must not exceed 250V in single-phase AC charging or 480V in AC three-phase charging [16].

Figure 2.6: Mode 1: Household socket and extension cord [6]

It is required that the electrical installation complies with safety regulations and must have an earthing system and a circuit breaker to protect against overload [6].

Besides the difficulty in guaranteeing that the defined requirements are achieved, since some electrical systems of old households may be faulty, the overall lack of safety measures in this charging mode has made it outlawed in several countries [20].

Mode 2

Mode 2 also consists of charging the vehicle via a standard power outlet, but doing so with a special cable, also known as "occasional use cable", which has a built-in protection device and is usually supplied when purchasing an EV from the manufacturer [19]. This solution is particularly expensive due to the specification of the cable [6].

The cable provides the following features [20]:

- In-cable RCD: prevents fatal electric shock from touching
- Over-current protection
- Over-temperature protection
- Detects if Protective Earth from wall socket is functional

Power only flows to the vehicle if the protection device has detected the Protective Earth from the socket, does not detect over-current or over-temperature, and if it detects a plugged in vehicle that has requested power. The detection of the vehicle and of the charging request is done via the Control Pilot pin.

With these conditions a moderate level of safety is assured and it is considered the minimum standard today for charging an EV [19].

The connection to the AC supply network via the outlet must not exceed 32 A. It also must not exceed 250V in single-phase AC charging or 480V in AC three-phase charging [16]. An illustration of this charging mode is presented below in Figure 2.7.

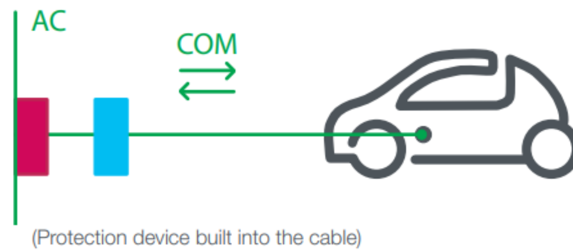


Figure 2.7: Mode 2: Domestic socket and cable with a protection device [6]

Mode 3

Mode 3 defines charging the vehicle by connecting it to a power supply system permanently connected to the electrical network via a specific socket and a dedicated circuit, as shown in Figure 2.8. Charging stations operating in Mode 3 are usually present in public places or installed at home [20], and they allow load-shedding so that electrical household appliances can run without issues during vehicle charging [6].

Mode 3 uses the standardized communication protocol with the control pilot function extending to also control equipment in the EVSE [19].

Because it uses a dedicated EV charger, the power range is higher, from 3.7kW up to 22kW AC [16]. This higher power range enables faster charging of electric cars, when compared to Modes 1 and 2, but may be limited by the on-board charger performance in converting AC to DC.

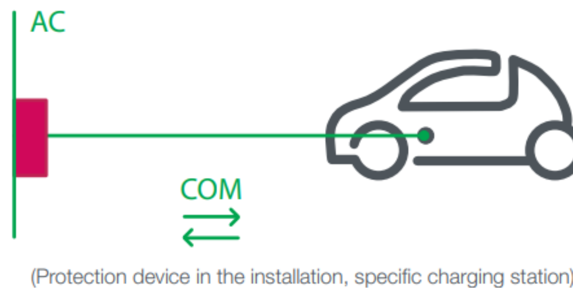


Figure 2.8: Mode 3: Specific socket on a dedicated circuit [6]

Mode 4

In Mode 4 the electric vehicle is connected to the power grid through an external charger, with the control and protection functions as well as the vehicle charging cable installed permanently in the installation [6], shown in Figure 2.9.

It is a wired-in DC charging station since the charger is part of the charging station and consequently it provides DC directly to the vehicle's battery, bypassing the on-board charger.

Charging of the EV is much faster than in previous modes, as the electrical power charging range is higher than 24kW, making it possible to achieve a empty to fully charged battery within an hour or less [5].

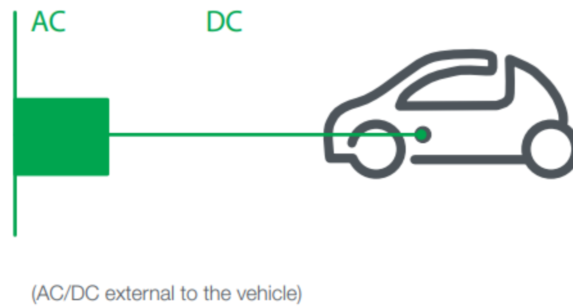


Figure 2.9: Mode 4: Direct current (DC) connection for fast charging [6]

When using a DC charging station, the initial connection is also done via PWM to start communication and then the EVSE checks if it is ready to give current. The rest of the communication is done via a High Level Communication (HLC), more specifically called Powerline Communication (PLC), which is defined in standard ISO 15118 explained below in Section 2.5.

2.5 ISO 15118

International Standards Organisation (ISO)[4] 15118 is an international standard that specifies a High Level Communication (HLC) protocol between EVs and charging stations. Such protocols are needed for authentication, clearing and coordination of charging processes to prevent local substation blackouts [21]. It defines smart charging as it enables active charge control, demand and response management, implements plug and charge and simplifies payment processes [10].

Being capable of authentication allows the plug-and-charge functionality to work, which consists of charging authorization being initiated simply by connecting the car to a charger. This simplifies the charging experience of the EV user and therefore can accelerate EV adoption. This process is enabled by a digital certificate located in the vehicle which allows it to communicate with the Charging Station Management System, providing automatic authentication and billing, bypassing the need for authentication via a RFID card [22].

Preventing local substation blackouts comes from the fact that this type of protocols enable bi-directional EV charging, also known as Vehicle-to-grid (V2G). With V2G, EVs are able to send energy back into the grid when needed, consequently reducing peaks and ensuring a more stable and reliable grid [23].

Powerline Communication (PLC) is the proposed technology for the HLC protocol ISO 15118, which describes the Vehicle-to-Grid Communication Interface in case of conductive charging [21].

2.5.1 Power Line Communication

Described in ISO 15118-3 [24], PLC uses the same communication medium as Low-Level Communication (LLC), the Control Pilot integrated into the charging cord, and it does so by modulating a high-frequency signal over the Control Point contact.

[4] `https://www.iso.org/`

Although utilizing PLC for AC charging is optional, it is required for DC charging [25]. The biggest example in Europe is Combined Charging System (CCS) since it uses PLC for the communication between the car and charger, and also uses HomePlug Green PHY as the physical layer for the V2G protocol.

HomePlug Green PHY

HomePlug Green PHY is the standard for PLC signals used in vehicle charging called out in ISO 15118 [26].

For future comparison, a simple connection between an EVSE and EV using basic signalling can be seen in Figure 2.10.

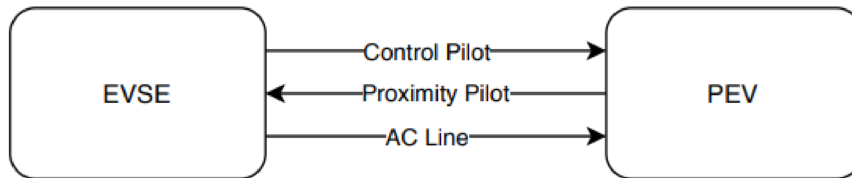


Figure 2.10: EVSE connects with a Plug-In Electric Vehicle (PEV) [27]

A possible HomePlug GP implementation is making it modulate over the Control Pilot wire, since the amplitude of the HomePlug GP signal is so low and the frequency so high when compared to the Control Pilot signal that there won't be interference with the older PWM standard [27].

Figure 2.11 demonstrates a possible implementation, where the zero cross signal generated from the AC line is used to time sync messages.

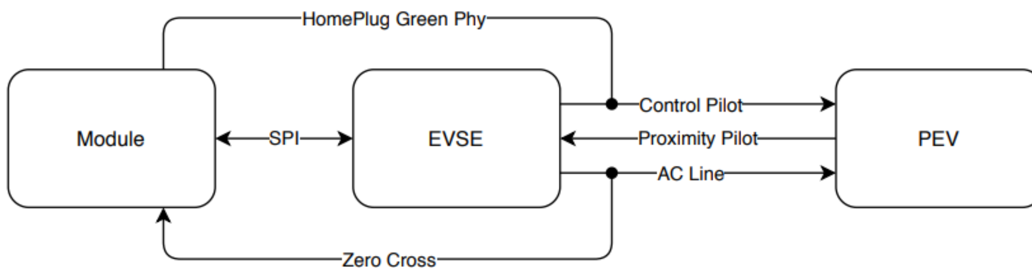


Figure 2.11: System compliant with HomePlug Green PHY [27]

PLC Drawbacks

The power line channel is a very harsh and noisy transmission medium that is very difficult to model because it is frequency-selective, impaired by noise and also the structure of the grid differs between grids sometimes even in the same country [28].

The usage of PLC also comes with a few drawbacks, mainly the difficulty in isolating the signal, since the high-frequency signals cross couple between wires, breakers and distribution transformers [5]. PLC modems are designed to deal with these drawbacks but utilizing them increases complexity and production cost of an EVSE.

A simple solution would be having dedicated communication wires in the cord, since a communication between an EVSE and an EV only requires a direct point-to-point link. Sadly, there are no dedicated communication wires, but IEC 61851 suggests a possible solution in IEC 61581 annex D [16]. This alternative is considered in Section 2.5.2 below.

2.5.2 IEC 61581 Annex D - extended pilot function

Annex D explains the possibility of using the Control Pilot wire for single-wire communication, hence avoiding the rather complex PLC. Therefore the Control Point wire can be used for low-voltage wired digital point-to-point communication, which allows the usage of mature communication standards such as Controller Area Network (CAN) [5].

The way it works is firstly the micro-controllers in the EVSE and the EV detect if Annex D communication is supported on both sides. If they do, the microcontrollers, that are connected to serial UART ports, can mirror the incoming communication directly to CAN packets. With this possibility, for example, Linux can be used to establish a full IP communication between the EVSE and the EV via the serial link [5].

If at least one side does not support this functionality, they default to the typical pilot function of utilizing a PWM signal.

Annex D can achieve rates up to 500kbps which may be a limitation. In the case that it is indeed a limitation, PLC can then be used to increase the communication speeds.

2.6 Open Charge Point Protocol

Acting as an interface between the CSMS and the charging stations it manages is the Open Charge Point Protocol (OCPP). OCPP is a protocol maintained by the Open Charge Alliance (OCA)[5]. Initially created in 2009, it defines an open back-end protocol for charge spots aiming at reducing integration and investment costs in charge spot development and operation [17].

It is currently the electric vehicle industry standard which provides a common ground between software and hardware suppliers. Having a protocol respected by both suppliers, minimizes planning in terms of ending the need to create a new type of communication and also ensures interoperability among different ecosystem components.

At the beginning, OCPP only had basic functionality like starting and stopping charging transactions and basic remote access for configuration, diagnostics and firmware updates. Nowadays there are three main versions of OCPP: OCPP 1.5, OCPP 1.6 and the more recent OCPP 2.0.

OCPP 1.5 specifies 25 different operations, with 10 of them being initiated by the charge point in the Charging Station and the remaining 15 being initiated by the CSMS [8]. These operations are listed in Table 2.6 below.

[5] `https://www.openchargealliance.org/`

Initiated by CSMS:	Initiated by Charge Point:
· Cancel Reservation · Change Availability · Change Configuration · Clear Cache · Data Transfer · Get Configuration · Get Diagnostics · Get Local List Version · Remote Start Transaction · Remote Stop Transaction · Reserve Now · Reset · Send Local List · Unlock Connector · Update Firmware	· Authorize · Boot Notification · Data Transfer · Diagnostics Status Notification · Firmware Status Notification · Heartbeat · Meter Values · Start Transaction · Status Notification · Stop Transaction

Table 2.6: Operations specified in OCPP 1.5 [8]

OCPP 1.6 [29] builds on OCPP 1.5 by adding on top of the functions already defined. It uses the SOAP framework to send messages in accordance to the Extensible Markup Language (XML) standard. This allows to send messages, pictures and even executable code with the advantage of being decipherable content [8]. This new version also eases the implementation of smart charging, which means that it allows to postpone or throttle charging based on real-time grid load of availability of energy [30].

2.6.1 OCPP 2.0.1

OCPP 2.0.1 [31] is the most recent adaptation, released in April of 2018 [8]. It brings various new features and improvements to cement it as a more thought out communication protocol. Open Charge Alliance lists the following added or improved functionalities when compared to the previous version[6]:

- Device Management: New features to modify configurations and monitor charging stations. Useful for Charging Station Operators (CSOs) that manage multiple charging stations from different vendors, since new features also include inventory reporting and improvements on state reporting.
- Improved Transaction Handling: Also useful for Charging Station Operators because it has better handling for large amounts of transactions, achieved with the usage of a single message for all transaction related communications.
- Added Security: Addition of security profiles for Charging Station and CSMS authentication, secure firmware updates, security related logging paired with event notifications, key management for client-side certificates for the authentication process and implemented secure communication with TLS.
- Added Smart Charging functionalities: For architectures with an Energy Management System (EMS), its inputs are now directed to a Charging Station. Added the idea of

[6] `https://www.openchargealliance.org/protocols/ocpp-201/`

a local controller inside the CSMS, which acts as a brain by controlling and making automated smart decisions.

- Support for ISO 15118: the standard allows a lot of the new features such as Plug and Charge and Smart Charging considering the input from the EV.
- Display and messaging support: to provide information on rates and tariffs by displaying it on the Human-Machine Interface (HMI) before the EV driver begins charging.
- More authorization options: besides the standard of authentication via a Radio-Frequency Identification (RFID) card, there are now more authentication options such as ISO 15118 Plug and Charge, using a credit card, a PIN-CODE or a simple start button coupled with a mobile application.

OCPP 2.0.1 also brings the possibility to use the much more compact JSON notation instead of the SOAP framework defined in previous versions [10].

The current biggest disadvantage that is complicating the adoption of this newer version is the fact that OCPP 2.0 is not backward compatible with the existing previous versions. This means that older clients are required to perform sizeable reworks in order to get the advantages of this new version.

When starting a new project and developing all code from scratch, it would be recommended to adopt the newer OCPP version in order to benefit from all the previously explained features and improvements.

2.7 EV Roaming

EV roaming is the cooperation between a Mobility Service Providers (MSP) and multiple Charge Point Operators (CPO) to give their customers more options to charge their vehicle. For an EV Driver that means that just by being under contract with a MSP they are able to charge their vehicle on a charging station whom the MSP has a contract with, even if the Driver does not.

The benefits for MSPs is that they can offer access to charging networks outside their own, making them more attractive to customers. For CPOs the benefit comes from the increased utilization rates of their charging network [32].

Some protocols have been designed to help implement the infrastructure needed for roaming. The main functionalities needed are authorization, billing, information about the charge point and giving remote start and stop commands [33].

Even though no standard has been adopted yet [33], some protocols already exist such as OCHP, OICP, eMIP, OCPI, etc. Besides OCPI, the others are developed by companies that force their own roaming hub to be used, as illustrated in Figure 2.12.

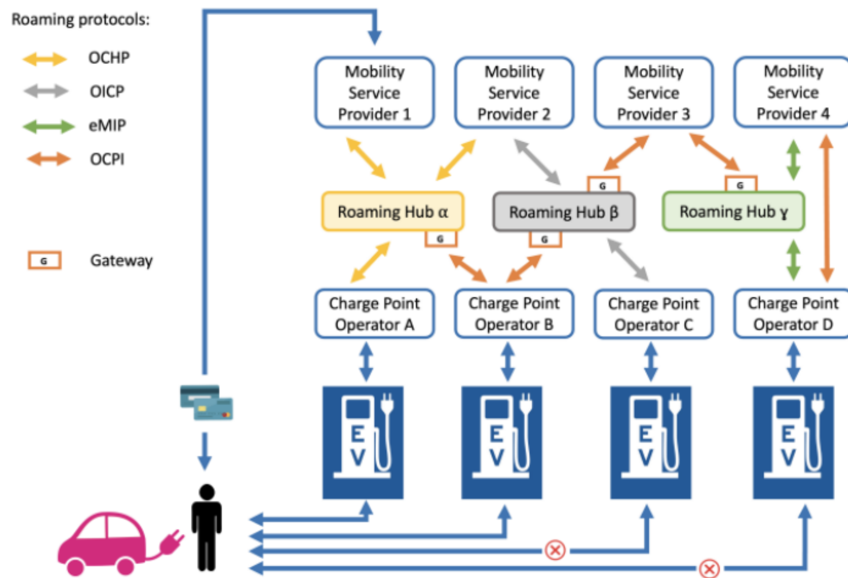


Figure 2.12: Reflection of current roaming landscape with the difference of OCPI shown as it is not forced to a single company's roaming hub [33]

Furthermore, for some MSPs offering exclusive access to their network is part of their business, with the biggest example being Tesla. Such companies prefer being the biggest in a fragmented infrastructure, than a part of an interconnected infrastructure [32].

2.8 Architecture

An architecture is the fundamental structure of a software system [34]. It defines the principles to be followed while designing the software and it determines how the system is organized in terms of its components and how they interact. Having a foundation in mind goes hand in hand with planning, which increases the chance of a successful development and, more times than not, helps to improve the quality of what is built on top of it.

Choosing an architecture allows to have in consideration which non-functional requirements we are looking for, such as maintainability, interoperability, security and/or performance. The importance of this decision is what makes the choice of an appropriate architecture so important in the primary phase of software development [34] in order to avoid future hard and costly reworks.

2.8.1 Monolithic Architecture

A monolithic architecture is the traditional unified model in the design of software programs, where all functionality is encapsulated in a single application [35].

If the system is represented as single and indivisible with all of its logic and code present in one place and highly interdependent components, the system can be classified as monolithic.

Using this approach is very appealing at first since current development tools and IDEs give a lot of support in developing monolithic applications. Testing, deployment and initial scaling are also easier as all the business logic is placed in one program and therefore can be

run in one process [36]. In addition, it requires less planning since performing the division of the logic into different components and establishing communication between them doesn't need to be thought-out.

However, using this architecture comes with very clear disadvantages long-term. Due to the code being all in one place and it being interdependent, the system becomes highly coupled and hard to maintain [35].

As the codebase grows in size, it becomes very hard to do even small updates which makes development time slower [37]. Changes may also cause unanticipated changes to other elements and, even if it does not, there is the need to redeploy the whole program, which also has a larger startup time.

In terms of management and teamwork, this architecture also has some drawbacks since the application can be difficult to understand, which hurts not only productivity (slower IDE, more time to understand the logic, etc.), but also deteriorates the quality of the code when changes are implemented [37]. It becomes harder to distribute tasks since it prevents teams from working independently as they must coordinate their development efforts and redeployments.

In order to adopt a newer platform framework or even just a newer version, it is possible that a rework of the entire application is needed [37] which is a risky and costly undertaking, meaning that a long-term commitment to a technology stack is required [35].

An example of how a Monolithic Architecture looks like can be observed in Figure 2.13.

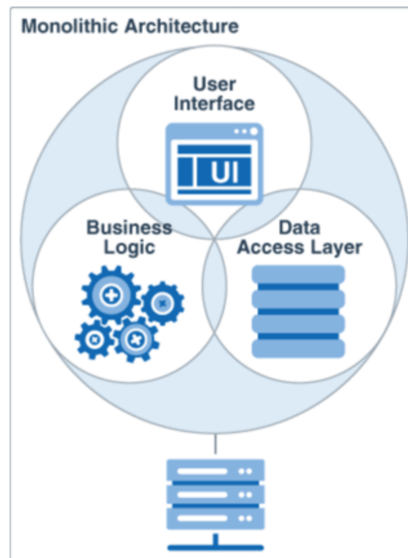


Figure 2.13: Monolithic architecture base example[7]

Migrating towards a microservices architecture may be the best option to solve existing issues and improve the future system maintainability [35].

[7] `https://docs.oracle.com/en/solutions/learn-architect-microservice`

2.8.2 Microservices Architecture

Microservices Architecture is one of the new popular solutions with a more modern approach [34]. It focuses on creating a distributed system by structuring the application, dividing components and logic into different modules.

The modules, running independently in its own process, then communicate with each other through lightweight mechanisms, such as Hypertext Transfer Protocol (HTTP), Application Programming Interface (API), Advanced Message Queuing Protocol (AMQP) (message brokers), or binary protocols like TCP, depending on the nature of each service [38].

A modularised system comes with improved maintainability, as each service is easier to understand and change. The development speed is improved as the code is easier to understand for new workers, is faster to deploy and each team can develop, test and deploy their services independently [38]. Also, it has improved fault isolation, for example, when a memory leak happens in a service then only that service is affected.

Technologically, microservices can rely on technology heterogeneity, since each service can use different technology in order to meet its own goals and performance [38]. This allows faster adoption of emerging technologies [37], such as frameworks or programming languages, eliminating long-term commitments to a specific technology stack.

Despite all of these advantages, this approach suffers from being a more complex implementation since it requires more planning, not only in the management of the developers but also in the division of the business logic into different modules [39].

Not only that but microservices are more difficult to test as a whole considering that to test a service it is first necessary to run all of the services associated with it. Therefore we can conclude that implementing inter-service communication and requests that span accross multiple services requires careful organization.

