

Recarga de carros elétricos inteligente

Caick Wendell Lopes dos Santos¹, João Lucas Silva Serafim Silva¹

¹Departamento de Tecnologia – Universidade Estadual de Feira de Santana (UEFS)
Av. Transnordestina, s/n, Novo Horizonte / Feira de Santana – BA, Brasil- 44036-9

Resumo.A popularidade dos carros elétricos vem crescendo continuamente. Entretanto, a infraestrutura de carregamento em território brasileiro acompanha esse aumento de forma irregular e há insatisfação dos usuários devido as longas filas nos postos. Este trabalho apresenta o desenvolvimento de um sistema de recarga inteligente que viabiliza a comunicação entre os veículos e os postos. O sistema utiliza o modelo cliente-servidor através de *sockets*, *threads* e mecanismos de sincronização fornecidos pela linguagem Python em conjunto com tecnologia de *containers*. A aplicação possibilita o registro de veículos e postos, reserva de pontos com base na distância do motorista e visualização do histórico de pagamento.

Abstract.The popularity of electric vehicles has been steadily increasing. However, the charging infrastructure across Brazil has been developing unevenly, leading to user dissatisfaction due to long queues at charging stations. This work presents the development of an intelligent charging system that enables communication between vehicles and charging stations. The system employs a client-server model using sockets, threads, and synchronization mechanisms provided by the Python programming language, combined with container technology. The application allows vehicle and station registration, reservation of charging points based on the driver's distance, and viewing of payment history.

1. Introdução

O investimento em carros elétricos tem aumentado consideravelmente, seus inúmeros benefícios contribuem para o crescimento da demanda e popularização da indústria. Crescimento que, por sua vez, é refletido de forma assimétrica no Brasil. Os pontos de carregamento estão aglomerados nos grandes centros urbanos e suas proximidades, tornando o produto menos convidativo para novos usuários. Além disso, a falta de informação em conjunto com a quantidade insuficiente de estações de recarga ofertadas tem tornado as filas para abastecimento longas e os motoristas insatisfeitos.

O documento a seguir tem por objetivo apresentar os métodos e ferramentas utilizadas para a construção de um sistema de recarga de carros elétricos. O qual propõe-se a incentivar o mercado ao aperfeiçoar a experiência do usuário. A aplicação é feita majoritariamente em linguagem Python e utiliza de *sockets* em uma arquitetura cliente-servidor 2.1 com auxílio de *threads*. Além disso, o sistema conta com a ferramenta Docker para implementação da tecnologia de *containers* 2.2 e foi projetado essencialmente em sistemas Linux através da IDE Visual Studio Code em conjunto com a plataforma GitHub.

O sistema permite o registro de veículos, pontos de carregamento e suas informações essenciais, tendo a maior parte de suas funcionalidades mediadas por um servidor. Os veículos minimamente descarregados são capazes de reservar pontos de carregamento próximos contanto que estejam disponíveis, além de efetuar e visualizar pagamentos. Este relatório está organizado da seguinte forma. A seção 2 refere-se aos conceitos

teóricos aplicados no projeto. A seção 3 apresenta as abordagens e ferramentas que viabilizaram a conclusão do projeto. Na seção 4 encontra-se a descrição dos resultados e falhas obtidos. Por último, a seção 5 condensa as reflexões inferidas ao concluir o projeto.

2. Fundamentação Teórica

Esta seção descreve sucintamente os principais conceitos utilizados para o desenvolvimento do produto em questão. O sistema se beneficia de *containers* em *docker* e é estruturado em volta do modelo cliente-servidor.

