EASYCHARGE

Camila de Araújo Bastos[1], Guilherme Ferreira Lopes[2], Ícaro de Souza Gonçalves[3]

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

camilaabastos7@gmail.com, guife.gf@hotmail.com, icarodsouza662@gmail.com

Resumo. *O crescimento da frota de veículos elétricos no Brasil exige uma infraestrutura de recarga mais eficiente. Este trabalho apresenta o EASYCHARGE, um sistema cliente-servidor desenvolvido em GoLang que indica os melhores pontos de recarga com base na ocupação e distância, permite reservas e pagamentos via PIX. Utiliza TCP/IP, persistência em JSON, containers Docker e conceitos de concorrência com threads e mutex. Os testes demonstraram eficiência no gerenciamento simultâneo de conexões e na redução de tempos de espera.*

1. Introdução

A frota de veículos elétricos no Brasil tem crescido significativamente nos últimos anos, impulsionada por incentivos fiscais, maior eficiência energética e preocupações com a sustentabilidade. No entanto, a infraestrutura de recarga não acompanha esse crescimento, resultando em desafios como filas, tempos de espera prolongados e falta de informações sobre disponibilidade e localização dos pontos de recarga. Esses problemas dificultam a popularização dos veículos elétricos no país. Diante desse cenário, nota-se a necessidade de um sistema inteligente e padronizado que promova a comunicação em tempo real entre veículos e pontos de recarga.

Como contribuição, este trabalho apresenta o desenvolvimento do protótipo inicial de um sistema cliente-servidor (ver Seção 2.1.1). O sistema EASYCHARGE auxilia motoristas ao indicar os melhores pontos de recarga com base na ocupação e distância, além de permitir reservas antecipadas e gerenciamento automatizado da liberação dos pontos após o carregamento. Adicionalmente, registra os valores das recargas realizadas, possibilitando pagamentos eletrônicos via PIX ou outros meios.

Para alcançar esses objetivos, foi adotada uma metodologia baseada na implementação de uma arquitetura cliente-servidor em GoLang. O sistema utiliza persistência de dados em JSON para garantir armazenamento estruturado e comunicação baseada em TCP/IP (ver Seção 2.2) para troca eficiente de mensagens entre os componentes. Além disso, foi empregada a tecnologia Docker (ver Seção 2.4) para orquestrar os serviços do sistema, garantindo escalabilidade e isolamento dos módulos.

Considerando a necessidade de atender múltiplos usuários simultaneamente, foram utilizados conceitos de concorrência com threads e mutex (ver Seção 2.5) para sincronização e controle de acesso aos recursos compartilhados.

Os resultados obtidos demonstram que o sistema é capaz de gerenciar múltiplas conexões simultâneas entre veículos e pontos de recarga, garantindo eficiência no atendimento e redução dos tempos de espera. Além disso, o sistema facilita o planejamento dos motoristas ao fornecer informações em tempo real sobre os pontos disponíveis.

Este relatório está organizado da seguinte forma:

- Na Seção 2, fundamentação teórica necessária para o entendimento do projeto.
- Na Seção 3, metodologia adotada para o desenvolvimento do sistema.
- Na Seção 4, resultados obtidos durante os testes do protótipo.
- Por fim, na Seção 5, conclusões e sugestões para trabalhos futuros.

2. Fundamentação Teórica

2.1. Redes de Computadores: Conceitos Básicos

Redes de computadores são sistemas que interconectam dispositivos para compartilhar recursos físicos ou lógicos por meio de protocolos de comunicação [Tanenbaum and Wetherall 2011]. Esses sistemas possibilitam:

- Compartilhamento de hardware e software;
- Comunicação entre usuários conectados à rede.

2.1.1. Modelo Cliente-Servidor

O modelo cliente-servidor é uma arquitetura amplamente utilizada em redes:

- **Cliente:** Solicita serviços, como navegação na web.
- **Servidor:** Fornece os serviços requisitados, como páginas da web por meio de um servidor HTTP.

