POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

MASTER's Degree Thesis

Integration of an Open-Source Stack for Electric Vehicle Charging on Existing Infrastructures

Supervisors

Prof. Renato FERRERO

Dr. Eng. Ilario PITTAU

Candidate

Riccardo SCANU

MARCH 2026

Integration of an Open-Source Stack for Electric Vehicle Charging on Existing Infrastructures

Riccardo Scanu

Abstract

Starting from 2020, an exponential growth in the use of electric cars has been reported, exceeding 55 million in late 2024. This increase led to a rise in the demand for public charging stations, which in that period were less than 6 million and could not keep up with the increase in vehicles. To support the increase in electric vehicles (EVs), charging points should be scalable, compatible, and flexible for the upcoming technologies.

Many of the charging stations on the market use closed-source software. These solutions have many positive sides, such as vendor support and maintenance, reliability, integrated ecosystem, and optimization based on the hardware; on the other hand, they expose stakeholders and users to significant risks, including vendor lock-in, high implementation costs, limited interoperability, and vulnerability to service discontinuation, as demonstrated in recent cases.

In this thesis, I integrated an open source software stack, based on EVerest, with custom hardware designed for charging EVs. Open-source software such as the Open Charge Point Protocol (OCPP) and the ISO15118 protocol offer the possibility of having a highly maintainable and compatible environment, while reducing the costs of implementation and having better accessibility by letting developers study and modify the source code as needed.

The goal of this thesis is to demonstrate that it is possible to use open-source solutions for EV charging, explaining these approaches for future infrastructures.

To evaluate the efficiency of the software, I performed integration tests using electric vehicle charging simulators. These simulators provide the possibility of emulating electric vehicle behavior in a charging scenario, from protocol compliance to error handling. Although more testing with real electric vehicles is still required to establish usability in a real-world scenario, the results highlighted the feasibility of integrating EVerest into custom hardware.

In conclusion, the thesis ends with a consideration of how open-source implementation can become a standard and still be valid across different types of electric vehicle.

to my parents

Table of Contents

1 Introduction **1**
1.1 Problem Analysis 1
1.1.1 Problem Context 2
1.1.2 Hardware Analysis 3
1.2 Goals of the thesis 5
1.2.1 Objectives 5
1.2.2 Scope 5
1.3 Thesis Structure 6

2 Literature Review and Background **8**
2.1 EV Charging 8
2.1.1 AC vs. DC charging 8
2.1.1.1 Charging Levels 9
2.1.2 Charging Components 10
2.2 EV Charging Systems 13
2.2.1 Communication Protocols 14
2.2.2 Software Architectures 20
2.3 Related work on open-source EV charging frameworks 22
2.3.1 Work related to open-source frameworks 22
2.3.2 A literature review on EVerest adaptations 23
2.4 From Challenges to Implementation Choice 24

3 System Design and Requirements **26**
3.1 System Overview 26
3.2 Design Objectives 27
3.3 Hardware Requirements 28
3.4 Software Framework 30
3.5 Communication Protocol Considerations 33
3.6 Software Requirements 35
3.7 System Architecture 36
3.8 Design Challenges and Trade-offs 37

4 Methodology **39**
4.1 Hardware Configuration 39

4.1.1 Hardware Overview 39
4.1.2 Hardware Setup 40
4.2 Software Development 42
4.2.1 EVerest Stack Integration 42
4.2.2 Custom Software Components 45
4.2.2.1 D-Bus Library Implementation 46
4.2.3 From *Manifest* to *Module* 49
4.2.3.1 Module for MCU Driver 50
4.2.3.2 Module for RFID Driver 54
4.2.4 Research of Testing Tools 56
4.2.4.1 OCPP server-side Test Environments 59
4.2.4.1.1 SteVe CSMS 59
SteVe CSMS 59
4.2.4.1.2 MaeVe CSMS 62
MaeVe CSMS 62
4.3 Software Used 64
4.3.1 Software Development Kit 65
4.3.2 Other Useful Tools 67
4.3.2.1 Wireshark 68
4.3.2.2 Exi-decoder 69

5 Results **72**
5.1 Test Environment 72
5.2 Validation Workflow 74
5.3 Testing Activities During Development 76
5.4 Limitation of the Test Environment 77
5.5 Discussion of Results 78
5.5.1 Validation of Custom Modules 78
5.5.2 SLAC Communication 79
5.5.3 RFID Validation 80
5.5.4 Start and Stop Charging 81
5.5.4.1 Start Routine 82
5.5.4.2 Stop Routine 83
5.5.5 Performance Validation 84

6 Conclusion **87**
6.1 Maturity of Open-source EVSE 87
6.2 Future Research Directions 91
6.3 Final Considerations 92

A ISO 15118 **94**
A.1 Communication States 94
A.2 Message Sequences 94
A.3 X.509 Certificates 97

B Software Code Snippets **100**
B.1 Module Configuration File 100
B.2 System Configuration File 101
B.3 OpenAMP Proxy & Adaptor 103

C SLAC Protocol **105**
C.1 SLAC Sequence Chart 105

Bibliography **108**

Dedications **111**

List of Figures

1.1 EV Connector types 5

2.1 AC vs DC overview 9
2.2 EVSE Components 11
2.3 Vehicle-to-Grid Schema 13
2.4 EV Charging System 14
2.5 OCPP Charging Flow 17
2.6 Smart Grid Illustration 18
2.7 EVerest high level overview 22

3.1 EVSE System Overview 27
3.2 STMP32MP151 Overview 30
3.3 EVerest Main Modules 31
3.4 Modules Communication 32
3.5 System Block Diagram 33
3.6 TLS Handshake Certificate Exchange 34
3.7 Possible System Architecture 37

4.1 EVSE Enclosure 41
4.2 Configured EVSE System 45
4.3 Emulator UI from Web Application 57
4.4 TLS Raw Message 58
4.5 Emulator CCS UI from Web Application 58
4.6 CSMS and EVSE Overview 59
4.7 SteVe UI Home Page 61
4.8 SteVe Add RFID Tag Form 62
4.9 Wireshark UI 69
4.10 Wireshark Raw Message 69
4.11 Wireshark Follow TCP Stream 71

5.1 Test Environment Overview 74
5.2 High-level validation workflow adopted for testing the customized EVerest EVSE implementation. 75
5.3 Slac Messages From Wireshark 80

5.4 Initialization Time for different numbers of modules, showing individual runs and averages with standard deviation. 85

6.1 Proprietary Ex-Novo 89
6.2 EVerest Customization 89

A.1 DC Charging Session 96
A.2 AC Charging Session 97

C.1 SLAC Sequence Chart 107

List of Tables

1.1 Overview of EV charging connectors 4

2.1 Overview of EV charging connectors 16
2.2 ISO 15118-2 and ISO 15118 Comparison 19

3.1 Software Requirements Overview 36

5.1 Summary of Performance Metrics for Key EVSE Sub-Processes (Average ± Std. Dev.) 86

A.1 ISO 15118 Communication States 94

C.1 SLAC Message Exchange Overview 106

Acronyms

EV Electric Vehicles.

EVSE Electric Vehicle Supply Equipment.

OBC On-board Charger.

CPO Charging Point Operator.

EMS Energy Management System.

EMSP Energy Management System Provider.

RTOS Real-Time Operating System.

OS Operating System.

MP Microprocessor.

EVCC Electric Vehicle Charge Controller.

SECC Supply Equipment Communication Controller.

OCPP Open Charge Point Protocol.

CP Control Pilot.

LLC Low-Level Communication.

HLC High-Level Communication.

MCU Micro-controller Unit.

ECU Electronic Control Unit.

SDK Software Development Kit.

ISO International Organization for Standardization.

PnC Plug & Charge.

SLAC Signal Level Attenuation Characterization.

PLC Power Line Communication.

ETH Ethernet.

LTE Long Term Evolution.

CAN Controller Area Network.

V2G Vehicle-to-Grid.

V2L Vehicle-to-Load.

V2V Vehicle-to-Vehicle.

TCP Transmission Control Protocol.

IP Internet Protocol.

TLS Transport Layer Security.

MQTT Message Queuing Telemetry Transport.

YP Yocto Project.

CSMS Charging Station Management System.

AC Alternating Current.

DC Direct Current.

RFID Radio-Frequency Identification.

PWM Pulse Width Modulation.

REQ Request.

CNF Confirmation.

RES Response.

OCSP Online Certificate Status Protocol.

EIM External Identification Means.

HMI Human-Machine Interface.

IPCC Inter Processor Communication Controller.

HSEM Hardware Semaphore.

HPGP HomePlug GreenPHY.

SPI Serial Peripheral Interface.

UID Unique Identifier.

CMD Command.

VAR Variable.

PKI Public Key Infrastructure.

CA Certificate Authority.

D-Bus Desktop Bus.

IPC Inter-Process Communication.

CCS Combined Charging System.

API Application Programming Interface.

EaC Environment as Code.

IDE Integrated Development Environment.

DIY Do-It-Yourself.

Chapter 1

Introduction

1.1 Problem Analysis

Over the past decade, the world of transportation has begun to shift towards more sustainable solutions, driven by the willing to reduce gas emissions and adopt more green solutions. Electric vehicles (EVs) are the center point of this transition. The increasing availability of this vehicles type led to a rapid rise in usage share worldwide. In Italy, incentives has been offered to purchase EVs, in order to increase the usage.

Starting from 2020, the share of vehicles increased from 10 million to 55 million by 2024 [1]. The International Energy Agency predicts that the total number of global electric vehicles will reach 250 million in 2030 and 525 million in 2035 [2]. his growth has resulted in a rising demand for public charging infrastructure worldwide. In 2024, the number of public charging points remained below 6 million [3], a figure that was insufficient to keep pace with the rapid expansion of the EV fleet. The difference between the number of vehicles and charging infrastructures shows the need to have more charging points, while also being compatible and flexible to adapt to different EVs.

Many of the public stations implement closed-source software, which was developed ad hoc for their hardware. This means that not all charging points are compatible with every EV, but several key factors must be evaluated, including connector type, charging protocol, and charging power.

In this topic, open-source software solutions are promising for future implementation, such as EVerest [4], Electric Vehicle Supply Equipment (EVSE) from Pazzk-labs [5], and OpenEVSE [6]. Those solutions usually work with different connector types, levels, and communication protocols like OCPP and ISO 15118, with different versions that are constantly being updated and developed.

For these reasons, the idea for this thesis was proposed by *Abinsula* [7], an Italian technology company specialized in designing and developing customized software and

embedded solutions across multiple domains, including automotive, Internet of Things (IoT), web and mobile applications, cybersecurity and cloud platforms. Founded in 2012, the company has established itself as a provider of innovative information and communication technology solutions, with a strong focus on automotive and embedded systems. Abinsula operates internationally, with multiple offices in Italy and abroad. Its expertise in integrating hardware and software for complex systems makes it particularly suited to supporting research and development in areas that require interoperability and multi-protocol communication frameworks.

The work presented in this thesis was carried out within the company, with the objective of integrating and developing an open-source software stack for EV charging, building upon existing open-source solutions, and evaluating their maturity and suitability for real-world industrial applications.

1.1.1 Problem Context

The software is not the only part involved in the EV charging environment, other roles are: Charging Point Operators (CPOs), eMobility Service Providers (EMSPs), energy utilities, and vehicle manufacturers.

CPOs are responsible for building, installing, and maintaining electric vehicle charging stations. They manage backend technologies to ensure reliable charging operations and serve as the bridge between charging infrastructure and e-mobility service providers, who engage directly with EV drivers.

EMSPs they are the final link between CPOs and companies, and play a major role thanks to their private or public roaming charging services; allowing end users to charge their vehicles without having a direct contract for each network present in the area.

Energy utilities are responsible for electricity generation, distribution, and grid management. In the context of EV charging, they play a key role in load balancing, energy monitoring, and the integration of charging infrastructure into the electrical grid, especially as charging demand increases.

Vehicle manufacturers design and implement the on-board charging systems and communication interfaces of electric vehicles. They define how vehicles interact with charging stations, including supported connectors, charging modes, and communication protocols.

Each of them could decide to work with different protocols and standards. While many are working to integrate open-source solutions, it is still rare to find ones that

integrate all of them.

The EVSE is more than just a power supply device, it is also a communication endpoint that exchanges information with the vehicle and a back-end service. The communication between EVSE and EV is used as a handshake to select the correct parameters and protocols of the charging process; while the information exchanged with the back-end is useful to authenticate the user, handle payment, monitor session, and perform diagnostic.

These stakeholders operate at different layers of the EV charging architecture, and because the interfaces between them must be standardized and regulated, multiple communication protocols coexist for each interaction. The communication between EVSE and EV relies on standards like ISO 15118-2 [8], ISO 15118-20 [9], IEC 61851 [10], and DIN 70121 [11]; each of them can implement different features like: connection to grid, and plug-and-charge; and define different charging modes, control signaling.

On the other hand, the communication between the EVSE and the backend services can be handled with protocols like Open Charge Point Protocol (OCPP) in its versions 1.6, 2.0.1, and 2.1 which are explained by *Open Charge Alliance* in their website page OCPP [12], and IEC 63110 in its versions 1, 2, and 3 [13]. These protocols are used for station management, authentication, and monitoring of the charging process.

Given the presence of these protocols and their versions, this thesis evaluates the maturity of the chosen open-source software, complemented by custom developed sections, with the goal of supporting multiple protocols within the same EV charging system.

1.1.2 Hardware Analysis

In the EV charging environment, different communication protocols and hardware configurations coexist depending on system requirements. Electrical power can be delivered either in Alternating Current (AC) or Direct Current (DC) mode. In AC charging, the power conversion from AC to DC occurs inside the vehicle through the on-board charger, which typically limits the charging power and results in longer charging times. In contrast, DC charging performs the power conversion within the charging station itself, enabling significantly higher power levels and faster energy transfer to the vehicle battery. In addition, various connector standards are used worldwide, categorized according to supported power levels and charging modes, as summarized in Table 1.1 and illustrated in Figure 1.1.

From a system perspective, charging stations may also integrate additional hardware and software modules to enable extended functionalities and service interoperability.

Stations integrate hardware in order to let internal components communicate, those can be controllers and network modules such as Power Line Communication (PLC), Ethernet (ETH), Long Term Evolution (LTE), (CAN), which enable data exchange between components and with external services. Although they play similar roles, they operate at different layers within communication architectures.

At the control layer, the Microcontroller Unit (MCU) plays a fundamental role in task management, signal processing, protocol handling, and hardware control. Given the real-time constraints and limited resource requirements of embedded systems, lightweight Real-Time Operating Systems (RTOSs) are typically adopted instead of general-purpose operating systems.

In addition to communication and control, EVSEs rely on power conversion and conditioning components to deliver the correct voltage and current to the EV, while maintaining efficiency and compliance with safety standards. Protection circuits, such as fuses, circuit breakers, and residual current devices, safeguard both the EVSE and the vehicle during operation. Together, these hardware layers support reliable and safe charging for different vehicles and protocols.

Connector	**Region/ Standard**	**Power (kW)**	**AC/DC**
Type 1 (SAE J1772)	North America, Japan	up to 7.4	AC
Type 2 (Mennekes)	Europe (EU standard)	up to 22 (AC), 43 (3-phase)	AC
CCS (Combo 1/2)	Global (NA = Combo 1, EU = Combo 2)	up to 350	DC
CHAdeMO	Japan, some EU markets	up to 200	DC
Tesla (NACS)	North America (Tesla)	up to 250	AC/DC
GB/T	China	up to 237	AC/DC

Table 1.1: Overview of EV charging connectors

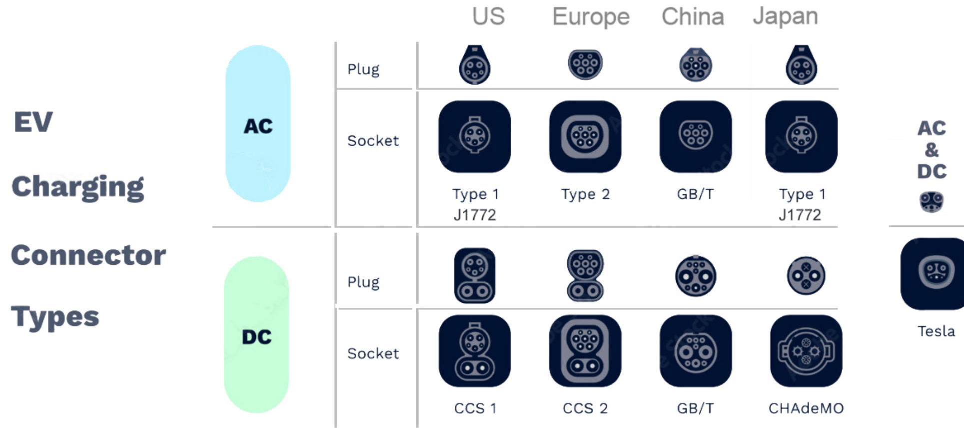


Figure 1.1: EV Connector types

1.2 Goals of the thesis

1.2.1 Objectives

The main objective of the thesis is to showcase the maturity and potential of open-source software solutions within the EV charging ecosystem. The purpose of the thesis is to showcase the potential impact on developing flexible, interoperable, and scalable infrastructures.
In particular, it aims to:

- Demonstrate the integration of open-source charging software with custom hardware, ensuring compatibility and smooth communication within the charging station;

- Showcase the horizontal scalability of the software that can lead to creating a custom stack by supporting additional modules without breaking the system;

- Evaluate the possibility of running multiple modules concurrently with different protocols, with the opportunity to develop custom modules and interfaces for the software, in order to cover cases that are not reached by the default ones.

1.2.2 Scope

The thesis covers the integration of an open-source stack in the context of the charging station control system, rather than designing a completely new stack starting from an existing one.

This work focuses on AC mode, imposed by the provided hardware, since the charging station used for this thesis is equipped with a Type 2 connector to comply with the EU standard and can provide up to 22 kW in AC mode and 43 in

3-phase, without the possibility of implementation of DC charging. Since no DC power modules were available, DC fast charging and other configurations could not be implemented or tested.

The ISO 15118-2 and DIN SPEC 70121 protocols are fully supported in the work conducted for this thesis, as are the OCPP communication protocols in its versions 1.6 and 2.0.1, which were tested against some open-source services. Although ISO 15118-20 could not be included because it was not available for AC mode and integration with a real CPO and EMSP backend was not performed, the work demonstrates a fully functional and extensible software stack for AC charging with multiple supported protocols.

Horizontal scalability was partially explored in terms of supporting multiple modules and charging stations in parallel since it could not be tested on the actual hardware, while tests for a single charging point were conducted and fully explored for each scenario. Large-scale tests and stress tests could also not be evaluated and are outside of the scope of this thesis.

However, to demonstrate the flexibility of the chosen open-source charging framework, custom modules were implemented for the hardware to showcase its extensibility, without introducing additional communication protocols beyond those already supported.

By defining these boundaries, the thesis narrows down the study to the integration and maturity evaluation of open-source software for AC charging scenarios while highlighting both its strengths and current limitations.

1.3 Thesis Structure

In the following chapters, the process from theory to practical implementation will be explained. The structure is showcased below:

- **Chapter 2 - Literature Review and Background**: This chapter starts by explaining different types and levels of charging, then goes on to give an overview of what an EVSE is and explains the internal components of both the charging station and the EV. It then discusses commonly used communication protocols and software architectures. Finally, it gives a brief idea of the challenges and limitations that could exist in the software.

- **Chapter 3 - System Design and Requirements**: The possible system overview and design are explored at the beginning of this chapter. Based on the objectives, the hardware requirements are listed as for the software ones. Some

limitations are also taken into account to choose the communication protocol. In the end, some design challenges and trade-offs are discussed to set the final designs.

- **Chapter 4 - Methodology**: The hardware configuration and setup, as well as the integration of an open-source stack, are detailed, including how custom modules were developed. It also presents the software development kit used to compile and build the software for the board, together with the simulation tools and framework used to perform tests.

- **Chapter 5 - Results**: In this section, results in different configurations are provided to showcase the performance of the resulting system. A brief description of the environment with its outcomes is provided to better explain the use cases. Finally, the results obtained are compared to the goals of the thesis.

- **Chapter 6 - Conclusion**: Taking into account the results obtained and discussed in the previous chapter in this one, some considerations are evaluated based on the objectives and scope of the thesis. In addition, some future research directions are explored to escape the limits of this work.

Chapter 2

Literature Review and Background

2.1 EV Charging

Every vehicle that has a battery requires an EV charger to keep it charged. It is an integral part of the EV industry and is generally in parallel with the development of electric vehicles. It is the process of supplying electricity to the battery to keep it powered and ready for the road.

EV batteries usually require DC power, but most charging stations supply AC. So, a component called On-Board Charger (OBC) comes in. This device converts AC to DC, ensuring that the battery stores the energy automatically and efficiently.

EV charging stations pull electricity from the power grid, use the parameters they got from the vehicle, and then transfer energy to it. Furthermore, there are different techniques to supply and transfer energy, for example smart charging, where the power delivery is based on demand and availability.

2.1.1 AC vs. DC charging

There are two different types of electric current, AC and DC. There are several differences between the two, their sources and how they are handled when charging vehicles. Although both can be used to charge the battery of the vehicle, there is a difference in where the current is converted to be stored in the battery that allows only DC. That is where OBC comes into play. When charging vehicles using AC charging stations the OBC converts the current before it enters the battery, while with DC this is not necessary since there is a converted placed inside the charging point.

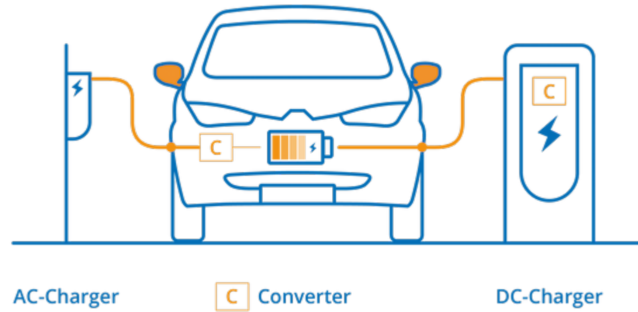


Figure 2.1: AC vs DC overview

AC charging can be slower due to the conversion that needs to be applied to make it DC. That is why it is more commonly found at home or in private spaces. In public spaces, DC is chosen over AC because the charging parking space needs to be rapidly freed so that other drivers can charge their vehicle.

The DC charging infrastructures are generally larger than the AC stations, as illustrated in Figure 2.1 from go-e [14], since the converter is inside it and does not use the one already built in. Therefore, the cost of the station is higher compared to AC. Furthermore, DC charging stations require a much higher input of more than 480 volts, which is not very common.

Another main difference between the two is the speed. Since DC charging bypasses the OBC, it goes straight into the battery, making this process in many cases faster, but it also depends on other key factors such as the EV model and the charger used. If this type of charge is used continuously, the performance and durability of the batteries may be affected.

Although both AC and DC mode are valid solutions for EV charging, AC charging remains the most prevalent in the ecosystem, for example, the number of AC charging stations in Europe was 550 thousand out of 632 thousand stations in late 2023 [15]. The reasons behind this popularity are many, starting from the lower price compared to the DC mode, the simpler infrastructure with fewer components, and the compatibility with pre-existing electric grids. Another reason why it is so popular is that EV users usually have a charger at home that in many cases does not support a DC power supply.

2.1.1.1 Charging Levels

There are three different levels of EV charging systems, that are Level 1 (120V), Level 2 (240V), and Level 3 (DC Fast Charging).

- **Level 1**: is the easiest and slowest method, since it only requires a household socket to charge the vehicle;

- **Level 2**: AC charging stations are used to charge an EV. Considering the low price and moderate charging speed, they are preferred over the default socket. They can be found in the public and can also be installed at home;

- **Level 3**: it is also known as DC Fast Charging. The unit uses DC and, because of this, is the fastest type of EV charging available. It requires a different cable and has a higher price per energy supply.

For this project, the focus was on Level 2 since it involves AC and is also the most used among all the different levels, providing a relevant platform to evaluate the integration of open-source software for this type of infrastructure.

2.1.2 Charging Components

The charging system includes a set of components that are responsible for handling the charging process: *Electric Vehicle Supply Equipment*, *On-Board Charger*, and *Grid Connection and Power Electronics.*

- **Electric Vehicle Supply Equipment** The EVSE manages the transfer of current from the local power supply to an EV, while keeping the vehicle safe by managing protection, communication, and proper energy delivery.

  From an external point of view, the EVSE is seen as a charger for our portable devices. In the big picture it is, but they are more complex than that. They can adapt simple or complex options; in both cases, referring to them as "EV chargers" simplifies their role.

  The main components for EVSEs are listed below:

  - **Housing or Enclosure**: is the shell that contains all the physical components of the EVSE;

  - **Electronics inside the housing**: all the circuits that control the power supply, and optionally the Human-Machine Interface (HMI);

– **Firmware**: read-only code that enables the components to operate, includes the code to communicate with the EV and to handle the HMI;

– **Network connectivity**: network module to communicate with back-end services and/ or mobile application;

– **Power connection**: EVSE can be directly connected to a household socket or directly to the electric system of a building, depending on the level of charge;

– **Port(s) on the housing and cable(s)**: ports or fixed cables provided by the EVSE to connect the EV;

– **Connectors on the cable that plug into the EV**: depending on the charging level, different cables are used.

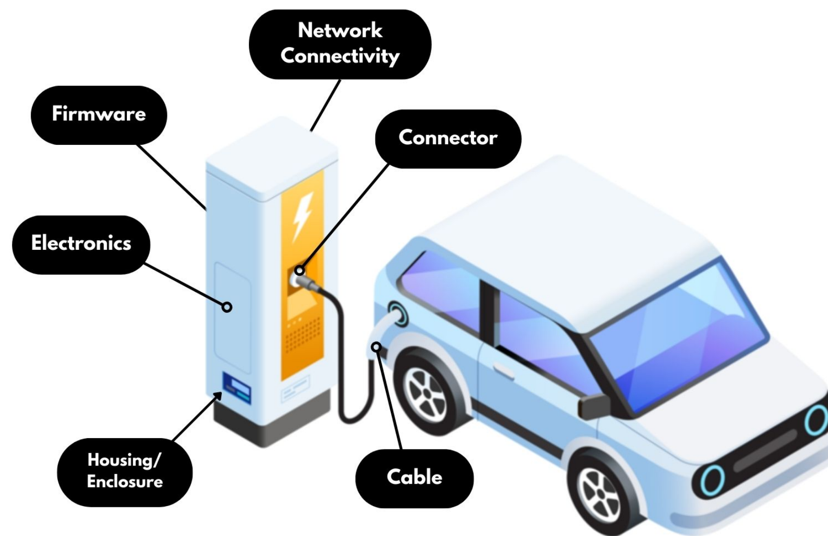


Figure 2.2: EVSE Components

In addition to these components, public infrastructures must include a payment system to let users pay for the service; also, they usually provide some sort of HMI to let the user interact with the EVSE. RFID cards or mobile applications are usually the choice to have a valid HMI.

- **On-board Charger** When the EV is charged in AC mode, the OBC is used to convert the power supply into DC to store it in the vehicle's battery. Before the charging process begins, the OBC together with the EV's communication

controller monitors the communication for parameters exchanged with the EVSE through the Control Pilot (CP) signal. The main parameter that gets exchanged is the Pulse Width Modulation (PWM) duty cycle value. Once the EVSE sets the duty cycle, the EV detects it and instructs the OBC to change the PWM value, enabling data exchange between the EV and the EVSE through the ISO 15118 protocol, which provides a standardized, secure, and extensible framework for exchanging charging parameters, authorization information, and energy management messages between EV and EVSE. Other protocols, such as DIN 70121 or proprietary OEM protocols, can also be used depending on vehicle and infrastructure support.

This step is essential for the charging process. Once ISO 15118 data exchange begins, OBC starts monitoring the status of its sensors and sends them to the vehicle's control unit (ECU), which will forward them to the EVSE, through the PLC. Those data are crucial to maintain safety during the charging process and adjust the charging parameters to optimize battery recharge.

- **Grid Connection & Power Electronics** The connection between the EVSE and the electric network of a city is called the "Grid Connection". The type of connection is fundamental for choosing the hardware of the EVSE. The electric components of the station should also be able to adapt to the limits of the EV in order to not supply more than what the vehicle can handle.

  Depending on the grid, charging stations may draw power using a single-phase or three-phase connection. The first is more common in residential areas, it is slower than the three-phase connection and therefore more common for slower charging sessions. On the other hand, three-phase is wide spread in industrial area since it can supply higher current at a faster pace, this can lead to an elevated price.

