Pbl 1: Recarga de carros elétricos inteligente

José Victor de Oliveira Correia, João Victor Macedo dos Santos Lima, Tiago de Figueiredo Moura

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

{josevictoroliveira160, tiagodefmoura}@gmail.com , joao.macedo.lima@hotmail.com

Resumo: Problema proposto com o objetivo de desenvolver um sistema de recarga para carros elétricos. Através do enriquecimento da base de conhecimentos utilizando comunicação socket TCP/IP e da linguagem python, a comunicação entre as entidades Nuvem, veículo e postos foi desenvolvida com o objetivo de sanar essa requisição.

1. Introdução

Através do problema “Problema 1 : Recarga de carros elétricos inteligente ”, foi proposto o desenvolvimento de um sistema capaz de lidar com a comunicação entre servidor (Nuvem) e clientes (Posto e Veículo), que permita o processo de recarga desses veículos de acordo com a disponibilidade dos postos de recarga.

Para isso, diversos conceitos, conteúdos e ferramentas foram utilizadas para o desenvolvimento desse projeto, entre estes: Linguagem Python, comunicação através do protocolo TCP/IP e o uso de Docker e suas ferramentas para simular diferentes sistemas sendo executados simultaneamente.

2. Fundamentação Teórica

Para o desenvolvimento desse sistema de recarga de veículos elétricos, foram utilizados fundamentos e ferramentas pontuais, segue abaixo a listagem destas:

- **Linguagem Python**

  A linguagem escolhida para o desenvolvimento do sistema foi o Python, característica por ser simples ,facilmente legível, por suportar a execuções em multiparadigma e utilizando POO, rica em bibliotecas nativas para o desenvolvimento de lógicas complexas e suporte a comunicação socket TCP/IP.

- **Transmission Control Protocol (TCP), TCP/IP e socket**

  Protocolo que é orientado à conexão e garante a entrega dos dados na ordem certa, sem perdas, e mediante o controle de fluxo e verificações Capaz de formar redes (locais ou não) para a troca de informações e garantindo a comunicação socket (canal bidirecional entre dois computadores) e no modelo cliente-servidor.

- **POO**

  Utilização de programação orientada a objetos para uma aproximação com a realidade, formato entidades específicas e com as suas devidas características e funções.

- **Docker**

  Plataforma nativa do Linux que tem como objetivo criar, empacotar e executar aplicações em containers simultaneamente, simulando dispositivos e sistemas próprios. Utiliza os moldes imagem e compose para realizar tal estruturação.

3. Metodologia, Implementação e Testes

O desenvolvimento do projeto foi executado no ambiente Laboratório de Redes e Sistemas Distribuídos - LARSID, da Instituição UEFS. Além disso, alguns momentos de teste e desenvolvimento foram realizados em momentos assíncronos e virtualmente.

Testes foram realizados nas máquinas com OS Linux, disponíveis no LARSID e também nas máquinas pessoais dos integrantes (Linux e Windows através de WSL). Dentre esses testes, os mais frequentes eram feitos para estabelecer a comunicação socket TCP/IP entre os containers desenvolvidos, em especial a Nuvem.

4. Arquiteturas

Arquitetura do Veículo:

O sistema e conexão cliente do contêiner “Veículo”, implementado no arquivo “main.py” através do “socket” em Python, é responsável simular o sistema de um veículo elétrico, sua conta de usuário com as informações necessárias e estabelecer comunicação com o servidor “Nuvem” quando necessário.

- **Estrutura de arquivos do Veículo:**

1. **main.py (main Vehicle):**

   É a classe de início do sistema, nela há a estruturação de uma interface de interação amigável ao usuário. Através do conceito POO, os processos foram divididos em classes e os devidos objetos responsáveis pela lógica de comunicação, pela formatação e exibição de dados, preenchimento e geração de informações e o fluxo geral do sistema são formados nesse arquivo.

2. **User.py**

   Classe responsável por guardar as informações do proprietário do veículo. As

variáveis que a compõem são:

- cpf : CPF do proprietário;
- name: Nome do proprietário;
- email: Email do proprietário;
- password: Senha de acesso do proprietário no sistema.

Com exceção da variável “name”, as outras 3 são utilizadas para realizar o processo de login de usuário na “main.py”, onde a informação login pode ser identificada pelo cpf ou email e a senha por password, respectivamente.

