Planejamento de Viagens para Recargas de Veículos Elétricos

[1]Brenda A. Trindade Oliveira, [2]Letícia Gonçalves Souza, [3]Naylane do N. Ribeiro

[1]Departamento de Tecnologia – Universidade Estadual de Feira de Santana (UEFS)
44036-900 – Feira de Santana – , Bahia

brendatrindde@gmail.com, letigsouza03@gmail.com,
naylaneribeiro4@gmail.com

Abstract. *The growing demand for electric vehicles requires efficient charging infrastructure. The challenge lies in optimizing trip planning and coordinating reservations between companies. This work presents a distributed system for charging management. It facilitates communication between vehicles and points, optimizing the reservation process. The system was developed in Go, using Docker for orchestration. Communication occurs via MQTT and REST API, with data persisted in JSON. Mutexes ensure safe concurrency of operations. Functionalities include trip scheduling and point reservation. The system allows confirmation, cancellation and automatic release by timeout. Distributed operations maintain atomicity and data integrity.*

Resumo. *A crescente demanda por veículos elétricos exige infraestrutura de recarga eficiente. O desafio reside em otimizar o planejamento de viagens e coordenar reservas entre empresas. Este trabalho apresenta um sistema distribuído para gerenciamento de recargas. Ele facilita a comunicação entre veículos e pontos, otimizando o processo de reserva. O sistema foi desenvolvido em Go, utilizando Docker para orquestração. A comunicação ocorre via MQTT e API REST, com dados persistidos em JSON. Mutexes garantem a concorrência segura das operações. As funcionalidades incluem programação de viagem e reserva de pontos. O sistema permite confirmação, cancelamento e liberação automática por timeout. As operações distribuídas mantêm a atomicidade e integridade dos dados.*

1. Introdução

O crescente cenário dos veículos elétricos impulsiona a necessidade de infraestruturas de recarga robustas e eficientes. Este projeto propõe um sistema distribuído que simula o gerenciamento de recargas para esses veículos, otimizando o planejamento de viagens de longa distância. O foco é garantir que os veículos encontrem e reservem pontos de recarga de diferentes empresas de forma eficiente e segura.

Gerenciar a alocação de pontos de recarga entre múltiplos veículos e empresas apresenta desafios complexos. A coordenação de reservas distribuídas e a garantia da integridade dos dados demandam soluções robustas.

Este trabalho propõe um sistema distribuído para simular o gerenciamento de recargas de veículos elétricos. Ele facilita a comunicação entre veículos e pontos, otimizando o processo de reserva de forma eficaz.

A aplicação é composta por um broker MQTT, múltiplos servidores representando empresas e veículos clientes. Todos os componentes são orquestrados utilizando Docker Compose, o que assegura isolamento e escalabilidade para a simulação. A comunicação entre cliente-servidor é viabilizada pelo MQTT, enquanto a API REST gerencia a coordenação entre servidores. Mutexes e goroutines garantem a concorrência segura das operações.

As funcionalidades implementadas incluem a programação de viagem e operações de reservar e cancelar reserva. O sistema suporta cancelamento e liberação automática por timeout, mantendo a atomicidade e integridade das operações distribuídas. O restante deste trabalho está organizado da seguinte forma: A Seção 2 apresenta os fundamentos teóricos relacionados com o problema. A Seção 3 discute aspectos de implementação e testes da solução. A Seção 4 apresenta e avalia os resultados. No final, na Seção 5, as conclusões e reflexões sobre os conhecimentos adquiridos

2. Fundamentação Teórica

Esta seção apresenta conceitos teóricos sobre temas oportunos relacionados com o sistema.

2.1. MQTT e Brokers

O MQTT (Message Queuing Telemetry Transport) é um protocolo de mensagens leve, baseado no padrão Publish/Subscribe (Publicar/Assinar). Sua arquitetura é centrada em um componente intermediário, o Broker MQTT. O broker atua como um ponto centralizado de distribuição de mensagens, desacoplando os clientes (publicadores e assinantes). Publicadores enviam mensagens para tópicos no broker, sem precisar conhecer os assinantes. Assinantes, por sua vez, registram-se para receber mensagens de tópicos específicos, sem precisar conhecer os publicadores. Esse modelo de comunicação assíncrona garante flexibilidade e escalabilidade, pois os clientes não precisam estar online simultaneamente para trocar informações.

No contexto deste projeto, o Eclipse Mosquitto é utilizado como broker MQTT. Ele gerencia a troca de mensagens entre os veículos (clientes) e os servidores das empresas de recarga, permitindo que as solicitações de reserva, pré-reserva e cancelamento sejam comunicadas de forma eficaz e em tempo real.

