DTU Wind

Implementation and test of the open charge point protocol in an autonomous charger for electric vehicles

Author
Martin Ellehammer Hansen

DTU Wind-M-0756
February 2024

Author:
Martin Ellehammer Hansen

Title:
Implementation and test of the open charge point protocol in an autonomous charger for electric vehicles

DTU Wind-M-0756
February 2024

ECTS: 30

Education: Master of Science

Supervisors:
Mattia Marinelli
Kristian Sevdari
DTU Wind & Energy Systems

Oliver Lund Mikkelsen

Remarks:
This report is submitted as partial fulfillment of the requirements for graduation in the above education at the Technical University of Denmark.

DTU Wind & Energy Systems is a department of the Technical University of Denmark with a unique integration of research, education, innovation and public/private sector consulting in the field of wind energy. Our activities develop new opportunities and technology for the global and Danish exploitation of wind energy. Research focuses on key technical-scientific fields, which are central for the development, innovation and use of wind energy and provides the basis for advanced education at the education.

Technical University of Denmark Department of Wind Energy
Frederiksborgvej 399
4000 Roskilde
Denmark
www.vindenergi.dtu.dk

Abstract

This master's thesis investigates the implementation of the Open Charge Point Protocol (OCPP) in an electric vehicle autonomous charger (ACDC). As an important standard, OCPP facilitates secure and seamless interactions between Charging Stations (CS) and their respective Charging Station Management Systems (CSMS). This research aims to provide practical insights into the OCPP implementation, contributing to a standardized and efficient electric vehicle charging infrastructure.

Based upon multiple factors, the chosen OCPP version to be implemented is version 2.0.1, especially because this is the newest, state-of-the-art version of the protocol on the market. This decision ensures that the system remains at the forefront of technological advancements, offering enhanced security, interoperability, and a wide range of features essential for modern electric vehicle charging infrastructure.

The CSMS is created as a secure WebSocket server through AWS API Gateway. The created WebSocket is based upon a "server-less" architecture and is greatly scalable. The implemented database is the one provided by AWS cloud services, DynamoDB. This database is used to store all relevant information received by the CSMS along with other critical information.

On the charging stations, the already installed Zephyr RTOS environment provides a WebSocket library, capable of connecting to the new CSMS providing real-time bi-directional communication. Core functions within the OCPP are implemented and tested, showcasing that the overall setup is working and that values can be changed on the charging station, directly from the CSMS.

Overall the fundamental steps of implementing OCPP 2.0.1 have been taken, providing a CSMS, capable of scaling, and an encrypted connection between the CS and the CSMS has been created. Thereby this thesis has provided the fundamental steps, paving the way for further development and implementation.

List of Tables

List of Figures

Contents

1 Introduction

The need for the transition to clean and sustainable energy is undeniable, and in 2020, the transportation sector was the second greatest emitter of greenhouse gasses globally[1]. The world will not reach its goal of zero emissions without great changes in this industry.

In 2023 the global sales of electric vehicles grew by 55%, reaching an all-time high of more than 10 million deployed electric vehicles worldwide[2]. The world is rapidly moving towards sustainable options for transportation, and as the number of electric vehicles (EV) is greatly increasing, a parallel need for intelligent management of the power grid and energy transmission arises. This is where the Open Charge Point Protocol (OCPP), the object of interest in this thesis, becomes relevant.

The OCPP offers a multitude of benefits that create a robust and efficient charging infrastructure. Thus, the OCPP will better the user experience [3], and therefore it may be expected to greatly increase the already rapid growth in the adoption of electric vehicles.

Furthermore, the complexity of making the charging infrastructure more efficient extends beyond just the user experiences in the charging of a vehicle. It involves providing grid services, making the charging stations more efficient, creating robust central management systems, and upgrading the electric vehicles themselves.

In the expanding world of EV infrastructure, OCPP is breaking through, acting as a universal language that can facilitate seamless communication between charging stations and central management systems. This interoperability is critical for real-time management of energy flows and grid services, mounting in a dynamic and responsive charging infrastructure.

The origin of OCPP is based upon the idea that to make electric vehicles widespread all over the world, a charging system that is easy to use, and in no way intimidating to the user is paramount [4]. OCPP handles this issue by greatly increasing interoperability, creating a shared communication framework, and ensuring different equipment can coexist within a charging network.

Thus, OCPP is more than just a technical specification created by the Open Charge Alliance (OCA). It is the realization, a collective understanding, that the introduced interoperability is fundamental in introducing a consumer-oriented experience.

1.1 Why is OCPP necessary?

The accelerating adoption of electric vehicles (EV) has highlighted the need for standardized and interoperable charging infrastructure, and this is precisely where the Open Charge Point Protocol (OCPP) steps in and addresses critical challenges[5].

Interoperability and User Convenience

The main goal of OCPP is to make sure charging stations and management systems work well together, no matter who made them. Without a common standard, the EV charging world would be split up, making it tough for drivers to easily use different charging networks. OCPP helps ensure that users have a smooth and straightforward experience at any charging station, which helps more people feel comfortable switching to electric vehicles.

Scalability for Growing EV Market

With the global surge in EV adoption, the scalability of charging infrastructure becomes paramount. OCPP's open standard allows for the seamless integration of new charging stations into existing networks. This flexibility is crucial for accommodating the exponential growth of electric vehicles on the road, ensuring that the charging infrastructure can scale dynamically to meet evolving demand[6].

Operational Efficiency and Remote Management

Moreover, OCPP greatly contributes to the operational efficiency of charging stations. The remote management capabilities introduced with the Charging Station Management System (CSMS) help the operator monitor, control, and troubleshoot from a centralized location[7]. The efficiency in operation will result in less downtime, quicker issue resolution, and optimized grid services.

Data Standardization for Informed Decision-Making

The standardization of the exchanged data between the CS and the CSMS is another key benefit of OCPP and part of the greatly increased interoperability. The standardization makes sure that all parties involved will have access to reliable information, including charging session data, energy consumption, and charging station statuses. This makes a great foundation for future infrastructure enhancements.

Enhancing User Experience

OCPP's role in promoting interoperability directly translates to an improved user experience. Electric vehicle owners benefit from a seamless and consistent charging process, regardless of the charging station's make or model. This user-centric approach is crucial, providing confidence for the user in the adaptation of EVs.

Enabling Innovation and Competition

The open standard framework of OCPP introduces healthy competition among charging equipment manufacturers and encourages fast-developing innovation. The environment of OCPP creates a platform for the development of cutting-edge technologies and a diverse range of charging solutions, in the end benefiting the consumer, and contributing to the advancement of the electric mobility environment.

1.2 Goals of the Thesis

The objectives of this thesis revolve around the preliminary design, structuring, and initial implementation phases of the Open Charge Point Protocol (OCPP) within an Autonomously Controlled Distributed Charger (ACDC) for electric vehicles, created in collaboration between Circle Consult and DTU in the EV4EU project. This exploration researches the complexity of integrating OCPP, focusing on the initial steps necessary for the implementation and progressing toward the establishment of fundamental functions, tested in alignment with the Open Charge Alliance's specified test scenarios. The key stages investigated include:

- Developing a WebSocket server to function as the Charging Station Management System (CSMS), enabling real-time, bidirectional communication with the Charging Station.
- Establishing and maintaining robust communication links between the Charging Stations (CS) and the CSMS, ensuring data integrity and system reliability.
- Crafting core OCPP functionalities and ensuring their compatibility and effectiveness through rigorous testing on an Autonomously Controlled Distributed Charger (ACDC).

This sets the foundation for a more detailed and precise implementation phase of OCPP, targeting specific functionalities and use cases. Through this systematic approach, the thesis aims to contribute to the advancement of the electric vehicle charging infrastructure, facilitating efficient energy management, enhanced security measures, and improved user experience, ultimately supporting the broader adoption of electric vehicles through the use of the OCPP.

1.3 Structure of the Thesis

This thesis has been structured in a comprehensive way to provide an overview of the research conducted.

Chapter 2 presents the current states of the different versions of the Open Charge Point Protocol, and the main differences between the versions. Additionally, it outlines the essential use cases required for a basic OCPP implementation, along with the corresponding security measures that must be addressed.

Chapter 3 titled *Methodology*, delves into the research methods and approaches utilized throughout this study. This chapter focuses on an *Requirement analysis*, the *Architecture* used for the CSMS, CS, and the *Communication* through WebSockets. Finally, test cases are formulated to be carried out.

Chapter 4 focuses on the Implementation of the project. Here insights are given into how to get started with development, how the CSMS is created, and what it entails, along with the implementation done on the Charging station.

Chapter 5 presents the *Results* obtained through testing, and analyzes and evaluates the results of the test cases.

Chapter 6 , *Discussion*, considers and discusses the implementation of different aspects along with the results presented in the preceding chapter.

Chapter 7 concludes the thesis, summarizing the findings and results, as well as giving an idea of what future work might entail for the integration of OCPP.

2 Background

This chapter describes the relevant terminology used for the implementation of OCPP. Further, it discusses which version of OCPP to implement, what the core of the OCPP implementation is, and the importance of security.

2.1 Which OCPP version is to be implemented?

The Open Charge Point Protocol is a living protocol, constantly being innovated and improved upon. Over the past years, different versions of the protocol have been created, with the most widespread version being OCPP 1.6. However, the newest, more capable, and complex version, OCPP 2.0.1, is the state of the art of the protocol.

2.1.1 OCPP 1.6

The OCPP version 1.6, developed by the Open Charge Alliance (OCA), is the most widespread version of the protocol and is an iteration of OCPP 1.5. The protocol provides a communication standard allowing the CS and CSMS to communicate effectively with each other. OCPP 1.6 includes several features, such as remote start and stop of a charging session, status notifications, firmware management, and data transfer capabilities over a WebSocket connection using either SOAP or JSON.

Despite its strengths, OCPP 1.6 has certain limitations, particularly in terms of security features, where additional security is introduced in the newer version [8]. It lacks advanced mechanisms for secure firmware updates and sophisticated authentication, which are essential for protecting against cyber threats. Additionally, the protocol supports only basic smart charging capabilities and does not include support for the ISO 15118 protocol, which offers advanced EV-to-charger communication and features like Plug & Charge. In terms of scalability, OCPP 1.6 might face challenges in handling large networks with complex use cases, especially when compared to the more advanced OCPP 2.0.1.

2.1.2 OCPP 2.0.1

OCPP 2.0.1 is a significant advancement in the development of the Open Charge Point Protocol, addressing many of the limitations found and requested in earlier versions. A strength of OCPP 2.0.1 is its enhanced security features, which include introducing more advanced authentication mechanisms while providing robust protection against cyber threats. OCPP 2.0.1 also introduces advanced smart charging capabilities, allowing more efficient and intelligent management of CS. Also significantly, OCPP 2.0.1 is compatible with the ISO 15118 standard, supporting vehicle-to-charger technology features like Plug & Charge. Making sure charging systems can work smoothly with future electric vehicle technology is essential to a smooth experience for drivers as they charge their EVs[9].

Moreover, OCPP 2.0.1 is designed to handle large-scale deployments and complex use cases, making it highly scalable and suitable for the foundation of the infrastructure of EV charging. The protocol offers improved data handling and efficiency. With these advancements, OCPP 2.0.1 provides a more comprehensive and robust solution for the infrastructure of EV charging.

Direct Vehicle Communication, another significant advantage of OCPP 2.0.1 is its ability to obtain valuable data directly from the vehicle. This includes crucial information such as the state of charge, charging preferences, battery capacity, and vehicle identification. Access to this data allows for more intelligent charging management, and tailored charging experiences. This is a feature very useful for the ACDC project, as it will eliminate a lot of the actions the consumer has to take, in order to start a smart charging session.

Limitations of OCPP 2.0.1 While OCPP 2.0.1 addresses many issues of previous versions, it does present certain limitations. One of the main challenges is the complexity of the implementation of the newer protocol. The advanced features and enhanced security measures can make deployment and integration more complex and resource-intensive, particularly for organizations transitioning from earlier versions of the protocol[8]. The lack of backward compatibility means that upgrading existing OCPP 1.6 systems to OCPP 2.0.1 can become time-consuming and costly.

Another potential limitation is the need for continuous updates and maintenance to keep up with the evolving standards and security requirements. This ongoing requirement can be a challenge for smaller operators or those with limited technical capabilities.

Despite these limitations, OCPP 2.0.1 represents a major step forward in the EV charging industry compared to OCPP 1.6, offering enhanced capabilities and security that are essential for modern charging infrastructure. Moreover, implementing the newest version of the standard will also present a charging solution that will be competitive on the market for many years to come.

2.1.3 The need for smarter features

Smart Charging As the growth of EVs on the market is rising, so is the demand requested from the electric grid. As smart charging introduces flexible demand by utilising the capabilities of unidirectional (V1G) or bidirectional (V2G) functionalities[10], it will be able to support the grid, by broadening the demand required at critical times by strategic charging, thereby being able to help with peak shaving the demand curve during critical hours. This effect is needed to release the stress on the grid, potentially enabling flexibility in the incorporation of renewable energy sources (RES)[11][12].

Interoperability is an important aspect to the consumer. Providing interoperability to the user through OCPP, is one of the main aspects, allowing the consumer to switch between different flexible providers. Moreover the strictness introduced in the OCPP protocol makes the protocol highly interoperable, and make the integration of a new charge point to a central system operate without any problems (or few)[13].

Ease of Use is another crucial feature to become a competitive CS in the market. Here the need for the ISO 15118 standard is needed, in order to use the plug & charge technology [8]. This feature is not implemented in OCPP 1.6, and many charging station providers have had to implement it around the OCPP protocol.

2.1.4 OCPP 2.0.1 for an Advanced Charging Infrastructure

The chosen protocol for the implementation in this thesis is OCPP 2.0.1, prioritizing the multiple benefits associated with the complexity of implementation. The key benefits are the critical need for enhanced cyber security, along with smart charging capabilities and Plug-&-charge technology.

The incorporation of the Plug and Charge functionality, supported by the ISO 15118 standard, is a key factor in the choice of OCPP 2.0.1. This feature streamlines the charging process, allowing for automatic vehicle identification and authorization, and enhancing user experience.

OCPP 2.0.1 enables direct access to crucial vehicle data, such as the state of charge, charging preferences, and battery capacity. This access streamlines the charging process by eliminating the need for users to manually input values before starting a charging session, as these key parameters can now be automatically read from the vehicle, which fits well in the scope of the ACDC project. This feature enhances smart charging by allowing dynamic adjustment of charging rates and schedules based on each vehicle's specific needs and battery status, leading to more efficient energy management and better user experiences.

In summary, while OCPP 2.0.1 presents a more complex implementation process, its advanced features align with the project's focus on security, efficiency, and robustness for the future. The timeline of the EV4EU project allows for thorough integration of these sophisticated capabilities, ensuring a robust and user-friendly charging infrastructure using OCPP 2.0.1.

2.2 Core Certification Profile

When implementing OCPP, the first overall goal is to become OCPP compliant and to do so, the core certification profile must always be present. It consists of a lot of different use cases, which have to pass specific testing.

Table 1: Core Certification profile [14]

Certification Profile	**Description**
Core	Basic Authentication
	TLS - server-side certificate
	Update Charging Station Password for HTTP Basic Authentication
	Security Event Notification
	Booting a Charging Station
	Configuring a Charging Station
	Resetting a Charging Station / EVSE
	Authorization incl. GroupId
	Stop Transaction with a Master Pass
	Local start transaction - Cable plugin first & Authorization first
	Start / Stop transaction options
	Disconnect cable on EV-side
	Check Transaction status
	Remote start / stop transaction
	Remote unlock Connector
	Remote Trigger
	Change Availability - Charging Station / EVSE / Connector
	Clock-aligned Meter & Sampled Meter Values
	Install CA certificates
	Retrieve certificates from Charging Station
	Delete a certificate from a Charging Station
	AdditionalRootCertificateCheck
	Retrieve Log Information
	Get / Clear Customer Information
	Secure Firmware Update
	Store / Clear Authorization Data in Authorization Cache
	Authorization through authorization cache

To achieve full OCPP certification, all certification profiles must be implemented. It is possible, however, to obtain partial certification by fulfilling at least the core certification profile. After establishing the core certification profile, additional profiles may be adopted to enhance capabilities, pursuing full OCPP compliance, as detailed in the ensuing list:

- Advanced Security
- Local Authorization List Management
- Smart Charging

- Advanced Device Management
- Advanced User Interface
- Reservation
- ISO 15118 support

2.3 Security

It is important to take serious measures regarding the security of charging stations, as these are more and more prone to cyber-attacks in the future. Therefore there are four well-defined security objectives in the documentation provided by the Open Charge Alliance (OCA)[15]:

- To allow the creation of a secure communication channel between the CSMS and the Charging Station. The integrity and confidentiality of messages on this channel should be protected with strong cryptographic measures.
- To provide mutual authentication between the Charging Station and the CSMS. Both parties should be able to identify who they are communicating with
- To provide a secure firmware update process by allowing the Charging Station to check the source and the integrity of firmware images, and by allowing non-repudiation of these images.
- To allow logging of security events to facilitate monitoring the security of the smart charging system. A list of security-related events and their 'criticality' is provided in the appendices.

