Sistema de Pontos de Recarga

Gabriel Santos Cruz [1], Luís Eduardo Leite Azevedo [2], Luis Guilherme Nunes Lima de Oliveira [3]

Departamento de Tecnologia – Universidade Estadual de Feira de Santana (UEFS) 44036–900 – Feira de Santana – Bahia

gabrielsantoscruz61@gmail.com[1] , eduardoleite705@gmail.com[2], guigonl16@gmail.com[3]

Resumo. *O aumento da frota de carros elétricos no Brasil enfrenta desafios críticos devido à infraestrutura insuficiente de recarga, como pontos concentrados em áreas urbanas, falta de informações em tempo real e longos tempos de espera. Para solucionar esse problema, este trabalho propõe um sistema inteligente baseado em arquitetura cliente-servidor, desenvolvido em containers Docker, que permite a comunicação direta entre veículos e pontos. Sua implementação foi feita através de sockets TCP/IP, sem uso de frameworks de mensagens, seguindo as restrições comerciais e o uso de containers Docker. O sistema demonstrou eficiência na distribuição de demanda e no suporte a grandes acessos simultâneos.*

1. Introdução

A mobilidade elétrica tem ganhado destaque no Brasil, impulsionada pelo crescimento anual da frota de veículos sustentáveis. Segundo a Associação Brasileira do Veículo Elétrico (2024), o país registrou mais de 300 mil carros elétricos em circulação, porém a infraestrutura de recarga ainda é insuficiente e mal distribuída. Motoristas enfrentam dificuldades como falta de pontos disponíveis, longos tempos de espera e ausência de informações em tempo real sobre localização e status dos carregadores. Esses desafios limitam a adoção em larga escala dessa tecnologia, especialmente em viagens interurbanas.

Diante desse cenário, este trabalho apresenta o desenvolvimento de um sistema inteligente de gerenciamento de recargas para veículos elétricos, baseado em uma arquitetura cliente-servidor implementada com sockets TCP/IP. A solução, projetada para operar em containers Docker, permite que os veículos identifiquem pontos de recarga disponíveis, considerando fatores como proximidade e ocupação, além de realizar reservas automatizadas e integração com pagamentos digitais (não implementado, pois apenas é simulação). A abordagem adotada evita frameworks de terceiros, atendendo a requisitos comerciais específicos, e prioriza a escalabilidade para futuras expansões da rede.

Os resultados obtidos demonstram a eficácia do sistema na redução do tempo médio de espera e na otimização da distribuição de demanda entre os pontos de recarga. Testes realizados em ambiente controlado validaram a robustez da comunicação entre os módulos do sistema, bem como a precisão na atualização de dados em tempo real. Este

relatório está organizado da seguinte forma: a Seção 2 descreve os fundamentos teóricos que embasam o projeto, incluindo conceitos de redes TCP/IP e mobilidade elétrica. A Seção 3 detalha a metodologia de desenvolvimento, com ênfase na arquitetura do sistema e nas escolhas de implementação. A Seção 4 apresenta os resultados dos testes. Por fim, a Seção 5 traz as conclusões e propostas para trabalhos futuros, destacando o potencial de aprimoramento e expansão da plataforma.

2. Fundamentação Teórica

2.1. Arquitetura Cliente-Servidor

A arquitetura cliente-servidor adotada no projeto é fundamental para sistemas distribuídos, onde os clientes (veículos elétricos) solicitam serviços a um servidor central (nuvem), que gerencia os pontos de recarga, funcionando como um servidor híbrido. Essa abordagem permite centralizar a lógica de negócios, como disponibilidade de carregadores e cálculo de rotas, enquanto os clientes focam na interação com o usuário e coleta de dados (Tanenbaum & Van Steen, 2007). A implementação em Python utilizou o módulo **socket** para comunicação direta via TCP/IP, garantindo baixa latência e confiabilidade na troca de mensagens.

2.2. Comunicação via Sockets TCP/IP

O protocolo TCP/IP foi escolhido por sua confiabilidade na entrega de pacotes e controle de congestionamento, essencial para atualizações em tempo real da disponibilidade de pontos de recarga. Em Python, a biblioteca padrão **socket** permite criar conexões entre clientes e servidor, com métodos como **socket()** para inicialização e **bind()/listen()** para gerenciar múltiplas requisições (Python Software Foundation, 2023). A ausência de frameworks de mensagens (como RabbitMQ) atendeu às restrições comerciais do projeto, exigindo a implementação manual de serialização de dados (ex: JSON) para estruturar as mensagens.