2.1.2. Endereço IP e Portas

- **Endereço IP:** Identificador único de dispositivos em uma rede.
- **Porta:** Identifica serviços específicos associados a um IP (ex.: porta 80 para HTTP).

2.2. Arquitetura TCP/IP

A arquitetura TCP/IP organiza a comunicação em redes em quatro camadas principais, conforme apresentado na Tabela 1.

Tabela 1. Camadas da arquitetura TCP/IP.

Camada	Função Principal	Protocolos
Aplicação	Interface com usuário/serviços	HTTP, FTP, DNS
Transporte	Comunicação end-to-end	TCP, UDP
Internet	Roteamento de pacotes	IP, ICMP
Enlace	Transmissão física	Ethernet, Wi-Fi

2.2.1. Fluxo de Dados

Os dados são encapsulados conforme passam pelas camadas:

- **Aplicação:** Segmentos TCP/UDP com identificação das portas.
- **Transporte:** Cabeçalhos contendo portas de origem e destino.
- **Internet:** Pacotes IP com endereços lógicos.
- **Enlace:** Frames com endereços MAC.

2.2.2. Sockets

Sockets são interfaces utilizadas para comunicação entre processos em redes. Eles combinam o endereço IP e uma porta específica para estabelecer conexões.

2.3. Protocolos de Transporte: TCP vs UDP

Os protocolos TCP e UDP possuem características distintas, conforme apresentado na Tabela 2.

Tabela 2. Comparação entre os protocolos TCP e UDP.

Característica	TCP	UDP
Overhead	Alto (confirmações)	Baixo
Controle de Fluxo	Sim	Não
Ordem dos Dados	Sim	Não
Confirmação de Entrega	Sim (ACKs)	Não

2.4. Docker: Plataforma de Conteinerização

Docker é uma tecnologia que permite criar ambientes isolados para execução de aplicações [Docker Inc. 2024]. Seus principais conceitos incluem:

- **Container:** Instância executável isolada e portátil.
- **Imagem:** Modelo read-only contendo código, dependências e configurações.
- **Dockerfile:** Arquivo que define etapas para criação de imagens.
- **Docker Compose:** Ferramenta para orquestrar múltiplos containers utilizando arquivos YAML.

2.5. Concorrência: Threads e Mutex

Threads e mutex são conceitos fundamentais para a implementação de sistemas concorrentes.

2.5.1. Threads

Threads são unidades independentes de execução dentro de um processo que compartilham memória. Sua utilização requer mecanismos de sincronização para evitar condições de corrida [Butenhof 1997].

2.5.2. Mutex

Mutex é um mecanismo que garante exclusão mútua no acesso a recursos compartilhados [Butenhof 1997]:

- **Bloqueio:** A thread adquire acesso exclusivo ao recurso.
- **Liberação:** O recurso é disponibilizado para outras threads após o uso.

3. Metodologia, Aplicação e Testes

O projeto EasyCharge foi implementado em GoLang, utilizando JSON para persistência de dados, Docker para orquestração de contêineres, GitHub para controle de versão e comunicação baseada em TCP/IP nativo para troca de mensagens em tempo real.

3.1. Etapas Evolutivas do Desenvolvimento

O processo de desenvolvimento foi dividido em cinco etapas principais, conectadas para formar uma solução robusta e escalável:

1. **Prototipagem da Comunicação Cliente-Servidor:** Implementação inicial da comunicação entre clientes (postos e veículos) e o servidor central, utilizando sockets TCP para troca de mensagens.
2. **Containerização dos Componentes:** Migração dos componentes para um ambiente Docker. Foram configurados três serviços independentes: servidor, cliente-posto e cliente-veículo.
3. **Modelagem de Entidades:** Criação das entidades fundamentais do sistema (Posto, Veículo e Servidor), cada uma definida com atributos específicos para representar suas características.
4. **Persistência em JSON:** Utilização de arquivos JSON como base de dados estruturada para armazenamento acessível.
5. **Orquestração Final com Docker Compose:** Integração dos componentes utilizando Docker Compose.