2.1. Paradigma Cliente-Servidor

A arquitetura cliente-servidor permite que muitos usuários tenham acesso à mesma base de dados [Oluwatosin 2014]. É uma arquitetura que é definida tanto para o cliente quanto para o servidor, caracteriza-se pela troca de mensagens e parte do princípio que o cliente enviará requisições e o servidor as responderá. Entre seus benefícios pode-se citar:

1. Divisão de processamento da aplicação em múltiplas máquinas
2. Fácil compartilhamento de recursos entre cliente e servidor
3. Redução da replicação de dados

2.2. Containers

Containers são ferramentas de virtualização que operam a nível de sistema operacional [Merkel et al. 2014]. Uma de suas propostas é resolver o problema de "*dependency hell*", onde uma aplicação se apoia em um serviço ou outra aplicação de forma a gerar problemas de dependência.

Containers compartilham do mesmo *kernel* do sistema operacional, contudo, isolam os processos do restante do sistema. Dessa forma, é possível economizar recursos uma vez que não há constante exigência dos mesmos. Outra vantagem é que *containers* podem ser criados e destruídos facilmente conforme a necessidade.

2.2.1. Docker

Docker é um *software* usado para usufruir da tecnologia de *containers* virtuais. Sua arquitetura consiste em quatro componentes principais:

- **Cliente:** Responsável por criar, gerenciar e executar *containers*.
- **Servidor:** Aguarda as solicitações da *API REST* e gerencia imagens e *containers*.
- **Imagem:** Descreve os requisitos para criação de um *container*.
- **Registro:** Hospeda e distribui imagens.

2.3. Protocolo TCP/IP

O *transmission control protocol* (TCP) é um padrão de comunicação que permite que dispositivos troquem mensagens em uma rede. O TCP garante a integridade dos dados ao estabelecer uma conexão entre os dispositivos e ao dividir grandes quantidades de dados em pacotes menores. O *internet protocol* (IP) por sua vez, tem por função fornecer um esquema de endereçamento único e global para cada dispositivo conectado em uma rede.

O protocolo TCP/IP [Comer 2016] foi desenvolvido pelo Departamento de Defesa dos Estados Unidos e é capaz de dividir uma mensagem em 4 camadas (*datalink*, internet, transporte e aplicativo) e remontá-la em seu formato original. Essa característica o torna eficiente e veloz uma vez que é capaz de enviar pacotes por diferentes rotas em caso de congestionamento ou indisponibilidade

2.4. *Socket* de Rede

O *socket* é o ponto final de um fluxo de comunicação, através da rede [Silva et al.]. Ele provê a comunicação entre duas pontas que estejam na mesma máquina ou rede. O *socket* que se comunica por uma rede é um *socket* TCP/IP e é definido por um endereço e uma porta.

3. Metodologia

Inicialmente, fizeram-se necessárias pesquisas e testes com o intuito de entender como utilizar as ferramentas de desenvolvimento propostas para o problema, em especial o software *Docker* (ou, mais especificamente, o motor da aplicação).

A pesquisa revelou que o *Docker* pode ser controlado de forma simples por meio de terminal de comando, contando com 3 tipos principais de objetos a serem controlados: imagens, redes e contêineres. Informações acerca da construção de uma imagem devem ser registradas em arquivo *.Dockerfile*, contido no diretório atualmente acessado pelo terminal.

A screenshot of a terminal window with a dark background. The text 'docker build -t python-redes-image .' is displayed in a light blue or cyan monospaced font. The text is on a single line and appears to be a command being entered or executed.

Figura 1. Comando para realizar a criação de uma imagem de nome "python-redes-img", conforme instruções contidas no arquivo *.Dockerfile*.

O próximo passo consistiu no desenvolvimento de um protótipo de protocolo para comunicação entre dois programas distintos, o qual por sua vez faz uso do já estabelecido e amplamente utilizado protocolo TCP-IP. A linguagem de programação Python possui biblioteca nativa dedicada para fazer uso de soquetes de rede por meio de protocolo TCP-IP.

```
def sendRequest(self, request):

    global serverAddress

    socket_sender = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    socket_sender.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    socket_sender.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)

    serializedRequest = json.dumps(request)

    try:
        socket_sender.connect((serverAddress, 8001))
        socket_sender.send(bytes(serializedRequest, 'UTF-8'))
    except:
        pass

    socket_sender.close()
```

Figura 2. Função em linguagem Python que cria, configura, utiliza e destrói um soquete TCP-IP em um dos lados (remetente) de uma linha de comunicação.