2.3. Conteinerização com Docker

A conteinerização via Docker garantiu portabilidade e isolamento entre os módulos do sistema (veículo, nuvem e ponto de recarga). Cada componente foi encapsulado em contêineres independentes, com imagens baseadas em Python 3.9 e configurações de rede personalizadas para simular um ambiente distribuído (Merkel, 2014). Isso facilitou testes escaláveis, como a simulação de múltiplos veículos conectados simultaneamente ao servidor.

2.4. Processamento Assíncrono em Python

Para lidar com múltiplas conexões concorrentes, o servidor empregou threads via módulo **threading** da Python, garantindo que requisições de diferentes clientes não bloqueassem

o sistema. Técnicas de sincronização, como Locks (**threading.Lock()**), evitaram condições de corrida durante atualizações da disponibilidade de pontos de recarga (Python Software Foundation, 2023).

3. Metodologia

3.1 Arquitetura do Sistema

O sistema foi desenvolvido como uma aplicação distribuída baseada no modelo **cliente-servidor**, utilizando containers Docker para simular múltiplos veículos (clientes) e pontos de recarga. A arquitetura é composta por três módulos principais:

1. **Cliente (cliente.py):**
    - Representa o veículo elétrico, com funcionalidades para:
        - Listar pontos de recarga próximos (com base na localização simulada).
        - Solicitar reserva em um ponto disponível.
        - Liberar o ponto após a recarga.
        - Consultar histórico de recargas.
    - Comunica-se diretamente com a nuvem via **sockets TCP/IP**, seguindo as restrições do problema (sem uso de frameworks de mensagens).
2. **Nuvem (nuvem.py):**
    - Atua como servidor central, gerenciando:
        - **Registro dinâmico** dos pontos de recarga (criados via docker-compose).
        - **Distribuição inteligente** de demandas, priorizando pontos mais próximos e disponíveis.
        - **Comunicação bidirecional** com clientes e pontos de recarga (reservas, liberações).
3. **Ponto de Recarga (ponto_recarga.py):**
    - Responsável por:
        - Controlar seu estado (disponível, reservado, ocupado).
        - Validar reservas e liberações via sockets.
        - Simular localização geográfica baseada no ID do contêiner.

