UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JOSÉ MÁRIO REISSWITZ

Implementação de uma plataforma backend para integração com protocolo OCPP

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Profa. Dra. Érika Cota
Co-orientador: Prof. Dr. Luigi Carro

Porto Alegre
2024

“All we have to decide is what to do
with the time that is given us.”
— J.R.R. TOLKIEN

AGRADECIMENTOS

Agradeço principalmente aos meus pais, por todo o apoio, suporte e incentivo aos estudos. Expresso também minha gratidão à minha companheira, Juliana Siebert, por ter me incentivado na reta final do curso. Sou grato aos amigos que fiz durante os anos de faculdade, que, de alguma forma, foram essenciais para que eu chegasse até a monografia. Por fim, mas não menos importante, agradeço à Professora Dra. Érika Cota pela excelente orientação.

RESUMO

A *start-up* SAVE enfrenta dificuldades na integração de seu sistema com o protocolo OCPP devido às limitações das soluções atuais no mercado, que frequentemente são caras e dependem de serviços pagos. Este trabalho apresenta o desenvolvimento de um servidor *backend* robusto e escalável, capaz de realizar a integração necessária com o protocolo OCPP. Além disso, o servidor oferecerá uma API para suportar as operações diárias da empresa, fornecendo dados essenciais como informações de carregamento, lista de estações conectadas e outras métricas relevantes. O objetivo é reduzir a dependência de soluções pagas, melhorar a eficiência operacional e permitir que a SAVE ofereça um serviço mais competitivo e acessível.

Palavras-chave: OCPP. Arquitetura de Software. Microsserviços.

Implementation of a backend platform for integration with the OCPP protocol

ABSTRACT

The start-up SAVE faces difficulties in integrating its system with the OCPP protocol due to the limitations of current market solutions, which are often expensive and reliant on paid services. This work presents the development of a robust and scalable backend server capable of performing the necessary integration with the OCPP protocol. Additionally, the server will offer an API to support the company's daily operations, providing essential data such as charging information, a list of connected stations, and other relevant metrics. The goal is to reduce dependence on paid solutions, improve operational efficiency, and enable SAVE to offer a more competitive and accessible service.

Keywords: OCPP, Software Architecture, Microservices.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AMQP	Advanced Message Queuing Protocol
CPMS	Charging Point Management System
CSV	Comma separeted values
DDD	Domain-driven Design
FIFO	First-in First-out
OCPP	Open Charge Point Protocol
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
TCP	Transmission Control Protocol
RFID	Radio-frequency identification
SAVE	South America Vehicle Electrification (empresa)
SMTP	Simple Mail Transfer Protocol
SSL	Secure Sockets Layer
URL	Uniform Resource Locator
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Com a popularização dos carros elétricos tornou-se necessária a padronização de um protocolo de comunicação aberto para infraestruturas de carregamento de carros elétricos, visando a interoperabilidade e a integração eficiente com os serviços de rede elétrica. Em resposta a esta demanda surgiu o OCPP (*Open Charge Point Protocol*), do inglês, Protocolo Aberto de Estações de Carregamento [Schmutzler, Andersen e Wietfeld 2013].

O OCPP é um protocolo de aplicação aberto que estabelece comunicação entre as estações de carregamento e o sistema de gerenciamento de estação de carregamento (do inglês, CPMS - *Charging Point Management System*). Ele é mantido pela *Open Charge Alliance*, organização formada por empresas com a missão de promover o desenvolvimento global, a adoção e a conformidade dos protocolos de comunicação na infraestrutura de carregamento de veículos elétricos [Venkata Pruthvi et al. 2019] e [OCA 2024].

O CPMS é um software complexo que serve como ponto central na gestão de estações de carregamento de carros elétricos em uma determinada rede. Nele são implementadas as lógicas de controle de carga, autorização, monitoramento, etc. [Venkata Pruthvi et al. 2019]. Ele conecta-se à estação de carga de maneira bi-direcional, recebendo e enviando comandos e operações através do protocolo OCPP, enquanto a estação de carga conecta-se aos carros elétricos.

A popularização dos carros elétricos não é algo apenas fora do Brasil: de acordo com a Associação Brasileira do Veículo Elétrico, as vendas de veículos leves eletrificados cresceram em todas as regiões do país em 2023 [ABVE 2024]. Dentro desse contexto, encontra-se a *start-up* SAVE (*South America Vehicle Electrification*, ou, pelo nome fantasia, Serviços de Abastecimento de Veículos Elétricos).

Fundada em 2022, a SAVE é uma empresa incubada no Centro de Empreendimentos em Informática (CEI) da UFRGS, com o objetivo de oferecer serviços de apoio ao carregamento de veículos elétricos. Tais serviços incluem a instalação de carregadores elétricos em um primeiro momento, mas, principalmente, um aplicativo que traga comodidade e segurança ao usuário do veículo. Esse objetivo tem se mostrado desafiador devido à relativa novidade da tecnologia. Por exemplo, a disponibilidade de carregadores é ainda incipiente, embora crescente.

Do ponto de vista do software, a *start-up* precisa de uma plataforma que permita monitorar seus recursos, como estações de carregamento em condomínios, e seus usos, como dados de carregamentos realizados nessas estações. No entanto, há uma carência

de boas alternativas no mercado que façam essa comunicação com a estação e o protocolo OCPP, função típica de um CPMS. As opções *open-source* (código aberto) disponíveis oferecem integração difícil ou incompleta, enquanto as alternativas pagas são caras e não se encaixam no orçamento de uma *start-up*.

Com isto em vista, a empresa entende que a melhor opção acaba sendo uma implementação própria, tendo maior controle sobre as funcionalidades, podendo ser até uma vantagem competitiva. Os requisitos funcionais para essa aplicação são:

- Listagem de estações: deve-se ser possível visualizar todas as estações cadastradas e verificar o estado de cada uma, como em uso, disponível, com defeito, entre outros.

- Dados de uma estação: cada estação deve possuir informações detalhadas associadas a ela, incluindo nome, endereço postal, quantidade de carregadores, e outros dados relevantes.

- Dados de um carregador de uma estação: Como cada estação tem um número fixo de carregadores, é essencial poder monitorar o estado individual de cada carregador, indicando se estão em uso, disponíveis, ou apresentando erros.

- Dados de carga: para a *start-up*, é crucial monitorar o uso de seus equipamentos, o que torna importante a disponibilidade de um histórico detalhado de todas as cargas realizadas.

Os requisitos técnicos para essa implementação são:

- Flexibilidade: deve-se ser possível alterar regras de negócio com facilidade, tendo em vista o cenário de rápida evolução e mudanças de uma *start-up*.

- Robustez: a solução deve ser robusta e lidar bem com eventuais erros, incluindo problemas de conexão com carregadores, dados inseridos incorretamente por usuários, etc.

- Escalabilidade: A solução deve ser capaz de escalar com facilidade, tanto para suportar um uso mais intenso conforme a empresa cresça, quanto para economizar recursos computacionais em momentos de pouco tráfego de dados ou uso.

1.1 Objetivos

O objetivo geral deste trabalho é desenvolver uma plataforma *backend* que suporte as principais operações do protocolo OCPP, coletando e disponibilizando esses dados de forma simplificada por meio de uma API (*Application Programming Interface* do inglês, Interface de Programação de Aplicações). Além disso, é crucial que essa aplicação seja de fácil manutenção e flexível para novos casos de uso, considerando a rápida evolução das necessidades de uma *start-up*.

1.2 Estrutura do Trabalho

Este trabalho está organizado da seguinte maneira: o Capítulo 2 revisa principais conceitos utilizados ao longo do estudo. O Capítulo 3 apresenta uma proposta de arquitetura, infraestrutura necessária e casos de uso para o sistema. O Capítulo 4 apresenta como a plataforma foi implementada, baseando-se no que foi proposto no Capítulo 3, incluindo desde tecnologias escolhidas, técnicas de desenvolvimento utilizadas e como os casos de uso propostos foram desenvolvidos. O Capítulo 5 conclui o trabalho, recapitulando o que foi feito, o que não foi possível concluir, limitações técnicas e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Ao longo deste capítulo, serão apresentados conceitos e definições importantes para a contextualização e compreensão deste trabalho. Na Seção 2.1 serão detalhados conceitos de arquiteturas de sistemas *Backend*, apresentando protocolos, técnicas e modelos escolhidos para a concepção do sistema proposto. Na Seção 2.2 serão abordados alguns detalhes do protocolo OCPP, como operações, dados enviados pelo carregador para o CPMS, etc.

2.1 Arquitetura de sistemas

Esta seção trata dos conceitos utilizados na arquitetura e infraestrutura de sistemas *backend*. Os assuntos explorados vão desde conceitos e padrões utilizados, como microsserviços, escalabilidade, *publisher-subscriber*, até protocolos, como HTPP, AMQP, SMTP, etc.

2.1.1 Modelo cliente-servidor

O modelo cliente-servidor é uma arquitetura de rede onde o cliente faz solicitações de serviços e recursos ao servidor, que os processa e responde de acordo. Por exemplo, dada uma plataforma de vendas online, o cliente pode solicitar uma lista de produtos à venda, o servidor a fornece, e então o cliente pode pedir mais detalhes de algum produto específico, etc [Tanenbaum e Steen 2007].

A comunicação entre cliente e servidor pode ocorrer de forma síncrona ou assíncrona. Na comunicação síncrona, o cliente envia uma solicitação e aguarda a resposta do servidor antes de prosseguir, o que pode resultar em latência se o servidor demorar a responder. Já na comunicação assíncrona, o cliente envia uma solicitação e pode continuar com outras tarefas enquanto espera pela resposta do servidor, que será processada assim que recebida. A comunicação assíncrona pode melhorar a eficiência e a capacidade de resposta do sistema, especialmente em aplicações que exigem alta interatividade ou que operam em ambientes distribuídos com latência variável. Em contrapartida, a comunicação síncrona é mais simples de implementar e pode ser suficiente para aplicações com requisitos de tempo de resposta previsíveis [Tanenbaum e Steen 2007].

2.1.2 Escalabilidade

Uma aplicação no mundo real raramente tem uma demanda de processamento constante. Por exemplo, plataformas web podem ter muitos usuários ativos durante o dia e pouco tráfego durante a noite, sistema de vendas online podem experimentar um aumento significativo de tráfego durante eventos de vendas especiais, como natal ou feriados, e uma demanda menor durante o restante do ano, etc.

Com base nisso é desejável que um sistema implemente estratégias de escalabilidade, para que em momentos de maior demanda seja possível usar mais *hardware*, e em momentos de menor demanda usar menos, para assim economizar recursos [Fowler 2016].

Segundo [Fowler 2016] existem duas estratégias principais para escalabilidade:

- Escalabilidade Vertical: O serviço escala utilizando hardware mais potente, como uma CPU mais rápida ou maior quantidade de memória RAM. Esse tipo de escalabilidade é relativamente simples de implementar. No entanto, pode não ser a solução mais desejável, pois os recursos disponíveis podem se esgotar. Por exemplo, se a aplicação já está utilizando o melhor hardware possível e ainda não consegue atender à demanda, essa abordagem se torna limitada.

- Escalabilidade Horizontal: O serviço escala adicionando mais máquinas ao sistema, distribuindo a carga de trabalho entre múltiplos servidores. Esse tipo de escalabilidade é altamente desejável, pois permite que o sistema continue a crescer conforme a demanda aumenta, sem depender das limitações de um único hardware. A escalabilidade horizontal oferece maior flexibilidade e pode melhorar a redundância e a resiliência do sistema. No entanto, implementar essa abordagem pode ser mais complexo, pois requer a gestão de balanceamento de carga, sincronização de dados entre servidores e possivelmente mudanças na arquitetura do software para suportar a distribuição.

2.1.3 Microsserviços

A arquitetura de microsserviços propõe um sistema composto por vários serviços menores, cada um com responsabilidades específicas e independentes. Em contraste, a arquitetura monolítica agrupa todas as responsabilidades e funcionalidades dentro de um

único serviço. Enquanto a abordagem monolítica pode ser mais simples de desenvolver inicialmente, ela tende a se tornar complexa e difícil de manter à medida que o sistema cresce. Em contrapartida, os microsserviços permitem maior flexibilidade e escalabilidade, facilitando a atualização, a manutenção e a implementação contínua, já que cada serviço pode ser desenvolvido, implantado e escalado de forma independente [Fowler 2016].

