Recarga de carros elétricos inteligente

[1] **Elmer Carvalho de Oliveira Filho,** [1] **Rodrigo Reis Lucena Nazareth,** [1] **Tarcio Passos Freitas**

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

{elmercarvalhofilho,adianello02.rodrigo,T.passos.2017.2}@gmail.com

Resumo. *A frota de carros elétricos tem crescido a cada ano no Brasil, mas ainda falta infraestrutura de carregamento adequada, dificultando a popularização e o uso dos veículos. Este artigo apresenta o EletroRota, um aplicativo desenvolvido para auxiliar motoristas a encontrar os postos de carregamento mais próximos, visando a praticidade no uso de carros elétricos no Brasil. Como resultado foi obtido um aplicativo capaz de indicar o posto de carregamento mais próximo do motorista, facilitando o uso dos veículos elétricos no país.*

1. Introdução

A crescente preocupação com a sustentabilidade e a busca por alternativas aos combustíveis fósseis têm impulsionado o uso de veículos elétricos em todo o mundo. Nesse contexto, a eficiência da infraestrutura de recarga torna-se um fator crítico para a adoção em larga escala dessa tecnologia. Com o objetivo de incentivar e facilitar o uso de carros elétricos, esse sistema foi criado focado no desenvolvimento de uma solução para esse importante mercado.

O primeiro desafio enfrentado pela equipe é a criação de um sistema inteligente. A proposta é implementar um protótipo inicial utilizando uma arquitetura cliente-servidor, permitindo que os motoristas sejam auxiliados via Internet. Por meio de um aplicativo instalado no veículo, o sistema será capaz de identificar o níveisl de bateria e indicar ao condutor os pontos de recarga disponíveis, levando em consideração fatores como a distância e a ocupação de cada estação.

Além disso, o sistema pretende otimizar a distribuição de veículos entre os pontos de recarga, minimizando o tempo de espera dos usuários. Funcionalidades adicionais incluem a possibilidade de reservar um ponto de recarga antes da chegada, liberação automática após o término do carregamento e o gerenciamento das cobranças por meio de contas de usuário, permitindo pagamentos eletrônicos. Este trabalho descreve a arquitetura proposta, os principais desafios enfrentados e os resultados obtidos na construção do protótipo inicial do sistema.

O restante deste trabalho esta organizado da seguinte forma. A seção 2 apresenta a fundamentação teorica das bibliotecas e conceitos mais importantes utilizados no projeto. A Seção 3 discute as metodologias que foram utilizadas na implementação dos conceitos citados na seção 2. A seção 4 apresenta e avalia os resultados. No final, na Seção 5, as conclusões e reflexões sobre os conhecimentos adquiridos.

2. Fundamentação Teórica

O sistema utiliza além da linguagem python e json, algumas bibliotecas e ferramentas com intuitos específicos, apenas os conceitos mais importantes serão abordadas aqui.

2.1. Biblioteca socket

A biblioteca socket do python é utilizada para a comunicação entre o cliente e o servidor. Com essa biblioteca é possível tornar as atualizações de informação mais eficiente, pois ela atualiza as informações para o cliente somente quando for necessário, reduzindo muito as requisições por parte dos clientes e por consequência tornando o sistema mais eficiente. Os selectors são usados para auxiliar o gerenciamento de sockets, arquivos ou pipes. O funcionamento consiste em aguardar um sinal enviado de um socket para iniciar algum evento. Também podem funcionar com vários sockets ao mesmo tempo. Isso será útil para atender varios clientes ao mesmo tempo de forma mais eficiente.

2.2. Biblioteca datetime

O datetime é um módulo do python que facilita o trabalho com as datas e horas e possibilita manipular os dados da data e hora de várias formas, inclusive extrair as informacões delas pra usar em outro lugar do sistema.

2.3. Biblioteca OS

O módulo OS permite interagir com o sistema operacional e está relacionado ao acesso de pastas e arquivos, tanto para criação como com deleção ou acesso aos arquivos. Aqui ele é importante pois o sistema utiliza de arquivos json e cvs para o armazenamento dos dados de clientes.