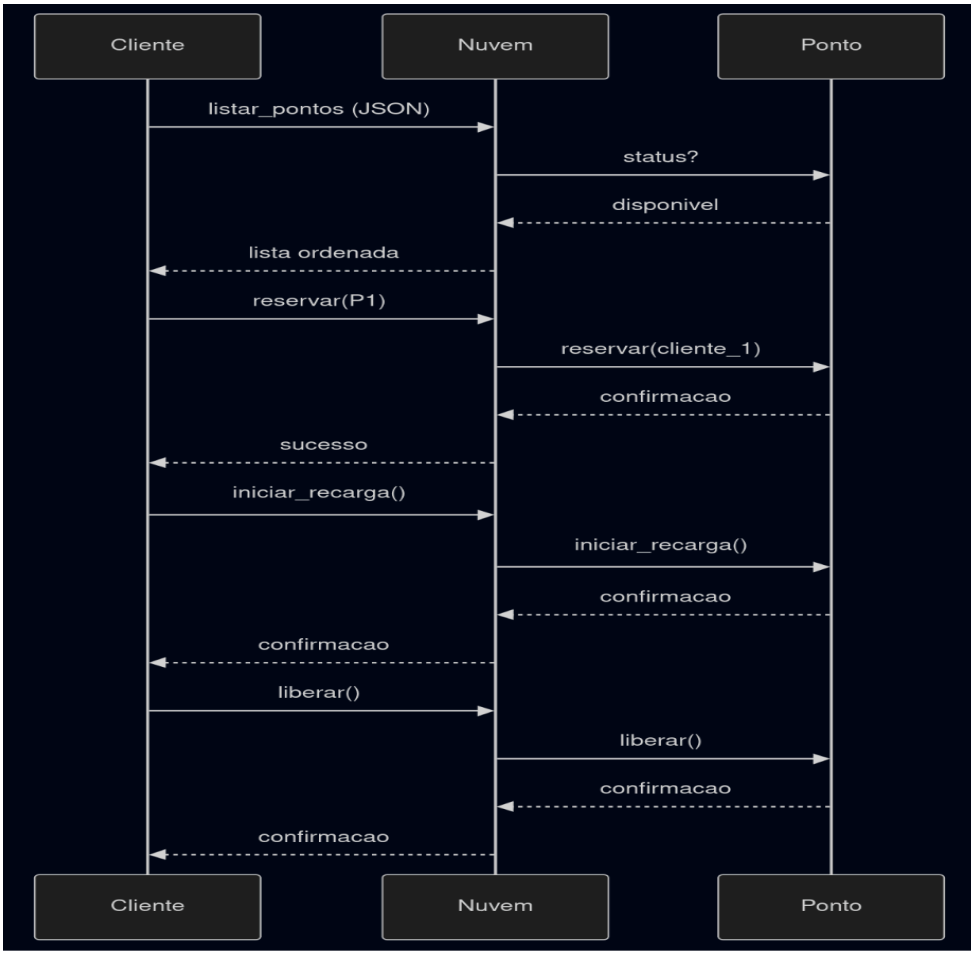

Figura 1. Diagrama de Sequência do projeto.

3.2 Implementação

Tecnologias e Ferramentas

- **Linguagem:** Python 3.9 (com módulos **socket**, **threading**, e **json** para comunicação e concorrência).
- **Contêinerização:** Docker (com **docker-compose** para orquestração).
- **Protocolo:** TCP/IP com sockets brutos (sem bibliotecas de alto nível).

Fluxo de Comunicação

- **Inicialização:**
  - O script **generate_compose.py** gera um arquivo **docker-compose.yml** com N clientes e M pontos de recarga.
  - Cada ponto é mapeado para uma única porta (ex.: **6001** para **Ponto 1**).
- **Reserva de Ponto:**
  - O cliente envia uma mensagem JSON (figura 2) à nuvem com sua localização e ação **solicitar_reserva**.

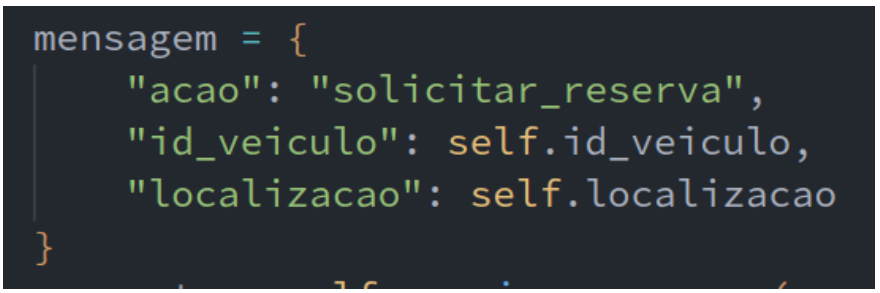

Figura 2. JSON da solicitação de reserva de pontos

    - A nuvem calcula os pontos mais próximos e tenta reservar o primeiro disponível, conectando-se diretamente ao ponto via socket.
- **Liberação de Ponto:**
    - Após a recarga, o cliente solicita a liberação, e a nuvem atualiza o status do ponto para **disponível**.

Tratamento de Concorrência

- **Threads no Servidor:** A nuvem usa **threading** para lidar com múltiplos clientes simultaneamente.
- **Locks Implícitos:** Os pontos de recarga garantem atomicidade nas operações de reserva/liberação através de verificações de status (ex.: não permitem reservar se já estiverem **ocupados**).

3.3 Testes e Validação

Teste de Estresse

- **Inicialização:**
    - O **docker-compose-stress.yml** instanciacria s com 200 clientes e 400 pontos de recarga.
    - As portas são recebidas de formas dinâmicas.
- **Ambiente:**
    - Os testes foram realizados em um ambiente local com a seguinte configuração de hardware: processador AMD Ryzen 7 1700, 16 GB de memória RAM em dual channel e sistema operacional Windows 10, versão 64 bits.
- **Recursos:**
    - No arquivo **docker-compose-stress.yml**, os recursos das instâncias foram configurados com limitações específicas visando a otimização do sistema. Para o ambiente em nuvem, foram alocados 2 vCPUs e 2 GB de memória RAM. As instâncias dos clientes foram configuradas com 5% de uma vCPU e 64 MB de memória RAM cada, totalizando 200 instâncias. Já para os pontos de recarga, cada instância recebeu 10% de uma vCPU e 128 MB de memória RAM, com um total de 400 instâncias.
- **Estabilidade:**
    - Foi comprovado que com mesmo com uma quantidade elevada de

containers de clientes e pontos de recarga, o sistema se mantém estável e sem erros em sua execução.