To secure the messages, and make sure no one can intercept and read the messages being communicated between the CS and the CSMS, the WebSocket Server created will be a WebSocket Secure (wss). This ensures that all messages being sent and received are TLS encrypted, and can not be intercepted and read by third parties.

When implementing the security to the system, introducing TLS authentication certificates for both the CS and CSMS is optimal. However, there are three accepted predefined security profiles that can be used. They are defined as in table 2:

Table 2: Overview of OCPP security profiles [15]

Profile	**Charging Station Authentication**	**CSMS Authentication**	**Communication Security**
1. Unsecured Transport with Basic Authentication	HTTP Basic Authentication	-	-
2. TLS with Basic Authentication	HTTP Basic Authentication	TLS authentication using certificate	Transport Layer Security (TLS)
3. TLS with Client Side Certificates	TLS authentication using certificate	TLS authentication using certificate	Transport Layer Security (TLS)

However, later on when the implementation of the protocol begins, to focus on getting fundamental parts working, eg the CSMS, the connection between the CS and the CSMS, and the OCPP functions, none of these authentication certificates are implemented. It will be run on a WSS, and have TLS peer verification as an optional parameter, making it easier to implement in the future. The following statement is specified in the OCA's documentation [15]:

- In some cases (e.g. lab installations, test setups, etc.) one might prefer to use OCPP 2.0.1 without implementing security. While this is possible, it is NOT considered a valid OCPP 2.0.1 implementation.

3 Methodology

3.1 Requirements analysis

To ensure a good starting point for the implementation of OCPP 2.0.1, it is important to have clarified the foundational steps and specifications necessary for a holistic OCPP implementation. This section outlines the core functional requirements vital for the system's architecture, creating a clear path toward effective integration.

3.1.1 Charging Station Integration Requirements

The implementation of OCPP 2.0.1 is carried out on the nRF-9160DK using Zephyr RTOS[16], which necessitates a specific setup to ensure clear development and integration. The critical components of this setup are:

- nRF-9160DK Board: The primary development platform, as the integrated communication board used in the ACDC, is the nRF-9160 board, taking advantage of its LTE-M and NB-IoT capabilities.

- Zephyr RTOS: A scalable real-time operating system (RTOS) for connected, resource-constrained devices like the nRF-9160 board. Zephyr provides a WebSocket library essential for establishing OCPP communication channels between the CSMS and the CS.

3.1.2 CSMS implementation requirements

The deployment of the OCPP 2.0.1 Central System will be done on AWS cloud services, and sets out so ensure a secure, scalable and reliable server-side environment for the CSMS. Components and considerations include:

- WebSocket server implementation: Utilizing AWS API Gateway to create and manage a WebSocket API acting as the CSMS for OCPP 2.0.1 communications. This setup facilitates real-time messaging between charging stations and the central system without the need for managing server infrastructure.

  The WebSocket API makes real-time messaging between the CS and the CSMS possible, without a huge need for managing server infrastructure, as AWS takes care of most.

- Security Configuration: Implement authentication and authorization with certificate authorities (CA), mechanisms through AWS API Gateway, using IAM roles and Lambda authorizers to validate connections and messages in AWS.

- AWS Lambda:

  - Server-less Computing: Deployment of AWS Lambda functions to handle OCPP messages, connections, and disconnections, providing a server-less architecture capable of scaling automatically with the number of incoming requests.
  - Integration: Ensure Lambda functions are integrated properly with API Gateway WebSocket routes for robust message processing and routing.

- Amazon DynamoDB: Leverage DynamoDB for storing transactional data, configurations, and different status information of the charging stations. DynamoDB offers fast, scalable NoSQL database capabilities perfect for handling the shifting amount of workload that will be introduced to the CSMS.

- Scalability and Reliability:
  - Managed Scaling: Benefit from the automatic scaling ability of API Gateway and Lambda handlers, which adjust the resources used based on traffic patterns, keeping efficient handling of OCPP communications without any intervention[17].
  - High Availability: AWS services used like API Gateway, Lambda, and DynamoDB are designed for high availability and a good fault tolerance across multiple availability zones, making it a service well suited for scaling to other markets in the future.

This analysis underscores the importance of a strategic approach to system architecture, emphasizing the need for secure, scalable, and efficient communication between charging stations and the central system. By adhering to these outlined requirements, the foundation is set for a successful OCPP 2.0.1 implementation that is well-positioned to support the evolving demands of the electric vehicle charging infrastructure.
This requirement analysis highlights the significance of a well-sorted approach to system architecture, focusing on the need for secure, scalable, and efficient communication between the CS and the CSMS. By following the outlined requirements, a solid foundation for a successful OCPP 2.0.1 implementation is set, capable of scaling for future needs.

3.2 Architecture - OCPP, WebSocket Server, Client-Side

3.2.1 The Architecture of the OCPP

The architecture of the Open Charge Point Protocol is well described in [18], including the information model, a 3-tier model, and a device model. Together this ensures an architecture providing standardized communication and good interoperability for the growing EV infrastructure.

The information model explains the structure and types of messages exchanged between Electric Vehicle Supply Equipment (EVSE) and Charging Station Management Systems (CSMS), making sure consistent definitions are kept across different manufacturers' devices and management software.

The 3-tier model outlines the physical and logical layers within the charging infrastructure of the entire CS, categorizing the system into Charging Station (CS), EVSE, and Connector levels. The representation can be seen in figure 1.

- Charging Station (CS): This is the physical system where EVs are charged. It may consist of one or more EVSEs and serves as the primary interface for electric vehicles.
- EVSE (Electric Vehicle Supply Equipment): Represents the actual charging points and can be seen as independently operated units within the Charging Station.
- Connector: This refers to the individual connectors or sockets on the EVSE where the electric vehicle is plugged in to charge.

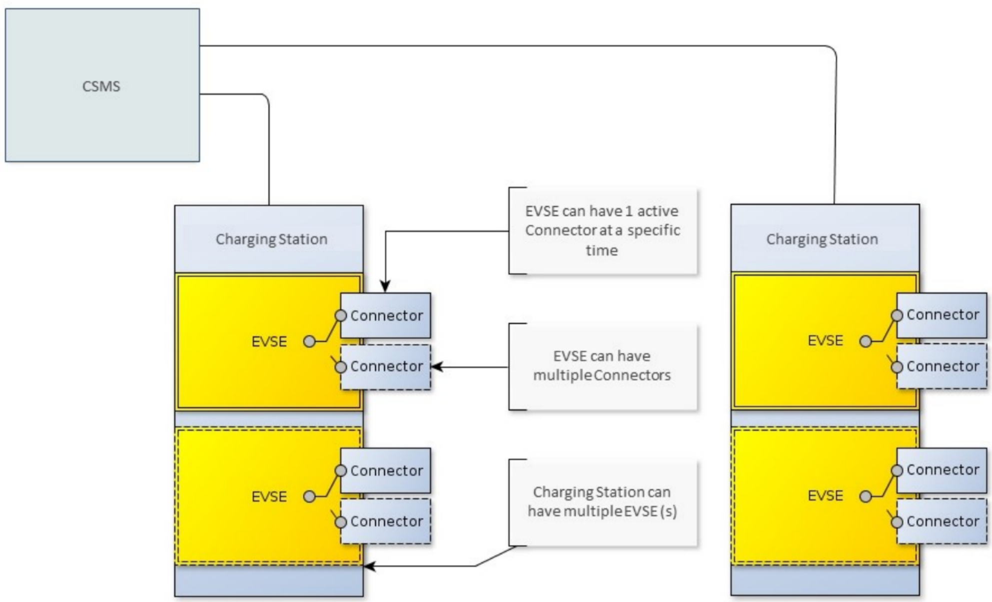

Figure 1: 3-tier model as used in OCPP [18]

Linked to these models, is the device model. The device model provides a detailed representation of the devices within the OCPP network. The attributes, capabilities, and status information of charging stations and their components are specified here. This creates effective device management, and monitoring as well as enhances the control of the OCPP system.

Together, these models enable an interoperable and scalable OCPP infrastructure capable of supporting everything from simple direct connections to complex networks involving multiple CSs and controllers.

3.2.2 CSMS - WebSocket Server

The architecture for the OCPP implementation leverages the server-less computing model offered by AWS API Gateway, to create a scalable, efficient and resilient CSMS. The architecture as mentioned centers around the AWS API Gateway, AWS Lambda and Amazon DynamoDB, facilitating communication and data management for the system.

AWS API Gateway serves as the entry point for all OCPP messages. Utilizing the WebSocket support implemented enables real-time, two-way communication between charging stations and the server-less back end, the CSMS. This approach eliminates the need for a traditional physical server for the developer, providing a "server-less" architecture that can dynamically scale in response to varying loads and requests to the CSMS. The API Gateway handles the connection and disconnection events, as well as the routing of OCPP messages to the appropriate Lambda handlers.

At the heart of the "server-less" CSMS architecture is the WebSocket server facilitated by AWS API Gateway. Unlike traditional WebSocket servers that require dedicated infrastructure to maintain persistent connections, a server-less WebSocket operates without the need for physical servers to be provisioned or managed by developers. Instead, AWS API Gateway acts as the server, managing the WebSocket connections dynamically, while scaling automatically is handled to accommodate the varying amount of active connections and messages, by dynamically allocating resources.

This server-less approach simplifies the complexity of the setup of the CSMS. It abstracts away the underlying infrastructure management for the developer, enabling developers to focus on the implementation of OCPP rather than on server maintenance, scalability, or availability concerns.

When a WebSocket connection is established, AWS API Gateway persists the connection state and routes incoming messages to the appropriate AWS Lambda function based on predefined routes, such as OnConnect, OnDisconnect, and OCPP_route. This routing allows for a decoupled, event-driven architecture where different Lambda functions can be triggered in response to specific types of messages. The use of AWS Lambda further emphasizes the server-less nature of the architecture, as these functions execute in a stateless environment, again scaling automatically with the number of requests on the CSMS.

The core of message processing lies within the OCPP_route Lambda function, which is responsible for interpreting and processing incoming OCPP messages. Depending on the message type and content, this function executes the necessary logic, which may involve querying or updating DynamoDB, and sends appropriate responses back to the charging station.

A downside of using the AWS API Gateway to create the WebSocket is the lack of configuration on specific settings on the server. Two specific things that are to be noted here, are that a WebSocket connection between the CS and the CSMS is closed after 10 minutes of inactivity. The other is that a connection between the CS and the CSMS can only be online for a maximum of two hours. If a session is to last longer than two hours, the connection is to be re-established to keep running smoothly.

Amazon DynamoDB is used to store and manage all relevant data, including charging station statuses, transaction details, live connections, and configuration settings. DynamoDB's fully managed, NoSQL database service offers a fast performance with seamless scalability, making it an ideal choice for handling the data received in a rapidly growing EV market.

Together, these components form a cohesive, server-less back-end architecture that not only reduces the complexity for the developer but also provides the flexibility and scalability needed to support the growing network of electric vehicle charging stations. The server-less model aligns with OCPP's requirements for reliable, real-time communication and efficient data management, ensuring that the infrastructure can adapt to a future increase in demands without significantly re-engineering the CSMS.

3.2.3 CS - ACDC

The client-side architecture of the Open Charge Point Protocol (OCPP) implementation is designed to meet the requirements, demanding reliability, real-time responsiveness, and efficient network communication. The implementation of the ACDC is happening in the programming language C, a language that focuses on performance and control over system resources, an architecture optimized for embedded systems.

Thread Management and Real-time Communication: Central to the client-side architecture is the creation of a separate thread for handling OCPP communication. This design decision allows the OCPP communication to operate independently of the ACDCs main control loop. The separate thread is responsible for managing WebSocket connections, sending Heartbeat messages to keep the connection alive, and processing incoming and outgoing OCPP messages while having access to all information on the ACDC.

WebSocket Communication: At the heart of the ACDC network communication for the OCPP is the WebSocket protocol, enabling bi-directional communication between the charging station and the server-less back end. Implementing and creating the WebSocket connection for the ACDC, utilizing the libraries within the already existing Zephyr environment. These libraries (should) support non-blocking socket operations and TLS encryption, creating a secure and efficient data exchange. This setup allows for real-time monitoring, remote control, and firmware updates, aligning with the OCPP's requirements for functionality and cyber security regarding the encrypted link between the CS and the CSMS. However, the currently used version of Zephyr has some bugs in the receiving algorithm used within the WebSocket library. For now, a workaround is implemented, described in section 4.3.

Security and Reliability: Recognizing the importance of security in OCPP communications, the client-side architecture incorporates robust encryption mechanisms and in the future authentication protocols. Utilizing Zephyr's security features, such as mbed TLS for encrypted data transmission, and the connection being a secure WebSocket, ensures that all messages exchanged with the CSMS are secure and encrypted.

3.2.4 Communication - secure WebSocket

The communication is as said done over a secure WebSocket connection. The format of the communication for the OCPP 2.0.1 implementation is JSON format. JSON is chosen by the OCA, for its lightweight nature and ease of use. It facilitates efficient data interchange between the charging station and the central system, ensuring that messages are compact and network bandwidth is conserved, which is crucial for the system.

The WebSocket protocol, defined in [RFC6455][19], enables full-duplex communication channels over a single TCP connection. This is essential for OCPP, which requires a persistent, real-time connection between the CS and the CSMS to support immediate execution, status updates, and monitoring. By utilizing WebSockets, the OCPP implementation can maintain an open channel for two hours, and can then immediately create a new connection whenever it is needed for seamless operability.

For the secure aspect, the secure WebSocket connections are established using Transport Layer Security (TLS), ensuring that all transmitted data is TLS encrypted. This security measure is critical to protect sensitive information related to charging transactions and to safeguard from potential eavesdropping or tampering. The communication loop, is a continuous real-time exchange of messages, providing a robust conduit for the JSON-formatted data stream between the CS and the central system. To better visualize this exchange, a schematic overview of the CS-CSMS connection will be presented in figure2, illustrating the flow of information within the OCPP architecture.

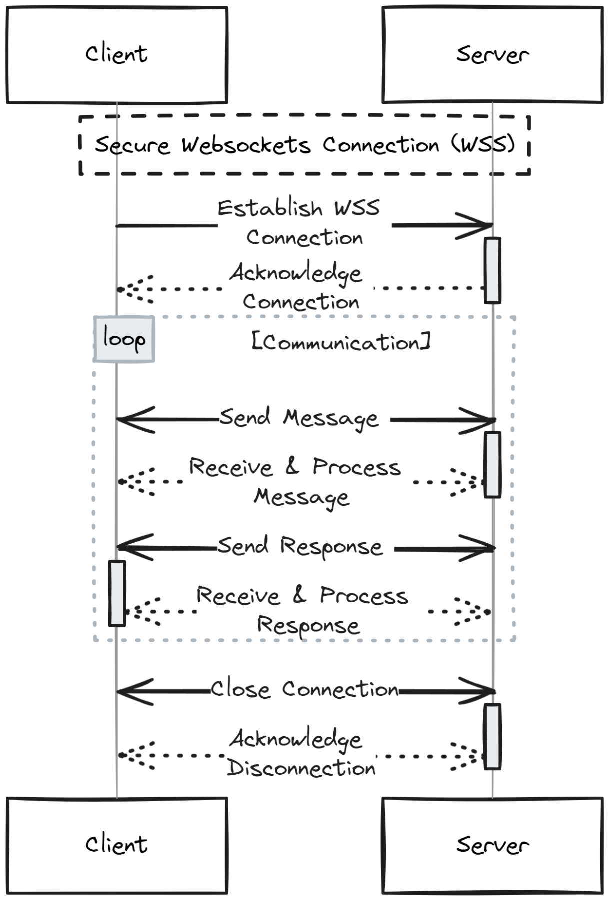

Figure 2: WebSocket Connection Overview

Furthermore, the use of JSON over WebSockets simplifies the parsing and generation of messages on both the client and server sides. It allows for a straightforward mapping of OCPP commands and responses to JavaScript Object Notation (JSON) objects, making the development and maintenance of the OCPP 2.0.1 system more manageable.