A seguir, foi iniciado o desenvolvimento das aplicações propriamente ditas, sendo que inicialmente o servidor recebia apenas requisições para a criação de veículos e estações, e os veículos e estações tão somente enviavam tais tipos de requisição.

O protocolo de requisições é feito no formato : ENVIO-RECEBIMENTO-RESPOSTA; o cliente (estação ou veículo) envia uma requisição que é recebida e processada pelo servidor, o qual deve então retornar uma resposta, além de registrar informações acerca do que foi enviado como resposta. Caso o cliente não receba resposta, este deve reenviar a requisição, a qual provoca apenas um reenvio da resposta caso seja identificado que é idêntica à requisição anterior enviada pelo mesmo cliente.

```

def registerVehicle(self):

    global requestID

    requestID = 64

    requestContent = [requestID, 'rve', '']

    self.sendRequest(requestContent)

    (add, response) = self.listenToResponse()

    while (len(response) != 24):

        self.sendRequest(requestContent)

        (add, response) = self.listenToResponse()

    if (int(requestID) < 63):
        requestID = str(int(requestID) + 1)
    else:
        requestID = "1"

    return response

```

Figura 3. Função para registro de veículo (parte do cliente) e envia requisições idênticas até que recebe uma resposta adequada.

O penúltimo passo foi implementar as demais funções referentes aos requisitos faltantes para o sistema. Neste momento foi percebida a necessidade da utilização de uma ferramenta de paralelismo para garantir o processamento de múltiplas requisições de forma simultânea. Para tal, foi utilizada a biblioteca de *threading* do Python, tanto para a criação e gerenciamento de *threads* em si, quanto para a chamada de travas do sistema (controle de acesso a zonas críticas).

Por fim, durante a implementação de suas últimas funções, o terminal do veículo recebeu uma interface gráfica (a qual faz uso da biblioteca *Custom TKinter*), e também passou a utilizar ferramentas de *threading* para garantir a atualização de certas informações exibidas na janela da aplicação.