Nessa arquitetura os serviços devem seguir os seguintes requisitos:

- Independência: cada serviço deve operar de forma autônoma, comunicando-se através de interfaces bem definidas, como APIs ou *message brokers*.
- Modularidade: cada serviço deve ter um foco específico, tendo poucas responsabilidades.
- Escalabilidade: cada serviço deve ser capaz de escalar de modo horizontal, dando maior flexibilidade para a gestão de recursos de hardware

Seguindo esses pré-requisitos é possível atingir um sistema com as seguintes vantagens:

- Agilidade no desenvolvimento: com os serviços menores a manutenção acaba ficando mais simples e intuitiva.
- Desenvolvimento independente: cada serviço, sendo independente, pode adotar quaisquer tecnologias que satisfaçam seus requisitos técnicos, sem se preocupar com tecnologias de serviços vizinhos.
- Escalabilidade e Flexibilidade: possibilita escalar individualmente os microsserviços de acordo com a demanda, otimizando o uso de recursos.

Também há desafios associados a essa arquitetura, como maior dificuldade de testes ponta a ponta, complexidade de dividir as responsabilidade entre os serviços de forma que não hajam dependências ou acoplamentos, etc. [Fowler 2016]

2.1.4 *Publisher-Subscriber*

O padrão *publisher-subscriber* é amplamente utilizado em sistemas distribuídos para facilitar a comunicação assíncrona entre diferentes componentes. Nesse modelo,

os *publishers* (do inglês, publicadores) enviam mensagens a um intermediário, frequentemente um *message broker*, sem conhecimento prévio de quais *subscribers* (do inglês, assinantes) irão receber essas mensagens. Os *subscribers* se inscrevem para receber tipos específicos de mensagens do *broker*, permitindo uma separação clara entre a produção e o consumo de dados. Essa abordagem promove uma arquitetura desacoplada, onde os componentes podem ser adicionados ou modificados sem impactar diretamente os outros. Além disso, o padrão *publisher-subscriber* melhora a escalabilidade e a flexibilidade do sistema, pois permite a distribuição da carga de trabalho e a adaptação a mudanças de demanda de maneira mais eficiente [Stoja, Vukmirovic e Jelacic 2013].

Segundo [Stoja, Vukmirovic e Jelacic 2013], a distribuição de mensagens num sistema *publisher-subscriber* pode ser feita de duas formas:

- Baseada em tópicos: mensagens são enviadas para diferentes tópicos pelos produtores e os consumidores se inscrevem nele. Todos os consumidores inscritos em um dado tópico receberão todas as mensagem enviadas para ele.
- Baseada em conteúdo: funciona como um filtro, onde mensagens só vão ser enviadas para os consumidores se os atributos ou conteúdo da mensagem foram satisfeitos pelos filtros definidos por ele.

2.1.5 *Domain-Driven Design*

DDD (*Domain-Driven Design*, do inglês, Design Orientado a Domínio) é um princípio de desenvolvimento de software que propõe que a implementação e o design de software devem refletir as regras de negócio de um dado problema ou domínio, em vez de serem guiados por tecnologias, linguagens, *frameworks*, etc. Para alcançar esse objetivo, o DDD sugere que a separação entre regras de negócio e tecnologias seja incorporada à arquitetura do software, frequentemente em conjunto com padrões como arquitetura hexagonal, arquitetura limpa ou arquitetura em camadas (como será mostrado na Seção 4.2) [Evans 2004].

2.1.6 HTTP

O HTTP (*HyperText Transfer Protocol*) é um protocolo de comunicação entre cliente e servidor, a nível de aplicação. Ele define uma interface padronizada que permite

a comunicação por meio de trocas de mensagens individuais na web. As características principais do HTTP são ser um protocolo de interface simples, com mensagens de fácil compreensão, semântica extensível e sem estado, ou seja, as requisições feitas em uma mesma conexão são independentes. Os principais métodos HTTP incluem *GET*, utilizado para solicitar dados do servidor; *POST*, utilizado para enviar dados ao servidor; *PUT*, utilizado para atualizar ou criar um recurso no servidor; e *DELETE*, utilizado para remover um recurso do servidor. Esses métodos fornecem uma base robusta para a construção de APIs e a interação entre sistemas na web [Nielsen et al. 1999].

2.1.7 *Websockets*

WebSockets são um protocolo de comunicação bidirecional que permite trocas contínuas de dados entre o cliente e o servidor através de uma única conexão TCP. Diferente do protocolo HTTP, que é baseado em requisições e respostas, WebSockets estabelecem uma conexão persistente, permitindo uma comunicação em tempo real com baixa latência. Isso é particularmente útil em aplicações onde atualizações frequentes e rápidas são necessárias, como em jogos online, plataformas de negociação financeira, sistemas de chat e colaboração em tempo real. WebSockets oferecem a vantagem de reduzir a sobrecarga de comunicação ao eliminar a necessidade de repetidas requisições HTTP, resultando em uma performance melhorada e um uso mais eficiente dos recursos da rede. Este protocolo é definido pela especificação RFC 6455 e é amplamente suportado pelos navegadores modernos, tornando-o uma escolha viável para a implementação de funcionalidades interativas e dinâmicas na web [Fette e Melnikov 2011].

2.1.8 REST

REST (*Representational State Transfer*) é um conjunto de princípios arquiteturais para sistemas operados em rede. Nesse modelo de arquitetura há restrições especificas que compõem as diretrizes da arquitetura REST, sendo estas [Fielding 2000]:

- *Client-Server*: é a principal restrição para aplicação web. Tem o objetivo de separar as tarefas com relação a interface do usuário com as tarefas do armazenamento de dados. Com essa separação permite que os componentes sejam independentes e permitindo uma maior escalabilidade.

- *Stateless*: outra restrição importante é a comunicação ser sem estados, isso é, o servidor não mantém nenhum contexto das solicitações e com isso o cliente deve enviar todas as informações necessárias para o processamento das requisições pelo servidor. Com essa restrição é possível melhorar a confiabilidade, visibilidade e escalabilidade do sistema.

- *Uniform Interface*: ao utilizar uma interface uniforme, obtém-se uma arquitetura mais simplificada e genérica, melhorando a visibilidade das interações.

2.1.9 APIs RESTful

Uma API *RESTful* é uma interface que utiliza as restrições e princípios da arquitetura REST para permitir a comunicação entre sistemas. Elas são baseadas no protocolo HTTP e utilizam seus métodos padrão, como *GET*, *POST*, *PUT* e *DELETE*, para realizar operações sobre recursos representados em formato JSON (*JavaScript Object Notation*) ou XML (*Extensible Markup Language*). Elas seguem princípios herdados dos princípios REST, como *stateless* e *client-server*, a uniformidade de interfaces e a manipulação de recursos através de URLs (*Uniform Resource Locator)*, facilitando a escalabilidade, a modularidade e a fácil integração entre diferentes serviços na web [Richardson e Ruby 2007].

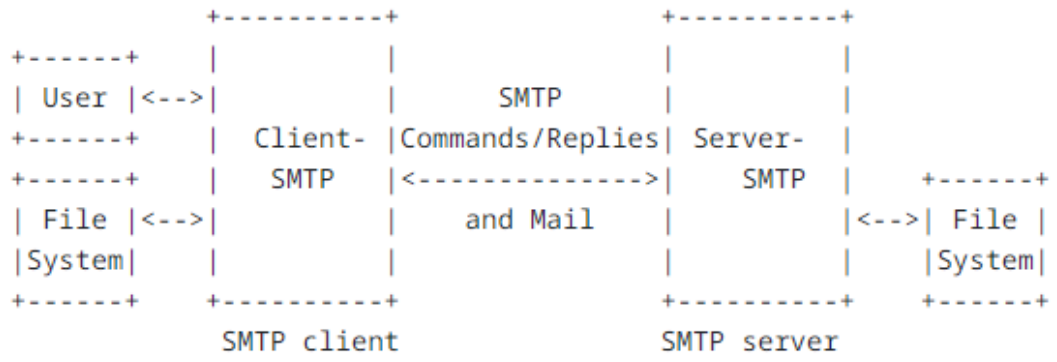
2.1.10 AMQP

O AMQP (*Advanced Message Queuing Protocol*) é um protocolo projetado para trocas de mensagens assíncronas, com um modelo robusto, de padrão aberto, multicanal, escalável, seguro e eficiente, tornando-se uma base sólida para comunicação entre diferentes aplicações e sistemas distribuídos. O AMQP, por ser um protocolo binário, torna-se mais eficiente para aplicações com transmissão de mensagens, quando a taxa de transferência de dados é relevante [Vinoski 2006].

2.1.11 SMTP

O SMTP (*Simple Mail Transfer Protocol*) é um protocolo de comunicação que tem o objetivo de enviar e receber mensagens de e-mail de maneira confiável e eficiente. Este protocolo é amplamente utilizado principalmente pela sua simplicidade, por ser independente da tecnologia e padronizado [Klensin 2008].

Figura 2.1 – Estrutura SMTP.



Fonte: [Klensin 2008]

Na Figura 2.1 pode-se observar a estrutura básica do SMTP. Esse modelo de comunicação tem como atores principais: *SMTP client* (remetente) e *SMTP server* (destinatário). No momento que a conexão é estabelecida, o *SMTP client* envia uma série de comandos específicos e o e-mail em si, incluindo cabeçalhos, conteúdo e outras estruturas. Se o e-mail for enviado para vários destinatários, o SMTP pode enviar somente uma cópia desse e-mail para todos os destinatários usando o mesmo domínio. O *SMTP server* recebe a mensagem e responde aos comandos enviados, tanto em casos de sucesso, onde a mensagem é aceita, ou em casos de erro. Após a transmissão ser concluída a conexão é encerrada [Klensin 2008].

2.1.12 SSL

O SSL (*Secure Sockets Layer*) é um protocolo de segurança que estabelece uma conexão criptografada entre um servidor Web e um navegador Web durante a transferência de dados. Esse protocolo foi desenvolvido para ficar entre a camada TCP e a camada de aplicação, permitindo a essa camada uma interface semelhante ao TCP [Weaver 2006].

O SSL atua durante o estabelecimento da conexão e a transferência de dados. Utilizando algoritmo de chave simétrica para criptografar as mensagem, o cliente irá solicitar

uma conexão com o servidor definido, enviando uma mensagem. Nessa mensagem é exibida a versão do SSL do cliente, as preferências do mesmo e a criptografia usada. O servidor escolhe as preferências do cliente e retorna com essas informações, fornecendo a sua chave pública para verificação de sua autenticidade. De forma opcional, o servidor pode também solicitar a verificação da chave pública do cliente, caso essa opção esteja habilitada. Após cliente e servidor calcularem as chaves, a comunicação é estabelecida e a transmissão de dados será feita de forma segura e criptografada [Chou 2002].

2.1.13 *Containers*

Containers são uma tecnologia de virtualização que permite encapsular uma aplicação e todas as suas dependências em uma única unidade executável. Isso garante que a aplicação rode de maneira consistente em diferentes ambientes, independentemente das diferenças nas configurações do sistema operacional subjacente. Utilizando um mecanismo de isolamento baseado no *kernel* do sistema operacional, os *containers* compartilham o mesmo sistema operacional, mas operam em espaços isolados, o que os torna mais leves e eficientes em comparação com as máquinas virtuais tradicionais. Essa abordagem facilita a portabilidade, a escalabilidade e a gestão de aplicações em ambientes de desenvolvimento, testes e produção, promovendo uma maior eficiência no uso de recursos e uma implementação contínua [Merkel 2014].

2.2 Protocolo Ocpp

O protocolo OCPP, como explicado no Capítulo 1, é responsável pela comunicação entre a estação de carregamento e o CPMS. Cada fabricante de carregadores elétricos é responsável por implementar em seus carregadores a comunicação com o protocolo, podendo haver variações entre as implementações, como regras para o identificador do carregador e o envio de parâmetros opcionais em algumas operações. Além disso, cada carregador pode não oferecer certas operações mais específicas, como definições de perfis de carga.

Na visão, tanto do OCPP quanto do CPMS, cada usuário é identificado através de uma *tag RFID* (*Radio-frequency identification*, do inglês, Identificar de Rádio Frequência). Para desbloquear o uso de uma estação de carregamento o usuário deve inserir sua

tag nela e esperar ser autorizado, ou pela própria estação, ou pelo CPMS. Todas as operações após a autorização do usuário serão consideradas feitas por ele.

Cada estação de carregamento possui também um identificador, usado para se identificar no momento de conexão com o CPMS. Esse identificador não possui um padrão, dependendo então de cada implementação do protocolo pela estação de carregamento, sendo geralmente no formato de texto, exemplo "BR010CRSC00".