Within OCPP, there are three different messaging types; CALL, CALLRESULT, and CALLERROR. A typical chain of messages between the CS and the CSMS is a string of CALLs and CALLRESULTs and can be initiated by either of them. The three messages are structured and explained as follows in the documentation:

- CALL: The initial call, from either the CS or the CSMS, containing the action of what is to happen.

Table 3: CALL Fields [20]

Field	Datatype	Meaning
MessageTypeId	integer	This is a Message Type Number which is used to identify the type of the message.
MessageId	string[36]	This is a unique identifier that will be used to match request and result.
Action	string	The name of the remote procedure or action. This field SHALL contain a case-sensitive string. The field SHALL contain the OCPP Message name without the "Request" suffix. For example: For a "BootNotificationRequest", this field shall be set to "BootNotification".
Payload	JSON	JSON Payload of the action, see: JSON Payload for more information.


```
[
  2,
  "19223201",
  "BootNotification",
  {
    "reason": "PowerUp",
    "chargingStation": {
      "model": "SingleSocketCharger",
      "vendorName": "VendorX"
    }
  }
]
```


• CALLRESULT: Containing the same MessageId as the one received in the CALL, and the appropriate response to the CALL.

Table 4: CALLRESULT Fields [20]

Field	Datatype	Meaning
MessageTypeId	integer	This is a Message Type Number which is used to identify the type of the message.
MessageId	string[36]	This must be the exact same ID that is in the call request so that the recipient can match request and result.
Payload	JSON	JSON Payload of the action, see: JSON Payload for more information.


```
[
```

```
    3,
    "19223201",
    {
        "currentTime": "2013-02-01T20:53:32.486Z",
        "interval": 300,
        "status": "Accepted"
    }
]
```

- CALLERROR: This is only used in two cases:
  - An error occurred under the transport of the message.
  - The call has been received; however, the contents do not fulfill the criteria necessary for a valid message.

Table 5: CALLERROR Fields [20]

Field	**Datatype**	**Meaning**
MessageTypeId	integer	This is a Message Type Number which is used to identify the type of the message.
MessageId	string[36]	This must be the exact same id that is in the call request so that the recipient can match request and result.
ErrorCode	string	This field must contain a string from the RPC Framework Error Codes table.
ErrorDescription	string[255]	Should be filled in if possible, otherwise a clear empty string "".
ErrorDetails	JSON	This JSON object describes error details in an undefined way. If there are no error details you MUST fill in an empty object {}.

```
[
  4,
  "162376037",
  "NotSupported",
  "SetDisplayMessageRequest not implemented",
  {}
]
```

3.3 Test cases

Careful testing and validation of functionality is critical when implementing a system that is to be OCPP compliant. Test cases serve as a demonstration of the CS capabilities to follow the protocol's stringent requirements, in the end ensuring the reliable and efficient operation within the EV charging infrastructure. This section will explain specific test cases chosen for investigation, focusing on the BootNotification test case, which is fundamental to the initial interaction between the CS and the CSMS.

In the documentation from Open Charge Alliance [21], many different test cases are provided, and the specific mandatory test cases to be passed for each use case are defined. Here, the mandatory tests for the **Cold Boot Charging Station**, will be carried out, focusing on the `Cold Boot Charging Station - Pending` in this thesis.
The profile outlined in the core certification profile, is a set of use cases and functionalities that a charging station must support to ever be able to achieve OCPP certification. The use cases in the profile include critical functionalities such as basic authentication, security event notification, and the BootNotification procedure, among others.

3.3.1 Test Case Analysis: BootNotification

When a charging station boots up, it sends a BootNotification request to the central system, which responds with a BootNotification response. This exchange of messages is the first step in establishing a session between the two entities, allowing the central system to recognize the charging station and configure it for further use. The BootNotification test case has several checks, such as:

- Validation of the message format and data integrity.
- Ensuring the payload contains necessary information like the charger model, serial number, and vendor information.
- Verification of the central system's response, whether it accepts the boot notification and properly registers the charging station, rejects the BootNotification, or gives a "Pending" response.

The outcome of this test case determines if the charging station can proceed to the next operational steps, such as status reporting and transaction initiation. Thus, it's a fundamental test that impacts the CS entire life cycle within the OCPP network.

Testing Methodology The methodology for testing the BootNotification involves simulating the charging station's boot process, and crafting valid and invalid BootNotification requests, explained by different test cases in the test documentation [21]. This rigorous testing ensures that the charging station adheres to the protocol's standards and behaves as expected when it eventually is deployed in a production environment.

In figure 3, a detailed schematic overview illustrates the communication process during the Boot-Notification test case, providing a clear visual representation of the sequence of messages and their respective roles in the OCPP ecosystem.

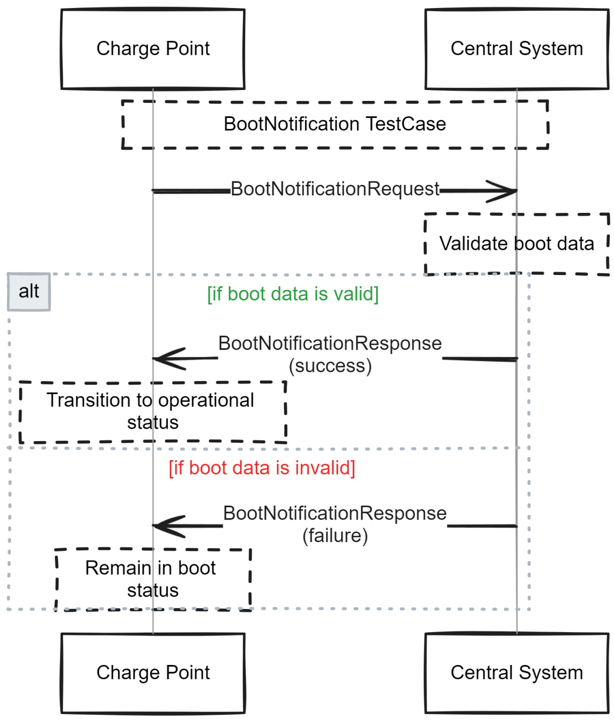

Figure 3: General BootNotification procedure

3.3.2 Cold Boot Charging Station - Pending

Multiple test cases within the BootNotification will be done; cold boot charging station - Accepted, Pending, and Rejected, with the test case IDs, TC_B_01_CS, TC_B_02_CS, and TC_B_03_CS. The case with pending will be gone through in detail, as it entails many different core functions, seen by the requirements and prerequisites described in the documentation, and displays how a series of different CALLS and CALLRESULTS, can be interacting. Moreover, the test also show-cases the functionality to get or set variables directly on the CS from the CSMS. The test case is defined as in table 6, by the Open Charge Alliance [21].

Table 6: Test case scenario for Cold Boot Charging Station - Pending

Main (Test scenario)	**Charging Station**	**CSMS**
Manual Action: Reboot the Charging Station.	1. The Charging Station sends a **BootNotificationRequest**	2. The OCTT responds with a **BootNotificationResponse** with status Pending interval <Configured heartbeatInterval>
	4. The Charging Station responds with **SetVariablesResponse**	3. OCTT sends **SetVariablesRequest** with: - variable.name = "OfflineThreshold" - component.name = "OCPPCommCtrl" - attributeValue = "300" - attributeType is omitted
	6. The Charging Station responds with **GetVariablesResponse**	5. OCTT sends **GetVariablesRequest** with: - variable.name = "OfflineThreshold" - component.name = "OCPPCommCtrl" - attributeType is omitted
	8. Charging Station responds with: **GetBaseReportResponse**	7. OCTT sends **GetBaseReportRequest** with: - requestId = <Generated requestId> - reportBase = FullInventory
	9. Charging Station responds with: **NotifyReportRequest**	10. OCTT sends **NotifyReportResponse**
	12. The Charging Station responds with a **RequestStartTransaction-Response**	11. The OCTT sends a **RequestStartTransactionRequest**
	14. The Charging Station responds with a **TriggerMessageResponse**	13. The OCTT sends a **TriggerMessageRequest** with requestedMessage BootNotification
	15. The Charging Station sends a **BootNotificationRequest**	16. The OCTT responds with a **BootNotificationResponse** with status Accepted interval <Configured heartbeatInterval>
	17. The Charging Station notifies the CSMS about the current state of all connectors.	18. The OCTT responds accordingly.

When the charging station is met with a status of pending from the OCTT (OCPP Compliance Testing Tool), a series of steps has to happen. In this test, the "OfflineThreshold" is changed/set to 300, meaning that the charging station shall send a new BootNotificationRequest in 300s if it is still offline at that time. Hereafter, the same variable is requested from the CS by the OCTT, to check that it has been successfully changed. The OCTT then requests a full inventory base report. The idea here is, to showcase that any variable can be set before the BootNotificationRequest is accepted, and controlled by the CSMS operator.

When the BootNotificationRequest has not yet been accepted, no transactions are to be allowed, which is then tested. When it has been rejected, the OCTT shall trigger a new BootNotificationRequest from the CS, which is to be accepted. At last, after the status of the BootNotificationRequest has been accepted, the availability of all connectors is to be reported to the CSMS. The expected results of this test case are defined in table 7.

Table 7: Tool validations for Cold Boot Charging Station - Pending

Test case name	**Cold Boot Charging Station - Pending**
Tool validations	*** Step 4:** **Message: SetVariablesResponse** - setVariableResult[0].attributeStatus Accepted *** Step 6:** **Message: GetVariablesResponse** - getVariableResult[0].attributeStatus Accepted *** Step 8:** **Message: GetBaseReportResponse** - status Accepted *** Step 12:** **Message: RequestStartTransactionResponse** - status Rejected *** Step 14:** **Message: TriggerMessageResponse** - status Accepted or NotImplemented *** Step 15:** **Message: BootNotificationRequest** - reason Triggered (If the status from the response from step 14 contained Accepted) *** Step 17:** **Message: StatusNotificationRequest** - connectorStatus Available **Message: NotifyEventRequest** - eventData[0].trigger Delta - eventData[0].actualValue "Available" - eventData[0].component.name "Connector" - eventData[0].variable.name "AvailabilityState" **Post scenario validations:** - A message to report the state of a connector has been received for all connectors.

3.3.3 Responsiveness

Important for the many features OCPP introduces is the responsiveness of the system. To test the responsiveness, the round-trip time it takes for a CALL is measured, as the CALL is sent from the CS, and the response from the CSMS is received.

The responsiveness is as mentioned important to many features, especially for the charging stations to be able to provide ancillary services, such as frequency containment reserve (FCR). The FCR is a critical grid ancillary service, necessitating rapid activation to counterbalance sudden frequency deviations and ensure grid stability, and is necessary to provide certain effective smart charging capabilities, through OCPP, which is shown to be possible through clusters of charging stations[22] [10].

3.3.4 Heartbeat & Multiple Connections

In addition, a test of the Heartbeat function while two different charging stations are connected, is also carried out. This is a simple but important function, to ensure that the link between the CS and the CSMS keeps intact. It sends a Heartbeat message at a predetermined interval, confirming its operational status to the CSMS. The CSMS must acknowledge each Heartbeat with a corresponding response. This test is crucial for maintaining an effective communication channel and ensuring the CS's availability is accurately reflected in the CSMS.

The WebSocket created with AWS API Gateway operates on a server-less architecture 3.2.2, and will automatically close inactive connections after a 10-minute timeout period. To maintain the connection, the Heartbeat function is configured to send a **HeartbeatRequest** whenever there have been nine minutes of inactivity on the communication link. This ensures continuous connectivity by preventing timeouts, as the connection will only be considered inactive if no messages have been exchanged for a full length of 10 minutes. Doing this while having multiple connections to the CSMS, proves that the CSMS and the CS can keep connections alive and that the system is scalable and ready to handle many connections simultaneously.

4 Implementation

This chapter describes the practical steps undertaken to realize the communication framework between the Charging Station Management System (CSMS) and the Charging Station (CS). Emphasis is placed on the initial setup required to operationalize the development environment and the subsequent establishment of a secure, serverless WebSocket connection. This implementation unfolds through the configuration of the NRF-9160 development kit board and the leveraging of AWS services for creating a responsive WebSocket server, detailing the challenges and solutions encountered in the process.

4.1 Basics/Setup

To start the implementation, all necessary programs and boards must be set up correctly and working. As mentioned, the board in the ACDC is the NRF-9160 board, and all implementation has been done on the development kit version of this, NRF-9160DK.

To work with this board, and the correct packages installed in all of the live ACDCs, the nRF connect is installed in the Visual Studio Code IDE. Furthermore, the nRF Connect SDK v2.3.0[23] has to be installed locally on the computer, to get the Zephyr RTOS environment installed.

Once the computer is set up properly, the next thing is to get the board up and running with the initial working program of the ACDC project (September 2023). The board needs to have a SIM card, for it to go online. Furthermore, the internal modem number/station address of the board needs to match a number located in the database, for it to work and boot properly. When the board runs the existing code and turns on properly, the implementation can now begin on the CS, ACDC.

4.2 The WebSocket Server - CSMS

To establish a connection with a WebSocket server, the server must first be created. Given that the existing connections and database of the ACDC project already utilize AWS services, AWS was also chosen for creating the WebSocket server. AWS API Gateway facilitates the creation of a *'server-less'*, as described in 3.2.2, action-based WebSocket server. In this context, *'action-based'* implies that incoming calls are managed through Lambda functions. Upon creation of the WebSocket server, standard routes, `$connect`, `$disconnect`, and `$default`, are configured alongside the server. These routes determine which Lambda functions are invoked, enabling direct interactions with the DynamoDB database. The `$connect` and `$disconnect` routes are automatically triggered when a new client (CS) connects to or disconnects from the WebSocket (CSMS), respectively, allowing for the execution of predefined actions. Additionally, a custom route named `$ocpp_request` is established to route all OCPP messages to a designated handler, facilitating the processing of OCPP-specific communications. [24]. In figure 4 an overview of all the interactions can be seen.

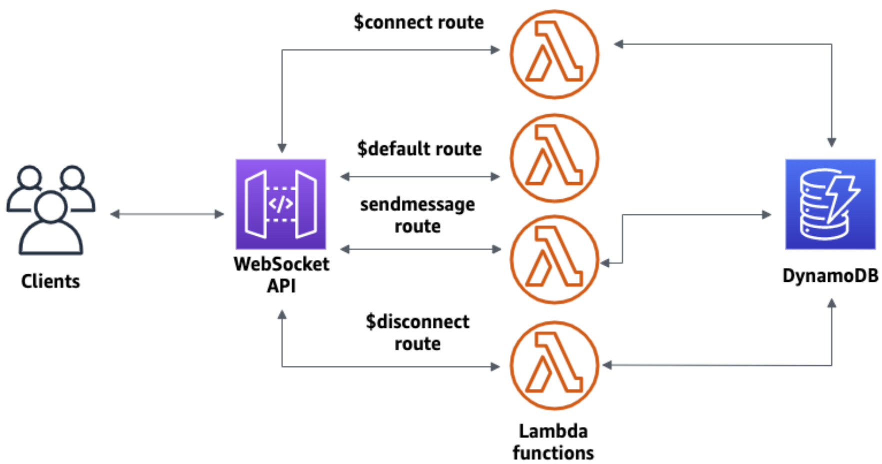

Figure 4: WebSocket API, overview of how it works [25]

When the WebSocket has been created, a URL connected to the CSMS is created. The URL format from AWS is typically as the following:

wss://API_ID.execute-api.REGION.amazonaws.com/PRODUCTION

This URL is important to remember and save (and can always be found again on AWS API Gateway), as this is the URL needed to connect to the CSMS. It will be used for all clients (CS)) trying to connect to the CSMS, and it also needs to be used directly in all of the Lambda handler functions.

4.2.1 $connect

In the connect Lambda function 7.1, new connections are handled. Whenever this is invoked, a new entry to the database holding live connections will be created. This holds the station address from the connected client, along with the "ConnectionID" given to that specific connection between the CS and the CSMS.

4.2.2 $disconnect

Opposite to the $connect function, the $disconnect function 7.1 makes sure to remove the closed connection from the same database, making sure that it is only live connections held by that database. This means, that it is possible to have an overlook of all existing connections to the WebSocket, and which specific ACDC is connected.

Furthermore, some cleanup is also handled on the disconnect route. As all OCPP CALLS have specific messageIDs, and the CALLRESULT needs to hold the same messageID, a database is created where these are stored along with their given action. As this database quickly can become large, all messageIDs older than 48 hours are deleted for now.

4.2.3 $ocpp_request

For all incoming OCPP messages, the $ocpp_request route is used 7.1. This means, all OCPP functions on the server side, CSMS, are placed and handled in this route. Depending on the received message, whether it be a CALL or a CALLRESULT, different functions are used. If it is a CALL coming from the ACDC, the CSMS handles the call, saves given values in the database, and creates a CALLRESULT to send back to the CS.