  Single-phase and Three-phase apply only to AC systems, since they describe sinusoidal waveforms of alternate current. On the other hand, DC provides a flat flow of current that corresponds to the maximum value, for the applied voltage, of the current that could be supplied.

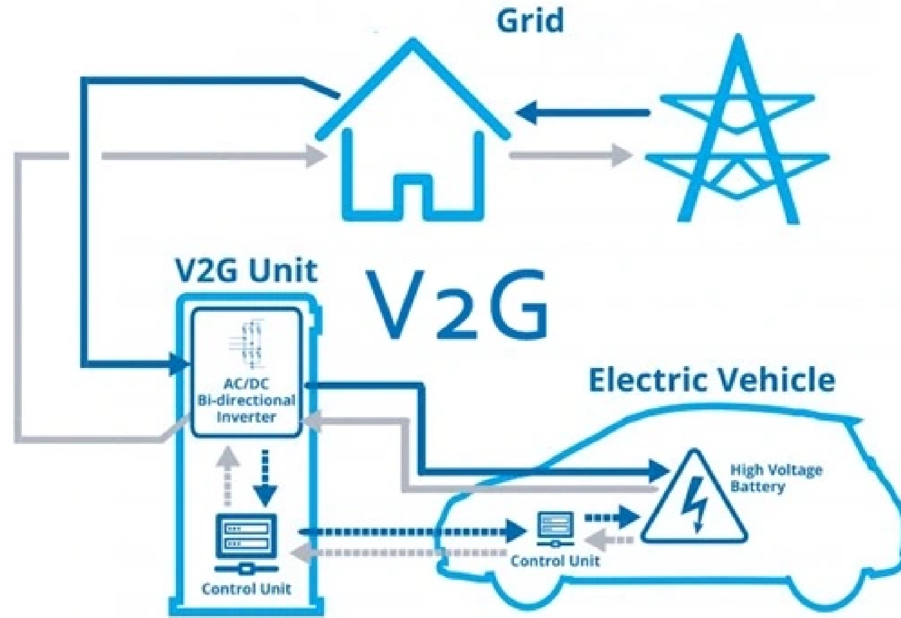


Figure 2.3: Vehicle-to-Grid Schema

In addition to the Grid Connection, there are power electronics components that are used for conversion of current, inverters for bidirectional flow, protection, and load balancing. Modern EVs have the ability to interact with the grid using inverters, enabling features such as Vehicle-to-Grid (V2G) exchange. When enabled, this feature allows the EV to send electricity back to the power grid as shown in the Figure 2.3 [16]. It is used mainly when the grid has a high demand and the EV supports it by discharging its battery. Related concepts of V2G that are mainly used in emergency cases are Vehicle-to-Load (V2L) and Vehicle-to-Vehicle (V2V), which allow the EV supply power directly to external devices or vehicles. Although they share similarities, unlike V2G they do not need an inverter with the utility grid, so the power is only available to "off-grid" loads.

2.2 EV Charging Systems

Besides EVSE and Grid Connection, but there are other roles that take part in the EV Charging System, as seen in Figure 2.4. The CPO and the e-Mobility Service Provider (EMSP), are fundamental to finalize the charging session.

The CPO is responsible for connecting the EVSE to back-end services, through communication protocols such as OCPP. The CPO can authorize or deny charging requests, station status, perform diagnostics, and handle pricing.

The EMSP is responsible for the interaction with the user, either through a mobile application or an RFID card. To authenticate, a user can use the mobile

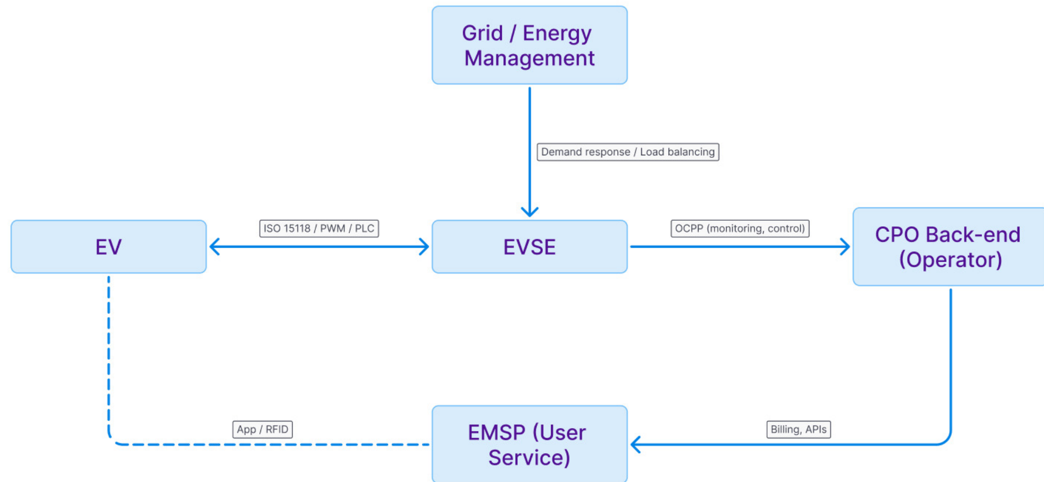


Figure 2.4: EV Charging System

application, the RFID card, or the Plug & Charge (PnC) feature of the ISO 15118 protocol. The authentication request is processed, and the authorization is passed to the CPO.

The two roles can be summarized as follows: the CPO handles the physical infrastructure, while the EMSP handles the customer services. Together, CPO and EMSP ensure that the EV charging system works smoothly.

2.2.1 Communication Protocols

In the EV Charging System, there are several components that must communicate in order to make recharge possible. Moreover, they do not reside at the same communication level, so different protocols must be applied. Although several protocols have been developed to standardize communication for each level, this thesis focuses on the most common ones in AC charging systems, which are also open-source.

- **Open Charge Point Protocol (OCPP)** The Open Charge Point Protocol, in addition to being a communication protocol, was also developed to facilitate compatibility between different charging stations and back-end systems. OCPP provides several services to the EVSE, for example, to start and stop charging sessions, retrieve charging data, update firmware, and manage user authentication.

  OCPP being open-source, meaning that it is not proprietary and can be freely implemented by manufacturers and developers, ensures that hardware from different brands can still communicate with each other and with the back-end system. This interoperability reduces the barriers between components and

promotes flexibility and compatibility, supporting the widespread adoption of open-source EV charging systems.

The continuous development of OCPP led to the existence of three versions of OCPP mainly used on the market, OCPP 1.6, OCPP 2.0.1, and OCPP 2.1. Once the version is stable, the shift starts towards that version. As illustrated in Table 2.1 [17], OCPP's continuous development is bringing new features, such as bidirectional charging, and is driving the industry toward newer implementations. The progression highlights improvements in newer versions that are also compatible with newer features.

	OCPP 1.6	**OCPP 2.0.1**	**OCPP 2.1**
Release	2015	2020	2025
Security	Basic TLS encryption, no strict authentication	Stronger encryption, certificate-based authentication, security profiles	Advanced encryption, enhanced security policies, more flexible authentication
Smart Charging	Load balancing, static charging profiles	Dynamic smart charging, flexible schedules, energy optimization	V2G support, enhanced grid interaction
ISO 15118 (PnC)	Not supported	Supported with improved handling of certificates	Fully supports ISO 15118-20 with bidirectional charging
Interoperability	Medium, vendor-specific implementations	High, with structured messaging	Very high, supports modular components and renewable energy integration
Device Management	Limited station monitoring & control	Device Model introduced for remote diagnostics	Enhanced diagnostics, real-time telemetry, improved component handling

UI & User Experience	Basic session handling	On-screen messages, multi-language support	Advanced notifications, real-time cost tracking, improved reservation system
Billing & Payments	Simple kWh-based billing	Flexible transaction models (kWh, time-based, fixed cost)	More advanced session management, resume transactions after interruptions
Reservations	Basic reservation functionality	Improved reservation for fleet and network operators	Enhanced pre-booking & dynamic scheduling
Market Adoption	Widely used	Growing adoption	Recently released, gaining initial industry support

Table 2.1: Overview of EV charging connectors

When OCPP is being used to charge an EV the charging sequence can be summarized in seven steps being:

- Reservation: usually through a mobile application;
- Charger Hold: charger is reserved by the back-end system and physical station;
- Authorization: user identify itself via RFID or another method;
- Charging: if the authorization is successful the charging process starts;
- Notification: user receives information about the status of charge;
- Charging Done: user disconnects the EV from the EVSE;
- Billing: back-end send an invoice to the user through the EMSP.

- **ISO 15118** Road Vehicles - Vehicle to Grid Communication Interface, or ISO 15118 is a proposed international standard for bidirectional and PnC for charging/ discharging EVs.

  I truly believe that ISO 15118 is one of the most important and future-proof standards available today, said Marc Mültin in his article for Switch [18].

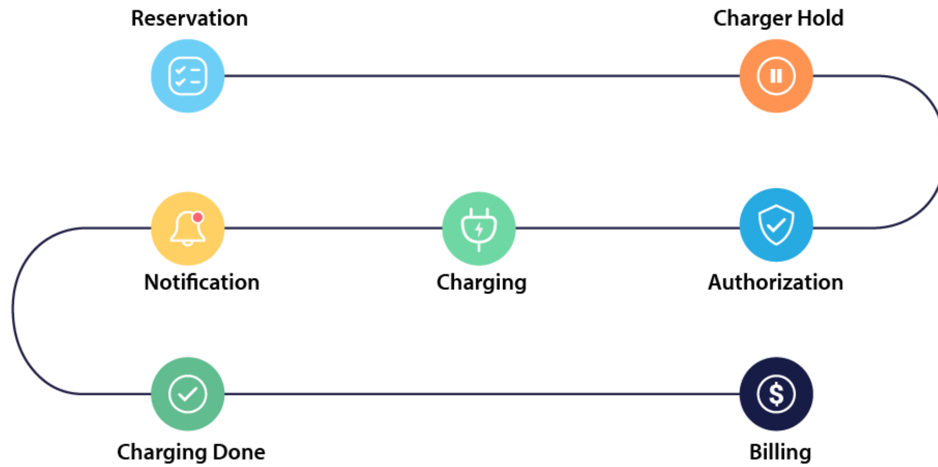


Figure 2.5: OCPP Charging Flow

The protocol defines the means of communication between EV and EVSE, and their common language by establishing message structure. Without ISO 15118, components would send data in different formats following different rules. Many operators in the EV market agree that protocols such as ISO 15118 are essential to shift towards more acceptance of electric mobility.

ISO 15118 allows PnC and bidirectional recharge. The first allows the vehicle to authenticate automatically without the need for a mobile application or an RFID card to initiate the charging process. The second is also known as V2G, and enables the integration of EVs into the Smart Grid. Smart Grid is an electrical grid that interconnects producers, consumers, and electrical components, to enable digital communication between each other, as illustrated in Figure 2.6. It is important that the Smart Grid supports the charging of multiple vehicles in parallel without overloading. The grid calculates a schedule for the connected vehicle using information received from all connected devices and also by the driver's needs.

Within the Smart Grid, large amounts of sensitive information, such as billing information, are continuously exchanged between EVs and EVSE. Those sensitive data are a potential target for malicious entities. ISO 15118 introduced the feature PnC, which exploits cryptographic mechanisms to guarantee secure communication, confidentiality, integrity, and authenticity of all data exchanged. During the connection stage, digital certificates (X.509 certificates Appendix A.3) and public-keys are exchanged between the EV and the EVSE to gain authorized access to recharge the vehicle battery. It is clear that in the future, mobile applications or RFID cards will no longer be needed.

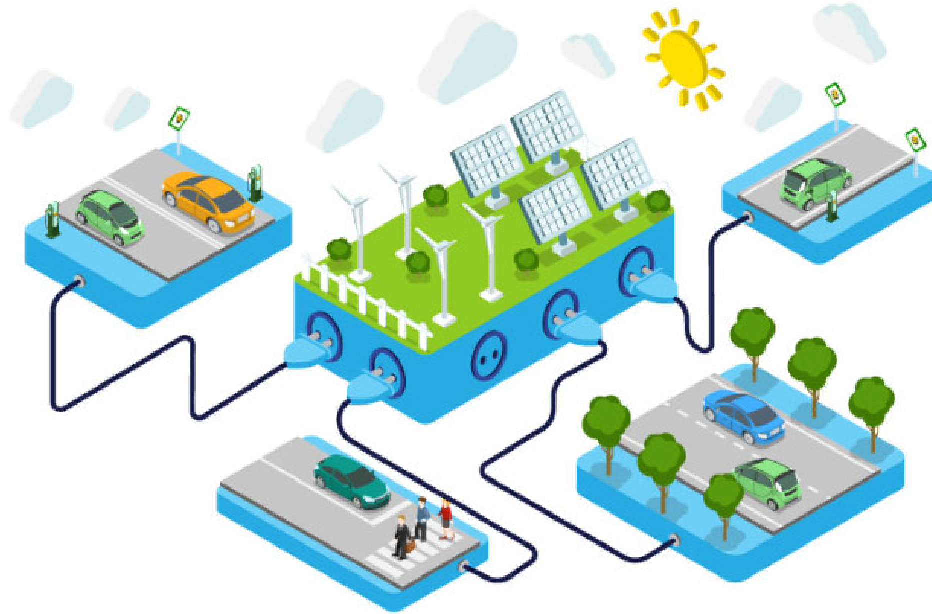


Figure 2.6: Smart Grid Illustration

The ISO 15118 developers are also looking towards the future with features like wireless recharge. However, the EV market is not ready for this standard. The majority of EVs, and EV Charging Points do not support it yet.

Starting with the first ISO 15118 release in 2013, developers kept putting out new versions, most of them being specific to ISO/OSI layers. ISO 15118-2 and ISO 15118-20 are the most recognized. The first is known as the original widely adopted version that enabled PnC. The second is known to be for optimizing some aspects of version two while also implementing new key features for future development. Other versions are less familiar to the mainstream public since the operate on specific layers, especially in the physical one that handles the relationship with PLC, and the network for the requirements. In Table 2.2, some differences are reported.

Feature	**ISO 15118-2**	**ISO 15118-20**
Publication	2014	2022
Energy Transfer	AC and DC charging using charging cable	AC and DC charging using charging cable, Bidirectional energy transfer, Inductive charging, Automatic charging
Charging Control	Scheduled	Scheduled, Dynamic

Transmission Layer	HP GreenPhy, Conductive (ISO 15118-3)	HP GreenPhy, Conductive (ISO 15118-3), IEEE 802.11n (ISO 15118-8) (Wi-Fi 4)
Security	TLS 1.2 optional	TLS 1.3 mandatory

Table 2.2: ISO 15118-2 and ISO 15118 Comparison

To handle the charging process, the ISO 15118 protocol separated the actions into eight functional groups, where the first half are the most important, since they handle the X.509 certificates, settings, and charging process. Furthermore, the message flow is slightly different for the AC and DC charging modes and differs from version to version. For a detailed overview of the groups and message sequences, see Appendix A.

X.509 Certificates play an important role in the ISO 15118 protocol. They are checked during the Identification, Authentication sequence to be installed or updated when EV has soon-to-be expired contract certificates. In case of PnC these are mandatory and used as an identification method for the charging station to automatically authenticate and authorize the driver. In addition to PnC, certificates can be used to secure communication between EV and EVSE, to ensure confidentiality, integrity, and authenticity.

A trusted authority issues contract certificates that are stored in each EV. The role of the EVSE is to verify the certificates and uses its own SECC certificate to prove its authority to the EV and establish a secure connection. The EVSE can also provide a secure connection to the authority for the EV in case some of the certificates are missing or if they are soon-to-be expired.

- **Pulse Width Modulation (PWM)** To communicate basic charging information, the EV and EVSE communicate over the Control Pilot (CP) using Pulse Width Modulation (PWM). The EVSE specifies the maximum charging current for the EV via the duty cycle, the EV then initiates the ISO 15118 over PLC where other negotiations take place. For example, setting the duty cycle to 5% forces the transition from Low-Level Communication (LLC) to High-Level Communication (HLC). In this case, the digital communication starts. The signal voltages also indicate the communication state.

  There is a difference with AC and DC and what the PWM indicates in both cases. In AC mode, PWM is used both as a wake-up and maximum current for the EV; in DC mode, the signal is used primarily as a presence and compatibil-

ity, while the actual current is negotiated over the PLC.

In newer versions, the signal is used as a support for backward compatibility, since this signal was previously used in the protocol IEC 61851, while currently different techniques are used to negotiate parameters and start the communication between EVSE and EV.

- **Signal Level Attenuation Characteristics (SLAC)** The matching process between EV and EVSE shall be based on the messages defined by SLAC. All SLAC messages follow a request/ confirmation structure where the request ".req" from one side is always answered with a confirmation ".cnf" from the other side, as explained in Appendix C.

  The EV sends a broadcast request to all EVSEs. The first to respond sends a confirmation message to the EV. Only EVSEs that are connected to an EV, detected by a valid CP and are in "Unmatched" state can answer the broadcast message.

  To ensure that EV and EVSE are physically connected to each other, SLAC is used. Measures the strength of a signal between the EV and the EVSE. The EV listens and measures how much the signal is attenuated by the cable, and based on the results, they agree whether a reliable PLC channel can be established or not. This procedure is especially important in situations where multiple EVSEs are available. SLAC ensures that EV do not accidentally lock onto the wrong SECC. The attenuation measurements act like a "fingerprint" to match the two.

2.2.2 Software Architectures

Previous sections examined the hardware and software ecosystem for the EV charging infrastructure. To merge both parts, a solid software architecture is needed. A robust software architecture should be able to connect and manage communication between all components, from low-level to high-level devices.

Software architectures can be divided into two macro-categories depending on the implementation adopted:

- **Vertical (Proprietary or Siloed)**: This approach typically involves a closed ecosystem in which a single vendor provides the hardware, software, and related services. The features are fixed and there is limited flexibility to interchange software components or integrate third-party solutions. The advantages of

vertical systems are optimized performance and a secure system, with little or no knowledge of how it works. At the same time, these approaches are more expensive and more complex.

- **Horizontal (Modular or Layered)**: In this approach, software and hardware are separated into modules, allowing flexibility in implementation. Different vendors can provide components such as EVSE hardware, back-end services, or mobile applications while adopting the same open standards, such as OCPP and ISO 15118, to maintain compatibility. This design enables scaling up or down based on the problem requirements, allowing developers to add or remove features.

In recent years, several open-source frameworks have been developed adopting horizontal architecture in the EV charging domain, such as OpenEVSE [6], which provides AC charger solutions with *Do-It-Yourself* (DIY) kits, allowing users to assemble and configure their own charging stations from pre-designed hardware components, and evse from *Pazzk-labs* [5], a newer open-source software that offers modularity for custom implementation but with limited integration of communication protocols, and EVerest [4], a solid open-source implementation that offers support for both AC and DC mode while also adopting the major communication protocols for EV charging, among others. These solutions offer the possibility to develop custom EV charging software while providing default implementation with already tested hardware. Some are more developed than others, for example, by implementing more communication protocols or by being tested on different architectures and boards.

In particular, for the purpose of this thesis, EVerest was employed as a Horizontal Architecture approach. EVerest is an open source modular framework that lets developers configure their stack for EV charging based on interchangeable modules that communicate with each other through Message Queuing Telemetry Transport (MQTT). Moreover, EVerest was chosen over the other open-source frameworks due to the features implemented by it, the possibility to adapt different communication protocols, and having the opportunity to develop and test the custom stack within the EVerest ecosystem. In addition to the software, Pionix provides a Yocto Project (YP) layer, ISO 15118 library, and OCPP library to get a ready-to-go environment to work with. Other open-source solutions typically do not provide full compliance with these standards, nor do they offer implementations for multiple charging modes. Additionally, several of them are built for specific boards, such as esp32 or Arduino, while thanks to the YP image, EVerest, can be built for many different boards.

Pionix describes EVerest as *an operating system for EV chargers* [19]

EVerest modules are loosely coupled to adapt to different development scenarios.

Modules can be anything, from hardware drivers, protocols, and more. Figure 2.7 shows a simplified schematic of EVerest implementation, which highlights the relationships between modules, the MQTT broker, and the EVerest framework.

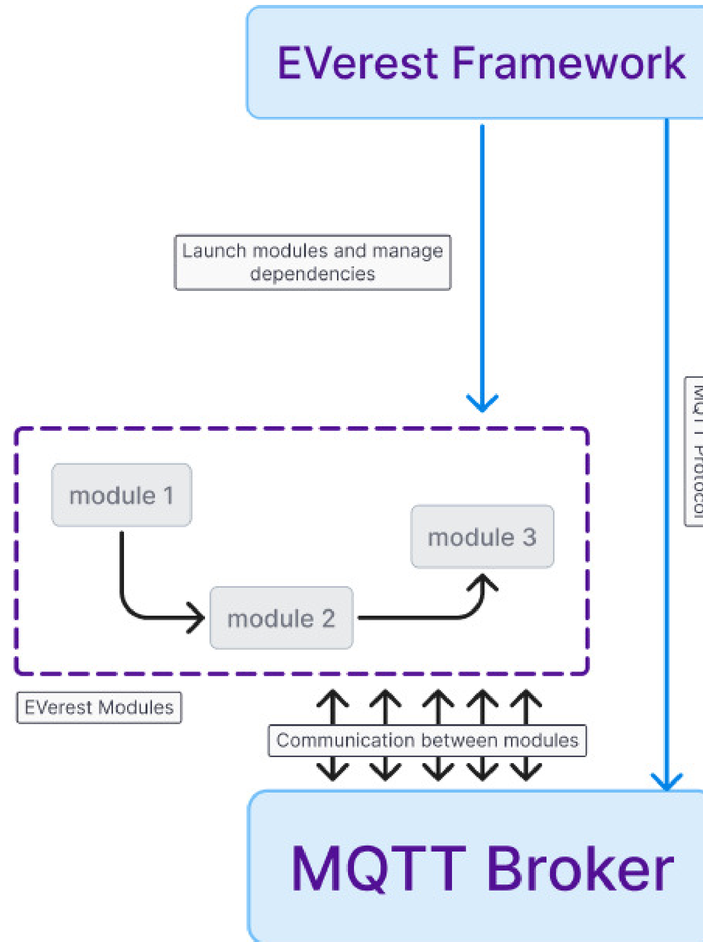


Figure 2.7: EVerest high level overview

2.3 Related work on open-source EV charging frameworks

2.3.1 Work related to open-source frameworks

In 2025, a research team conducted a study on effective and scalable OCPP security and privacy [20], this also covered one of the previously mentioned open-source frameworks OpenEVSE. Although OpenEVSE implements version 1.6 of OCPP, using EmuOCPP, a framework capable of emulating complex OCPP networks up to hundreds of hosts and diverse configurations, vulnerabilities were found even in versions 2.0.1 and 2.1, including Man-in-the-Middle (MitM), impersonation, and Denial-of-Service (DoS). The authors and their analysis demonstrated the need for open-source testing frameworks and provided insights into securing OCPP networks.

Another interesting paper by Kramžar Aleš [21] in 2020 shows the possibilities of altering OpenEVSE boards to implement new features such as DC charging and hardware-level customizations. Despite the successful customization of the hardware and software, the author points out some practical limitations in this work. At the time, the system's RFID handling was relatively basic, prone to secure authentication limitations, and the hardware used was not compliant with European electrical standards.

D'Agostino [22] in the thesis *Implementation of a PKI-based security communication and Value Added Service for EV charging using ISO 15118 standard* focused on providing a trusted message exchange between the EV and the EVSE. Secure communication enables some ISO 15118 features, such as PnC. The implementation of PKI mechanisms is common among the open-source solutions for EV charging, for example, EVerest, RISE-V2G, and OpenV2G implement certificates or similar to be compliant with ISO 15118 requirements.

2.3.2 A literature review on EVerest adaptations

Research on the EVerest framework can be classified into three main areas: security-oriented analysis, protocol compliance and interoperability validation, and hardware-focused implementation studies.

The first category includes works addressing security requirements and architectural robustness. For example, the Ph.D. thesis by Debora Maria Marettek [23] focuses on the elicitation and formalization of security requirements for modular open-source EVSE software. This research emphasizes the complexity introduced by distributed stakeholders and highlights the importance of defining traceable security properties within the EVerest architecture. By structuring and classifying the security requirements, the study contributes to improving the reliability and long-term maintainability of open charging infrastructures.

In a similar context, recent work such as EVECTOR [24] proposes a modular orchestration framework to analyze EV charging infrastructures, including simulation of attack scenarios and fuzzification of communication protocols such as OCPP. Although not specifically focused on customization or hardware integration, this work reflects the broader research interest in assessing the robustness and security of open EV charging stacks, including EVerest.

A second research stream focuses on protocol compliance and interoperability. The master's thesis by Serra Ilgin [25] evaluates the feasibility of building an ISO 15118 compliant charger using the EVerest stack and supported hardware, assessing

communication performance across different EVs and backend systems. The results demonstrate that EVerest can reliably handle high-level communication protocols in real charging scenarios.

Similarly, the interoperability and compliance aspects have been evaluated in collaborative industry contexts. During OCPP *Plugfest* 2025 [26], the *ChargeX Consortium* [27] presented results that highlight EVerest's progress and which test cases were performed [28], including support for Minimum Required Error Codes (MREC). These activities demonstrate ongoing efforts to align the framework with standardized backend communication requirements.

Furthermore, initial developments toward supporting ISO 15118-20 have been reported, extending capabilities beyond ISO 15118-2. This newer standard introduces enhanced features, such as bidirectional energy transfer and improved digital communication mechanisms, including the *Extensible SECC Discovery Protocol* (ESDP) and the *Event Notification Protocol* (ENP). These advancements position EVerest within the evolution of next-generation EV charging standards.

A prototype implementation of ISO 15118-202 messages has been developed within a fork of the EVerest framework, adding support for the Extensible SECC Discovery Protocol (ESDP) and demonstrating message encoding and decoding [29]. However, this implementation, does not include full integration with the core EVerest modules, and much of the exchanged information is set statically for demonstration purposes.

In addition, EXPy provides a native Python interface to the EVerest V2G protocol stack, enabling interaction with the C/C++ libraries and supporting future EXI-based protocols such as ISO 15118-20 [30]. These projects exemplify recent efforts to extend the capabilities of the framework and simplify development for both testing and research purposes.

The third category concerns hardware-oriented implementations and practical adaptations. These studies investigate the integration of EVerest with specific hardware platforms, custom drivers, and embedded systems, evaluating its flexibility when operating under real-world constraints. This research stream is particularly relevant for assessing the framework's adaptability to non-native hardware configurations and its suitability for experimental or prototype-based development.

2.4 From Challenges to Implementation Choice

Despite being a promising solution for the future, open-source approaches are still in an early stage, and the studies mentioned highlight it. Technical limitations, such as

partial support for newer protocols, scalability, and insufficient testing, raise concerns about reliable operations.

Their limited deployment in the EV charging market indicates that standards, protocols, and best practices are still being developed and are yet to be tested. In addition, incomplete documentation, fewer developers, and continually evolving standards make the early stage more challenging.

Although each open-source software has positive and negative sides, EVerest is the most known in the public, being present at convention and working with organizations to make sure that their work is compliant with private software and still being competitive and optimized with newer standards. In addition, it implements various communication protocols and is available for different power supply modes. Furthermore, it implements security techniques that have been discussed in previous studies, such as PKI to secure data exchange between the EV and the EVSE through certificates.

Compared to other open-source EV charging solutions, EVerest appears to be the most mature option for public deployment, despite the fact that it requires customization to be fully deployable. Although solutions such as OpenEVSE, Pazzk-labs evse, and RISE-V2G are more ready for DIY projects and private deployment, the EVerest ecosystem, with its active development community and its compliance with current standards, makes it more suitable for public and private infrastructures.

EVerest, which itself is in an early development phase, is still implementing some parts of the protocols discussed, while others are already fully supported. They provide support to the community by enabling them to fix bugs and work on the documentation to have instructions and information about the project up to date. This collaborative approach helps the project keep up with the ongoing developments in the EV charging ecosystem.

Chapter 3

System Design and Requirements

The project focuses on AC charging. The choice reflects the relevance of AC-based systems in today's EV charging ecosystem, while also providing, possibly, greater opportunities to conduct tests in real-case scenarios.

This focus on AC systems defines the design objectives and hardware requirements carried out in this thesis, while being coherent with the real-world scenarios and scope of the project.

3.1 System Overview

The EV charging system is mainly composed of three key elements: $EV \leftrightarrow EVSE \leftrightarrow$ Back-end. However, it is much more complex than that. This section describes the architecture of the EVSE system.