O protocolo OCPP possui uma coleção de operações que representam estados da estação de carregamento (disponível, em uso, etc.), eventos (como início de uma carga, autenticação de um usuário, etc.) e comando remotos para a estação (por exemplo um comando de reserva de carga na estação).

Abaixo é listado as operações centrais para o desenvolvimento do trabalho, de acordo com [OCA 2017].

- *Boot Notification*: Notifica que uma estação de carregamento se conectou ao sistema, passando o identificador da estação e outras informações, como modelo, fabricante, etc.

- *Authorize*: Operação contendo os dados de uma autenticação de usuário, tendo principalmente um RFID (*Radio-frequency identification*, do inglês, Identificar de Rádio Frequência) do usuário.

- *Start Transaction*: Notifica que a estação de carregamento começou a carregar um EV, passando o identificador da estação, identificador do carregador da estação, o RFID do usuário e o medidor em Wh (*watt-hour*, do inglês, watt-hora) no instante do início da carga.

- *Stop Transaction*: Notifica que uma carga previamente iniciado foi finalizada, tanto como sucesso ou erro, a razão da finalização e o estado do medidor no momento de finalização, em Wh. As possíveis razões de parada de uma operação de *Stop Transaction* são mostrados na Tabela 2.1.

- *Status Notification*: Notifica quando ocorre uma mudança de estado na estação de carregamento ou em algum carregador específico. Os estados possíveis estão listados na Tabela 2.2.

- *Meter Values*: Manda mensagem contendo os valores atuais de algumas medidas da estação de carregamento, em Wh, e também outras informações opcionais, como, por exemplo, voltagem, temperatura, frequência, etc.

Razão	Significado
EmergencyStop	Botão de parada de emergência foi acionado.
EVDisconnected	Cabo foi desconectado do EV.
HardReset	Um comando de *reset* foi recebido.
Local	Pedido de finalização local do usuário.
PowerLoss	Perda completa de energia elétrica da estação.
Reboot	Estação foi reiniciada localmente.
Remote	Pedido de finalização remoto de um usuário.
SoftReset	Um comando de *reset* leve foi recebido.
UnlockCommand	Comando de liberar carregador recebido.
DeAuthorized	Transação finalizada por um *Start Transaction* de outro usuário.
Other	Outras razões.

Tabela 2.1 – Estados possíveis para uma operação de *Stop Transaction*

Estado	Evento que leva ao novo estado
Available	Carregador fica disponível para o usuário.
Preparing	Carregador fica indisponível porém ainda não começou uma carga.
Charging	Carregador começa uma carga.
SuspendedEVSE	Quando o EVSE (*Electric Vehicle Supply Equipment/System*, do inglês, Rede de Carregamento para Veículos Elétricos) suspende a carga.
SuspendedEV	Quando o EVSE está pronto mas o EV ainda não.
Finishing	Quando uma carga acaba mas o carregador ainda não esta disponível.
Available	Carregador fica disponível para o usuário
Unavailable	Carregador se torna indisponível através de um comando de mudança de disponibilidade.
Faulted	Carregador ou Estação de carga tornam-se inoperante por algum erro.

Tabela 2.2 – Estados possíveis para uma operação de *Status Notification*

2.2.1 Implementação *Open-source* do protocolo OCPP

Existem implementações de código aberto que desempenham o papel de CPMS, sendo o *SteVe* uma das mais populares, com mais de 2300 *commits* no *GitHub*. Desenvolvido em 2013 pela Universidade de Aachen, na Alemanha, o *SteVe* oferece uma interface gráfica e diversas funcionalidades, incluindo:

- Listagem de informações básicas: visualização de estações de carregamento conectadas, RFIDs conhecidos, histórico de cargas, entre outros.
- Regras de autorização de conexão: configuração de permissões, permitindo que apenas estações aprovadas se conectem ao sistema, além de manter um registro de tentativas de conexão negadas.
- Enriquecimento de dados das estações: permite que os usuários adicionem informações complementares às estações de carregamento, como endereço postal, nome e coordenadas geográficas.

No entanto, o *SteVe* apresenta algumas limitações significativas:

- Dificuldade na exportação de dados: o sistema não oferece uma API para consumo externo ou interação, restringindo o acesso aos dados apenas à interface de usuário.
- Compatibilidade limitada: suporta apenas a versão 1.6 do protocolo OCPP, o que impede a conexão com estações que utilizam a versão 2.0.1 do protocolo.

3 PROPOSTA

Este capítulo apresenta a proposta para a implementação do sistema que lida com carregadores de carros elétricos, detalhando a arquitetura, a infraestrutura de suporte e os casos de uso que serão abordados. A proposta é fundamentada em práticas recomendadas de desenvolvimento de software, com o objetivo de criar um sistema robusto, escalável e flexível, permitindo uma fácil manutenção e evolução do mesmo. Cada Seção a seguir fornecerá uma visão abrangente das decisões e estratégias adotadas para alcançar esses objetivos.

3.1 Arquitetura

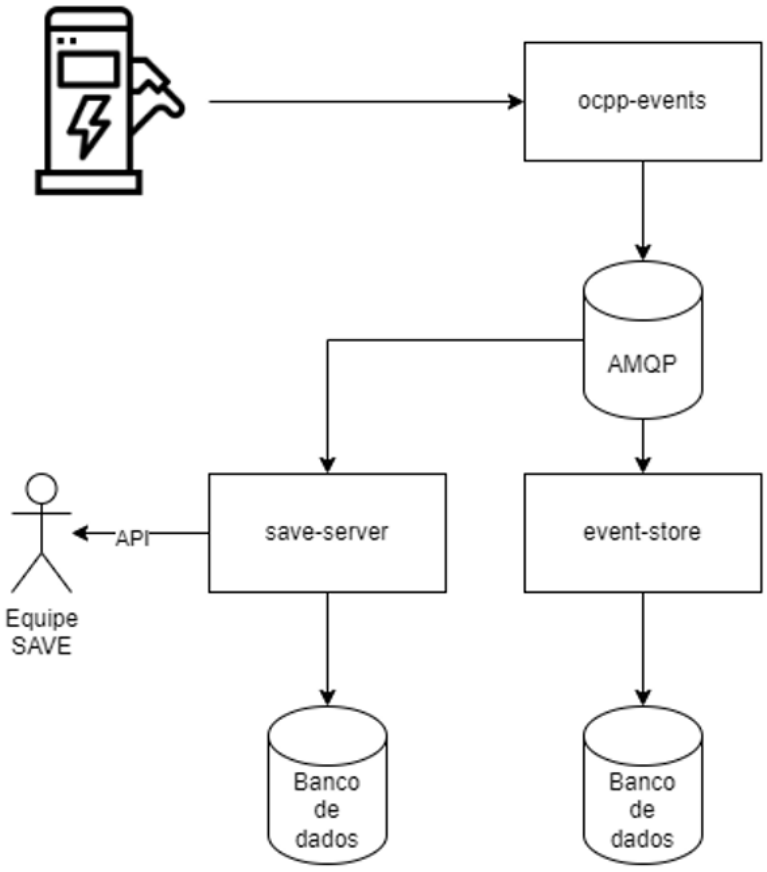
Para a implementação de um sistema que lida com carregadores de carros elétricos, foi escolhida uma arquitetura de microsserviços, representada pela Figura 3.1, onde cada componente funciona de forma independente. Há um serviço dedicado a lidar diretamente com as estações de carregamento e o protocolo OCPP (*ocpp-events*), outro serviço responsável pela implementação das regras de negócio (*save-server*), sem precisar se comunicar diretamente com os carregadores elétricos, oferecendo uma visão mais abstrata dos carregadores, e um último microsserviço responsável por disponibilizar os eventos enviados pelo sistema, para fins de depuração e reenvio (*event-store*). Cada um desses microsserviços será analisado em mais detalhes nas próximas seções.

Também foi avaliado o uso do *SteVe*, que poderia tanto se comunicar com o carregador quanto fornecer informações ao usuário. No entanto, apesar de o *SteVe* oferecer uma interface web gráfica, ele não disponibiliza nenhuma API ou outros tipos de integrações, tornando-se extremamente inflexível para atender às necessidades da *start-up*.

3.1.1 Microsserviço *ocpp-events*

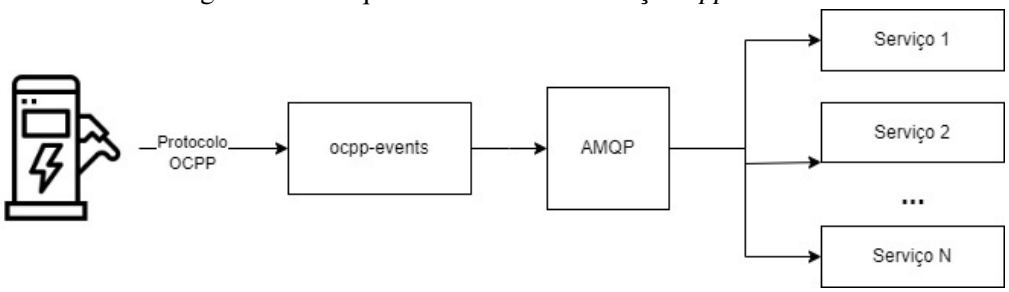
Baseado na arquitetura escolhida, é proposto o microsserviço *ocpp-events*, responsável por lidar com a comunicação do sistema com o carregador, utilizando o protocolo OCPP. Dessa forma, é possível desacoplar os detalhes de implementação do protocolo das regras de negócio. Cada comunicação com a estação de carregamento gera um evento que é enviado para um *Message Broker* AMQP, responsável por distribuir esse evento

Figura 3.1 – Arquitetura do sistema



Fonte: O Autor

Figura 3.2 – Arquitetura do microsserviço *ocpp-events*



Fonte: O Autor

para quaisquer outros serviços, conforme representado na Figura 3.2.

Uma vantagem dessa implementação é possibilitar que esse serviço seja leve e tenha poucas responsabilidades, já que ele deve manter a conexão via *websockets* com a estação de carregamento durante todo o tempo em que a estação estiver ligada, tirando essa responsabilidade de qualquer outro serviço que precise dos dados da estação. Além disso, é possível abstrair o protocolo OCPP dos outros serviços, permitindo que eles se concentrem principalmente nas regras de negócio ou nos serviços de suporte ao sistema.

Cada nova conexão de um carregador feita no *ocpp-events* é armazenada em seu repositório local até que se desconecte ou permaneça inativa por um longo período. Cada troca de mensagem realizada com o carregador através do protocolo OCPP gera um evento que é enviado para outros serviços via protocolo AMQP, possibilitando o uso do padrão *publisher-subscriber*, garantindo, assim, um maior desacoplamento.

Os eventos enviados são praticamente uma cópia da mensagem do protocolo *OCPP*, mas com alguns atributos adicionais, como a versão do protocolo, identificador do carregador, data da mensagem, entre outros.

Cada serviço que escuta os eventos provenientes do *ocpp-events* não deve se preocupar com os detalhes de implementação do *ocpp-events*, tampouco deve depender de maneira síncrona dele.

3.1.2 Microsserviço *save-server*

Para a implementação dos casos uso propostos pela SAVE foi criado o microsserviço *save-sever*, responsável por escutar os eventos do *ocpp-events*, guardando e gerenciando os dados necessários para os casos de uso. Sua principal responsabilidade é disponibilizar os dados das estações e seus usos de uma maneira mais próxima dos casos de uso e mais distante do protocolo *OCPP*.

Alguns dos dados gerenciados por esse serviço são:

- Carga: obtido através dos eventos de *StartTransaction* e *StopTransaction* emitidos pelo *ocpp-events*. Contém dados como de total de energia carregadas e datas de início e fim da carga. Também salva se houve erro na carga.
- Estado da estação de carga: obtido através de diversos eventos, como *StatusNotification*, *MeterValues*, *StartTransaction*, *StopTransaction*, etc. Tem dados de disponibilidade do carregador, última carga realizada, informações técnicas de energia, etc.

Um ponto importante na implementação desse serviço é evitar que detalhes do protocolo OCPP vazem para as camadas de regras de negócio. Por exemplo, caso no futuro o OCPP deixe de ser o padrão ou o mais popular, seria possível integrar qualquer outro serviço ao sistema, apenas modificando como os dados são transformados ao entrarem na plataforma. Em outras palavras, se um protocolo hipotético fornecer os mesmos dados, mas com objetos completamente diferentes, não devemos nos preocupar com a implementação dos objetos de regras de negócio, apenas com a camada que recebe e trata esses dados, transformando-os em objetos internos.

