PBL 2 - Gerente de Recarga de Carros Elétricos Distribuídos

José Victor de Oliveira Correia, João Victor Macedo dos Santos Lima

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

josevictoroliveira160@gmail.com , joao.macedo.lima@hotmail.com

Resumo: Problema proposto com o objetivo de desenvolver um sistema de reservas e recarga para carros elétricos que precisam realizar viagens de longa distância. Através do enriquecimento da base de conhecimentos utilizando o protocolo de comunicação MQTT, arquitetura API REST e da linguagem python, a comunicação entre clientes foi desenvolvida com o objetivo de sanar essa requisição.

1. Introdução

Através do problema “Problema 2: Recarga Distribuida de Veículos Elétricos”, foi proposto o desenvolvimento de um sistema capaz de lidar com a comunicação entre clientes (Servidores e Veículos), que permita o processo de reserva e recarga de veículos em viagens de longa distância.

Para isso, diversos conceitos, conteúdos e ferramentas foram utilizadas para o desenvolvimento desse projeto, entre estes: Linguagem Python, comunicação MQTT, arquitetura API REST 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, mesmo que externo para os tipos de comunicações utilizados.

- **API REST**

Conhecido como **Representational State Transfer,** é um protocolo de arquitetura utilizado em comunicações via internet entre clientes e servidores, baseado em recursos acessados por URLs e manipulados com verbo HTTP.

- **MQTT**

Conhecido como **Message Queuing Telemetry Transport** , é um protocolo de comunicação para sistemas, dispositivos e na área da Internet das coisas (IoT) que dependem de comunicação rápida, eficiente e com baixo consumo de energia e largura de banda. Tem como características e conceitos principais:

- **Broker:** Servidor responsável por troca de mensagens entre as entidades na comunicação local. Neste projeto o broker escolhido foi o **Mosquitto,** que tem suporte na linguagem python através da biblioteca externa " **paho-mqtt** ".
- **Cliente**: São as entidades de conexão local. Se baseiam no conceito de publisher/subscriber, onde podem enviar para ou receber mensagens de outras entidades. Neste projeto, tanto veículo como os servidores se comportam como clientes, trocando mensagens entre si através do broker, para alcançar o objetivo do sistema.
- **Tópicos:** São os endereços por onde os clientes se inscrevem para receber ou publicar mensagens entre si, os tópicos são gerenciados pelo broker. No projeto, os tópicos definidos foram:

Publisher:

Tópico	Descrição
"server/create_reservations/vehicle”	Tópico utilizado para criar reservas

Subscriber:

Tópico	Descrição
vehicle/create_reservations/server"	Tópico utilizado para receber as reservas efetuadas

- **M2M**

Conhecido como **Machine to Machine,** é a troca de informações em redes diretamente entre máquinas e sem a influência humana em nenhum ponto. Baseado em outros conceitos como automação, baixo consumo, comunicação em tempo real, é frequentemente atrelado ao protocolo de comunicação MQTT.

- **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 MQTT e via internet baseado no protocolo de arquitetura API REST entre os containers desenvolvidos.

4. Arquiteturas

Arquitetura do Veículo:

O sistema e conexão cliente do contêiner “Veículo”, implementado no arquivo “main.py” através do “mqtt” em Python, é responsável simular o sistema de um veículo elétrico, sua conta de usuário com as informações necessárias e estabelecer comunicação com o Broker MQTT e Servidores 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 utilizados neste arquivo.
Nela, o usuário tem várias opções de interação, primeiramente há as opções de Login, Registro de conta, e nas opções internas de Login há: Fazer reservas, visualizar reserva realizadas, ver informações de conta, adicionar crédito na conta, voltar para o ínicio do sistema (sem a necessidade de fazer login novamente) e por fim, encerrar completamente o programa.

2. **User.py**

Classe responsável por guardar as informações do proprietário do veículo. As variáveis que a compõem são:

Nome	Descrição	Tipo
cpf	CPF do proprietário	str
name	Nome do proprietário	str
email	Email do proprietário	str

password	Senha de acesso proprietário ao sistema	str

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.
Os dados atribuídos a essas variáveis são gerados no arquivo “main.py”, utilizando a biblioteca externa ao python, chamada de **Faker**.

3. Vehicle.py

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

Variáveis:

Nome	Descrição	Tipo
vid	ID do veículo.	str
owner	Proprietário do veículo.	User
licensePlate	Placa do veículo.	str
moneyCredit	Crédito do veículo.	float
currentEnergy	Energia atual do veículo.	int
maximumBattery	Capacidade máxima da bateria do veículo.	int
reservations	Lista de reservas do veículo.	list[dict]
utility	Objeto de métodos utilitários.	VehicleUtility

Métodos:

Nome	Parâmetros	Descrição
showInformations	self	Método para visualização de dados de conta.
showReservations	self	Método para visualizar todas as reservas efetuadas .
savingLoginData	self, dataFilePath(str)	Método para salvar novos dados gerados na opção "2 - CRIAR CONTA".
updateCredit	self, dataFilePath(str), value((float), operation(str)	Método para atualizar o valor de crédito da conta.
loadingData	self, dataFilePath(str), reservationsFilePath(str)	Método para carregar dados de conta (dados gerados e salvos anteriormente).
keepReservations	self,	Método para salvar as novas reservas.

	reservationsFilePath(str), newReservations(list[dict])	

Os dados usados para preencher as variáveis são gerados no arquivo “main.py”, utilizando as bibliotecas **Faker** e **Random**.