2.2. Interfaces de Programação de Aplicações (APIs)

As Interfaces de Programação de Aplicações (APIs) são conjuntos de definições e protocolos que permitem que diferentes softwares se comuniquem e interajam entre si. Funcionam como uma ponte, especificando como os componentes de software devem interagir, definindo os tipos de chamadas ou requisições que podem ser feitas, os formatos de dados que devem ser usados, as convenções a serem seguidas e as funcionalidades disponíveis. O principal objetivo de uma API é abstrair a complexidade interna de um sistema, expondo apenas as funcionalidades necessárias para que outros desenvolvedores possam utilizá-las de forma simplificada.

Em sistemas distribuídos, as APIs permitem a modularidade e a interoperabilidade entre serviços independentes. Elas facilitam a integração de funcionalidades e dados entre aplicações distintas, mesmo que desenvolvidas em diferentes linguagens de programação ou rodando em plataformas diversas. Isso promove a reutilização de código, a escalabilidade e a manutenção de sistemas complexos.

2.3. API REST

A API REST (Representational State Transfer) é um estilo arquitetural para sistemas distribuídos que define um conjunto de princípios e restrições para a troca de dados. Ela se baseia no protocolo HTTP e utiliza operações padronizadas (GET, POST, PUT, DELETE) para manipular recursos. Uma de suas principais características é ser stateless (sem estado), o que significa que cada requisição do cliente ao servidor contém todas as informações necessárias para que o servidor a entenda e a processe, sem depender de contexto armazenado em sessões anteriores. Isso promove a escalabilidade e a resiliência do sistema.

No contexto de sistemas distribuídos, a API REST é amplamente utilizada para comunicação síncrona e coordenação entre diferentes serviços. Ela permite que diferentes componentes de um sistema interajam de forma padronizada, trocando informações em formatos como JSON (JavaScript Object Notation), que é leve e de fácil leitura para máquinas e humanos.

2.4. Atomicidade e Consistência

Em sistemas distribuídos, a atomicidade é uma propriedade crítica que garante que uma sequência de operações seja tratada como uma única unidade lógica e indivisível. Isso significa que a transação é executada por completo ou não é executada de forma alguma. Não há estados intermediários visíveis para outros componentes do sistema. Em caso de falha em qualquer etapa, todas as operações realizadas até aquele ponto são desfeitas (rollback), retornando o sistema ao seu estado original antes do início da transação. Este conceito é essencial para manter a consistência dos dados, que assegura que o estado do sistema permaneça válido após a execução de operações concorrentes.

3. Metodologia, Implementação e Testes

Esta seção, especifica como o problema foi solucionado utilizando as teorias, métodos e tecnologias descritas na Seção 2. A implementação do sistema de planejamento de viagens e recargas de veículos elétricos foi pautada por uma arquitetura distribuída e modular, visando robustez e escalabilidade. Todo o ambiente foi orquestrado utilizando Docker e Docker Compose, permitindo que cada componente fosse executado em contêineres isolados. A persistência dos dados, como informações de empresas, regiões e veículos, é realizada por meio de arquivos JSON, montados como volumes nos contêineres.

3.1. Arquitetura do Sistema

O sistema foi projetado para funcionar em ambiente de containers Docker interconectados por uma rede interna definida no docker-compose, garantindo isolamento, escalabilidade e simulação de concorrência.

A Figura 1 apresenta o **Diagrama de Sequência do Sistema**, que ilustra o fluxo de comunicação entre os principais componentes distribuídos da arquitetura. O diagrama descreve, passo a passo, como as mensagens são trocadas entre o veículo, o broker MQTT e os servidores das empresas durante os processos de programação de viagem, pré-reserva, confirmação e liberação de pontos de recarga. Essa representação gráfica é essencial para compreender como o sistema mantém a atomicidade, sincronização e a consistência das operações em um ambiente distribuído.

O sistema é composto por três componentes principais:

- Broker MQTT: Responsável por viabilizar a comunicação assíncrona entre todos os serviços. Utiliza a imagem oficial do Eclipse Mosquitto e é exposto na porta 1883.
- Servidor: Desenvolvido em Go (Golang), cada servidor representa uma empresa de recarga. Eles expõem uma API REST para coordenação entre si e gerenciam dados de empresas, regiões e pontos de recarga, carregados de arquivos JSON. Recebem solicitações de veículos via MQTT e utilizam goroutines e mutexes para garantir concorrência segura.
- Veículo: Implementado também em Go, simula um cliente que interage com o sistema via terminal. Ele permite ao usuário informar origem, destino, nível de bateria e autonomia, recebendo rotas sugeridas e pontos de recarga necessários. A comunicação com os servidores ocorre via MQTT, publicando solicitações e recebendo respostas em tópicos específicos.