3. **Vehicle.py**

É a classe responsável por estruturar as informações do veículo. As variáveis e métodos existentes são:

Variáveis:

- **vid:** ID do veículo;
- **owner:** Proprietário do veículo;
- **licensePlate:** Número de placa do veículo;
- **moneyCredit:** Crédito do veículo;
- **currentEnergy:** Energia atual do veículo;
- **criticalEnergy:** Porcentagem de energia definida como crítica;
- **distanceFromDestination**: Distância até o trajeto do veículo;
- **distanceFromChargingStation:** Distância até o posto mais próximo;
- **maximumBattery** : Capacidade máxima da bateria do veículo;
- **coordinates**: Lista para guardar as coordenadas do veículo;
- **reservations:** Lista responsável por guardar as reservas realizadas entre o posto e veículo.

Métodos:

- **definePosition**: Responsável por guardar as coordenadas na lista **“coordinates”**;
- **archiveReservation**: Responsável por guardar as reservas;
- **showReservation:** Responsável por exibir todas as reservas feitas com o tempo.

4. **VehicleUtility.py**

Classe responsável por ter métodos de utilidade para o sistema, os seus métodos são:

- **defineCoordinates:** Gera 2 coordenadas aleatórias para o veículo;
- **simulation:** Simula o processo de gasto de bateria e pedido de reserva para a nuvem;
- **clearTerminal:** Realiza a limpeza do terminal, com o objetivo de deixar o fluxo de exibição mais fluido e menos poluído;
- **endAnimation:** Gera uma pequena animação de encerramento do sistema.

5. **VehicleClient.py**

   Classe responsável pela comunicação entre cliente Veículo e servidor Nuvem. Os métodos e veículos são:

   Variáveis:

   - **server_host:** Host do server Nuvem
   - **server_port:** int: Porta do server Nuvem

   Métodos:

   - **sendRequest:** Responsável por estabelecer a comunicação socket TCP/IP, enviar e receber as informações pertinentes através de arquivos JSON.

6. **Dockerfile e docker-compose.yml:**

   Dockerfile é o arquivo utilizado como para estruturar a imagem do container do sistema e o **docker-compose.yml** é utilizado para estruturar um roteiro de execução desse container

   - A **imagem base** utilizada foi "**python:3.13.2**";
   - Os arquivos para gerar a imagem são copiadas do diretório "**src/Vehicle**";
   - O arquivo principal a ser executado é "**main.py**";
   - O **nome do container** do servidor será "**vehicle_client**";
   - A variável de ambiente **“SERVER_HOST=cloud”** e **“SERVER_PORT=64352”** foram definidas para **importação e estabelecer a comunicação com o servidor;**
   - A **rede** ao qual o cliente estará conectado é **"recharging_manager"**;

- **“stdin_open”** e **“tty”** são definidos como **true** para permitir a execução do sistema de modo **interativo** (-it).

Para executar o docker-compose.yml é preciso executá-lo com o seguinte roteiro:

1. **docker network create recharging_manager**
2. **docker compose run --rm vehicle**

obs:

- Caso a network já esteja estabelecida, o passo 1 pode ser ignorado;
- O passo 2 não pode ser substituído por **docker compose up –build** pois, o container **vehicle_client** depende do modo interativo para funcionar corretamente, e o passo citado acima não permite esse modo;
- Tenha certeza que os outros containers (Posto e Nuvem) estejam sendo executados anteriormente para que a comunicação e a troca de informações ocorra corretamente;
- O passo 2 deve ser executado diretamente do diretório onde se encontra.

Arquitetura do Sistema da Nuvem:

O sistema e servidor do container “Nuvem”, implementado no arquivo “Cloud.py” através do “socket” em Python, é responsável por receber requisições dos clientes (veículos ou postos de recarga), armazenar informações sobre postos disponíveis e seus pontos de carregamento e reservas em um banco de dados em arquivo JSON, e processar ações como: criar ou apagar uma reserva, cadastrar ou excluir um posto, descobrir o posto mais próximo do veículo, através de um plano cartesiano, entre outros.
O servidor utiliza multithreading para lidar com a concorrência entre múltiplas conexões simultâneas, evitando que o socket deixe outras requisições esperando, enquanto uma já está sendo atendida. Dessa maneira, cada nova requisição feita ao servidor pelo cliente, gera um novo thread para atendê-la.
O multithreading foi escolhido pois é mais simples de implementar e suficiente para o sistema.
O servidor aceita requisições de qualquer dispositivo na rede local, mas primeiro é necessário criar uma rede personalizada no Docker através do seguinte comando:

“docker network create recharging_manager”

E então, conectar o cliente ao servidor através dessa rede e do host “cloud_server” na porta “64352”.
A imagem do servidor pode ser executada através do docker-compose, acessando o diretório “app/Cloud” e executando o seguinte comando através do terminal:

“docker compose up --build"

- **Estrutura dos Arquivos da Nuvem:**

1. Cloud.py:

Este é o servidor principal do sistema. Ele é responsável por iniciar o socket da Nuvem, ativar o modo de escuta do socket, criar threads para novas solicitações dos clientes, fazer a manipulação de dados necessárias no banco de dados, responder aos clientes e finalizar a conexão com os clientes.
O servidor escuta qualquer dispositivo conectado na mesma rede local, que se comunica com a porta "64352".
Ao receber uma requisição, interpreta o conteúdo em JSON, analisa qual ação é necessária, através de uma chave/campo de ação para cada solicitação do cliente, e executa a ação correspondente.

Campos/Chaves no JSON enviado pelo Cliente e suas ações no servidor:

- **newChargingStation:**
  - O cliente do posto de recarga solicita a criação e salvamento de suas informações no banco de dados do servidor.
  - O servidor responde com o ID cadastrado para o posto de recarga no banco de dados.
- **updateChargingStation:**
  - O cliente do posto de recarga solicita a atualização de sua localização no banco de dados do servidor.
  - o servidor responde com a string: "Sucesso".
- **deleteChargingStation:**
  - O cliente do posto de recarga solicita a sua exclusão do banco de dados do servidor.
  - o servidor responde com a string: "Sucesso".
- **newChargingPoint:**
  - O cliente do posto de recarga solicita a criação e salvamento de um dos seus pontos de carregamento no banco de dados do servidor.
  - O servidor responde com o ID cadastrado para o ponto de carregamento no banco de dados.
- **updateChargingPoint:**
  - O cliente do posto de recarga solicita a atualização das informações de um dos seus pontos de carregamento no banco de dados do servidor.

    - O servidor responde com a string: "Sucesso".
- **deleteChargingPoint:**
    - O cliente do posto de recarga solicita a exclusão de um dos seus pontos de carregamento no banco de dados do servidor.
    - o servidor responde com a string: "Sucesso".
- **receiveAllChargingPoints:**
    - O cliente do posto de recarga solicita uma lista de todos os seus pontos de carregamento e suas respectivas informações do banco de dados.
    - O servidor responde com um JSON com todos os pontos de carregamento.
- **receiveAllReservations:**
    - O cliente do posto de recarga solicita uma lista com informações de todas as reservas, para todos os seus pontos de carregamento, no banco de dados.
    - O servidor responde com um JSON com todas as reservas agendadas.
- **scheduleReservation:**
    - O cliente do veículo solicita o agendamento de uma reserva, pois a bateria está no nível crítico.
    - O servidor responde com as informações do agendamento para um ponto de carregamento com a fila vazia, ou com o menor tempo de espera, no posto de recarga mais próximo.
- **findReservation:**
    - O cliente do veículo solicita as informações de sua reserva agendada, se existir alguma.
    - O servidor responde com um JSON com as informações da reserva, armazenados no banco de dados.
- **deleteReservation:**
    - O cliente do veículo solicita a exclusão de uma reserva agendada.
    - o servidor responde com a string: "Sucesso".

Cada ação é tratada por meio de classes auxiliares, que manipulam os dados nas listas e os atualizam no banco de dados em JSON, como: **ReservationsFile** (Para Reservas), **ChargingPointsFile** (Para Pontos de Carregamento) e **ChargingStationsFile** (Para Posto de Recarga).

Além disso, caso ocorra um dos seguintes erros na solicitação, o servidor irá

responder com a string **"None"**:

- Não existem postos de recarga ou pontos de carregamento cadastrados;
- Posto de recarga não encontrado, para atualização ou exclusão;
- Ponto de carregamento não encontrado, para atualização ou exclusão;
- Não existem reservas cadastradas para o posto de recarga ou veículo;
- A reserva não foi encontrada, para exclusão.