Figure 3.1 presents a simplified version of the components that work inside the EVSE board. An advantage of the board is that it is connected to the vehicle only through the CP cable that it uses to transmit both PWM and PLC messages. Although the board has only one physical connection to the vehicle, it is connected via the network to back-end services, and it has a physical connection to the grid.

The connections to the grid and to the network are handled by modules of the core software in this architecture, the Custom Software made by EVerest plus the modules added for this project. Although the board offers some key components such as the microprocessor (MP), the RTOS, and the PLC device, the work is done by EVerest and its interfaces. The core of the software layer is EVerest that with both physical and communication modules is able to interact with each component of the system.

EVerest modules can serve different purposes, such as communication, by im-

plementing communication standards (e.g. ISO 15118, OCPP, etc.), physical, by implementing the driver of a specific hardware of the infrastructure, controller, where a module is able to exchange data with upper layers and lower layers. To achieve a coherent flow, a key module orchestrates the majority of connections between different modules, called EVSE Manager. In the following paragraphs, a further explanation with images illustrating the final structure of the thesis work will be provided.

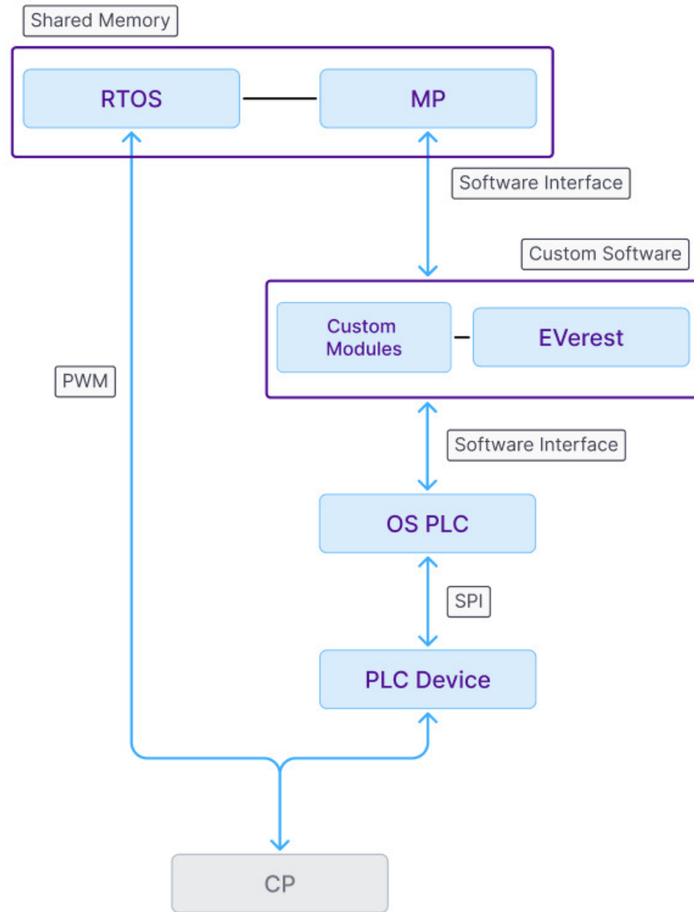


Figure 3.1: EVSE System Overview

3.2 Design Objectives

The main goal of this thesis is to integrate an EV charging stack in an AC-based EVSE with the possibility of implementing standards and features for future development. The stack should be capable of handling communication protocols such as ISO 15118 and OCPP. In addition to communication protocols, the stack should be able to interact with the Operating System (OS) PLC and send information to the MP to trigger the PWM signal over the CP.

The EVSE maintains communication with the grid, the EV, and the back-end

services, so the stack needs to implement communication protocols and back-end interaction. Many versions of the same protocol exist, so the stack should be able to recognize which one is being used to apply the correct one and be more flexible. Another goal of the stack is to achieve a successful communication with every protocol available in the stack and for the specific hardware.

Since EVerest is a modular stack that is able to integrate custom modules for further scalability, the goal for this work is to add custom modules without changing the overall structure of the stack and keeping the pre-existing features to connect it to other modules. This will allow the software to be maintained and extended with upcoming updates or protocols while remaining backward compatible with existing vehicle infrastructures.

To further test the compatibility of the EVSE stack, pre-existing open-source testing tools should be involved, such as Charge Point Management Systems (CSMSs) for OCPP. These tools allow the developer to create a database similar to the one used in real-case scenarios to add testing cards used for authentication. By validating communication protocols against testing tools, the system can demonstrate its compliance and consistency with back-end infrastructures.

Additionally, the maturity of the stack is studied to understand its limitations with protocols, scalability, and maintainability. Especially for the modularity side, where EVerest needs to connect with devices that are not handled by default. Another design objective is to evaluate the portability of the stack by deploying it on a board different from the ones suggested by the software. This goal highlights the flexibility and portability of the software if achieved successfully.

A further objective of this project is to ensure that the system can be validated in realistic EV charging scenarios. The EVerest stack with the addition of the custom parts allows practical testing for communication protocols such as ISO 15118 and OCPP, in AC mode.

3.3 Hardware Requirements

Since design objectives are fundamental for the thesis goals, the hardware requirements must match those goals. The project focuses on AC charging, so the hardware must support the signaling and protection mechanisms defined in IEC-61851, which is foundational for ISO 15118 establishing physical connection. Although there are two different levels of AC charging, the first is used in a low-power environment, and the second, which allows higher current, is most popular for public infrastructures; the EVSE developed in this project was configured for the first. Since the EVSE had to operate in an office environment, it dictated the choice of level 1 AC to use it with

a simple household socket.

To locate the amount of current supplied to the EV and keep it safe during the process, a set of sensors is needed. The fundamental ones are the power meter and the current and voltage sensors. Power metering is used to measure energy consumption (kWh), needed for billing and for verifying the charging sessions. Current and voltage sensors are used not only to check the actual values of current and voltage but also to serve a security role. EVs must be charged under certain parameters, which is why sensors are needed for the security of both the vehicle and the EVSE.

To handle EVSE tasks, a microcontroller unit (MCU) is required. The choice fell on the stmp32mp151 board, a multiprocessor system that allows independent firmware to run on its two ARM CPU cores. The master one is called Cortex-A7 and is optimized to run Linux based OS, referred to as MP; the slave core is called Cortex-M4 and can run RTOS optimized for micro-controllers, referred to as RTOS for simplicity. The two processors that communicate can use two different shared memory regions, where they initialize the shared buffer. To manage the inter-communication, a dedicated peripheral named Inter Processor Communication Controller (IPCC) is required for signaling mailboxes. In addition to IPCC, a semaphore (HSEM) can be used to handle concurrent access to the data. The EVerest stack runs in the MP while the RTOS process is used to handle real-time operations such as PWM communication through CP, with MP signaling through shared memory [31]. Figure 3.2 illustrates the hardware overview. The IPCC does not contain any data, but is just a doorbell to notify processes that the data is available in the shared memory.

The CP is the only cable that connects the EVSE to the EV and has to support different protocols at the same time. To enable ISO 15118 communication and data exchange over the CP, the EVSE requires a HomePlug GreenPHY (HPGP) PLC modem, and the Qualcomm QCA 7005 board was implemented in this hardware architecture to cover this role. The PLC modem is connected to the MCU through a Serial Peripheral Interface (SPI), allowing the custom stack to send data through the CP ensuring compliance with the ISO 15118 standard.

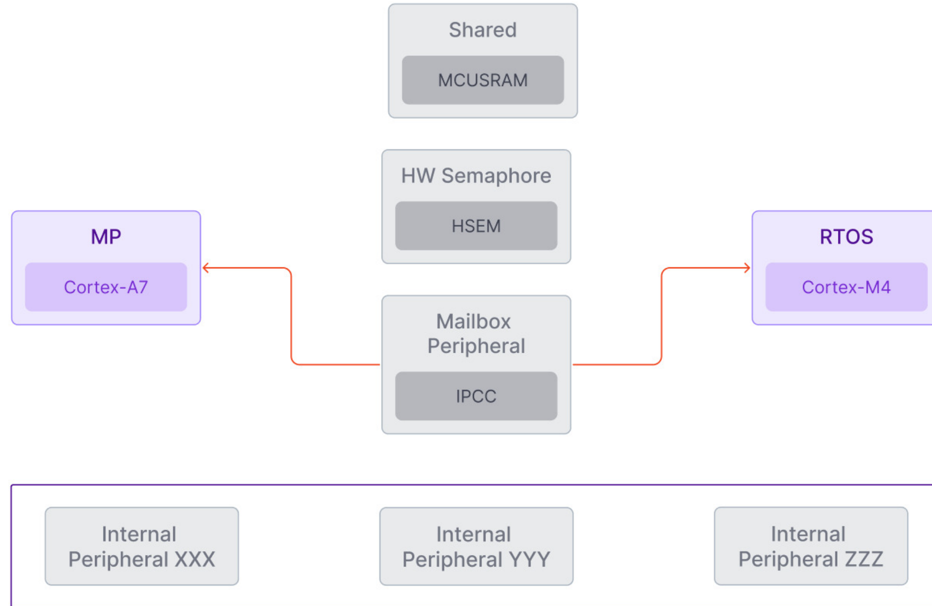


Figure 3.2: STMP32MP151 Overview

Although there are many ways to authenticate the user, the only one that requires dedicated hardware is the RFID method. For this authentication method, the user has a card equipped with an RFID tag that stores a Unique Identifier (UID). When the RFID card is presented to the RFID reader integrated in the enclosure of the EVSE, it reads and transmits the UID to the MCU. The system can then either check a local list or send it through OCPP to a back-end service to authenticate the user. For the implementation of RFID authentication, the NXP PN544 board was adopted.

3.4 Software Framework

The custom software framework, composed of EVerest and additional modules, is the core of the EVSE system. Without it, the hardware would not be able to operate and coordinate. EVerest works with modules, which means that each feature of EVSE is covered by one or more EVerest modules. Figure 3.3 [32] illustrates some of the main components grouped by category. The EvseManager is the main module that contains some control logic for charging configurations, while the other can be organized based on the setup of the project.

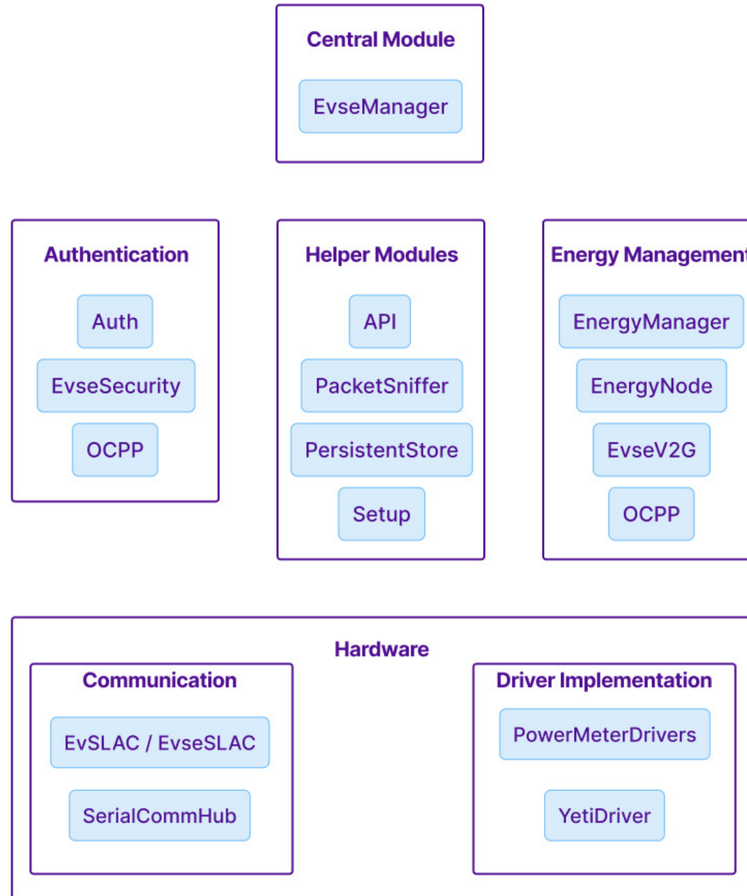


Figure 3.3: EVerest Main Modules

Modules need to communicate with each other and, to do so, interfaces are used to create a link between two modules, handled by a MQTT topic subscription. Interfaces can either be required or provided by a module and can supply a set of commands (CMDs) and variables (VARs). When a module needs specific commands or variable updates to operate it must require a specific interface from another module, vice-versa, it can provide commands or variable updates to other modules providing an interface. Commands are used to start a procedure; for example, the EvseManager can use a command to start the charging session. On the other hand, variables provide updates on parameters offered by an interface; for example, the real-time values of a charging session. An example of a publish-subscribe communication flow is illustrated in Figure 3.4 where Module A provides updates for some variables after a command request from Module B. MQTT plays a crucial role in the communication flow, acting as the message bus of the software, allowing modules to create specific topics for commands and variables.

Every module has a configuration file called *Manifest.yaml*, see the Appendix B.1 for clarity, divided into three sections:

- **Config**: section where the parameters of a module are defined to be later configured (e.g. charge_mode for AC or DC);

- **Provides**: interfaces that other modules can connect to receive specific data;
- **Requires**: interfaces needed from the module in order to operate.

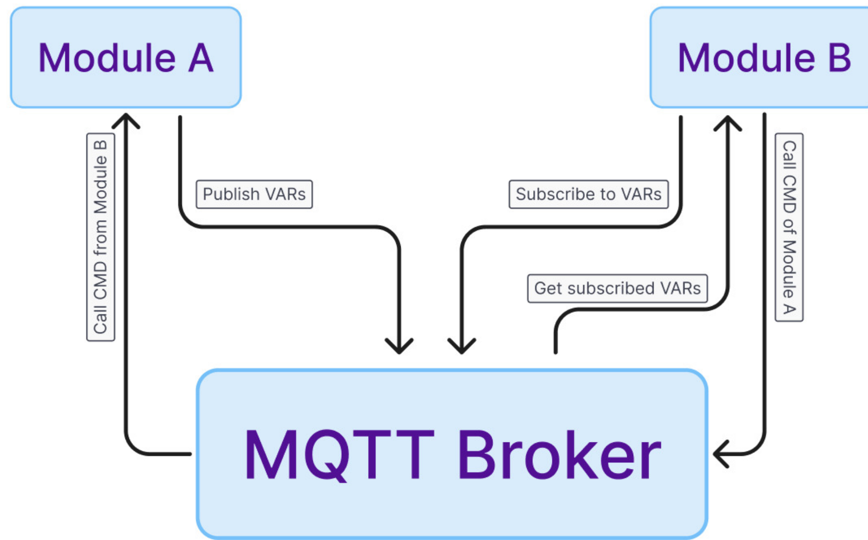


Figure 3.4: Modules Communication

In addition to configuration files for each module, EVerest come up with configuration files for the EVSE system. These configuration files are used to create the EVSE infrastructure declaring all modules needed, its parameters, and its relationships. Furthermore, the configuration file is also used by the framework to initialize and start those modules once the EV charging station is running. There are two available ways to create a configuration file for the system; the first is by manually writing the configuration and each parameter; the second is by using the HMI *EVerest Admin Panel* which enables developers to visualize the configuration through a block diagram, and by clicking on each module parameters can be adjusted to the requirements. The second approach makes it easier for developers since the configuration files can become long and difficult to interpret. In Appendix B.2 an actual configuration file is provided, and in Figure 3.5 the block diagram for the same configuration is provided.

In the configuration shown in Figure 3.5 three modules are connected to each other with directional links. The EvseManager serves as a central module, YetiDriver is used to interact with the board, and EvseSlac is used to start the SLAC communication protocol. With this information, the system is not complete but can only start and complete the SLAC protocol before starting the EV charging process.

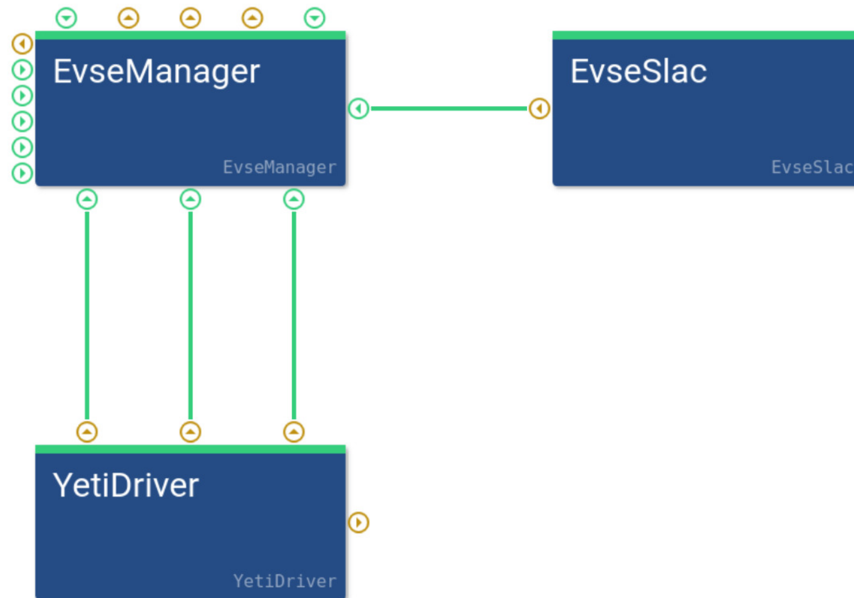


Figure 3.5: System Block Diagram

3.5 Communication Protocol Considerations

One of the goals of this project is to implement the communication protocols that are the most widely used in the EV charging ecosystem. Among those the most well known are ISO 15118 and OCPP. After some consideration and research, the communication protocols adopted for this thesis are as follows: ISO 15118-2 and OCPP 1.6 and 2.0.1. ISO 15118-2 and OCPP 1.6/2.0.1 were chosen because are widely adopted in the EV charging ecosystem and are considered a standard, ensuring compatibility with a majority of existing EVSEs and vehicles. Second, these versions are well-documented and stable, reducing implementation complexity and the risk of unexpected behavior during integration and testing. Third, the available hardware supports AC charging, and ISO 15118-2 provides full functionality in this mode, whereas ISO 15118-20, was unavailable for AC at the time of this project. Finally, OCPP 1.6 and 2.0.1 allow for back-end interactions required for authentication, monitoring, and session management, while remaining compatible with open-source testing tools like SteVe. Taken together, these technical, compatibility, and availability considerations made ISO 15118-2 and OCPP 1.6/2.0.1 the most suitable choices for this thesis work.

Implementing ISO 15118-2 over TCP/IP is not cost effective as it does not talk directly through a physical layer, but it exploits the Internet Protocol for data exchange. ISO 15118 protocol defines the structure and meaning of the messages, while TCP/IP handles the ordered message delivery between the EV and the EVSE, and the reliability of the protocol. ISO 15118-2 also enables the feature PnC, where exchanged data are more sensitive since the EV must authenticate itself, because of this Transport Layer Security (TLS) 1.2 is being run together with TCP/IP in order

to provide authentication, integrity, and confidentiality of the exchanged data.

TLS is used to encrypt the communication between the EV and the EVSE providing an additional level of security. Digital certificates X.509 enable EVs and EVSEs to be authenticated in this type of communication, and are crucial for PnC. X.509 certificates link a Public Key Infrastructure (PKI) to an entity (e.g., a person, a server, a device, etc.), and these certificates are emitted and signed digitally by a Certificate Authority (CA); see Appendix A.3 to better understand the structure of these certificates.

In the EVSE system, a module that verifies these certificates is needed to enable this feature. To actually test the PnC feature in a controlled environment, a set of X.509 certificates must be created, including root, sub-CA, SECC, CSMS, and chain certificates. Figure 3.6 from *Switch* [33] explains what happens during the handshake sequence between the EV and the EVSE, and can be taken as a guide to implement a custom certificate chain.

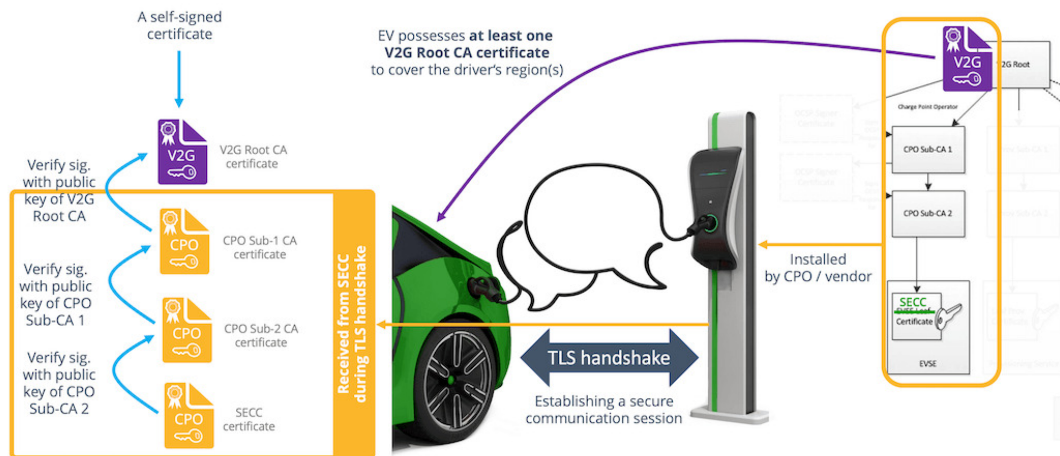


Figure 3.6: TLS Handshake Certificate Exchange

Unlike the ISO 15118 protocol, OCPP has more configuration to do. Instead of having the *Manifest* configuration file, it has additional configuration files in *JSON* format. The OCPP protocol defines many standardized configuration keys that are used as part of the functional requirements of the specification. The structure of the configuration *JSON* file is also different from the two, while remaining customizable according to the requirements. These keys may vary from version to version of the protocol with OCPP 1.6 needing fewer keys than 2.0.1.

Table 2.1 shows the differences from the protocol and provides information on when to use them. Since one of the objectives of this thesis is to also implement the PnC feature, OCPP 2.0.1 would be needed since it supports the PnC feature together with the support of the TLS protocol to secure communication with back-end services.

The back-end service in this case is called the Charging Station Management System (CSMS), while OCPP only acts as a web-socket client. CSMS is reached through TLS, so X.509 certificates are needed to verify those servers.

OCPP not only covers the authorization of a user, but is also able to monitor and control, check for firmware updates, do diagnostics, handle reservations for charging points, and implement billing reporting. For validation purposes, open-source CSMS implementations (e.g. SteVe, MaeVe, or OpenOCPP) can be used to test and verify those back-end features.

3.6 Software Requirements

With the project goals and design considerations defined, the next step is to specify the software requirements. The requirements are defined starting from the pre-existing functionalities of EVerest and identifying the additional features to be implemented.

To provide a clear overview of how the defined software requirements are fulfilled in the EVSE system, using EVerest, Table 3.1 provides the mapping with each functional requirement followed by the modules involved to implement it. This approach highlights how each requirement is supported and provides a clear view of what is missing to be developed.

Feature	**Modules Involved**	**Description**
Controller	EvseManager	Involved in the majority of features by handling data exchanged between components;
Board Interface	MCUDriver	Not available in default EVerest, requires custom implementation when using custom boards. Driver that enables interaction with the board;
ISO 15118 Communication	EvseV2G, EvseSecurity, EvseSlac	The first is used to implement the ISO 15118-2 protocol and the second is used to implement the TLS part for the X.509 certificates, the third is used to start and complete the SLAC protocol before the charging session;
OCPP 1.6/ 2.0.1	OCPP, OCPP201	Enable back-end integration and interaction with back-end services;

Authentication	Auth, OCPP, TokenProvider	Token provider for the chosen board was not provided by EVerest so it needs custom implementation to interact with OCPP and Auth for the authentication process;
Charging Session	EnergyManager, EnergyNode	EnergyManager implements logic to distribute power to EnergyNodes on energy requests;
System Configuration	Config.yaml, JSON files	To be written since EVerest only provides configurations for pre-existing modules.

Table 3.1: Software Requirements Overview

3.7 System Architecture

Taking all the modules listed in Table 3.1, it is possible to build the configuration for a fully working EVSE. Each module is an encapsulation of specific functions that makes it easy to swap with different modules that implement the same interfaces. In Figure 3.7, for example, the OCPP block is shown in generic form, but in practice it can represent either the OCPP 1.6 or the OCPP 2.0.1 module. The choice of version depends only on the configuration file and does not require to re-write any code. At the same time, the OCPP block can be swapped to use a different communication protocol or different techniques to validate tokens. The same happens for the TokenProvider, in this project the token is read from an RFID card while it could be implemented in different ways.

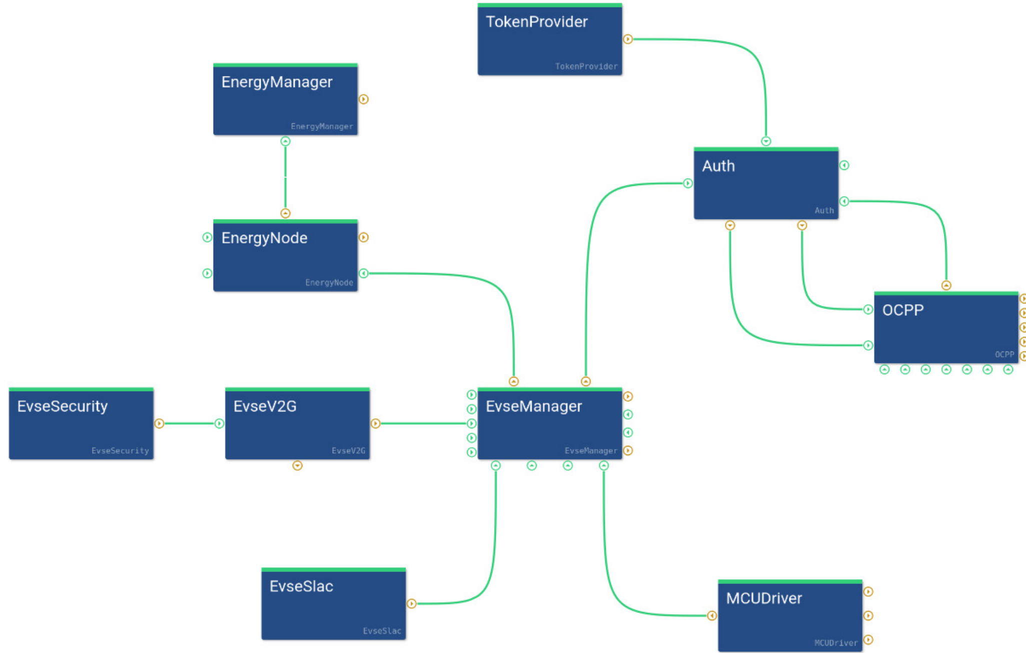


Figure 3.7: Possible System Architecture

3.8 Design Challenges and Trade-offs

During the implementation of the EVerest framework, some challenges have arisen. For instance, due to limited maturity of the framework, at the time of development, ISO 15118-20, for AC mode, and the IsoMux module to support switching over between different ISO module implementations could not be implemented in this project. Instead, the project focused on a more stable version of the protocol. The chosen version, ISO 15118-2, allows one to apply the communication protocol for both AC and DC mode, but due to hardware limitations only the version for AC was tested.

Furthermore, version 2.1 for the OCPP communication protocol was not available, so some of the features implemented in this version could not be integrated into the project, such as bidirectional charging.

Another limitation with respect to the full implementation of the EVerest stack was related to hardware capabilities. Although EVerest, by default, with the right configuration of the EVSE, is able to support multiple charging sockets and parallel charging sessions, the hardware provided only had one available socket. This prevented the testing of this feature even tho the modules were fully available to apply in a possible configuration. In a real-world scenario, such testing would be crucial to ensure a proper deployment.

Finally, another minor challenge was understanding how to create new modules and integrate them into the EVerest ecosystem, ensuring compatibility with the

rest of the stack. Furthermore, it was mandatory to write a *Manifest.yaml* file, in which all configurable parameters and interfaces, both required and provided, are defined. Choosing the right parameters was fundamental to the correct working of the module. Additionally, when implementing the module, interfaces for relationship with other module had to be chosen. Since the modules were similar to others already implemented, it was easier to decide which interfaces to use and which were superficial for the purpose and goals of this project.

Chapter 4

Methodology

4.1 Hardware Configuration

To proceed with integration and customization of the EVerest stack, it is important to create an environment where each component has a defined role and is compliant with the goals of the project. Making a list of all the components adopted with their role can help move on with their setup. Furthermore, documenting the hardware environment allows developers to maintain and further develop the system in an easier way. In the following sections, an illustration of the hardware overview and the setup of the components is provided, which covers the making of the EVSE.