There is also more code duplication, since while in monolithic applications the codebase is shared, in microservices the different services may have duplicated code in order to achieve looser coupling [37].

Partial failure of only one module, which is an advantage in itself, also requires some extra work by implementing how the situation is supposed to be dealt with and how it affects the other modules that depend on it.

Microservices also require continuous and automated monitoring in order to automatically decide whether the deployment was successful and if the service is running properly [37].

An example of how a Microservices Architecture looks like can be observed in Figure 2.14.

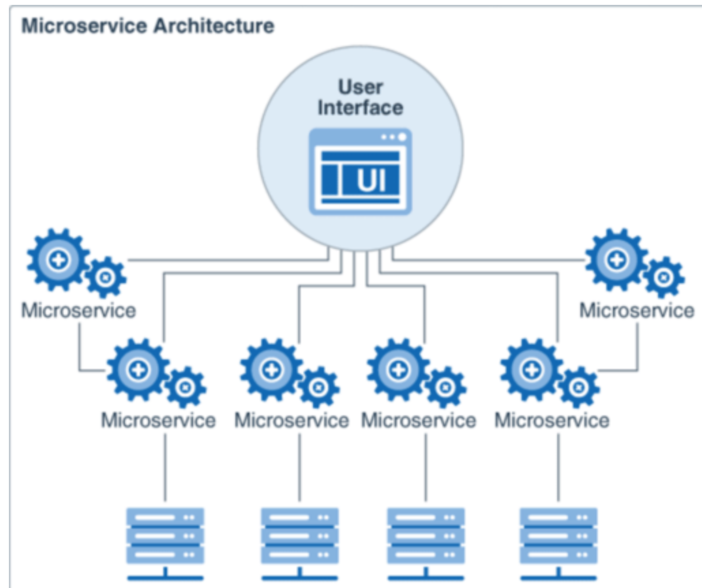


Figure 2.14: Microservices architecture base example[8]

2.9 Message-Oriented Middleware

Message-Oriented Middleware (MOM) is an approach for distributed systems acting as a message delivery mechanism. It acts as a middle layer for the whole distributed system when there is internal communication by simplifying data transfer between applications. Its gradual evolution has focused on improving system decoupling, asynchronous messaging and achieving high throughput [40].

In a Microservices Architecture, a MOM design can be implemented by utilizing a Message Broker as the middleware.

In order to understand how a Message Broker works, the following terminology is needed [41]:

- Producer – the application responsible for producing or publishing messages to the exchange;
- Consumer – the endpoint that consumes messages by subscribing to a queue;
- Queue/topic – used to store produced messages to be consumed;
- Exchange - the initial destination for all published messages and the entity in charge of applying routing rules for these messages to reach their destinations;
- Binding - a virtual connection between an exchange and a queue. A routing key can be associated with a binding for the exchange to know which messages should flow to a certain queue;

A basic structure example of a MOM implementation with the just explained terminology and concepts is shown in Figure 2.15 below.

[8] `https://docs.oracle.com/en/solutions/learn-architect-microservice`

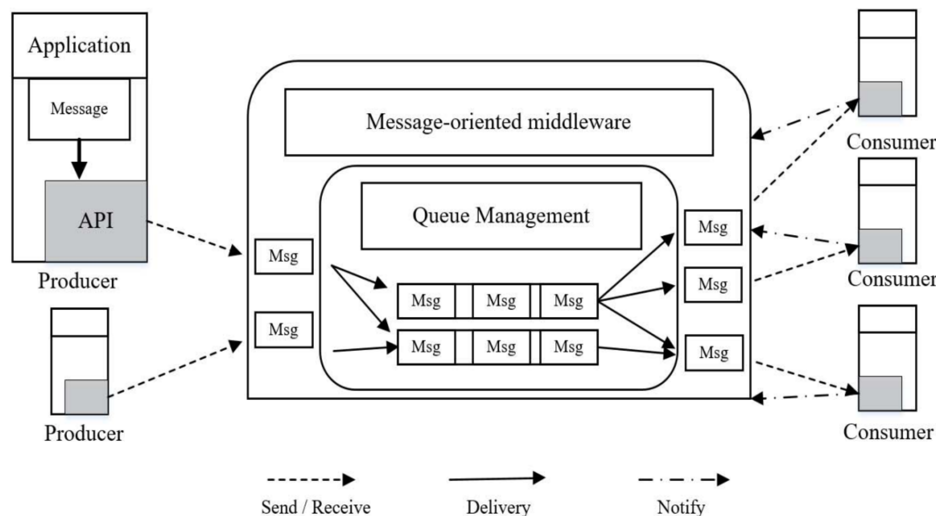


Figure 2.15: Basic structure of message-oriented middleware [40]

A message broker is a piece of software which enables services and applications to communicate with each other using messages. The message structure is formally defined and independent from the services that send them which allows applications to share information even if they are using a different programming language or technology stack.

The most popular and commonly used Message Brokers are Apache Kafka and RabbitMQ.

2.9.1 Apache Kafka

Apache Kafka[9,10] is an open-source streaming data platform originally developed by LinkedIn and after donated to Apache for future development [42].

Kafka provides a solution to deal with real-time volumes of information and route it to multiple consumers quickly. It was mainly designed with the following characteristics in mind [42]-[43]:

- Persistent messaging: information loss cannot be afforded, therefore Kafka is designed to serve as a storage system, storing data as along as necessary with disk structures that provide performance even with large volumes of stored messages.
- Scaling: Kafka operates as a modern distributed system that runs as a cluster and can scale to handle any number of applications.
- High throughput: Kafka is designed to support millions of messages per second.
- Multiple client support: Kafka supports easy integration of clients from different platforms such as Java, .NET, PHP, Ruby, and Python.
- Real time: messages produced should be immediately visible to consumers, and Kafka transmits data from producers to consumers with very low latency (5 milliseconds).

A large amount of data is generated by web-based applications which typically include user-activity like logins, page visits, clicks and other events. Kafka is suitable for data collection operations for services like these, that generate large amounts of data [40].

[9] `https://kafka.apache.org/`
[10] `https://docs.confluent.io/platform/current/clients/index.html`

An example of Apache Kafka is shown in Figure 2.16 below.

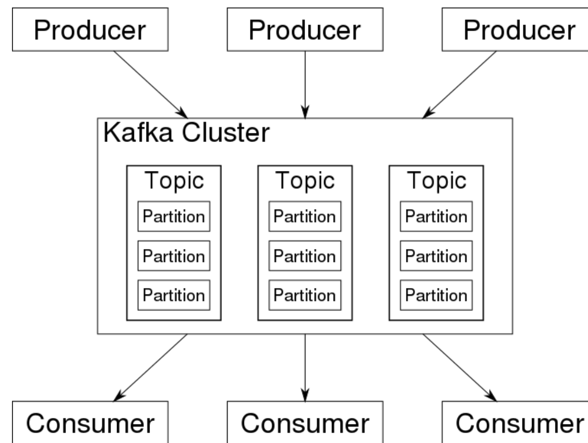


Figure 2.16: Apache Kafka example [42]

Kafka is less appropriate for data transformations, data storing or when only a simple task queue is needed. Its scope and scale is important for big data applications, but its overall complexity becomes unnecessary in some scenarios that do not take advantage of it.

As it is designed for high volumes of data, it becomes overkill when used to process only a small amount of messages. In situations where there is a smaller data set or only a dedicated task queue is needed, RabbitMQ becomes a better option [42].

2.9.2 RabbitMQ

RabbitMQ[11,12] is a lightweight and easy to deploy broker with several official client libraries for multiple different programming languages.

It is an open source messaging system developed in Erlang language, which improves the concurrency of the system, and is suitable for scenarios with high data consistency, stability and reliability requirements [40].

It is reliable to fault tolerance since the messages are only removed from queues after receiving the acknowledge from the consumer and the messages can also be saved on disk, allowing retainment and easy recover from a server failure.

RabbitMQ most important benefits and features are the following [44]:

- Platform and vendor neutral - since it implements the Advanced Message Queuing Protocol (AMQP) specification, there are clients available for almost any programming language and on all major computer platforms.
- Lightweight - requires less than 40 MB of RAM to run the core RabbitMQ application along with the Management UI (it increases as messages are added to queues).
- Multiple client support - has client libraries targeting most modern programming languages, acting as a centerpiece between applications written in languages such as Java, Ruby, Python, PHP, JavaScript, C, Golang and others.

[11] `https://www.rabbitmq.com/`
[12] `https://www.rabbitmq.com/devtools.html`

- Flexibility in controlling messaging trade-offs - allows the trade-off between reliable messaging and message throughput and performance by letting the user choose if messages are persisted to disk prior to delivery.
- Third-party plugins - provides a vast and flexible plugin system, for example, plugins to allow storing messages directly into databases by using database writes.

An example of RabbitMQ is shown in Figure 2.17 below.

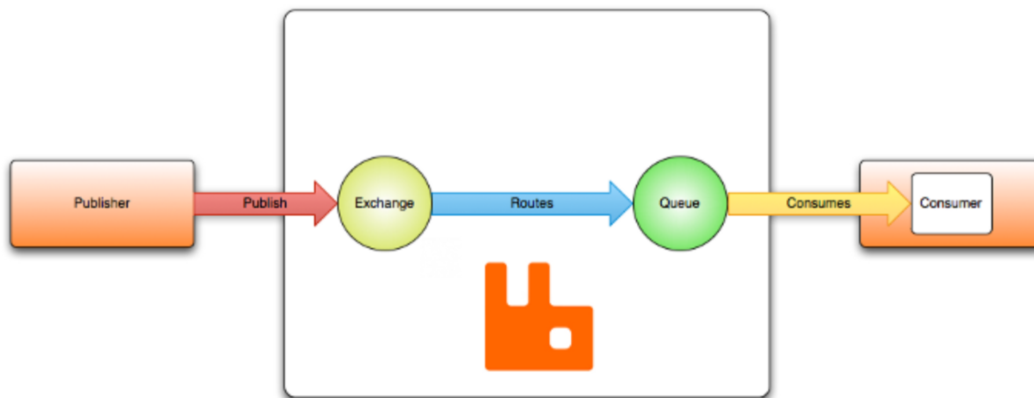


Figure 2.17: RabbitMQ example[13]

RabbitMQ also facilitates the delivery of messages in complex routing scenarios. The Exchange takes a message and routes it into zero or more queues. The routing algorithm used depends on the exchange type and rules called bindings. RabbitMQ defines four exchange types:

- Direct Exchange: A queue binds to an exchange with a routing key. If a message with the same routing key arrives to the exchange, it routes it to the queue;
- Default Exchange: A queue binds to an exchange with a routing key equal to its own name. Therefore it is a Direct Exchange but only one queue receives the message since different queues cannot have the same name;
- Fan-out Exchange: The exchange routes messages to all of the queues that are bound to it and the routing key is ignored;
- Topic Exchange: A queue binds to an exchange with a routing key. The exchange routes messages to one or many queues based on the matching between the message's routing key and the pattern that was used to bind the queue to the exchange;

An illustration on the difference of these exchange types can be observed in Figure 2.18.

[13] `https://www.rabbitmq.com/tutorials/amqp-concepts.html`

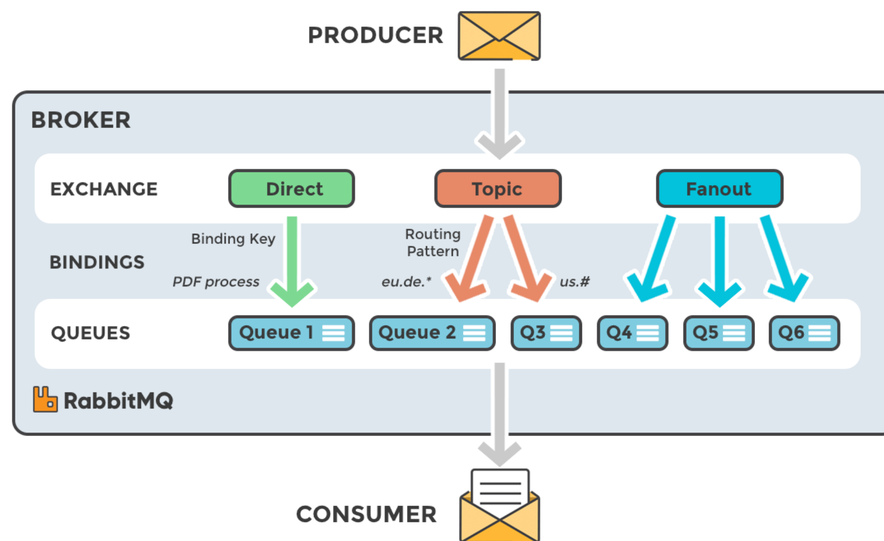


Figure 2.18: RabbitMQ exchange types[14]

2.10 Summary

As a reminder, the objective of this project was to develop a Charging Station Management System (CSMS) and a Charging Station.

After evaluating the differences between a Monolithic Architecture and a Microservices Architecture in Section 2.8, the latter shows more advantages with better scalability and most importantly, improved maintainability, being easier to update.

This comes as an even bigger advantage since as discussed in Section 1.1.2, the EV landscape is still evolving with standards yet to be chosen which might force implementations to switch protocols. With a Microservices Architecture most modules can be reutilized, allowing an easier transition to the adopted standards.

In the discussion of Message-Oriented Middleware in Section 2.9, RabbitMQ shows as a better option for this project. The main reason comes from it being lightweight, facilitating routing scenarios and being a simpler option for systems with low message amounts.

In Section 2.1 the Charging Infrastructure was described in order to understand the concepts of CSMS and Charging Station. To implement the communication between them, the Open Charge Point Protocol was brought up in Section 2.6 where the newer OCPP 2.0.1 version was presented with more and improved features. Due to it not being backwards compatible with few to none real-life implementations up to this point, it was chosen in this project as it is future-proof, preventing a sizeable rework of the project's protocol in the future.

Section 2.3 stated the Type 2 connector adoption as a standard by the European Commission. When developing the Charging Station it should have multiple connectors so it can be ready for both AC (Mode3) and DC (CCS Combo 2) charging.

[14] `https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html`

To develop a software driver that communicates with the connector hardware, it must implement the Low-Level Communication which uses the PWM signal as described in the standard IEC 61581 in Section 2.4.

In order to provide the choice to charge in DC CCS, the logic must implement the Power Line Communication as described in the standard ISO 15118 in Section 2.5.

CHAPTER 3

Implementation

3.1 Architecture Overview

The chosen architecture is composed of two distinct systems: the Central System and the Charging Station.

The Central System is an implementation of a Charging Station Management System that connects to multiple Charge Points via OCPP, controlling and managing them. It is composed by the following modules:

- CSMS: Implements all the defined OCPP functions with the expected payload types. It redirects requests/responses from a Charging Station to the Controller module via RabbitMQ, and redirects requests/responses from the Controller to the Charging Station via OCPP.
- Controller: It is the holder of all OCPP logic and the orchestrator of the Central System. The idea of a Controller module was discussed previously on the OCPP 2.0.1 Section 2.6.1: "Added the idea of a local controller inside the CSMS, which acts as a brain by controlling and making automated smart decisions".
- HTTP Server: Handles external requests. It can be used by Charging Station Operators in order to obtain needed information and even act upon it. Another use case is the possibility to create a mobile application for clients that communicates via a HTTP channel which allows users to make reservations of a specific EVSE.

The other system part of the architecture is the Charging Station itself, managed by a Central System. It can contain multiple EVSEs each with multiple Connectors. A Charging Station is composed by the modules below:

- Charge Point: The CSMS module equivalent in the Charging Station. Contains all the OCPP functions but redirects the requests/responses to the Decision Point.
- Decision Point: The Controller module equivalent in the Charging Station. Holds all the OCPP logic and acts as the brain of the Charging Station.
- Energy Driver

- Mode3 Driver (part of an EVSE)
- Human-Machine Interface (part of an EVSE)
- Parking Sensor (part of an EVSE)
- RFID CardReader (part of an EVSE)

The following Figure 3.1 presents the implemented system architecture where the inter-systems and inter-module communications are both shown. The Figure contains an example of a Charging Station with a single EVSE with two Connectors.

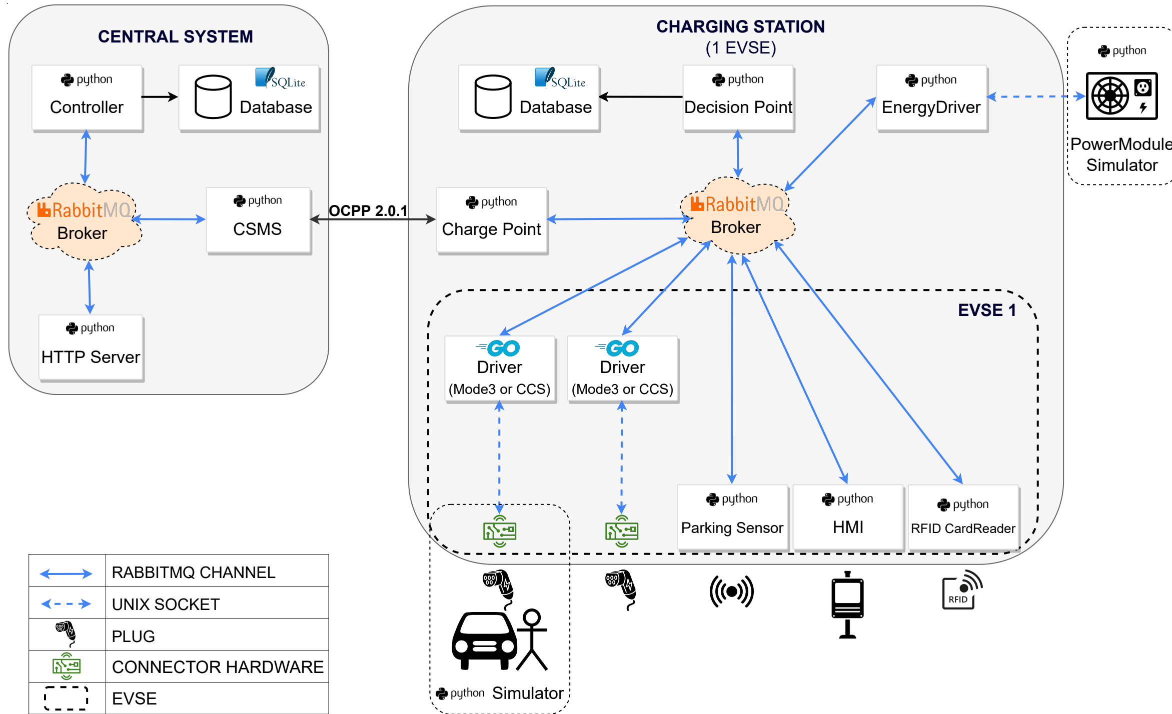


Figure 3.1: Architecture Overview

The possibility of multiple EVSEs was made for cases where there are multiple chargers in a parking lot. An EVSE allows for the cable to be in a closer proximity to the vehicle, reducing possible energy losses. It also provides a better experience to the EV Driver by having a RFID CardReader physically closer to the user.

Since a single EVSE does not generate a very high throughput of messages, a single Decision Point can handle multiple EVSEs, which reduces hardware costs by only needing a single computer.

Therefore, we only require one Charge Point for the whole parking lot, meaning we only need one public IP to communicate with the Central System to provide all the information of the parking lot. This reduces not only communication costs, but also facilitates the management by the CSMS as it reduces the number of connections needed when compared to what it would need if it had a connection per EVSE.

This project is a continuation of another thesis from the previous year called “Electric Vehicle Charging Control System via OCPP 2.0” done by Alexandre Machado [7].

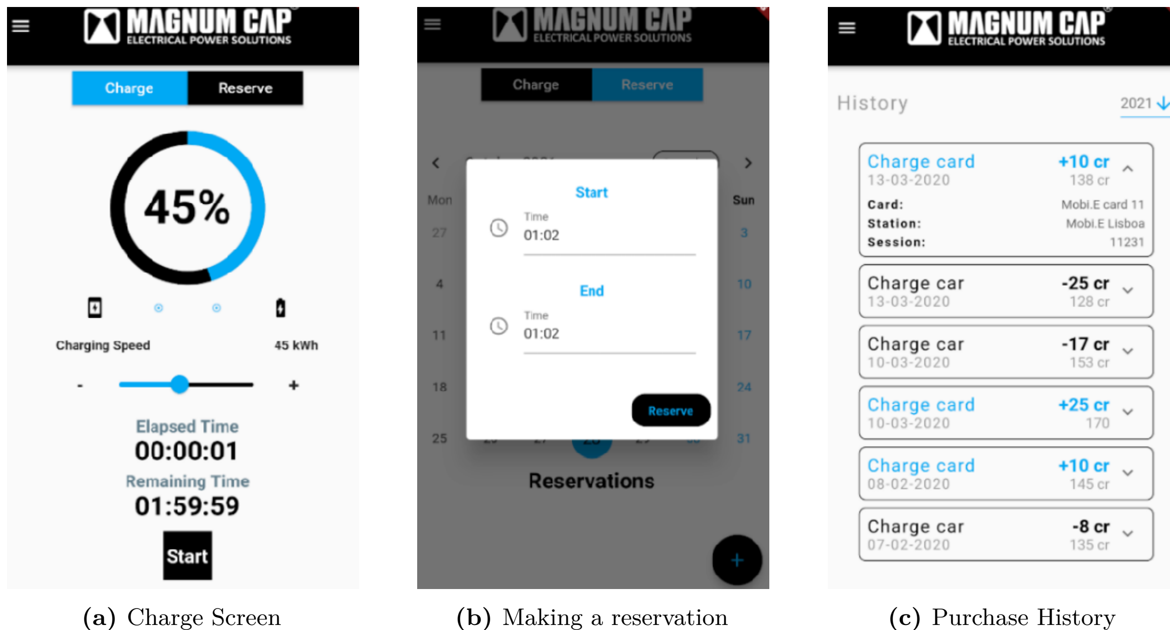
The previous development focused on the “backend” side of the project by creating the Central System and starting the Charging Station, mainly the Charge Point module in order to fully implement the chosen OCPP communication.