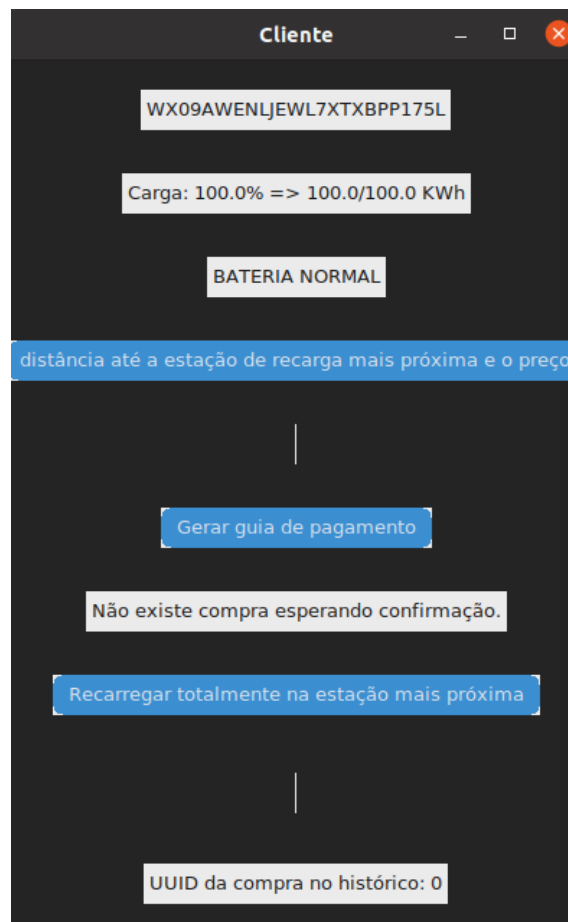


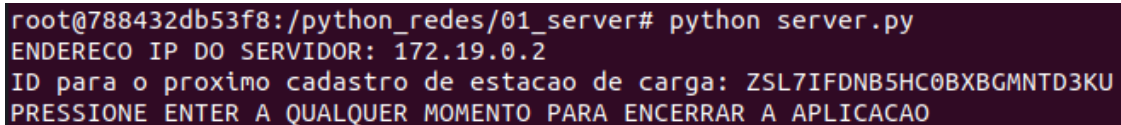
Figura 4. Interface gráfica da aplicação utilizada no veículo (usuário final).

4. Resultados

Esta sessão é dedicada à descrição dos resultados obtidos durante a operação do sistema.

4.1. Inicialização do servidor

Após sua inicialização, o terminal do servidor exibirá informações básicas do servidor: seu endereço IP, o ID para o cadastro da próxima estação de recarga e um *prompt* informando ao usuário do terminal como encerrar a aplicação (pressionando ENTER).

A screenshot of a terminal window with a dark background and light-colored text. The text shows the command to run the server script and the output information.

```
root@788432db53f8:/python_redes/01_server# python server.py
ENDereco IP DO SERVIDOR: 172.19.0.2
ID para o proximo cadastro de estacao de carga: ZSL7IFDNB5HC0BxBGMNTD3KU
PRESSIONE ENTER A QUALQUER MOMENTO PARA ENCERRAR A APLICACAO
```

Figura 5. Tela inicial do terminal da aplicação do servidor.

4.2. Cadastro do veículo

O processo de cadastro de um veículo só requer ao usuário inserir o endereço IP do servidor e a capacidade em KWh do veículo. Assim como para a estação de recarga, o programa não detecta e não corrige um endereço IP incorreto, e portanto pode ser necessária a reinicialização do programa caso seja feita uma entrada incorreta.

A seguir, a interface gráfica do programa será exibida, contendo todas as informações referentes ao nível de carga do veículo (incluindo aviso caso fique abaixo de 30%), estação mais próxima, próxima compra e histórico de compras, bem como botões para executar ações de busca de estação disponível mais próxima (e suas informações), geração de guia de pagamento, confirmação de pagamento e navegação do histórico de compras.

```
I have no name!@13c545d3c217:/python_redes/03 vehicles$ python client.py
FontManager error: [Errno 13] Permission denied: './.fonts/'
FontManager error: Directory does not exist: './.fonts/'
FontManager error: Directory does not exist: './.fonts/'
FontManager error: Directory does not exist: './.fonts/'
customtkinter.windows.widgets.font warning: Preferred drawing method 'font_shapes' can not be used because the font file could not be loaded.
Using 'circle_shapes' instead. The rendering quality will be bad!
Insira o endereço IP do servidor: []
```

Figura 6. Tela inicial da aplicação do veículo (terminal).