2.4. Bootstrap

No python é uma técnica utilizada para lhe dar com dados. Ela funciona da seguinte forma: pega um tipo de dado e cria amostras dela com informações diferentes, ou como é o caso do sistema, usa isso para popular o banco de dados com as informações de novos clientes. Ainda existem outras possibilidades de uso do bootstrap como a análise de dados, estatísticas e outras coisas, mas isso não interessa aqui.

2.5. Biblioteca sys

É uma biblioteca utilizada para trabalhar com dados obtidos durante a execução atual de um script, como capturar os dados do caminho de execução do arquivo ou alguma variável de ambiente, ou seja, interagir com o ambiente onde o Python está rodando.

2.6. Conexão stateless

É um modelo de conexão que trata de cada requisição separadamente sem precisar guardar informações sobre elas. O funcionamento dela é simplesmente o servidor capturar os dados enviados pelo cliente em cada requisição, e somente se o cliente enviar uma requisição, e processar a resposta sem precisar armazenar qualquer informação posteriormente. É utilizada nesse sistema principalmente para entregar mais eficiência ao lidar com múlmtiplos clientes ao mesmo tempo.

2.7. Biblioteca threading

A biblioteca threading possibilita executar várias tarefas ao mesmo tempo no programa, ou seja, possibilita a programação multithread. Cada thread é como um mini-processo dentro do sistema, que pode rodar uma função separadamente de forma concorrente.

2.8. Bridge

No contexto do sistema, a bridge em Node.js é um componente intermediário que serve para traduzir a comunicação entre dois mundos diferentes:o o frontend, que se comunica via WebSocket e o backend em Python, que entende apenas conexões TCP/IP padrão. A bridge é basicamente quem recebe mensagens WebSocket do navegador, converte para o formato correto e envia para o backend Python via TCP. E vice-versa.

2.9. Google Maps API

A Google Maps API é um conjunto de serviços criado pela Google que permite que desenvolvedores adicionem mapas, rotas, localização, informações geográficas e outras coisas. É utilizada no sistema principalmente pela exibição dos mapas e rotas.

3. Metodologia

O desenvolvimento do Sistema de Gerenciamento de Carregamento de Carros Elétricos foi conduzido com base em teorias e tecnologias de redes de computadores, concorrência e persistência de dados, descritas na Seção 2. O objetivo principal foi criar uma solução que simulasse a interação de motoristas de veículos elétricos com postos de recarga, utilizando um servidor TCP assíncrono para processar requisições de múltiplos clientes de forma eficiente e confiável. Esta seção detalha como o problema foi abordado, as características da implementação e os testes realizados para validar a solução.

3.1. Metodologia e Aplicação das Tecnologias

O problema foi solucionado utilizando o protocolo TCP para comunicação cliente-servidor, multiplexação assíncrona com a biblioteca selectors do Python para concorrência, e persistência em arquivos JSON/CSV para armazenamento de dados. A abordagem stateless foi adotada para simplificar o design e garantir que cada requisição pudesse ser processada independentemente, utilizando apenas os dados enviados pelo cliente e os arquivos locais. A seguir, descrevemos como essas tecnologias foram aplicadas:

Protocolo TCP: Escolhido por sua confiabilidade na entrega de mensagens, essencial para garantir que requisições como LOGIN ou PAYMENT fossem processadas sem perda de dados. O uso de sockets padrão do Python (módulo socket) permitiu uma implementação leve e portátil, adequada ao escopo de simulação. Multiplexação com selectors: Para suportar múltiplas conexões simultâneas sem o overhead de threads individuais por cliente, foi utilizada a multiplexação assíncrona. O módulo selectors monitora eventos de leitura e escrita em sockets registrados, permitindo que o servidor processe requisições em um único thread principal. Isso foi fundamental para escalar a simulação para até 150 clientes pré-populados. Persistência em Arquivos: Dados de carros, usuários e postos foram armazenados em arquivos JSON no diretório server/data/, enquanto os postos iniciais foram extraídos de um arquivo CSV (feira-de-santana-stations.csv). A escolha por arquivos em vez de um banco de dados relacional reflete a simplicidade requerida pela simulação, com threading.Lock garantindo concorrência segura em operações de leitura/escrita. Monitoramento Dinâmico: Uma thread separada, implementada em station-monitor.py, verifica continuamente os tempos estimados de carregamento (estimated-timestamp) dos veículos nos postos, removendo-os e liberando vagas quando concluídos. Essa abordagem assegura que o sistema reflita um ambiente realista e dinâmico.