A HTTP Server was also created to allow managing by Charging Station Operators and to take advantage of OCPP 2.0.1 capabilities that allows reservations by clients or remote start charging through a mobile application.

The mobile application for clients which would interact with the HTTP Server was not successfully implemented, however a mock-up layout for a future application was done and is shown in Figure 3.2 and Figure 3.3.

(a) Login Screen **(b)** Map Screen **(c)** Charging Station information

Figure 3.2: Mobile application mock-up: Part 1 [7]



(a) Charge Screen **(b)** Making a reservation **(c)** Purchase History

Figure 3.3: Mobile application mock-up: Part 2 [7]

The Central System modules (Controller, Database, CSMS and HTTP Server) can be considered as complete at the starting point of the project.

Regarding the Central System persistence, two databases were created that the Controller moduler interacts with in order to store important and relevant information about its internal state [7]:

- Database.db: contains all high-level information and known data about real stations, connectors, cards, registered users and charging reservations.
- OCPP.db: responsible for storing data of the OCPP API such as the current authorization list, custom tariffs, OCPP's data types which define charging limits and profiles, and incoming messages from each Charging Station which include periodic meter values from connectors, start transaction requests, among other things.

The CSMS module is responsible for communicating directly with the Charging Stations via OCPP 2.0. It is the direct receptor of all the connected Charging Stations' requests, subsequently redirecting them to the appropriate request processing unit, the Controller, and finally sending the generated responses to the respective Charging Station [7].

The Controller is accountable for updating the different databases, for keeping track of the several Charging Stations and for executing the logic required to fulfill each OCPP request methodology [7].

The Controller is also a communication mediator for the HTTP Server, redirecting all the requests that arrive from it to the CSMS module while at the same time knowing that they happened. For example, if a specific Charging Station was reserved by a client, the Controller will save the information in its own database but also order the CSMS to inform the Decision Point of the reserved Charging Station that such reservation occurred.

On the Charging Station, the Charge Point module was complete as it was a mirror of the CSMS module, implementing the expected OCPP functions and redirecting the requests or

responses to the module responsible for the logic, which in the case of the Charging Station is the Decision Point.

The Decision Point module had already some OCPP logic implemented, with the Database interactions also present as they are similar to the Controller's Database.

A configuration file with global variables needed for a correct OCPP implementation was also present, which is needed so the Charging Station knows what mandatory requirements it should impose in order to allow a charging session to start (EV plugged-in, RFID Card, Parking Sensor detection, Remote Start Charging).

Most other modules of the Charging Station were implemented as dummy modules with the only objective of demonstrating some of the Central System functionalities. These modules were then disposed of at the start of development, or heavily rewritten.

In terms of changes to Central System modules, only a small change to the Controller on how it deals with handling a charging transaction cost was made during implementation.

The RabbitMQ implementation was using the Default Exchange type which only allowed to send a message directly to one queue. This was causing modules to publish the same message twice when they wanted the same message to reach two different modules. Also, each module had a queue for requests and a queue for responses, which was deemed unnecessary as a single one suffices.

One of the first steps in the implementation was exactly remaking the RabbitMQ implementation by switching to a Topic Exchange type, as is explained in Section 3.3 below.

3.3 Message Broker

As previously stated, the chosen message broker to implement the inter-module communication was RabbitMQ using the Topic Exchange type.

As a reminder of Section 2.9.2, the Topic Exchange type routes messages to one or many queues based on the matching between a message's routing key and the pattern that was used to bind a queue to an exchange.

This means that within a certain module, its queue's bindings can be chosen in a way that only needed messages arrive to be consumed. A simple logic was chosen, with most messages being part of one of three routing keys:

- Status - Messages by modules that periodically inform the system of their current status. Useful for the Decision Point to consume in order to monitor the Charging Station, and for the Human-Machine Interface (HMI) to show some information to the client currently charging the vehicle.
- Events - Messages by modules with asynchronous events, such as when the charging protocol Driver detects that an EV cable has been plugged-in or when a client presses a button of the HMI.
- Actions - Messages that force modules to take an action, usually sent by the Decision Point. For example, when a request arrives from the Central System to reduce energy, the Decision Point orders the charging protocol Driver to reduce its current output.

The only exception is the "ocpp" routing key, which the Charge Point uses to communicate with only the Decision Point.

However, if in the future there is a need for a module to also consume these messages, it is only needed to bind the "ocpp" routing key to the queue that this module consumes from. This is one of the main advantages of this implementation, allowing to introduce new modules and select the exact messages they need to receive, without requiring any changes to already existing modules.

The bindings of the queues from Charging Station modules are the following:

- ChargePoint bindings: [ocpp.chargepoint]
- DecisionPoint bindings: [ocpp.decisionpoint, status.#, events.#]
- HMI bindings: [status.#, events.#]
- ProtocolDriver bindings: [actions.#]
- EnergyDriver bindings: [actions.#]

The other Charging Station modules only publish messages, therefore they do not have any bindings since they do not consume any messages.

Simply put, to publish a message to a topic exchange a routing key needs to be provided and all queues that have bound that routing key, or a short version of it, will receive the message. An example of how a message published with the routing key "status" is handled by the Topic Exchange in the Charging Station is shown in Figure 3.4.

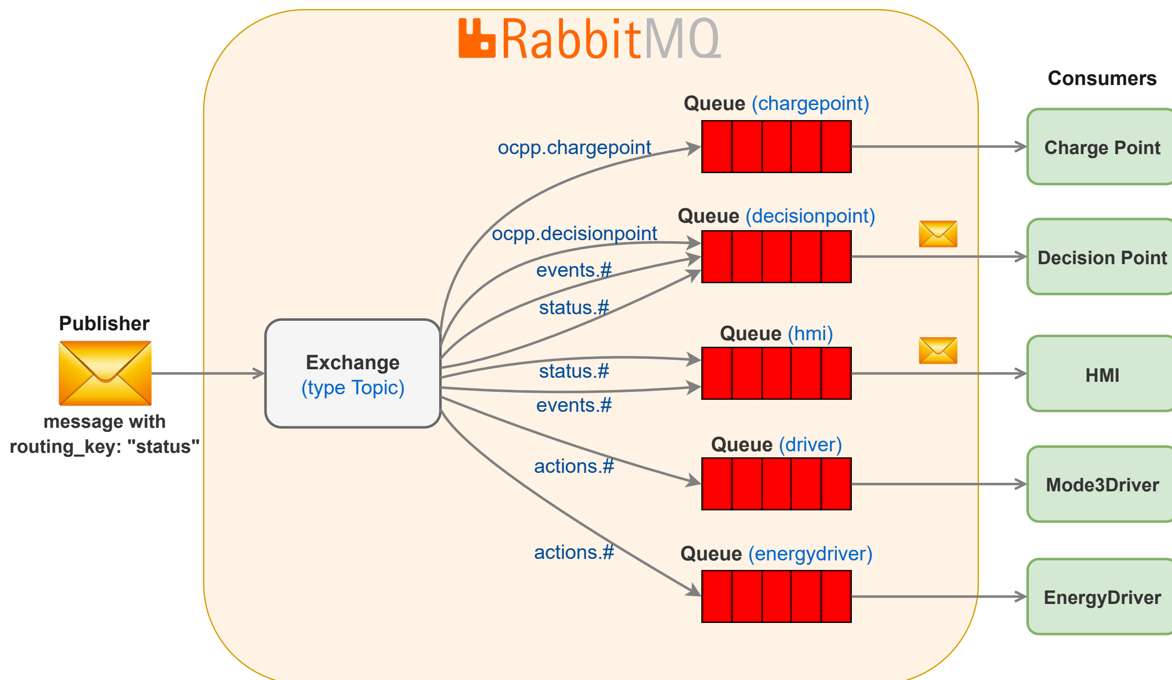


Figure 3.4: Topic Exchange handling a "status" message in the Charging Station

After understanding how the inter-module communication is structured, a more in depth explanation of each Charging Station module is done in the next sections.

3.4 Charge Point

The Charge Point module acts as a bridge between the Central System's communication via OCPP and the Decision Point. The location of the Charge Point in the system architecture is shown in Figure 3.5 below.

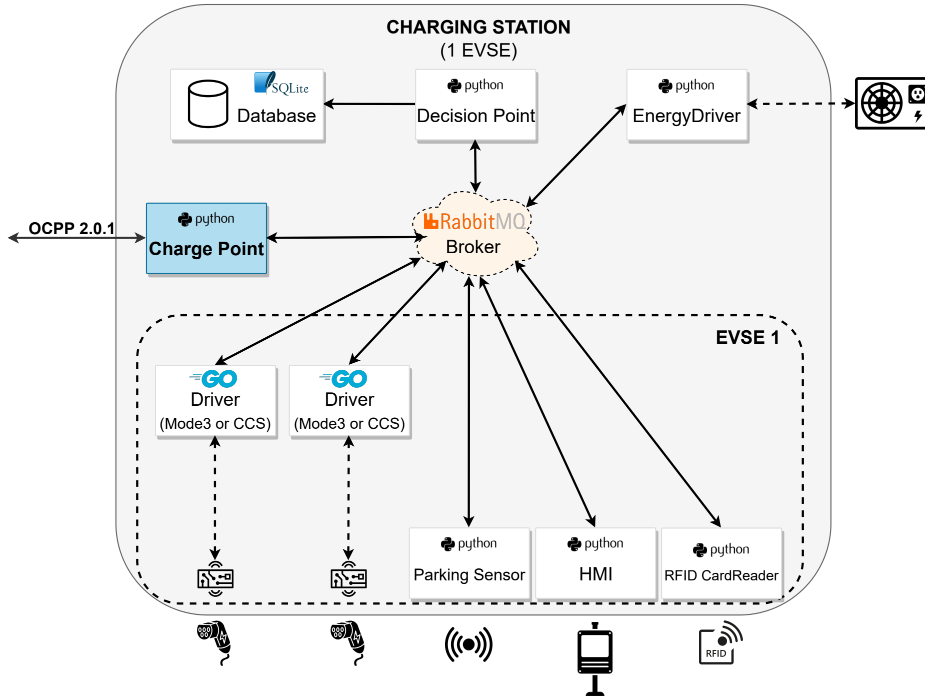


Figure 3.5: Charge Point module highlighted in the Architecture

In the implementation, the responsability of implementing the OCPP logic is given to the Decision Point. The Charge Point handles the OCPP connection itself and constructs the json payloads according to the OCPP specification in order to send the responses back to the CSMS.

An example of how a request is handled by the Charge Point is shown in Figure 3.6, where the resemblance from the CSMS module and the Charge Point can be observed as they only act as a entrypoint to both systems.

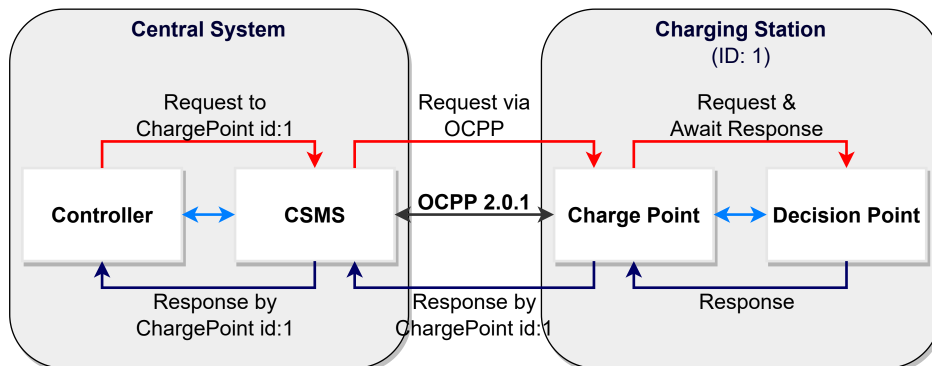


Figure 3.6: CSMS to Charging Station Request

Being able to separate both responsibilities is an advantage of the modularity that the Microservices Architecture provides, making the development of the Decision Point easier, since when responding to the Charge Point it does not need to obey to the OCPP specification and instead can use the internal payload type used for all RabbitMQ communication within the Charging Station.

By implementing this new version of OCPP, the Charge Point module needs to be ready to receive the 39 possible incoming requests from the Central System. These 39 incoming messages are the following:

1. CancelReservation
2. CertificateSigned
3. ChangeAvailability
4. ClearCache
5. ClearChargingProfile
6. ClearDisplayMessage
7. ClearVariableMonitoring
8. CostUpdated
9. CustomerInformation
10. DataTransfer
11. DeleteCertificate
12. GetBaseReport
13. GetChargingProfiles
14. GetCompositeSchedule
15. GetInstalledCertificateIds
16. GetLocalListVersion
17. GetLog
18. GetMonitoringReport
19. GetReport
20. GetTransactionStatus
21. GetVariables
22. InstallCertificate
23. PublishFirmware
24. RequestStartTransaction
25. RequestStopTransaction
26. ReserveNow
27. Reset
28. SendLocalList
29. SetChargingProfile
30. SetDisplayMessage
31. SetMonitoringBase
32. SetMonitoringLevel
33. SetNetworkProfile
34. SetVariableMonitoring
35. SetVariables
36. TriggerMessage
37. UnlockConnector
38. UnpublishFirmware
39. UpdateFirmware

Besides being able to send the response to each of the previous messages, the Charge Point also needs to be able to send the 25 possible request messages to the Central System, as defined in OCPP. These 25 possible outgoing request messages are the following:

1. Authorize
2. BootNotification
3. ClearedChargingLimit
4. DataTransfer
5. FirmwareStatusNotification
6. Get15118EVCertificate
7. GetCertificateStatus
8. Heartbeat
9. LogStatusNotification
10. MeterValues
11. NotifyChargingLimit
12. NotifyCustomerInformation
13. NotifyDisplayMessages
14. NotifyEVChargingNeeds
15. NotifyEVChargingSchedule
16. NotifyEvent
17. NotifyMonitoringReport
18. NotifyReport
19. PublishFirmwareStatusNotification
20. ReportChargingProfiles
21. ReservationStatusUpdate
22. SecurityEventNotification
23. SignCertificate
24. StatusNotification
25. TransactionEvent

3.5 Decision Point

The Decision Point module acts as a brain of the Charging Station and manages all the EVSEs that are part of it.

It analyzes and stores all information it receives from the multiple EVSEs, making decisions and assigning actions when it deems necessary. Most of the actions go through it for confirmation or to make the Decision Point aware of which actions are currently happening.

It comes as no surprise that the Decision Point module is the most extensive and complex module of the Charging Station. The location of the Decision Point in the system architecture is shown in Figure 3.7 below.

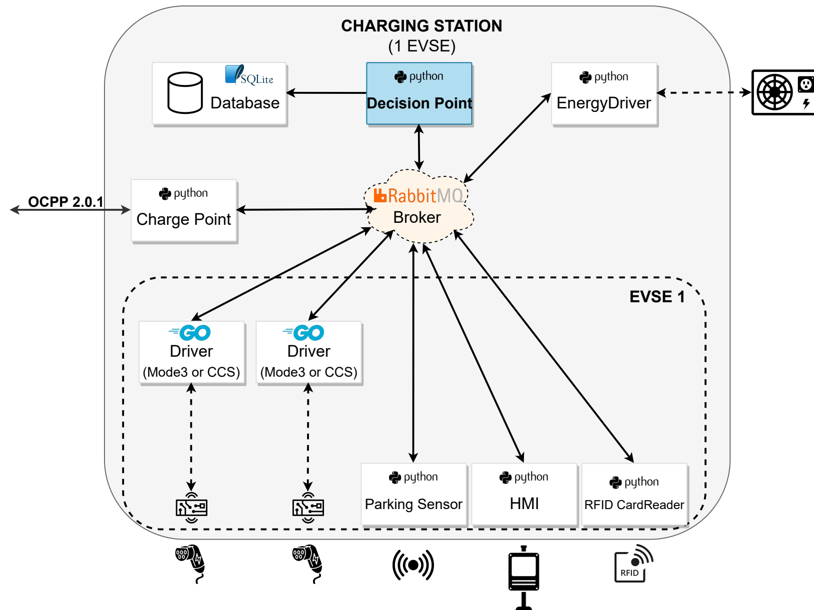


Figure 3.7: Decision Point module highlighted in the Architecture

The internal architecture is designed around threads each with their own role within the module. This design can be seen in Figure 3.8.

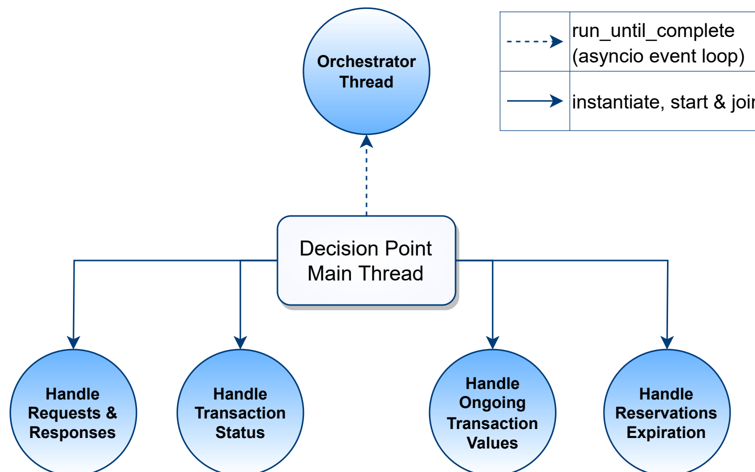


Figure 3.8: Decision Point Threads

3.5.1 Orchestrator

The "Orchestrator" thread starts off by executing a cold boot method in accordance with the OCPP 2.0 specification. A "BootNotification" is sent to the CSMS via the Charge Point module.

The response, if "Accepted", should also contain an integer which defines the value in seconds a periodic "HeartBeat" request should occur, which helps confirm normal operation. The response should also contain a date and time because if the Charging Station has no in-built real-time clock hardware that is correctly preset, it may not have a valid time setting, making it impossible to later determine the date and time of transactions.

After the positive response, the Decision Point should send a "StatusNotification" request to the CSMS for all the Connectors of each EVSEs it controls. The described cold boot of a charging station is shown in Figure 3.9.

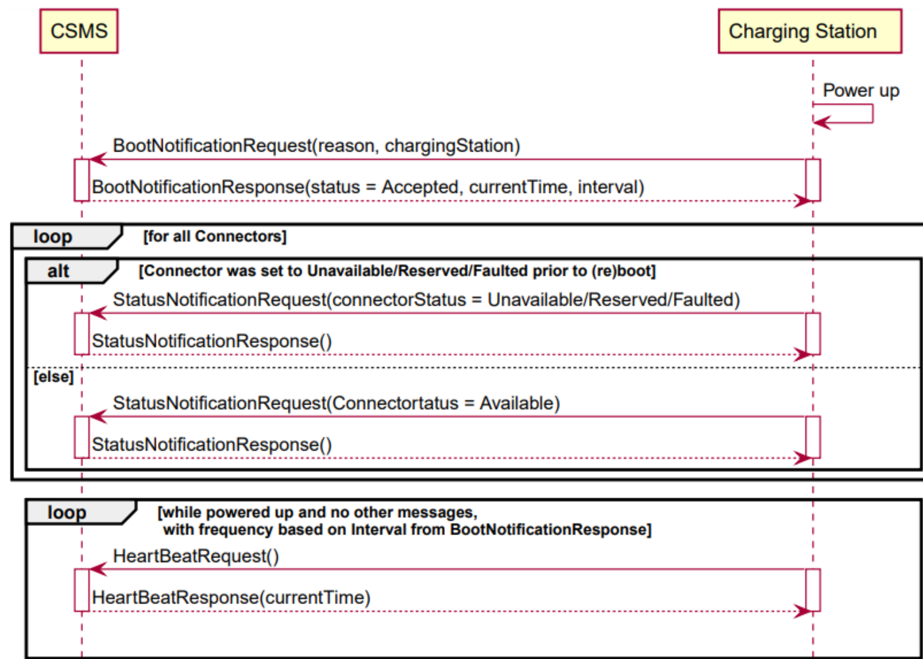


Figure 3.9: Sequence Diagram:: Cold Boot Charging Station (from OCPP 2.0: Part 2 - Specification [45])

After the cold boot, the Orchestrator thread permanently loops checking the value of a long list of control variables. In the case that a control variable is enabled, the Orchestrator calls the respective method with the implemented logic required for that request. These control variables value is controlled by the "Handle Requests & Responses" thread, explained below.