Para a implementação interna do serviço deve-se deixar quaisquer dependências externas isoladas, através de interfaces. Por exemplo é possível ter uma interface de repositório para cada objeto que será salvo, tendo sua implementação feita nas camadas externas utilizando alguma tecnologia específica. Em outras palavras é possível ter na camada de domínio uma interface *ChargingStationRepository* (do inglês, Repositório de estações de carga) e na camada de infraestrutura uma implementação dessa interface utilizando qualquer tecnologia, como, por exemplo, *PsqlChargingStationRepository* (do inglês, Repositório de estações de carga PSQL), utilizando tecnologia *PostgreSQL*. Ainda no exemplo, nas camadas de domínio e serviço é feita apenas referência à interface *ChargingStationRepository*, podendo então, num futuro, trocar a implementação dela por qualquer outra sem precisar se preocupar com o código das camadas internas.

3.1.3 Microsserviço *event-store*

Para facilitar a depuração de eventos no sistema e possibilitar o reenvio de uma coleção de eventos, foi criado o microsserviço *event-store*, responsável por guardar em seu banco de dados uma cópia de cada evento enviado pelo microsserviço *ocpp-events*.

Esse serviço possui uma *API* que retorna eventos armazenados com base em alguns critérios, como data, tipo e estação emissora. Além disso, existe a funcionalidade de reenviar um grupo de eventos para outros serviços através de um comando na *API*, utilizando parâmetros similares aos de consulta.

3.2 Infraestrutura de suporte

A implementação de um sistema robusto e eficiente para lidar com carregadores de carros elétricos requer uma infraestrutura de suporte sólida. Esta seção descreve os requisitos técnicos dos componentes e tecnologias essenciais que sustentam a solução proposta, garantindo sua operação contínua, escalabilidade e segurança. Abordaremos as ferramentas utilizadas para a gestão de microsserviços, armazenamento de dados, comunicação entre componentes e monitoramento do sistema, bem como as práticas adotadas para assegurar a resiliência e eficiência da solução.

3.2.1 *Message Broker*

O *Message Broker* é a infraestrutura mais importante para o sistema proposto, sendo responsável pela gestão das mensagens enviadas e consumidas pelos microsserviços. Requisitos:

- permitir comunicação assíncrona entre os componentes, oferecendo um sistema de filas que são consumidas e alimentadas pelos componentes;
- fornecer um mecanismo de *publisher-subscriber*, onde um tipo de evento específico pode ser multiplexado para ser consumido por uma quantidade arbitrária de serviços;
- permitir que serviços se inscrevam para receber algum tipo de evento específico;
- ter um mecanismo de armazenamento de mensagem nas filas, para que um serviço possa consumi-las quando conseguir, podendo acumular uma certa quantidade de eventos nela;
- as filas devem ser consumidas de modo FIFO (*First-in First-out*, do inglês, primeiro a entrar, primeiro a sair);

3.2.2 Banco de dados

A infraestrutura de banco de dados desempenha um papel fundamental na sustentação da solução proposta para o sistema de carregadores de carros elétricos. É essencial que esta infraestrutura seja capaz de armazenar, de maneira segura e durável, uma ampla variedade de dados, garantindo a facilidade de consulta e a otimização do desempenho das operações. São necessários dois bancos de dados distintos, um para o serviço *save-server* e outro para o *event-store*, onde cada um possui uma necessidade um pouco diferente:

- Microsserviço *save-server*: como tratará de dados de regras de negócio é desejável uma solução que lide bem com dados bem estruturados, fornecendo uma sólida abordagem relacional.
- Microsserviço *event-store*: é responsável por receber dados e salva-los sem precisar conhecer a estrutura deles, não sendo então necessária uma abordagem relacional. É importante uma solução que consiga lidar com uma quantidade de dados maior que o serviço *save-server*, sendo que armazenará todos eventos emitidos pelo serviço *ocpp-events* (e num momento futuro por quaisquer outros microsserviços que venham a emitir eventos).

3.2.3 *Container*

Uma questão essencial para esse sistema é a mantenabilidade, e para isso deve-se preocupar com a infraestrutura em que os serviços serão executados, e para isso definiu-se o uso de *containers*. Os requisitos técnicos esperados para essa infraestrutura são:

- Consistência: os *containers* deverão ser de fácil reprodução e compatíveis com qualquer sistema operacional da máquina hospedeira, sem que seja necessário quaisquer alterações na aplicação.
- Portabilidade: Todas as dependências externas da aplicação, como bibliotecas, configurações, etc. deve estar contido dentro do *container*.
- Isolamento: *Containers* oferecem isolamento leve entre aplicações e seus ambientes. Isso significa que você pode executar várias aplicações em um único hospedeiro sem que elas interfiram umas com as outras, proporcionando maior segurança e estabilidade.

- Configuração: Deve ser possível configurar a aplicação, tanto por variáveis de ambiente quanto por arquivos de configuração, possibilitando, por exemplo, a mesmo *container* ser utilizado no ambiente de produção, desenvolvimento e local apenas mudando tais configurações.

3.3 Proposta de casos de uso

Para a proposição dos casos de uso, foram realizadas reuniões síncronas com a equipe da SAVE. Nessas reuniões, discutiram-se aspectos técnicos do protocolo OCPP (como vistos no Capítulo 1 e na Seção 2.2) e funcionalidades importantes para a *start-up*. Com base nisso priorizou-se os casos de uso apresentados nessa Seção.

3.3.1 Listagem detalhada das estações de carregamento

Os usuários do sistema, nesse primeiro momento sendo a equipe da SAVE, gostariam de ter uma listagem das estações de cargas do sistema, tanto as conectadas ou não, trazendo informações importantes para a operação. Essas informações são tanto informações do próprio protocolo OCPP, como estado do carregador (disponível, em uso, desconectado, terminando carga, em erro, etc.), última carga feita, último usuário autorizado, dados de energia, etc. como também informações inseridas manualmente, através da plataforma, pela equipe da SAVE, como latitude, longitude, nome amigável para a estação, endereço, data inicial de conexão, entre outros.

Para termos esses dados disponíveis o microsserviço *save-server* precisa escutar vários tipos de eventos diferentes emitidos pelos serviço *ocpp-events* e organiza-los de forma lógica em seu banco de dados, formando uma visão final e desacoplada do protocolo OCPP. É preciso também que essa gestão de dados seja flexível, de forma que adição ou remoção de propriedades da estação de carga sejam possíveis de maneira simples, e que existam testes automatizados para garantir a integridade e montagem lógica de tais dados.

3.3.2 Relatório de cargas por e-mail

Os usuários do sistema, a equipe da SAVE, desejam ter relatórios disponíveis sobre o uso do sistema para tomar decisões de negócios mais informadas e monitorar seus recursos.

Para esse caso de uso, é necessário primeiro ter o histórico de cargas salvo no microsserviço *save-server*, de forma desacoplada tanto do microsserviço *ocpp-events* quanto do protocolo OCPP. Como o protocolo não fornece diretamente esses dados da carga, é preciso inferi-los a partir dos eventos de início e fim de carga, consolidando uma visão completa da carga.

Os dados de carga devem incluir informações como data de início e término da carga, quantidade de energia utilizada, estação de carregamento utilizada e o usuário responsável pela carga.

Para o envio de e-mails é preciso ter uma configuração de frequência desses e-mail e também filtros para quais tipos de cargas serão enviados. Esses filtros podem ser, por exemplo data de início de carga, estação, usuário, etc. Para fins de implementação base é utilizado o envio de e-mails numa frequência de uma vez por semana, contendo todos as cargas que ocorreram nessa semana, independente da estação ou do usuário.

Os dados disponibilizados no caso de uso serão em formato CSV e anexados ao e-mail. Cada linha deve conter informações de data de início e fim de carga, total utilizado, usuário que fez a carga, estação que fez a carga e qual estado final da carga (sucesso, erro de desconexão do servidor, erro de desconexão do carro elétrico, etc.)

3.4 Cenário de falha

Como cenário de falha o sistema será capaz de lidar com cargas que não acabem em sucesso, seja por falha na conexão com a estação de carga, falha da conexão da estação de carga com o sistema (por falta de energia elétrica na estação de carga, por exemplo), etc.

Para isso o microsserviço *ocpp-events* enviará um evento de *disconnect* (do inglês, desconexão) para os outros serviços. Esse evento não existe no protocolo OCPP e será emitido toda vez que a conexão *websocket* com uma estação de carga acabar.

Também será preciso levar em consideração os eventos de *stop-transaction* e *status-notification*, pois eles são responsáveis por informar ao *CPMS* quando há algum

erro. Nos casos de desconexão eles são enviados assim que a estação de carga se conecta novamente ao sistema.

4 DESENVOLVIMENTO

Neste capítulo, serão apresentados os resultados experimentais obtidos a partir da implementação dos casos de uso e do sistema proposto. Será descrito em detalhe como cada caso de uso foi desenvolvido, os desafios enfrentados durante a implementação e as soluções adotadas para superá-los. Além disso, será explorada a interação entre os componentes do sistema e como eles colaboram para atingir os objetivos definidos. Esta análise fornecerá uma visão abrangente sobre a funcionalidade e a robustez da solução implementada.

4.1 Tecnologias utilizadas

Nesta Seção, serão discutidas as diferentes tecnologias escolhidas para o desenvolvimento do sistema. Abordaremos desde a linguagem de programação e os *frameworks* selecionados até as tecnologias empregadas para a infraestrutura de suporte. Mostraremos como essas escolhas atendem aos pré-requisitos estabelecidos no Capítulo 3, garantindo que o sistema seja robusto, escalável e eficiente. A seguir, detalharemos cada uma dessas tecnologias e suas justificativas, destacando a importância de cada componente na construção de uma infraestrutura sólida e confiável para o nosso sistema.

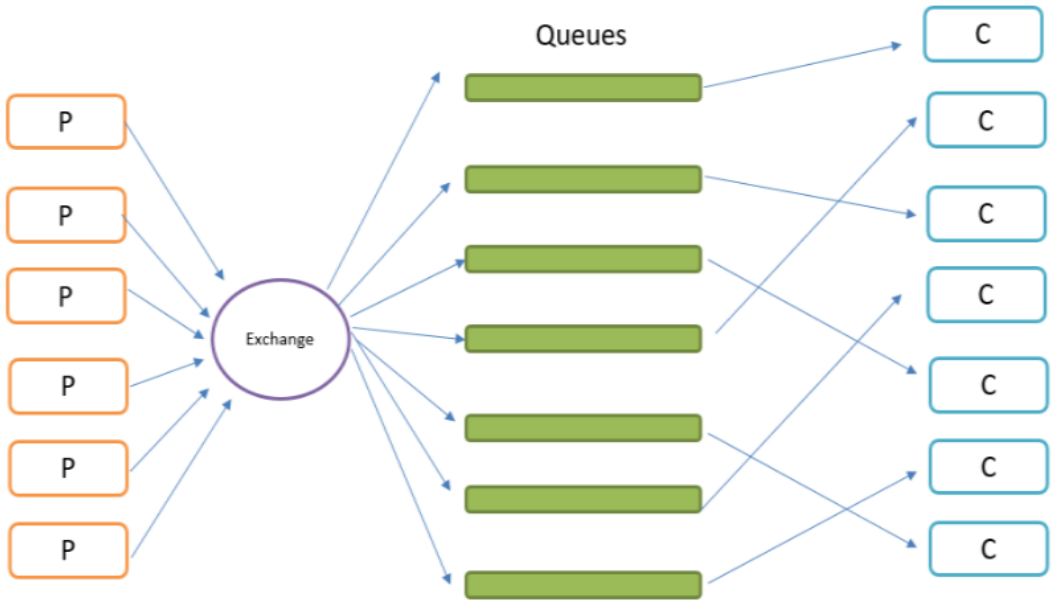
4.1.1 *RabbitMq*

Para o *Message Broker*, foi escolhido o *RabbitMQ* [Pathak e Kalaiarasan 2021], que é uma implementação popular e de código aberto do protocolo AMQP, gerenciando o fluxo de mensagens entre diferentes aplicações, sistemas e serviços. Em um cenário com vários produtores e consumidores, cada produtor publica mensagens em um *exchange*, que é responsável por implementar o padrão *publisher-subscriber*, distribuindo essas mensagens para seus consumidores. Os consumidores, por sua vez, consomem as mensagens das filas seguindo o padrão FIFO.

Sendo consumidores chamados de "C"e produtores de "P", pode-se ver na Figura 4.1 como arquitetura do *RabbitMQ* se comporta.

Para o trabalho, foi criada uma *exchange* chamada *ocpp-events*, possibilitando o uso do padrão *publisher-subscriber*. Por exemplo, o microsserviço *ocpp-events* publica

Figura 4.1 – Arquitetura RabbitMQ.