If a CALL is made from the CSMS, the messageID is stored in a database along with the accompanied action. This is done to be able to act accordingly, when the CALLRESULT is received from the ACDC, based solely on the messageID.

All of the code contained within these lambda functions can be found in the appendix 7.1.

4.3 The client - EVSE

The first step to take is to connect the board to the secure WebSocket server. This is done using Zephyrs WebSocket and BSD socket libraries. First, a TCP sock is created to the URL given by the WebSocket server. Thereafter this connection is upgraded to become a WebSocket connection, with the build in WebSocket library. For now, while developing the TLS peer verification is set to be optional, to easier be able to successfully create the connection between the ACDC and the Websocket, eg the CS and the CSMS, and focus on other implementation aspects in this step of the implementation. This process is done in the function named `int initialize_websocket(int *ws_sock)`, given in the code presented in the appendix. 7.1

To make sure the new ongoing implementation does not interfere with the working ACDC, a separate thread is created to run the OCPP communication out of the main loop, making sure the main functions are not altered because of the OCPP communication.

As the connection is established with the CSMS, messages can now be both sent and received between the two parties. However, a limitation in Zephyr v3.2.99 affects the WebSocket library's message reception, necessitating a workaround. It should be able to block for a short amount of time, to listen for incoming messages, however this is not the case. In this version, the function is either entirely blocking, meaning it will stay in the waiting stage until a message is received, or it can be non-blocking, meaning that if there is a new message inbound since the last check, it will read the oldest message in the queue. To make sure that all messages are received, a short delay is introduced to all messages sent by the CSMS, to make sure two messages are never sent almost simultaneously. To make sure that the messages are received by the CS, the created separate thread will be running many times each second, to check for new incoming messages. In newer versions of Zephyr, the receiving algorithm has been updated multiple times, meaning that these delays can probably be altered/discarded in the future, when the version will be upgraded. The Zephyr version is sometimes upgraded, as Zephyr is a part of the nRF toolbox installed on the ACDCs.

5 Results

This chapter presents the results of the test cases. The testing is done on the communication board, NRF-9160, situated in the ACDC located in the Circle Consult office in Nærum.

5.1 Tested cases

The testing of the described **Cold Boot Charging Station - Pending** is shown in detail. It will be conducted on the communication board taken directly from the ACDC placed at Circle Consult's office in Nærum. To test the communication board, it is removed from the ACDC, and a DC voltage is applied to turn it on. To flash the new code to the communication board, a J-link base classic programmer is used. The setup looks as in figure 5

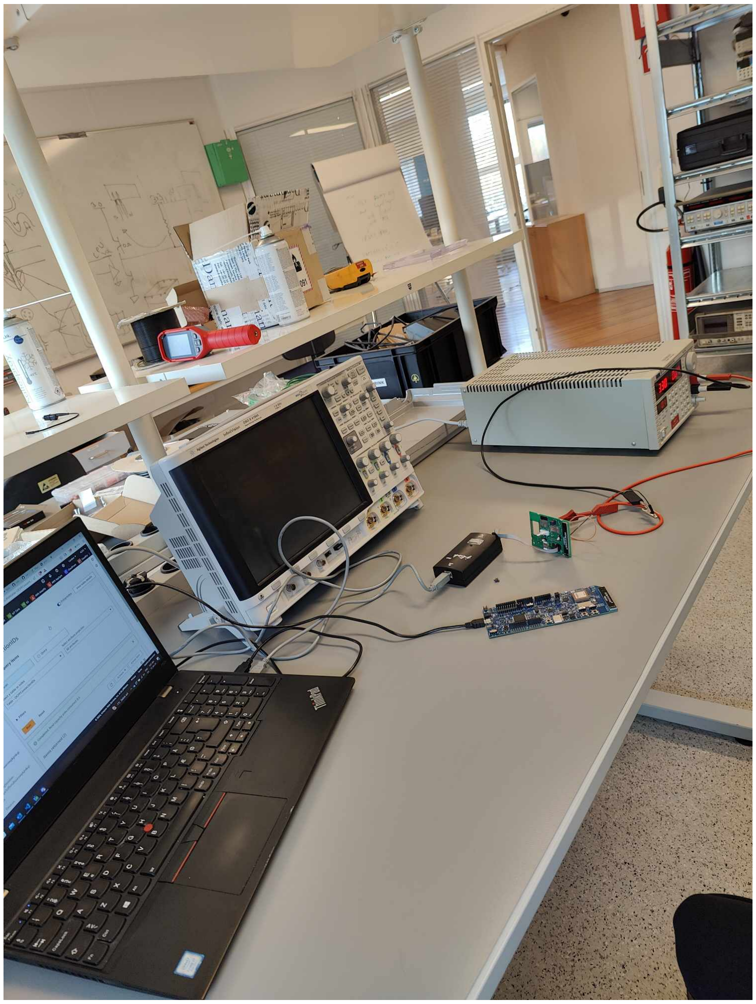

Figure 5: Test Setup With 2 Connected Devices

5.1.1 Cold Boot Charging Station - Pending

Starting the test with a **BootNotificationRequest**, as the following, yielding in all of the following messages to be received by the CS:
Initial request:


```json
[
  2,
  "81765948",
  "BootNotification",
  {
    "reason": "LocalReset",
    "chargingStation": {
      "model": "SingleSocketCharger",
      "vendorName": "VendorX"
    }
  }
]
```


Subsequent messages received by the CS:

```
[00:00:33.316,711] <inf> OCPP: Received message: [3, 81765948, {"status": "Pending", "currentTime": "2024-01-12T10:05:27.000Z", "interval": 100}]

[00:00:33.327,484] <inf> OCPP: Received message: [2, 7788320, "SetVariables", {"setVariableData": [{"component": {"name": "OCPPCommCtrlr"}, "variable": {"name": "OfflineThreshold"}, "attributeValue": 300}]}]

[00:00:33.432,464] <inf> OCPP: Received message: [2, 4520469, "GetVariables", {"getVariableData": [{"component": {"name": "OCPPCommCtrlr"}, "variable": {"name": "OfflineThreshold"}}]}]

[00:00:33.637,939] <inf> OCPP: Received message: [2, 2599330, "GetBaseReport", {"requestId": 7975112, "reportBase": "FullInventory"}]

[00:00:33.831,634] <inf> OCPP: Received message: [2, 7744396, "RequestStartTransaction", {"evseId": "", "remoteStartId": 4236631, "idToken": 1234, "chargingProfile": "", "groupIdToken": ""}]

[00:00:34.025,878] <inf> OCPP: Received message: [2, 9017299, "TriggerMessage", {"requestedMessage": "BootNotification", "evse": null}]

[00:00:37.658,142] <inf> OCPP: Received message: [3, 85377754, {}]

[00:00:38.072,814] <inf> OCPP: Received message: [3, 70216275, {"status": "Accepted", "currentTime": "2024-01-12T10:05:32.000Z", "interval": 100}]

```

```
[00:00:38.534,942] <inf> OCPP: Received message: [3, 91391541, ""]

[00:00:38.707,153] <inf> OCPP: Received message: [3, 12669709, ""]
```

Listing 1: OCPP Log Messages

At the same time, the messages received by the CSMS are as follows:

```
{"action": "OCPP_request", "message": [2, 81765948, "BootNotification", {
    "reason": "LocalReset",
    "chargingStation": {
        "vendorName": "VendorX",
        "model": "SingleSocketCharger",
        "serialNumber": "",
        "firmwareVersion": ""
    }
}]}
{"action": "OCPP_request", "message": [3, "1543325", {
    "SetVariablesResponse": [{
        "component": "OCPPCommCtrlr",
        "variable": "OfflineThreshold",
        "attributeStatus": "Accepted"
    }]
}]}
{"action": "OCPP_request", "message": [3, "4520469", {
    "getVariableResult": [{
        "component": "OCPPCommCtrlr",
        "variable": "OfflineThreshold",
        "attributeStatus": "Accepted",
        "attributeValue": "300"
    }]
}]}
{"action": "OCPP_request",
  "message": [
    3,
    "7975112",
    {
      "requestId": 7975112,
      "status": "Accepted"
    }
  ]}
{"action": "OCPP_request", "message": [2, 85377754, "NotifyReport", {
    "requestId": 2599330,
    "tbc": false,
    "seqNo": 0,
    "reportData": [{
```

```
        "component": "EVSE",
        "variable": "power_ref_amp",
        "variableAttribute": "0"
    }, {
        "component": "EVSE",
        "variable": "group_fuse",
        "variableAttribute": "0"
    }, {
        "component": "EVSE",
        "variable": "power_data",
        "variableAttribute": "0"
    }, {
        "component": "EVSE",
        "variable": "trafo_rating",
        "variableAttribute": "0"
    }, {
        "component": "EVSE",
        "variable": "distributed_amp",
        "variableAttribute": "0"
    }, {
        "component": "EVSE",
        "variable": "trafo_pi_available",
        "variableAttribute": "0"
    }, {
        "component": "OCPPCommCtrlr",
        "variable": "OfflineThreshold",
        "variableAttribute": "300"
    }]
}]}
{  "action": "OCPP_request",
  "message": [
    3,
    "7744396",
    {
      "status": "Rejected"
    }
  ]}
{"action": "OCPP_request", "message": [2, 70216275, "BootNotification", {
    "reason": "Triggered",
    "chargingStation": {
        "vendorName": "VendorX",
        "model": "SingleSocketCharger",
        "serialNumber": "SerialNumber_trigger",
        "firmwareVersion": "FirmwareVersion_trigger"
    }
```

```
84 }]}
85 {"action": "OCPP_request", "message": [2, 91391541, "StatusNotification",
    {
86    "timestamp": "1970-01-01T00:00:00Z",
87    "connectorStatus": "Available",
88    "connectorId": 1,
89    "evseId": ""
90 }]}
91 {"action": "OCPP_request", "message": [2, 12669709, "StatusNotification",
    {
92    "timestamp": "1970-01-01T00:00:00Z",
93    "connectorStatus": "Available",
94    "connectorId": 2,
95    "evseId": ""
96 }]}
```

Comparing these results with the expected results from 7, it can be seen that the desired responses are achieved. The `setVariableResult[0].attributeStatus Accepted`, is seen in line 14. The `getVariableResult[0].attributeStatus Accepted`, can be seen in line 21, with the associated value of 300, which has just been set by the SetVariablesRequest, seen in line 3 of the messages received by the CS. This continues, and most importantly it is seen that the `RequestStartTransactionResponse` is rejected and that the status of the connectors is received after the second `BootNotificationRequest` has been received and accepted with the reason being "Triggered".

This test showcases the successful implementation of the CSMS, the communication created between the CS and the CSMS, and the ability for the CSMS to read and write variables directly to the CS. This lays a solid foundation to continue further implementation of the OCPP protocol.

5.2 Responsiveness

To comprehensively assess the responsiveness of the OCPP integrated into the ACDC, a series of tests aimed at quantifying the latency in the communication loop between the Charging Station (CS) and the Charging Station Management System (CSMS) is done.

To test this, a Boot Notification Request was dispatched from the CS at an internal uptime marker of 18,209 milliseconds. The corresponding acknowledgment from the CSMS was registered at 18,876 milliseconds, creating a round-trip time of 667 milliseconds for this specific message exchange. Averaging the results over multiple trials yielded a mean communication delay of approximately 600 milliseconds for this sequence of interactions.

Such a delay is deemed acceptable within the operational parameters of the system, characterized by its promptness. It is anticipated that optimizations in the receiving algorithm could further diminish this latency. Notably, the system's responsiveness aligns with the stringent requirements for engaging as a Frequency Containment Reserve (FCR) within the electrical grid. The system's capacity to adhere to these response times makes the OCPP highly usable in supporting and maintaining grid operability through FCR.

5.3 Heartbeat & multiple connections

To test that the CSMS is capable of handling multiple connections, both the nRF-9160 board from the ACDC and the nRF-9160DK board are turned on and connected simultaneously. As shown in figure 6, both of the boards connect successfully, with their unique station address, and connectionID.

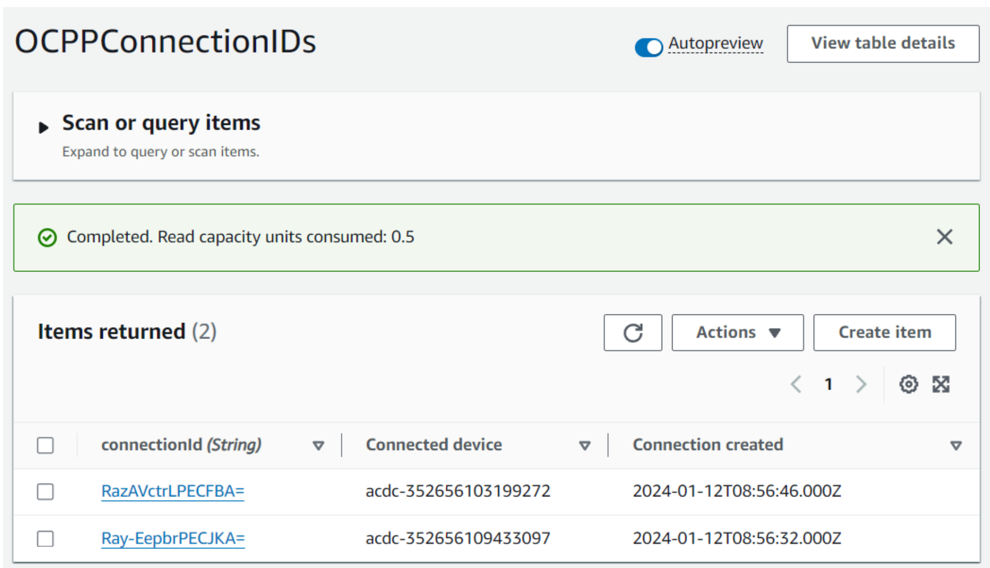

Figure 6: 2 devices connected to the CSMS, shown in the DynamoDB

At the same time, inactivity for nine minutes is introduced to the devices, resulting in the automatic `Heartbeat` function to be triggered, and answered by the CSMS. The following response is logged on the ACDC:

```
[00:09:26.323,242] <inf> OCPP: Received message: [3, "91391541", {"currentTime": "
    2024-01-12T09:06:14.000Z"}]
```

Listing 2: OCPP Log Heartbeat response

As it can be seen, a response to the `Heartbeat` request is received after nine minutes of inactivity, keeping the connection alive.

6 Discussion

In this chapter, we delve into the outcomes and insights derived from the study, focusing on the primary inquiries and goals outlined in section 1.2. The setup of the WebSocket server, communication with Charging Stations (CS), and the security of the communication link, will be discussed. Additionally, the foundational aspects of deploying the Open Charge Point Protocol (OCPP) are discussed, underscoring its critical role within the scope of this research.

6.1 OCPP v2.0.1

Choosing the wanted version of the Open Charge Point Protocol to implement, is based upon the time frame of the EV4EU project, and the many extra necessary features this version provides, especially to implement the state-of-the-art version of the OCPP. It will make the ACDC have the newest version of OCPP, and make it competitive in an increasing electric vehicle market, compared to many other charging stations, as OCPP 1.6 is the widest spread implemented version. The only true negative about version 2.0.1 of OCPP is its much more complex implementation than the implementation of OCPP 1.6.

6.2 Evaluation of implemented WebSocket server and client code

Reflecting upon the fundamental implementations this research set out to investigate and implement, an assessment can be done.

- A "server-less" WebSocket server has been implemented through AWS API Gateway. This server is able to hold multiple connections at the same time and is set up in an environment with the ability to be vastly scaled in the future. The server created is a WebSocket secure server, wss, making sure all communication over the created connection is encrypted by TLS. However, if a session is to take more than two hours, the architecture of the server will be closed, and a new connection needs to be established immediately, with saved information from before the connection was closed to continue smoothly.

- The connection created between the CSMS and the CS, is created through the mentioned wss connection. The communication is created on a new separate thread on the ACDC, for it to not change the functioning of the already existing ACDC code. In the currently used version of the Zephyr RTOS environment, flaws were found, regarding the receiving capabilities of messages for the CS, and a workaround had to be implemented. Moreover, the certificates discussed in section 2.3, part of the security profiles, have not been implemented yet. The connection created is as of now created with the certificate verification as being optional, and not used in testing.

- The creation of the fundamental function, `BootNotification` has been implemented, with a set of core functions also implemented for it to work. This function along with some other basic functions has been tested, and shown to deliver expected results, within reasonable response times.