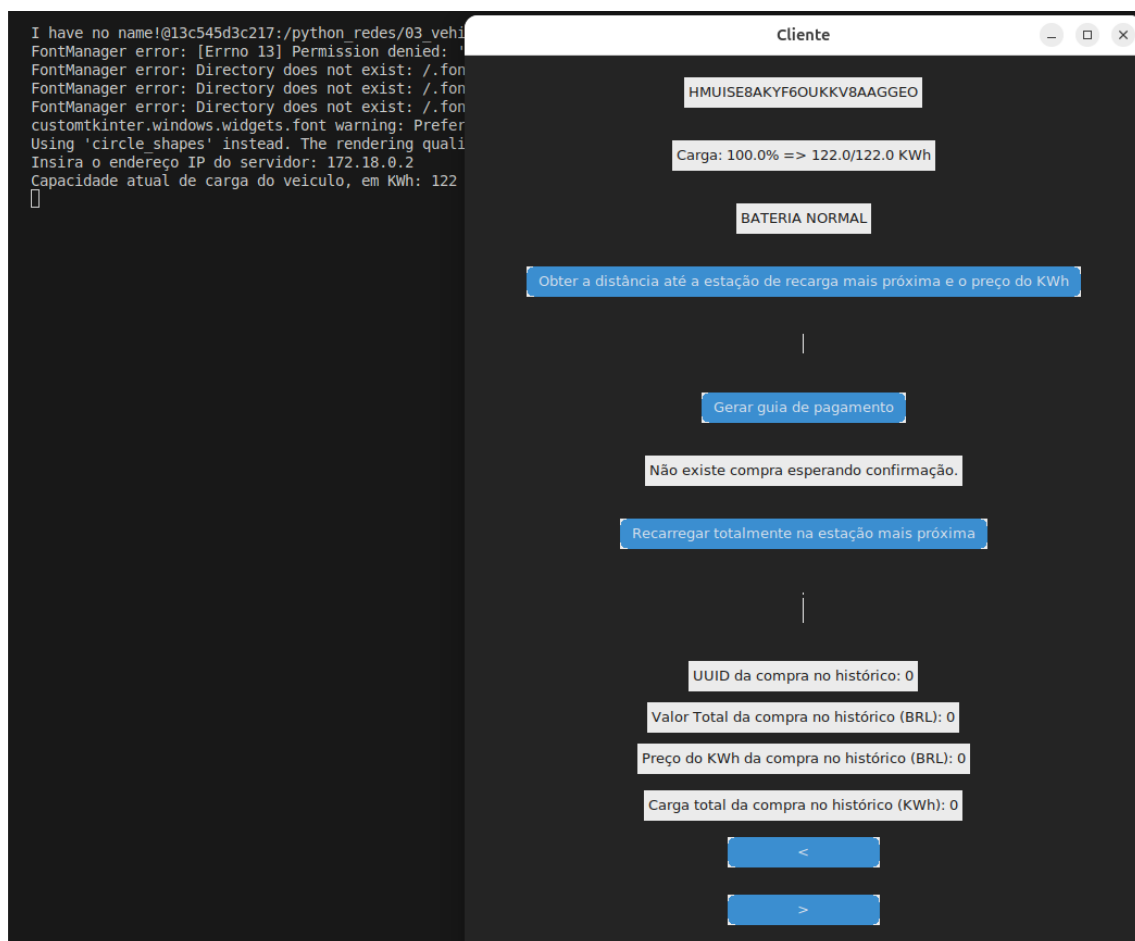


Figura 7. Interface gráfica da aplicação do veículo.

4.3. Cadastro da estação de recarga

Quando executada, a aplicação da estação de recarga requisita ao usuário a entrada do endereço IP do servidor, seguido de informações da estação e do ID para cadastro de estação fornecido por um administrador do sistema com acesso ao terminal do servidor. É importante notar que o programa não detecta e não corrige um endereço IP incorreto, sendo necessária a reinicialização para que esse valor seja mudado. Ademais, caso um ID correto falhe em cadastrar, basta repeti-lo 1 ou 2 vezes.

Após tais informações serem fornecidas e em cada inicialização subsequente do programa, o terminal exibirá o ID da estação e o preço unitário de seu KWh.

```
root@3e4c271dd679:/python_redes/02_station# python client.py
Insira o endereço IP do servidor: 172.18.0.2
Coordenada x do posto de recarga: 76.90
Coordenada y do posto de recarga: 21.54
Preço unitario do KWh, em BRL: 2.18
ID para a estacao de carga (como fornecido pelo servidor): OVUW9SBFIME5AJIFNMVUUX2G
ID para a estacao de carga (como fornecido pelo servidor): OVUW9SBFIME5AJIFNMVUUX2G
*****
ID: OVUW9SBFIME5AJIFNMVUUX2G
Preço do Kwh (BRL): 2.18
*****
```

Figura 8. Tela do terminal da aplicação da estação após o processo de cadastro.