Fonte: [Pathak e Kalaiarasan 2021]

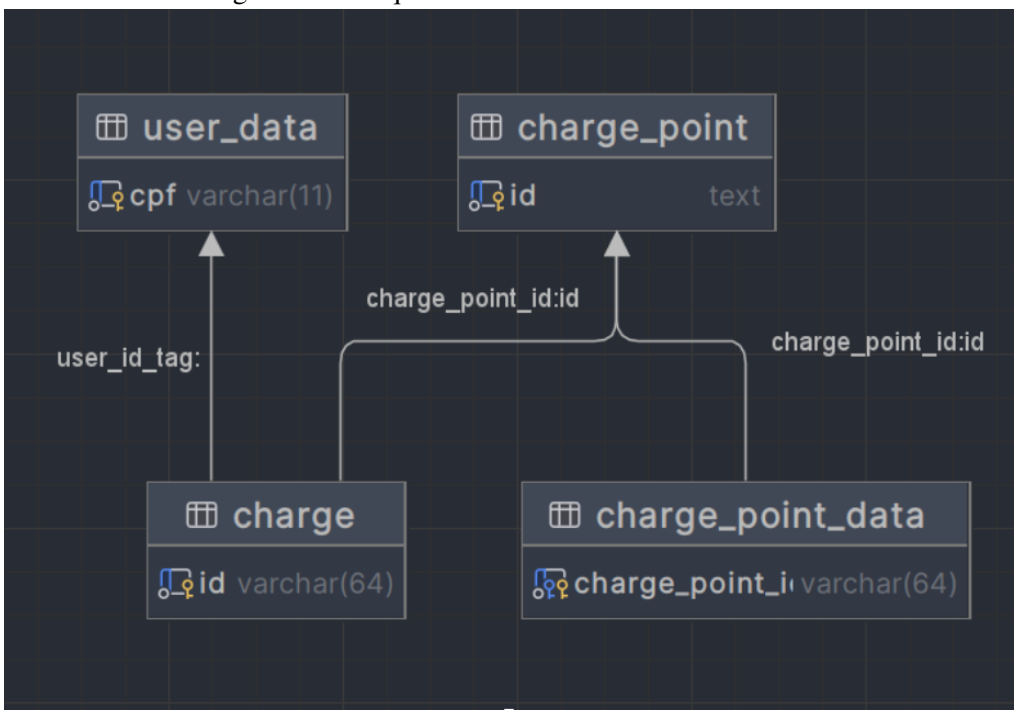
uma mensagem de *ocpp.heartbeat* na *exchange ocpp*, e todos os serviços que escutam a fila *ocpp.heartbeat* recebem uma cópia dessa mensagem, podendo tratá-la conforme necessário.

As mensagens permanecem na fila até serem consumidas pelo serviço que a recebeu, e as filas foram configuradas no modo durável, isto é, elas são persistidas em disco, não sendo perdidas caso o *RabbitMQ* seja reiniciado [Pathak e Kalaiarasan 2021].

4.1.2 *PostgreSql*

Para o banco de dados do microsserviço *save-server*, foi escolhido o *PostgreSQL*, um banco de dados relacional que atende aos requisitos deste microsserviço. A abordagem relacional permite uma melhor estruturação e organização dos dados, e o uso do *PostgreSQL* também fornece mecanismos de atomicidade, ao contrário de soluções não relacionais. O uso de chaves estrangeiras também ajuda a manter os dados consistentes [Stonebraker, Weisberg e Zdonik 2018]. A Figura 4.2 mostra o esquema relacional utilizado, mostrando o uso de chave estrangeiras e modelagem do banco.

Figura 4.2 – Esquema relacional do banco de dados



Fonte: O Autor

4.1.3 *MongoDB*

Para o banco de dados do microsserviço *event-store* foi escolhida uma abordagem não relacional, por se tratar de dados não estruturados. A tecnologia escolhida foi o *MongoDB*, solução escalável e popular para base de dados *NoSQL*. Todos eventos são persistidos em uma única tabela e é possível filtrar e agrupar por sub-campos do documento [Kumar e Patel 2014]. A Figura 4.3 mostra a tabela *events*, que contém a data do evento, o tipo do evento e os dados de um evento.

4.1.4 *Docker*

Todos os componentes do sistema foram transformados em *containers Docker*, possibilitando seu funcionamento em praticamente qualquer sistema operacional [Merkel 2014]. Temos uma imagem *Docker* para os microsserviço *ocpp-events*, *save-server* e *event-store*, além de utilizarmos as imagens oficiais do *PostgreSQL*, *RabbitMQ* e *MongoDB*. Para facilitar a comunicação entre os *containers* e a gestão das variáveis de ambiente, é utilizado o *docker-compose*.

As três imagens *Docker* das aplicações implementadas utilizam a mesma imagem

Figura 4.3 – Tabela do *events* do banco MongoDB

{} date	{} type	{} payload
1721279604	ocpp-events.boot-notific…	{"charge_point_base": {"ocpp_version": "ocp
1721279617	ocpp-events.disconnect	{"charge_point_base": {"ocpp_version": "ocp
1721279617	ocpp-events.boot-notific…	{"charge_point_base": {"ocpp_version": "ocp
1721279619	ocpp-events.authorize	{"charge_point_base": {"ocpp_version": "ocp
1721279619	ocpp-events.start-transa…	{"charge_point_base": {"ocpp_version": "ocp
1721279620	ocpp-events.stop-transac…	{"charge_point_base": {"ocpp_version": "ocp
1721279620	ocpp-events.heart-beat	{"charge_point_base": {"ocpp_version": "ocp
1721279621	ocpp-events.meter-values	{"charge_point_base": {"ocpp_version": "ocp
1721279622	ocpp-events.status-notif…	{"charge_point_base": {"ocpp_version": "ocp
1721279622	ocpp-events.meter-values	{"charge_point_base": {"ocpp_version": "ocp
1721279623	ocpp-events.meter-values	{"charge_point_base": {"ocpp_version": "ocp
1721279624	ocpp-events.meter-values	{"charge_point_base": {"ocpp_version": "ocp
1721279625	ocpp-events.meter-values	{"charge_point_base": {"ocpp_version": "ocp
1721279627	ocpp-events.meter-values	{"charge_point_base": {"ocpp_version": "ocp
1721279628	ocpp-events.meter-values	{"charge_point_base": {"ocpp_version": "ocp

Fonte: O Autor

base do *Python* versão 3.11, sendo ela a *python:3.11.8-slim*.

O *Dockerfile* do microsserviço *save-server* pode ser visto na Figura 4.4, e contém as seguintes ações de configuração de ambiente:

1. Declarada a imagem base *python:3.11.8-slim*

2. Cópia dos arquivos do diretório atual para o diretório */app* na imagem.

3. Configurado o diretório */app* como diretório padrão da imagem

4. São instaladas, utilizando a ferramenta *Pip* do *Python* as dependências do projeto, sendo elas bibliotecas *Python* como *ocpp*, *requests*, *FastAPI*, etc. Cada microsserviço tem suas dependências no arquivo *requirements.txt*, tido como padrão da linguagem.

5. A porta 8080 da imagem é configurada como exposta.

6. É configurado o comando padrão que será executado quando essa imagem for utilizada.

Figura 4.4 – Imagem *Docker* do microsserviço *save-server*

```
FROM python:3.11.8-slim
COPY . /app
WORKDIR /app
RUN pip install . && pip install "uvicorn[standard]"
EXPOSE 8080
CMD ["uvicorn", "save_server.__main__:app", "--host", "0.0.0.0", "--port", "8080"]
```

Fonte: O Autor

Figura 4.5 – Rota de API de cargas para o microsserviço *save-server*

```
router = APIRouter(prefix=f"/charges")

@router.get("/")
def index(
    charging_station_id: str = None,
    start_date: datetime = None,
    end_date: datetime = None
):
    filter_query = ChargeFilter(
        charge_point_id=charging_station_id,
        start_date=start_date,
        end_date=end_date
    )
    return charge_service.find_by(filter_query)
```

Fonte: O Autor

4.1.5 Linguagem de programação *Python*

A linguagem de programação escolhida para o desenvolvimento do sistema foi *Python*, devido à sua popularidade na implementação de sistemas *backend* e à ampla disponibilidade de bibliotecas que auxiliam no desenvolvimento. Além disso, *Python* oferece uma sintaxe simples e de fácil entendimento, o que contribui para a produtividade dos desenvolvedores. Todos os microsserviços deste projeto foram implementados utilizando a mesma linguagem, facilitando a troca de contexto entre os desenvolvedores quando trabalham em diferentes partes do sistema.

4.1.6 *FastAPI*

Para a implementação das APIs das aplicações foi utilizado o *framework Python FastAPI*, que tem uma abordagem simples e leve para declaração das rotas.

Um exemplo do uso da *FastAPI* pode ser visto na Figura 4.5, onde é declarada

uma rota *GET /charges/*, e são esperados três parâmetros, todos eles opcionais:

- *charging station id* (em inglês, identificador da estação de carga): identificador de uma dada estação de carga, no formato de texto.
- *start date* (em inglês, data de início): data que filtrará todas as cargas com data de início igual ou posterior a esta.
- *end date* (em inglês, data de térmico): análogo a data de início, filtrando por todas as cargas que têm um fim até esta data.

Dentro da função implementada os parâmetro informados são usados para popular um objeto de filtro e passados para um método do serviço *ChargeService* (em inglês Serviço de Carga) que retorna uma coleção de objetos. O *framework* transforma automaticamente os objetos *Python* em JSON, deixando a resposta da API pronta.

4.1.7 *Pika*

Para a comunicação do código com o *message broker*, utilizamos a biblioteca *Pika*, da linguagem *Python*. Esta biblioteca oferece uma boa abstração para o sistema de filas e *exchangers* do *RabbitMQ*. O uso da *Pika* foi restrito à camada de infraestrutura, mantendo essa tecnologia desacoplada das regras de negócio.

4.1.8 Biblioteca *Python ocpp*

Para a integração do código com o protocolo OCPP foi utilizada a biblioteca *ocpp*, uma dependência externa da linguagem *Python*. Ela fornece uma abstração para as conexões via *websockets* e da troca bidirecional de mensagens com as estações de carga. O uso dessa biblioteca fica restrito ao microsserviço *ocpp-events* e o mais desacoplado possível das regras de negócio do serviço, como será visto na Seção 4.3.

4.2 Arquitetura dos microsserviços

Para a arquitetura interna dos microsserviços, foi escolhido o uso de uma arquitetura de camadas, como proposto pelos princípios de DDD, vistos na Seção 2.1.5. Essas

camadas são: domínio, serviço e infraestrutura, cada uma com suas responsabilidades e abstrações, conforme proposto por [Vernon 2013]. Nas próximas subseções serão mostradas em maiores detalhes as responsabilidade e restrições de cada camada.

4.2.1 Camada de domínio

A camada de domínio é responsável pelas entidades e seus comportamentos, com o objetivo de implementar as regras de negócio de um dado domínio. Nesta camada, evitamos incluir quaisquer dependências externas, tanto de bibliotecas da linguagem *Python* quanto de tecnologias de infraestrutura, utilizando apenas a linguagem de programação pura. Um exemplo disso é a classe *Charge*, como parcialmente mostrada na Figura 4.6. Também não se deve incluir nenhuma classe ou função das outras camadas. Nesta classe, existem os campos utilizados por uma carga e dois métodos que exemplificam o comportamento de finalização. A partir de uma instância da classe *StopTransaction* (do inglês, finalização de transação), atualizamos os campos de uma carga existente, seguindo as regras de negócio estabelecidas. Isso inclui calcular o total de carga a partir da medição inicial (campo *start meter value*) e da medição final (campo *meter values* da classe *StopTransaction*), além de atualizar o status da carga de acordo com as lógicas predefinidas.

Para a implementação das regras de negócio, é crucial considerar a infraestrutura de suporte, como o banco de dados. No entanto, como mencionado anteriormente, é importante evitar dependências externas de bibliotecas e tecnologias específicas. Para contornar essa questão, optamos por declarar interfaces para essas infraestruturas. Um exemplo é a classe *ChargeRepository*, ilustrada na Figura 4.7, que define apenas a interface com os métodos a serem implementados. A implementação detalhada dessa classe será abordada na Seção 4.2.3.