4. **VehicleUtility.py**

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

Nome	Parâmetros	Descrição	Retorno
clearTerminal	-	Realiza a limpeza do terminal, com o objetivo de deixar o fluxo de exibição mais fluido e menos poluído.	-
endAnimation	self	Método que cria uma pequena animação de encerramento de programa.	-
defineRoute	self, origin(str), destination(str)	Método responsável por determinar os codinomes das cidades de origem e de destino, que são passados para o servidor, baseado nas respostas do usuário. Se as entradas forem válidas, retorna uma lista com os codinomes e caso alguma das entradas não corresponda com alguma das cidades da rota, o método retorna False e é executado novamente.	list[str] ou False
writeReplyBack	self, wrongActions (bool), repeat (bool)	Método utilizado para voltar para o ínicio ou para encerrar o programa, dentro das opções de login(opções 1, 2, 3 e 4).	-
nomalizeName	self, fakeName(list[str])	Método para normalizar(tratar) a representação de caracteres especiais(ç) ou acentuações(á, é, à, è, õ,ã, ê, ô, â ...) na geração de nomes aleatórios(str).	genericName(list[str])
startAnimation	self	Método de exibição inicial no programa.	-

Na classe VehicleUtility há a importação das seguintes bibliotecas: **time**, **os** e **unicodedata**, para a estruturação da lógica dos métodos correspondentes.

5. **VehicleClient.py**

Classe responsável pela comunicação entre cliente Veículo e Servidores. A comunicação é baseada no protocolo MQTT(usando o broker Mosquitto) e no conceito de M2M. As variáveis e métodos da classe são:

Variáveis:

Nome	Descrição	Tipo
client	Objeto que estabelece a comunicação MQTT e usado para definir os seus parâmetros e métodos de execução.	mqtt.client
serverHOST	Host do server na comunicação MQTT.	str
serverPORT	Porta de rede na comunicação MQTT.	int
cost	Custo de serviço de todas reservas realizadas para a rota.	float
reservations	Novas reservas que serão adicionadas à conta do veículo.	list[dict]
message	Variável usada para receber a resposta do servidor com as reservas. Posteriormente recebe um tipo json.	None
utility	Objeto de métodos utilitários.	VehicleUtility
vData	Dicionário estruturado para enviar as informações necessárias ao servidor ao realizar uma reserva. Ele é convertido em json posteriormente para ser enviado.	dict

Métodos:

Nome	Parâmetros	Descrição	Retorno

on_connect	self, client, userdata, flags, rc	Método de callback definido para ser executado ao estabelecer a conexão MQTT.	-
on_message	self, client, userdata, msg	Método de callback definido para tratar a mensagem/resposta na comunicação MQTT.	-
on_publish	self, client, userdata, mid	Método de callback para quando o client publica algo no tópico correspondente.	-
defineIP	self	Método para obter IP e determinar server baseado em um script. Utilizado para testes do projeto.	-

6. **Dockerfile, docker-compose.yml e requirements.txt :**

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.

Dockerfile:

- A **imagem base** utilizada foi "**python:3.13.2**";
- O arquivo principal a ser executado é "**main.py**";
- Através da definição das tags **COPY requirements.txt .** e **RUN pip install --no-cache-dir -r requirements.txt** , as bibliotecas e dependências sejam instaladas no contêiner (flask, Faker e paho-mqtt).

docker_compose:

- O **nome do container** do cliente é "**vehicle_client**";
- **“stdin_open”** e **“tty”** são definidos como **true** para permitir a execução do sistema de modo **interativo** (-it);
- A **rede** no modo **"network_mode: host"**;

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

➔ **docker compose run --rm vehicle**

obs:

- O comando deve ser executado do diretório "**src/app/Vehicle**"
- 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 (Servidores e Brokers Mosquitto) 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 o arquivo docker-compose.yml.
- Ao encerrar o programa, por meio da tag **--rm** , o contêiner é removido.

7. **Arquivos data.json e reservations.json :**
   Arquivos utilizados para persistência de dados referentes à conta do veículo, o “data.json” é utilizado para salvar os dados referentes às informações tanto do proprietário como do próprio veículo, e o arquivo “reservations.json” é utilizado para salvar todas as reservas efetuadas na respectiva conta do veículo.