Overall, the fundamental steps have been created, as the CSMS has been created as a WebSocket Secure Server, a secure connection is established between the CS and the CSMS, and some core OCPP functionalities have been implemented.

However, some aspects of the implementation can be wished differently, as the need for re-establishing the WebSocket connection after two hours, is not optimal. This is a known "down" side of using the server-less architecture provided by AWS API Gateway, but is chosen as the many upsides of the developer not having to manage and operate the WebSocket server directly, and the automatic scalability involved in using this service outweighs the negatives. Moreover, the need for a workaround in the communication on the receiving end of the CS is not ideal, however, this can be fixed in the future when the Zephyr environment is updated on the ACDCs.

7 Conclusion

This thesis has presented and developed the fundamental starting steps when implementing the Open Charge Point Protocol to the Autonomously Controlled Distributed Charger created by DTU and Circle Consult within the EV4EU project. The motivation behind the need for this research amounts to the very important ease of use and interoperability that the Open Charge Point Protocol (OCPP) brings to the electric vehicle charging infrastructure. By implementing OCPP, this initiative not only streamlines the interaction between charging stations and management systems but also creates a standardized ecosystem helpful to the widespread adoption of electric vehicles. This foundational work sets the stage for future developments, making the ACDC OCPP compliant, and ultimately contributing to the sustainability and resilience of the global electric vehicle market.

The created CSMS is a secure WebSocket, available to scale greatly using the architecture provided by AWS API Gateway. By having this fundamental step setup, future implementation of the OCPP should be streamlined, and the development can focus on the specifications of the OCPP provided by Open Charge Alliance (OCA).

By utilizing the existing WebSockets library within the already installed Zephyr RTOS, a secure connection is established to the CSMS from the CS, ready for real-time, bi-directional communication. Through this connection, the BootNotification function specified by the OCA is implemented.

A working test is carried out using the communication board from an ACDC located in Nærum. This showcases the working communication between the CS and the CSMS, and values are saved from the CS to the database, and values on the CS are also directly changed by request from the CSMS.

The delay that is presented between messages and actions between the CS and the CSMS is also seen to be fast enough, to make the ACDC have smart charging capabilities in the future, including the ability to act as Fast Frequency Reserve.

7.1 Future Work

For future work, the first step would be to introduce the TLS authentication certificates, to get as much security as needed when the protocol is to be tested publicly. The first goal hereafter would be to become partially OCPP compliant, implementing the core certification profile.

Further work will introduce the most needed certification profiles beyond the core profile, such as the smart charging, advanced security, or the ISO 15118 Support (Plug & Charge) certification profiles. With the ultimate goal being the ACDC becoming fully OCPP compliant with all certification profiles implemented.

Acronyms

- OCPP: Open Charge Point Protocol
- EV: Electric Vehicle
- EVSE: Electric Vehicle Supply Equipment
- OCA: Open Charge Alliance
- CS: Charging Station
- CSMS: Charging Station Management System
- OCTT: OCPP Compliance Testing Tool
- FCR: Frequency containment reserve
- ACDC: Autonomously Controlled Distributed Charger
- API: Application Programming Interface
- JSON: JSON JavaScript Object Notation
- TCP: Transmission Control Protocol
- TLS: Transport Layer Security
- WSS: WebSocket Secure
- RES: Renewable Energy Sources
- V2G: Vehicle to Grid

Bibliography

[1] Statista, Global share of CO2 emissions from fossil fuel and cement, Accessed: 2023-17-10, **2023**.

[2] International Energy Agency, Tracking Clean Energy Progress 2023, Accessed: 2024-02-14, **2023**, `https://www.iea.org/reports/tracking-clean-energy-progress-2023` (visited on 12/14/2023).

[3] AMPECO, The Complete Open Charge Point Protocol (OCPP) Guide, Accessed: 2024-02-14, **2024**.

[4] Monta, What is Open Charge Point Protocol (OCPP)?, Last updated: 7 October, 2023, **2023**.

[5] EVBox, Understanding OCPP: Why Interoperability Matters, `https://evbox.com/us-en/understanding-ocpp`, Accessed: 2023-10-14, **2023**.

[6] Greenlots, Open vs. Closed Charging Stations: Advantages and Disadvantages, tech. rep., Accessed: 2024-10-16, Greenlots, **2018**.

[7] B. Kegler, What is OCPP 2.0.1 and Why Does it Matter?, **2023**, `https://www.switch-ev.com/blog/what-is-ocpp-2-0-1-and-why-does-it-matter` (visited on 02/14/2024).

[8] Monta, Upgrade to OCPP 2.0.1: The key to advancing the EV charging infrastructure, `https://monta.com/uk/blog/upgrade-to-ocpp-2-0-1/`, Last updated: 23 May, 2023, **2023**.

[9] ChargePanel, OCPP 1.6 vs OCPP 2.0.1 - Key Differences, Updates and Functionality, Accessed: 2023-10-11, **2023**, `https://www.chargepanel.com/ocpp-1-6-vs-ocpp-2-0-1-key-differencesupdates-and-functionality/`.

[10] K. Sevdari, L. Calearo, P. Andersen, M. Marinelli, *Renewable and Sustainable Energy Reviews* **2022**, *167*, DOI `10.1016/j.rser.2022.112666`.

[11] A. Malkova, S. Striani, J. Zepter, M. Marinelli, L. Calearo in Proceedings of 58th International Universities Power Engineering Conference (UPEC 2023), 58th International Universities Power Engineering Conference, UPEC 2023 ; Conference date: 29-08-2023 Through 01-09-2023, IEEE, United States, **2023**.

[12] M. Marinelli, S. Striani, K. Pedersen, K. Sevdari, M. Hach, O. Mikkelsen, M. Rakowski, ACDC project – Autonomously Controlled Distributed Chargers: Final report, Det Energiteknologiske Udviklings- og Demonstrationsprogram, **2023**.

[13] P. Klapwijk, L. Driessen, EV Related Protocol Study: When to use which protocol? A use case based approach, tech. rep., ElaadNL, Arnhem, The Netherlands, **2017**.

[14] Open Charge Alliance, OCPP 2.0.1: Part 5 - Certification Profiles, Open Charge Alliance, **2023**.

[15] Open Charge Alliance, OCPP 2.0.1: Part 2 - Specification, Open Charge Alliance, **2022**.

[16] Zephyr Project Documentation, `https://docs.zephyrproject.org/latest/index.html`, Accessed: Feb 13, 2024, **2024**.

[17] DataScientest, AWS Lambda Explained: Unveiling the Power of Serverless Functions on Amazon Web Services, Accessed: 2023-10-25, **2023**.

[18] Open Charge Alliance, OCPP 2.0.1: Part 1 - Introduction, Open Charge Alliance, **2020**.

[19] A. Melnikov, I. Fette, The WebSocket Protocol, RFC 6455, **2011**.

[20] Open Charge Alliance, OCPP 2.0.1: Part 4 - JSON over WebSockets implementation guide, Open Charge Alliance, **2020**.

[21] Open Charge Alliance, OCPP 2.0.1: Part 6 - Test Cases, Open Charge Alliance, **2023**.

[22] N. Banol Arias, S. Hashemi, P. B. Andersen, C. Traholt, R. Romero in Proceedings of 2018 IEEE International Conference on Industrial Technology, IEEE, **2018**, pp. 1814–1819.

[23] Nordic Semiconductor, nRF Connect SDK Release Notes 2.3.0, `https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/releases_and_maturity/releases/release-notes-2.3.0.html`, Accessed: 2023-11-01, **2023**.

[24] B. A. B. Dev, AWS API Gateway Websocket Tutorial With Lambda | COMPLETELY SERVERLESS!, YouTube video, **2021**, `https://www.youtube.com/watch?v=FIrzkt7kH80` (visited on 10/30/2023).

[25] AWS, Amazon API Gateway quotas and important notes, AWS, **2023**, `https://docs.aws.amazon.com/apigateway/latest/developerguide/limits.html`.

Appendix

ACDC code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <zephyr/logging/log.h>
#include <zephyr/fs/fs.h>
#include <date_time.h>

#include <sys/time.h>

#include <zephyr/net/socket.h>
#include <zephyr/net/websocket.h>
#include <mbedtls/sha1.h>
#include <zephyr/net/net_ip.h>

#include <zephyr/net/tls_credentials.h>
#include <zephyr/shell/shell.h>
// #include <zephyr/arpa/inet.h>

#include <cJSON.h>

#include "modem.h"
#include "test_file.h"
#include "actionFunctions.h"
#include "virtual_aggregator.h"
#include "state_machine.h"
#include "validation.h"
#include "watch.h"

// Initializing the logging module for OCPP with a custom log level
LOG_MODULE_REGISTER(OCPP, CC_LOG_LEVEL);


// Variables to keep track of the time of the last received message and the
    current time
static int64_t last_received_message = 0;
static int64_t current_time_ms = 0;
static int8_t flag = 0;
static int64_t status_change_time = 0; // Timestamp when the status last changed


// OfflineThreshold: Duration in seconds after which, upon reconnection, a
    StatusNotificationRequest should be sent for each connector

bool is_offline = false;
```

```c
struct timeval offline_start_time;

// Structure to hold component and variable names
typedef struct {
    const char* componentName;
    const char* variableName;
} ComponentVariable;

// Structure for OCPP communication controller with required fields as per OCPP
    2.0.1
typedef struct {
    char* bootStatus;
    char* chargingStationVendorName;
    char* chargingStationModel;
    char* chargingStationSerialNumber;

    uint16_t* OfflineThreshold;
    uint16_t* bootInterval;
    // Additional fields based on OCPP 2.0.1 requirements
    uint16_t ResetRetries;
    char* FileTransferProtocols;  // Could be a comma-separated list of protocols
    char* NetworkConfigurationPriority;  // Comma-separated list or another format
} OCPPCommCtrlr;

// Structure for authorization controller
typedef struct {
    const char* AuthorizeRemoteStart;
} AuthCtrlr;

// Structure to keep track of recent requests and their corresponding actions
typedef struct {
    int messageId;
    char action[50]; // Adjust the size as needed
} RequestRecord;

// Define the maximum number of recent requests to track locally.
#define MAX_REQUESTS 3

// Define uint16_t variables
static uint16_t offlineThresholdValue = 0;
static uint16_t bootIntervalValue = 0;


// Declare instances of the defined structures
static OCPPCommCtrlr myOCPPCommCtrlr = {
    .bootStatus = NULL,
    .chargingStationVendorName = NULL,
    .chargingStationModel = NULL,
    .chargingStationSerialNumber = NULL,
```

```
    .OfflineThreshold = &offlineThresholdValue,
    .bootInterval = &bootIntervalValue,
    .ResetRetries = NULL,
    .FileTransferProtocols = NULL,
    .NetworkConfigurationPriority = NULL
};

static AuthCtrlr myAuthCtrlr;
static RequestRecord lastRequests[MAX_REQUESTS];
static int requestIndex = 0; // Index to keep track of the next request to overwrite


// Server configurations for WebSocket connection
#define SERVER_PORT 443
#define SERVER_ADDR4 "wss://example.execute-api.eu-west-2.amazonaws.com/production" // Replace with your IPv4 address
#define TMP_BUF_SIZE 1024


// Sample data and buffer sizes for testing
#define MAX_RECV_BUF_LEN 2048
// static uint8_t recv_buf_ipv4[MAX_RECV_BUF_LEN];
#define EXTRA_BUF_SPACE 30
static uint8_t temp_recv_buf_ipv4[MAX_RECV_BUF_LEN + EXTRA_BUF_SPACE];



// Function to initialize WebSocket connection
int initialize_websocket(int *ws_sock) {
    int ret;
    int32_t timeout = 1000;
    // const char *ip_address = "example.execute-api.eu-west-2.amazonaws.com";
    // const char *port = "443";

    struct addrinfo hints = {
        .ai_flags = 0,
        .ai_family = AF_INET,
        .ai_socktype = SOCK_STREAM,
        .ai_protocol = IPPROTO_TLS_1_2,
    };

    // The resulting address info struct.
    struct addrinfo *result;
    int ip_address = getaddrinfo("example.execute-api.eu-west-2.amazonaws.com", "443", &hints, &result);

    // Print the IP address in the result
    struct sockaddr_in *addr = (struct sockaddr_in *)result->ai_addr;
```

```c
    char ip[INET_ADDRSTRLEN];
    // Convert IP address to string format
    inet_ntop(AF_INET, &addr->sin_addr, ip, sizeof(ip));

    // Create a TCP socket
    int tcp_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TLS_1_2);

    if (tcp_sock < 0) {
        LOG_ERR("Failed to create TCP socket\n");
        return -1;
    }

    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(443); // Port for WebSocket connection

    // Set the server's IP address (replace this with the actual IP)
    if (inet_pton(AF_INET, ip, &server_addr.sin_addr) <= 0) {
        LOG_ERR("Invalid address or address not supported\n");
        close(tcp_sock);
        return -1;
    }

    // Set up TLS peer verification.
    enum {
        NONE = 0,
        OPTIONAL = 1,
        REQUIRED = 2,
    };
    int tls_verify = OPTIONAL;

    // Set the socket options for TLS verification.
    ret = setsockopt(tcp_sock, SOL_TLS, TLS_PEER_VERIFY, &tls_verify, sizeof(
   tls_verify));

    // Connect the TCP socket to the server
    ret = connect(tcp_sock, (struct sockaddr *)&server_addr, sizeof(server_addr));
    if (ret < 0) {
        LOG_ERR("Failed to connect TCP socket\n");
        close(tcp_sock);
        freeaddrinfo(result);
        return ret;
    }
    // Read the parameters for the publish message.
    char* client_id = modem_get_client_id();
    // Create the Origin header string with the client_id
    char origin_header[100];  // Adjust the buffer size as needed
    snprintf(origin_header, sizeof(origin_header), "Origin: Charger/device: %s\r\n
```

```
", client_id);
    const char *extra_headers[] = {
        origin_header,
        NULL
    };

    struct websocket_request wreq;
    memset(&wreq, 0, sizeof(wreq));

    wreq.host = "example.execute-api.eu-west-2.amazonaws.com";
    wreq.url = "/production/";
    wreq.optional_headers = extra_headers;
    wreq.tmp_buf = temp_recv_buf_ipv4;
    wreq.tmp_buf_len = sizeof(temp_recv_buf_ipv4);

    // Connect to the WebSocket server using the TCP socket
    // int timeout = 5000; // Timeout in milliseconds
    *ws_sock = websocket_connect(tcp_sock, &wreq, timeout, NULL);
    if (*ws_sock < 0) {
        LOG_ERR("Failed to connect to WebSocket server, with error: %d\n", *
ws_sock);
        close(tcp_sock);
        freeaddrinfo(result);
        return *ws_sock;
    }

    freeaddrinfo(result);
    return 0; // Success
}

// Function to handle going offline
void go_offline() {
    if (!is_offline) {
        gettimeofday(&offline_start_time, NULL);
        is_offline = true;
    }
}

// Function to handle going online
void go_online() {
    is_offline = false;
}

// Function to get the current time in a formatted string
void get_current_time_and_date(char *buffer, size_t buffer_size) {
    // Get the current uptime in milliseconds
    int64_t uptime = date_time_now(&uptime);

    // Convert uptime to seconds
```

```
    time_t now = uptime / 1000;

    // Convert time_t to tm struct
    struct tm *tm_now = gmtime(&now);

    // Format the current time and date into the provided buffer
    snprintf(buffer, buffer_size, "%04d-%02d-%02dT%02d:%02d:%02dZ",
             tm_now->tm_year + 1900, tm_now->tm_mon + 1, tm_now->tm_mday,
             tm_now->tm_hour, tm_now->tm_min, tm_now->tm_sec);
}

// Function to check and handle offline duration
void offline_duration(){
    if (is_offline) {
        struct timeval now;
        gettimeofday(&now, NULL);
        long elapsed_seconds = now.tv_sec - offline_start_time.tv_sec;

        if (elapsed_seconds > myOCPPCommCtrlr.OfflineThreshold) {
            // Offline period exceeded the threshold
            // Queue or send StatusNotification
            go_online(); // Reset offline status
        }
    }
}

// Function to handle receiving WebSocket messages
void receive_ws_message(int ws_sock) {
    uint8_t buf[MAX_RECV_BUF_LEN];
    size_t buf_len = MAX_RECV_BUF_LEN;
    uint32_t message_type;
    uint64_t remaining;
    int32_t timeout = 0; // Should be in ms, but i think a conversion is happening somewhere. 0 = non blocking.
    int result;
    int err;

    // Sends a message if no message has been received for 9 minutes.
    ocpp_heartbeat(ws_sock);
    // check_boot_status_duration(ws_sock);

    do {
        result = websocket_recv_msg(ws_sock, buf, buf_len, &message_type, &remaining, timeout);
        offline_duration();

        // LOG_INF("remaining and buf: %d, %d", remaining, buf);
        if (result < 0) {
            if (result == -EAGAIN) {
```

```
                // timeout on the websocket_recv_msg function. No message received.
                break; // Exit the loop if there's a timeout
            } else if (result == -ENOTCONN) {
                // No socket connection
                go_offline();
                LOG_ERR("No websocket connection");

                if (ws_sock) {
                    websocket_disconnect(ws_sock); // Make sure to close the existing socket
                    ws_sock = NULL;
                }
                // Reconnect to the WebSocket server
                err = initialize_websocket(&ws_sock);
                if (ws_sock) {
                    go_online(); // Update state to online if reconnection is successful
                    LOG_INF("Reconnected to websocket.");
                } else {
                    LOG_ERR("Failed to reconnect to websocket.");
                    // Handle reconnection failure (e.g., retry after a delay)
                }
                break; // Exit the loop if the connection is closed
            } else {
                // Other error conditions.
                LOG_ERR("Error receiving message: %d", result);
                // Reconnect to the WebSocket server no matter the error, unless it is nothing to read right now.
                err = initialize_websocket(&ws_sock);
                break; // Exit the loop on other errors
            }
        } else if (result > 0) {
            // Message received successfully
            // Get the timestamp in readable format.