Essa é a camada com mais classes do sistema, tendo ao total trinta e cinco classes. Há classes que representam as entidades de domínio, como por exemplo *ChargePoint*, *Charge*, *StartTransaction*, *StopTransaction*, *ChargeHistoryReport*, *File*, *Email*, etc. Há também interfaces que fazem parte das regras de negócio mas dependem de tecnologias externas. Exemplos dessas interfaces são os banco de dados, como *ChargePointRepository*, *ChargeRepository*, *MeterValuesRepository*, e interfaces para envio de e-mail e armazenamento de arquivos, nomeadas *EmailSender* e *FileStorage*, reepectivamente.

Figura 4.6 – Classe *Charge* do microsserviço *save-server*

```python
@dataclass
class Charge:
    charge_id: ChargeId
    charging_point_id: str
    transaction_id: int
    status: ChargeStatus
    charging_start_time: datetime
    start_meter_value: float
    charging_end_time: datetime or None
    charging_amount: float or None
    elapsed_charging_time_seconds: float or None
    stop_status: str or None

    def finish(self, stop_transaction: StopTransaction):
        self.charging_amount = stop_transaction.meter_stop - self.start_meter_value
        self.charging_end_time = stop_transaction.timestamp
        self.elapsed_charging_time_seconds = int(
            (self.charging_end_time - self.charging_start_time).total_seconds()
        )
        self._update_status(stop_transaction)

    def _update_status(self, stop_transaction: StopTransaction):
        if stop_transaction.reason is None:
            self.status = ChargeStatus.FINISHED
            self.stop_status = StopTransactionReason.unknown.value
            return
        if stop_transaction.reason == StopTransactionReason.local:
            self.status = ChargeStatus.FINISHED
        else:
            self.status = ChargeStatus.ERROR
        self.stop_status = stop_transaction.reason.value
```

Fonte: O Autor

Figura 4.7 – Classe *ChargeRepository* do microsserviço *save-server*

```python
class ChargeRepository(ABC):
    @abstractmethod
    def find_by(self, query_filter: ChargeFilter) -> List[Charge]:
        pass

    @abstractmethod
    def add(self, charge: Charge) -> Charge:
        pass

    @abstractmethod
    def update(self, charge: Charge) -> Charge:
        pass

    @abstractmethod
    def find_by_id(self, charge_id: ChargeId) -> Charge:
        pass
```

Fonte: O Autor

Figura 4.8 – Classe *ChargeService* do microsserviço *save-server*

```
class ChargeService:
    def __init__(self, charge_history_repository: ChargeRepository):
        self.charge_history_repository = charge_history_repository

    def process_stop_transaction(self, stop_transaction: StopTransaction):
        charges = self.charge_history_repository.find_by(
            ChargeFilter.from_stop_transaction(stop_transaction)
        )
        charge.finish(stop_transaction)
        self.charge_history_repository.update(charge)
```

Fonte: o autor

4.2.2 Camada de serviço

Esta camada é responsável por implementar detalhes da aplicação que são essenciais porém não fazem parte das regras de negócio, como controle de transação e autorização. Nesta camada também orquestramos os componentes da camada de domínio para implementar os casos de uso. Na camada de serviço só deve-se ter acesso às classes dela mesma e da camada de domínio, evitando acessar classes da camada de infraestrutura.

Um exemplo de implementação dessa camada é a classe *ChargeService*, como visto parcialmente na Figura 4.8. Essa classe recebe um repositório de carga, visto na Seção 4.2.1, por inversão de dependência, não dependendo de detalhes de sua implementação. Temos então o método *process_stop_transaction* que recebe uma instância da classe *StopTransaction*. A partir dessa classe busca-se no repositório de cargas pela correspondente a esse fim de transação, chamando o método *finish* da carga, visto na Seção 4.2.1, onde as regras de negócio são implementadas. Por fim é atualizada essa carga em seu repositório.

Pela natureza dessa camada, de ser apenas uma ponte entre as camadas de infraestrutura e domínio, ela acaba sendo a menor camada, com menos classes, apenas dez classes atualmente. Algumas delas são: *ChargePointService*, *ChargeReportService*, *EmailSenderService*, *StartTransactionService*, etc.

4.2.3 Camada de infraestrutura

Por fim, a camada de infraestrutura é a terceira e última camada utilizada, sendo responsável pela comunicação do sistema com o mundo externo, tendo aqui as imple-

Figura 4.9 – Classe *SqlChargeRepository* do microsserviço *save-server*

```python
class SqlChargeRepository(ChargeRepository):

    def add(self, charge: Charge) -> Charge:
        with session_scope() as session:
            sql_charge = SqlCharge.from_domain(charge)
            session.add(sql_charge)
            return charge

    def find_by_id(self, charge_id: ChargeId) -> Charge:
        with session_scope() as session:
            return self._find_by_id(charge_id, session).to_domain()
```

Fonte: O Autor

mentações de APIs e a comunicação com as filas do *Message Broker*. Nessa camada é também onde ficam as implementações de interfaces declaradas nas camadas anteriores, como interfaces de repositórios, e-mail, persistência em disco, etc.

Um exemplo é a implementação da classe *ChargeRepository* vista na Seção 4.2.1, onde foi utilizada a biblioteca *sql-alchemy* da linguagem de programação *Python*, fornecendo uma implementação da técnica ORM (*Object Relational Mapping* (do inglês, mapeamento relacional de objeto)). Na Figura 4.9 podemos ver a classe *SqlChargeRepository* e dois metódos abstratos implementados, o *add* e o *find_by_id*.

Nessa camada encontram-se implementações das interfaces das outras camadas, como *SqlChargePointRepository*, *SqlChargeRepository*, *SqlMeterValuesRepository*, *LocalFileStorage*, *SmtpEmailSender*, entre outras. Além disso, há classes para lidar com a comunicação com o mundo externo. Exemplos dessas classes para APIs incluem *ChargePointController* e *ChargeController*, enquanto para a comunicação com o *Message Broker* temos *StartTransactionHandler*, *BootNotificationHandler* e *MeterValuesHandler*. Ao todo, há vinte e seis classes nesta camada.

4.3 Integração com protocolo OCPP

A integração com o protocolo OCPP é um ponto central do sistema e do trabalho apresentado. Essa integração, conforme exposto na Seção 3.1.1, é responsabilidade do microsserviço *ocpp-events* e é restrita a ele. Para alcançar esse objetivo, utilizou-se a biblioteca *ocpp* da linguagem *Python*, tomando cuidado para evitar o acoplamento excessivo a essa dependência externa. O uso da biblioteca foi restrito à camada de infraestrutura, conforme discutido na Seção 4.2.3.

Figura 4.10 – Classe *ChargePointOcppV16* no microsserviço *ocpp-events*

```
class ChargePointOcppV16(ChargePointFromLib):
    def __init__(self,
        charge_station_id,
        connection,
        ocpp_version: OcppVersion,
        notifier: Notifier
    ):
        super().__init__(charge_station_id, connection)
        self.notifier = notifier
        self.ocpp_version = ocpp_version
        self.charge_point_base = ChargePointBase(
            self.ocpp_version, ChargePointId(charge_station_id)
        )
        self.start_transaction_counter = 0
```

Fonte: o autor

A biblioteca fornece, para cada operação do protocolo, um método correspondente com os respectivos dados mapeados para um método de uma classe abstrata na linguagem *Python*. É possível, então, estender essa classe com os comportamentos desejados pelo componente. A classe *ChargePointOcppV16* estende a classe *ChargePointFromLib*, que é uma dependência externa, conforme mostrado na Figura 4.13. Essa classe possui como atributos o identificador OCPP da estação, versão do protocolo e um objeto do tipo *Notifier*. Para cada nova conexão com uma estação de carga, uma instância dessa classe é criada e mantida em um banco de dados em memória de uma instância do microsserviço, permanecendo lá até a desconexão da estação.

Para cada operação do protocolo OCPP existe então:

- uma classe de domínio correspondente à operação como por exemplo a classe *StopTransactionEvent*, demonstrada na Figura 4.11, contendo atributos mapeados da operação correspondente do protocolo. Essa classe é o evento que é transmitido para o resto do sistema.

- um método na classe *ChargePointOcppV16* mapeando uma operação com seus atributos, como por exemplo o método *on stop transaction* visto na Figura 4.12. Esse método tem como parâmetros uma correspondência um para um para os da operação correspondente. Dado esses parâmetros é montado então um objeto de domínio, sendo aqui como exemplo a classe *StopTransactionRequest*. Esse objeto é passado para a instância da classe *Notifier* e por fim é mapeada uma resposta para a estação de carga, através da classe *StopTransactionPayload*, esta sendo da biblioteca *ocpp*.

Figura 4.11 – Classe *StopTransactionEvent* no microsserviço *ocpp-events*

```python
@dataclass
class StopTransactionEvent:
    charge_point_base: ChargePointBase
    meter_stop: int
    timestamp: str
    transaction_id: int
    reason: str
    id_tag: Union[str, None] = None
```

Fonte: o autor

Figura 4.12 – Método *on stop transaction* da classe *ChargePointOcppV16*

```python
@on(Action.StopTransaction)
def on_stop_transaction(
    self, meter_stop: int, timestamp: str,
    transaction_id: int,  reason: Optional[Reason] = None,
    id_tag: Optional[str] = None, transaction_data: Optional[List[dict]] = None
):
    stop_transaction_event = StopTransactionEvent(
        self.charge_point_base, meter_stop, timestamp,
        transaction_id, reason, id_tag, transaction_data
    )
    self.notifier.notify(stop_transaction_event)
    return call_result.StopTransactionPayload(
        self.basic_authorization(id_tag)
    )
```

Fonte: o autor

A classe *Notifier* é uma interface responsável por notificar de fato os outros componentes do sistema sobre eventos recebidos. Sendo uma interface ela pode ter qualquer tipo de implementação, seja notificações síncronas por chamadas de API ou por mensageria via um *Message Broker*.

4.4 Implementação dos casos de uso

Nesta Seção, será detalhada a implementação dos casos de uso definidos para o sistema. Será apresentado o processo de desenvolvimento de cada funcionalidade, desde a sua concepção inicial até a integração final no sistema. Vamos explorar os desafios técnicos encontrados, as soluções adotadas e as ferramentas utilizadas para garantir a eficiência e a eficácia da implementação. Esta análise permitirá compreender como cada caso de uso contribui para o funcionamento geral do sistema.

4.4.1 Listagem de estações de carga

Como descrito na Seção 3.3.1 é preciso agrupar dados de diversas fontes para a listagem, fornecendo uma visão mais rica e detalhada das informações. Isso é feito no microsserviço *save-server*, responsável pelas regras de negócio, sendo as fontes mensagens do protocolo OCPP enviadas pelo microsserviço *ocpp-events* ou informadas através de rotas de *API REST*.

Quando uma estação de carregamento se conecta ao sistema através do microsserviço *ocpp-events*, uma operação de *Boot Notification* é enviado pelo protocolo OCPP. O *ocpp-events* registra essa conexão em seu banco de dados em memória e envia um evento de *BootNotification* para todos os outros microsserviços através do *message broker*. Quando o microsserviço *save-server* recebe o evento de *BootNotification*, ele cria uma nova entrada para a estação de carga em seu banco de dados utilizando os dados recebidos do evento, ou atualiza as informações caso a estação já seja conhecida. A diferenciação entre as estações é feita utilizando seu identificador único.

A estação de carregamento, enquanto conectada, continua a atualizar o *ocpp-events* sobre qualquer mudança em seu estado através de diversas operações específicas. Essas operações incluem *MeterValues*, *StatusNotification*, *StartTransaction* e *StopTransaction*. Para cada uma delas, um evento correspondente é enviado ao *message broker*,

propagando as informações para todos os microsserviços interessados nessas atualizações. Essas mensagens desempenham um papel crucial na manutenção do estado preciso da estação de carregamento, como será detalhado posteriormente nesta Seção.

Além disso, existe uma rota de API dedicada ao enriquecimento dos dados de uma estação de carga, permitindo a inclusão de informações como nome da estação, endereço, latitude e longitude. É importante observar que em sistemas semelhantes, como o *Steve* e o *ChargeCore*, é comum informar esses dados à aplicação primeiro e somente então aceitar conexões de estações de carga que tenham sido previamente registradas dessa forma. No entanto, para este projeto, optou-se por seguir uma abordagem inversa, na qual as estações se conectam inicialmente e os dados adicionais são informados posteriormente. Essa escolha foi feita para facilitar o desenvolvimento e a experimentação dos componentes, porém, como contrapartida, isso pode comprometer a segurança do sistema, pois um atacante pode conectar várias estações simuladas, resultando em um alto consumo de recursos computacionais.