4.1.1 Hardware Overview

In this section, the list of hardware chosen for the project is listed, similar to a *bill of materials*, to give a high-level perspective on the work. With each component, a brief explanation is provided of what they are used for and how they interact with the system.

- **MCU Board (STMP32MP151)**: dual-core architecture with one core used to run a custom Linux OS, responsible for running the EVerest stack, and the other used for RTOS, responsible for the real-time operations.

- **PLC Modem (Qualcomm QCA7005)**: it is required to apply the ISO 15118-2 communication protocol and enables HPGP communication over the CP cable.

- **RFID Reader (PN544)**: retrieves the authentication token when an RFID card is presented, reads the UID, and sends it to the MCU.

- **Power Sensors (current/voltage sensors + power meter)**: used for

protection, monitoring, and billing.

- **Networking Interfaces**: Ethernet or WiFi connection for OCPP back-end communication and integration.

- **Test EVSE Enclosure / Grid Connection**: connected to the grid through a household socket. The enclosure to contain all hardware components. The external Trialog [34] tester board to simulate the EV behavior.

4.1.2 Hardware Setup

The first hardware component to configure is the MCU board, as it is the core component of the EVSE system and to which all the other components need to be connected in order to implement additional features. The STMP32MP151 board has two cores, each with a different OS.

The Cortex-A7 uses a Linux-based OS that is created through a Yocto Project [35] image. YP is an open-source project that helps developers create custom Linux-based systems regardless of hardware architecture. This allows developers to create minimal OS images that contain only the libraries, drivers, and dependencies required for a specific application. It is commonly used in embedded systems, since boards should be optimized for just their scope. Although EVerest provides a YP image for supported boards, in this project, this provided image could not be adopted due to the hardware not supporting it. Instead, an older version was used. However, this image did not include all the dependencies needed to run the default EVerest stack, so I had to manually add them to the board simply by connecting to it through *ssh* or just by copying the files into the board memory.

The Cortex-M4 is optimized for real-time operations, such as PWM and managing time-sensitive tasks. An RTOS is needed to achieve these optimizations. FreeRTOS [36] has a scheduler that is designed to provide a predictable execution pattern. Unlike most operating systems, which appear to allow multiple programs to run concurrently but with a scheduler that is responsible for deciding which program to run and when, in RTOS the scheduler predicts which application or task to execute based on priority to each thread given by the user. To load FreeRTOS into the Cortex-M4 core, STM32Cube was used, which handles building and flashing bare-metal + FreeRTOS firmware.

After the board configuration, the next component is the PLC Modem, which enables data communication over the CP between the EV and the EVSE. For this project, the Qualcomm QCA7005 module was chosen, as it supports the HPGP

standard required for ISO 15118 communication. The QCA7005 is connected to the MCU board via an SPI, allowing EVerest to receive and send data through the CP cable, while also ensuring that the two *understand* each other.

The PN544 RFID reader is also connected to the MCU board through an SPI, and uses a serial data bus library to exchange data with the board. When an RFID card is presented, PN544 reads the UID from the card and transmits it to the MCU, which then forwards it to the software for validation. This device runs a custom software with some interesting features that can be used to facilitate the use of the module, such as activating or deactivating the RFID reader and understanding the type of RFID.

Sensors are also important for the safety of the EVSE since they retrieve data that are handled by the software to understand whether it is working correctly or not. In addition to safety, the values retrieved by the sensors are used to handle billing or to dynamically adjust the power supply of a charging session. To do so, sensors need minimal firmware that allows them to communicate with the MCU board and retrieve data.

Figure 4.1: EVSE Enclosure

The enclosure and the network interface do not really need a configuration, the second is optional since it is mainly used for firmware update or to implement specific communication protocols, such as OCPP that needs to connect with a back-end service to authenticate RFID cars or to handle charging reservations. Network interfaces can also be configured to only allow white-listed IPs to connect to communicate. However, the enclosure is a discretion of the client, which means that there is no specific enclosure where the EVSE resides.

Figure 4.1 illustrates the hardware configuration within the enclosure. This layout does not represent a final configuration. Additional features and devices could be implemented to polish and improve the design, such as integrating a display for users to have real-time details about the charging process.

4.2 Software Development

Once the hardware configuration is complete, all dependencies for the EVSE stack must be installed to ensure that the software runs without errors. However, the board comes with a minimal operating system, which requires the use of a Software Development Kit (SDK) to adapt dependencies from a standard Linux environment to the YP image of the board. In a later section, SDKs will be described in detail, including the steps taken to verify that the SDK used in this project was compatible with both the development OS and the board system.

Additionally, the process of creating and implementing modules in the EVerest stack is analyzed, from writing their *Manifest* to identifying the interfaces required for full integration into a potential EVSE configuration. Furthermore, the configuration functionality is analyzed, along with the procedures to write one properly to set up the EVSE.

4.2.1 EVerest Stack Integration

The integration of EVerest Stack starts with understanding which modules can be reused and which ones need to be swapped with custom ones. As illustrated in figure 3.3, modules can be divided according to their purpose within the EVSE, and this helps with the skimming of modules. The driver modules are more tight to specific boards than others, so the BoardDriver and the RFIDDriver are the ones that need to be exchanged with custom modules. Similarly, I did the research about the most important modules to keep the configuration minimal and functional. The following overview summarizes these modules, grouped according to their purpose, with their primary functionalities.

■ ***Central Module***

- **EvseManager**: is the core module of each configuration, it manages the EVSE operations, handles energy measurements, supports billing if necessary, and enables configuration for basic features like AC or DC, communication protocols supported, payment options, safety options, and misc parameters such as UK smart-charging options.

  – Key configurations: `charge_mode=AC`, `max_current_import_A=32`, `ac_hlc_enabled=true`, `evse_id=IT*XXX*XXXXXX*X`, `ac_nominal_voltage=230`.

- **MCUDriver**: is the module that lets the EVSE interface with the MCU board, enabling data exchange between the software and the board.

  – Key configurations: `caps_min_current=-1`, `caps_max_current=32`

■ **Communication Modules**

- **EvseSlac**: establishes the secure PLC link required for ISO 15118 communication. Supports configuration for retries, timing, chip reset, attenuation, and debugging. It is also compatible with the QCA7005 chip used in this project, with proper configuration.

  – Key configurations: `device=plc1`, `number_of_sounds=10`, `ac_mode_five_percent=true`, `set_key_timeout_ms=1000`

- **EvseV2G**: implements the ISO 15118 communication protocol together with DIN 70121, specifically the HLC between EV and EVSE. This module supports both PnC and External Identification Means (EIM). In addition to session setup and authorization, it also supports energy negotiation and TLS security.

  – Key configurations: `device=plc1`, `supported_DIN70121=true`, `supported_ISO15118_2=true`, `tls_security=true`, `verify_contract_cert_chain=true`

■ **Security Modules**

- **EvseSecurity**: handles certificates that are used during TLS sessions, PnC authorization, and back-end communication such as CSMSs. Additionally, it handles the private keys that are fundamental for secure

communication.

- Key configurations: `v2g_ca_bundle=/path/to/v2gRootCA.pem`, `secc_leaf_cert_directory=/path/to/certDir`, `secc_leaf_key_directory=/path/to/keyDir`, `private_key_password=pwd`

■ **Authorization Modules**

- **Auth**: not only handles authorization for a charging process, but can also implement the logic for reserving a slot for charging. Furthermore, it has some parameters for setting timeouts, priority queue, and the possibility to ignore connector faults.

  - Key configurations: `connection_timeout=10`, `selection_algorithm=FindFirst`

- **TokenProvider (RFIDDriver)**: has the role to read the UID from an RFID card when presented.

  - Key configurations: `timeout_s=50`

- **TokenValidator**: validates tokens locally or via external services such as the OCPP module, which connects to backend systems (as shown in Figure 2.4).

  - Key configurations (OCPP): `ChargePointConfigPath=config.json`

  - Key configurations (Local): `LocalWhiteList=/path/to/list`

■ **Power Supply Modules**

- **EnergyManager**: energy coordinator for all EVSE/Charging stations in the infrastructure. Coordinates power distribution among all available charging stations and provides parameters to continuously monitor and redistribute available current based on nominal AC voltage, update intervals, and energy scheduling. It communicates with the nodes that are below it to spread the current dynamically.

  - Key configurations: `update_interval_s=30`, `schedule_interval_duration_h=1`

- **EnergyNode**: it can be compared to a current fuse in the energy supply chain. It enforces current limits for a security matter while also allowing dynamically restrictions.
  - Key configurations: `fuse_limit_A=32`, `phase_count=1`

Each module has several parameters, so it is recommended to go through all of them to understand which ones are the most important for each implementation. For example, for the EvseSecurity module, certificates will not be needed if the TLS is not enabled. Similarly, the TokenValidator does not have only one possible implementation, so the parameters to verify the UID or the user could be different from each other.

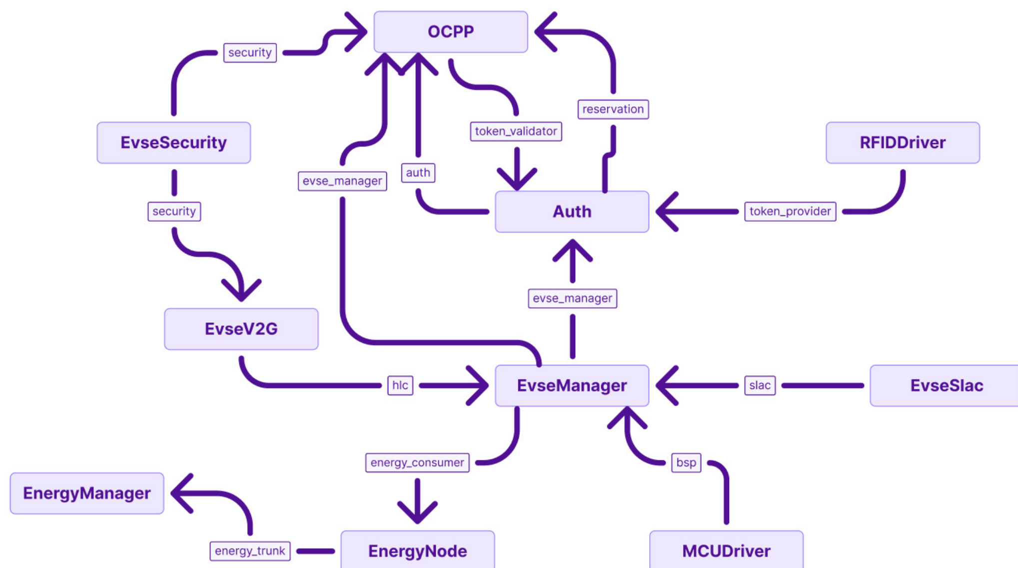


Figure 4.2: Configured EVSE System

Once configured, the EVSE system should look like Figure 4.2, once the two missing modules are developed and implemented. In this figure, the name of the interface that connects two modules is highlighted together with the direction of the link; when is going towards a module it means that that interface is required, when it starts from a module it means that is provided.

4.2.2 Custom Software Components

The following sub-sections detail the design and development of the two custom modules implemented to handle the connection with the MCU board and the RFID card reader. The first, also called MCUDriver, allows the software to connect to the

MCU through Desktop-Bus (D-Bus), similarly the RFIDDriver is connected through the same bus to retrieve information about the presented card, if it is enabled.

4.2.2.1 D-Bus Library Implementation

The D-Bus is an Inter-Process Communication (IPC) system, which allows different running processes to exchange data and synchronize their actions, which is essential for coordinating activities, dividing work, and improving system modularity. Through D-Bus, an application can register itself and offer services to client applications, giving clients the opportunity to search for specific services by asking the D-Bus daemon for the list of services provided by a specific server.

To create new D-Bus services and clients, interfaces are needed, so I added them to the *sdbus-cpp* library, by *Kistler Group* [37], using *XML* files defining the skeleton of those interfaces. In addition to the interfaces, I created a D-Bus Manager class that initializes the custom interfaces implemented to create an adapter and a proxy from the interfaces in order to be used in a framework.

The first interface that I implemented is the Open Asymmetric Multi-Processing (OpenAMP), which enables the two processes to communicate using standardized messaging. However, for this work, only the D-Bus layer is implemented with signals and methods regarding the EV state, such as the CP, the contactor, the charging limits, and the five percent duty cycle, enabling EVerest to communicate with the board and send requests through the MCU to the RTOS processor.

Listing 4.1: OpenAMP XML Configuration

```
<node name="/com/everest_rs/OpenAMP/Adaptor">
  <interface name="com.everest_rs.OpenAMP.Adaptor">

    <!-- CPState: enum/bitfield -->
    <property name="CPState" type="u" access="readwrite"/>

    <!-- EVContactorsState: bitmask -->
    <property name="EVContactorsState" type="u" access="readwrite"/>

    <!-- EVSE AC Charge Limits -->
    <property name="EVSEACChargeLimits" type="(iiiiiiiiiuuu)" access="
    readwrite"/>

    <!-- Five Percent Duty Cicle-->
    <property name="FivePercentDutyCycle" type="b" access="readwrite"/>

    <!-- Signals -->
    <signal name="CPStateChanged">
      <arg name="new_state" type="(uu)"/>
    </signal>
```

```
    <signal name="EVContactorsStateChanged">
      <arg name="new_mask" type="(uuu)"/>
    </signal>

    <signal name="EVSEACChargeLimitsChanged">
      <arg name="limits" type="(iiiiiiiiiuuu)"/>
    </signal>

    <!-- Methods -->
    <method name="ReqCPState">
      <arg name="result" type="s" direction="out"/>
    </method>

    <method name="SetEVSEACChargeLimits">
      <arg name="limits" type="(iiiiiiiiiuuu)" direction="in"/>
    </method>

    <method name="chargeConnector">
      <arg type="(uudd)" direction="in"/>
      <arg type="u" name="result" direction="out"/>
    </method>

  </interface>
</node>
```

In the Listing 4.1, a simplified *XML* file of the *OpenAMP* implementation is presented to provide an example of how the library creates auto-generated client and server, a deeper look into these two files is shown in Appendix B.3. This method of implementation, similar to what EVerest does with *Manifest* files, allows the user to understand quickly what the interfaces do without going through all the code. However, the *OpenAMP* interfaces will not work without a proper initialization so from the *DBusHandler* class a method called *initOpenAMP* is required, as Listing 4.2 shows.

Listing 4.2: initOpenAMP code snippet

```
int DbusHandler::initOpenAMP()
{
    const std::string destName { OPENAMP_GATEWAY_SERVICE };
    const std::string objPath { OPENAMP_GATEWAY_OBECJT };
    const std::string managerPath { OPENAMP_GATEWAY_OBECJT };
    if ( !openAMP )
    {
        try
        {
            openAMP = std::make_unique< OpenAMP_Proxy >( *
    sessionConnection, destName, objPath );
            object_manager__openamp__adaptor =
                std::make_unique< ObjectManagerAdaptor >( *
    sessionConnection, managerPath );
            return 0;
```

```
        }
        catch ( const sdbus::Error &e )
        {
            return -1;
        }
    }
    return -1;
}
```

I applied the same process to the RFID section in order to create an interface that is able to signal whether or not a card was presented, when it is removed, to enable the reading of cards, and to retrieve errors when encountered, as illustrated in Listing 4.3.

Listing 4.3: RFID Reader XML

```
<node name="com/everest_rs/RFIDReader/Rfid">
    <interface name="com.everest_rs.RFIDReader.Rfid">

        <!-- readEnable: used to change the state of the board to
    reading or resting -->
        <property name="readEnable" type="b" access="readwrite">
            <annotation name="org.freedesktop.DBus.Property.
    EmitsChangedSignal" value="false"/>
        </property>

        <!-- Signals -->
        <signal name="cardDetected">
            <arg type="u"/>
            <arg type="ay"/>
        </signal>

        <signal name="cardRemoved">
        </signal>

        <signal name="readEnableChanged">
            <arg type="b"/>
        </signal>

        <signal name="error">
            <arg type="t"/>
        </signal>
    </interface>
</node>
```

Once the *sdbus* library implements these two important interfaces, the process to develop the custom modules for EVerest can start, allowing them to use the library to exchange data with the components on the other side by being a client of them.

4.2.3 From *Manifest* to *Module*

The *Manifest* file is the first file to be written in order to create a new EVerest *Module*. This file can be written in a simple text editor and added to a folder with the name assigned to that specific module.

EVerest provides a *Workspace* that contains all the tools needed to generate *stub* files for custom implementation and can be easily set up following their documentation [38]. The two main tools used are *edm* and *ev-cli*, and are also a requirement for the whole environment to work.

Inside the directory where all modules reside in *everest-core*, a new folder must be created with the name of the module to be developed. The *Manifest* file should be placed inside this folder. By executing a simple *ev-cli* command, as shown in Listing 4.4, a set of auto-generated *stub* files is created within the folder to serve as a starting point for development. These *stub* files provide function signatures with empty implementations, allowing the developer to implement the required functionality.

Listing 4.4: bash Command

```
ev-cli module create --schemas-dir $EVEREST_WORKSPACE/everest-framework
    /schemas MCUDriver --licenses $EVEREST_WORKSPACE/everest-utils/ev-
    dev-tools/src/ev_cli/licenses
```

A similar process can be followed to create new *Interfaces*, by writing a new *YAML* file that describes a needed interface to link two or more modules, as illustrated in Listing 4.5. And also here, by using a simple bash command 4.6 the *stub* files to customize this interface will be generated. After these commands, the structure of the project should look like the one in Listing 4.7

Listing 4.5: YAML Interface Example

```
description: The interface of the tutorial module.
cmds:
  command_tutorial:
    description: A command the tutorial module's interface provides. It
     receives a simple string.
    arguments:
      payload:
        description: An arbitrary string that can be sent to the module
        type: string
    result:
      description: The answer of the module (which per default will
    just be "everest").
      type: string
```

Listing 4.6: bash Command Interface

```
ev-cli interface generate-headers --schemas-dir $EVEREST_WORKSPACE/
    everest-framework/schemas interface_tutorial
```

Listing 4.7: Project Structure Overview

```
.
|-- build
|   `-- generated
|         (...)
|-- config
|-- interfaces
|   `-- interface_tutorial.yaml
`-- modules
    `-- TutorialModule
        |-- CMakeLists.txt
        |-- TutorialModule.cpp
        |-- TutorialModule.hpp
        |-- doc.rst
        |-- docs
        |   `-- index.rst
        |-- interface_impl_tutorial
        |   |-- interface_tutorialImpl.cpp
        |   `-- interface_tutorialImpl.hpp
        `-- manifest.yaml
```

4.2.3.1 Module for MCU Driver

The *MCU Driver* is the missing part that lets us link *EVerest* with the custom hardware in this custom implementation. It connects the high-level with the low-level part of the software by using *sd-bus* library to send messages to the *RTOS* processor and also *interfaces* to cover the *high-level* communication through *MQTT*. This module not only communicates with the hardware but also interprets the signals sent from the hardware to adjust, for example, it listens to changes on the *CP* cable in order to check the charging state, it is also useful to trigger errors and to go from one state to another. Another important feature of this module is to enable the *5% Duty Cycle*, which makes it possible to start the charging process.

Listing 4.8: MCU Driver YAML Configuration

```
description: Module used to connect to a sdbus and talk to EvseManager
config:
  caps_min_current_A:
    description: Minimal current on AC side. For AC this is typically
    6, but for HLC this can be less. -1 means use limit reported by HW.
    type: integer
    default: -1
  caps_max_current_A:
```

```
    description: Maximum current on AC side. For AC this is typically
    16 or 32, but for HLC this can be less. -1 means use limit reported
    by HW.
    type: integer
    default: -1
provides:
  board_support:
    interface: evse_board_support
    description: board support interface to low level control pilot,
    relays, motor lock
enable_telemetry: true # with telemetry enabled, the module will
    collect data about performance, usage etc.
metadata:
  license: https://opensource.org/licenses/Apache-2.0
  authors:
    - Riccardo Scanu, Abinsula
```

The *Manifest* and header files provide information on the purpose and implementation of this module. The two parameters, *caps_min_current_A* and *caps_max_current_A*, can be set by default, but the driver can also request them from the hardware. During initialization, the driver sets several key parameters, such as the minimum and maximum current for import and export, the minimum and maximum phase counts, the support for changing phases during charging, and the connector type, which are collectively referred to as capabilities or limits. In addition, the module provides the *board_support* interface, which handles low-level messages including the control pilot status, the relay status that indicates whether the vehicle is charging and the limits that must be respected during charging. These signals are received via the *sd-bus* library, necessitating the creation of a client capable of listening to messages sent by the real-time processor.

The code shown below is from a custom class that initializes the *OpenAMP Client* needed by *evse_board_support* interface, in order to communicate with other modules and let them know that an event occurred, such as the control pilot status or the contactor state changed. The communication of this event is made possible thanks to the type *board_support_common* provided in the default implementation of *EVerest*, so I just needed to create a custom function that converts the parameters received through the *OpenAMP* to objects that are understandable by *EVerest* without implementing a custom object just for these parameters.

This specific function first initializes the *OpenAMP* client through *D-Bus*, in case of success, it registers lambda functions as callbacks for three main *OpenAMP* events: *CP State*, *Contactors State*, and *EVSE AC Charging Limits*. I used *lambda* functions to handle these events, since those could be triggered asynchronously and at the same time they needed access to various objects of the class, for example the *mutex lock* to modify the capabilities. In addition to that, without *lambda* functions, the code becomes longer with the need to implement static methods and proper access

to global objects; but with this style, the code is more readable and compact since each of these callbacks have few lines of code to react immediately to the registered signals and is also safer.

Listing 4.9: Create of the OpenAMP Client

```
int SdbusProxy::createOpenAMPClient() {
    int error = -1;
    if ( mDbusHandler )
    {
        error = mDbusHandler->initOpenAMP();
        if ( !error )
        {
            EVLOG_info << "Connected to OpenAMP gateway service: " <<
    OPENAMP_GATEWAY_SERVICE << OPENAMP_GATEWAY_OBECJT;
            mDbusHandler->OpenAMP()->register_onCPStateChangedCb(
                [this]( const sdbus::Struct< uint8_t, uint8_t > &
    CPByConnectorId ) {
                    cpState = static_cast< MCU::CP_State >(
    CPByConnectorId.get< 1 >() );
                    types::board_support_common::BspEvent lBspEvent =
    convertCpStateToEvent( CPByConnectorId.get< 1 >() );
                    EVLOG_info << "Received CP state change: " <<
    event_to_string( lBspEvent.event );
                    setBspEvent( lBspEvent );
                } );
            mDbusHandler->OpenAMP()->
    register_onContactorsStateChangedCb(
                [this]( const sdbus::Struct< uint8_t, uint8_t, uint8_t
    > &contactorsByConnectorId )
                {
                    bool lRelaysStateIsClosed = (
    contactorsByConnectorId.get< 2 >() == 1 ); // Should be closed when
    1
                    EVLOG_info << "Received Relays state closed: " <<
    lRelaysStateIsClosed;
                    types::board_support_common::BspEvent lBspEvent =
    convertRelaysStateToEvent( lRelaysStateIsClosed );
                    setBspEvent( lBspEvent );
                } );
            mDbusHandler->OpenAMP()->register_onEvseAcChargeLimitsCb(
                [this]( const sdbus::Struct< int, int, int, int, int,
                    int, int, int, int, int,
                    uint32_t, uint32_t, uint32_t > &evseAcChargeLimits
    ){

                    std::lock_guard<std::mutex> lock(capsMutex);

                    int minCurrent = evseAcChargeLimits.get<0>();
                    int maxCurrent = evseAcChargeLimits.get<1>();
                    int minPhases  = evseAcChargeLimits.get<2>();
                    int maxPhases  = evseAcChargeLimits.get<3>();
```

```
                    // Update capabilities based on received limits
                    caps.min_current_A_import = (minCurrent >= 0) ?
    minCurrent : 0;
                    caps.max_current_A_import = (maxCurrent >= 0) ?
    maxCurrent : 32;
                    caps.min_phase_count_import = (minPhases > 0) ?
    minPhases : 1;
                    caps.max_phase_count_import = (maxPhases > 0) ?
    maxPhases : 1;
                    caps.supports_changing_phases_during_charging =
    false;

                    EVLOG_info << "Received Evse AC Limits";

                    setCapsEvent( caps );
                }
            );
        } else {
            EVLOG_error << "Failed to connect to OpenAMP gateway
    service: " << OPENAMP_GATEWAY_SERVICE
                        << OPENAMP_GATEWAY_OBECJT;
        }
    }

    return error;
}
```

Lambda functions are also used in the implementation of the *evse_board_support* interface, in its *init* function, to register signals defined by the custom class, by doing so, I managed to divide the core business logic of these two components.

The module *EvseManager* handles the state machine, which is important to understand the status of the charging process, and depending on the state it delegates a specific module to execute some tasks. For example, once the state machine enters the *WaitingForAuthentication* state, while *HLC* is enabled, the *pwm_on* event is published and received by the *MCU Driver*, which sets the duty cycle to *5%* to be compliant with *ISO 15118* protocol. The driver then uses the *OpenAMP* function to send the command to the vehicle. Similarly, to handle *pwm_off* the *EvseManager* publishes the event, which once received by the driver it needs to retrieve the CP state to stop the charging process and release the EV. In the Listing 4.10 the parameter *MPU_AC_STOP* is also set to let the architecture know that current should not flow anymore and stop it.

Listing 4.10: handle_pwm_off Implementation

```
void evse_board_supportImpl::handle_pwm_off() {
    // your code for cmd pwm_off goes here
    EVLOG_info << "AbiMCUDriver - evse_board_supportImpl::
    handle_pwm_off()";
```

```
    MCU::CP_State cpState = sdbus_proxy.getCpState();
    if( cpState == MCU::CP_State::CP_STATE_A1 || cpState == MCU::
    CP_State::CP_STATE_A2 ) {
        // If the CP state is A1 or A2, the charge can be stopped
        EVLOG_info << "Stopping charge on connector 1";
        sdbus::Struct< uint8_t, uint8_t, double, double > params(
            static_cast<uint8_t>(MCU::ConnectorId1),
            static_cast<uint8_t>(MCU::AC_ChargeCommand_enum::
    MPU_AC_STOP),
            ( double ) 0.00,
            ( double ) 0.00
        );
        sdbus_proxy.setAbiMcuChargerSocket( params );
    }
}
```

4.2.3.2 Module for RFID Driver

The *RFID Driver* is a very useful module that enables users to authenticate and start the charging process with a physical card, without needing a third party application for this process. This module connects the software with the hardware using the previously mentioned *sd-bus* library, with the custom interface to use the module only when needed, with a function that enables the read and deactivates the device when is not needed anymore. The logic behind this module is minimal since it does not have to verify the token but it publishes the *uid* to let other modules do it.

Listing 4.11: RFID Driver YAML Configuration

```yaml
description: Abinsula RFID token provider for the PN544
provides:
  main:
    description: Implementation of Abinsula PN544 RFID token provider
    interface: auth_token_provider
    config:
      timeout:
        description: Time a new token is valid (in s)
        type: number
        minimum: 0
        maximum: 120
        default: 30
      read_timeout:
        description: Time between subsequent card reads (in s)
        type: integer
        minimum: 0
        maximum: 120
        default: 5
metadata:
  license: https://opensource.org/licenses/Apache-2.0
  authors:
    - Riccardo Scanu, Abinsula
```

In its *manifest* there are only two parameters defined, *timeout* and *read_timeout*; the first one is used to define the window of time during which the provided token is still valid; while the second one is a timeout that can be used between reads to not create a bottleneck in between reads. Additionally, it provides the *auth_token_provider* interface that the module uses to share through *MQTT* the token read from the user card. By combining this information and the *sd-bus* interface, I managed to make a module capable of working in the software ecosystem.

The code shown in Listing 4.12 illustrates how the *lambda* expressions are registered to the callbacks provided by the *sd-bus* interface defined in the *XML* file 4.3. The two main functions are *register_onTagDetectedCb* and *register_onReadEnableChangedCb*. The first is used to retrieve the information from the device when an RFID card is presented for authentication, and when it happens, a new *ProvidedIdToken* object is created and filled with the information for authorization, such as the *uid* and the type. Then the token is published by the module so that the module in charge of authentication can retrieve it and try to match it against a database of active ids. Finally, the thread goes to sleep for a time defined by *read_timeout*. On the other hand, the *register_onReadEnableChangedCb* lets the software decide when to activate the device and when to turn it off using a simple boolean value, so that resources to keep the module active all the time will not be wasted.