3.5.2 Handling Requests & Responses

The "Handle Requests & Responses" thread consumes messages from the 'decisionpoint' queue. As shown in the message broker section, this means that it receives the requests and responses from the CSMS via the Charge Point module, along with the "Status" and "Event" messages from other Charging Station modules.

Some CSMS requests only require an “Accepted” or an information from the DecisionPoint in their response, which can be instantly processed and responded to. However, there are requests that require more steps, and after the “Accepted” message, they trigger certain control variables which activates the Orchestrator thread to call certain methods.

An example of this is the “ReserveNow” request from the CSMS that expects an “Accepted” message if the reservation was possible but also expects the Decision Point to send a “StatusNotification” for each Connector in the reserved EVSE, informing that such Connector is now reserved. This communication is shown in Figure 3.10.

Because of this, after sending the positive “Accepted” response to the CSMS, a “ReserveNow” control variable is activated, making the Orchestrator invoke a method that sends a “StatusNotification” per Connector. Between every “StatusNotification” request, the Orchestrator awaits the response, which arrival is known when the “Handle Requests & Responses” thread consumes the response and activates a flag variable.

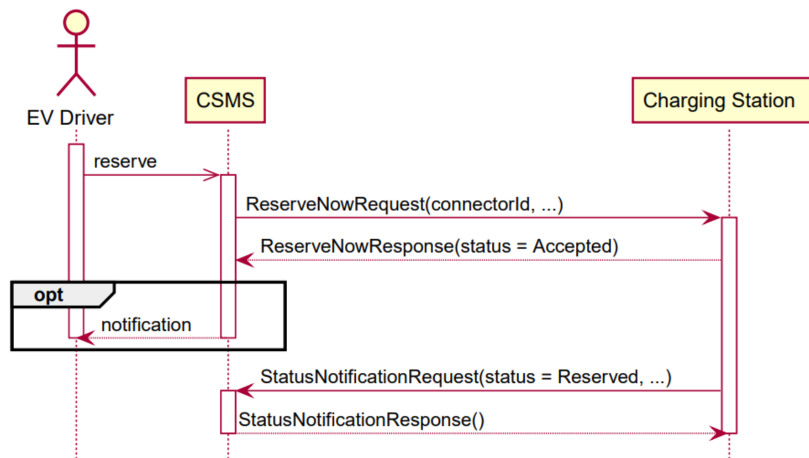


Figure 3.10: Sequence Diagram: Reserve a specified Connector at a Charging Station (from OCPP 2.0: Part 2 - Specification [45])

Messages from the other Charging Station modules always trigger control variables and the execution is also handled by the Orchestrator.

3.5.3 Transactions

In the configuration file, the ‘TxStartPoint’ variable defines what requirements need to be fulfilled in the EVSE in order for a transaction to be started. The variable can contain one or a combination of these values: “ParkingBayOccupancy”, “EVConnected”, “Authorized” and “PowerPathClosed”.

Handling Transaction Status

The “Handle Transaction Status” thread changes a transaction’s status between “Started”, “Updated” and “Ended”. It does so by looping through all EVSEs’ information and checking if the requirements defined in the ‘TxStartPoint’ are fulfilled in order to create a transaction with a “Started” state. It also checks on-going transactions to see if any requirement stopped being fulfilled, ending the transaction if that occurs.

The “ParkingBayOccupancy” value is guaranteed by a ParkingSensor module in the EVSE that detects the presence of a parked vehicle. There are some use cases where only needing this requirement to start a transaction request would be useful, since this way the CSMS knows that a charging is gonna happen and therefore has some time to allocate resources or prepare beforehand.

For Charging Stations in public spaces, it is also common to require the “Authorized” value to be fulfilled in order to start a transaction. Authorization is done by the CSMS, and the implemented logic allows for three different types of authorization, all of them defined in OCPP 2.0 [45]:

- C01 - EV Driver Authorization using RFID: Mode which requires the EV Driver to present his RFID card. Consequently, the DecisionPoint sends a “AuthorizeRequest” to the CSMS with the information read from the RFID card. After the authorization step, if accepted by the CSMS, the “Authorized” value of the respective EVSE becomes positive.
- C04 - Authorization using PIN-code: Mode which requires the EV Driver to enter a PIN-code. Consequently, the DecisionPoint sends a “AuthorizeRequest” with the inserted PIN-code to the CSMS. After the authorization step, if accepted by the CSMS, the “Authorized” value of the respective EVSE becomes positive.
- C05 - Authorization for CSMS initiated transactions: Mode in which the EV Driver uses an app to control the charging remotely. The app sends a request to the CSMS which then sends a “RequestStartTransaction” to the Charging Station. The “Authorized” value of the respective EVSE becomes positive, and the Decision Point accepts starting the transaction if all other requirements are fulfilled.

“C02 - Authorization using a start button” is a mode which does not require authorization. It is when the ‘TxStartPoint’ variable requires “EVConnected” but not does not require “Authorized”. In this mode the DecisionPoint sends a “StatusNotification” informing that a specific Connector is occupied and a “TransactionEvent” with “Started” when the cable is plugged in. Therefore the transaction is at a “Started” state right after the cable is plugged in.

The EV Driver can then start the charging by the press of a physical start button in the Human-Machine Interface (HMI). This specific mode is shown in Figure 3.11.

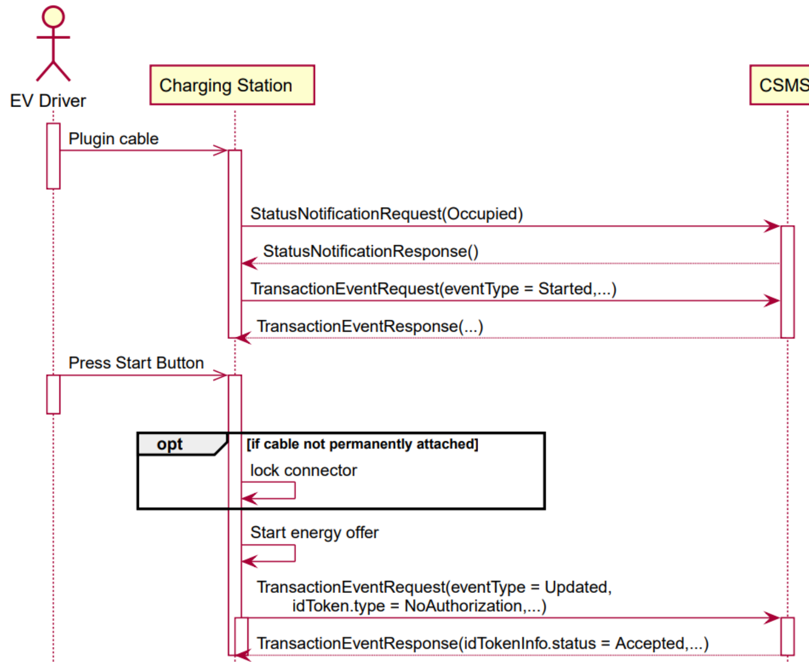


Figure 3.11: Sequence Diagram: Authorization using a start button (from OCPP 2.0: Part 2 - Specification [45])

A transaction switches from "Started" to "Updated" when charging begins. This can happen when the EV Driver physically presses the start button which triggers an event by the HMI module, or when the EV Driver remotely presses a start button on an app which triggers an incoming message from the CSMS.

The transaction changes to "Ended" if the transaction is stopped through a stop button, remotely or via the one in the HMI, or if the vehicle becomes fully charged, which triggers an event by the Mode3Driver.

Handling Ongoing Transaction Values

Since each Connector's status periodically arrives to the Decision Point during charging, it saves that information regarding power relay and energy values to a specific database table for Connector Meter Values. The "Handle Ongoing Transaction Values" thread periodically updates the ongoing transactions with the information of the respective Connector that is handling that transaction.

This allows to implement the following specification in OCPP 2.0 [45]: "Frequent (e.g. 1-5 minute interval) meter readings taken and transmitted (usually in "real time") to the CSMS, to allow it to provide information updates to the EV user (who is usually not at the Charging Station), via web, app, SMS, etc., as to the progress of the transaction".

Saving and logging the values over the course of the transaction also allows to have an idea on how the Connector values varied over time during the transaction. During a charging process of an EV there are energy variations, priority charging fees depending on an user's contract and even V2G moments in which the vehicle is temporarily giving energy to the grid. This is the reason why the HMI of Charging Stations cannot show the price of charging cost

increasing in real-time, because all of this information needs to go to the Operator, and only then the final price is calculated.

3.5.4 Reservations

EV Drivers can do reservations via an app. This interaction and how it is handled by the Charging Station was shown in Figure 3.10 of a previous section. It is the Charging Station duty, and therefore, the Decision Point's duty to handle the reservations, saving them in a database and then confirming if a user identification matches with the one that reserved the EVSE.

The "Handle Expiration of Reservations" thread periodically checks the end date of all reservations, removing the expired ones. Since upon reservation all connectors of the reserved EVSE become "Reserved", as shown in Figure 3.10, after cancelling or finishing a reservation, a "StatusNotification" for each connector is sent to the CSMS informing that they are now available. The consequences of missing a reservation is not responsibility of the Charging Station, and should be handled by the operator or the owner of the application used by the EV Driver.

3.5.5 Local Database

The Decision Point module interacts with a local database in order to store the Charging Station information. Having local stored information allows to not be fully dependent on the Central System in case of a loss of connection.

The database file is responsible for storing high-level information in different tables. The most important used tables are shown in Table 3.1.

Table Name	Description	Important Columns
Reservations	Stores information about charging reservations	reservationID, reservationCardIdTag, reservationStartDate, reservationEndDate, evseID
EVSE	Contains data about all the EVSEs	evseID, model, UUID, address, postcode, town, country, latitude & longitude, operator, isPayAtLocation, isMembershipRequired, isAccessKeyRequired, status
Connectors	Contains data about all the connectors	connectorID, evseID, protocol, status, maxCurrentAllowed
ConnectorMeter	Stores the periodic messages with the meter values coming from the connectors	connectorID, evseID, transactionID, reservationID, currentImport, stateOfCharge, temperature, isEvConnected
SalesTariff	During Authentication the EV Driver provides a token and the CSMS returns the tariff applicable. This table contains the information of all types of Tariffs.	id, salesTariffDescription, startValue, ePriceLevel, amountMultiplier

Table 3.1: Description of Database.db tables (Part 1)

Table Name	Description	Important Columns
Transactions	Contains information about known transactions	transactionID, state, timestamp, triggerReason, seqNo, chargingState, evseID, connectorID, idToken
BootNotifications	Contains responses to the BootNotificationRequests sent to the Central System	timestamp, status

Table 3.1: Description of Database.db tables (Part 2)

3.6 Mode3 Protocol Driver

The protocol driver implements the logic and communication in order to interact with the connector hardware while following a defined charging protocol, in this case, Mode 3. The location of the protocol driver in the system architecture is shown in Figure 3.12.

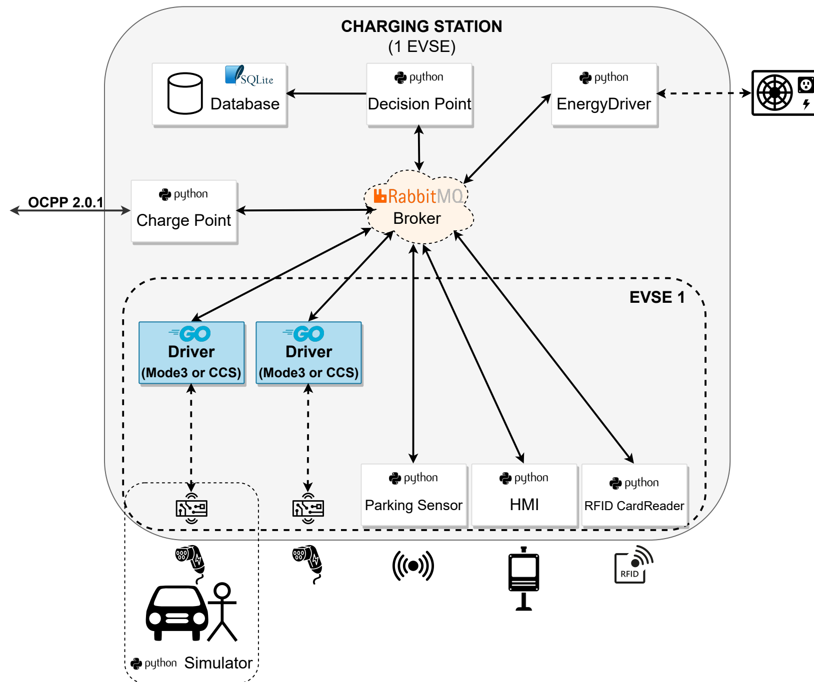


Figure 3.12: Mode3 Protocol Driver module highlighted in the Architecture

The programming language chosen to implement the protocol driver was Golang, mainly because it is lightweight and performant, allowing to meet the time requirements expected by the electric vehicles, but also because it excels in simplifying concurrency for the programmer.

Since there is a predefined order of actions that need to be executed before, during and after a charging process, a finite state machine was used to organize and develop the driver's logic. The states and their transitions were based on a provided hardware documentation (Appendix A), with the goal of representing the phases present in electric vehicles charging. The developed finite state machine with the states and transitions is shown in Figure 3.13.

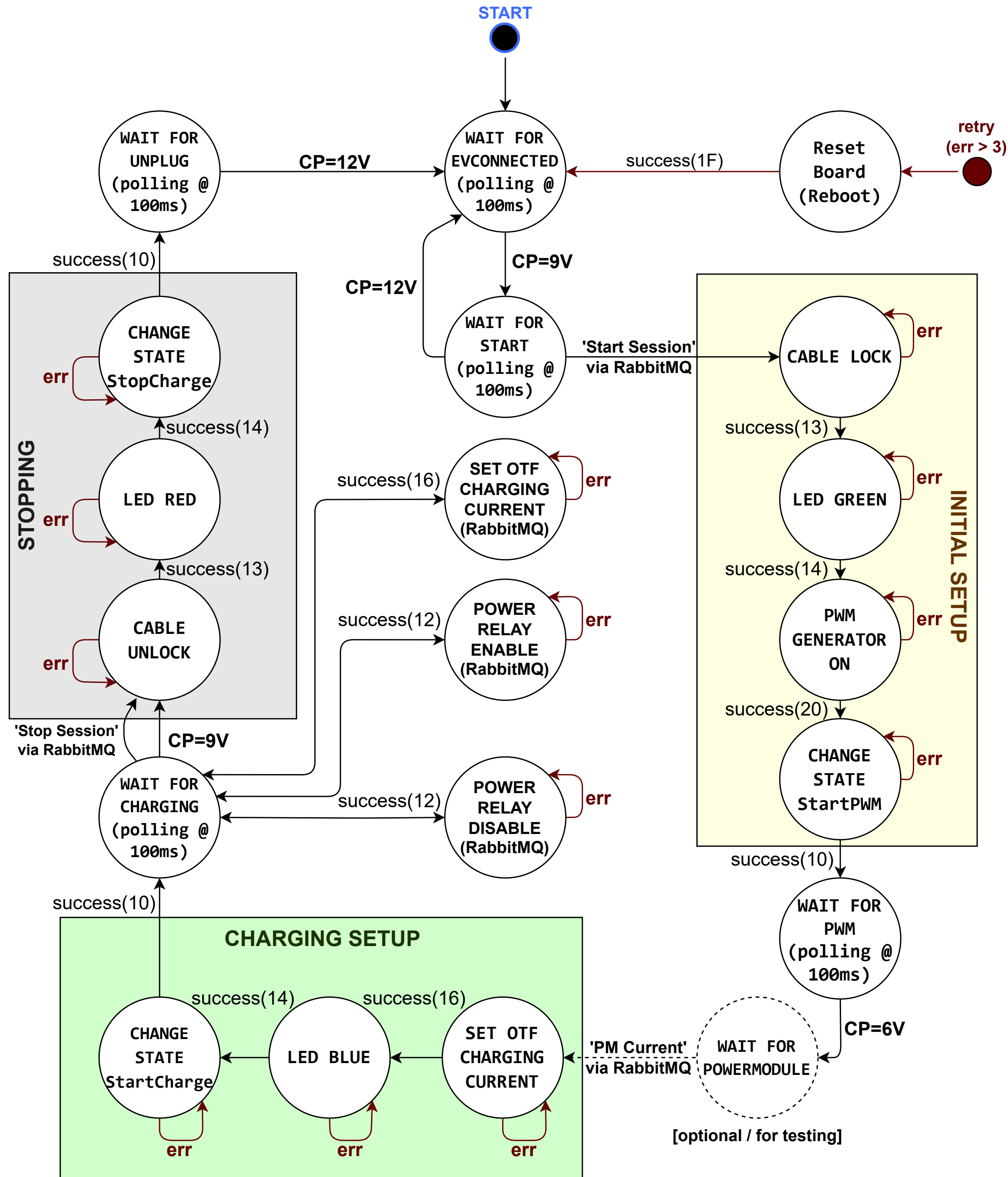


Figure 3.13: Mode3 Driver Finite State Machine: States and Transitions

Four important waiting states can be observed, where a polling is done every 100ms via a 'status request' command. The control pilot (CP) pin value in volts present in the response decides if a transition to the next state occurs, as shown in Table 3.2.

State	CP value to keep polling	CP value to change state
Waiting for EV to plug	12V $\pm$ 0.1	9V $\pm$ 0.1
Waiting for PWM to finish	9V $\pm$ 0.1	6V $\pm$ 0.1
Waiting for Charging to end	6V $\pm$ 0.1	9V $\pm$ 0.1
Waiting for EV to unplug	9V $\pm$ 0.1	12V $\pm$ 0.1

Table 3.2: Waiting states' transition conditions

This utilization of the control pilot (CP) value was previously explained in Section 2.4, where the Low-Level Communication specified in the IEC 61581 standard was described, with a similar logic being shown in Table 2.3.

Besides the control pilot (CP) value, the 'status request' response from the hardware provides various other information such as:

- State of the Cable lock mechanism
- State of Power Relay (Enabled or disabled)
- Resistance of proximity pilot (PP) pin
- The decided current between the connector hardware and the EV (after PWM phase)
- The on-the-fly output current that it is currently providing

All of this information received is then periodically sent within a message which is published to the message broker with a routing key "Status" which has a similar result as the routing that was shown in Figure 3.4 of the Message Broker Section. This means that the message is consumed by the HMI of the EVSE that connector is part of and also by the Decision Point, which then stores this information in its local database and acts upon it if necessary.

Another example of a situation where the Mode3 driver needs inter-module communication is when an user inserts the cable into a connector, making the control point (CP) pin value decrease from 12 Volts to 9 Volts. The information that an EV has been connected to a specific connector is then relayed with an "Event" routing key, like shown in figure 3.14.

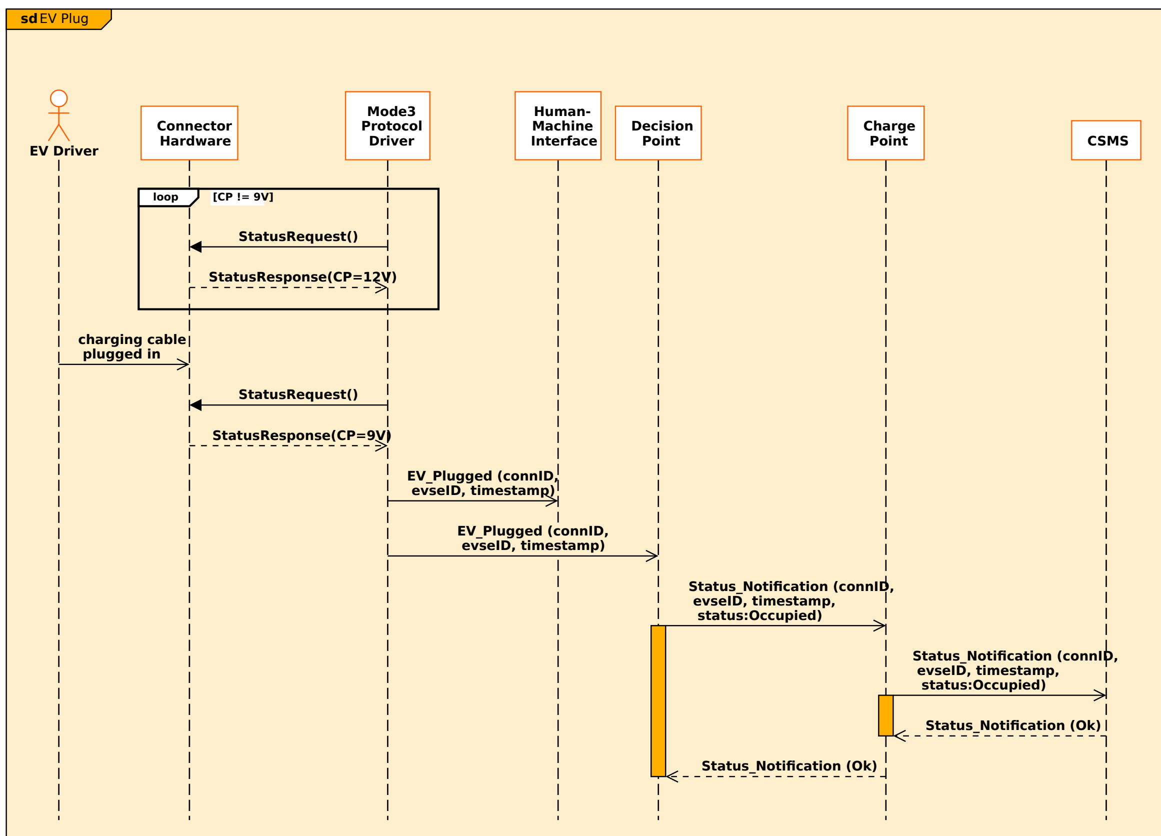


Figure 3.14: Sequence Diagram: EVPlug

Interactions with the Decision Point module go both ways as three state transitions can be seen during the "Wait for Charging" state (Figure 3.13) which are triggered by an incoming message, consumed through RabbitMQ. These messages are sent by the Decision Point module with the purpose of reducing the on-the-fly current or pausing the charging completely. They can be triggered by the CSMS sending a request to the Decision Point to reduce energy usage or by the Decision Point itself making a smart autonomous decision in trying to achieve a fair energy distribution between the EVSEs it controls.