3.2. Organização Modular do Sistema

A estrutura do projeto foi organizada para separar responsabilidades de forma lógica (ver Figura 1):

- **Diretório Cliente-Posto:** Contém o arquivo cliente-posto.go, responsável pelas funcionalidades específicas dos postos.
- **Diretório Cliente-Veículo:** Armazena o arquivo cliente.go, que gerencia as operações relacionadas aos veículos.
- **Diretório Modelo:** Centraliza as definições das estruturas de dados e respostas (ex.: posto.go, responses.go e veiculo.go).
- **Diretório Servidor:** Inclui o arquivo principal servidor.go e os arquivos JSON (postos.json e veiculos.json), que servem como base de dados persistente.
- **Arquivos Docker:** O Dockerfile e o docker-compose.yml garantem a execução eficiente em ambientes isolados.

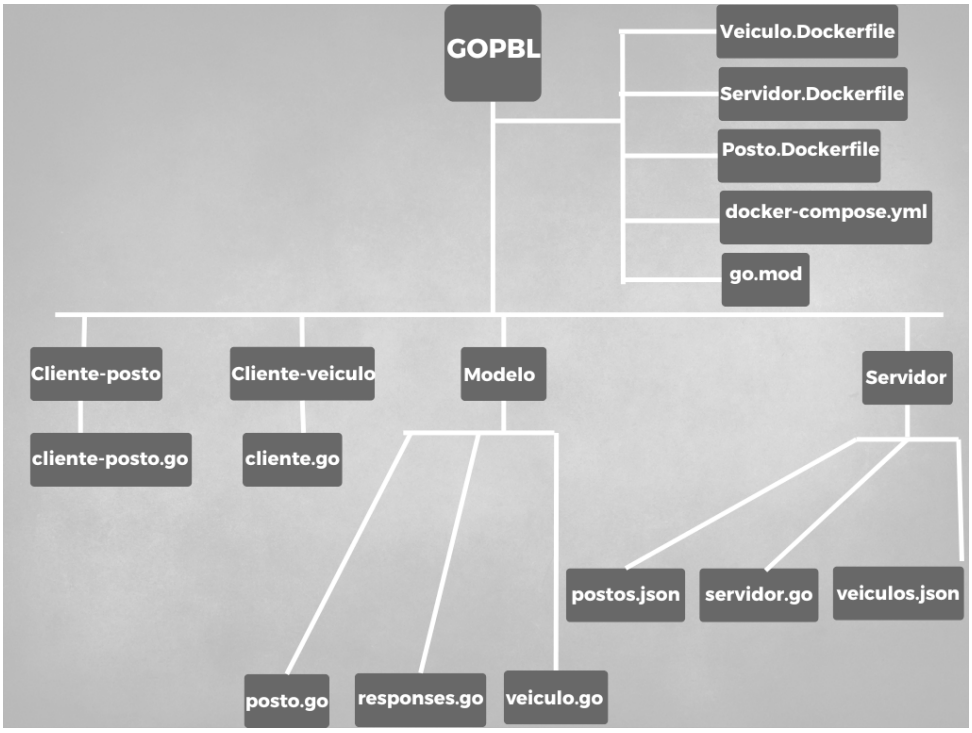

Figura 1. Diagrama organizacional do código

3.3. Arquitetura Modular e Fluxo Operacional

A arquitetura modular sustenta o fluxo operacional do sistema, conforme ilustrado no Diagrama Sequencial (Figura 2). Este diagrama detalha as interações dinâmicas entre veículos, postos e servidor.

3.3.1. Estabelecimento de Conexão

- Veículo e Posto estabelecem conexões TCP independentes com o Servidor.
- Após cada conexão, o servidor identifica se é um posto ou veículo e armazena as conexões em listas separadas.

3.3.2. Cadastro de Entidades

- Veículo envia ao servidor informações como ID, latitude, longitude e nível de bateria por meio do comando `cadastrar-veiculo`.
- Posto envia ID, latitude e longitude com o comando `cadastrar-posto`.
- O servidor armazena os dados recebidos em arquivos JSON para persistência.

3.3.3. A Fila de Veículos

Cada posto mantém uma fila de veículos:

- Gerencia a ordem de atendimento.