3.2. Implementação

A implementação foi estruturada em módulos para promover modularidade e manutenibilidade. O fluxo principal do sistema é descrito a seguir.

O ponto de entrada é o servidor, implementado no arquivo server.py por meio da classe Server. Essa classe gerencia conexões TCP, registrando o socket principal com selectors para aceitar novas conexões e processar dados de clientes existentes em um loop assíncrono. Cada mensagem recebida é decodificada como JSON e enviada ao roteador no arquivo controller.py. O roteador centraliza o despacho das requisições para handlers específicos com base no campo type do JSON, como START, LOGIN, NAVIGATION, SELECTION-STATION e PAYMENT, aplicando validações iniciais para garantir a presença dos campos obrigatórios.

Cada tipo de requisição é tratado por uma classe dedicada nos arquivos do diretório handlers/:

StartManager (start.py) retorna dados iniciais, como modelos de carros e informações sobre postos disponíveis. AuthManager (auth.py) gera um user-id único para cada usuário e cria arquivos correspondentes em server/data/users/. TripManager (trip.py) calcula a viabilidade de rotas com base na autonomia atual do veículo. StationManager (station.py) seleciona o melhor posto com base em distância e tempo, além de processar pagamentos, atualizando os arquivos de postos em server/data/stations/. Paralelamente, o monitoramento é executado em uma thread separada no arquivo station-monitor.py, iniciada no server.py. Essa thread verifica os arquivos de postos, remove veículos cujo tempo de carregamento foi concluído, incrementa o número de vagas disponíveis e deleta os arquivos de usuários correspondentes em server/data/users/. O modelo de dados é definido no arquivo electric-car.py, que contém a classe ElectricCar para cálculos de autonomia, tempos de viagem e tempos de carregamento.

O protocolo de comunicação utiliza mensagens JSON com os campos type, data, status e timestamp, sempre terminadas por . Por exemplo, uma requisição LOGIN cria um usuário e retorna um user-id, que é usado em requisições subsequentes como NAVIGATION ou SELECTION-STATION. A arquitetura é composta por um servidor central que interage com clientes via TCP, roteia requisições para handlers e mantém os dados consistentes por meio de arquivos e locks, enquanto o monitoramento atualiza o estado dos postos em tempo real.

3.3. Testes

A solução foi testada em duas frentes: testes unitários automatizados e simulação em larga escala.

Os testes unitários foram implementados no script controller.py, que inclui um conjunto de requisições de teste cobrindo todas as funcionalidades principais. O teste de START verifica o retorno de dados iniciais, como modelos de carros e postos. O teste de LOGIN confirma a geração de um user-id único e a criação do arquivo de usuário correspondente. O teste de NAVIGATION avalia o cálculo de autonomia para uma rota de 250 km com um Tesla Model 3 (60 kWh). O teste de SELECTION-STATION valida a escolha de um posto a 50 km com vagas disponíveis. Por fim, o teste de PAYMENT verifica a reserva de uma vaga quando confirmation=True e a deleção do usuário quando

confirmation=False. Esses testes foram executados sequencialmente, com as saídas em JSON impressas no console e validadas manualmente. A execução foi realizada com o comando python server/controller.py.

Para a simulação em larga escala, o script bootstrap.py gerou dados sintéticos para 150 clientes e postos, utilizando informações baseadas no arquivo feira-de-santana-stations.csv. O servidor foi iniciado com o comando python server/server.py, e a thread de monitoramento foi testada ajustando manualmente o campo estimated-timestamp em arquivos de postos para valores passados, como 1600000000. Após 10 segundos, o sistema liberou vagas e excluiu os arquivos de usuários correspondentes, confirmando o funcionamento dinâmico.

Testes adicionais foram realizados com um cliente TCP real utilizando netcat. O comando echo ’"type": "LOGIN", "data": "user-name": "joao", "selected-car": "Tesla Model 3", "battery-car": 60, "status": "code": 200, "message": "Sucesso

, "timestamp": "2025-04-07T13:45:12Z

’ | nc 127.0.0.1 8888 enviou uma requisição de login, e a resposta foi validada, seguida por requisições subsequentes. O ambiente Docker também foi testado em ambos os modos: docker-compose up –build server para execução com dados pré-existentes e docker-compose up –build server-with-bootstrap para inicialização completa com o bootstrap.py.