Arquitetura de Sistema dos Servidores:

Os servidores são responsáveis por atender aos agendamentos de reservas para usuários de veículos elétricos em diferentes cidades do país. Através de uma requisição atômica, de servidor para servidor, sem centralização das reservas.

Cada servidor pertence a uma empresa específica, que tem seus postos de recarga e que atua em algum estado do Brasil, atualmente: Bahia na empresa “VoltPoint”, Ceará na empresa “E-Flux” e Pernambuco na empresa “EcoCharge”.

A lógica principal de cada servidor foi implementada em seu arquivo “Server.py”, através de uma API REST desenvolvida em Flask e comunicação peer-to-peer através do MQTT.

A comunicação entre os servidores e os seus clientes (veículos) é realizada através de mensagens MQTT em tópicos de inscrição e publicação. Dessa maneira, os servidores atuam como subcribers, que recebem mensagens, e publishers, que publicam mensagens.

Os servidores utilizam multithreading para lidar com a concorrência entre múltiplas conexões simultâneas, e bloqueios do tipo “threading.Lock()” nas funções de agendamento de reservas. Dessa maneira, cada nova requisição feita ao servidor pelo cliente, gera um

novo thread para atendê-la.

Como Usar os Servidores:

Cada servidor tem o seu próprio Broker Mosquitto, que atuará como um intermediário entre as mensagens do cliente e do servidor, e que deve ser iniciado em um container Docker através do arquivo “docker-compose.yml” em “app/mosquitto”.

Antes de iniciar a comunicação com o servidor, ele deve ser alimentado com os dados dos postos de recarga e pontos de carregamento da empresa ao qual ele pertence. Esses dados são armazenandos em um banco de dados nos arquivos “data/charging_stations.json” e “data/charging_points.json”.

Cada servidor possui o seu “docker-compose.yml” dentro da sua respectiva pasta em “app”, que pode ser executado através do comando “docker compose up --build”.

Antes de iniciar os servidores, indique para cada um deles os endereços IP dos outros, editando as variáveis de ambiente nos arquivos “docker-compose.yml” de cada servidor.

Estrutura dos Arquivos dos Servidores:

Server.py:

Arquivo principal, onde está implementada a API REST, através do framework Flask, e a execução do MQTT.

Contém a lógica da rota de criação de reservas, que recebe os dados para reservas através da função “on_message” do MQTT em “mqttFuncions.py”, ou de outros servidores, e realiza as manipulações necessárias para agendamento, como agendamento nos seus próprios postos ou redirecionamento para outros servidores, e o retorno das reservas, para serem enviadas via MQTT.

mqttFunctions.py:

Contém as funções de criação, configuração e inicialização do cliente MQTT, que é executado assincronamente, permitindo a reconexão, caso haja perda de conexão com o Broker Mosquitto. Além do bloqueio dos threads via “threading.Lock()”.

Utils.py:

Contém funções úteis para tratamento de exceções e redirecionamento de reservas para APIs de outros servidores.

RoutesFiles.py:

Responsável por criar, atualizar e excluir rotas, com cidades onde o veículo poderá trafegar, através do arquivo “data/routes.json”.

Possui também uma função que gera uma rota de acordo com a cidade de partida e destino desejada pelo veículo.

ChargingStationsFile.py:

Responsável por armazenar, manipular e recuperar os dados dos postos de recarga do servidor, a partir do arquivo "data/charging_stations.json".

ChargingPointsFile.py:

Lógica semelhante ao arquivo “ChargingStationsFile.py”, porém armazena pontos de carregamento pertencentes a um posto de recarga específico.

ReservationsFile.py:

Um banco de dados de reservas do servidor no arquivo “data/reservations.json”.

ReservationHelper.py:

Responsável por calcular em quais cidades o veículo deve parar para recarregar e salvá-las em uma lista de cidades onde agendar reservas, de acordo com as especificações técnicas do veículo e a rota para trafegar escolhida pela função “findRoute” em “RoutesFiles.py”.

Possui também uma função que escolhe um ponto de carregamento, para cada posto de recarga na rota de reservas, de acordo com a sua disponibilidade e fila.

5. Resultados e Discussões

Ao final do desenvolvimento, tais conclusões podem ser feitas: O desenvolvimento dos sistemas clientes(Servidores e veículos) e broker foram realizadas e a comunicação entre eles foi bem sucedida. Entretanto, alguns requisitos do problema não foram devidamente tratados: A concorrência entre entidades e o tratamento de persistência e exibição das reservas que correspondem aos respectivos veículo

As demais funcionalidades e requisitos principais foram desenvolvidos e atendidos ao final do projeto, a comunicação e troca de pacotes de informações entre as entidades 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.

FERSING, Pierre; LIGHT, Roger. **Paho-mqtt 2.1.0**. [S. l.], 29 abr. 2024. Disponível em: https://pypi.org/project/paho-mqtt/#client. Acesso em: 1 maio 2025.