- Calcula o tempo estimado de espera e carregamento.
- Organiza o atendimento sequencial dos veículos.
- Reordena-se automaticamente conforme a chegada e saída de veículos.

A fila é constantemente atualizada para garantir estimativas precisas.

3.3.4. Listar e Importar Dados

- **Veículo:** Envia `listar-veiculos`, recebe a lista do servidor, seleciona um ID e o cadastra localmente novamente.
- **Posto:** Utiliza `listarPostosDoArquivo` para receber a lista de postos do servidor, seleciona um ID e o cadastra localmente novamente.

3.3.5. Recomendação de Postos

O cálculo do posto recomendado considera:

1. Distância até o posto, com base na soma dos módulos das diferenças de latitude e longitude multiplicada por um fator de tempo (15 segundos por unidade de distância).
2. Tempo estimado na fila: inclui o tempo de deslocamento e o tempo de carregamento de veículos à frente (1 minuto por 1% de bateria).
3. Tempo total estimado: soma do tempo de deslocamento e espera.

O posto com menor tempo total estimado é selecionado.

3.3.6. Listar Postos Ativos

- Veículo envia `listar-postos`.
- O servidor limpa a lista atual, envia `get-posto` para cada posto conectado.
- Recebe os dados atualizados de cada posto e retorna apenas os ativos.

Esse procedimento assegura dados em tempo real dos postos.

3.3.7. Reserva de Vaga

- O veículo calcula o valor da reserva (R$ 0,50 por cada 1% de bateria faltante).
- Após pagamento, envia `reservar-vaga` ao servidor com o ID do posto.
- O servidor redireciona a requisição ao posto, que adiciona o veículo à fila.
- Confirmações são enviadas de volta ao veículo, que inicia seu deslocamento.

3.3.8. Atualização Contínua

- Veículos em deslocamento enviam `atualizar-posicao-veiculo`.
- Postos enviam atualizações sobre o estado da fila.
- O servidor orquestra a comunicação entre veículos e postos.

3.3.9. Formato e Transmissão de Dados

- Todas as mensagens são transmitidas em formato JSON.
- Cada mensagem inclui um campo de comando e uma estrutura de dados associada.
- Há validações em cada ponto de recepção, com tratamento de erros e registro para depuração.