3.6.1 Connector Hardware & EV Driver Simulator

In order to test the implementation, a simulator of the hardware board was created with the purpose of simulating the expected responses to the bytes sent by the Mode3 driver, according to the documentation (Appendix A). It can also simulate the asynchronous events of an EV Driver, like inserting and removing the cable. This way we can simulate the real-life scenario where the only information that reaches the Mode3 Driver is the information in the 'status request' response.

What is simulated is visually represented in the Figure 3.15 below, where the Simulator written in Python simulates the Connector Hardware, the EV Driver's actions and the PWM communication between the connector and the electric vehicle.

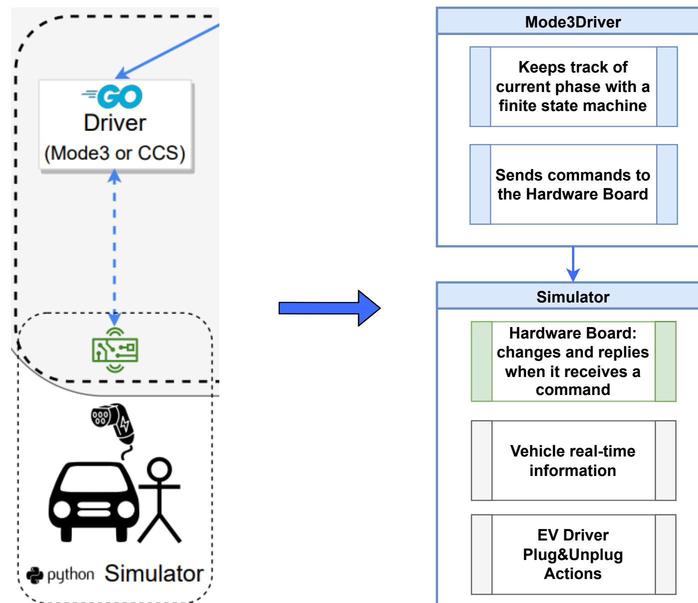


Figure 3.15: Role of Simulator in the system

An interface window created with the help of the PySimpleGUI library opens up during run-time which allows the simulation of connector related actions, such as buttons to simulate the physical cable insertion done by an EV driver. This interface contains all the outside or real world information that need to be simulated, like the pulse-width modulation phase with the duty-cycle variations or the real-time charged percentage of the car battery, information that is invisible to the Mode3Driver since it is not included in the response of the 'status request' command. This interface is shown in Figure 3.16.

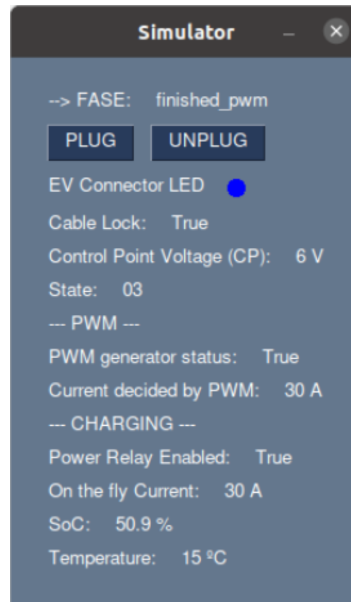


Figure 3.16: Simulator Graphical User Interface

The "Plug" button at the start makes the Mode3 Driver start receiving status responses with a control pilot (CP) value of 9Volts instead of 12V, and the "Unplug" button at the end makes it receive a value of 12Volts instead of 9V.

3.7 Energy Driver & PowerModule

The Energy Driver implements the communication in order to interact with the PowerModules. PowerModules are the hardware that transforms AC to DC, therefore, their presence in the system is only needed when the EVSE is providing DC directly to the vehicle's battery, for example in a CCS or CHAdeMO charging mode. Because of this, the EnergyDriver is not needed for Mode3 charging.

The location of the Energy Driver in the system architecture is shown in Figure 3.17.

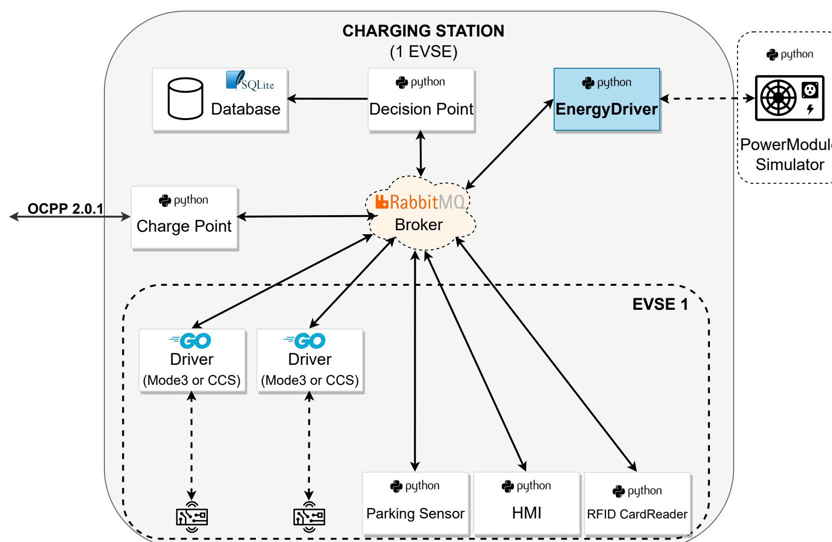


Figure 3.17: Energy Driver module highlighted in the Architecture

The Energy Driver module can handle energy requests from multiple Connectors. The protocol driver of a DC charging mode needs to internally request the current before being able to provide it. This particular sequence of events and how the messages are relayed throughout the system can be observed in Figure 3.18.

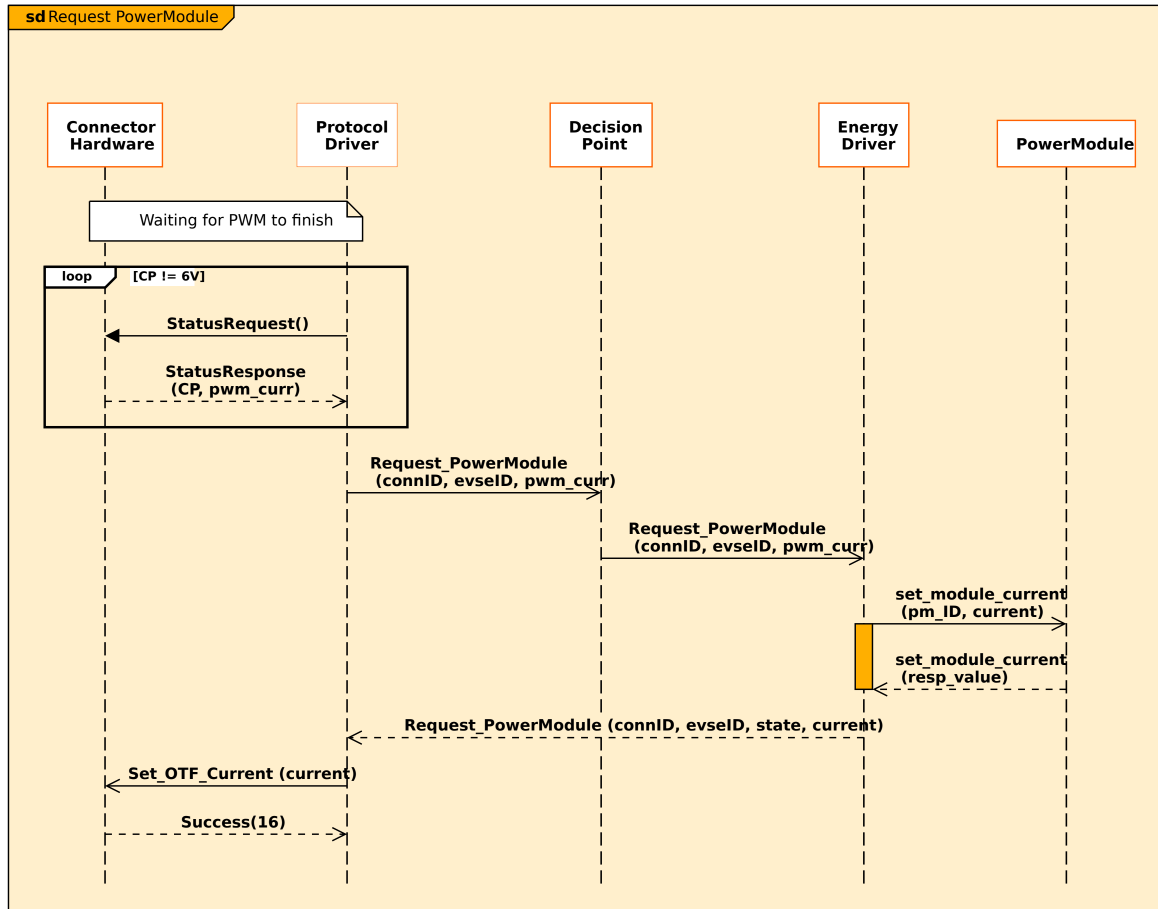


Figure 3.18: Sequence Diagram: Request PowerModule

There may not be enough available energy to fully provide the requested current, and in that case the current is given only partially, with the new smaller value given in the response. This way the Protocol Driver knows that it must not set the charging at the amount it previously requested, but set it at the amount of current that the EnergyDriver managed to provide.

Figure 3.18 shows that the communication goes through the Decision Point as it acts as the middleman between the modules in this scenario. This allows the Decision Point to contain the information of which Connectors requested current and how much, while also knowing the amount of current the EnergyDriver actually accepted to give.

Having this information allows the Decision Point to make future smart decisions. If the need to reduce energy consumption arises, it knows which Connectors are better to reduce energy from. In an opposite case of having more energy becoming available, it knows which Connectors got a lower current than they asked for and will benefit from an increase in current.

The Energy Driver also periodically polls the PowerModule, obtaining information on state, provided current and temperature. If the temperature rises above a certain threshold, the EnergyDriver takes action by reducing the current that the PowerModule is providing and, consequentially, informs the Connector that was receiving said current (and the Decision Point) of the new updated value.

3.7.1 PowerModule Hardware Simulator

A simulator of a PowerModule hardware was created with the purpose of responding to the EnergyDriver's request messages and also to simulate its internal state. To implement the internal state a simple finite state machine was developed where the state transitions are dependant on the amount of current being provided.

Besides the provided current, the states differ on how they impact the temperature and how the PowerModule reacts to new current requests. These states are more precisely described in Table 3.3.

State	Waiting	Partially Providing Energy	Fully Providing Energy
Provided Current	0	$0 < current < max$	$current == max$
Module's Temperature	Decreases or stays the same	Increases proportionally to the amount of provided current	Increases proportionally to the amount of provided current
New Requests	Accepts current requests from Connectors, providing the requested value, fully or partially	Accepts current requests from Connectors, providing the requested value, fully or partially	Does not accept new current requests from Connectors

Table 3.3: PowerModule Internal States

Despite the fact that the EnergyDriver periodically polls the PowerModule in order to control cases of overheating, a fail-safe mechanism was also implemented in case the PowerModule loses connection with the Energy Driver. This mechanism works by reducing its own output current if its temperature surpasses an even higher threshold than the one supervised by the Energy Driver.

3.8 Parking Sensor

The Parking Sensor is an optional module present in each EVSE. The objective of this module is to detect the electric vehicle when it arrives and while it remains parked within range of the EVSE. The location of the Parking Sensor module in the system architecture is shown in Figure 3.19.

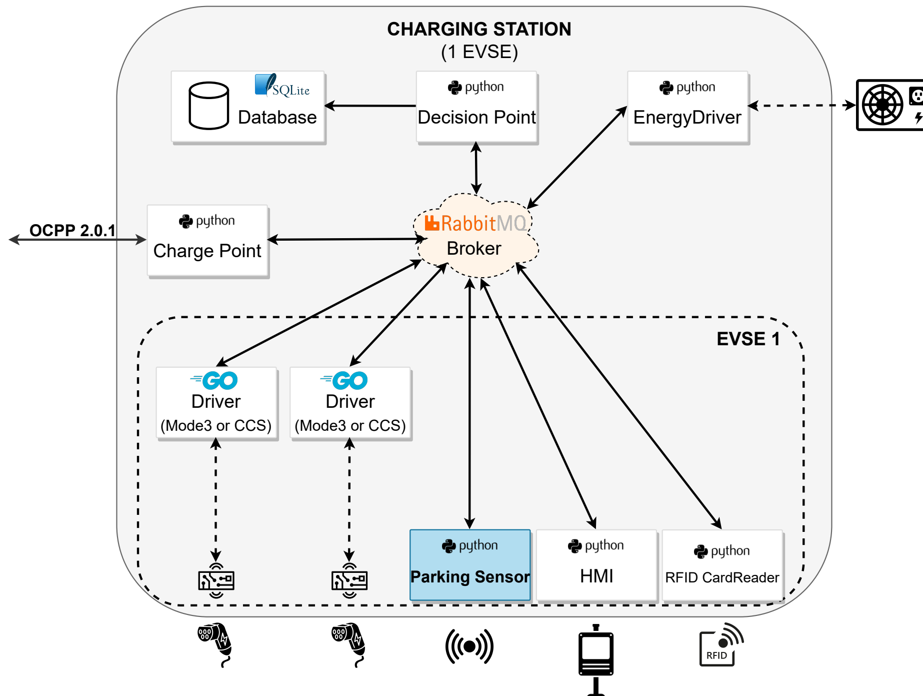


Figure 3.19: Parking Sensor module highlighted in the Architecture

Upon detecting an electric vehicle, it publishes a message with an "Events" routing key which contains the id of the EVSE the Parking Sensor is part of. When the Decision Point module consumes the message, it can enable the previously discussed "ParkingBayOccupancy" variable of that specific EVSE. If the configuration file only has this condition on the 'TxStartPoint' variable, then a transaction is started, like shown in Figure 3.20.

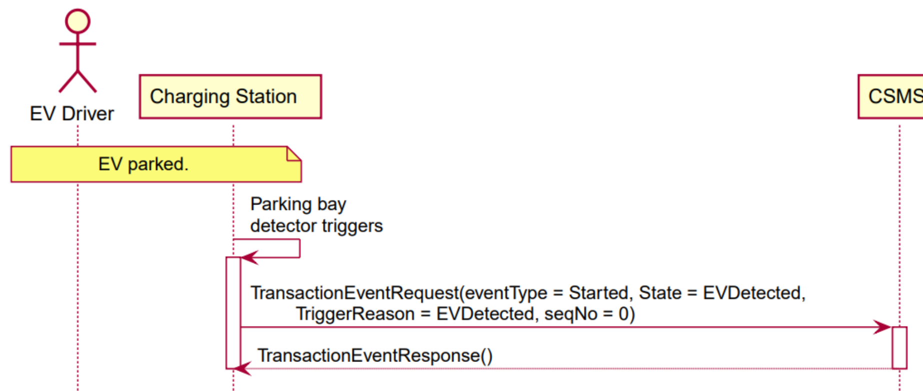


Figure 3.20: Sequence Diagram: Start Transaction options - ParkingBayOccupancy (from OCPP 2.0: Part 2 - Specification [45])

The Parking Sensor can be physically on the HMI or on the ground, and it can detect the vehicle visually with the help of AI, with weight sensors or by utilizing LiDAR technology. By accurately detecting the presence of a vehicle, it can wake up the EVSE in advance or in the absence of a vehicle it can hibernate, reducing energy consumption.

It has a simple implementation and is to be considered as a "dummy" module. An interface window created with the help of the PySimpleGUI library opens up during run-time which allows the simulation of vehicle detection with buttons to switch between parked and departed. This interface is shown in Figure 3.21.

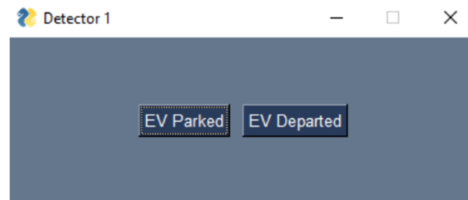


Figure 3.21: Interface Layout of the Parking Sensor module

3.9 Human-Machine Interface

In a real-life scenario at a physical charging station, the Human-Machine Interface (HMI) is the entity responsible for interacting with the user at the EVSE. The location of the HMI in the system architecture is shown in Figure 3.22.

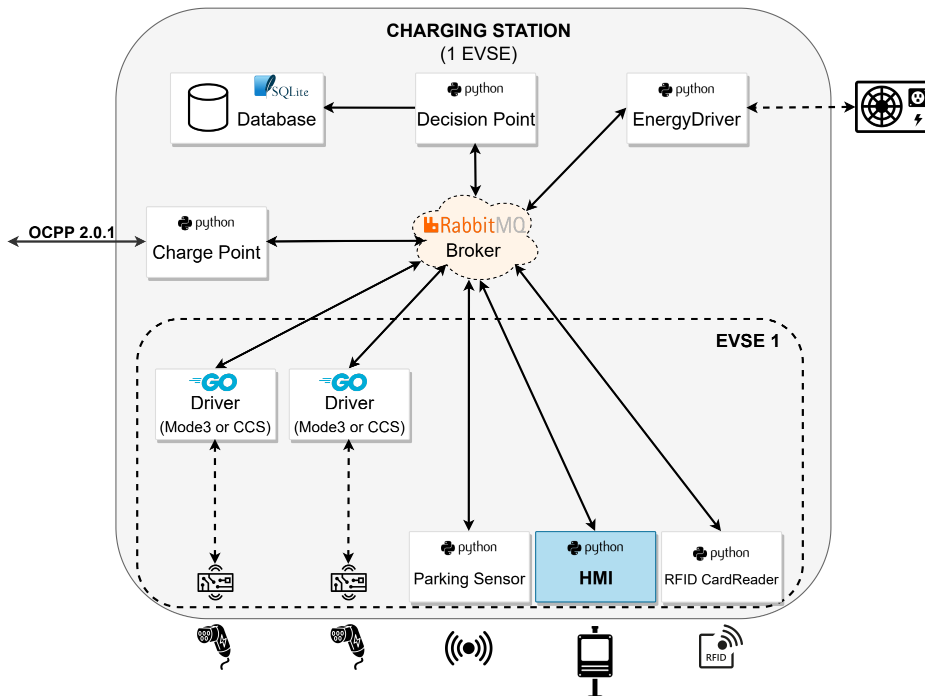


Figure 3.22: HMI module highlighted in the Architecture

It implements various functionalities, such as:

- Display messages on a screen;

- Show Information during Charging;
- Perform user authorization via PIN-code;
- Manually start and stop charging with buttons;

By displaying messages on a screen it is possible to inform or guide users on which action to perform next, such as asking the user to plug the cable or to follow an authorization method. It can also show the applicable tariff to the user's profile after the authorization step.

After the charging as been started, it can also show information about the time elapsed, the current energy value that is being provided to the vehicle and also the state of charge, which is the level of charge of the electric battery relative to its capacity, shown as a percentage.

This is possible since the HMI module consumes the "Status" routing key and therefore, consumes the periodic status from the Connector's Protocol Driver which contains the on-the-fly current output and the name of the charging protocol, among other information. The state of charge is known only if the charging is being done via CCS, since the Mode3 Driver does not receive such information.

Another functionality is the ability to perform user authorization through PIN-code. This type of authorization was first referenced in DecisionPoint's Section 3.5, where it was mentioned as an option to enable the "Authorized" variable of an EVSE, when required by the 'TxStartPoint' defined in the configuration file.

This mode of authorization was implemented in the HMI module since entering a PIN-code requires a physical interaction with the user. This mode of authorization is shown in Figure 3.23.