2. ChargingStationsFile.py:

Este arquivo contém a classe "**ChargingStationsFile**", responsável por armazenar, manipular e recuperar os dados dos postos de recarga, a partir do arquivo "**charging_stations.json**". Os postos de recargas são identificados por um "**chargingStationID**" único, e cada um deve conter uma localização única nas coordenadas "**x**" e "**y**" do plano cartesiano.

3. ChargingPointsFile.py:

A classe "**ChargingPointsFile**" manipula o arquivo "**charging_points.json**", que contém os pontos de carregamento disponíveis em cada posto de recarga.
Cada ponto é identificado por um "**chargingPointID**" único, e contém as seguintes informações:

- **"chargingStationID"**: a qual posto de recarga ele pertence.
- **"power"**: a potência do carregador em kW.
- **"kWhPrice"**: o preço por kWh.
- **"availability"**: o seu estado de disponibilidade, como "**livre**", "**ocupado**" ou "**reservado**".

4. ReservationsFile.py:

A classe "**ReservationsFile**" interage com "**reservations.json**", armazenando, lendo, criando, editando e excluindo as reservas com as seguintes informações:

- **"reservationID"**: seu ID, deve ser único para o mesmo ponto de carregamento.
- **"chargingStationID"**: o ID do posto de recarga onde ela será realizada.
- **"chargingPointID"**: o ID do ponto de carregamento onde ela será realizada.
- **"chargingPointPower"**: a potência do carregador em kW.
- **"kWhPrice"**: o preço por kWh.
- **"vehicleID"**: o ID do veículo que usará a reserva.
- **"duration"**: o tempo necessário em horas para finalizar a carga completa do veículo.

- **"startDateTime"**: a data e hora de início da reserva.
- **"finishDateTime"**: a data e hora de finalização da reserva, a data de início somada com o tempo necessário para finalizar uma carga completa do veículo.
- **"price"**: o preço da reserva, a ser pago.

5. Informações sobre o Dockerfile e Docker Compose do servidor:

- A **imagem base** utilizada foi "**python:3.13.2**".
- Os arquivos para gerar a imagem são copiadas do diretório "**src/Cloud**".
- O arquivo principal a ser executado é "**Cloud.py**".
- O **nome do container** do servidor será "**cloud_server**".
- A variável de ambiente "**PYTHONUNBUFFERED=1**" foi definida, para permitir a **exibição dos prints no terminal.**
- A **rede** ao qual o servidor estará conectado é **"recharging_manager"**.
- A **porta** do servidor exposta é **"64352"**.

5. Resultados e Discussões

Ao final do desenvolvimento, tais conclusões podem ser feitas: O desenvolvimento dos sistemas Nuvem e Veículo foram realizadas e a comunicação entre eles foi bem sucedida. Entretanto, o sistema posto acabou não sendo finalizado totalmente. Logo, foi utilizada uma classe de teste com template básico de um posto para gera-los automaticamente, sem que a comunicação entre os 3 sistemas fosse prejudicada pela falta do sistema posto.
As funcionalidades principais foram desenvolvidas e atendidas ao final do projeto, a comunicação e troca de pacotes de informações com a Nuvem ocorreu de forma eficiente.

Referências:

VERONEZ, Fabricio. Docker do zero ao compose: Parte 01. YouTube, transmissão ao vivo. Disponível em: https://www.youtube.com/live/GkMJJkWRgBQ. Acesso em: 28 fev. 2025.

VERONEZ, Fabricio. Docker do zero ao compose: Parte 02. YouTube, transmissão ao vivo. Disponível em: https://www.youtube.com/live/KPb_ZqDTDyQ. Acesso em: 5 mar. 2025.

FERREIRA, Rafael. Docker do zero ao compose: Parte 03. Professor Lhama. YouTube, vídeo. Disponível em: https://youtu.be/SxyHmEWtkuM. Acesso em: 10 mar. 2025.

CARVALHO, Glauko. Docker do zero ao compose: Parte 04. YouTube, vídeo. Disponível em: https://youtu.be/kP1kktlbUTs. Acesso em: 18 mar. 2025.