Os atributos de uma estação de carga no microsserviço *save-server* e suas fontes são os seguintes:

- identificador: Um identificador numérico único fornecido pelo próprio microsserviço *save-server*, interno a ele.
- identificador *OCPP*: Obtido através das mensagens do protocolo OCPP de *BootNotification*. É um identificador único fornecido pelo fabricante da estação de carga.
- estado da estação de carga: Pode ser "Conectada"ou "Desconectada". Uma estação sempre é considerada conectada até receber um evento de *Disconnect*, enviado pelo microsserviço *ocpp-events*
- latitude e Longitude: Informados para o microsserviço *save-server* através de uma rota de *API* por um usuário da SAVE
- endereço: Análogo à latitude e longitude, sendo o endereço postal da estação de carga
- nome: Um nome opcional e não único atribuído à estação de carga, utilizado para identificação pelos usuários. Por exemplo "Estação condomínio Palmeiras".

Cada estação de carga pode tem um número fixo de carregadores, tendo sempre pelo menos um. Na listagem das estações também é informado dados sobre cada carregador específico:

- *Id*: Identificador numérico do carregador único para a estação de carga
- Estado dos carregadores: "Em uso", "Disponível", "Reservado", "Indisponível"ou "Em erro". Esses estados são obtidos através de eventos enviados pelo microsserviço *ocpp-events*, conforme Tabela 4.1
- Usuário autenticado: RFID do usuário que autenticou por último para esse carregador. Pode ser vazio caso não exista nenhum usuário nesse estado.
- Data de último uso: Data da última vez em que o carregador saiu do estado de "Em uso".

Evento	Estado transicionado
BootNotification	Disponível
StatusNotification.Available	Disponível.
StatusNotification.Preparaing	Em uso.
StatusNotification.Charging	Em uso.
StatusNotification.SuspendedEV	Disponível.
StatusNotification.SuspendedEVSE	Disponível.
StatusNotification.Finishing	Em uso.
StatusNotification.Reserved	Reservado.
StatusNotification.Unavailable	Indisponível.
StatusNotification.Falted	Em erro.

Tabela 4.1 – Transição de estados de um carregador de uma estação de carregamento

4.4.2 Envio Histórico de cargas por e-mail

Conforme discutido na Seção 3.3.2, os usuários da SAVE expressaram interesse em receber periodicamente relatórios por e-mail contendo informações sobre o uso de seus equipamentos, ou seja, um histórico das operações de carga realizadas. Tendo isto em vista podemos separar esse caso de uso em dois, sendo o primeiro a disponibilização de um histórico e o segundo o envio deste histórico por e-mail.

4.4.3 Histórico de cargas

A informação de carga não é diretamente fornecida pelo protocolo OCPP. Para obter esses dados, é necessário armazenar e processar os eventos de *StartTransaction* e *StopTransaction*, a partir dos quais é possível deduzir as informações de carga realizada.

Uma sessão de carga é iniciada quando um usuário se autentica em uma estação de carregamento e conecta o carregador ao seu veículo elétrico. Nesse momento, a estação de carregamento envia uma operação de *StartTransaction* para o microsserviço *ocpp-events*, que inclui informações do medidor em Wh naquele instante, horário e o RFID identificado do usuário. O microsserviço *ocpp-events* então propaga uma mensagem de *StartTransaction* para os demais componentes do sistema.

Cada vez que o microsserviço *save-server* recebe um evento de *StartTransaction*, é assumido que uma nova carga foi iniciada. Portanto, um novo registro de carga é criado no banco de dados, contendo informações como a data de início, o RFID do usuário, a estação de carga, o identificador do conector e o valor do medidor (todos esses dados são fornecidos no evento recebido). Neste momento, o estado da carga é definido como "Carregando".

Quando o usuário finaliza a carga e remove o carregador do veículo, a estação de carga envia um evento de *StopTransaction*, contendo informações como a medição em Wh, o horário de conclusão da carga e o estado final da operação (incluindo possíveis estados de erro). O microsserviço *ocpp-events* então propaga o evento para o resto dos componentes.

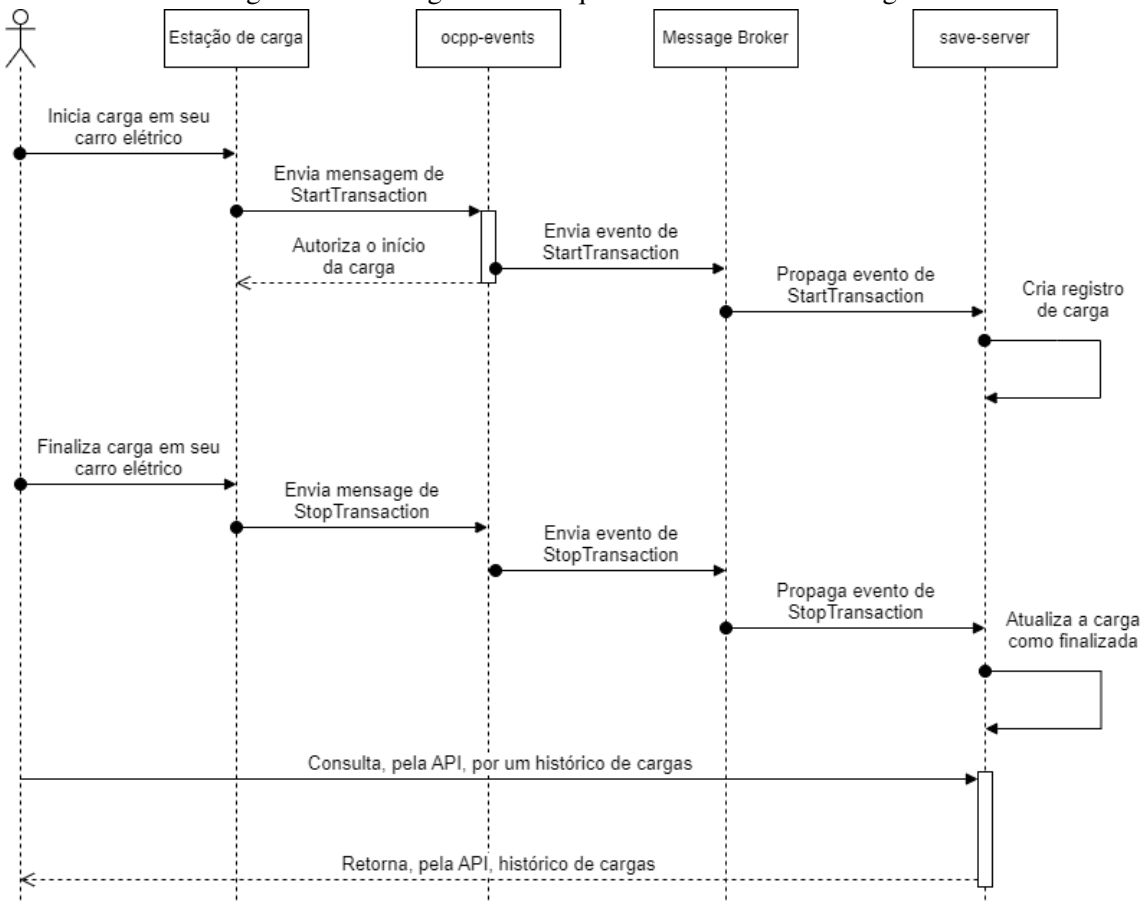
Ao receber o evento de *StopTransaction*, é possível correlacioná-lo com o evento anterior de *StartTransaction* utilizando o identificador da estação de carregamento e do conector. Esse evento também fornece o valor final do medidor em Wh, permitindo calcular o total de energia elétrica consumida durante a carga. Dessa forma, a entidade de carga no sistema é atualizada para refletir o término da operação.

Em casos de erro, o evento *StopTransaction* é sempre acionado e inclui um código de erro, permitindo identificar a causa do problema, como desconexão do carro elétrico, queda de energia ou perda de conexão com a internet. Quando a estação de carregamento perde comunicação com o sistema, o evento *StopTransaction* é enviado assim que uma nova conexão é restabelecida.

Os estados possíveis de uma carga são determinados pelos eventos recebidos. Inicialmente, ao ser criada, a carga recebe o estado "Carregando". Dependendo da *reason* recebida no evento de *StopTransaction*, o estado pode transicionar para "Erro"ou "Finalizada". Conforme listado na Seção 2.2, a carga será considerada "Finalizada"se a *reason* for *Local* ou *Remote*, e será marcada como "Erro"para qualquer outra *reason*.

Por fim, o usuário pode consultar, via API, um histórico de cargas com base em alguns filtros, como código da estação ou um intervalo de datas, como pode ser visto na

Figura 4.13 – Diagrama de Sequência de histórico de cargas



Fonte: o autor

Seção 4.5.2. A Figura 4.13 mostra o fluxo simplificado para esse caso de uso, omitindo a parte de autenticação do usuário.

4.4.4 Envio de e-mail

Foi implementada interface chamada *EmailSender* no microsserviço *save-server* para gerenciar o envio de e-mails, a qual é ilustrada na Figura 4.14. Essa estrutura mantém a implementação do envio de e-mails separada das lógicas de negócio. Uma rotina automatizada verifica periodicamente a base de dados de cargas, seleciona as que se enquadram em um intervalo de tempo estabelecido, converte-as em um arquivo *CSV* e envia este arquivo como anexo em um e-mail por meio de uma implementação da classe *EmailSender*.

Para o componente de envio de e-mails, optou-se por uma configuração inicial simples, empregando um servidor SMTP local desenvolvido em *Python* e utilizando a biblioteca *smtplib*. Decidiu-se não incluir camadas de SSL no servidor de e-mail para

Figura 4.14 – Interface *EmailSender*

```python
class EmailSender(ABC):
    @abstractmethod
    def send_email(self, request: EmailSenderRequest):
        pass
```

Fonte: o autor

manter a simplicidade, o que, infelizmente, resulta na rejeição de mensagens por alguns provedores, como o *Gmail* da *Google*. Visando aprimorar o sistema, seria vantajoso migrar para um serviço de e-mail mais sofisticado, como o SES (*Simple Email Service*) da AWS (*Amazon Web Services*). Para tal, seria necessário apenas adaptar a interface *EmailSender* no microsserviço *save-server* para integrar com este novo serviço [AWS 2024].

Considerando uma configuração de periodicidade semanal, com disparos de e-mails programados para os domingos às 23:00 horas, o sistema consulta o banco de dados em busca de todas as cargas finalizadas após o domingo anterior às 23:00 horas e antes do domingo corrente, também às 23:00 horas. Os dados selecionados são então processados pelo componente *CsvFactory*, que é encarregado de converter os dados para o formato CSV e salvá-los no disco. Após isso, é gerada uma requisição de e-mail que anexa o arquivo CSV e define como destinatário o endereço de e-mail de um administrador, o qual é configurável.

4.5 Apis implementadas

Nesta Seção estão documentadas as rotas de API mais importantes para os usuários do sistema. Todas essas rotas ficam no microsserviço *save-server* e foram implementadas usando o *framework FastAPI*, como mostrado na Seção 4.1.6

Em cada Seção será falado sobre qual o objeto da rota de API, sua rota de fato, os dados retornados, os filtros implementados e os tipos de retorno.

4.5.1 Estações de cargas

Essa rota de estações de carregamento recupera todas as estações que atendem aos critérios de busca. É possível aplicar um filtro opcional com base no estado da estação,

por exemplo, todas as estações que estão conectadas.

A rota é um *GET save-server/charging-stations/* e retorna um conjunto de estações de carregamento que possuem os seguintes atributos:

- Estado: conectada, desconectada, etc
- Data da última conexão
- Data de primeira conexão
- Alguns dados técnicos fornecidos pelos eventos de *MeterValues*, como voltagem, corrente oferecida, etc.
- Nome da estação (opcional)
- Latidude e longitude (opcional)
- Endereço postal (opcional)
- Um conjunto de carregadores associados

  Os carregadores de uma estação de carga possuem os seguintes campos:

- Estado: estado atual do carregador (disponível, em uso, reservado, etc).
- Usuário: RFID do usuário que está utilizado. Será nulo caso não esteja em uso.
- Data de última carga: última vez que algum usuário realizou uma carga nesse carregador.

Caso não existam estações de carga que satisfaçam o filtro de busca a requisição também retorna com sucesso e *status code* 200, com o *body* contando uma lista vazia.

4.5.2 Cargas

A rota de cargas recupera todas as informações de cargas que atenderem os critérios de busca. É possível aplicar um filtro de cargas a partir de uma data, cargas até uma data e por estação de carregamento, todos opcionais.