Listing 4.12: Create RFID Client

```
int SdbusProxyRfid::createRFIDClient()
{
    int error = -1;
    if( mDbusHandler )
    {
        error = mDbusHandler->initRfid();
        if ( !error )
        {
            EVLOG_info << "Connected to RFID handler service: " <<
RFID_HANDLER_SERVICE << RFID_HANDLER_OBECJT;
            setRFIDReadEnable( true );
            mDbusHandler->Rfid()->register_onTagDetectedCb(
                [this]( const std::string &uid )
                {
                    EVLOG_info << "Received RFID tag detected";
                    types::authorization::ProvidedIdToken
provided_token;
                    provided_token.id_token = { uid, types::
authorization::IdTokenType::Local };
                    provided_token.authorization_type = types::
authorization::AuthorizationType::RFID;
                    setProvidedToken( provided_token );
                    std::this_thread::sleep_for(std::chrono::seconds(
config.read_timeout));
                }
```

```
            );

            mDbusHandler->Rfid()->register_onTagRemovedCb(
                [this]()
                {
                    EVLOG_info << "Received RFID tag removed";
                    resetRFID();
                }
            );

            mDbusHandler->Rfid()->register_onReadEnableChangedCb(
                [this]( const bool &enable )
                {
                    EVLOG_info << "Received RFID read enable changed:"
    << enable;
                }
            );

            mDbusHandler->Rfid()->register_onErrorCb(
                [this]( const uint64_t &errorCode )
                {
                    EVLOG_error << "Received RFID error:" << errorCode;
                    setRFIDErrorEvent(errorCode);
                }
            );
        } else {
            EVLOG_error << "Failed to connect to RFID Manager service:"
                        << RFID_HANDLER_SERVICE
                        << RFID_HANDLER_OBECJT;
        }
    }
    return error;
}
```

4.2.4 Research of Testing Tools

In order to verify the written code, I needed a strong testing configuration able to emulate the behavior of a real EV, from the SLAC process to the end of the charging process. To do so, I used an EV-emulator from *Trialog*, this device needs to be connected to the enclosure through the plug, replicating the physical interface of an EV, but it also needs an Ethernet cable for the PLC communication and to be plugged to the current to operate. The internet connection is also important to access the web application where the emulator can be configured to create different charging scenarios, as illustrated in a simplified Figure 4.3 where the main possible parameters can be set among the provided ones. Using an emulator instead of a physical EV offers full control over the charging sequence, the ability to reproduce consistent test conditions, and detailed access to ISO 15118 and Combined Charging System (CCS) communication layers that would otherwise be difficult to capture in a real-vehicle environment.

The UI also shows two more sections, called *History* and *CCS*, the first one making it possible to record and download the full communication log as a file that could be used for debugging and analysis; while the second one, CCS, is called like that since lets the user monitor or simulate the CCS communication, allowing one to visualize the handshake sequence, observe CP status, view ISO 15118 messages exchanged over PLC, and check log messages. In addition to monitoring features, the emulator's page provides some buttons and configuration fields that allow the developer to control the charging process more deeply, such as buttons to simulate cable connection or disconnection, trigger a manual stop of the charging process, or run the entire scenario automatically.

Figure 4.3: Emulator UI from Web Application

The emulator web page was fundamental: it allowed me to rapidly view the data that the EV was exchanging with the EVSE, instead of having to convert the messages using other external software, such as Wireshark and Exi-decoder, where the messages appear in their raw form, like the one in Figure 4.4 making it harder to quickly point out the exact content just by looking at it. On the other hand, the *CCS* page from the web application already provides an encoded and refactored representation of the message, organized under *Before Charging Loop* and *During Charging Loop* sections, as illustrated in the Image 4.5.

In the same page on the left side, the previously mentioned buttons are located to make real-time changes, making it easy to control the flow of the process and check the output at the same time. These buttons can be used to trigger different actions that can also generate unexpected behaviors, for example, by forcing an

early stop using the *Request to Stop* button or unplugging the CP cable with the *CP* button. Such interactions are extremely valuable for triggering edge cases and observing how the EVSE implementation reacts under unexpected or unusual conditions.

In order to have a complete view, it is important to balance the two methods: use the UI to have a real-time understanding, while adding human interactions through buttons, and keep the full history (log) of messages that can be loaded to an external software, making it possible to inspect the process at a slower pace and perform more detailed debugging whenever required.

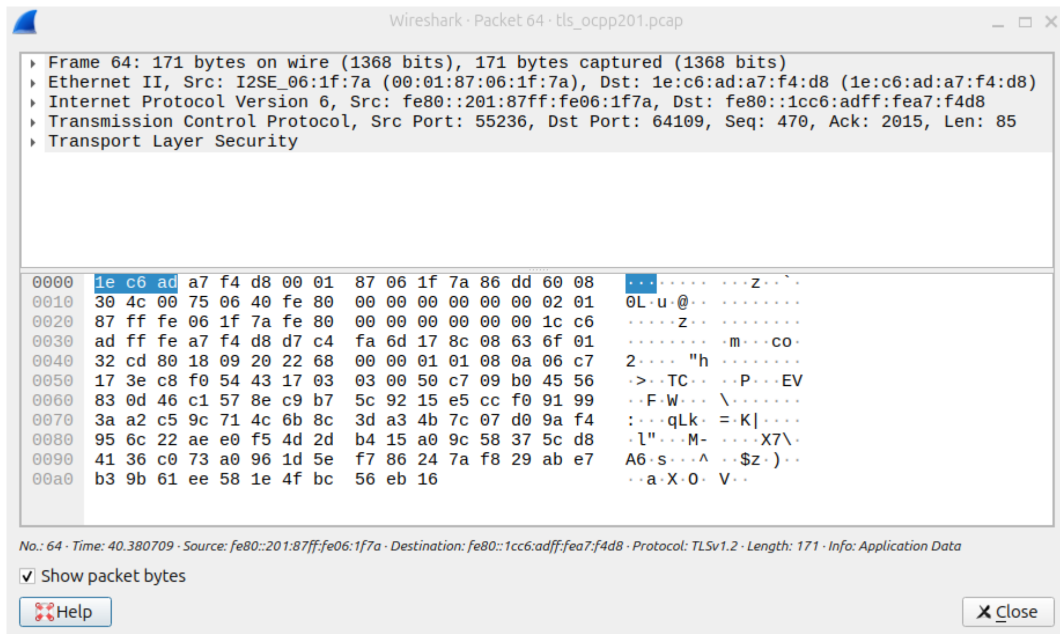


Figure 4.4: TLS Raw Message

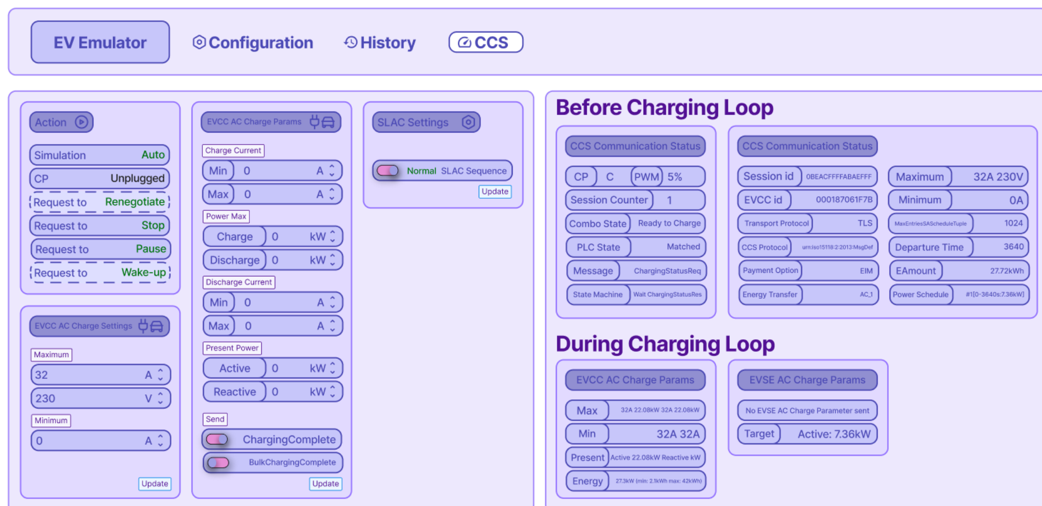


Figure 4.5: Emulator CCS UI from Web Application

4.2.4.1 OCPP server-side Test Environments

Since one of the main goals of this thesis is to make sure that the EVerest custom implementation is compliant with the versions 1.6 and 2.0.1 of the OCPP protocol, to achieve this it was necessary to rely on external tools that behave like real CSMS and are aligned with the OCPP protocol functionalities. For this purpose, I selected two open-source tools: *SteVe* [39] and *MaeVe* [40]; and even though they cannot be referenced as an official implementation of the Open Charge Point Protocol, they are widely adopted in research and development environments due to their ability to emulate the real protocol and cover each functionality, making them suitable as practical testing tools for EVerest software.

SteVe supports the OCPP protocol until the 1.6J version, while *MaeVe* can also support the 2.0.1 version; the first was employed during the tests of the EVerest configuration with OCPP 1.6 while the second with the configuration with OCPP 2.0.1, and both were used for the RFID card authentication process, although they provide other CSMS features.

Both tools require minimal configuration, primarily based on the setup of a database, which is going to be automatically generated by the tools, server parameters, and administrator credentials for accessing the web dashboard. On the EVerest side, the OCPP modules must be configured using *JSON* files, provided by the *libocpp* library, which define how the module is going to connect to the CSMS services and specify the parameters for the OCPP protocol.

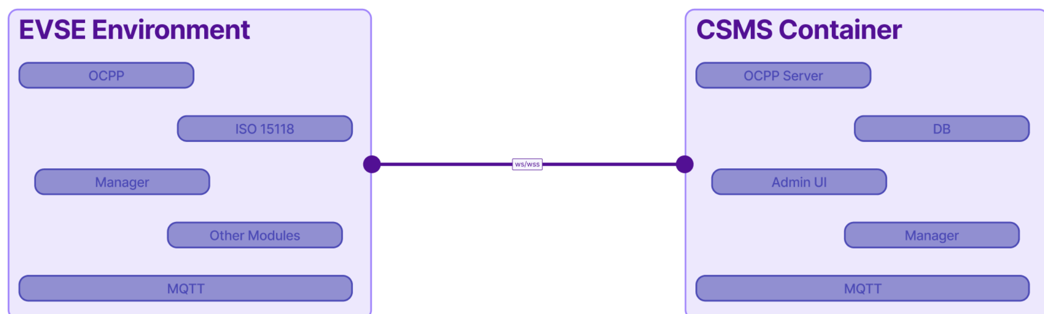


Figure 4.6: CSMS and EVSE Overview

4.2.4.1.1 SteVe CSMS

To configure SteVe, there are a few key steps to follow. First, the system requires access to a relational database, such as MySQL or PostgreSQL, which is automatically generated by *SteVe* during the first startup of the software. This database is used to store all the information related to OCPP that CSMS needs to handle, such as registered charge points, RFID identifiers, user accounts, transaction records, and authorization logs. In practice, the database acts as the central data store for the

entire testing environment, to not lose data after each iteration.

An administrator account must then be created to access the web dashboard, which is locally hosted and provides a UI to the underlying database, which can be accessed by the developer to add or modify charge point configurations, manage RFID tokens, inspect ongoing or past charging sessions, and monitor *JSON* based OCPP communication in real time, such as *Active Charge Points*, *Received Heartbeats* and *Connector Status*.

Each EVerest module requires a *JSON* configuration file that looks like the one in Listing 4.13, which illustrates how the EVerest OCPP module is configured to connect to *SteVe* and use its services to authorize EV recharge. Additionally, this configuration file should be placed in a directory that is reachable by the module, and the path should be specified in the configuration file of the EVSE system. In the configuration file, it is important to put the right web socket, which enables the EVerest OCPP module to connect to the services offered by SteVe, but keep in mind that this approach is not secure since web sockets are vulnerable to different cyber attacks and are used for this work for the purpose of testing the environment.

Listing 4.13: OCPP 1.6 Module Configuration File

```
{
    "Internal": {
        "ChargePointId": "abi001",
        "CentralSystemURI": "192.168.1.225:8080/steve/websocket/
    CentralSystemService/",
        "ChargeBoxSerialNumber": "abi001",
        "ChargePointModel": "Abi",
        "ChargePointVendor": "Abinsula",
        "FirmwareVersion": "1.0.0",
        "LogMessagesFormat": []
    },
    "Core": {
        "ConnectionTimeOut": 60,
        "GetConfigurationMaxKeys": 35,
        "HeartbeatInterval": 14400,
        "LocalAuthorizeOffline": true,
        "LocalPreAuthorize": true,
        "MeterValuesAlignedData": "Energy.Active.Import.Register",
        "MeterValuesSampledData": "Energy.Active.Import.Register",
        "MeterValueSampleInterval": 5,
        "NumberOfConnectors": 1,
        "ResetRetries": 3,
        "StopTransactionOnEVSideDisconnect": true,
        "StopTransactionOnInvalidId": true,
        "SupportedFeatureProfiles": "Core,LocalAuthListManagement,
    SmartCharging",
        "TransactionMessageAttempts": 3,
```

```
        "TransactionMessageRetryInterval": 5,
        "UnlockConnectorOnEVSideDisconnect": true
    },
    "LocalAuthListManagement": {
        ...
    },
    "Security": {
        ...
    },
    "SmartCharging": {
        ...
    }
}
```

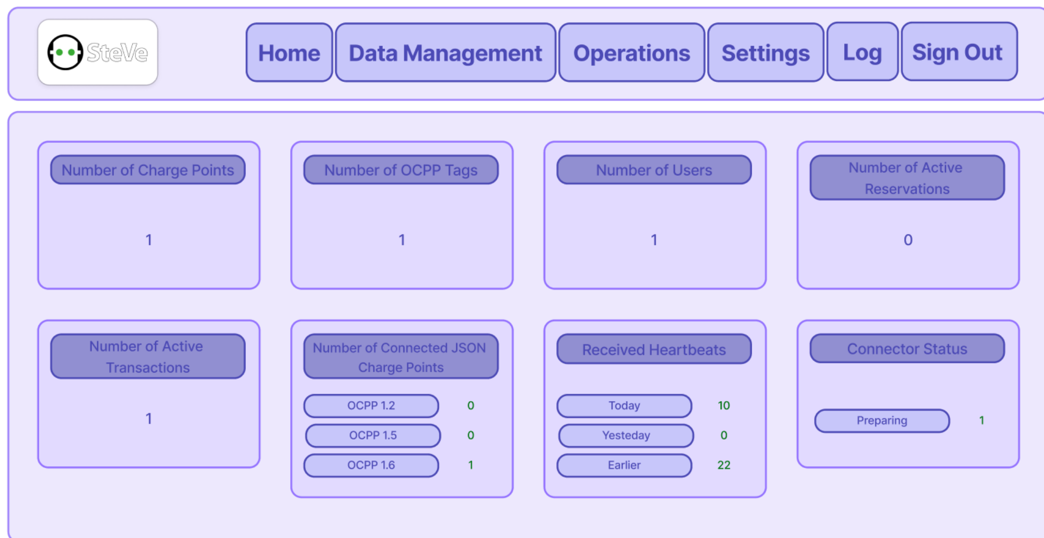


Figure 4.7: SteVe UI Home Page

Once everything is configured, *SteVe* can be started and it will show the direct link to access the web interface that looks like the one shown in Figure 4.7. From the main page, by clicking on the different sections, the developer can add the configuration for the EVSE and for the RFID cards. In addition, users can set up and connect them to specific RFID ids and also check real-time data, such as *Received HeartBeats* and *Number of Connected JSON Charge Points*.

To add a *Charge Point*, *SteVe* provides a very simple form that can be filled through the dashboard with information related to the EVSE, such as the ID, the OCPP protocol that it uses, name of the vendor, firmware version, and others; it can also be filled with additional information that can be used to track down the exact EVSE with a *Serial Number* and its location with different parameters, such as *address*, *city*, *zip code*.

As the image shows, *SteVe* provides as a feature a section equal to *Wireshark*,

called *Log*, where the communication between the devices can be debugged, even though I mainly used the external tool to debug and verify the integrity of the communication, it is pleasant to have a tool integrated that can already filter the communication instead of relying on other tools.

Figure 4.8: SteVe Add RFID Tag Form

The most important section for the purpose of this work is the *OCPP Tag Details* page, which handles the IDs of RFID cards, and maintains the list of already approved Tags (from the Active to the Expired ones). An example of this page can be seen in Figure 4.8, where the information is grouped into three different categories: *See Operations*, *OCPP*, and *Misc*. Through this interface, the developer can register new Tags, enable or disable already existing ones, associate them to a parent RFID Card, and define the expiry date and time. This page is crucial since it allowed me to save the ID of the card and test the authentication phase with EVerest software, which through the RFID module retrieves the identifier and gets sent to the CSMS by the OCPP module, then *SteVe* will match against the database the received information and will send back the result as *Accepted* or *Rejected*. In addition, the possibility to rapidly update the Tag information through the dashboard made it easier to validate authentication scenarios, such as expired or invalid tags.

4.2.4.1.2 MaeVe CSMS

MaeVe is the second tool employed as the testing tool for the validation of the OCPP protocol, specifically for the version 2.0.1. This makes *MaeVe* suitable for testing more advanced features of the protocol and how the EVerest software behaves, even though for this work it was used to test the authentication of RFID cards.

Compared to *SteVe* it has more parameters to configure based on the fact that it implements more features, and those can be configured through a set of *JSON*

files that looks like the one shown in the Listing 4.14. These configuration files, similar to the version 1.6, must be made available to the OCPP module through the configuration file of the EVSE system, so that it can correctly connect to the service and exchange messages with it. Furthermore, *MaeVe* gives the opportunity to use Web Services Security *WSS*, to establish a secure connection using certificates, which I had to create in order to verify the validity of this feature.

Listing 4.14: Project Structure Overview

```
.
`-- config
    `-- custom
        |-- Connector_1_1.json
        |-- EVSE_1.json
    `-- standardized
        |-- AlignedDataCtrlr.json
        |-- AuthCacheCtrlr.json
        |-- AuthCtrlr.json
        |-- ChargingStation.json
        |-- ChargingStatusIndicator.json
        |-- ClockCtrlr.json
        |-- CustomizationCtrlr.json
        |-- DeviceDataCtrlr.json
        |-- DisplayMessageCtrlr.json
        |-- InternalCtrlr.json
        |-- ISO15118Ctrlr.json
        |-- LocalAuthListCtrlr.json
        |-- MonitoringCtrlr.json
        |-- OCPPCommCtrlr.json
        |-- ReservationCtrlr.json
        |-- SampleDataCtrlr.json
        |-- SecurityCtrlr.json
        |-- SmartChargingCtrlr.json
        |-- TrafficCostCtrlr.json
        |-- TxCtrlr.json
```

Unlike *SteVe*, *MaeVe* does not have a dashboard that can be accessed in a browser, but it is containerized using a set of Docker containers, after generating the TLS certificates and executing an already provided `run.sh` script, and the manager operations, such as registering a new RFID ID or inserting a new charge station profile, are done through the built-in Manager Application Programming Interface (API), which are accessible by commands like the one in Listing 4.15. The command illustrated is used to create or update a certain charging station, by using its `cs-id` and setting its security profile level equal to 2, which means that this EVSE will use TLS with client certificates. If the EVSE is not already registered, *MaeVe* will create a new entry in its database with all the parameters required, otherwise, it will update its content.

Listing 4.15: Project Structure Overview

```
$ curl http://localhost:9410/api/v0/cs/<cs-id> -H 'content-type:
    application/json' -d '{"securityProfile":2}'
```

The Manager API is also used to perform the administrative operation of registering a new Token, which is the object that stores the RFID ID plus additional information such as issuer, contract ID, and validity. Once done, *MaeVe* can accept or deny authentication requests from EVerest sent through OCPP 2.0.1.

Listing 4.16: Project Structure Overview

```
$ curl http://localhost:9410/api/v0/token -H 'content-type: application
    /json' -d '{
  "countryCode": "GB",
  "partyId": "TWK",
  "type": "RFID",
  "uid": "DEADBEEF",
  "contractId": "GBTWK012345678V",
  "issuer": "Thoughtworks",
  "valid": true,
  "cacheMode": "ALWAYS"
}'
```

Due to the minimal user interface, *MaeVe* is not as user friendly as *SteVe*, so many of the validations must be done through some other tools, like the EVerest logging or by capturing all the messages so that they could be analyzed and decrypted afterwards. Since these messages are more complex due to the parameters and feature profiles required by OCPP, and are also encrypted by the TLS protocol, software like *Exi-decoder* comes in place to simplify this process, allowing the content to be interpreted.

Another important aspect is the Environment as Code (EaC) used to create a Docker container, which allows *MaeVe* Environment-its dependencies, ports, and network layout-to be exposed by a single file instead of having to manually configure each service on the host system. This approach not only makes it possible to easily reproduce the environment, but also makes it consistent among different devices, which is the core point of creating a realistic architecture. For the validation tasks carried out during this thesis, it was essential to replicate modern charging infrastructures where services are deployed across different microservices and sometimes also modular and containerized.

4.3 Software Used

Throughout this chapter, the main software components and newly developed EVerest modules have been discussed, but there are a few additional tools that were

just mentioned in the paragraphs, even though they play a crucial role during the development, testing, and analysis phases of the thesis. These tools were referenced in earlier chapters but not fully described, which are Software Development Kit (SDK), Wireshark, and Exi-decoder, and the following sections provide a deeper explanation of each tool, outlining how they were used during this work.

4.3.1 Software Development Kit

In the IT world, a Software Development Kit is a set of tools used for the development and documentation of software. Each SDK differs from the others in size and tools provided, depending on the specific that it is used for, for example, an SDK for a simple library to make a calculator is going to be much smaller than the one needed by EVerest or a large project in general since it will contain less libraries, tools, and files in general. Usually, the main tools that can be found in an SDK are:

- **Compiler**: translate the source code into an executable;
- **Standard Libraries**: equipped with public interfaces called APIs;
- **Documentation**: on the programming language for which the SDK was developed;
- **Licenses**: to use when distributing programs developed with the use of the provided SDK.

But these basic tools can be extended with other tools which can enrich the SDK:

- **Other Compilers**: for different programming languages;
- **Debugger**: to help developers during the analysis of the code;
- **Versioning Programs**: to handle the versioning of the program or to interface with them;
- **Source Code Editors**: tools that help developers write code more efficiently and accurately, typically offering basic features such as syntax highlighting and code validation;
- **Integrated Development Environment (IDE)**: source code editors but with more advanced features, such as a graphic interface, autocomplete, and others.

While these two lists present the high-level structure of the SDK, a typical structure can be seen in the Listing 4.17. In the main folder called `sdk` there are two folders called `host` and `target`, which usually are named accordingly to the device's

architecture; for example, my host was `x86_64` since my laptop has that architecture; and there is a file called `environment-setup-target`, which is used to set all the environment variables used by the cross-compiler to translate the code for the target architecture.

The two directory inside the `sysroots` folder are:

– **Host**: containing all the tools needed by your computer to build programs for the `target` device, in this case the EVSE STM32. Everything in this folder runs on the development machine, including compilers, build tools, and the libraries these tools depend on in order to generate the cross-compiled version of the program.

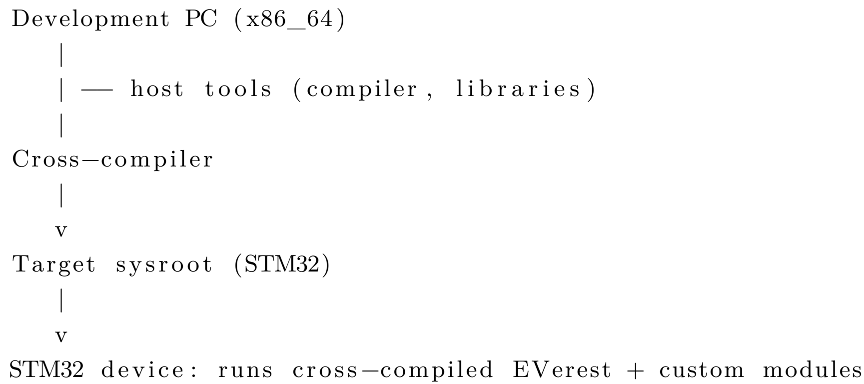
– **Target**: represents the directory structure of the device, in our case the `STM32`, and all the files contained in this folder are compiled for its architecture and are used by the cross-compiler to ensure that the version of the program runs correctly on the target hardware.

Listing 4.17: Typical Structure of an SDK

```
sdk/
`-- environment-setup-target   # Environment variables needed to use
|                              # the cross-toolchain
`-- sysroots/
    `-- host/
    |   |-- usr/               # Host executables and utilities
    |   |-- lib/               # Libraries used to translate the code
    |   `-- var/               # Host toolchain metadata and
    |                          # configuration files
    `-- target/
        |-- bin/               # Target binaries
        |-- boot/              # Bootloader- and kernel-related files
        |-- usr/               # Libraries, binaries, headers
        |-- sbin/              # System binaries
        |-- sys/               # Kernel interface
        |-- etc/               # Configuration files
        |-- var/               # State/configuration directories
        `-- lib/               # Libraries for target
```

Abinsula [7] provided me with a simple *SDK* to cross-compile programs for the provided board. However, it lacked several dependencies in order to cross-compile EVerest and the new custom modules. Before starting the development of the custom modules, I had to add the missing libraries and test whether EVerest could be built for the provided board. After adding the missing dependencies, I was able to use the custom SDK to build EVerest and the new modules for the target device. A simplified version of the cross-compilation workflow process, from the development machine to the STM32, is illustrated in Listing 4.18.

Listing 4.18: Cross-Compilation Workflow



```
Development PC (x86_64)
   |
   | -- host tools (compiler, libraries)
   |
Cross-compiler
   |
   v
Target sysroot (STM32)
   |
   v
STM32 device: runs cross-compiled EVerest + custom modules
```

But why did I use a custom SDK?

Pionix [41] provides a Yocto Layer in the *meta-everest* repository [42], with many different versions and releases, which could be used to rapidly integrate EVerest software, but was not compatible with the provided device, as it could not support newer *Ubuntu* versions. Making a new Yocto Layer from scratch was too much time expensive without knowing if it would actually work on the board, that is why I decided to cross-compile the missing dependencies first and leave the custom layer as a non-urgent task for later on.

4.3.2 Other Useful Tools

In order to better understand the flow and content of the messages exchanged between the EVSE and EV, I used two packet-analyzer software; the first one is the open-source *Wireshark*, while the second one is licensed by *Trialog* and it is called *Exi-decoder*.

Although both tools serve the general purpose of analyzing communication packets, the features they provide are quite different. *Wireshark* shows the messages, raw as they are, while being exchanged in real-time, making it possible to apply filters on the source and/ or destination address, protocol used, and many others.

On the other hand, *Exi-decoder* does not offer a real-time packet sniffer, it works by importing previously captured traces and translating them, especially when they contain EXI-encoded messages, making it play a very crucial role for testing the communication, since *Wireshark* does not natively decrypt those messages. When they were encrypted using TLS, some other methods had to come in play, for example, a man-in-the-middle or logging using one of the two sides of the communication to make them print in clear text the content.

In addition to *Wireshark* and *Exi-decoder*, I used *tcpdump*, which allowed me to capture every packet exchanged over the network interface *plc1*, including several different protocols such as TCP, TLS, HomePlug, ICMPv6, and UDP. With the command below, *tcpdump* captured all the messages exchanged over the PLC interface and saved them in a file with *pcap* extension, which could be analyzed by the other two tools.

```
sudo tcpdump -i plc1 -w file_name.pcap
```

In the following paragraphs, an overview of the two tools is provided along with some screenshots during the debugging and testing of the EVerest software, showcasing what they were used for.

4.3.2.1 Wireshark

Wireshark is one of the most widely used network protocol analyzers and played a fundamental role throughout the debugging and testing activities of this project. It is mainly used to visually inspect network traffic, which can be either in real-time or through a file, like *pcap*, where network messages are written by external tools like *tcpdump*. When a *pcap* file is imported into *Wireshark*, all the messages are loaded and displayed ordered by their number, which makes them readable sequentially, as illustrated in Figure 4.9. However, the content of each frame is not in clear text, but is shown in *raw* format, meaning that a sequence of bytes represents the message, as in Figure 4.10.

Wireshark's filters were essential during debugging and analysis since by applying them I managed to isolate the different sections, for example, the Handshake, the SLAC process, or the TCP charging process for better debugging them. Additionally, *Wireshark* provides, as a feature, the possibility to follow TCP or TLS streams enabling to fully inspect the messages, which is also crucial to export the trace to *Exi-decoder* for translating the content of each frame.