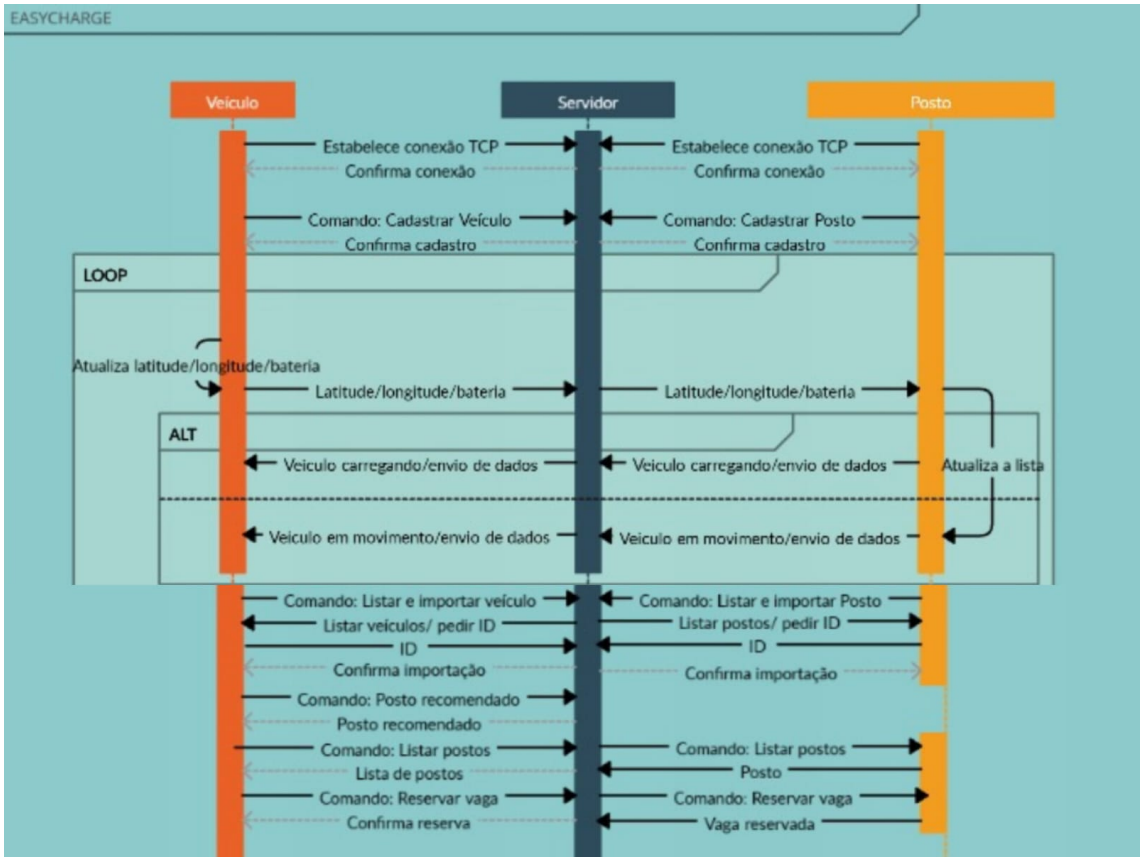

Figura 2. Diagrama sequencial exemplo de uso das operações do sistema

3.4. Gerenciamento de Concorrência

Para lidar com múltiplas conexões simultâneas entre veículos, postos de recarga e o servidor central, foram implementados os seguintes mecanismos:

- Multi-threading com Goroutines: Cada requisição é processada em sua própria goroutine.
- Mutexes para Sincronização: Proteção contra condições de corrida durante atualizações críticas.
- Processamento Paralelo: Requisições simultâneas são tratadas sem bloqueios entre threads.

3.5. Testes Realizados

Os testes validaram a comunicação, cadastro, comandos, concorrência, ordenação e desconexão. As Tabelas 3, 4 e 5 apresentam um resumo dos principais cenários testados.

Tabela 3. Testes de Comunicação e Cadastro

Categoria	Cenário	Ação	Resultado
Comunicação	Veículo conecta	Iniciar TCP	Confirmação do servidor
Comunicação	Posto conecta	Iniciar TCP	Confirmação do servidor
Cadastro	Posto ID duplicado	Enviar ID repetido	Sobrescreve registro
Cadastro	Veículo ID duplicado	Enviar ID repetido	Sobrescreve registro

Tabela 4. Testes de Comandos e Concorrência

Categoria	Cenário	Ação	Resultado
Comando Inválido	Veículo envia 5 (menu 0–4)	Enviar opção inválida	Erro: opção inválida
Comando Inválido	Posto envia 2 (menu 0–1)	Enviar opção inválida	Erro: opção inválida
Concorrência	3 veículos reservam no mesmo posto	Reservar-vaga simultâneo	Fila por chegada/prioridade
Concorrência	Veículo reserva em 3 postos	Reservar em todos	Vai para o último, mantém nas 3
Histórico de Pagamento	Veículo solicita ver histórico	Enviar comando de listagem	Exibe lista de pagamentos do veículo

Tabela 5. Testes de Ordenação e Desconexão

Categoria	Cenário	Ação	Resultado
Ordenação	Veículo mais próximo	Reservar no posto	Fila: 1ª posição
Ordenação	Veículo mais distante	Reservar no posto	Fila: 2ª posição
Desconexão	Posto desconectado	Encerra conexão	Fora das recomendações
Desconexão	Veículo desconectado	Encerra conexão	Fora das filas

4. Resultados e Discussões

Os testes realizados demonstraram que o sistema atende, de forma geral, aos requisitos funcionais propostos, com destaque para a comunicação eficaz entre clientes (postos e veículos) e o servidor, além do gerenciamento dinâmico das filas de espera. As Tabelas 3, 4 e 5 detalham os principais cenários testados.