Cada empresa é implementada como servidor. O sistema possui 3 empresas previamente cadastradas que podem se conectar simultaneamente. Cada empresa possui a própria área de cobertura, baseadas em capitais do Nordeste, sendo:

- Empresa N-Sul: responsável por Salvador, Aracaju e Maceió;
- Empresa N-Centro: responsável por Recife, João Pessoa e Natal;
- Empresa N-Norte: responsável por Fortaleza, Teresina e São Luís.

Essas informações são armazenadas em um arquivo json.

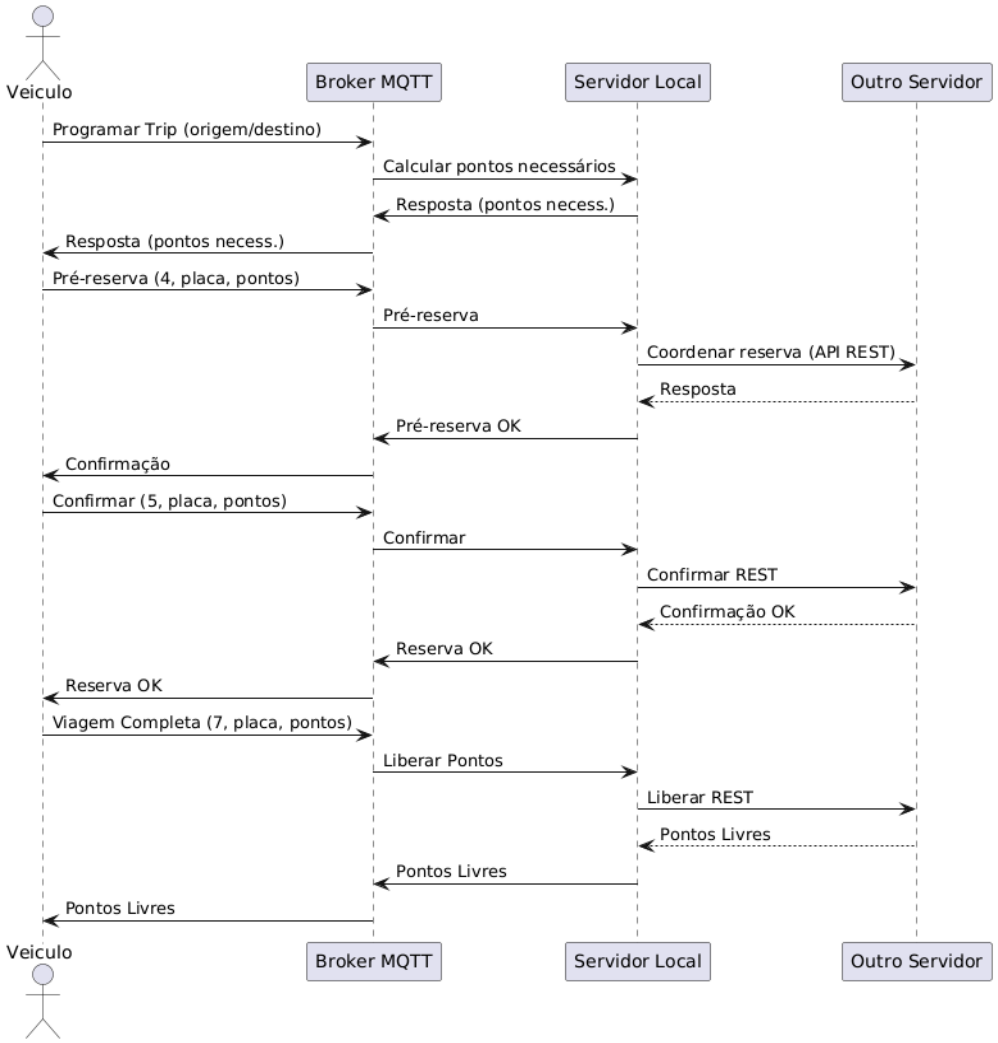

Figura 1. Diagrama de Sequência do Sistema

Fonte: Autor

3.2. Protocolos de comunicação

A comunicação dentro do sistema é híbrida, combinando o protocolo MQTT para interações assíncronas entre veículos e servidores, e a API REST para coordenação síncrona entre os servidores das diferentes empresas.

O protocolo MQTT é utilizado para que os veículos se comuniquem de forma eficiente com os servidores. As mensagens seguem um formato padronizado que inclui um código de operação, a placa do veículo e, quando necessário, uma lista de pontos de recarga. Por exemplo, o código "1" representa uma solicitação de reserva, "4" indica uma pré-reserva, e "7" refere-se à liberação de pontos ao término da viagem. Essa abordagem permite uma comunicação rápida e desacoplada, essencial em ambientes distribuídos.