Even though the clear text of each frame is not available in this tool, especially when they are protected by the TLS protocol, it is still possible to get a detailed visibility into the overall communication flow. It is possible to follow the sequence of messages exchanged between the EV and the EVSE, inspect the structure of the TLS handshake, and verify the exchange of certificates. However, starting from this point, since the communication becomes secure, the messages will be encrypted and appear in the UI section *info* as *Application Data*, but their general information will still be available, such as number, timestamp, source, destination, and protocol version, which makes it a little bit easier to understand if those are getting received and sent.

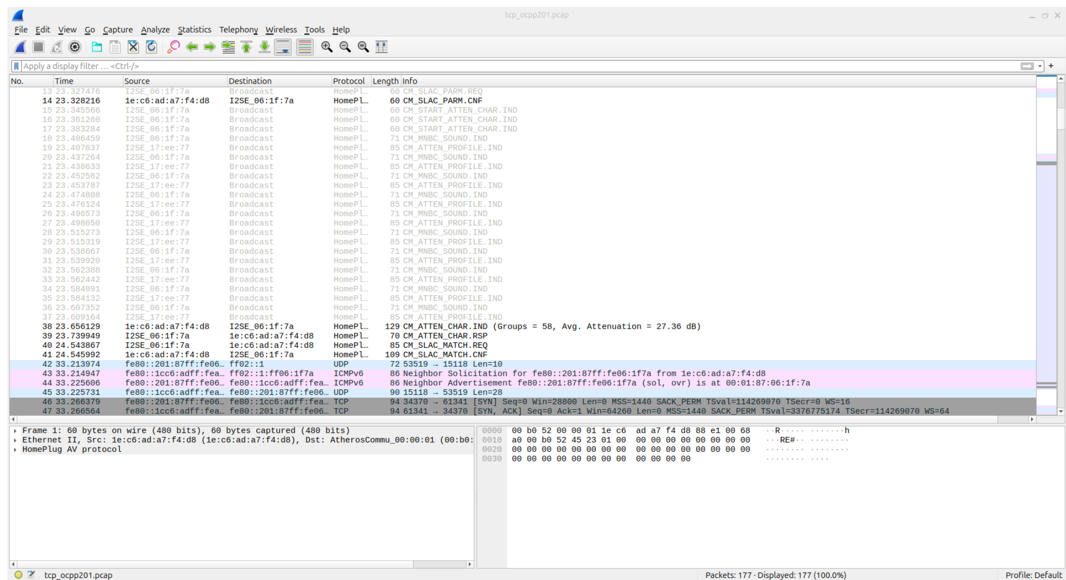


Figure 4.9: Wireshark UI

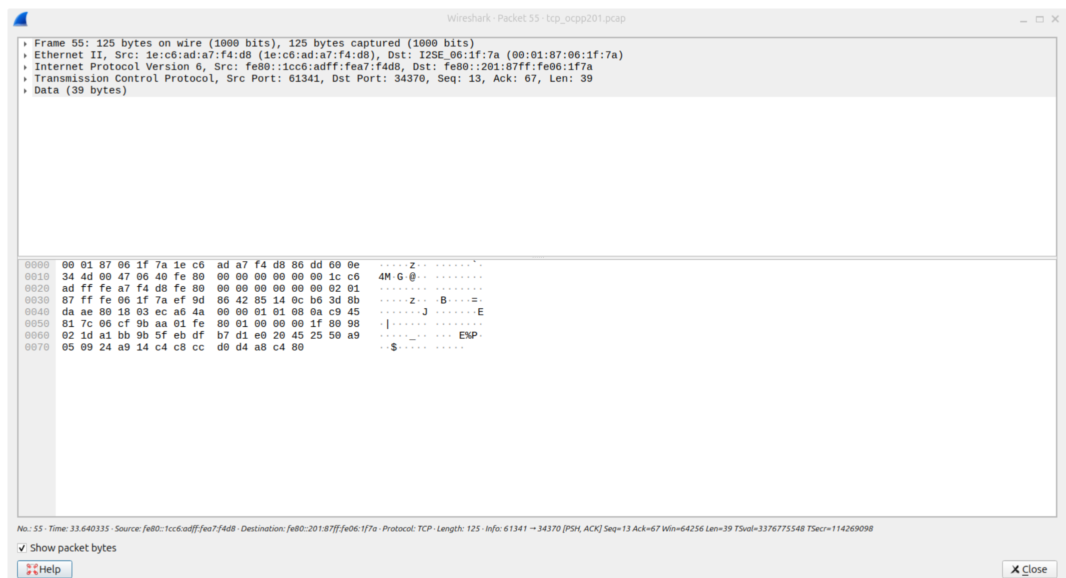


Figure 4.10: Wireshark Raw Message

4.3.2.2 Exi-decoder

The *Exi-decoder* software is used to translate messages exchanged using Efficient XML Interchange (EXI), which is a binary XML format used to encode those files in a binary data format that can be sent over the network. This type of representation is widely used in the ISO 15118 protocol for high-level communication between EVSE and EV. Since these types of messages are not readable in their raw format and *Wireshark* does not provide a decoder that makes them understandable by a human, *Exi-decoder* can take the content and make the content clear as illustrated in Listing 4.20.

In the Listing, an authorization request takes place and starts with the EV asking the EVSE to be authorized for the charging, then the charging station answers, but since the authorization is not done yet it sets the *EVSEProcessing* field to `1`. Only later, after a second request from the EV, the EVSE sends back a *Finished* data frame, setting the *EVSEProcess* value to `0`, which means that the authorization succeeded.

Listing 4.19: Exi Message

```
> 01fe80010000000d8098021b83dbde9b5f5fd3d008

< 01fe80010000000f8098021b83dbde9b5f5fd3d0100200

> 01fe80010000000d8098021b83dbde9b5f5fd3d008

< 01fe80010000000f8098021b83dbde9b5f5fd3d0100000
```

Listing 4.20: Decoded Exi Message

```
> AuthorizeReq
    SessionId: "6E0F6F7A6D7D7D4F"

< AuthorizeRes Ongoing
    SessionId: "6E0F6F7A6D7D7D4F"
    ResponseCode: 0
    EVSEProcessing: 1

> AuthorizeReq
    SessionId: "6E0F6F7A6D7D7D4F"

< AuthorizeRes Finished
    SessionId: "6E0F6F7A6D7D7D4F"
    ResponseCode: 0
    EVSEProcessing: 0
```

Before getting here, it is important to retrieve a clean raw representation of the messages that can be analyzed through *Exi-decoder*. *Wireshark* provides a feature called *Follow TCP Stream*, which opens a new window displaying all raw data with alternating colors between frames to make their boundaries easier to identify when a TCP packet from the flow of interest is selected.

The underlying window is illustrated in Figure 4.11, where it is also possible to change the type representation under the section *Show data as*, but since the ISO 15118 protocol is not an option, I was forced to import the data into *Exi-decoder*. From this feature, it is also possible to find sequences of hexadecimal numbers, which can be extremely useful to expert developers, who can already understand the content

without having to translate the content.

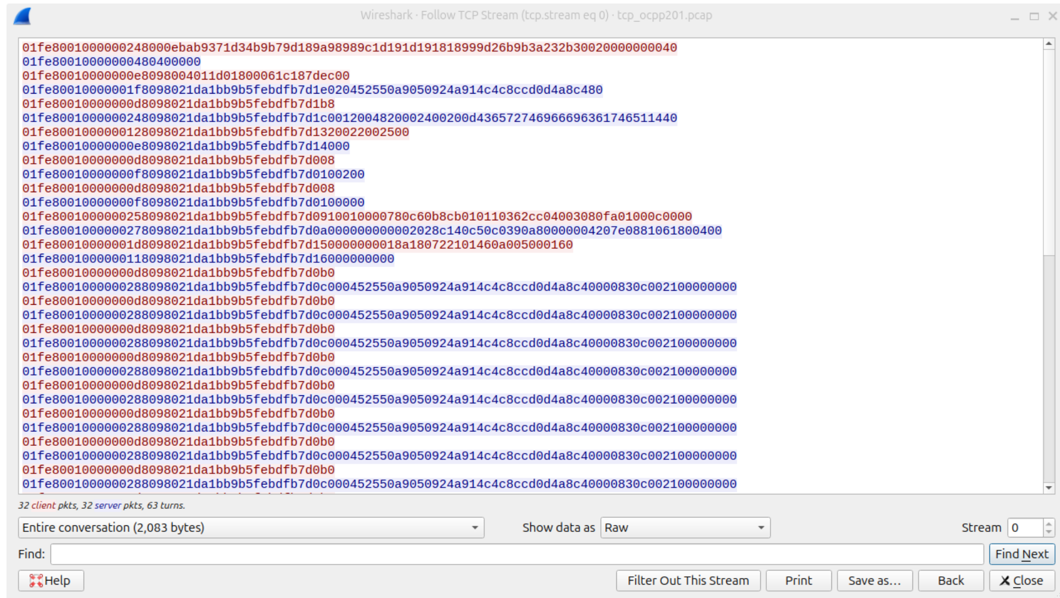


Figure 4.11: Wireshark Follow TCP Stream

Chapter 5

Results

In this chapter, a distinction is made between *testing* and *validation*. Testing activities were carried out during development to identify and fix integration issues and incorrect behaviors. In contrast, validation activities were performed once the implementation was completed, with the goal of verifying that the final system met the functional and performance requirements defined for the project.

5.1 Test Environment

The Test Environment is a system composed of all the devices, applications, data, and network needed to emulate a real scenario, to test the software without affecting the production environment. The main goal of the Test Environment is to identify bugs and evaluate the maturity of the product under test, to ensure that the application can be released.

For the purpose of this work the Test Environment is composed of: the EVerest software with a chosen configuration; an open-source CSMS from the ones mentioned in Chapter 4; and the *Trialog* EV emulator. In Figure 5.1, an illustration of what the connection of the three represents. Furthermore, it shows some of the main components of each actor to better understand their roles.

In the "*EVSE Software*" block of Figure 5.1, there is a complete configuration of the environment that can be used in a real EVSE implementation. Within this configuration, it is possible to support the simultaneous charging of multiple vehicles through the *Energy Sink*. This feature was not part of this work, due to the absence of a second charging port, but it can be easily enabled through the configuration section as illustrated in Listing 5.1. EVerest already provides examples about this type of connection so it can be replicated and it is scalable.

Listing 5.1: Energy Sink Configuration

```
energy_manager:
```

```
    connections:
      energy_trunk:
        - implementation_id: energy_grid
          module_id: grid_connection_point
    module: EnergyManager
grid_connection_point:
    mapping:
      module:
        evse: 0
    config_module:
      fuse_limit_A: 32
      phase_count: 1
    connections:
      energy_consumer:
        - implementation_id: energy_grid
          module_id: evse_manager_1_api_sink
    module: EnergyNode
evse_manager_1_api_sink:
    module: EnergyNode
    mapping:
      module:
        evse: 1
    config_module:
      fuse_limit_A: 32.0
      phase_count: 1
    connections:
      energy_consumer:
        - module_id: evse_manager_1_ocpp_sink
          implementation_id: energy_grid
  evse_manager_1_ocpp_sink:
    module: EnergyNode
    mapping:
      module:
        evse: 1
    config_module:
      fuse_limit_A: 32.0
      phase_count: 1
    connections:
      energy_consumer:
        - module_id: connector_1
          implementation_id: energy_grid
```

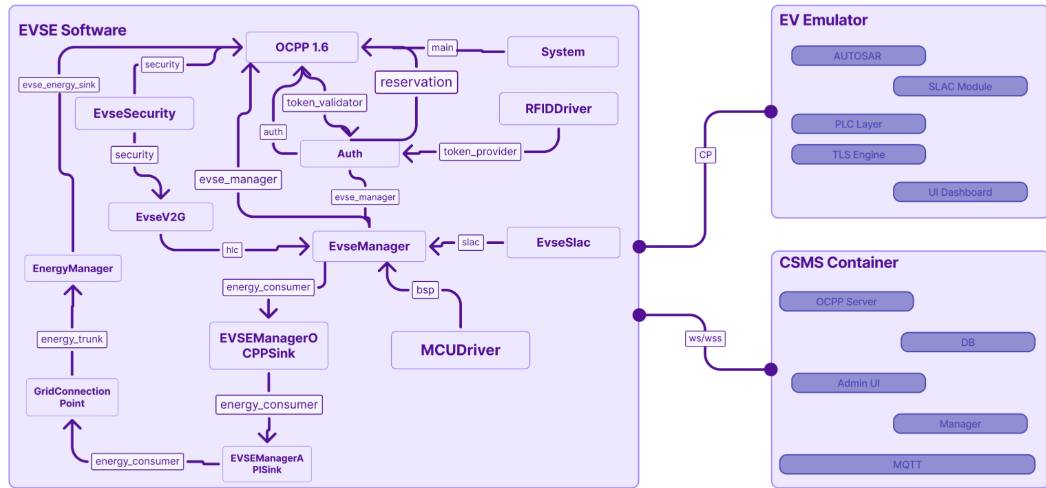


Figure 5.1: Test Environment Overview

5.2 Validation Workflow

To validate the final implementation of the customized EVerest software within the defined Test Environment, a structured validation workflow was adopted. The overall validation workflow adopted in this work is illustrated in Figure 5.2. The diagram summarizes the sequence of steps followed during the validation process, highlighting the successful execution flow from system initialization to communication analysis and final verification against protocol specifications. Defining a clear sequence of steps was fundamental to ensure repeatability of the tests, simplify debugging activities, and allow consistent analysis of the communication between the EV, the EVSE, and the CSMS.

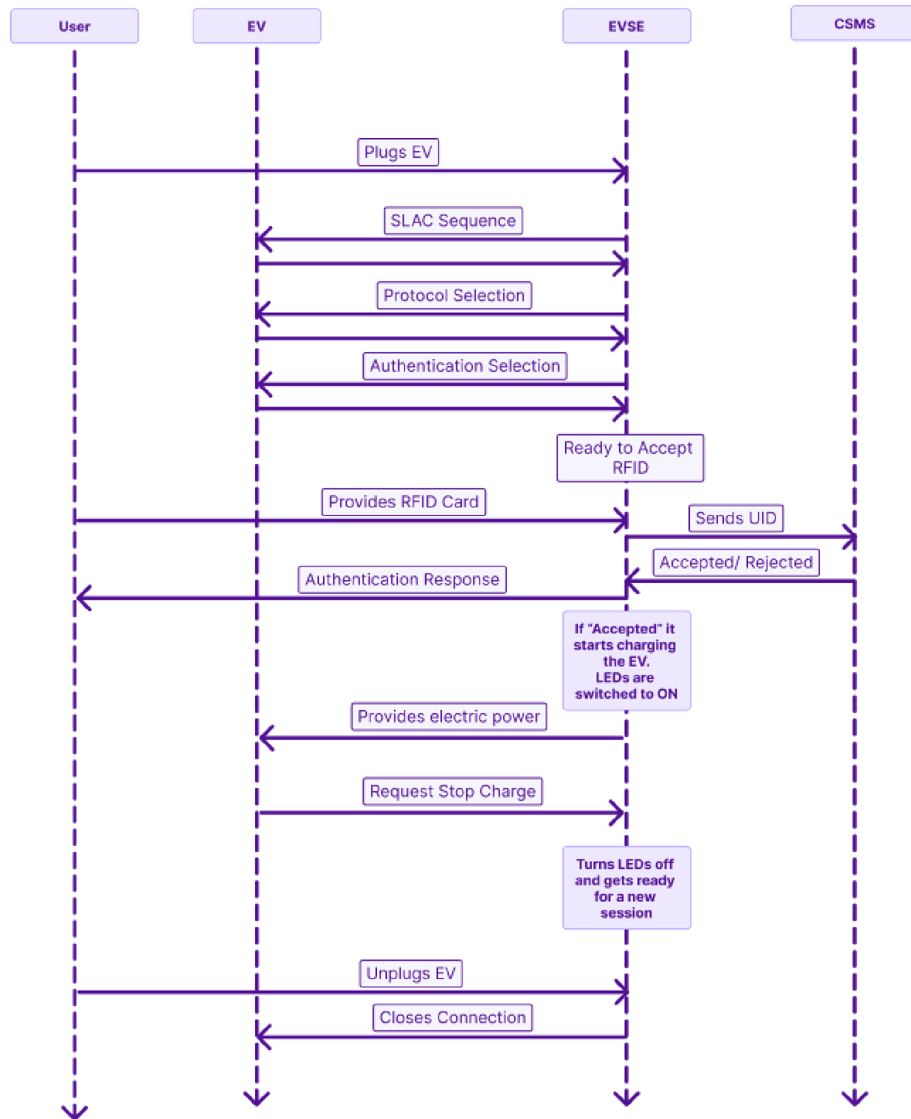


Figure 5.2: High-level validation workflow adopted for testing the customized EVerest EVSE implementation.

The workflow started with the initialization of all the components involved in the Test Environment. First, the CSMS backend was started to guaranty its availability for incoming connections from the charging station. Then, the MQTT Server was started to ensure communication within the EVerest software. Once EVerest was running, the correct connection to the CSMS was verified.

After the EVSE was fully operational, the *Trialog* EV emulator was started. The emulator was configured to simulate an electric vehicle that supports ISO 15118 high-level communication over the PLC interface, AC charging, and TCP or TLS protocol. From this point on, a charging session could be initiated through the provided web interface, triggering the full communication flow between the EV and the EVSE.

The execution of a charging session can be decomposed into different sub-processes that work for a common goal, and each corresponds to a specific control phase. Each sub-process involved the analysis of the interaction between hardware and software, allowing verification of both the logic of the components and the physical layer behavior.

Throughout the test session, the network traffic on the PLC interface was captured using *tcpdump*. All exchanged packets were saved in files with the *pcap* extension, allowing offline inspection and analysis. The captured traffic included multiple protocols, such as TCP, TLS, HomePlug Green PHY, and ISO 15118 application messages.

After completion of the test session, the captured traces were analyzed using *Wireshark*, to inspect the overall communication flow, verify the sequence of messages, and analyze specific phases of the charging process by applying filters, such as the selection of a specific protocol. For messages encoded using the EXI format, the raw data was extracted from the traces and imported into *Exi-decoder*, which allowed the translation of binary payloads into a human-readable format.

Finally, the decoded messages and the observed behavior of the system were compared against the expected behavior defined by the ISO 15118 and OCPP specifications, as well as the intended customization of the EVerest software. This step made it possible to identify configuration issues or unexpected behaviors and iteratively adjust the system setup.

In conclusion, adopting a consistent test workflow composed of initialization, execution, traffic capture, analysis, and validation allowed a systematic evaluation of the EVerest software within the Test Environment.

5.3 Testing Activities During Development

During the development phase, several testing activities were carried out in this project to verify the correct integration of custom software components within the EVerest stack. In addition to functional correctness, particular attention was given to the timing aspects of the hardware–software interaction. The tests were therefore designed not only to verify that hardware-triggered events, authorization mechanisms, and communication protocols were correctly handled throughout the charging process but also to assess the temporal coordination between physical-layer events and software-level responses.

The first set of tests focused on the correct initialization and configuration of the EVerest software. These tests verified that the selected modules could be correctly instantiated, that custom modules were properly loaded through their *Manifest* files, and that all required interfaces were correctly connected. This verification was

performed by analyzing the logs produced by the EVerest software during startup, which explicitly report the initialization status of each module and highlight possible failures or missing dependencies.

Another important category of tests targeted the integration of the custom *MCU Driver* module. These tests verified the correct exchange of information between the EVSE software and the MCU through the *sdbus* interface. In particular, changes in Control Pilot state, contactor state, and AC charging limits were monitored to ensure that hardware events were correctly translated into EVerest internal events and propagated to the relevant modules, such as *EvseManager*. The EVerest logs come into play also for this section, since it prints out when a new charging state is reached, so it is easy to understand when the communication between the two is failing.

Tests related to the charging state machine were also performed. These tests validated that state transitions within *EvseManager* correctly triggered hardware actions, such as enabling or disabling the PWM signal and setting the *5% duty cycle* required by the ISO 15118 protocol. The correct handling of events such as *pwm_on* and *pwm_off* was verified by observing both software logs and hardware responses.

Authorization-related tests were conducted using the custom *RFID Driver* module. These tests verified that RFID card detection, removal, and error events were correctly received from the hardware and published within the EVerest ecosystem. The interaction between the RFID Driver, the *Auth* module, and the token validation logic was analyzed to ensure that authorization requests were correctly triggered and handled. In addition, these tests were useful to observe and validate the interaction between the EVSE software and an open-source CSMS that emulates the behavior of a real backend service, allowing verification that authorization events generated locally were correctly propagated to the CSMS and processed according to the OCPP protocol.

In general, the tests performed allowed for validation of the correctness and robustness of the customized EVerest software, with a particular focus on hardware integration, event handling, and protocol-driven behavior.

5.4 Limitation of the Test Environment

Although the tests performed allowed for a thorough validation of the customized EVerest software and its integration with external components, some limitations must be considered when interpreting the results. These limitations are mainly related to hardware availability, the use of emulated components, and the scope of the testing activities.

The first limitation concerns the hardware setup used during the tests. The available infrastructure supported the charging of a single electric vehicle at a time, which prevented the execution of tests involving multiple simultaneous charging sessions. While the EVerest configuration allows scalability through features such as the *Energy Sink*, this behavior was only validated at a configuration level and not through physical multi-EV testing.

Another limitation is related to the use of emulators to reproduce the behavior of external systems. EV behavior was simulated using the *Trialog* EV emulator, and backend communication relied on an open-source CSMS that emulates the behavior of a real charging management system. Although these tools are widely used for development and testing purposes, they cannot fully replicate all the timing, load, and edge-case conditions that may occur in a real-world deployment.

Finally, the scope of the tests primarily focused on functional correctness, integration validation, and the assessment of hardware–software interaction, including its temporal behavior during the different charging sub-procedures. While particular attention was given to verifying proper event handling and timing coordination, long-term stability tests, stress tests under high load, and comprehensive performance evaluations were not fully addressed within the scope of this work due to the available resources.

5.5 Discussion of Results

The validation activities performed provided a comprehensive understanding of the behavior and maturity of the customized EVerest software within the defined Test Environment. Overall, the system demonstrated correct initialization, integration of custom modules, and proper handling of communication and authorization processes.

5.5.1 Validation of Custom Modules

The logs produced by EVerest during the initialization confirmed that all selected modules, including the custom *MCUDriver* and *RFIDDriver*, were correctly instantiated. Each module reported its initialization status, and no critical errors or missing dependencies were observed. The logs were particularly useful for verifying that the custom modules loaded their respective *Manifest* files correctly and that all required interfaces were connected. As shown in the logs in Listing 5.4, once all the different modules of the configuration are configured and initialized correctly, the software is considered healthy and can run.

Listing 5.2: EVerest Logs Messages During Initializations

```
...
[INFO] abi_mcu_driver       : Module abi_mcu_driver initialized [1476ms
    ]
[INFO] abi_token_provider   : Module abi_token_provider initialized
    [1485ms]
[INFO] iso15118_charger     : Module iso15118_charger initialized [1501
    ms]
[INFO] connector_1          : Module iso15118_charger initialized
    [1623]
[INFO] manager              : Clearing retained topics published by
    manager during startup
[INFO] manager              : All modules are initialized. EVerest up
    and running [2865ms]
```

5.5.2 SLAC Communication

SLAC communication was first tested with a minimal EVSE configuration where only the custom *MCUDriver*, the *EvseSlac*, and the *EvseManager* modules were present. Only when all the bugs were fixed was it implemented in more complex configurations. As illustrated in Figure 5.3, the sequence is one to one to the theoretic Figure C.1, which means that the implementation of the custom *MCUDriver* was successful. Additionally, in Appendix C there is a more in depth view of the SLAC sequence and a brief description for each of the messages in the *Wireshark* snippet.

Listing 5.3: EVerest Logs Before SLAC Sequence

```
...
[INFO] abi_mcu_driver    : evse_board_supportImpl::
    handle_ac_set_overcurrent_limit_A()
[INFO] abi_mcu_driver    : Received CP state change: B
[INFO] abi_mcu_driver    : evse_board_supportImpl::handle_allow_power_on
    ()
[INFO] slac:EvseSlac     : Entering Matching state, waiting for
    CM_SLAC_PARM_REQ
[INFO] abi_mcu_driver    : evse_board_supportImpl::handle_allow_power_on
    ()
[INFO] connector_1       : EVSE IEC Session Started: EVConnected
[INFO] connector_1       : EVSE IEC Set PWM On (5.00%) took 0 ms
[INFO] abi_mcu_driver    : evse_board_supportImpl::handle_allow_power_on
    ()
[INFO] abi_mcu_driver    : MPU->MCU | setFivePercentCpDutyCycle
[INFO] connector_1       : EVSE IEC EIM Authorization Received
[INFO] connector_1       : EVSE IEC Transaction Started (0 kWh)
[INFO] connector_1       : EVSE IEC AC mode, HLC enabled(5percent),
    matching already started. Go through t_step_X1 and disable 5 percent
    .
[INFO] connector_1       : EVSE IEC Charger state: Wait for Auth->
    T_step_X1
[INFO] connector_1       : EVSE IEC Enter T_step_X1
[INFO] connector_1       : EVSE IEC Setp PWM Off
```

```
[INFO] abi_mcu_driver    : Received CP state change: B
[INFO] abi_mcu_driver    : evse_board_supportImpl::
    handle_allow_power_off()
...
```

```
13 28.684522   I2SE_06:1f:7a       Broadcast           HomePl…   60 CM_SLAC_PARM.REQ
14 28.685284   1e:c6:ad:a7:f4:d8   I2SE_06:1f:7a       HomePl…   60 CM_SLAC_PARM.CNF
15 28.701578   I2SE_06:1f:7a       Broadcast           HomePl…   60 CM_START_ATTEN_CHAR.IND
16 28.718816   I2SE_06:1f:7a       Broadcast           HomePl…   60 CM_START_ATTEN_CHAR.IND
17 28.742850   I2SE_06:1f:7a       Broadcast           HomePl…   60 CM_START_ATTEN_CHAR.IND
18 28.762818   I2SE_06:1f:7a       Broadcast           HomePl…   71 CM_MNBC_SOUND.IND
19 28.762863   I2SE_17:ee:77       Broadcast           HomePl…   85 CM_ATTEN_PROFILE.IND
20 28.787569   I2SE_06:1f:7a       Broadcast           HomePl…   71 CM_MNBC_SOUND.IND
21 28.787614   I2SE_17:ee:77       Broadcast           HomePl…   85 CM_ATTEN_PROFILE.IND
22 28.813313   I2SE_06:1f:7a       Broadcast           HomePl…   71 CM_MNBC_SOUND.IND
23 28.815301   I2SE_17:ee:77       Broadcast           HomePl…   85 CM_ATTEN_PROFILE.IND
24 28.831723   I2SE_06:1f:7a       Broadcast           HomePl…   71 CM_MNBC_SOUND.IND
25 28.831768   I2SE_17:ee:77       Broadcast           HomePl…   85 CM_ATTEN_PROFILE.IND
26 28.855323   I2SE_06:1f:7a       Broadcast           HomePl…   71 CM_MNBC_SOUND.IND
27 28.856631   I2SE_17:ee:77       Broadcast           HomePl…   85 CM_ATTEN_PROFILE.IND
28 28.878288   I2SE_06:1f:7a       Broadcast           HomePl…   71 CM_MNBC_SOUND.IND
29 28.879490   I2SE_17:ee:77       Broadcast           HomePl…   85 CM_ATTEN_PROFILE.IND
30 28.899313   I2SE_06:1f:7a       Broadcast           HomePl…   71 CM_MNBC_SOUND.IND
31 28.900510   I2SE_17:ee:77       Broadcast           HomePl…   85 CM_ATTEN_PROFILE.IND
32 28.921437   I2SE_06:1f:7a       Broadcast           HomePl…   71 CM_MNBC_SOUND.IND
33 28.922670   I2SE_17:ee:77       Broadcast           HomePl…   85 CM_ATTEN_PROFILE.IND
34 28.943347   I2SE_06:1f:7a       Broadcast           HomePl…   71 CM_MNBC_SOUND.IND
35 28.944499   I2SE_17:ee:77       Broadcast           HomePl…   85 CM_ATTEN_PROFILE.IND
36 28.965401   I2SE_06:1f:7a       Broadcast           HomePl…   71 CM_MNBC_SOUND.IND
37 28.966694   I2SE_17:ee:77       Broadcast           HomePl…   85 CM_ATTEN_PROFILE.IND
38 29.011836   1e:c6:ad:a7:f4:d8   I2SE_06:1f:7a       HomePl…  129 CM_ATTEN_CHAR.IND (Groups = 58, Avg. Attenuation = 27.86 dB)
39 29.102938   I2SE_06:1f:7a       1e:c6:ad:a7:f4:d8   HomePl…   70 CM_ATTEN_CHAR.RSP
40 29.901370   I2SE_06:1f:7a       1e:c6:ad:a7:f4:d8   HomePl…   85 CM_SLAC_MATCH.REQ
41 29.902043   1e:c6:ad:a7:f4:d8   I2SE_06:1f:7a       HomePl…  109 CM_SLAC_MATCH.CNF
```

Figure 5.3: Slac Messages From Wireshark

5.5.3 RFID Validation

In this case, the EVerest logs show more details than the messages retrieved with *tcpdump*, since they can also allow the *RFIDDriver* to be debugged by checking the card *UID* and other information, which are illustrated and translated in the Listing 5.5, where the details are not clearly displayed for safety reasons.