4. Resultados e Discussões

Durante o desenvolvimento do projeto e a fase de testes do sistema distribuído de gerenciamento de pontos de recarga, enfrentamos algumas dificuldades relacionadas principalmente à comunicação em rede no ambiente Docker e à geração de IDs dos containers. Inicialmente, havia a ideia de utilizar o ID do container para definir uma localização específica para cada ponto de recarga. No entanto, esse método se mostrou inviável, pois, a cada nova criação de imagens e containers, os IDs eram gerados de forma aleatória e diferente da vez anterior. Isso impossibilitou o uso confiável desses IDs como identificadores fixos de localização.

Diante disso, optamos por abandonar essa abordagem e focar em uma solução mais estável. A principal diferença percebida foi que, ao contrário da execução local via localhost, os containers Docker não se comunicam diretamente usando IPs locais. Em ambientes Docker, a comunicação entre serviços deve ser feita através dos nomes dos serviços definidos no arquivo docker-compose.yml. Isso foi essencial para garantir o funcionamento correto do sistema distribuído.

Para superar esses desafios, foi implementada uma lógica de verificação do ambiente de execução por meio da variável de ambiente DOCKER_ENV. Com isso, o sistema passou a adaptar dinamicamente os endereços utilizados na comunicação, garantindo compatibilidade tanto em testes locais quanto no ambiente dockerizado.

Além disso, a utilização de logs em tempo real diretamente no terminal dos containers permitiu acompanhar de forma precisa toda a comunicação entre os módulos. Isso facilitou bastante a depuração e a validação das mensagens trocadas entre o cliente, a nuvem e os pontos de recarga.

Os testes realizados demonstraram que:

- A comunicação cliente-servidor via sockets TCP funcionou corretamente;
- A lógica de reserva, liberação e controle dos pontos de recarga atendeu aos requisitos propostos;
- A nuvem foi capaz de receber solicitações, responder com a lista de pontos de recarga próximos e encaminhar comandos adequadamente;
- O uso de scripts em Python para a geração automática do arquivo docker-compose.yml se mostrou eficaz e escalável, facilitando a realização de diversos testes com múltiplos containers.

5. Conclusão

Esse projeto mostrou que é possível criar uma solução funcional e inteligente para ajudar na organização dos pontos de recarga de veículos elétricos, mesmo com recursos simples como Python, sockets e Docker. A ideia principal era fazer com que os veículos conseguissem encontrar pontos de recarga disponíveis e reservar automaticamente, tudo em um ambiente simulado com containers.

Durante o desenvolvimento, enfrentamos alguns perrengues, principalmente com a comunicação entre os containers e a tentativa de usar o ID dos containers como localização – que não deu certo porque esses IDs mudam toda hora. Mas conseguimos

contornar isso usando os nomes dos serviços definidos no docker-compose.yml e uma variável de ambiente (DOCKER_ENV) para adaptar o sistema de acordo com o ambiente de execução.
Os testes mostraram que a comunicação funcionou bem, a lógica de reserva e liberação deu conta do recado, e o sistema se manteve estável mesmo com centenas de containers rodando ao mesmo tempo. Além disso, os scripts para gerar os arquivos de orquestração facilitaram bastante a criação de cenários variados para testes.
No fim das contas, o projeto cumpriu seu objetivo e ainda mostrou um caminho promissor para evoluir no futuro, com possibilidades como adicionar pagamentos, autenticação ou até usar dados reais de localização. Foi uma experiência desafiadora, mas muito enriquecedora para entender na prática como sistemas distribuídos funcionam e como podem ser aplicados a problemas do mundo real.

Referências

Coulouris, G. F., Dollimore, J., and Kindberg, T. (2005). *Distributed systems: concepts and design*. pearson education.

de Oliveira, R. L. S., Shinoda, A. A., Schweitzer, C. M., and Prete, L. R. (2014). Using mininet for emulation and prototyping software-defined networks. In *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*, pages 1–6. IEEE.

Mininet. (2017). *Home page*. Available: https://www.mininet.org. Accessed: 20 Feb. 2017.

Fromer, C. (2025). Com vendas de junho, mercado bate em 80 mil eletrificados no 1º semestre e 300 mil em circulação – ABVE. *ABVE*. Available: https://abve.org.br/80-mil-eletrificados-so-no-primeiro-semestre/. Accessed: 7 Apr. 2025.