4.4. Requisição de recarga

Um processo de recarga bem-sucedido inicia-se com a busca pela estação disponível mais próxima, utilizando para tal o botão Obter a distância até a estação de recarga mais próxima e o preço do KWh. As informações obtidas em tal passo serão utilizadas na geração da guia de pagamento e na tentativa de agendamento subsequentes.

Em seguida, o usuário deve gerar uma guia de pagamento por meio do botão Gerar guia de pagamento. O processo de geração de guia de pagamento é tão somente um substituto temporário para uma API de serviço de pagamento real, e gera um identificador único uuid4.

Por fim, o usuário deve confirmar que efetuou o pagamento pressionando o botão Recarregar totalmente na estação mais próxima.

Se entre a busca da estação e a confirmação do pagamento nenhum outro veículo agendar com sucesso o local de recarga, o usuário conseguirá agendar a recarga de seu próprio veículo, cabendo ao software de controle do equipamento da estação de recarga verificar o ID do veículo quando este chegar até o ponto e então realizar a recarga.

Cliente

HMUISE8AKYF6OUKKV8AAGGEO

Carga: 25.0% => 30.5/122.0 KWh

BATERIA EM NÍVEL CRÍTICO!

Obter a distância até a estação de recarga mais próxima e o preço do KWh

DISTANCIA: 244.46000000000004 Km | Preço do KWh: 2.18

Gerar guia de pagamento

Não existe compra esperando confirmação.

Recarregar totalmente na estação mais próxima

Compra de UUID <5c86a128-e88b-4020-a518-bc8fa20e4c3f> bem-sucedida. Espere de 1 a 2 minutos para começar o processo de recarga.

UUID da compra no histórico: 0

Valor Total da compra no histórico (BRL): 0

Preço do KWh da compra no histórico (BRL): 0

Carga total da compra no histórico (KWh): 0

<

>

Figura 9. Resultado de uma operação de recarga bem-sucedida

No entanto, caso outro veículo consiga agendar o local de recarga durante a compra, o usuário em questão será notificado de que não conseguiu agendamento e que sua compra foi automaticamente cancelada (estornada), o que de fato acontece no servidor (é chamada uma função PLACEHOLDER para API de serviço de pagamentos).

Vale lembrar que alterações nos parâmetros do veículo (incluindo coordenadas e nível de carga da bateria) devem ocorrer nos arquivos de configuração encontrados em `\vehicledata\vehicle_data.json`, e não por meio do uso do terminal ou da interface gráfica.

Qualquer usuário com ao menos uma compra bem-sucedida realizada pode navegar seu histórico de compras por meio dos botões `<` e `>`. Note que os espaços referentes às informações da compra permanecem vazios até que um dos botões seja pressionado, mesmo após ao menos uma compra ser feita.