In this case, the EVerest logs proved to be more useful for development and debugging purposes than the messages captured with *tcpdump*. The internal logs provide a clearer and more detailed view of the system behavior, including information from the *RFIDDriver* such as the detected card *UID* and related metadata. These details cannot be seen in the *tcpdump* files where just the *SessionId* and *ResponseCode* are provided.

Listing 5.4: EVerest Logs Messages During Initializations

```
...
[info] abi_token_provider    : Received RFID tag detected: <uid>
[info] auth                  : Received new token:
    {
        "authorization_type": "RFID",
        "connectors": [ 1 ],
        "id_token": {
            "type": "Local",
            "value': "[redacted] hash: XXXXXXXXXXXXXXXX"
        }
    }
[info] auth                  : Result for token: [redacted] hash:
   XXXXXXXXXXXXXXXX: <status>
```

```
[info] auth                  : Providing authorization to evse#<id>
[info] ocpp                  : Authorize.req for token: <uid>
[info] ocpp                  : Authorize.conf for token: <uid> (status:
    Accepted)
[info] connector_1           : Ready to start charging
...
```

Listing 5.5: RFID Messages from Tcpdump

```
...
> PaymentServiceSelectionReq EIM
     SessionId: "009e2f4ffeee6f57"
     SelectedPaymentOption : 1
     SelectedServiceList : {
          SelectedService 0 : {
               ServiceID : 2
          }
          SelectedService 1 : {
               ServiceID : 1
          }
     }

> None
     SessionId: "009e2f4ffeee6f57"

> AuthorizeReq
     SessionId: "009e2f4ffeee6f57"

< AuthorizeRes Ongoing
     SessionId: "009e2f4ffeee6f57"
     ResponseCode: 0
     EVSEProcessing: 2

> AuthorizeReq
     SessionId: "009e2f4ffeee6f57"

< AuthorizeRes Finished
     SessionId: "009e2f4ffeee6f57"
     ResponseCode: 0
     EVSEProcessing: 0
...
```

In this example, *MaeVe* accepted the RFID card and, by setting the *ResponseCode* to `0`, it allows the start of the charging process.

5.5.4 Start and Stop Charging

Similarly to other tests, the start and stop processes could be analyzed at first through the *EVerest* software logs and to gain a deeper knowledge through the TCP messages retrieved during the charging session. Those two are essential for the correctness of the software since the two combined allow the charging process to begin and end.

Furthermore, it is important to evaluate the temporal behavior of these procedures, specifically the time required for the EVSE to gather all necessary information and transition to an active charging state, as well as the time needed to return to a ready state after the session has ended.

5.5.4.1 Start Routine

The log in Listing 5.6 shows the transition from the *SLAC* matching phase to the initialization of the ISO 15118 communication. Once the *SLAC* matched and the *D-LINK* is read, the EVSE disables the *PWM* signal and then re-enables it with the *5% duty cycle*, as required by the standard to indicate readiness for high-level communication. These actions are sent to the hardware through the custom *MCU Driver*, confirming the correctness of the integration between software and board.

Immediately after the *iso15118_charger* module receives the first packets from the EV, the protocol selection routine starts between the two, and ISO 15118-2 is chosen. From this point, *EvseManager* starts interacting with the vehicle for the setup of the charging process, where the EVerest module sends some requests to the "*CAR*" which provides the responses needed. Just before turning on the power, the custom driver receives a request to close the relay, indicating that the procedure has been successfully executed.

Listing 5.6: EVerest Logs Start Routine

```
...
[INFO] slac:EvseSlac      : Entered Matched state
[INFO] connector_1        : EVSE IEC SLAC MATCHED
[INFO] connector_1        : EVSE IEC D-LINK_READY (true)
[INFO] connector_1        : EVSE IEC Exit T_step_X1
[INFO] connector_1        : EVSE IEC Set PWM Off
[INFO] abi_mcu_driver     : evse_board_supportImpl::
    handle_allow_power_off()
[INFO] abi_mcu_driver     : MPU->MCU | setFivePercentCpDutyCycle
[INFO] abi_mcu_driver     : evse_board_supportImpl::handle_allow_power_on
    ()
[INFO] iso15118_charger : Receuved packet from [...] with security 0x10
     and protocol 0x00
[INFO] iso15118_charger : SDP requested NO-TLS, announcing NO-TLS
[INFO] iso15118_charger : sendto([...]) succeded
[INFO] iso15118_charger : Incoming connection on plc1 from [...]
[INFO] iso15118_charger : Started a new TCP connection thread
[INFO] iso15118_charger : Selected protocol: ISO15118-2
[INFO] connector_1        : CAR ISO V2G SupportedAppProtocolReq
[INFO] connector_1        : EVSE ISO V2G SupportedAppProtocolRes
[INFO] connector_1        : CAR ISO V2G ServiceDiscoveryReq
[INFO] connector_1        : EVSE ISO V2G ServiceDiscoveryRes
[INFO] iso15118_charger : Selected service id 2 found
```

```
[INFO] iso15118_charger : SelectedPaymentOption: ExternalPayment
[INFO] connector_1      : CAR ISO V2G PaymentServiceSelectionReq
[INFO] connector_1      : EVSE ISO V2G PaymentServiceSelectionRes
[INFO] connector_1      : CAR ISO V2G AuthorizationReq
[INFO] connector_1      : EVSE ISO V2G AuthorizationRes
[INFO] connector_1      : CAR ISO V2G AuthorizationReq
[INFO] connector_1      : EVSE ISO V2G AuthorizationRes
[INFO] iso15118_charger : Parameter-phase started
[INFO] iso15118_charger : Selected energy transfer mode:
    AC_single_phase_core
[INFO] connector_1      : CAR ISO V2G ChargeParameterDiscoveryReq
[INFO] connector_1      : EVSE ISO V2G ChargeParameterDiscoveryRes
[INFO] iso15118_charger : Waiting for contactor is closed
[INFO] connector_1      : CAR ISO AC HLC Close contactor
[INFO] abi_mcu_driver   : Received CP state change: C
[INFO] connector_1      : CAR IEC Event CarRequestedPower
[INFO] connector_1      : EVSE IEC Charger state: PrepareCharging->
    Charging
[INFO] abi_mcu_driver   : evse_board_supportImpl::handle_allow_power_on
    ()
[INFO] abi_mcu_driver   : Received Relais state closed: true
[INFO] connector_1      : EVSE IEC Event PowerOn
...
```

5.5.4.2 Stop Routine

The log in Listing 5.7 shows the correct termination of the charging session, starting from an EV-side request to stop power delivery. The process begins with the *ISO 15118* command to open the contactor, which is received by the EVSE and propagated to the hardware through the custom *MCUDriver*. The subsequent change of the Control Pilot state from *C* to *B* confirms that the vehicle is no longer requesting power.

After the contactor is opened, the EVSE switches to the power-off state and the hardware feedback confirms that the relay is no longer closed. The *iso15118_charger* module then completes the high-level communication by closing the TCP connection and handling the *SessionStop* message exchange.

Once the communication channel is closed, the Control Pilot switches to state *A*, indicating that the EV has been disconnected. The EVSE disables the PWM signal and finalizes the charging session, updating its internal state from "charging" to "finished". The sequence concludes with the emission of the *Transaction Finished* event, confirming that the charging process was cleaned up and correctly terminated.

Listing 5.7: EVerest Logs Stop Routine

```
...
[INFO] connector_1      : CAR ISO AC HLC Open contactor
[INFO] abi_mcu_driver   : Received CP state change: B
```

```
[INFO] abi_mcu_driver   : evse_board_supportImpl::handle_allow_power_on
    ()
[INFO] abi_mcu_driver   : Received Relais state closed: false
[INFO] connector_1      : EVSE IEC Event PowerOff
[INFO] connector_1      : CAR ISO V2G SessionStopReq
[INFO] iso15118_charger : Closing TCP Connection
[INFO] connector_1      : CAR ISO V2G SessionStopRes
[INFO] abi_mcu_driver   : Received CP state change: A
[INFO] abi_mcu_driver   : evse_board_supportImpl::
    handle_allow_power_off()
[INFO] abi_mcu_driver   : Stopping charge on connector 1
[INFO] connector_1      : EVSE ISO D-LINK_READY (false)
[INFO] connector_1      : CAR IEC Event CarRequestedStopPower
[INFO] connector_1      : EVSE IEC Charger state: Car Paused->Finished
[INFO] connector_1      : EVSE IEC Transaction Finished: EVDisconnected
...
```

5.5.5 Performance Validation

For the system performance evaluation, I took into account the times for some of the main sub-processes of the entire charging cycle. These measurements provide insight into the responsiveness of the system.

- ***Initialization Time***: corresponds to the time required for the EVSE to read its configuration file and complete the startup of all modules, marking the point at which the components are ready to operate. The data are taken from different configuration files, each with a different number of modules, starting from a simpler configuration with 7 modules and finishing with a more complex one with 17.

  The number of modules varied according to the features and to evaluate the scalability of the system, for example, the configuration with 7 modules does not support authentication through an RFID card, and so the custom module *RFIDDriver* and the OCPP protocol are not initialized; while the largest configuration not only supports the OCPP protocol and RFID authentication, but also has some modules to store errors and exposes some APIs for external connections.

  Table 5.7 summarizes the data retrieved but the test runs, and for the three different configurations used against the EVSE, the table reports the *Average Time* and the *Standard Deviation*. Whereas Figure **??** summarizes the different runs with their final results.

- ***SLAC Sequence***: corresponds to the elapsed time to compute all the sequence of messages between EV and EVSE. The message "*D-LINK_READY (true)*"

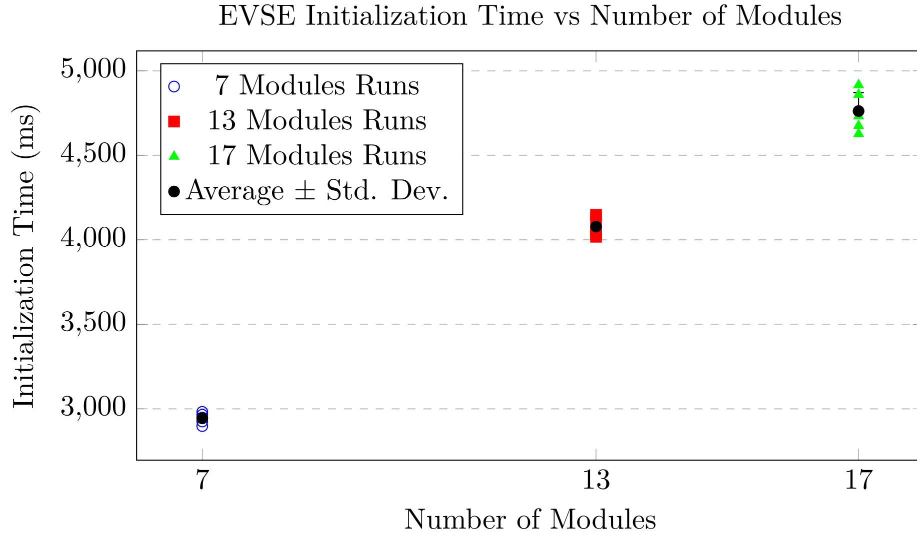


Figure 5.4: Initialization Time for different numbers of modules, showing individual runs and averages with standard deviation.

defines the finish line for this sub-process.

This metric is used to analyze communication between the vehicle and the charging point. Table **??** reports the average *SLAC Sequence* time and its standard deviation across multiple test runs.

- ***RFID Detection***: corresponds to the time elapsed between the physical detection of an RFID card by the hardware to the instant the appropriate module receives it for analysis.

  In this benchmark, card ID processing and validation are not taken into account, as they depend on external factors, such as network latency. The goal here is to analyze the hardware-software interaction and communication capabilities.

- ***Start Stop Performance***: The *Start Sequence* measures the elapsed time from the completion of SLAC matching to the activation of power delivery. This includes the re-initialization of the PWM signal, the selection and setup of the ISO 15118-2 protocol, authorization checks, and the closing of the hardware relay via the custom *MCUDriver*, marking the point at which charging begins.

  The *Stop Sequence* captures the time from an EV-initiated stop request to the complete termination of the charging session. It includes opening the contactor, transitioning the Control Pilot state, closing the TCP communication channel, and updating the EVSE internal state from "charging" to "finished," culminating in the emission of the *Transaction Finished* event.

Table 5.1 summarizes the average durations and standard deviations for the main EVSE sub-processes, including initialization, SLAC, RFID detection, and session start/stop, providing a consolidated overview of system performance.

Table 5.1: Summary of Performance Metrics for Key EVSE Sub-Processes (Average ± Std. Dev.)

Sub-Process	Average Time (ms)	Std. Dev. (ms)
Initialization (7 modules)	2948.75	56.85
Initialization (13 modules)	4082.6	68.93
Initialization (17 modules)	4735.82	124.78
SLAC Sequence	1.24	0.035
RFID Detection	5786.54	505.34
Start Sequence	11362.70	459.65
Stop Sequence	4482.34	49.82

Chapter 6

Conclusion

6.1 Maturity of Open-source EVSE

After the tests conducted and the considerations made during the implementation and evaluation of communication protocols, it is possible to assess the maturity of EVerest. The analysis focuses on how well the software handles the recharge of EV in AC mode, its support for different protocols, and the integration of different hardware components, while also highlighting its downsides, limitations, and potential improvements.

- **Strengths**

  The validation activities demonstrate several strengths of the customized EVerest software and the developed Test Environment. The system successfully initializes the EVSE platform with all required modules, including the custom *MCUDriver* and *RFIDDriver*, confirming the robustness of the modular architecture.

  A major strength lies in the support of multiple charging and communication standards within a unified framework. The ISO 15118-2, DIN 70121, and OCPP (1.6 and 2.0.1) protocols were successfully executed, including authorization handling, SLAC matching, high-level message exchange, and secure communication over TCP and TLS 1.2. The complete charging cycle was validated under both encrypted and unencrypted conditions, confirming the interoperability between different protocol layers.

  It is important to emphasize that the EV charging domain is highly regulated and technically heterogeneous. The charging infrastructure must comply with a wide range of standards and specifications, including ISO 15118 (and its future extensions), DIN 70121, OCPP, TLS security requirements, and grid-related constraints. Furthermore, real-world vehicles and CSMS implementations may not strictly follow standards, introducing additional interoperability challenges. Implementing and maintaining support for this ecosystem in a custom or monolithic architecture can rapidly become extremely complex.

In this context, the modular and scalable design of EVerest represents a significant engineering advantage. The framework provides dedicated modules for protocol handling, energy management, communication, and hardware abstraction, reducing integration complexity and facilitating compliance verification. Compared to traditional fragmented implementations—where protocol stacks, communication layers, and logging systems are often developed independently or sourced from different vendors—the EVerest architecture offers a coherent, maintainable, and extensible solution.

This architectural coherence not only simplifies development and debugging activities, but also significantly supports validation, certification processes, and multi-platform deployment. The ability to integrate custom modules without affecting the core stack further highlights the flexibility of the framework and its suitability for future protocol extensions and evolving regulatory requirements.

- **Impact on Development Time and Integration Effort**

  An additional strength of the adopted architecture is the significant reduction in development and integration time achieved through the use of EVerest.

  A previous internal project carried out by *Abinsula*, based on a largely proprietary software architecture and independently integrated protocol stacks, required approximately two years of development before achieving a fully operational EVSE capable of supporting DIN 70121, ISO 15118 communication, and backend connectivity.

  In contrast, the present project, built on top of the EVerest framework, reached a fully functional EVSE implementation within approximately seven months. This time frame included adapting the framework to the target hardware platform, implementing the custom *MCUDriver* and *RFIDDriver* modules, and integrating SLAC, ISO 15118, and OCPP communication flows.

  To better illustrate the impact of EVerest on development and integration effort, Figures 6.1 and 6.2 provide a visual comparison of the project timelines.

  Figure 6.1 shows the timeline of the proprietary, from-scratch implementation carried out by *Abinsula*, which required approximately two years to reach a fully operational EVSE.

Figure 6.2 depicts the timeline of the current project based on the EVerest framework, where the adaptation of the framework, implementation of custom modules, and integration of communication protocols were completed within approximately seven months.

These figures highlight the significant reduction in development time and the efficiency gains achieved through a modular, standards-compliant architecture.

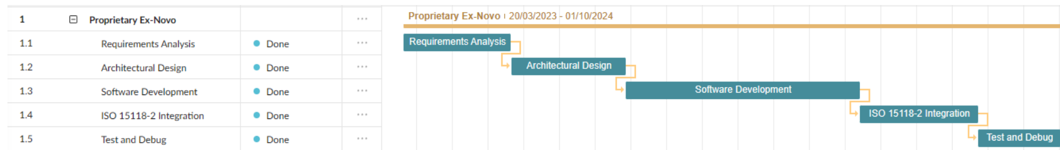


Figure 6.1: Proprietary Ex-Novo

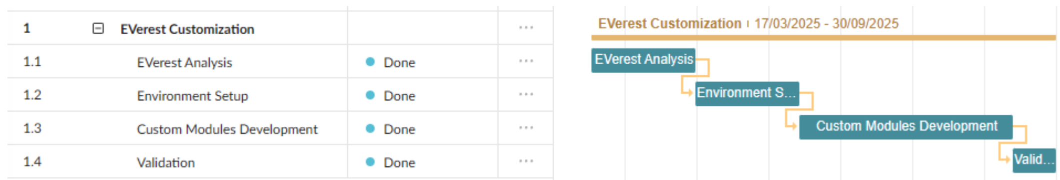


Figure 6.2: EVerest Customization

This outcome highlights the architectural advantages of a modular framework that is compliant with different standards. By providing pre-integrated protocol modules, structured communication between the modules, and built-in logging and debugging capabilities, EVerest significantly reduces the need for low-level protocol implementation and troubleshooting between different layers.

The reduction in development time therefore reflects not only improved implementation efficiency, but also enhanced maintainability, scalability, and long term sustainability of the software architecture in a highly regulated domain such as EV charging infrastructure.

- **Software Limitations**

  Software limitations pose a challenge for developers. These limitations can arise from different sources starting from incomplete protocol implementations, missing protocol implementations, lack of support for some charging modes, but also from non code related factors like insufficient documentation and limited long-term validation in real-world scenarios.

During the project, it happened with EVerest, while trying to implement ISO 15118-20, I observed the infeasibility since this new protocol fully supported the DC mode while being in an early stage for the AC mode. At the time of development, ISO 15118-20 primarily supported DC charging, while ISO 15118-2 supported both AC and DC mode. This limitation restricted the applicability of advanced features such as PnC.

However, being modular and offering the possibility to develop layers if the basic open-source protocols are not yet ready, developers can contribute to the development of missing parts. Although allowing customization of the layers and modules can compensate for the lack of complete protocol implementations, it also increases the complexity of the development and requires more effort from the developers and a deeper understanding of both the protocols and the software stack.

Organizations like *CharIn* (Charging Interface Initiative (CharIN) e. V.) [43], which is a global non-profit industry alliance focusing on promoting interoperability and standardization in the EV charging ecosystem. For example, their initiative *Testival* [44], is a collaborative testing event where manufacturers of EVs, EVSE hardware, and software providers bring their products together in a controlled environment to perform interoperability tests. During the *Testival*, EV manufacturers, charging station producers, component suppliers, testing organizations, and other tech stakeholders are brought together to test real vehicles, charging stations, communication systems, and controllers in a live environment. Unlike a typical conference, this is practical and companies use physical hardware and vehicles to test their products and systems together to check that their solutions work as expected.

- **Compatibility and Interoperability issues**

  One of the challenges of open-source EV charging systems is ensuring compatibility between different components and infrastructures. Open-source protocols such as OCPP and ISO 15118 aim to do so, while stacks like EVerest have as a goal to create a fully working infrastructure that includes all protocols in different scenarios; they must ensure backward compatibility, since in real-world scenarios many systems still rely on earlier protocols.

  Although complete standard implementations can be achieved by developing and implementing missing protocols, many vehicles implement these protocols differently. Usually, standards do not have strict rules about implementation so developers can decide how to create their finite-state machine while also adding more intermediate states. This remains a significant obstacle.

Another challenge for open-source solutions is creating drivers that can interface and interact with multiple hardware components manufactured by different companies, including PLC modems, microcontrollers, and RFID readers. Developing and maintaining reliable drivers for diverse hardware platforms requires significant effort, but at the same time, having a guideline to base custom implementation is a very good base for software like EVerest.

- **Additional Inconveniences**

  The EVerest Admin Panel, which should have helped during the making of the different configuration used both for test and development, was not working during the course of this work, which made the process of configuration and linking of the different modules longer and more complicated, since no HMI was available for those *Manifest* files other than that.

  Another inconvenience I encounter during this work is the inconsistencies in the documentation. It stated that ISO 15118-20 was fully supported while it only covered the DC charging mode, and there were others like this, so I had to go back and forth to check the validity of the statements in the documentation, which led to some time loss.

6.2 Future Research Directions

In future research, it would be useful to define a structured set of structured validation in which the EVerest software is deployed with a variety of real electric vehicles, complementing the emulator-based environment used in this work. The Test Environment created for this project, composed of the Trialog EV emulator and open-source CSMS platforms such as SteVe and Maeve, closely replicates a real-world deployment scenario and proved to be highly effective for integration and protocol-level validation. These tools implement the relevant standards and allow controlled, repeatable, and specification-compliant testing of ISO 15118 and OCPP interactions.

Nevertheless, extending the validation to multiple real EV models would allow the assessment of interoperability aspects that arise from manufacturer-specific implementations and timing behaviors. Collecting and analyzing logs from such sessions would provide additional insight into the robustness of the software under heterogeneous conditions.

Furthermore, scalability studies involving parallel charging sessions could be performed to evaluate how the timing of the different sub-processes evolves under

higher load conditions. Emulated CSMS platforms such as SteVe and Maeve already provide realistic backend behavior and allow comprehensive validation of OCPP interactions. In addition, integrating a production-grade CSMS in future studies would enable validation of large-scale end-to-end deployments, including performance, interoperability, and infrastructure-level constraints.

Another important direction to refine this project would be the implementation of protocols that were not available for the hardware provided, such as ISO 15118-20 or the latest version of the OCPP protocol, 2.1. This would allow for the evaluation of advanced features such as PnC and improved interoperability with the modern EVSE infrastructure.

Another important direction for future research is the automation of the testing workflow and its integration into a continuous development pipeline. Automating key test procedures, including SLAC sequence validation, RFID handling, start/stop routines, and analysis of software and hardware logs, would simplify the verification process and reduce manual effort. By incorporating these automated tests into a CI/CD framework, developers could validate new modules, protocol implementations, and hardware integrations more frequently and systematically. This approach would not only improve the reliability and robustness of the EVerest software but would also accelerate the identification of issues, enabling faster iteration and deployment of new features.

During future research, time can be spent validating the software with a broader set of hardware components, including different connector types and extension to DC charging, which could also include support for both connectors at the same time, giving the customers the possibility to choose between the two. By allowing DC mode, a wider range of vehicles can be covered. Moreover, the software, with some adjustments, could be tested against various microcontrollers, PLC modems, and RFID boards, which would strengthen the code and offer a wider range of uses. Such investigations would also make it possible to assess how hardware-specific characteristics influence timing behavior, communication reliability, and overall system performance.

6.3 Final Considerations

This work aimed to evaluate the flexibility and adaptability of the EVerest software stack within a controlled Test Environment, with particular focus on its customization capabilities. The main objective was to extend the existing architecture through the integration of dedicated drivers and configuration adjustments tailored to the available hardware. Through the development of custom modules, structured testing workflows, and validation procedures, it was possible to assess how effectively the

stack can be adapted to specific implementation requirements while maintaining functional correctness and system stability.

The results demonstrate that the EVerest architecture can be effectively customized to support project-specific requirements, particularly through the integration of dedicated drivers and tailored configurations. The successful adaptation of the stack to the available hardware confirms the flexibility of its modular design. Rather than requiring intrusive modifications to the core components, the implementation relied on extending the existing framework.

At the same time, the project highlighted the importance of thorough validation when working with open-source charging infrastructures. Limitations related to protocol completeness, hardware availability, documentation inconsistencies, and interoperability constraints underline that such systems, while mature in many aspects, still require careful configuration and technical expertise.

From an engineering perspective, this work confirms that open-source EVSE stacks can represent a viable and scalable solution for research, prototyping, and potentially industrial deployment. However, their effective adoption depends not only on protocol support but also on systematic testing methodologies, hardware compatibility, and continuous validation across evolving standards.

Overall, the project demonstrates that with proper configuration, testing, and targeted customization, EVerest provides a solid foundation for further development in the field of electric vehicle charging infrastructure.

Appendix A

ISO 15118

A.1 Communication States

In ISO 15118 the EV charging process is partitioned into eight functional groups, which form a state machine that handles the communication between the EV and the EVSE. Each group represents a status of the vehicle during the charging process. Table A.1 [8] illustrates the groups with their functionalities.

Group	**Function**	**Description**
A	Start of the charging process	Initiation after plugging-in, sets basis for charging
B	Communication setup	Establishes connection between EVCC and SECC
C	Certification handling	All communication related to certificates
D	Identification and Authorization	Methods for identification and authorization
E	Target setting and charging scheduling	Info from EV and SECC to start charging
F	Charging controlling and rescheduling	Commands to control elements during charging
G	Value-added services	Extra services not directly necessary for charging
H	End-of-charge process	Triggers signaling end of charging

Table A.1: ISO 15118 Communication States

A.2 Message Sequences

The states only define the different state in which communication can be. To better understand the communication flow and the transition between states, the protocol

defines detailed message sequences with the data exchanged. The message sequence is not fixed but can vary depending on the charging mode and also on the ISO version, since the parameters required and the information are different in each case.

Understanding the message flow is fundamental for implementing the protocol and creating a finite-state machine specific for the problem needs. A simplified message sequence for DC and AC mode for the ISO 15118-2 version is illustrated in Figure A.1 and Figure A.2 of Typhoon [8].

The DC implementation is more complex than the AC since there is a constant exchange of information such as Status of Charge (SoC), current, and cable check.

It is important to note that ISO 15118-20 is fundamentally similar to ISO 15118-2 since it is built upon it. It extends ISO 15118-2 with additional features to be handled, such as bidirectional energy transfer (V2G), dynamic scheduling, and wireless charge. These features introduce new messages such as ScheduleExchangeReq/Res.