Por outro lado, a API REST é empregada para coordenar ações críticas entre os servidores, garantindo que as operações de reserva mantenham a consistência global do

sistema. São utilizados endpoints como **/api/reserva**, **/api/confirmar-prereserva** e **/api/cancelamento**, todos com trocas de dados no formato JSON. Essa padronização assegura interoperabilidade, clareza na estrutura das requisições e facilidade de manutenção do sistema.

Essa arquitetura híbrida permite explorar o melhor dos dois mundos: a leveza e eficiência do MQTT para comunicação veicular e a confiabilidade e controle da REST API na lógica de comunicação entre empresas.

3.3. Gerenciamento de Concorrência e Integridade

Para garantir a integridade dos dados e prevenir condições de corrida em um ambiente distribuído, o sistema emprega mecanismos de concorrência.

Cada ponto de recarga possui um mutex (lock) próprio, garantindo exclusão mútua em operações críticas. Antes de modificar o estado de um ponto (por exemplo, ao reservar), o servidor adquire o lock com lock.Lock(), e o libera com lock.Unlock() após a operação. Isso impede que dois veículos reservem o mesmo ponto simultaneamente.

As operações de reserva são projetadas para serem atômicas: ou todos os pontos solicitados são reservados, ou nenhum deles é. Em caso de falha em algum servidor durante a coordenação, as reservas temporárias são automaticamente canceladas para manter a consistência do sistema.

O sistema implementa timeout automático, que libera pontos que foram pré-reservados, mas não foram confirmados dentro de um período definido, otimizando a disponibilidade dos recursos.

Além do mais, o Go permite o uso de goroutines para lidar com múltiplas conexões simultâneas, processando requisições de forma concorrente sem bloquear a execução principal do servidor.

4. Resultados e discussões

Foram realizados testes para verificar a funcionalidade do sistema. As instâncias dos servidores (empresas) e cliente (veículo) foram executadas em contêineres isolados, simulando um ambiente distribuído.

4.1 Funcionalidades alcançadas

O sistema faz com sucesso a programação de viagem e a sugestão de pontos de recarga, permitindo que o usuário informe origem e destino. A simulação da viagem, baseada na autonomia e nível de bateria do veículo, identifica os pontos de recarga necessários ao longo da rota, otimizando o percurso. A funcionalidade de pré-reserva permite a alocação temporária desses pontos antes da confirmação da viagem, garantindo a disponibilidade em um ambiente multi-empresa e multi-servidor. Após a pré-reserva, o sistema suporta a confirmação para converter alocações temporárias em definitivas, com coordenação entre os servidores para assegurar atomicidade e consistência global.

O cancelamento manual de reservas e pré-reservas é permitido, complementado por um mecanismo de timeout automático que libera pontos não confirmados, evitando o bloqueio desnecessário de recursos. Por fim, a liberação de pontos pós-viagem permite

que o veículo sinalize o término do uso, atualizando o estado do sistema e tornando-os disponíveis para outros usuários.

4.2 Discussões

Durante os testes, foi possível observar que a comunicação entre os servidores e o cliente-servidor ocorreu de forma estável, com as mensagens sendo corretamente enviadas, recebidas e interpretadas.

A integração de MQTT e API REST se mostrou uma solução robusta para comunicação assíncrona e síncrona, respectivamente. MQTT permitiu a troca de mensagens eficiente entre veículos e servidores, enquanto a API REST foi fundamental para a coordenação de operações distribuídas e a garantia da atomicidade das reservas entre as diferentes "empresas"

O sistema atingiu os objetivos de demonstrar a coordenação e a concorrência em um sistema distribuído para recargas de veículos elétricos. A simulação da rota e a interação com o usuário via terminal validam o conceito de um planejamento de viagem interativo e otimizado.

6. Conclusão e trabalhos futuros

Através da integração de tecnologias como MQTT para comunicação assíncrona e API REST para coordenação síncrona entre os serviços, o sistema provou ser uma solução sólida para otimizar o processo de reserva de pontos de recarga em um ambiente distribuído.

A implementação em Go, com a utilização estratégica de goroutines e mutexes, assegurou a garantia de atomicidade e integridade nas operações de reserva, prevenindo condições de corrida e mantendo a consistência do estado global. O sistema alcançou seus objetivos de oferecer funcionalidades como programação de viagem, pré-reserva, confirmação, cancelamento e liberação automática de pontos.

Para aprimoramentos futuros do sistema, diversas extensões podem ser consideradas, como integração com Banco de Dados, interface gráfica e expandir a aplicação para mais regiões.

Referências

Donovan, A. A. and Kernighan, B. W. (2016). The Go Programming Language. Addison-Wesley.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley.

Tanenbaum, A. S. and Van Steen, M. (2007). Distributed Systems: Principles and Paradigms (2nd ed.). Pearson Prentice Hall.