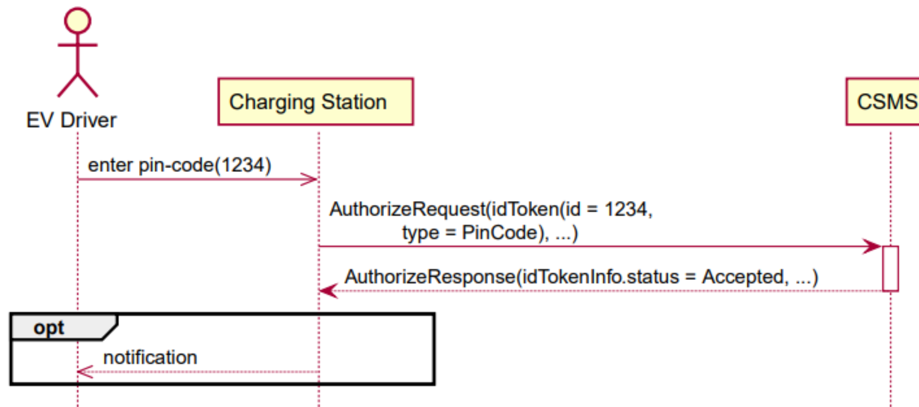


Figure 3.23: Sequence Diagram: Authorization using PIN-code (from OCPP 2.0: Part 2 - Specification [45])

In order to not only allow the user to enter a PIN-code but also to manually start and stop charging through buttons, an interface window was created. It was created with the help of the PySimpleGUI library and opens up during run-time, showing a typical PIN-code layout that consists of all single-digit numbers, an OK button and a DELETE button. It also contains both START and STOP buttons, alongside the indications and charging information that would be displayed to the EV Driver. This interface is shown in Figure 3.24.

Figure 3.24: HMI Graphical User Interface

3.10 RFID Card Reader

The RFID CardReader is an optional but common module usually present in each EVSE of public charging stations. The objective of this module is to detect and read the information of the EV Driver's RFID card. The location of the RFID CardReader module in the system architecture is shown in Figure 3.25.

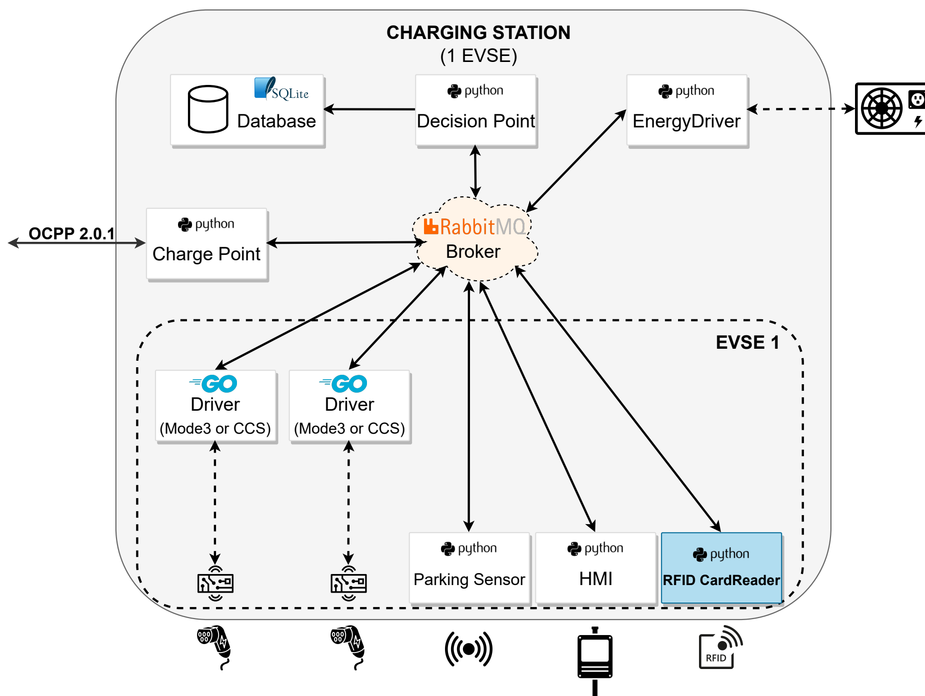


Figure 3.25: RFID CardReader module highlighted in the Architecture

The main functionality of the module is the ability to perform user authorization through a RFID card. Upon detecting the RFID card, it publishes a message with an "Events" routing key which contains the information read from the card and the id of the EVSE it is part of.

This type of authorization was first referenced in DecisionPoint's Section 3.5, where it was mentioned as an option to enable the "Authorized" variable of an EVSE, when required by the 'TxStartPoint' defined in the configuration file.

In a real-life scenario, this mode of authorization is physically close to the HMI and the way it is implemented in the system is shown in Figure 3.26.

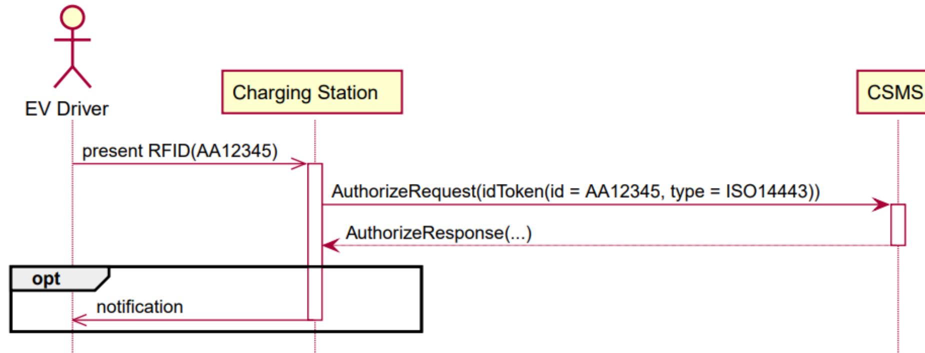


Figure 3.26: Sequence Diagram: Authorization using RFID (from OCPP 2.0: Part 2 - Specification [45])

It has a simple implementation and is to be considered as a "dummy" module. An interface window created with the help of the PySimpleGUI library opens up during run-time which allows the simulation of an EV Driver presenting a RFID card. This interface is shown in Figure 3.27.

Figure 3.27: RFID Card Reader Graphical User Interface

GitHub

All of the code developed to implement the modules described in this chapter, as well as all the created diagrams and documentation, are available in GitHub's "DETI - UA" organization at `https://github.com/detiuaveiro/EVCharger_Thesis`.

CHAPTER 4

Results

4.1 Analyzing an Entire Charging Session

After discussing the implementation of each module and their individual role in the system, the process of charging an EV from beginning to the end is going to be analyzed. The purpose of this analysis is to show how the implemented modules contribute and work together to achieve the desired functionality.

This charging scenario assumes that the 'TxStartPoint' variable of the configuration file contains this three requirements: "ParkingBayOccupancy", "EVConnected" and "Authorized".

4.1.1 Before Arriving at Charging Station

Even before the EV Driver arrives at the Charging Station, a reservation of an EVSE can be done via a HTTP Client app that interacts with the HTTP Server. The reservation contains the evseID and the EV Driver's information and is saved in a database, as shown in Figure 4.1.

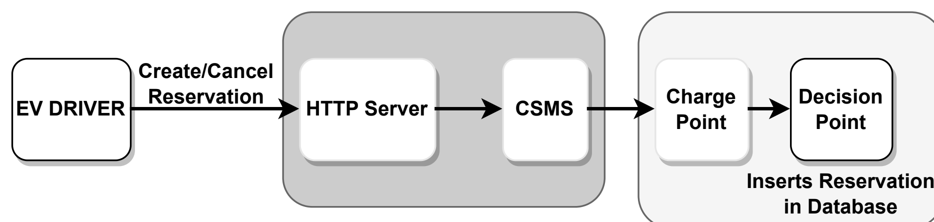
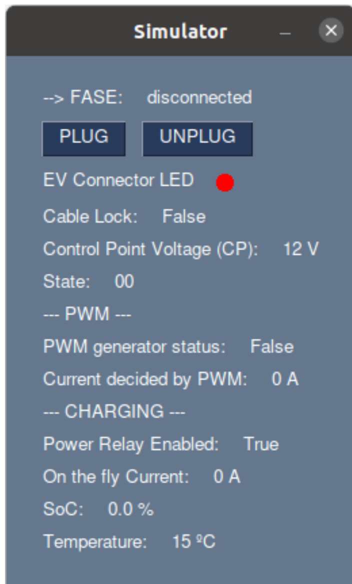
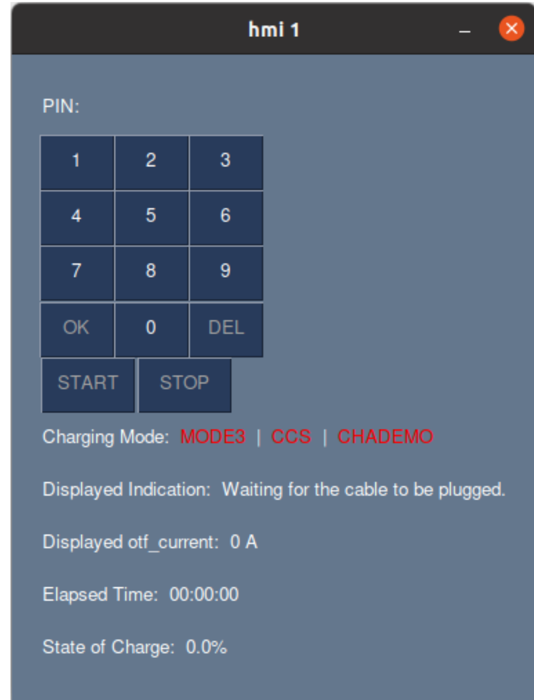


Figure 4.1: Before Arriving: Create/Cancel Reservation

While waiting for an user, the Graphical User Interface (GUI) of both the Simulator and the HMI are shown in Figure 4.2.



(a) Simulator GUI



(b) HMI GUI

Figure 4.2: Simulator and HMI: Before Arriving

4.1.2 Arriving at Charging Station

When the EV Driver arrives at the Charging Station, the vehicle is detected by the Parking Sensor, which acts like shown in Figure 4.3. When the EV Driver plugs the cable the Control Pilot pin goes from 12 Volts to 9 Volts. The impact of this event on the system is shown in Figure 4.4.

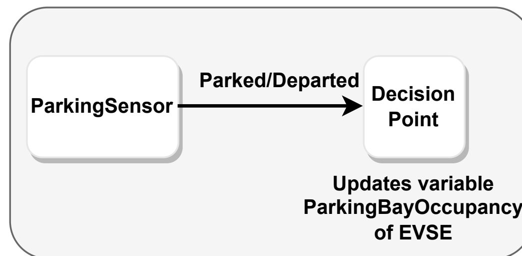


Figure 4.3: At arrival: ParkingSensor detects EV

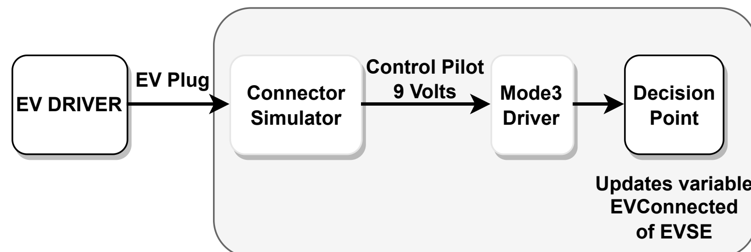


Figure 4.4: At arrival: EV Plug

After plugging the cable, the GUI of both the Simulator and the HMI are shown in Figure 4.5.

(a) Simulator GUI: CP at 9 Volts and waiting to start PWM

(b) HMI GUI: Highlights charging mode and Requests authorization from EV Driver

Figure 4.5: Simulator and HMI: EVPlug

4.1.3 Authorization at Charging Station

A authorization step may be required and the indication for the EV Driver to choose a method of authorization is shown in the HMI's screen display. The EV Driver can present a RFID card (Figure 4.6) or enter a PIN-code (Figure 4.7). Either way the information has to be confirmed by the Central System.

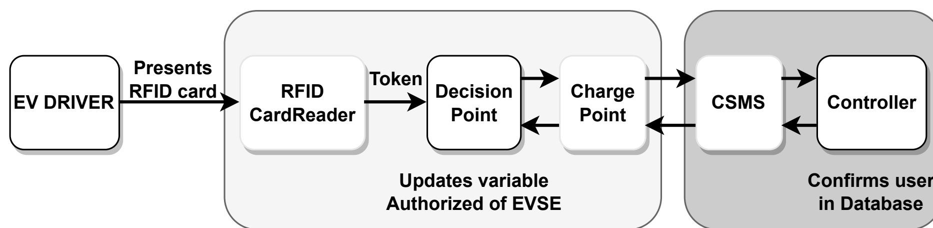


Figure 4.6: Authorization: RFID Card

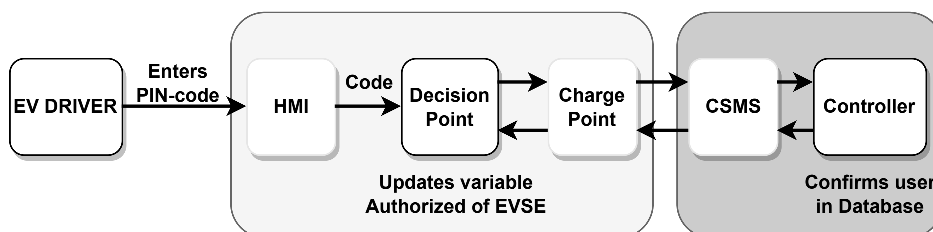


Figure 4.7: Authorization: PIN-code

When the EV Driver tries to authorize himself, the GUI of the HMI changes as shown in Figure 4.8.

(a) HMI GUI: Waiting for authorization confirmation

(b) HMI GUI: Indicates to EV Driver that it is ready to start charging

Figure 4.8: Simulator and HMI: Authorization

Upon receiving the Central System's confirmation, the response message can also contain which SalesTariffType is applicable to this user. This is why the Decision Point also contains a SalesTariff table in its database. After consulting the tariff's information, the Decision Point can send this information to the HMI in order to be displayed as shown in Figure 4.9.

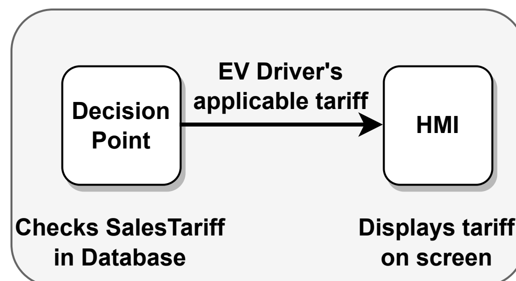


Figure 4.9: After Authorization: Displaying tariff

However, this is only useful for situations where the Charging Stations, the CSMS and the energy provider are all owned by the same party, which then can provide an accurate charging cost to the EV Driver before the charging even starts, since there is no other parties involved where the cost would depend on.

Since it is common that all of these three sectors are owned by different companies and

that the charging cost is only calculated at the end by the operator, this was the approach chosen to implement this feature, meaning that the SalesTariff is not used, even though its functionality was partially developed.

4.1.4 Start Charging Session

At this point the vehicle has been detected by the Parking Sensor, the electric vehicle has been plugged to the EVSE and the EV Driver has confirmed their identity with an authorization method. In order to start charging there are two options: the EV Driver can press the start button on the HMI (Figure 4.10) or use an app to perform a remote start (Figure 4.11).

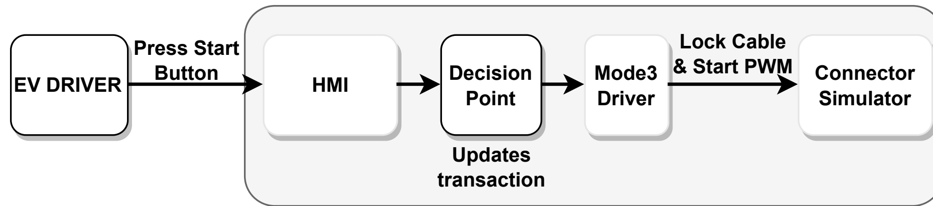


Figure 4.10: Start Charging: Start Button

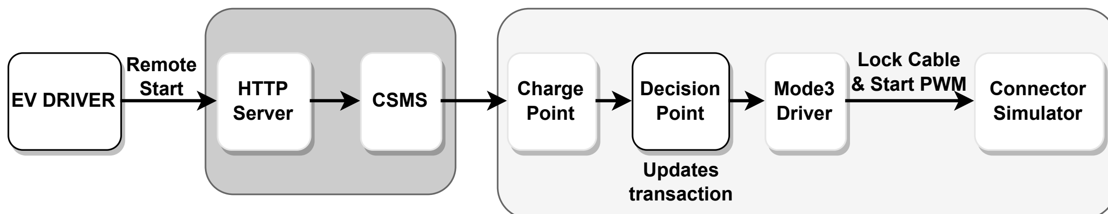


Figure 4.11: Start Charging: Remote Start

The PWM phase where the Connector Hardware and the electric vehicle are deciding on a charging current by alternating the duty-cycle is done in this system by the "Connector Simulator" module. After the PWM decides on a current, the Control Pilot pin goes from 9 Volts to 6 Volts. The way this event unfolds throughout the system is shown in Figure 4.12.

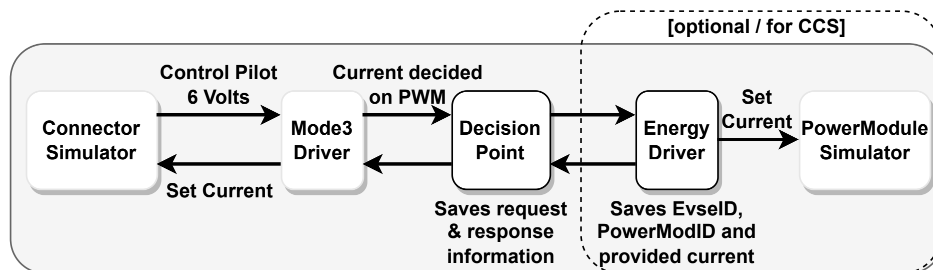
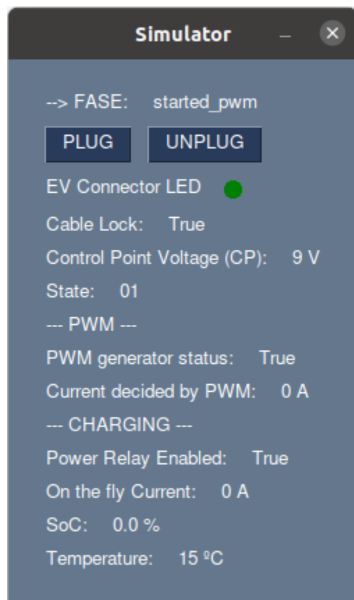
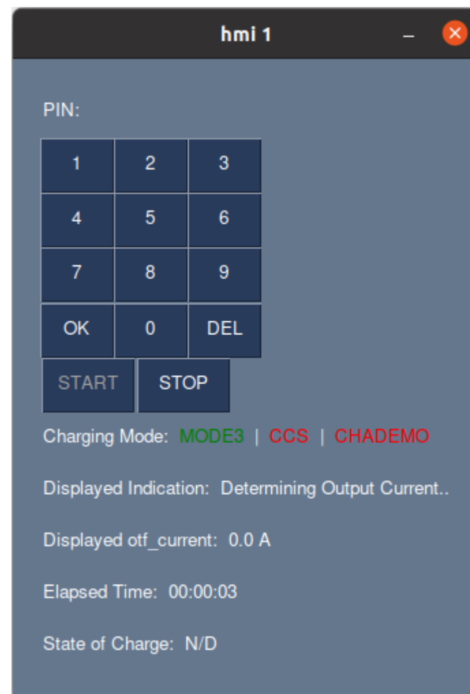


Figure 4.12: After PWM: Requesting Current decided by PWM

While the PWM phase is currently ongoing, the GUI of both the Simulator and the HMI is shown in Figure 4.13.



(a) Simulator GUI: Cable locked and PWM Generator started



(b) HMI GUI: Waiting for output current to be determined

Figure 4.13: Simulator and HMI: PWM

4.1.5 During Charging Session

After knowing which amount of current to use, charging begins and energy starts flowing from the EVSE to the vehicle's on-board charger (if Mode3) or to the electric vehicle's battery (if DC charging). During charging the Mode3 Driver periodically sends the connector's status like shown in Figure 4.14.