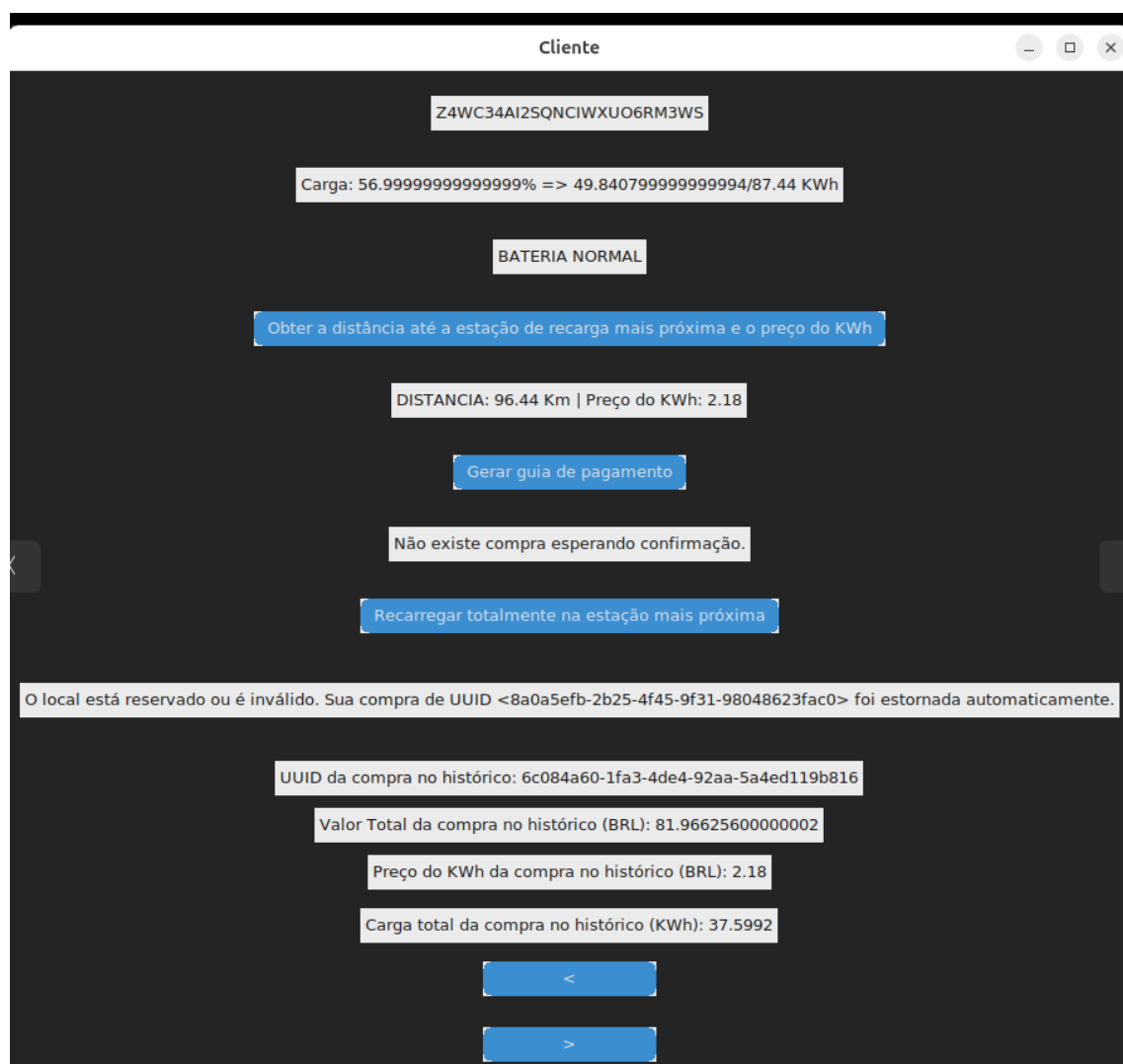


Figura 10. Resultado, na interface gráfica, de falha no processo de agendamento de recarga, após pagamento.

4.5. Processo de recarga

Quando um veículo agenda com sucesso uma recarga, a estação agendada receberá suas informações em até 1 minuto, iniciando o processo de recarga.