A rota é um *GET save-server/charges/* e retorna um conjunto de cargas que possuem os seguintes atributos:

- Identificador da carga

- Identificador da estação de carga
- RFID do usuário que fez a carga
- Estado da carga
- Data de início da carga
- Data de fim da carga
- Total carregado, em Wh
- Tempo total carregando

Caso não existam informações de cargas no banco de dados que satisfaçam o filtro a requisição também retorna com sucesso e *status code* 200, com o *body* contendo uma lista vazia.

4.5.3 Adicionar dados à uma estação de carga

Para adicionar dados de regras de negócios às estações de carga, como nome e endereço, também foi criada uma rota de API. A rota é um POST *save-server/charging-station/<identificador da estação>/data* contendo um *body* com os dados, e sendo o identificador da estação o identificador OCPP. Todos os dados são opcionais, mas deve existir pelo menos um deles:

- Nome: um nome amigável para a estação, como por exemplo "Estação Condomínio Porto".
- Endereço: Endereço postal da estação, como por exemplo "Rua Sobradinho, 269, Bairro São Jorge, Novo Hamburgo".
- Latitude: número inteiro representando a latitude da estação de carga
- Longitude: número inteiro representando a longitude da estação de carga

A resposta para essa requisição, para casos de sucesso, são os dados atuais atualizados. Por exemplo se a estação já tiver informações de endereço e for enviado uma requisição apenas com o nome seria retornado tanto o nome quanto as informações anteriores, nesse caso as de endereço. Se for preciso mudar o nome basta mandar outra requisição com o nome atualizado.

Caso a estação não exista no banco de dados é retornado uma resposta de erro com *status code* 404.

4.5.4 Eventos Publicados

Para ajudar no desenvolvimento e depuração de eventos foi criado o microsserviço *event-store*, como descrito na Seção 3.1.3. Esse serviço provê uma rota para recuperar eventos publicados pelo microsserviço *ocpp-events*.

A rota é um GET *event-store/events/* e retorna todos eventos publicados. Há 3 tipos de filtros possíveis, por eventos a partir de certa data, eventos até certa data ou por um tipo de eventos específico.

Os dados dos eventos fornecidos são os seguintes:

- Tipo do evento: por exemplo *BootNotification*, *StartTransaction*, etc.
- Data do evento: Data de envio do evento.
- Dados do evento: Fornecidos como JSON, são uma cópia do evento recebido, sem uma estrutura pré definida.

4.6 Validações

Esta seção descreve os testes realizados para validar o correto funcionamento do sistema desenvolvido, tanto em cenários de sucesso quanto de falha. O objetivo desses testes é garantir que o sistema possa lidar adequadamente com as diferentes situações que podem ocorrer durante o uso das estações de carregamento, assegurando a integridade dos dados e a confiabilidade das operações. Nas subseções a seguir, serão apresentados os cenários de sucesso, que comprovam o funcionamento esperado do sistema em condições normais, e os casos de falha, que avaliam a robustez do sistema frente a situações adversas.

4.6.1 Cenários de sucesso

Para validar os cenários de sucesso, foram realizadas conexões com um carregador real, de propriedade da *start-up*. Inicialmente, o carregador foi ligado e verificou-se que a conexão estava correta, com o sistema reconhecendo e exibindo o carregador e a estação

de carga como disponíveis. Em seguida, iniciou-se um processo de carga, confirmando que o sistema manteve-se atualizado, refletindo o estado de uso tanto do carregador quanto da estação de carga. Ao finalizar a carga, verificou-se que os estados da estação e do carregador retornaram ao status de disponíveis. Além disso, foi confirmado que o registro da carga foi realizado com sucesso, apresentando as informações corretas.

Aqui está uma versão revisada do parágrafo:

4.6.2 Casos de falha

Para validar os cenários de falha, também foi utilizado um carregador real da *start-up*. A validação envolveu estabelecer uma conexão entre a estação de carga e o sistema, iniciando uma sessão de carregamento. A partir desse estado, foram avaliados quatro cenários distintos, com suas respectivas expectativas:

- Carregamento finalizado pelo usuário usando o botão de emergência: foi verificado que o estado final da carga foi registrado como "erro", mas com os dados de carregamento até o momento da interrupção devidamente armazenados. Além disso, confirmou-se que tanto o carregador quanto a estação retornaram ao estado de "disponível".

- Desligamento da internet do carregador durante o carregamento: foi validado que o estado da sessão de carregamento permaneceu em aberto, mesmo com a estação sendo marcada como "indisponível". Após o restabelecimento da conexão, a carga foi finalizada como sucesso, e o sistema conseguiu recuperar corretamente os dados, como a energia total utilizada e a data de finalização.

- Desligamento da internet do carregador durante o carregamento e finalização da carga *offline*: similar ao caso anterior, mas com o carregamento sendo encerrado pelo usuário durante a indisponibilidade da rede. Foi validado que, após a recuperação da conexão, o sistema conseguiu recuperar as informações perdidas e registrar os dados de carga corretamente, incluindo a data correta de finalização de carregamento.

- Desligamento da energia elétrica do carregador durante o carregamento: análogo ao cenário anterior: a estação foi marcada como "indisponível"no sistema, e os dados de carregamento permaneceram em aberto. Após o restabelecimento da energia

elétrica, o sistema recuperou as informações de carga, com o estado final da carga registrado como "erro", mas com os dados de energia utilizada e a data de finalização corretamente atualizados.

5 CONCLUSÃO

Esse trabalho partiu a partir de uma demanda da *start-up* SAVE de ter um software capaz de se comunicar com estações de carregamento de carros elétricos, através do protocolo OCPP, trazendo informações sobre disponibilidade de carregadores, cargas realizadas, etc. Depois de analisadas algumas alternativas *open-source* decidiu-se por uma implementação proprietária partindo do zero, capaz de fornecer uma API com operações e dados importantes para o dia-a-dia da empresa.

Dois casos de uso prioritários foram definidos em colaboração com a equipe da SAVE para orientar a implementação do sistema: uma API que lista estações de carregamento e seus estados (disponível, em uso, desligada, etc.) e o envio de relatórios de carregamento por e-mail. Além desses requisitos funcionais, o software precisa atender a critérios técnicos importantes. Ele deve ser robusto (capaz de lidar eficazmente com erros e falhas), flexível (permitindo a adição fácil de novas funcionalidades e acomodando a evolução da *start-up*), e escalável (para otimizar o uso de recursos computacionais conforme a demanda aumentasse).

Nesse contexto, foi definido um software do tipo cliente-servidor, com uma arquitetura de microsserviços desacoplada. Há um microsserviço dedicado à comunicação com os carregadores, outro para implementar as regras de negócio e um terceiro para auxiliar no desenvolvimento. Cada microsserviço pode escalar e evoluir independentemente, garantindo uma boa escalabilidade e flexibilidade. Além disso, existem políticas de reprocessamento de mensagens para lidar com casos de erro, bem como mecanismos para tratar eventuais perdas de conexão com a estação de carregamento.

O desenvolvimento deste trabalho foi bastante desafiador, sendo a principal dificuldade a escassez de documentação adequada sobre o protocolo OCPP. Embora o manual oficial do protocolo forneça uma visão geral dos parâmetros utilizados, ele não explica detalhadamente o que são esses parâmetros e para que servem. Para superar essa limitação, a implementação foi guiada pelo uso de um simulador simples de código aberto e, posteriormente, validada com um carregador real da equipe SAVE, em reuniões síncronas com a equipe. A dificuldade em entender as capacidades do protocolo também atrasou significativamente a definição dos casos de uso, o que resultou na redução do escopo do trabalho.

A decisão de implementar um software proprietário do zero traz diversas vantagens, como maior controle sobre o desenvolvimento e possibilidade de ter funcionalidades

únicas, que dificilmente uma implementação de código aberto ofereceria, como cupons de desconto, facilidade de pagamento, etc. Porém um desenvolvimento também traz custos: esse trabalho ofereceu uma plataforma *backend* com algumas funcionalidades iniciais, porém não foi implementada nenhuma interface para o usuário, o que dificulta a utilização pela equipe da SAVE, fazendo necessário uma implementação *frontend* no futuro.

Ainda considerando mais passos para a evolução do sistema, é essencial implementar uma gestão aprimorada de usuários, permitindo a criação e administração de senhas, além de atribuir papéis distintos a cada usuário, como administrador, operador e usuário. Outro aspecto crucial é a utilização de SSL nas requisições HTTP, assegurando que os dados transmitidos não possam ser facilmente interceptados por atacantes. Isso protegerá informações sensíveis contra interceptações, garantindo a integridade e a confidencialidade da comunicação entre clientes e servidores.

REFERÊNCIAS

ABVE, A. B. d. V. E. **Elétricos crescem em todas as regiões do Brasil**. 2024. <https://abve.org.br/veiculos-eletricos-crescem-em-todo-o-pais/>. Acesso em: 2 de julho de 2024.

AWS, A. W. S. **Amazon Simple Email Service**. 2024. <https://aws.amazon.com/pt/ses/>. Acesso em: 20 de julho de 2024.

CHOU, W. Inside ssl: the secure sockets layer protocol. **IT Professional**, v. 4, n. 4, p. 47–52, 2002.

EVANS, E. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. [S.l.]: Addison-Wesley, 2004.

FETTE, I.; MELNIKOV, A. **The WebSocket Protocol**. 2011. <https://tools.ietf.org/html/rfc6455>. (RFC 6455). Acesso em: 2 de julho de 2024.

FIELDING, R. T. **Architectural styles and the design of network-based software architectures**. Tese (Publication) — University of California, Irvine, 2000. Disponível em: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.

FOWLER, S. J. **Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2016. ISBN 1491965975.

KLENSIN, D. J. C. **Simple Mail Transfer Protocol**. RFC Editor, 2008. RFC 5321. (Request for Comments, 5321). Acesso em: 2 de julho de 2024. Disponível em: <https://www.rfc-editor.org/info/rfc5321>.

KUMAR, A.; PATEL, D. A comparative study of mongodb and mysql. **International Journal of Modern Trends in Engineering and Research**, 2014.

MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. **Linux Journal**, Belltown Media, Inc., n. 239, 2014.

NIELSEN, H. et al. **Hypertext Transfer Protocol – HTTP/1.1**. RFC Editor, 1999. RFC 2616. (Request for Comments, 2616). Acesso em: 2 de julho de 2024. Disponível em: <https://www.rfc-editor.org/info/rfc2616>.

OCA, O. C. A. **Manual Ocpp versão 1.6**. 2017. <https://openchargealliance.org/my-oca/ocpp/>. Acesso em: 2 de julho de 2024.

OCA, O. C. A. **About us - Open Charge Alliance**. 2024. <https://openchargealliance.org/about-us/>. Acesso em: 2 de julho de 2024.

PATHAK, A.; KALAIARASAN, C. Rabbitmq queuing mechanism of publish subscribe model for better throughput and response. In: **2021 Fourth International Conference on Electrical, Computer and Communication Technologies (ICECCT)**. [S.l.: s.n.], 2021. p. 1–7.

RICHARDSON, L.; RUBY, S. **RESTful Web Services**. [S.l.]: O'Reilly Media, Inc., 2007.

SCHMUTZLER, J.; ANDERSEN, C. A.; WIETFELD, C. Evaluation of ocpp and iec 61850 for smart charging electric vehicles. In: **2013 World Electric Vehicle Symposium and Exhibition (EVS27)**. [S.l.: s.n.], 2013. p. 1–12.

STOJA, S.; VUKMIROVIC, S.; JELACIC, B. Publisher/subscriber implementation in cloud environment. In: **2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing**. [S.l.: s.n.], 2013. p. 677–682.

STONEBRAKER, M.; WEISBERG, A.; ZDONIK, S. B. Postgresql: The world's most advanced open source database. **Communications of the ACM**, ACM New York, NY, USA, v. 61, n. 10, p. 78–85, 2018.

TANENBAUM, A. S.; STEEN, M. V. **Distributed Systems: Principles and Paradigms**. [S.l.]: Pearson Prentice Hall, 2007.

Venkata Pruthvi, T. et al. Implementation of OCPP Protocol for Electric Vehicle Applications. In: **E3S Web of Conferences**. [S.l.: s.n.], 2019. (E3S Web of Conferences, v. 87), p. 01008.

VERNON, V. **Implementing Domain-Driven Design**. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN 978-0-321-83457-7.

VINOSKI, S. Advanced message queuing protocol. **Internet Computing, IEEE**, v. 10, p. 87 – 89, 12 2006.

WEAVER, A. Secure sockets layer. **Computer**, v. 39, n. 4, p. 88–90, 2006.