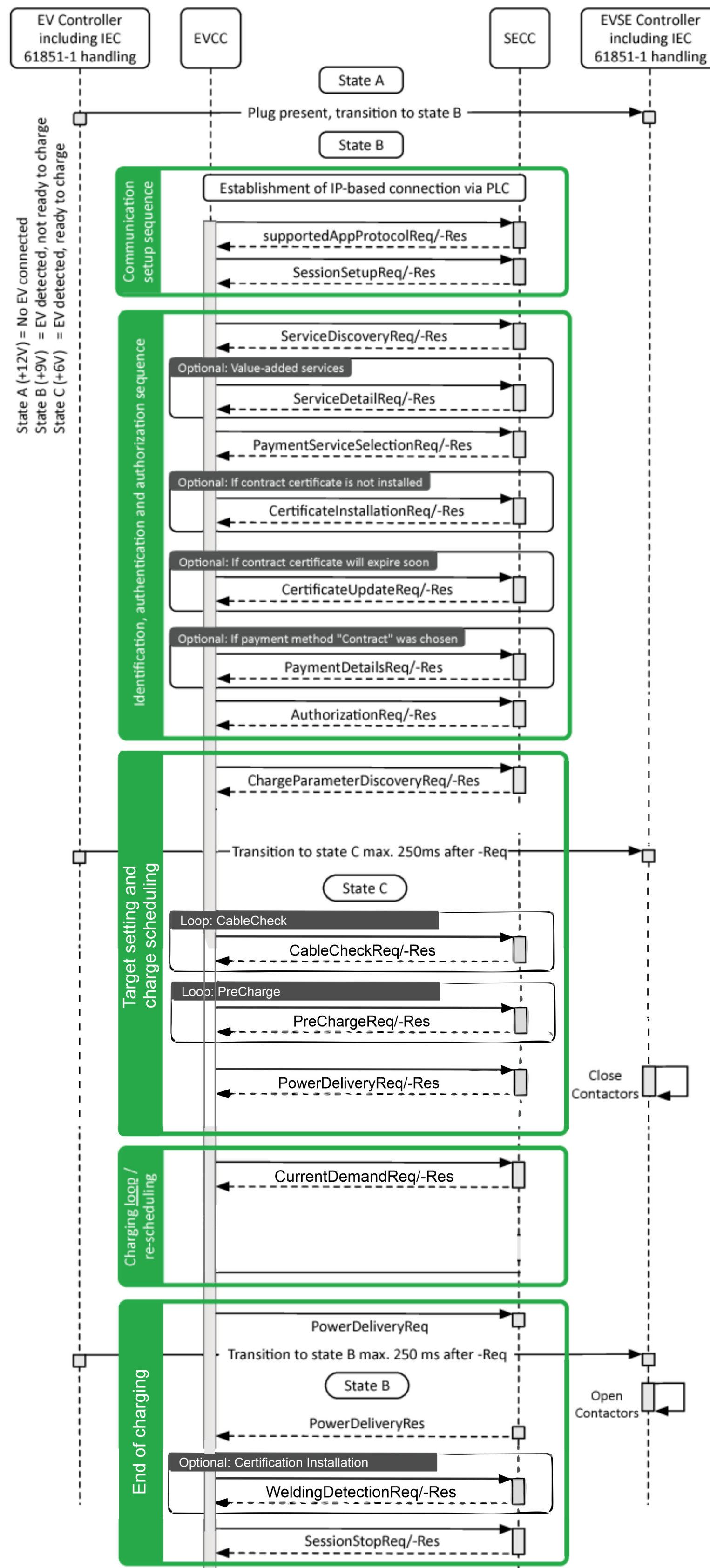


Figure A.1: DC Charging Session

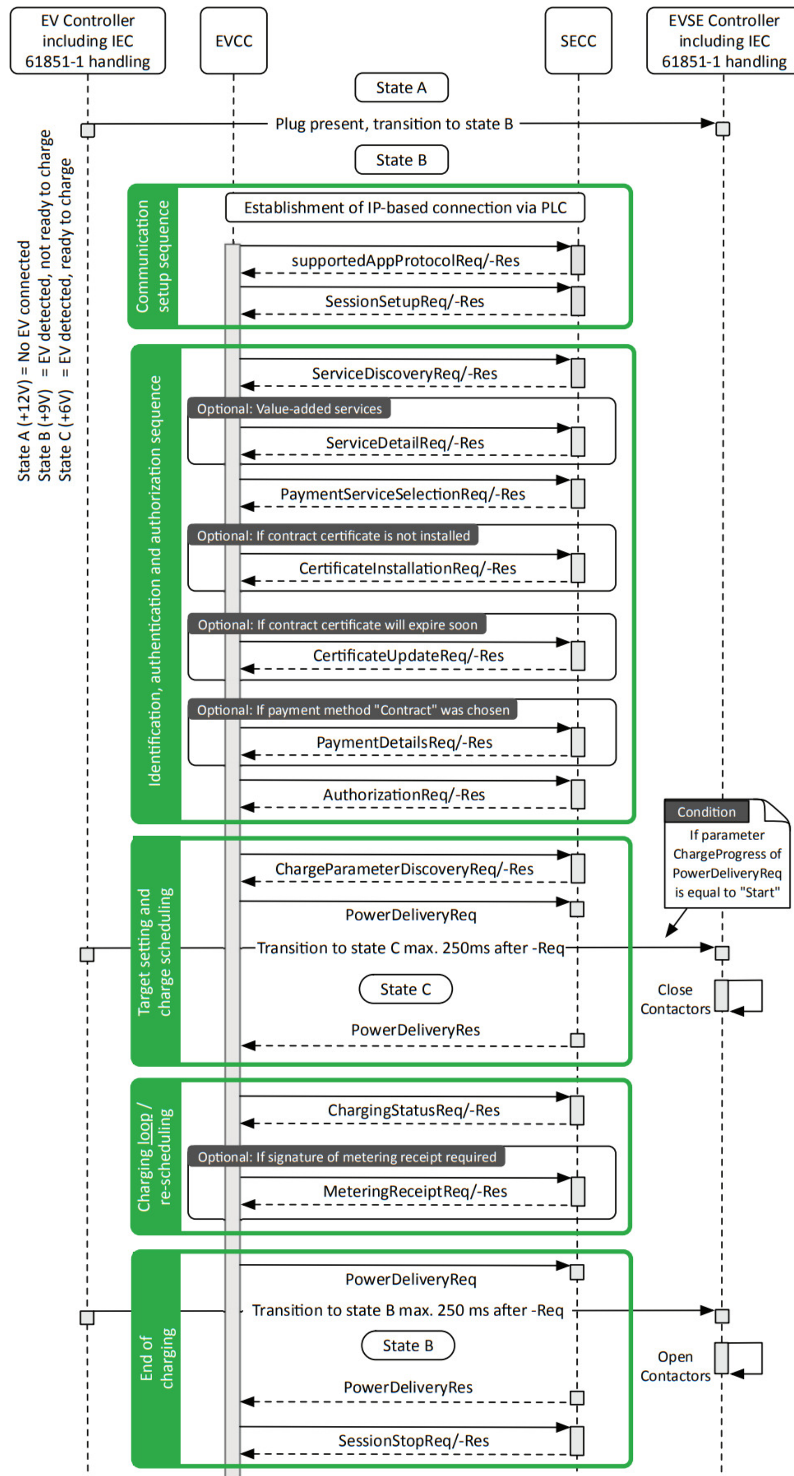


Figure A.2: AC Charging Session

A.3 X.509 Certificates

X.509 certificates are used to validate the identity and transmission of encrypted data so that only the owner (e.g., a person, an organization, a device, etc.) of a specific certificate is able to decrypt and read. These are emitted by a Certificate Authority (CA),

a third-party entity that ensures the relationship between a public key and its identity.

EVSE must rely on a digital certificate hierarchy that is defined by the V2G-PKI (Vehicle-to-Grid Public Key Infrastructure). At the top of the chain lies the V2G Root Certificate Authority (V2G Root CA), which issues certificates to subordinate authorities. From this root a chain of certificates is built:

- *SECC Certificate (Supply Equipment Communication Controller)*: a certificate installed in the EVSE that authenticates the charging station when communicating with an EV
- *CPO Certificate (Charging Point Operator)*: ensures that the EVSE is managed by an authorized identity, linking it to its backend operator
- *Other Certificates*: depending on the PKI design, intermediate authorities can issue certificates for Original Equipment Manufacturer, mobility operators, CSMS, or sub-CAs

The listings A.1 illustrate a decrypted X.509 certificate so that it is easier to understand the various sections. The core sections of these certificates are the serial number, the validity dates, subject and authority, and the final signature at the bottom of the certificate. All these parameters, along with the others, are used to verify the validity of the certificate when a handshake occurs during a communication between two entities.

Listing A.1: X.509 Certificate Structure

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 12:34:56:78:9A:BC:DE:F0
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Test CA, CN = Test Root CA
        Validity
            Not Before: Sep 01 00:00:00 2025 GMT
            Not After : Sep 01 23:59:59 2030 GMT
        Subject: C = US, ST = California, L = San Francisco, O = Test
    Org, CN = test.example.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
```

```
                12:34:56:78:9A:BC:DE:F0:12:34:56:78:9A:BC:DE:F0
:12:34:56:78
            X509v3 Authority Key Identifier:
                keyid:AB:CD:EF:12:34:56:78:90:AB:CD:EF:12:34:56:78:90:
AB:CD:EF:12
 Signature Algorithm: sha256WithRSAEncryption
        12:34:56:78:9A:BC:DE:F0:12:34:56:78:9A:BC:DE:F0:12:34:56:78
        90:AB:CD:EF:12:34:56:78:9A:BC:DE:F0:12:34:56:78:90:AB:CD:EF
```

Appendix B

Software Code Snippets

B.1 Module Configuration File

The *Manifest.yaml* is a crucial part of each module, since it gives an overview of all the parameters and interfaces that can be configured without having to go through the code to understand it. Furthermore, these files are used to create new modules that the framework will setup using the command *ev-cli module create*. Additionally, it provides a guide for the system configuration file, typically named *<name>-config.yaml*. While *Manifest.yaml* provides information about a specific module, the configuration file provides the setup for the EVSE where the parameters can be adjusted or left unset to default and connections can be declared.

Listing B.1 illustrates the configuration file of the module *Auth*, used in the EVSE system to handle user. It has a list of configurable parameters and a list of provided and required interfaces. To make it easier for other developers, a short description, a type, and a default value are provided for each entry in the file.

Listing B.1: Module: Auth

```
description: >-
  This module implements the authentication handling for the EVerest.
    It is responsible for providing authorization to
  the connected evse managers. In addition to that, it handles the
    reservation management.
config:
  selection_algorithm:
    description: >-
      PlugEvents: Selection of connector is based on EV Plug In events
      FindFirst: Simply selects the first available connector that has
   no active transaction
      UserInput: Placeholder, not yet implemented
    type: string
    default: FindFirst
  connection_timeout:
    description: >-
```

```
      Defines how many seconds an authorization is valid before it is
    discarded.
      Defines how many seconds a user can provide authorization after
    the plug in
      of a car
    type: integer
  master_pass_group_id:
    description: >-
      IdTokens that have this id as groupId belong to the Master Pass
    Group.
    type: string
    default: ""
  prioritize_authorization_over_stopping_transaction:
    description: Boolean value to describe the handling of parent id
    tokens.
    type: boolean
    default: true
  ignore_connector_faults:
    description: Boolean value to describe the handling of faults on
    connectors.
    type: boolean
    default: false
provides:
  main:
    description: This implements the auth interface for EVerest
    interface: auth
  reservation:
    description: This implements the reservation interface for EVerest.
    interface: reservation
requires:
  token_provider:
    interface: auth_token_provider
    min_connections: 1
    max_connections: 128
  token_validator:
    interface: auth_token_validator
    min_connections: 1
    max_connections: 128
  evse_manager:
    interface: evse_manager
    min_connections: 1
    max_connections: 128
  kvs:
    interface: kvs
    min_connections: 0
    max_connections: 1
```

B.2 System Configuration File

The system configuration file is the core point of a running EVSE, without it the software cannot run. If used correctly and with an appropriate HMI, multiple config-

uration files can be stored in the EVSE so that different scenarios can be covered.

Listing B.2 is a simple configuration file that includes three different modules: EvseManager, YetiDriver, and EvseSlac. This configuration file is not a complete system but represents a system that only starts and executes the SLAC protocol for a charging session, without going forward.

In the *active_modules* section, all modules are listed with some custom parameters and connections. It is possible to give a different name to the module, for example, *EvseManager* became *connector_1*.

Listing B.2: Configuration File Example

```
active_modules:
  connector_1:
    config_module:
      ac_enforce_hlc: false
      ac_hlc_enabled: true
      ac_hlc_use_5percent: false
      ac_nominal_voltage: 230
      charge_mode: AC
      connector_id: 1
      ev_receipt_required: false
      evse_id: IT*XXX*XXXXXX*X
    connections:
      bsp:
        - module_id: yeti_driver
          implementation_id: board_support
      slac:
        - module_id: slac
          implementation_id: main
    module: EvseManager
    telemetry:
      id: 1
  slac:
    module: EvseSlac
    config_implementation:
      main:
        device: plc1
        number_of_sounds: 10
        ac_mode_five_percent: false
        set_key_timeout_ms: 1000
        sounding_attenuation_adjustment: 0
        publish_mac_on_match_cnf: true
        publish_mac_on_first_parm_req: false
        do_chip_reset: true
        chip_reset_delay_ms: 100
        chip_reset_timeout_ms: 500
        link_status_detection: false
        link_status_retry_ms: 100
```

```
        link_status_timeout_ms: 5000
  yeti_driver:
    module: YetiDriver
```

B.3 OpenAMP Proxy & Adaptor

Proxy and Adaptor are client and server, respectively, that are first implemented as header files used to implement interfaces that can also extend the default features listed in the *XML* file. The initial auto-generated header files are abstract classes, meaning that functions are defined ar virtual so they do not have an implementation, so any class deriving from them must implement these functions.

Listing B.3: Abstract Class Functions

```
class Adaptor_proxy
{
    public:
        static constexpr const char* INTERFACE_NAME = "com.everest_rs.
    OpenAMP.Adaptor";
    protected:
        Adaptor_proxy(sdbus::IProxy& proxy)
            : m_proxy(proxy){}

        ...

        virtual void onCPStateChanged(const sdbus::Struct< uint32_t,
    uint32_t >& newCPState) = 0;

        virtual void onEVContactorsStateChanged(const sdbus::Struct<
    uint32_t, uint32_t, uint32_t > &contactors) = 0;

        virtual void onEVSEACChargeLimitsChanged(const sdbus::Struct<
    int32_t, int32_t, int32_t, int32_t, int32_t, int32_t, int32_t,
    int32_t, int32_t, int32_t, uint32_t, uint32_t, uint32_t>& limits) =
    0;

        ...
};
```

Although I created classes to implement the abstract functionalities those use callback functions that act as place holders instead of implementing the actual code of the defined functions. This technique leaves space for custom implementation while using the sdbus library and allowed me to do some testing during development phase instead of having to build the library after adjustments.

Listing B.4: OpenAMP Proxy Example

```
void OpenAMP_Proxy::onCPStateChanged(const sdbus::Struct< uint32_t,
    uint32_t >& newCPState)
```

```
{
    if (cpStateChangedCallback)
    {
        cpStateChangedCallback(newCPState);
    }
}

void OpenAMP_Proxy::registerCPStateChangedCallback(std::function<void(
    const sdbus::Struct< uint32_t, uint32_t >&)> &&callback)
{
    cpStateChangedCallback = std::move(callback);
}
```

Appendix C

SLAC Protocol

C.1 SLAC Sequence Chart

This appendix provides a detailed illustration of the SLAC message exchange between an EV and an EVSE during the link establishment process. While the main text (Section 2.2.1) describes the general principles and purpose of SLAC, this chart shows the actual sequence of messages, including request and confirmation pairs, and highlights the transitions between the states of both devices.

Table C.1: SLAC Message Exchange Overview

Message	**Description**
CM_SLAC_PARM.REQ	Broadcast request sent by the EV to discover available EVSEs and initiate the SLAC matching process by exchanging basic configuration parameters.
CM_SLAC_PARM.CNF	Confirmation message sent by an EVSE in response to the parameter request, indicating its availability for SLAC matching.
CM_START_ATTEN_CHAR.IND	Message sent by the EVSE to trigger the beginning of the attenuation characterization phase, instructing the EV to start listening for sounding signals.
CM_MNBC_SOUND.IND	Multicast sounding message sent by the EVSE over the power line to allow the EV to measure signal attenuation through the charging cable.
CM_ATTEN_CHAR.IND	Message sent by the EV containing the measured attenuation values for each sounding message received during the characterization phase.
CM_ATTEN_CHAR.RSP	Response from the EVSE acknowledging receipt of the attenuation measurements and confirming successful processing.
CM_ATTEN_PROFILE.IND	Message providing the calculated attenuation profile, used as a unique physical link identifier between the EV and the EVSE.
CM_SLAC_MATCH.REQ	Request sent by the EV to finalize the matching process based on the attenuation profile and establish a logical association with the EVSE.
CM_SLAC_MATCH.CNF	Confirmation sent by the EVSE indicating that the SLAC matching was successful and that the PLC link is established.

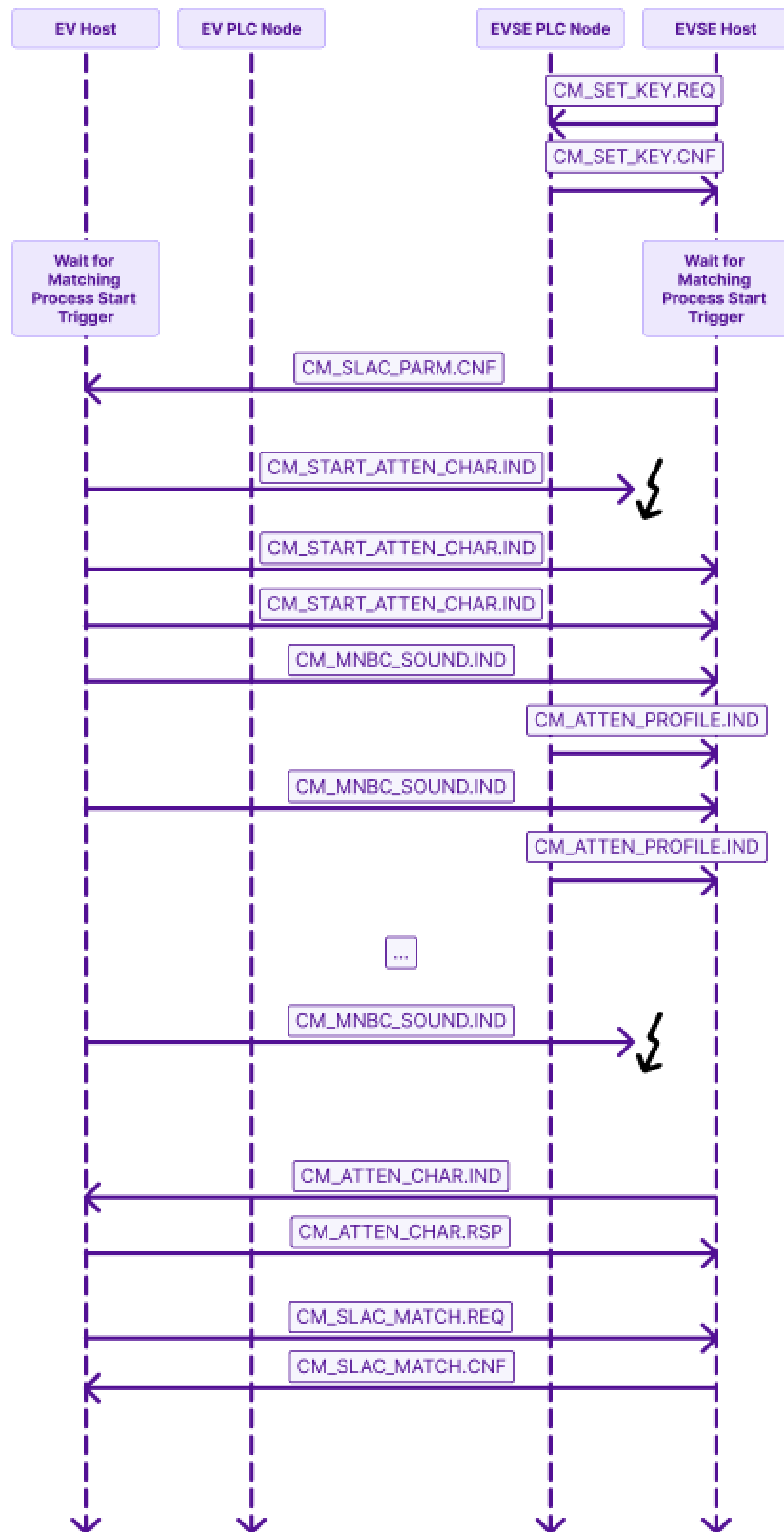


Figure C.1: SLAC Sequence Chart

Bibliography

[1] Our World in Data. "Electric vehicles: Number of electric cars worldwide, 2020–2024". URL: https://ourworldindata.org/electric-vehicles.

[2] Shengbo Cui and Nanting Zhao. "A Study on the Current Status and Future Prospects of EV Automotive Market". In: *Journal of Social Science and Cultural Development* 1 (2024). DOI: 10.70767/jsscd.v1i2.296.

[3] International Energy Agency (IEA). "Global stock of public charging points by speed, 2018–2024". 2025. URL: https://www.iea.org/data-and-statistics/charts/global-stock-of-public-charging-points-by-speed-2018-2024.

[4] EVerest. "EVerest". URL: https://github.com/EVerest/EVerest.

[5] Pazzk. URL: https://pazzk.net/en.

[6] OpenEVSE. URL: https://www.openevse.com/.

[7] Abinsula. "Costruisci un software personalizzato per la tua azienda". URL: https://abinsula.com/it/.

[8] Typhoon-Hil Inc. "ISO 15118-2 Protocol". In: *Tyhoopn HIL Documentation* (2025). URL: https://www.typhoon-hil.com/documentation/typhoon-hil-software-manual/References/iso15118_protocol.html.

[9] Typhoon-Hil Inc. "ISO 15118-20 Protocol". In: *Tyhoopn HIL Documentation* (2025). URL: https://www.typhoon-hil.com/documentation/typhoon-hil-software-manual/References/iso_15118_20_protocol.html?hl=iso%2C15118-20%2Cprotocol.

[10] eInfochips. "IEC 61851: Everything You Need to Know About the EV Charging Standard". In: (2025). URL: https://www.einfochips.com/blog/iec-61851-everything-you-need-to-know-about-the-ev-charging-standard/.

[11] Joint. "What is DIN 70121 for EV Chargers?" In: (2025). URL: https://jointcharging.com/what-is-din-70121-for-ev-chargers/.

[12] Open Charge Alliance. "Open Charge Point Protocol". In: (). URL: https://openchargealliance.org/protocols/open-charge-point-protocol/.

[13] Pionix. "EVerest Documentation - EVerest Framework". URL: https://everest.github.io/nightly/index.html#.

[14] go-e. "AC vs DC Charging: 7 Fundamental Differences". In: (2025). URL: https://go-e.com/en/magazine/ac-dc-charging.

[15] acea. "Around one out of every eight EU public chargers is a fast charger". In: *acea* (2024). URL: https://www.acea.auto/figure/around-one-out-of-every-eight-eu-public-chargers-is-a-fast-charger/.

[16] Greenc-Ev. "Tecnologia Vehicle-to-Grid per auto elettriche nel 2024". URL: https://greenc-ev.com/it/tecnologia-Vehicle-to-Grid-per-le-auto-elettriche-nel-2024/.

[17] Žiga Lesjak. "OCPP Protocols (OCPP 1.6, OCPP 2.0.1, and OCPP 2.1)". In: *Tridens* (2025).

[18] Marc Mültin. "What is ISO 15118". In: *Switch* (2021). URL: https://web.archive.org/web/20251116005819/https://www.switch-ev.com/blog/what-is-iso-15118.

[19] Pionix. "EVerest Powering EV-Charging". URL: http://everest.github.io.

[20] Soumaya Boussaha, Victor Fresno Gómez, Thomas Barber, and Daniele Antonioli. *EmuOCPP: Effective and Scalable OCPP Security and Privacy Testing.* Seattle, United States, 2025. URL: https://hal.science/hal-05127201.

[21] ALEŠ KRAMŽAR. "Modifikacija polnilne postaje za električna vozila OpenEVSE". PhD thesis. Univerza v Ljubljani, Fakulteta za elektrotehniko, 2020.

[22] Filippo D'Agostino. "Implementation of a PKI-based security communication and Value Added Service for EV charging using ISO 15118 standard". PhD thesis. Politecnico di Torino, 2023. URL: mplementation%20of%20a%20PKI-based%20security%20communication%20and%20Value%20Added%20Service%20for%20EV%20charging%20using%20ISO%2015118%20standard.

[23] Debora Maria Marettek. "Elicitation and Classification of Security Requirements for EVerest". PhD thesis. Karlsruher Institut für Technologie (KIT), 2024.

[24] Devki Nandan Jha et al. "EVECTOR: An orchestrator for analysing attacks in electric vehicles charging system". 2026. arXiv: 2602.13926 [cs.OH]. URL: https://arxiv.org/abs/2602.13926.

[25] Serra Ilgin. "Efficient and Interoperable EV Charging: Building an ISO15118-Compliant Charger using EVerest Software Stack". MA thesis. Università degli Studi di Padova, 2023. URL: https://hdl.handle.net/20.500.12608/62275.

[26] Open Charge Alliance. "OCPP Plugfest". In: 2025. URL: https://openchargealliance.org/ocpp-plugfest/.

[27] Idaho National Laboratory. URL: https://inl.gov/chargex/.

[28] Casey W Quinn and Dhananjay Anand. *ChargeX Consortium Presentation at OCPP Plugfest.* Tech. rep. Idaho National Laboratory (INL), Idaho Falls, ID (United States), 2025.

[29] Ed Watt, Pranav Gadamsetty, and Mayuresh Savargaonkar. *Implementation of ISO 15118-202 messages within Everest EV Charging Open Source Framework [SWR-25-56]*. Tech. rep. National Renewable Energy Laboratory (NREL), Golden, CO (United States ..., 2025.

[30] Kenneth Rohde and Jake M Guidry. *Efficient Xml Interchange (exi) For Python (expy)*. Tech. rep. Idaho National Laboratory (INL), Idaho Falls, ID (United States), 2025.

[31] ST. "STM32MP1 COPROC". URL: https://www.st.com/content/ccc/resource/training/technical/product_training/group1/5c/0a/4f/2b/a1/b8/4a/a5/STM32MP1-Software-Coprocessor_management_COPROC/files/STM32MP1-Software-Coprocessor_management_COPROC.pdf/_jcr_content/translations/en.STM32MP1-Software-Coprocessor_management_COPROC.pdf.

[32] Pionix. "How To: Develop New Modules". URL: http://everest.github.io/nightly/explanation/detail-module-concept.html.

[33] Marc Mültin. "The basics of Plug & Charge". In: *Switch* (2020). URL: https://web.archive.org/web/20251214132349/https://www.switch-ev.com/blog/basics-of-plug-and-charge.

[34] Trialog. URL: https://www.trialog.com/en/home/.

[35] TheYoctoProject. URL: https://www.yoctoproject.org/.

[36] freeRTOS. "What is FreeRTOS?" URL: https://www.freertos.org/Why-FreeRTOS/What-is-FreeRTOS.

[37] Kistler Group. "sdbus-cpp". In: (2024). URL: https://github.com/Kistler-Group/sdbus-cpp/tree/v2.1.0.

[38] Pionix. "How To: Develop New Modules". URL: https://everest.github.io/nightly/tutorials/new_modules/index.html.

[39] SteVe-community. "Steve". URL: https://github.com/steve-community/steve.

[40] Thoughtworks. "maeve-csms". URL: https://github.com/thoughtworks/maeve-csms.

[41] Pionix. "The Backbone of EV Charging". URL: https://www.pionix.com/.

[42] Pionix. "meta-everest". URL: https://github.com/EVerest/meta-everest.

[43] CharIN. "Empowering the next level of green mobility and energy." URL: https://www.charin.global/.

[44] Iohanna Vater. "Testival - Become our next host for a CharIN Testival & Conference in 2025 or 2026!" In: *Charin* (2025). URL: https://www.charin.global/news/testival-rfq-2025-2026/.

Dedications

I would like to sincerely thank my thesis supervisor, Professor *Renato Ferrero*, for the guidance, valuable suggestions, and continuous support provided throughout the development and writing of this thesis. I am especially grateful for the time and attention that he dedicated to reviewing this work multiple times, providing thoughtful comments, and giving both the thesis and its development great importance.

I would like to express my sincere gratitude to my company tutor, *Ilario Pittau*, for the continuous support, technical guidance, and availability throughout the entire internship. Working with him allowed me not only to develop new technical skills, but also to discover how engaging and rewarding working on projects of this kind can be.

I am particularly grateful for the trust he placed in me and for the many opportunities he gave me to learn and grow professionally. Beyond the professional guidance, his mentorship and encouragement made this experience especially meaningful, and it is a collaboration I will always remember. I truly hope to have the opportunity to work with him again in the future on new and challenging projects.

I would also like to thank *Abinsula* for giving me the opportunity to work on an innovative project involving the EVerest framework and EV charging infrastructure. This experience allowed me to gain hands-on exposure to real-world EVSE development, including protocol integration, system validation, and interaction with complex charging standards.

Finally I want to thank my number one fans *Mamma & Babbo*, or should I say *Monica & Roberto*. They believed in me even when I didn't. They gave me the courage to come to *Turin* and start something new, something I have always loved. Their constant encouragement, patience, and support have always been the foundation that allowed me to pursue my goals. They cheered for me more than I,

even for small goals, and I could not be more grateful to have them in my life.

After six long years of study, I am proud to say that I finally made it and none of this would have been possible without them and the rest of my family, who have always believed in me and supported me throughout this journey.