4.1. Comunicação e Cadastro

A comunicação foi estabelecida com sucesso por conexões TCP independentes, com o servidor diferenciando corretamente veículos e postos e mantendo listas separadas. As requisições de cadastro funcionaram corretamente, com persistência dos dados em arquivos JSON.

Contudo, foi observada uma limitação importante na estratégia de tratamento de IDs duplicados: ao receber um cadastro com um ID já existente, o sistema sobrescreve os dados anteriores. Essa abordagem pode gerar perda de informações e inconsistência histórica. Uma melhoria recomendada é a rejeição de cadastros com IDs já utilizados, acompanhada de uma mensagem de erro informativa ao usuário.

4.2. Validação

Comandos inválidos enviados por veículos e postos foram tratados corretamente, com respostas apropriadas e sem comprometimento do funcionamento do sistema.

4.3. Histórico de Pagamentos

Foi testada com sucesso a funcionalidade de listagem do histórico de pagamentos de um veículo, permitindo que o usuário consulte os registros.

4.4. Fila de Espera e Concorrência

Os testes de concorrência evidenciaram que o sistema é capaz de gerenciar múltiplos veículos realizando reservas em um mesmo posto, organizando a fila conforme critérios de chegada e prioridade por nível de bateria.

4.5. Desconexões e Consistência

O sistema lida corretamente com desconexões inesperadas. Postos que se desconectam deixam de ser considerados nas recomendações de atendimento, e veículos desconectados são removidos das filas de espera. Isso garante que os dados em tempo real reflitam o estado atual da rede, mantendo a confiabilidade das operações.

4.6. Desempenho

Durante a análise, identificou-se que o uso exclusivo do protocolo TCP, embora confiável, pode representar um custo computacional elevado em cenários com muitos veículos e postos conectados. A escolha por TCP foi adequada para operações críticas, como reservas e controle de fila, mas pode ser excessiva em interações simples, onde o uso de UDP seria mais eficiente e escalável.

Além disso, a estratégia atual baseada em múltiplas goroutines simultâneas mostrou-se pouco escalável. A substituição por uma estrutura com buffers centralizados e canais permitiria que as requisições fossem processadas por menos goroutines, otimizando o uso de CPU e facilitando a gestão da concorrência no sistema.

4.7. Considerações Finais

De maneira geral, o sistema apresenta uma estrutura sólida e comportamento coerente frente a múltiplos cenários operacionais. As principais melhorias identificadas para versões futuras são:

- Rejeitar cadastros com IDs duplicados, evitando sobrescrita de registros.
- Otimizar o desempenho da comunicação com uso seletivo de UDP em interações não críticas.
- Substituir goroutines excessivas por um modelo de buffers com canais para melhorar a escalabilidade.

5. Conclusão

O projeto consistiu na implementação de um sistema de gerenciamento de recomendação e reservas entre veículos elétricos e postos de carregamento, utilizando comunicação via sockets TCP, gerenciamento de filas com prioridade e suporte a múltiplas conexões concorrentes. Foram realizados testes para validar funcionalidades como comunicação, cadastro, comandos, concorrência e ordenação de filas.

Foram identificadas melhorias importantes, como a necessidade de impedir cadastros com IDs repetidos e a substituição de goroutines excessivas por canais com buffers para melhor escalabilidade. Também se propõe o uso seletivo de UDP para otimizar interações não críticas..

Apesar dessas limitações, o projeto proporcionou um aprendizado significativo em áreas como redes de computadores, controle de concorrência, tratamento de exceções e organização de fluxos em sistemas distribuídos. Essa experiência prática fortalece a base técnica necessária para o desenvolvimento de soluções mais robustas e escaláveis em futuros projetos com foco em comunicação eficiente entre dispositivos.

Referências

Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison-Wesley, Boston.

Docker Inc. (2024). Docker Documentation. Acesso em: 7 abr. 2025.

Tanenbaum, A. S. and Wetherall, D. J. (2011). *Redes de Computadores*. Pearson Prentice Hall, São Paulo, 5 edition. Tradução da 5ª edição norte-americana.