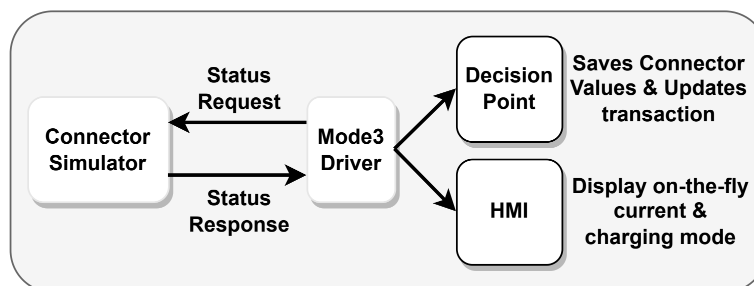
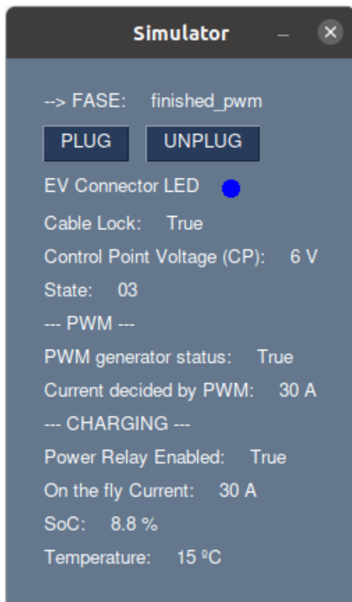
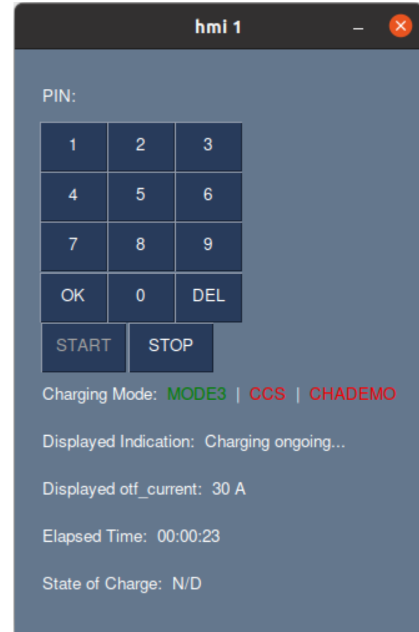


Figure 4.14: During Charging: Connector Status

During charging, the GUI of both the Simulator and the HMI is shown in Figure 4.15.



(a) Simulator GUI: Set on the fly current and CP at 6 Volts



(b) HMI GUI: Shows elapsed time and on the fly current

Figure 4.15: Simulator and HMI: Charging Ongoing

Requests may arrive to the Mode3 Driver to either reduce current (Figure 4.16) or to completely interrupt power relay (Figure 4.17). This messages come from the Decision Point but can, in some cases, originate from the Central System.

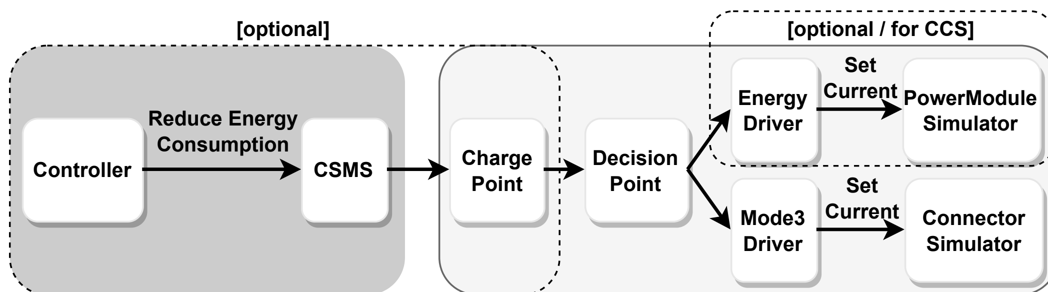


Figure 4.16: During Charging: Reduce Energy Consumption

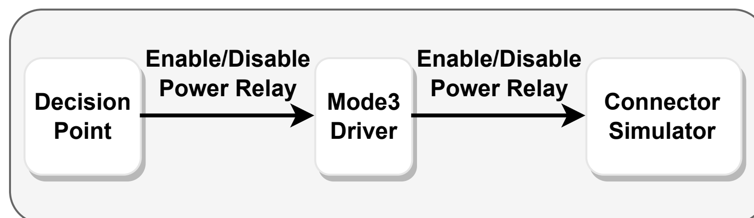


Figure 4.17: During Charging: Enable/Disable Power Relay

In case it detects overheating problems, the Energy Driver may decide to decrease the amount of current it is currently providing to a certain connector during charging. This functionality is shown in Figure 4.18.

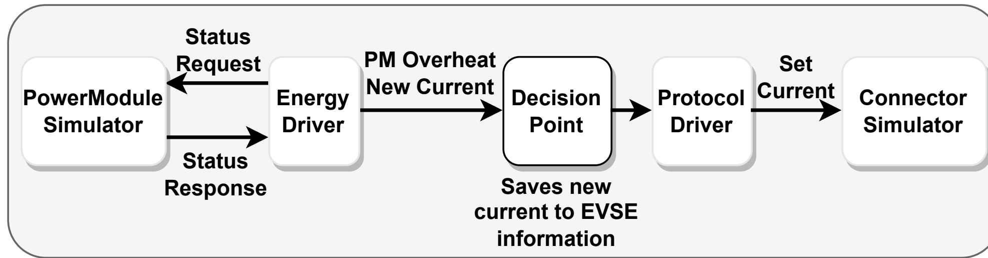
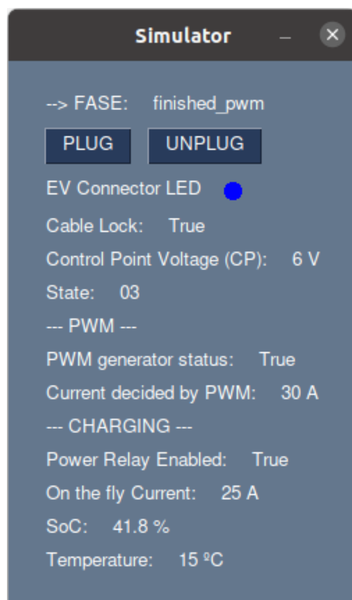
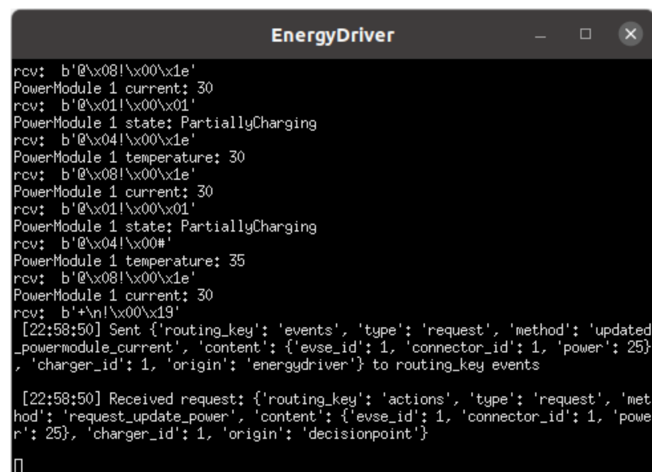


Figure 4.18: During Charging: PowerModule Overheating (for CCS)

Even though using the EnergyDriver is only necessary when providing DC, in order to test its functionality, there is the option to still have the module present when charging in Mode3. This is why there is an optional "Wait for PowerModule" state in its finite state machine, which allowed to test these functionalities and get the picture below. The reaction of the EnergyDriver after seeing the PowerModule overheat, as well as the GUI of the Simulator after the current change is shown in Figure 4.19.



(a) Simulator GUI: On the fly current reduced



(b) EnergyDriver: Detects overheating and requests an update of current

Figure 4.19: Simulator and EnergyDriver: Updating Current

4.1.6 Stop Charging Session

There are two ways for a charging session to be stopped. When the vehicle's battery becomes fully charged, the Control Pilot pin goes from 6 Volts to 9 Volts indicating that charging is finished, as shown in Figure 4.20. The other way is by pressing the stop button on the HMI while a charging session is ongoing, as shown in Figure 4.21.

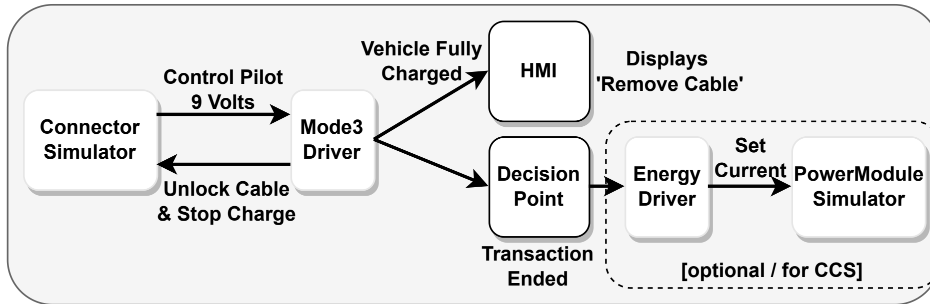


Figure 4.20: Stop Charging: Vehicle Fully Charged

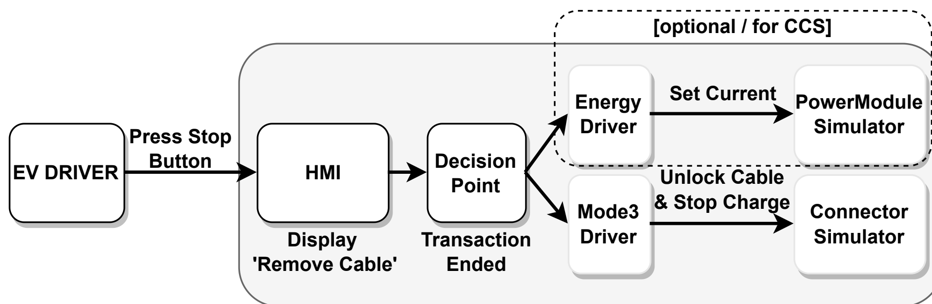
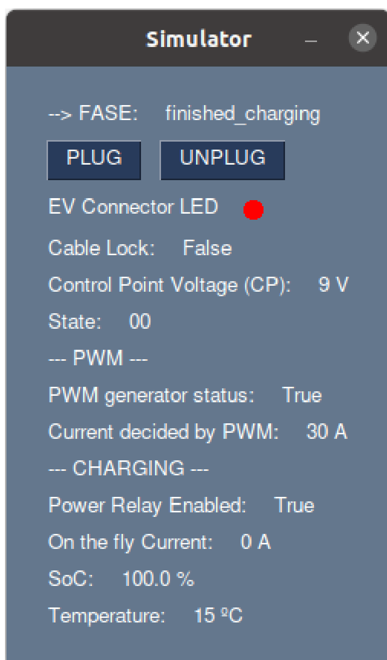
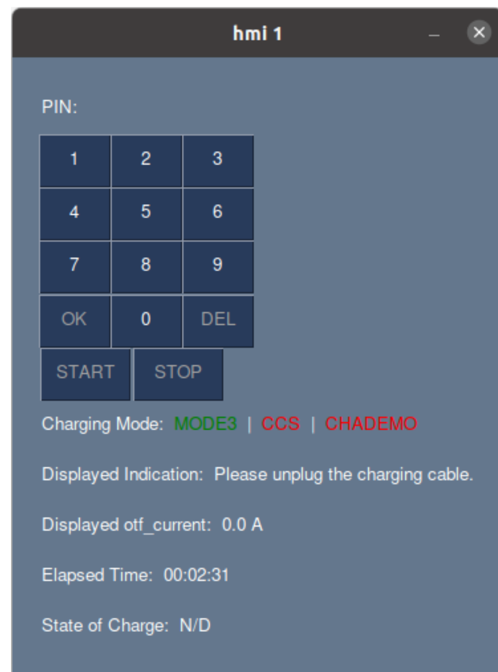


Figure 4.21: Stop Charging: Stop Button

After ending the session via a button or because the vehicle has been fully charged, the GUI of both the Simulator and the HMI is shown in Figure 4.22.



(a) Simulator GUI: Cable unlocked and Output current set to zero



(b) HMI GUI: Indicates the EV Driver to unplug cable

Figure 4.22: Simulator and HMI: Waiting for cable unplug

4.1.7 After Charging

After charging ends the EV Driver is required to unplug the vehicle which causes the Control Pilot pin to go from 9 Volts to 12 Volts, as shown in Figure 4.23.

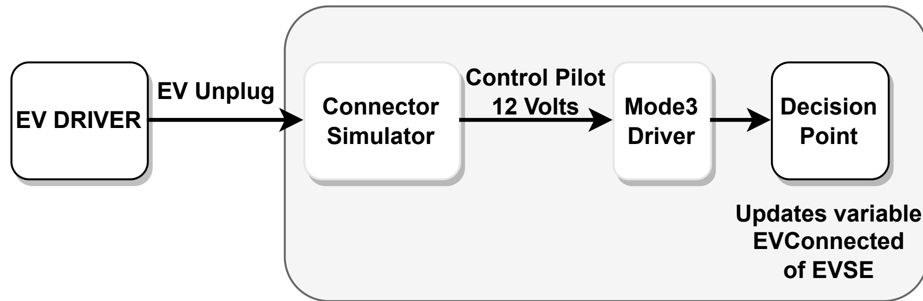


Figure 4.23: After Charging: EV Unplug

As previously explained, due to many factors it is not possible to calculate a charging cost in real time. To obtain the payment amount, the Decision Point needs to send the several transaction logs it saved during charging to an Operator, which calculates the charging cost and returns the bill. This is shown in Figure 4.24, however, since developing an Operator was outside of the project's scope, this responsibility was implemented in the Controller.

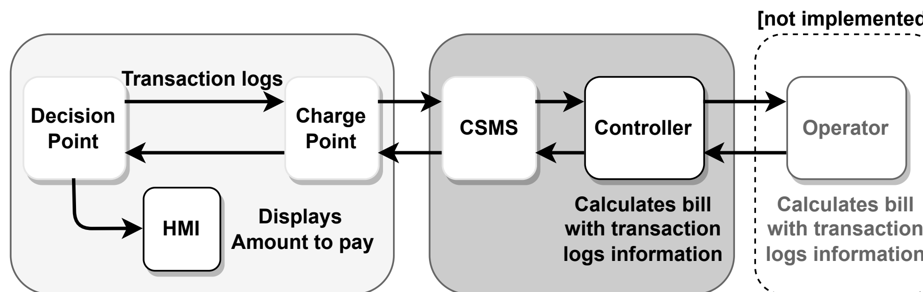


Figure 4.24: After Charging: Display Charging Cost

4.2 Mode3Driver Finite State Machine - Unit Testing

As described in the module's section, the "Mode3Driver" organizes its logic with a finite state machine, shown again in Figure 4.25 below.

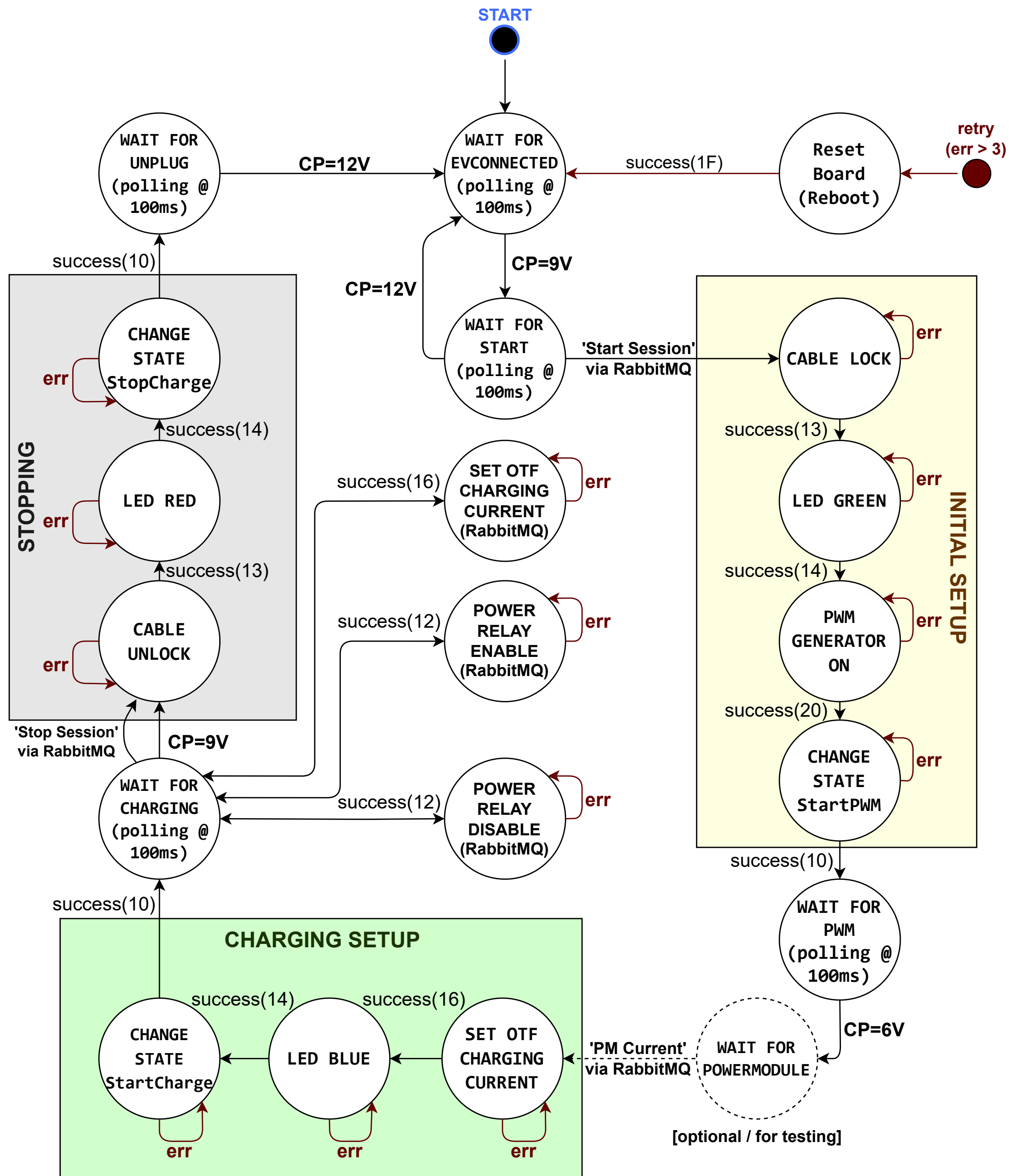


Figure 4.25: Mode3Driver Finite State Machine

With the purpose of confirming that the state transitions are acting as intended, a unit test was developed. A unit test is a way of testing a piece of code that can be logically isolated in a system, usually a function or a method.

Since the Mode3Driver was written in the Golang programming language, to develop this test the Golang testing package[1] was utilized. The testing package provides the tools needed to write the unit tests, which are then executed by the "go test" command.

[1] `https://pkg.go.dev/testing`

The developed unit test focused on the “step(state State)” function, which receives the current state and, depending on the input, alters the value of the “nextState” variable.

Due to the repetitiveness of writing tests, it is common to use a table-driven style, where test inputs and expected outputs are listed in a table and a single loop walks over them and performs the test logic. This was the way the unit test was developed in order to test all possible inputs and their impact on state transitions.

The unit testing results are shown in the following Table 4.1.

Starting State	Input	Expected nextState	Testing Result
WAIT FOR EV	Status (CP=9V)	WAIT FOR START	✓
	Status (CP=12V)	WAIT FOR EV	✓
WAIT FOR START	Status (CP=12V)	WAIT FOR EV	✓
	Status (CP=9V)	WAIT FOR START	✓
	’Start Session’	CABLE LOCK	
CABLE LOCK	Success	LED GREEN	✓
	Error	CABLE LOCK	✓
LED GREEN	Success	PWM GENERATOR ON	✓
	Error	LED GREEN	✓
PWM GENERATOR ON	Success	CHANGE STATE STARTPWM	✓
	Error	PWM GENERATOR ON	✓
CHANGE STATE STARTPWM	Success	WAIT FOR PWM	✓
	Error	CHANGE STATE STARTPWM	✓
WAIT FOR PWM	Status (CP=6V)	SET OTF CURRENT	✓
	Status (CP=9V)	WAIT FOR PWM	✓
WAIT FOR POWERMODULE	’PM Current’	SET OTF CURRENT	
SET OTF CURRENT	Success	LED BLUE	✓
	Error	SET OTF CURRENT	✓
LED BLUE	Success	CHANGE STATE STARTCHARGE	✓
	Error	LED BLUE	✓
CHANGE STATE STARTCHARGE	Success	WAIT FOR CHARGING	✓
	Error	CHANGE STATE STARTCHARGE	✓

Table 4.1: Finite State Machine: Unit Testing (Part 1)

Starting State	Input	Expected nextState	Testing Result
WAIT FOR CHARGING	Status (CP=9V)	CABLE UNLOCK	✓
	Status (CP=6V)	WAIT FOR CHARGING	✓
	'Update Power'	SET OTF CURRENT (rabbitmq)	
	'Enable Power'	POWER RELAY ENABLE (rabbitmq)	
	'Disable Power'	POWER RELAY DISABLE (rabbitmq)	
	'Stop Session'	CABLE UNLOCK	
CABLE UNLOCK	Success	LED RED	✓
	Error	CABLE UNLOCK	✓
LED RED	Success	CHANGE STATE STOPCHARGE	✓
	Error	LED RED	✓
CHANGE STATE STOPCHARGE	Success	WAIT FOR UNPLUG	✓
	Error	CHANGE STATE STOPCHARGE	✓
WAIT FOR UNPLUG	Status (CP=12V)	WAIT FOR EV	✓
	Status (CP=9V)	WAIT FOR UNPLUG	✓
SET OTF CURRENT (rabbitmq)	Success	WAIT FOR CHARGING	✓
	Error	SET OTF CURRENT (rabbitmq)	✓
POWER RELAY ENABLE (rabbitmq)	Success	WAIT FOR CHARGING	✓
	Error	POWER RELAY ENABLE (rabbitmq)	✓
POWER RELAY DISABLE (rabbitmq)	Success	WAIT FOR CHARGING	✓
	Error	POWER RELAY DISABLE (rabbitmq)	✓
RESET BOARD	Success	WAIT FOR EV	✓

Table 4.1: Finite State Machine: Unit Testing (Part 2)

There are some transitions with an empty "Testing Result". This is because the required input was a consumed RabbitMQ message. In the module's implemented logic, in these situations, updating of the "nextState" variable is done by the consumer itself after consuming the respective message, therefore it was not possible to test those specific transitions in this unit test.