Na atual versão de teste do programa, a recarga é feita apenas pressionando a tecla ENTER no terminal da estação.

```
root@3e4c271dd679:/python_redes/02_station# python client.py
Insira o endereço IP do servidor: 172.18.0.2
Coordenada x do posto de recarga: 76.90
Coordenada y do posto de recarga: 21.54
Preço unitario do KWh, em BRL: 2.18
ID para a estacao de carga (como fornecido pelo servidor): OVUW9SBFIME5AJIFNMVUUX2G
ID para a estacao de carga (como fornecido pelo servidor): OVUW9SBFIME5AJIFNMVUUX2G
*****
ID: OVUW9SBFIME5AJIFNMVUUX2G
Preço do KWh (BRL): 2.18
*****
.....
ID do veiculo: HMUISE8AKYF60UKKV8AAGGEO
Carga restante: 91.5
.....
PRESSIONE ENTER PARA REALIZAR A RECARGA TOTAL
```

Figura 11. Resultado no terminal da estação de uma ação de recarga.

4.6. Geração de entrada no registro (log)

O recebimento de mensagens, bem como a execução de ações em cima do banco de dados do servidor, são todas operações registradas em arquivos de texto (logs), os quais podem ser encontrados nas pastas /logs/received/ (mensagens recebidas) e logs/performed/ (ações executadas pelo servidor).

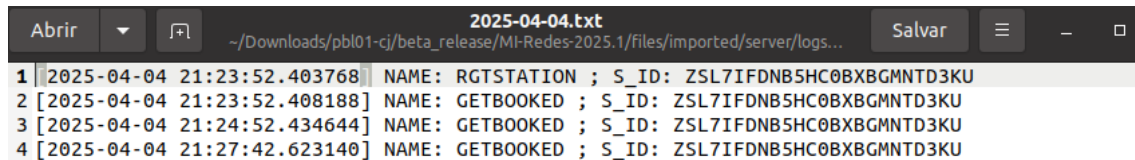


Figura 12. Nome de arquivo e conteúdo de um arquivo .txt referente a registros (logs).

5. Conclusão

O sistema entrega todos os requisitos básicos propostos, sendo estes: cadastro de veículo (usuário final) e estação de recarga, interface de usuário com aviso de estado crítico da bateria, requisição das informações de um ou mais postos próximos, reserva da estação de recarga (incluindo geração de uma pseudo-guia de pagamento e confirmação por parte do servidor), liberação do posto após a recarga, e obtenção por parte do terminal do veículo das informações de cada uma das compras anteriores (histórico) associadas a determinado veículo.

Percebe-se, entretanto, que a solução para conflitos em requisições concorrentes - cancelar a compra caso seja para uma estação atualmente ocupada - é deveras simplista, o que pode ser considerado um ponto em falta no sistema, apesar de ainda cumprir o básico exigido.

Assim sendo, uma melhoria possível é um sistema de filas para os veículos; é importante notar que tal mudança não afetaria a questão de agendamento, ou seja, o processo de agendamento ainda seria feito de forma atômica e sem pré-reserva em seu início, de forma a evitar que clientes mal-intencionados mantenham o sistema ocupado. Neste caso, caberia ao cliente escolher, antes do envio da solicitação de reserva para um ponto de recarga ainda não reservado, se de fato prefere entrar em uma fila caso aconteça reserva antes do fim do processo ou se quer automaticamente cancelar sua compra.

Quanto ao aprendizado proporcionado pelo desenvolvimento, podem ser citados: instalação da *Docker Engine* e utilização de seus contêineres para o desenvolvimento e o teste de sistemas distribuídos, uso de soquetes TCP-IP, uso de linguagem de programação Python com suporte a *multithreading* e uso de biblioteca de criação de interface gráfica para Python (*Custom Tkinter*).

Referências

- Comer, D. (2016). *Interligação de Redes com TCP/IP—: Princípios, Protocolos e Arquitetura*, volume 1. Elsevier Brasil.
- Merkel, D. et al. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2.

Oluwatosin, H. S. (2014). Client-server model. *IOSR Journal of Computer Engineering*, 16(1):67–71.

Silva, M. G., Moreira, L. O., and Coutinho, E. F. Uma proposta de middleware para envio e recebimento de mensagens de objeto (intents) através de sockets: Estudo de caso do progma.