            LOG_INF("Uptime before: %lld", k_uptime_get());
            LOG_INF("Received message: %.*s", result, buf);

            // Set time of last received websocket message
            err = date_time_now(&last_received_message);
            go_online();
            if (err) {
                LOG_ERR("Failed to get current time: %d", err);
            }

            // Reset flag
            flag = 0;

```

```c
            // Parse and process the JSON message
            cJSON *json_message = cJSON_ParseWithLength((const char *)buf, (size_t)result);
            if (json_message != NULL) {
                process_json_message(json_message, ws_sock);
                cJSON_Delete(json_message);
                LOG_INF("Uptime after: %lld", k_uptime_get());
            } else {
                const char *error_ptr = cJSON_GetErrorPtr();
                if (error_ptr != NULL) {
                    LOG_ERR("Error before: %s", error_ptr);
                }
            }
            // receive_ws_message(ws_sock); // Call the function again to check for more messages
        }
    } while (result > 0);
}

// Function to handle sending heartbeat messages over WebSocket
void ocpp_heartbeat(int ws_sock){
    // Compare current_time_ms and last_received_message
    int err;

    // Get the current time
    err = date_time_now(&current_time_ms);
    if (err) {
        // Handle error
        LOG_ERR("Failed to get current time: %d", err);
    } else {
        // Success on getting current time
    }

    // Heartbeat hardcoded to send a heartbeat after 9 minutes of no messages received, as connection closes after 10 minutes of idle time.
    int64_t time_difference_ms = current_time_ms - last_received_message;
    // const int64_t nine_minutes_in_ms = 9 * 60 * 1000;  // 9 minutes in milliseconds
    const int64_t nine_minutes_in_ms = 1 * 30 * 1000;  // 30 seconds milliseconds

    if (last_received_message > 0 && flag==0 && time_difference_ms >= nine_minutes_in_ms) {
        // Do something if last_received_message is at least 9 minutes older
        LOG_INF("The last received message is at least 9 minutes older than the current time.");
        int messageId = generate_message_id();
        // Buffer to hold the formatted message
        char ws_message[256];
        snprintf(ws_message, sizeof(ws_message), "[2, \"%d\", \"Heartbeat\", {}]",
```

```c
    messageId);
        send_ws_message(ws_sock, ws_message);
        addRequest(messageId, "Heartbeat");
        flag = 1;
    }
}


// Function to send a formatted message over WebSocket
void send_ws_message(int ws_sock, const char* message) {
    char formattedMessage[2048]; // Buffer for the formatted message.
    //  Format the message into OCPP JSON structure complying with the created server.
    snprintf(formattedMessage, sizeof(formattedMessage), "{\"action\": \"OCPP_request\", \"message\": %s}", message);
    int ret = websocket_send_msg(ws_sock, (const uint8_t*)formattedMessage, strlen(formattedMessage), WEBSOCKET_OPCODE_DATA_TEXT, true, true, -1);
    LOG_INF("Uptime when sending payload: %lld", k_uptime_get());
    if (ret < 0) {
        // Handle send error
        LOG_ERR("Failed to send message over WebSocket");
    }
}



// Function to process received JSON messages
void process_json_message(cJSON *json_message, int ws_sock) {
    // Extract messageType, messageID, action, and payload
    cJSON *messageTypeItem = cJSON_GetArrayItem(json_message, 0);
    cJSON *messageIDItem = cJSON_GetArrayItem(json_message, 1);
    int messageID = messageIDItem->valueint;

    // Check if received message is a request or a response
    if (messageTypeItem->valueint == 2) {
        cJSON *actionItem = cJSON_GetArrayItem(json_message, 2);
        cJSON *payloadItem = cJSON_GetArrayItem(json_message, 3);
        char messageIDStr[20]; // Assuming a reasonable buffer size

        // Convert the integer to a string
        sprintf(messageIDStr, "%d", messageID);
        // Handle the different requests from the server
        if (strcmp(actionItem->valuestring, "GetVariables") == 0) {
            handleGetVariablesRequest(payloadItem, messageIDStr, ws_sock);
        }
        else if (strcmp(actionItem->valuestring, "SetVariables") == 0) {
            handleSetVariablesRequest(payloadItem, messageIDStr, ws_sock);
        }
        else if (strcmp(actionItem->valuestring, "GetBaseReport") == 0) {
```

```
            get_base_report_response(payloadItem, messageIDStr, ws_sock);
        }
        else if (strcmp(actionItem->valuestring, "RequestStartTransaction") == 0){
            request_start_transaction_response(payloadItem, messageIDStr, ws_sock);
        }
        else if (strcmp(actionItem->valuestring, "TriggerMessage") == 0){
            trigger_message_response(payloadItem, messageIDStr, ws_sock);
        }
        else {
            // Handle unknown request
            LOG_ERR("Unknown request: %s", actionItem->valuestring);
        }
    } else if (messageTypeItem->valueint == 3) {
        // Received a response and act accordingly.

        cJSON *payloadItem = cJSON_GetArrayItem(json_message, 2);
        // Loop through the lastRequests array to find the matching messageId and corresponding action
        for (int i = 0; i < MAX_REQUESTS; ++i) {
            if (lastRequests[i].messageId == messageID) {
                // Found the matching messageId, now check the action
                if (strcmp(lastRequests[i].action, "BootNotification") == 0) {
                    // Perform action for BootNotification
                    handle_bootnotification_response(payloadItem, ws_sock);
                }
                else if (strcmp(lastRequests[i].action, "Heartbeat") == 0){
                    // Perform action for Heartbeat. Nothing to do here.
                }
                else {
                    // Handle unknown action
                    if (i == MAX_REQUESTS - 1){
                        LOG_ERR("Unknown action or missing action: %s", lastRequests[i].action);
                    }
                }
            }
        }
    }
}

void handle_bootnotification_response(cJSON *payload, int ws_sock){
    cJSON *statusItem = cJSON_GetObjectItem(payload, "status");
    cJSON *intervalItem = cJSON_GetObjectItem(payload, "interval");
    uint16_t interval = intervalItem->valueint;
    if (strcmp(statusItem->valuestring, "Accepted") == 0) {
        LOG_INF("Uptime when receiving accepted BootNotification Request: %lld", k_uptime_get());
        myOCPPCommCtrlr.bootStatus = "Accepted";
```

```
        status_notification_request(ws_sock, 1, "");
        status_notification_request(ws_sock, 2, "");
    } else if (strcmp(statusItem->valuestring, "Rejected") == 0) {
        myOCPPCommCtrlr.bootStatus = "Rejected";
        *myOCPPCommCtrlr.bootInterval = interval;
        status_change_time=k_uptime_get();
    } else if (strcmp(statusItem->valuestring, "Pending") == 0) {
        myOCPPCommCtrlr.bootStatus = "Pending";
        *myOCPPCommCtrlr.bootInterval = interval;
        status_change_time=k_uptime_get();
    } else {
        LOG_ERR("Unknown status from Boot notification: %s", statusItem->valuestring);
    }
}

void boot_notification_request(int ws_sock, const char* chargingStationVendor, const char* chargingStationModel,
                               const char* chargingStationSerialNumber, const char* firmwareVersion, const char* bootReason) {
    // TODO: CREATE A STRUCT TO HOLD ALL INFORMATION ABOUT THE CHARGING STATION.
    // Assign default values if NULL is passed.
    chargingStationVendor = chargingStationVendor ? chargingStationVendor : "";
    chargingStationModel = chargingStationModel ? chargingStationModel : "";
    chargingStationSerialNumber = chargingStationSerialNumber ? chargingStationSerialNumber : "";
    firmwareVersion = firmwareVersion ? firmwareVersion : "";
    bootReason = bootReason;

    cJSON *requestRoot = cJSON_CreateArray();
    cJSON_AddItemToArray(requestRoot, cJSON_CreateNumber(2)); // Message Type ID for Call
    int messageId = generate_message_id();
    cJSON_AddItemToArray(requestRoot, cJSON_CreateNumber(messageId)); // Unique message ID
    cJSON_AddItemToArray(requestRoot, cJSON_CreateString("BootNotification")); // Action

    // Creating the payload for the BootNotification request
    cJSON *requestPayload = cJSON_CreateObject();
    cJSON_AddItemToObject(requestPayload, "reason", cJSON_CreateString(bootReason)); // Boot reason is a required field
    cJSON_AddItemToObject(requestPayload, "chargingStation", cJSON_CreateObject());
    cJSON *chargingStation = cJSON_GetObjectItem(requestPayload, "chargingStation");

    // Model and vendorname are required fields. Serial number and firmware version are optional.
```

```c
    cJSON_AddItemToObject(chargingStation, "vendorName", cJSON_CreateString(chargingStationVendor));
    cJSON_AddItemToObject(chargingStation, "model", cJSON_CreateString(chargingStationModel));
    cJSON_AddItemToObject(chargingStation, "serialNumber", cJSON_CreateString(chargingStationSerialNumber));
    cJSON_AddItemToObject(chargingStation, "firmwareVersion", cJSON_CreateString(firmwareVersion));

    cJSON_AddItemToArray(requestRoot, requestPayload);

    // Convert cJSON object to string (JSON format)
    char *requestMessage = cJSON_Print(requestRoot);

    // Send the request message over WebSocket
    send_ws_message(ws_sock, requestMessage);

    // Add the messageId and action to the lastRequests array.
    addRequest(messageId, "BootNotification");

    // Free the cJSON object and string
    free(requestMessage);
    cJSON_Delete(requestRoot);
}

// Function to check if the bootStatus has been "Rejected" or "Pending" for more than bootInterval
void check_boot_status_duration(int ws_sock) {

    if (myOCPPCommCtrlr.bootStatus != NULL && myOCPPCommCtrlr.bootInterval != NULL) {
        int64_t current_time = k_uptime_get(); // Get current uptime in milliseconds
        // status_change_time is set when a bootnotification is sent and the status is "Rejected" or "Pending".

        // Check if the status has just changed to "Rejected" or "Pending"
        if ((strcmp(myOCPPCommCtrlr.bootStatus, "Rejected") == 0 || strcmp(myOCPPCommCtrlr.bootStatus, "Pending") == 0)) {
            // Calculate the elapsed time since the status last changed
            int64_t elapsed_time = current_time - status_change_time;

            // Convert bootInterval to milliseconds for comparison
            int64_t boot_interval_ms = *(myOCPPCommCtrlr.bootInterval) * 1000;

            // Check if elapsed time is greater than the bootInterval
            if (elapsed_time > boot_interval_ms) {
                // Reset the status change time
                status_change_time = k_uptime_get();
```

```
                boot_notification_request(ws_sock, "VendorX", "SingleSocketCharger", NULL, NULL, "PowerUp");
            }
        }
    }
}


void handleGetVariablesRequest(cJSON *payload, const char* messageId ,int ws_sock) {
    // cJSON *messageId = cJSON_GetArrayItem(payload, 1); // Assuming messageId is part of the payload
    cJSON *getVariableData = cJSON_GetObjectItem(payload, "getVariableData");
    int size = cJSON_GetArraySize(getVariableData);


    cJSON *responseRoot = cJSON_CreateArray();
    cJSON_AddItemToArray(responseRoot, cJSON_CreateNumber(3));
    cJSON_AddItemToArray(responseRoot, cJSON_CreateString(messageId));
    cJSON *responsePayload = cJSON_CreateObject();
    cJSON *getVariableResult = cJSON_AddArrayToObject(responsePayload, "getVariableResult");

    for (int i = 0; i < size; i++) {
        cJSON *item = cJSON_GetArrayItem(getVariableData, i);
        cJSON *componentName = cJSON_GetObjectItem(item, "component");
        cJSON *componentNameStr = cJSON_GetObjectItem(componentName, "name");
        cJSON *variableName = cJSON_GetObjectItem(item, "variable");
        cJSON *variableNameStr = cJSON_GetObjectItem(variableName, "name");

        // Fetch the actual value for each variable
        char *variableValue = getVariableValue(variableNameStr->valuestring, componentNameStr->valuestring);

        cJSON *resultItem = cJSON_CreateObject();
        cJSON_AddItemToArray(getVariableResult, resultItem);
        cJSON_AddItemToObject(resultItem, "component", cJSON_CreateString(componentNameStr->valuestring));
        cJSON_AddItemToObject(resultItem, "variable", cJSON_CreateString(variableNameStr->valuestring));
        cJSON_AddItemToObject(resultItem, "attributeStatus", cJSON_CreateString("Accepted"));
        cJSON_AddItemToObject(resultItem, "attributeValue", cJSON_CreateString(variableValue));
    }

    cJSON_AddItemToArray(responseRoot, responsePayload);
    char *responseMessage = cJSON_Print(responseRoot);
    send_ws_message(ws_sock, responseMessage);  // Send the response over
```

```
    WebSocket

    free(responseMessage);
    cJSON_Delete(responseRoot);
}


void handleSetVariablesRequest(cJSON *payload, const char* messageId, int ws_sock) {
    cJSON *setVariableData = cJSON_GetObjectItem(payload, "setVariableData");
    int size = cJSON_GetArraySize(setVariableData);

    // Prepare the response array
    cJSON *responseArray = cJSON_CreateArray();

    for (int i = 0; i < size; i++) {
        cJSON *item = cJSON_GetArrayItem(setVariableData, i);
        cJSON *componentName = cJSON_GetObjectItem(item, "component");
        cJSON *componentNameStr = cJSON_GetObjectItem(componentName, "name");
        cJSON *variableName = cJSON_GetObjectItem(item, "variable");
        cJSON *variableNameStr = cJSON_GetObjectItem(variableName, "name");
        cJSON *value = cJSON_GetObjectItem(item, "attributeValue");

        // Determine the status of the variable setting
        char *status = "Rejected"; // or "Rejected" based on your logic

        if (cJSON_IsString(variableNameStr) && (strcmp(variableNameStr->valuestring, "OfflineThreshold") == 0)) {

            *myOCPPCommCtrlr.OfflineThreshold = (value->valueint);
            status = "Accepted";
            // Additional logic to determine if setting is successful
        }else if (cJSON_IsString(variableNameStr) && (strcmp(variableNameStr->valuestring, "OfflineThreshold") != 0)){
            status = "UnknownVariable";
        }

        // Construct response for this variable
        cJSON *responseItem = cJSON_CreateObject();
        cJSON_AddItemToObject(responseItem, "component", cJSON_CreateString(componentNameStr->valuestring));
        cJSON_AddItemToObject(responseItem, "variable", cJSON_CreateString(variableNameStr->valuestring));
        cJSON_AddItemToObject(responseItem, "attributeStatus", cJSON_CreateString(status));
        cJSON_AddItemToArray(responseArray, responseItem);
    }

    // Construct the complete response
```

```
    cJSON *completeResponse = cJSON_CreateArray();
    cJSON_AddItemToArray(completeResponse, cJSON_CreateNumber(3)); // Message Type ID for CALLRESULT
    cJSON_AddItemToArray(completeResponse, cJSON_CreateString(messageId));
    cJSON_AddItemToArray(completeResponse, cJSON_CreateObject());
    cJSON *lastItem = cJSON_GetArrayItem(completeResponse, 2);
    cJSON_AddItemToObject(lastItem, "SetVariablesResponse", responseArray);

    // Convert to string and send
    char *responseString = cJSON_Print(completeResponse);
    send_ws_message(ws_sock, responseString);