All the other transitions, which had an input of bytes that would come from the Connector Hardware's reply, showed a positive testing result, confirming that the logic is working as intended.

4.3 Mode3Driver Time Requirements Analysis

There are some expected time requirements in the communication between the Connector Hardware and the Electric Vehicle. This section focuses on analyzing the time it takes from ending the PWM phase until the charging actually begins.

The PWM phase ends when the Mode3Driver detects that the control pilot has a value of 6 Volts, in the "Wait for PWM" state. The charging begins when the Connector Hardware receives the command to start charging, which is done in the "Change State StartCharge" state.

Therefore, a test was made which gathered the elapsed time from reading a status response containing a control pilot value of 6, until the Mode3Driver finally sends the command to start charging. As seen in Figure 4.25 above, this means going through the following states: "Wait for PWM", "Set OTF Current", "Led Blue", "Change State StartCharge" and, optionally, "Wait for PowerModule".

Testing the normal functionality of the Mode3 Driver provides very fast results since the communication was implemented with a simulator that answers instantly via a UNIX socket. An example of 10 gathered values in a normal execution of Mode3 is shown in Figure 4.26:

```
0- 2022/10/28 15:35:52 End of PWM to Start Charging took 308.696µs
1- 2022/10/28 15:37:50 End of PWM to Start Charging took 383.564µs
2- 2022/10/28 15:38:54 End of PWM to Start Charging took 283.613µs
3- 2022/10/28 15:39:24 End of PWM to Start Charging took 226.449µs
4- 2022/10/28 15:40:26 End of PWM to Start Charging took 166.216µs
5- 2022/10/28 15:42:01 End of PWM to Start Charging took 795.905µs
5- 2022/10/28 15:42:59 End of PWM to Start Charging took 1.721355ms
6- 2022/10/28 15:43:37 End of PWM to Start Charging took 171.825µs
7- 2022/10/28 15:44:19 End of PWM to Start Charging took 6.34318ms
8- 2022/10/28 15:45:45 End of PWM to Start Charging took 379.319µs
9- 2022/10/28 15:46:23 End of PWM to Start Charging took 381:249µs
```

Figure 4.26: Time Requirement Analysis: normal Mode3

This result shows an average of $1.11ms$.

However, this test started counting the time only from the moment it read 6 Volts in the control pilot pin. Since status polling is only done every $100ms$, in the worst case scenario the vehicle can change the control pilot value and only after $100ms$ does the Mode3Driver initiate the process to start the charge. This means that an extra $100ms$ should be expected or at least accounted for in the average result above.

Another test was done by including the "Wait for PowerModule" state. In this test, there was an obvious bottleneck by the added state, since it requests the current decided by PWM and awaits for a response, meaning that it is dependant on the EnergyDriver module and on the message broker performance.

Testing with this state allows to understand the impact of a state that depends on another module of the system. This sets an expectation for a future developed Protocol Driver of a DC charging mode that would require to interact with the Energy Driver.

An example of 10 gathered values while using the EnergyDriver is shown in Figure 4.27:

```
0- 2022/10/22 15:34:46 End of PWM to Start Charging took 201.480070ms
1- 2022/10/22 15:35:58 End of PWM to Start Charging took 202.872699ms
2- 2022/10/22 15:36:30 End of PWM to Start Charging took 202.334083ms
3- 2022/10/22 15:36:56 End of PWM to Start Charging took 202.506235ms
4- 2022/10/22 15:37:42 End of PWM to Start Charging took 202.514921ms
5- 2022/10/22 15:38:06 End of PWM to Start Charging took 202.133085ms
6- 2022/10/22 15:45:11 End of PWM to Start Charging took 204.268758ms
7- 2022/10/22 15:45:53 End of PWM to Start Charging took 202.203654ms
8- 2022/10/22 15:46:42 End of PWM to Start Charging took 202.402712ms
9- 2022/10/22 15:47:28 End of PWM to Start Charging took 202.103141ms
```

Figure 4.27: Time Requirement Analysis: with optional EnergyDriver

This result shows an average of 202.47ms, with also the possible worst case scenario of an extra 100ms due to the polling interval.

4.4 HTTP Server Testing

The HTTP Server is supposed to be a entry point to the system, allowing requests and providing responses for administrators and operators while also handling actions for all the possible clients interacting with it via a future mobile application.

Depending on how many Charging Stations the Central System manages and also how many EVSEs are in each station, the possible number of requests and load the HTTP Server needs to be ready to handle can be somewhat big.

Therefore, in the previous implementation [7] some tests were created in order to test the correct functionality of the HTTP Server and its capability in handling higher loads.

The k6[2] tool was utilized in order to do the following tests on the module:

- Smoke testing[3]: acts as a sanity check to see if the system can handle minimal load without any problems;
- Load testing[4]: evaluates the current performance of the system in terms of concurrent users or requests per second;
- Stress testing[5]: a load test but with the purpose of testing the availability and stability of the system while under extreme conditions;

4.4.1 Smoke Testing

A brief Smoke Test was created which tested the "/" and "/login" endpoints of the HTTP Server. The implemented threshold required that 99% of requests must be completed in less than 2.5 seconds. This higher value can be justified as the "/" endpoint makes several requests to fetch data to display on the browser. This specific test only indicates the generic performance of the HTTP Server in terms of Web Page fetching.

The login functionality and the "/" worked as intended, with each request taking an average of 1.03.

[2] `https://k6.io/docs/test-types/`
[3] `https://k6.io/docs/test-types/smoke-testing/`
[4] `https://k6.io/docs/test-types/load-testing/`
[5] `https://k6.io/docs/test-types/stress-testing/`

Another smoke test was created that briefly tested the performance of the HTTP Server when interacting with other components of the Central System. The remotely requested cancel reservation was the chosen method.

Since the HTTP Server just redirects the incoming requests to the Central System itself, the time it took to complete was independent of the internal execution of that request. Consequently, this resulted in low times, achieving an average of 13.08ms over 10 iterations.

4.4.2 Load Testing

In order to examine how the system performed under normal and peak situations, load tests were created. The values considered for normal and peak situations were 100 and 200 virtual users, respectively. It experiments the interaction between a mobile application user (the EV Driver) and the HTTP Server. Two different load tests were created and are explained below.

For the first test, it was assumed that the virtual users only cycled through "/stations/page/1/", "/stations/page/2" and "/reservations", which are endpoints that only fetch data and do not need to interact with other system components. A threshold of maximum 5 seconds was applied to the HTTP request duration, but it failed since the maximum duration seen was 33.99 seconds, even though the average request duration was 4.35 seconds. The full operation cycle of the virtual user took an average of 30.51 seconds and the test did not have a single failed request.

In the second test, some endpoints that do interact with the Central System were tested, such as "/cancel_reservation", "/request_start_transaction" and "/reserve_now". The HTTP Server did not break at any time even when the 200 virtual users made the three requests simultaneously. It took an average of 700ms to receive a response to a request. The full operation cycle of the virtual user took an average of 3.8 seconds.

4.4.3 Stress Testing

To find out the HTTP Server's maximum capacity and its breaking point, a stress test was created. The test was configured in many gradual steps, with some steps increasing the concurrent load, while other steps just holding the established load for an amount of time. Two different stress tests were executed and are explained below.

The first test assumed that the 600 virtual users only cycled through "/stations/page/1/", "/stations/page/2" and "/reservations", which are the endpoints that take more time as they fetch data to display. The results showed that 7% of the requests failed, with an average duration of 21.45 seconds per request. It is important to note that the test was running on a personal computer not suited for such quantities of solicitations, meaning that the results can be improved by utilizing a more powerful server.

The second test assumed that the 600 virtual users cycled through "/cancel_reservation", "/request_start_transaction" and "/reserve_now", which are endpoints that take less time since the requests are executed by the Central System. The results showed that 2.78% of the requests failed, with an average duration of 2.44 seconds per request.

A request is deemed as failed on the absence of a response, but since the execution of these requests is done in the Central System, some of the failed requests may have actually been successfully complete.

4.4.4 Considerations

The test results could have been better if the system was deployed on a proper hardware.

It is important to notice that the Central System used for the endpoints that interacted with it was connected to only one Charging Station, overloading its Decision Point.

This would be a rare situation in real-life, as there is a limited number of EVSEs per Charging Station, meaning that it is unlikely that there would be 200 users simultaneously interacting with it.

CHAPTER 5

Conclusion & Future Work

5.1 Conclusion

The conceptualized system architecture using the newest version of OCPP that was developed presents itself as a good solution for the issues listed in the beginning. It details a possible implementation that can be used in actual businesses by being modular, scalable, robust and easily updatable, while at the same time being future-proof since it implements the newest version of the protocol.

There is room for improvement in the developed system since the implementation focused more on tangible features and functionalities that allow a whole charging session to occur. For the charging station to be utilized in a real-life scenario, the Decision Point would need to implement more OCPP functions and their logic, specifically the ones that allow firmware updates, exchange of certificates and ISO 15118 communication, since to be competitive in the current charging station market, the option of charging in DC/CCS should be provided. The autonomous decisions to achieve energy balance could also be improved, since the current implementation is only ready to receive these requests from the Central System, with the Decision Point not acting by itself.

In terms of making the system as realistic as possible, the Mode3 Driver and the Energy Driver could benefit from testing with real hardware to definitely confirm their potential, as well as the Parking Sensor and the RFID CardReader, which could also be implemented with real hardware.

The broad landscape of this topic allows multiple fronts of the project that could be continued and developed on top of, but in the end, a positive outcome can be considered as most objectives were achieved, with the project presenting itself as a good foundation for a Central System and Charging Station implementation, with the most important functionalities for an user's charging experience implemented.

5.2 Future Work

As future work, there are some elements that can be implemented:

- develop an actual Parking Sensor and a RFID CardReader instead of them being dummy modules and also test the Mode3 Driver and the Energy Driver with real hardware instead of simulated hardware;
- improving the Decision Point module making it usable in a real-life scenario by developing the logic of OCPP functions that handle certificates, firmware updates and that allow ISO 15118 for DC charging;
- introducing a CCS driver to the system which can be implemented seamlessly as long as it implements the same RabbitMQ communications of the Mode3 counterpart, since the EnergyDriver is already developed and because the system handles protocol Drivers the same, regardless of charging mode;
- develop the mobile application that interacts with the HTTP Server and allows users to perform remote actions, as designed in the mock-up in the previous implementation;
- create a module or system that acts as an Operator in order to have a complete simulated system, with the charging cost being calculated in this module and it having access to all Charging Station's information via the HTTP Server;

Referências

[1] M. Koengkan, J. A. Fuinhas, M. Teixeira, *et al.*, "The capacity of battery-electric and plug-in hybrid electric vehicles to mitigate co2 emissions: Macroeconomic evidence from european union countries," *World Electric Vehicle Journal*, vol. 13, no. 4, p. 58, 2022.

[2] J. A. Fuinhas, M. Koengkan, N. C. Leitão, *et al.*, "Effect of battery electric vehicles on greenhouse gas emissions in 29 european union countries," *Sustainability*, vol. 13, no. 24, p. 13 611, 2021.

[3] M. Siddi, "The european green deal: Asseasing its current state and future implementation," 2020.

[4] C. Fetting, "The european green deal," *ESDN Report, December*, 2020.

[5] J. S. Johansen *et al.*, "Fast-charging electric vehicles using ac," *Technical University of Denmark*, 2013.

[6] C. Ricaud and P. Vollet, "Connection method for charging systems–a key element for electric vehicles," in *Schneider Electric Conference, France*, 2010, pp. 1–11.

[7] A. C. R. Machado, "Sistema de controle de carregamento de veículos elétricos por ocpp 2.0," Master's Thesis, Universidade de Aveiro, 2021.

[8] T. V. Pruthvi, N. Dutta, P. B. Bobba, and B. S. Vasudeva, "Implementation of ocpp protocol for electric vehicle applications," in *E3S Web of Conferences*, EDP Sciences, vol. 87, 2019, p. 01 008.

[9] O. C. Alliance, "Ocpp 2.0: Part 1 - architecture & topology," *April*, 2018.

[10] D. Wellisch, J. Lenz, A. Faschingbauer, R. Pöschl, and S. Kunze, "Vehicle-to-grid ac charging station: An approach for smart charging development," *IFAC-PapersOnLine*, vol. 48, no. 4, pp. 55–60, 2015.

[11] D. LLC, "Ac vs dc (alternating current vs direct current)," *Diffen.com*, 2022.

[12] N. S. Pearre, W. Kempton, R. L. Guensler, and V. V. Elango, "Electric vehicles: How much range is required for a day's driving?" *Transportation Research Part C: Emerging Technologies*, vol. 19, no. 6, pp. 1171–1184, 2011.

[13] M. C. Falvo, D. Sbordone, I. S. Bayram, and M. Devetsikiotis, "Ev charging stations and modes: International standards," in *2014 International Symposium on Power Electronics, Electrical Drives, Automation and Motion*, IEEE, 2014, pp. 1134–1139.

[14] Wikipedia, *Type 2 connector — Wikipedia, the free encyclopedia*, `http://en.wikipedia.org/w/index.php?title=Type%202%20connector&oldid=1108318498`, 2022.

[15] ——, *CHAdeMO — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/wiki/CHAdeMO`, 2022.

[16] I. E. Commission *et al.*, "Iec 61851-1: 2017-02 "electric vehicle conductive charging system—part 1: General requirements"," *International Electrotechnical Commission: Geneva, Switzerland*, 2017.

[17] J. Schmutzler, C. A. Andersen, and C. Wietfeld, "Evaluation of ocpp and iec 61850 for smart charging electric vehicles," *World Electric Vehicle Journal*, vol. 6, no. 4, pp. 863–874, 2013.

[18] S. Ruthe, J. Schmutzler, C. Rehtanz, and C. Wietfeld, "Study on v2g protocols against the background of demand side management.," *Int. J. Interoperability Bus. Inf. Syst.*, vol. 11, pp. 33–44, 2011.

[19] C. Dericioglu, E. Yirik, E. Unal, M. Cuma, B. Onur, and M. Tumay, "A review of charging technologies for commercial electric vehicles," *International Journal of Advances on Automotive and Technology*, vol. 2, no. 1, pp. 61–70, 2018.

[20] T. Braunl, "Ev charging standards," *University of Western Australia: Perth, Australia*, pp. 1–5, 2012.

[21] C. Lewandowski, S. Gröning, J. Schmutzler, and C. Wietfeld, "Interference analyses of electric vehicle charging using plc on the control pilot," in *2012 IEEE International Symposium on Power Line Communications and Its Applications*, IEEE, 2012, pp. 350–355.

[22] Driivz, *The ev charging industry protocols and standards list*, https://driivz.com/blog/ev-charging-standards-and-protocols/, 2022.

[23] GreenFlux, *Learn about charging protocols ocpi, ocpp and oscp*, https://www.greenflux.com/spotlights/open-protocols//, 2021.

[24] I. O. for Standardization, "Iso 15118-3:2015 road vehicles — vehicle to grid communication interface — part 3: Physical and data link layer requirements," 2015.

[25] OpenECU, *Interface evse with combined charging system (ccs) using openecu™ m560 or m580*, https://openecu.com/case_study/interface-evse-with-combined-charging-system-ccs-using-openecu-m560-or-m580/, 2020.

[26] ——, *Power-line communication (plc)*, https://openecu.com/power-line-communication-plc/, 2020.

[27] R. Pallander, *Implementation of homeplug green phy standard (iso15118) into electric vehicle supply equipment*, 2021.

[28] L. Lampe, A. M. Tonello, and T. G. Swart, *Power Line Communications: Principles, Standards and Applications from multimedia to smart grid.* John Wiley & Sons, 2016.

[29] O. C. Alliance, *Open charge point protocol 1.6*, 2015.

[30] A. Hoekstra, R. Bienert, A. Wargers, H. Singh, and P. Voskuilen, "Using openadr with ocpp," *Montréal, Canada*, 2016.

[31] O. C. Alliance, *Open charge point protocol 2.0*, 2020.

[32] M. van der Kam and R. Bekkers, "Comparative analysis of standardized protocols for ev roaming," *Report D6*, vol. 1, 2020.

[33] ——, "Mobility in the smart grid: Roaming protocols for ev charging," *IEEE Transactions on Smart Grid*, 2022.

[34] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, IEEE, 2020, pp. 150–153.

[35] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review," in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, IEEE, 2019, pp. 1–7.

[36] L. De Lauretis, "From monolithic architecture to microservices architecture," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, 2019, pp. 93–96.

[37] M. Kalske *et al.*, "Transforming monolithic architecture towards microservice architecture," 2018.

[38] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, IEEE, 2018, pp. 000 149–000 154.

[39] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[40] J. Yongguo, L. Qiang, Q. Changshuai, S. Jian, and L. Qianqian, "Message-oriented middleware: A review," in *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*, IEEE, 2019, pp. 88–97.

[41] D. Dossot, *RabbitMQ essentials.* Packt Publishing Ltd, 2014.

[42] J. Franklin, *Apache kafka use cases: When to use it and when not to*, https://www.upsolver.com/blog/apache-kafka-use-cases-when-to-use-not, 2022.

[43] N. Garg, *Apache kafka*. Packt Publishing Birmingham, UK, 2013.

[44] G. M. Roy, *RabbitMQ in depth*. Simon and Schuster, 2017.

[45] O. C. Alliance, "Ocpp 2.0: Part 2 - specification," *April*, 2018.

Appendix A

Mode3 Hardware Board Documentation

Structure of byte frames to send to board:
SOF:ADDRESS:CMD:LEN:DATA[LEN]:CRC:EOF

SOF → Start of frame (0x01)
ADDRESS →Board address, in case of mode3 it starts in 0xA1
CMD → Command to be executed
LEN → Size of the *DATA* field
DATA →Data field with size *LEN*
CRC → 8 bit Cyclic Redundancy Check
EOF → End of frame (0x04)

Figure 1: Frame byte structure

m3_change_state_cmd

CMD = 0x10
Used to start and stop charging, activate PWM phase and out-of-service

Request:
LEN = 0x01
DATA = m3_cmd

```
enum m3_command
{
	cmd_out_off_service,
	cmd_stop,
	cmd_start,
	cmd_start_b2,
};
```

cmd_out_off_service (0x00) → puts the board in an error state, making it impossible to begin charging
cmd_stop (0x01)→ ends an ongoing charging session
cmd_start (0x02) → starts charging
cmd_start_b2 (0x03) → activates PWM communication

SOF	ADDRESS	CMD	LEN	DATA[LEN]	CRC	EOF
0x01	0xA1	0x10	0x01	0x01	0xC6	0x04

Reply:
LEN = 0x01
DATA = cmd_result
cmd_result → 0x01 - command successful, 0x00 - not executed or error

SOF	ADDRESS	CMD	LEN	DATA[LEN]	CRC	EOF
0x01	0xA1	0x10	0x01	0x01	0xC6	0x04

Figure 2: Mode3 Change State CMD

m3_status_req_cmd

CMD = 0x11
Gets general status of board and its I/Os

Request:
LEN = 0x01
DATA = 0

SOF	ADDRESS	CMD	LEN	DATA[LEN]	CRC	EOF
0x01	0xA1	0x11	0x01	0x00	0x36	0x04

Reply:
LEN = 0x07
DATA = state:cp:pp:io_status:max_curr:pwm_curr:pp_curr
state → Current state according to IEC61851-1
cp → Tension of the control pilot (CP) pin, in Volts
pp → Resistance of the proximity pilot (PP) pin
io_status → Status of I/O (ups, epo, power_relay, lock, lock_feed), 1 bit each

bit0 – UPS
bit1 – EPO
bit2 – power_relay
bit3 – lock
bit4 – lock_feed
bit5-7 – unused

max_curr → Maximum current
pwm_curr → Current value limited by PWM
pp_curr → Current value limited by PP resistance

SOF	ADDRES S	CMD	LEN	DATA[LEN]							CRC	EOF
0x01	0xA1	0x11	0x07	0x01	0x0C	0x02	0x00	0x20	0x20	0x20	0x89	0x04

Figure 3: Mode3 Status Request CMD