Os resultados dos testes demonstraram que todas as requisições retornam status 200 quando bem-sucedidas, os arquivos de usuários são criados e deletados corretamente, os postos são atualizados dinamicamente com o incremento de available-slots após a conclusão de carregamentos, e o sistema suporta múltiplas conexões sem falhas, escalando até os 150 clientes simulados.

3.4. Dados Utilizados

Os dados reais foram obtidos do arquivo feira-de-santana-stations.csv, baseado em localizações reais de Feira de Santana, adaptado para incluir apenas endereços e capacidade de vagas (entre 3 e 5 slots por posto). Os dados sintéticos foram gerados pelo bootstrap.py, criando 150 clientes com nomes aleatórios, carros extraídos de car-models.json (como Tesla Model 3 e Nissan Leaf) e níveis de bateria variando entre 50

3.5. Frontend

O frontend é a parte mais estética do sistema, porém vale apena citar como funciona a comunicação dele com o backend. A interface web foi desenvolvida com ReactJS e TailwindCSS, integrando a API do Google Maps para exibição de mapas e rotas. O frontend é responsável por capturar a localização atual do veículo, exibir postos de recarga disponíveis em tempo real, calcular rotas até o posto selecionado e gerenciar o estado da carga da bateria e reservas de pontos. O maior desafio seria a comunicação entre o frontend (web/mobile) e o backend em Python através de sockets nativos TCP/IP pois é incomum em aplicações web modernas. Para contornar as limitações dos navegadores (que não permitem conexão direta via TCP), foi implementado o componente Bridge, que já foi explicado na seção 2.

4. Resultados e Discussões

A implementação resolveu o problema proposto ao criar um sistema realista de gerenciamento de carregamento de carros elétricos. A multiplexação com selectors e a thread de monitoramento garantiram eficiência e dinamismo, enquanto a persistência em arquivos simplificou o design. Os testes unitários, em larga escala e com cliente real validaram a robustez da solução, confirmando sua eficácia para a simulação proposta.

5. Conclusão

A realização deste trabalho proporcionou um aprendizado significativo sobre os conceitos de redes de computadores, concorrência, comunicação assíncrona, manipulação de dados e integração de sistemas heterogêneos. A construção deste sistema para veículos elétricos demonstrou, a importância da escolha de arquiteturas eficientes para lidar com múltiplos clientes de forma escalável. Quanto ao problema proposto, foi interessante de se trabalhar, pois está alinhado com uma tendência de mudança mundial para os veículos elétricos e sustentabilidade, além de permitir a aplicação de conceitos novos para a equipe. O problema foi uma excelente oportunidade de expandir os conhecimentos em sistemas distribuídos, redes de computadores e integração de tecnologias.

Referências

Python Software Foundation. Python Language Reference, version 3.10. Disponível em: https://docs.python.org/3/. Acesso em: 05 abr. 2025.

R. W. Stevens. UNIX Network Programming, Volume 1: The Sockets Networking API. Prentice Hall, 3ª edição, 2003.

Google Developers. Google Maps Platform Documentation. Disponível em: https://developers.google.com/maps/documentation. Acesso em: 05 abr. 2025.

G. Van Rossum, F. L. Drake. The Python Language Reference Manual. Network Theory Ltd., 2011.

Bootstrap. Bootstrap Documentation. Disponível em: https://getbootstrap.com/docs/. Acesso em: 05 abr. 2025.

Node.js Foundation. Node.js Documentation. Disponível em: https://nodejs.org/en/docs/. Acesso em: 05 abr. 2025.

D. E. Comer. Internetworking with TCP/IP: Principles, Protocols, and Architecture. Pearson, 6ª edição, 2013.

D. Beazley. Python Cookbook: Recipes for Mastering Python 3. O'Reilly Media, 3ª edição, 2013.

M. J. Quinn. Parallel Programming in C with MPI and OpenMP. McGraw-Hill, 2003.

B. W. Kernighan, R. Pike. The Practice of Programming. Addison-Wesley, 1999.