    // Free memory
    free(responseString);
    cJSON_Delete(completeResponse);
}


// Function to create and send GetBaseReportResponse based on request payload
void get_base_report_response(cJSON *requestPayload,const char* messageId , int ws_sock) {
    // Extract requestId and reportBase from the request payload

    cJSON *requestIdItem = cJSON_GetObjectItem(requestPayload, "requestId");
    cJSON *reportBaseItem = cJSON_GetObjectItem(requestPayload, "reportBase");

    int requestId = requestIdItem ? requestIdItem->valueint : 0;
    const char *reportBase = reportBaseItem ? reportBaseItem->valuestring : "";

    // Determine the status based on reportBase
    char *status;
    if (strcmp(reportBase, "FullInventory") == 0 || strcmp(reportBase, "SummaryInventory") == 0 || strcmp(reportBase, "ConfigurationInventory") == 0) {
        status = "Accepted";
    } else {
        status = "NotSupported"; // If the reportBase is not supported or recognized
    }

    // Create the response JSON object
    cJSON *response = cJSON_CreateObject();
    cJSON_AddItemToObject(response, "requestId", cJSON_CreateNumber(requestId));
    cJSON_AddItemToObject(response, "status", cJSON_CreateString(status));

    // Create the complete OCPP response array
    cJSON *completeResponse = cJSON_CreateArray();
    cJSON_AddItemToArray(completeResponse, cJSON_CreateNumber(3));  // Message Type ID for CALLRESULT
    cJSON_AddItemToArray(completeResponse, cJSON_CreateString(messageId));
```

```
    // cJSON_AddItemToArray(completeResponse, cJSON_CreateNumber(requestId));
    cJSON_AddItemToArray(completeResponse, response);

    // Convert to string and send via WebSocket
    char *responseString = cJSON_Print(completeResponse);
    send_ws_message(ws_sock, responseString);

    // Send NotifyReportRequest if status is Accepted. Otherwise, the request is ignored.
    if (strcmp(status, "Accepted") == 0) {
        // Send NotifyReportRequest
        notify_report_request(requestId, reportBase, false, 0, ws_sock);
    }

    // Free memory
    free(responseString);
    cJSON_Delete(completeResponse);
}


// Function to create and send NotifyReportRequest
void notify_report_request(int requestId, const char* reportBase, bool tbc, int seqNo, int ws_sock) {
    cJSON *reportDataArray = cJSON_CreateArray();
    // Add report data items to reportDataArray
    // Each item in the array should be a JSON object representing a piece of report data

    if (strcmp(reportBase, "FullInventory") == 0) {

        // Add outlet and avaliability of the outlet to the reportDataArray.
        const ComponentVariable variables[] = {
            {"EVSE", "power_ref_amp"},
            {"EVSE", "group_fuse"},
            {"EVSE", "power_data"},
            {"EVSE", "trafo_rating"},
            {"EVSE", "distributed_amp"},
            {"EVSE", "trafo_pi_available"},
            {"OCPPCommCtrlr", "OfflineThreshold"}
            // Add more component-variable pairs as needed
        };
        int numVariables = sizeof(variables) / sizeof(variables[0]);

        for (int i = 0; i < numVariables; ++i) {
            cJSON *dataItem = cJSON_CreateObject();
            const char* variableValue = getVariableValue(variables[i].variableName, variables[i].componentName);
            cJSON_AddItemToObject(dataItem, "component", cJSON_CreateString(variables[i].componentName));
```

```c
            cJSON_AddItemToObject(dataItem, "variable", cJSON_CreateString(variables[i].variableName));
            cJSON_AddItemToObject(dataItem, "variableAttribute", cJSON_CreateString(variableValue));
            cJSON_AddItemToArray(reportDataArray, dataItem);
        }
    } else if (strcmp(reportBase, "SummaryInventory") == 0) {
        cJSON *dataItem = cJSON_CreateObject();
        cJSON_AddItemToObject(dataItem, "component", cJSON_CreateString("ExampleComponent"));
        cJSON_AddItemToObject(dataItem, "variable", cJSON_CreateString("ExampleVariable"));
        cJSON_AddItemToObject(dataItem, "variableAttribute", cJSON_CreateString("ExampleValue"));
        cJSON_AddItemToArray(reportDataArray, dataItem);
    } else if (strcmp(reportBase, "ConfigurationInventory") == 0) {
        cJSON *dataItem = cJSON_CreateObject();
        cJSON_AddItemToObject(dataItem, "component", cJSON_CreateString("ExampleComponent"));
        cJSON_AddItemToObject(dataItem, "variable", cJSON_CreateString("ExampleVariable"));
        cJSON_AddItemToObject(dataItem, "variableAttribute", cJSON_CreateString("ExampleValue"));
        cJSON_AddItemToArray(reportDataArray, dataItem);
    }

    // Create NotifyReportRequest JSON
    cJSON *notifyReportRequest = cJSON_CreateObject();
    cJSON_AddItemToObject(notifyReportRequest, "requestId", cJSON_CreateNumber(requestId));
    cJSON_AddItemToObject(notifyReportRequest, "tbc", cJSON_CreateBool(tbc));
    cJSON_AddItemToObject(notifyReportRequest, "seqNo", cJSON_CreateNumber(seqNo));
    cJSON_AddItemToObject(notifyReportRequest, "reportData", reportDataArray);

    // Create the complete OCPP message array
    cJSON *completeMessage = cJSON_CreateArray();
    cJSON_AddItemToArray(completeMessage, cJSON_CreateNumber(2));  // Message Type ID for CALL
    int messageId = generate_message_id();
    cJSON_AddItemToArray(completeMessage, cJSON_CreateNumber(messageId));  // Unique message ID
    cJSON_AddItemToArray(completeMessage, cJSON_CreateString("NotifyReport"));  // Action
    cJSON_AddItemToArray(completeMessage, notifyReportRequest);

    // Convert to string and send via WebSocket
    char *messageString = cJSON_Print(completeMessage);
    send_ws_message(ws_sock, messageString);
```

```

    // Add the message ID and action to the lastRequests array.
    addRequest(messageId, "NotifyReport");

    // Free memory
    free(messageString);
    cJSON_Delete(completeMessage);
}

void request_start_transaction_response(cJSON *requestPayload, const char* messageId, int ws_sock){

    // cJSON *evseId = cJSON_GetObjectItem(requestPayload, "evseId"); // Unused for now, but can be retreived if needed.
    // cJSON *remoteStartId = cJSON_GetObjectItem(requestPayload, "remoteStartId"); // Unused for now, but can be retreived if needed.
    // cJSON *idToken = cJSON_GetObjectItem(requestPayload, "idToken"); // Unused for now, but can be retreived if needed.
    // cJSON *chargingProfile = cJSON_GetObjectItem(requestPayload, "chargingProfile"); // Unused for now, but can be retreived if needed.
    // cJSON *groupIdToken = cJSON_GetObjectItem(requestPayload, "groupIdToken"); // Unused for now, but can be retreived if needed.


    cJSON *responseRoot = cJSON_CreateArray();
    cJSON_AddItemToArray(responseRoot, cJSON_CreateNumber(3)); // Message Type ID for CallResult
    cJSON_AddItemToArray(responseRoot, cJSON_CreateString(messageId));

    // Creating the payload for the response
    cJSON *responsePayload = cJSON_CreateObject();

    // Basic response, used for testing.
    if (strcmp(myOCPPCommCtrlr.bootStatus, "Accepted")==0){
        cJSON_AddItemToObject(responsePayload, "status", cJSON_CreateString("Accepted"));
    } else{
        cJSON_AddItemToObject(responsePayload, "status", cJSON_CreateString("Rejected"));
    }


    cJSON_AddItemToArray(responseRoot, responsePayload);

    // Convert cJSON object to string (JSON format)
    char *responseMessage = cJSON_Print(responseRoot);

    // Send the response message over WebSocket
    send_ws_message(ws_sock, responseMessage);
```

```

    // Free the cJSON object and string
    free(responseMessage);
    cJSON_Delete(responseRoot);
}


void trigger_message_response(const char* Payload, const char* messageId, int ws_sock) {

    cJSON *triggerAction = cJSON_GetObjectItem(Payload, "requestedMessage");


    // Check which message is requested and prepare the response
    cJSON *responseRoot = cJSON_CreateArray();
    cJSON_AddItemToArray(responseRoot, cJSON_CreateNumber(3)); // Message Type ID for CallResult
    cJSON_AddItemToArray(responseRoot, cJSON_CreateString(messageId));

    cJSON *responsePayload = cJSON_CreateObject();

    if (strcmp(triggerAction->valuestring, "BootNotification") == 0) {
        cJSON_AddItemToObject(responsePayload, "status", cJSON_CreateString("Accepted"));
        cJSON_AddItemToArray(responseRoot, responsePayload);
        // Convert cJSON object to string (JSON format)
        char *responseMessage = cJSON_Print(responseRoot);

        // Send the response message over WebSocket
        send_ws_message(ws_sock, responseMessage);

        boot_notification_request(ws_sock, "VendorX", "SingleSocketCharger", "SerialNumber_trigger", "FirmwareVersion_trigger", "Triggered");
        free(responseMessage);
    } else {
        // Handle other message types or unknown message
        cJSON_AddItemToObject(responsePayload, "status", cJSON_CreateString("Rejected"));
        cJSON_AddItemToArray(responseRoot, responsePayload);
        // Convert cJSON object to string (JSON format)
        char *responseMessage = cJSON_Print(responseRoot);

        // Send the response message over WebSocket
        send_ws_message(ws_sock, responseMessage);
        free(responseMessage);
    }

    // Free the cJSON object and string
    cJSON_Delete(responseRoot);
```

```c
}

void status_notification_request(int ws_sock, int connectorId, const char* evseId)
    {
    cJSON *requestRoot = cJSON_CreateArray();
    cJSON_AddItemToArray(requestRoot, cJSON_CreateNumber(2)); // Message Type ID
   for Call
    int messageId = generate_message_id();
    cJSON_AddItemToArray(requestRoot, cJSON_CreateNumber(messageId)); // Unique
   message ID
    cJSON_AddItemToArray(requestRoot, cJSON_CreateString("StatusNotification"));
   // Action

    // Create the current time and date string
    char time_buffer[30];
    char* connector_status;
    // Getting the entire struct



    if (connectorId == 1){
        // charge_point_t *charge_point_1_ptr = get_charge_point_1();
        // state_t outlet1 = charge_point_1_ptr->state;
        state_t outlet1 = get_state_1();
        if (outlet1 == STATE_IDLE){
            connector_status = "Available";
        } else if (outlet1 == STATE_CONNECTED){
            connector_status = "Occupied";
        } else if (outlet1 == STATE_CHARGING){
            connector_status = "Occupied";
        } else {
            connector_status = "Faulted";
        }
    }else if (connectorId == 2){
        // charge_point_t *charge_point_2_ptr = get_charge_point_2();
        state_t outlet2 = get_state_2();
        if (outlet2 == STATE_IDLE){
            connector_status = "Available";
        } else if (outlet2 == STATE_CONNECTED){
            connector_status = "Occupied";
        } else if (outlet2 == STATE_CHARGING){
            connector_status = "Occupied";
        } else {
            connector_status = "Faulted";
        }
    }else {
        connector_status = "Faulted";
    }

```

```
    get_current_time_and_date(time_buffer, sizeof(time_buffer));

    // Creating the payload for the StatusNotification request
    cJSON *requestPayload = cJSON_CreateObject();
    cJSON_AddItemToObject(requestPayload, "timestamp", cJSON_CreateString(
    time_buffer));

    cJSON_AddItemToObject(requestPayload, "connectorStatus", cJSON_CreateString(
    connector_status)); // Change to get the status from the EVSE.
    cJSON_AddItemToObject(requestPayload, "connectorId", cJSON_CreateNumber(
    connectorId));
    cJSON_AddItemToObject(requestPayload, "evseId", cJSON_CreateString(evseId));

    cJSON_AddItemToArray(requestRoot, requestPayload);

    // Convert cJSON object to string (JSON format)
    char *requestMessage = cJSON_Print(requestRoot);

    // Send the request message over WebSocket
    send_ws_message(ws_sock, requestMessage);

    // Free the cJSON object and string
    free(requestMessage);
    cJSON_Delete(requestRoot);
}


char* getVariableValue(const char* variableName, const char* componentName) {
    static char value[20]; // Buffer to hold the string representation
    // va_charge_point_t *va_charge_point_1_ptr = get_va_charge_point(); //Dont
    know how to get this pointer correctly with information.

    if (strcmp(componentName, "EVSE")==0){
        if (strcmp(variableName, "power_ref_amp") == 0) {
            snprintf(value, sizeof(value), "%u", get_power_ref_amp_value());
        } else if (strcmp(variableName, "group_fuse") == 0) {
            snprintf(value, sizeof(value), "%u", get_group_fuse());
        } else if (strcmp(variableName, "power_data") == 0) {
            power_data_t* power_data = get_power_data();
            snprintf(value, sizeof(value), "%u", power_data->data_source[3]);
        } else if (strcmp(variableName, "trafo_rating") == 0) {
            snprintf(value, sizeof(value), "%u", get_trafo_rating());
        } else if (strcmp(variableName, "distributed_amp") == 0) {
            snprintf(value, sizeof(value), "%u", get_distributed_amp());
        } else if (strcmp(variableName, "trafo_pi_available") == 0) {
            snprintf(value, sizeof(value), "%u", get_trafo_pi_available());
        } else {
            strncpy(value, "Unknown Variable", sizeof(value));
        }
```

```
    } else if (strcmp(componentName, "OCPPCommCtrlr") == 0) {
        if (strcmp(variableName, "OfflineThreshold") == 0) {
            snprintf(value, sizeof(value), "%hu", *myOCPPCommCtrlr.OfflineThreshold);
            // strncpy(value, myOCPPCommCtrlr.OfflineThreshold, sizeof(value) - 1);
            // value[sizeof(value) - 1] = '\0'; // Ensure null-termination
        } else {
            strncpy(value, "Unknown Variable", sizeof(value));
        }
    } else {
        strncpy(value, "Unknown Variable", sizeof(value));
    }
    return value;
}


// Function to add a messageID and corresponding action to the lastRequests array. Holds 3 at the time and overwrites the oldest.
void addRequest(int messageId, const char* action) {
    lastRequests[requestIndex].messageId = messageId;
    strncpy(lastRequests[requestIndex].action, action, sizeof(lastRequests[requestIndex].action) - 1);
    lastRequests[requestIndex].action[sizeof(lastRequests[requestIndex].action) - 1] = '\0'; // Ensure null-termination

    requestIndex = (requestIndex + 1) % MAX_REQUESTS; // Update index in a circular manner
}

// Function to generate a random message ID between 1 and 99999999
int generate_message_id() {
    return rand() % 99999999 + 1;
}

void perform_test(int ws_sock, char* test_case){
    // Test case TC_B_01_CS
    if (strcmp(test_case, "Boot_accepted") == 0){
            boot_notification_request(ws_sock, "VendorX", "SingleSocketCharger", NULL, NULL, "PowerUp");
    }
    // Test case TC_B_02_CS
    else if (strcmp(test_case, "Boot_pending") == 0){
            boot_notification_request(ws_sock, "VendorX", "SingleSocketCharger", NULL, NULL, "LocalReset");
    }
    // Test case TC_B_03_CS
    else if (strcmp(test_case, "Boot_rejected") == 0){
            boot_notification_request(ws_sock, "VendorX", "SingleSocketCharger",
```

```
    NULL, NULL, "Unknown");
    }
}

// Function to create a thread for handling the incoming websocket communication.
void websocket_thread(void *data) {
    int ws_sock = *(int *)data; // Cast and dereference the passed-in socket

    const char* ocppHearbeat = "[2, \"19223201\",\"Heartbeat\", {}]";
    // send_ws_message(ws_sock, ocppHearbeat);

    k_sleep(K_MSEC(10000));

    // k_sleep(K_MSEC(2000));

    int8_t flag2 = 0;

    perform_test(ws_sock, "Boot_pending");
    // perform_test(ws_sock, "Boot_accepted");

    while (1) {
        // Call your receive_ws_message function

        // This can be a blocking and non blockin call, depending on timeout value within websocket_recv_msg().
        // If blocking, the heartbeat function will not be called until a message is received, but receiving messages properly will be ensured.
        // If non blocking, the heartbeat function will be called every time the receive_ws_message function is called, but messages might be lost/queued
        // untill next incoming messages are received. Maybe call heartbeat in main thread instead? Remember this is blocking now, so it cant
        // send messages over the websocket while waiting for a message to be received. Timeout within the websocket_recv_msg() function is bugged in this zephyr version,
        // and is fixed in newer versions of zephyr.
        receive_ws_message(ws_sock);

        // Add any additional logic needed for handling WebSocket communication

        if (k_uptime_get() > 40000 && flag2 == 0){
            flag2 = 1;
            // change_state_from_other_file();
            // k_sleep(K_MSEC(2000));
            // perform_test(ws_sock, "Boot_pending");
            // websocket_disconnect(ws_sock);
        }

        k_sleep(K_MSEC(10)); // Sleep for a short duration to yield CPU time
    }
```

```
}
```

Listing 3: ACDC code

Custom Validation Code (client side)

Listing 4: connect route

WebSocket Server code

$connect code

```python
import json
import boto3
import time

client = boto3.client('apigatewaymanagementapi',endpoint_url="https://example.
    execute-api.eu-west-2.amazonaws.com/production")

# Initialize DynamoDB client
dynamodb = boto3.resource('dynamodb')
table_connectionId = dynamodb.Table('OCPPConnectionIDs')


def lambda_handler(event, context):
    print(event)
    print("****")
    print(context)

    connectionId = event["requestContext"]["connectionId"]

    # Extract the Origin header
    headers = event.get("headers", {})
    origin = headers.get("Origin", "")

    # Extract charger/device number from the Origin header
    charger_device_number = origin.split(": ")[-1] if origin else "Unknown"

    # Log the time of the connection
    currentTime = time.strftime("%Y-%m-%dT%H:%M:%S.000Z", time.gmtime())

    # Check for existing entry with the same charger_device_number
    response = table_connectionId.scan(
        FilterExpression=boto3.dynamodb.conditions.Attr('Connected device').eq(
    charger_device_number)
    )

    # Delete existing entry if found
    for item in response.get('Items', []):
        old_connection_id = item['connectionId']
        table_connectionId.delete_item(
            Key={'connectionId': old_connection_id}
        )


    # Store connectionId and charger/device number in DynamoDB
    response = table_connectionId.put_item(
```

```
        Item={
            'connectionId': connectionId,
            'Connected device': charger_device_number,
            'Connection created': currentTime
        }
    )

    return {"statusCode":200}
```

Listing 5: connect route

$disconnect code

```
import time
import boto3
from boto3.dynamodb.conditions import Key, Attr


# Initialize DynamoDB client
dynamodb = boto3.resource('dynamodb')
table_connectionId = dynamodb.Table('OCPPConnectionIDs')
message_id_table = dynamodb.Table('OCPPMessageIDs')

def lambda_handler(event, context):
    print(event)

    # Extract connectionId from the event
    connectionId = event["requestContext"]["connectionId"]

    # Delete the entry with the given connectionId from DynamoDB
    response = table_connectionId.delete_item(
        Key={'connectionId': connectionId}
    )

    # Delete messageIDs older than 60 minutes.
    delete_old_messages()

    # Optional: Add error handling based on the response

    return {"statusCode": 200}



def delete_old_messages():
    # Calculate the time threshold (60 minutes ago)
    time_threshold = time.strftime("%Y-%m-%dT%H:%M:%S.000Z", time.gmtime(time.time
    () - 3600)) #3600s = 1h = 60m

    # Query for messages older than 60 minutes
    response = message_id_table.scan(
        FilterExpression=Attr('time').lt(time_threshold)
    )
    messages = response.get('Items', [])

    # Delete each old message
    for message in messages:
        message_id_table.delete_item(
            Key={
                'messageId': message['messageId']
            }
```

```
            )
```

Listing 6: disconnect route

$ocpp_handler code

```
import json
import urllib3
import boto3
import time
import os
import random

client = boto3.client('apigatewaymanagementapi',endpoint_url="https://example.
    execute-api.eu-west-2.amazonaws.com/production")

# Initialize DynamoDB client
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('Devices')
message_id_table = dynamodb.Table('OCPPMessageIDs')
table_connectionId = dynamodb.Table('OCPPConnectionIDs')


last_send_time = None

def get_station_address(connectionId):
    # Query DynamoDB for the item with the specified connectionId
    response = table_connectionId.get_item(Key={'connectionId': connectionId})

    # Check if the item was found
    if 'Item' in response:
        # Extract the charger/device number.
        charger_device_number = response['Item'].get('Connected device', None)
        return charger_device_number
    else:
        # Item not found, handle accordingly
        return None


# Create a timer mechanism to make sure to never send two messages in a row. (
    could introduce a timer in the database to ensure this over each
# called lambda handler to the same client)
# Call this before posting to client.
def ensure_message_delay(delay=0.2):
    global last_send_time
    current_time = time.time()

    if last_send_time is not None:
        elapsed_time = current_time - last_send_time
        if elapsed_time < delay:
            time.sleep(delay - elapsed_time)
    last_send_time = time.time()

```

```python

def lambda_handler(event, context):
    print(event)
    connectionId = event["requestContext"]["connectionId"]
    message_data = json.loads(event["body"])
    inner_message_data = message_data["message"]
    message_id = inner_message_data[1]
    message_type = inner_message_data[0]

    if message_type == 2:  # Request from Client
        handle_client_request(connectionId, inner_message_data)
    elif message_type == 3:  # Response from Client
        handle_client_response(connectionId, inner_message_data)
        # response_message = {"Handled a response type call"}
        # response = client.post_to_connection(ConnectionId=connectionId, Data=
json.dumps(response_message).encode('utf-8'))
        pass
    elif message_type == 4:  # Error from Client
        response_message = {"Handle an error call"}
        ensure_message_delay()
        response = client.post_to_connection(ConnectionId=connectionId, Data=json.
dumps(response_message).encode('utf-8'))
        pass
    return {"statusCode": 200}


def handle_client_request(connectionId, message):
    message_id = message[1]
    action = message[2]
    payload = message[3]

    if action == "BootNotification":
        # This sends a response message.
        boot_notification_response(connectionId, payload, message_id)

    elif action == "StatusNotification":
        connectorStatus = payload["connectorStatus"]
        evseId = payload["evseId"]
        connectorId = payload["connectorId"]
        # timestamp = payload["timestamp"] # This is not working properly on the
board at the moment.
        timestamp = currentTime = time.strftime("%Y-%m-%dT%H:%M:%S.000Z", time.
gmtime())
        # Determine the attribute name based on connectorId
        status_attr = f"ConnectorStatus{connectorId}"
        response_message = [3, message_id,""]
        station_address_used = get_station_address(connectionId)

        response_dynamodb = table.update_item(
```

```
            Key={'station_address': station_address_used},
            UpdateExpression=f"set {status_attr} = :c, ConnectorChecked = :t",
            ExpressionAttributeValues={
                ':c': connectorStatus,
                ':t': timestamp
            },
            ReturnValues="UPDATED_NEW"
        )
        ensure_message_delay()
        response = client.post_to_connection(ConnectionId=connectionId, Data=json.dumps(response_message).encode('utf-8'))
        # Update database to set the appropriate status of the connectorId (outlet), and set the status of the charging station to idle.
        pass
    elif action == "Heartbeat":
        send_get_variables_request(connectionId,None,"baseReport") # Used heartbeat to test the call.
        currentTime = time.strftime("%Y-%m-%dT%H:%M:%S.000Z", time.gmtime())
        response_message = [3, message_id,{
            "currentTime": currentTime,
        }]
        ensure_message_delay()
        response = client.post_to_connection(ConnectionId=connectionId, Data=json.dumps(response_message).encode('utf-8'))
        ensure_message_delay()
        pass
    elif action == "Authorize":
        id = payload["idToken"]
        type = payload[""]
        ensure_message_delay()
        response = client.post_to_connection(ConnectionId=connectionId, Data=json.dumps(response_message).encode('utf-8'))
        pass
    elif action == "NotifyReport":
        response_message = [3, message_id,{}]
        ensure_message_delay()
        response = client.post_to_connection(ConnectionId=connectionId, Data=json.dumps(response_message).encode('utf-8'))




# Handle a response from the EVSE
def handle_client_response(connectionId, message):
    messageId = message[1]
    payload = message[2]
    action = get_action_from_messageId(messageId)

    # Check what action is associated with the response.
```

```
    if action == 'GetVariables':
        handle_get_variables_response(connectionId, payload)
    elif action == 'SetVariables':
        # insert function to handle logic for the status of the setVariables request.
        pass
    elif action == "RequestStartTransaction":
        # Handle the response from the RequestStartTransaction request.
        pass


def send_get_variables_request(connectionId, componentName, variableName):
    # Generate a unique messageId (random for now, can be made incrementing)
    messageId = random.randint(1000000, 9999999)

    # Store messageId in DynamoDB
    store_messageId_in_database(messageId, 'GetVariables')

    if variableName == "baseReport":
        get_variables_request = [2, messageId, "GetVariables", {
            "getVariableData": [
                {"component": {"name": "EVSE"}, "variable": {"name": "power_ref_amp"}},
                {"component": {"name": "EVSE"}, "variable": {"name": "group_fuse"}},
                {"component": {"name": "EVSE"}, "variable": {"name": "power_data"}},
                {"component": {"name": "EVSE"}, "variable": {"name": "trafo_rating"}},
                {"component": {"name": "EVSE"}, "variable": {"name": "distributed_amp"}},
                {"component": {"name": "EVSE"}, "variable": {"name": "trafo_pi_available"}}
            ]
        }]
    else:
        get_variables_request = [2, messageId, "GetVariables", {
            "getVariableData": [
                {"component": {"name": componentName}, "variable": {"name": variableName}},
            ]
        }]
    ensure_message_delay()
    client.post_to_connection(ConnectionId=connectionId, Data=json.dumps(get_variables_request).encode('utf-8'))


def boot_notification_response(connectionId, payload, message_id):
    # Extracting specific values from the inner message_data
```

```
    reason = payload["reason"]
    charging_station = payload["chargingStation"]
    model = charging_station["model"]
    vendor_name = charging_station["vendorName"]

    store_messageId_in_database(message_id,"BootNotification")

    status = ""

    currentTime = time.strftime("%Y-%m-%dT%H:%M:%S.000Z", time.gmtime())

    if reason == "PowerUp":
        status = "Accepted"
        interval = 100 #Interval not needed, as it is accepted. Interval = time to wait to resend bootnotification.
    elif reason == "LocalReset":
        status = "Pending"
        interval = 100
    elif reason == "Triggered":
        status = "Accepted"
        interval = 100
        ensure_message_delay(0.15)
    elif reason == "Unknown":
        status = "Rejected"
        interval = 10
    else:
        status = "Rejected"
        interval = 100

    station_address_used = get_station_address(connectionId)

    response_dynamodb = table.update_item(
        Key={'station_address': station_address_used},
        UpdateExpression="set Model = :m, VendorName = :v, #S = :s, LastBootTime = :t",
        ExpressionAttributeValues={
            ':m': model,
            ':v': vendor_name,
            ':s': status,
            ':t': currentTime
        },
        ExpressionAttributeNames={
            '#S': 'Status'  # Substitute 'Status' with a placeholder
        },
        ReturnValues="UPDATED_NEW"
    )

    # Form a response message
    response_message = [3,message_id,{
```

```
        "status": status,
        "currentTime": currentTime,
        "interval": interval
    }]
    ensure_message_delay()
    client.post_to_connection(ConnectionId=connectionId, Data=json.dumps(response_message).encode('utf-8'))


    # Handle pending status (Test TC_B_02_CS)
    if status == "Pending" and reason == "LocalReset":
        # Set to default set OfflineThreshold = 300 right now.
        # ensure_message_delay()
        set_variable_request(connectionId,None,None)
        # ensure_message_delay()
        send_get_variables_request(connectionId, "OCPPCommCtrlr", "OfflineThreshold")
        # ensure_message_delay()
        get_base_report_request(connectionId,"FullInventory")
        # ensure_message_delay()
        request_start_transaction_request(connectionId, idToken=1234)
        # ensure_message_delay()
        send_trigger_message_request(connectionId,"BootNotification")
    pass

def set_variable_request(connectionId, variableName, variableValue):
    # Generate a unique messageId (random for now, can be made incrementing)
    messageId = random.randint(1000000, 9999999)

    # Store messageId in DynamoDB
    store_messageId_in_database(messageId, 'SetVariables')

    # For a start, set the offline threshold for the bootnotification pending test.
    set_variables_request = [2, messageId, "SetVariables", {
        "setVariableData": [
            {"component": {"name": "OCPPCommCtrlr"}, "variable": {"name": "OfflineThreshold"}, "attributeValue": 300},
        ]
    }]
    ensure_message_delay()
    client.post_to_connection(ConnectionId=connectionId, Data=json.dumps(set_variables_request).encode('utf-8'))



def get_base_report_request(connectionId, reportBase):
    messageId = random.randint(1000000, 9999999)

```

```
    # reportBase = FullInventory, SummaryInventory or ConfigurationInventory.
    get_base_report_request = [2, messageId, "GetBaseReport", {
        "requestId": messageId,
        "reportBase": reportBase  # e.g., "FullInventory", "SummaryInventory", etc.
    }]
    ensure_message_delay()
    client.post_to_connection(ConnectionId=connectionId, Data=json.dumps(get_base_report_request).encode('utf-8'))



def request_start_transaction_request(connectionId, idToken, evseId = None, chargingProfile = None, groupIdToken = None):
    # Generate a unique messageId (random for now, can be made incrementing)
    messageId = random.randint(1000000, 9999999)

    # remoteStartId number. Probably also save this in the database for later use.
    remoteStartId = random.randint(1000000, 9999999)

    # Store messageId in DynamoDB
    store_messageId_in_database(messageId, 'RequestStartTransaction')

    # Create RequestStartTransactionRequest message
    # For a start, set the offline threshold for the bootnotification pending test.
    request_start_transaction_request = [2, messageId, "RequestStartTransaction", {
        "evseId": evseId if evseId else "",
        "remoteStartId": remoteStartId,
        "idToken": idToken,
        "chargingProfile": chargingProfile if chargingProfile else "",
        "groupIdToken": groupIdToken if groupIdToken else "",
    }]
    ensure_message_delay()
    client.post_to_connection(ConnectionId=connectionId, Data=json.dumps(request_start_transaction_request).encode('utf-8'))



def send_trigger_message_request(connectionId, requestedMessage, evse = None):
    """
    Send a TriggerMessage request to the charging station.

    :param connectionId: WebSocket connection ID to the charging station.
    :param requestedMessage: The type of message to trigger, eg "BootNotification".
    """
    messageId = random.randint(1000000, 9999999)  # Generate a unique messageId
```

```

    # Store messageId in DynamoDB with the action 'TriggerMessage'
    store_messageId_in_database(messageId, 'TriggerMessage')

    # Create TriggerMessage request
    trigger_message_request = [2, messageId, "TriggerMessage", {
        "requestedMessage": requestedMessage, # eg "BootNotification"
        "evse": evse if evse else None,
    }]

    # Send the request over WebSocket
    ensure_message_delay()
    client.post_to_connection(ConnectionId=connectionId, Data=json.dumps(trigger_message_request).encode('utf-8'))



def get_action_from_messageId(message_id):
    # Retrieve the item with the given messageId from DynamoDB
    response = message_id_table.get_item(Key={'messageId': message_id})

    if 'Item' in response:
        item = response['Item']
        return item.get('action', None)  # Returns the action type if available
    else:
        return None  # Returns None if the messageId is not found



def handle_get_variables_response(connectionId, payload):
    # Process the payload to extract and store data
    for result in payload.get('getVariableResult', []):
        component_name = result['component']  # Access 'component' directly
        variable_name = result['variable']    # Access 'variable' directly
        value = result['attributeValue']

        # Update the database with the new values
        update_database_with_variable(connectionId, component_name, variable_name, value)


def store_messageId_in_database(messageId, action):
    # Store the messageId in DynamoDB
    currentTime = time.strftime("%Y-%m-%dT%H:%M:%S.000Z", time.gmtime())
    message_id_table.put_item(
        Item={
            'messageId': str(messageId),  # Storing messageId as a string
            'action': action,
            'status': 'sent',
```

```python
            'time': currentTime,
        }
    )


def update_database_with_variable(connectionId, component_name, variable_name,
    value):
    try:
        station_address_used = get_station_address(connectionId)
        response = table.update_item(
            Key={'station_address': station_address_used},
            UpdateExpression="set #var = :v",
            ExpressionAttributeNames={'#var': variable_name},
            ExpressionAttributeValues={':v': value},
            ReturnValues="UPDATED_NEW"
        )
        return response
    except Exception as e:
        # Handle any exceptions (e.g., log the error, return an error message)
        print(f"Error updating item: {e}")
        return None
```

Listing 7: ocpp request route