Recarga de carros elétricos inteligente Concorrência e Conectividade

André V. D. Freitas[1], **Carlos A. B. Nunes**[2], **Arthur Teles D. Oliveira Freitas**[3]

[1]Universidade Estadual de Feira de Santana (UEFS)
Caixa Postal 252 – 44.036-900 – Feira de Santana – BA – Brazil

[2]Laboratório de Redes e Sistemas Distribuidos (LARSID) – UEFS

carlos17.fsa15@gmail.com, andrevinciusdiz@gmail.com, arthurnoyesit@gmail.com

Abstract. *The charging infrastructure for electric vehicles (EVs) requires coordination. Drivers face difficulties finding available charging stations, resulting in wait times. This work describes a client-server system prototype for managing charging stations. The system was developed in Go. Client-server communication uses TCP for commands; station-server communication uses UDP for status updates. The server manages concurrency with mutexes. Data is formatted in JSON. The system is distributed via Docker. The prototype allows vehicles to locate and reserve charging stations. Charging stations update their status (available, reserved, charging). The system manages user accounts for billing.*

Resumo. *A infraestrutura de recarga para veículos elétricos (VEs) requer coordenação. Motoristas enfrentam dificuldades para encontrar pontos de recarga livres, resultando em espera. Este trabalho descreve um protótipo de sistema cliente-servidor para gerenciamento de pontos de recarga. O sistema foi desenvolvido em Go. A comunicação cliente-servidor usa TCP para comandos; a comunicação ponto-servidor usa UDP para status. O servidor gerencia concorrência com mutexes. Os dados são formatados em JSON. O sistema é distribuído via Docker. O protótipo permite aos veículos localizar e reservar pontos. Pontos de recarga atualizam seu estado (livre, reservado, carregando). O sistema gerencia contas de usuários para cobrança.*

1. Introdução

A adoção de veículos elétricos (VEs) cresce globalmente, impulsionada por avanços tecnológicos e preocupações ambientais. No entanto, a eficiência dessa transição depende de uma infraestrutura de recarga acessível e bem gerida. A ausência de informações centralizadas sobre o estado das estações (livre, ocupado, manutenção) gera incertezas e filas desnecessárias.

Este trabalho propõe um sistema cliente-servidor para gerenciar pontos de recarga de VEs, oferecendo informações atualizadas e a possibilidade de reserva antecipada. O protótipo, desenvolvido em Go, utiliza TCP/UDP para comunicação, JSON para troca de dados e Goroutines para concorrência. A solução é conteinerizada com Docker.

O sistema permite que clientes localizem pontos próximos disponíveis e que os pontos de recarga atualizem seu status periodicamente. Também registra o valor das recargas por usuário.

2. Fundamentação Teórica

Esta seção apresenta os conceitos e tecnologias fundamentais que formam a base para o desenvolvimento do protótipo do sistema de gerenciamento de recarga de veículos elétricos.

2.1. Conceitos Fundamentais

- **Arquitetura Cliente-Servidor**: O sistema adota o modelo arquitetural cliente-servidor. Neste modelo, clientes (aplicativos nos veículos) requisitam serviços a um servidor centralizado (gerenciador de recarga) [Tanenbaum and Van Steen 2007]. Esta arquitetura facilita a centralização da lógica de negócio e do estado do sistema.
- **Comunicação em Rede (TCP/UDP)**: A interação entre componentes distribuídos requer protocolos de comunicação. Os protocolos TCP (Transmission Control Protocol) e UDP (User Datagram Protocol) da pilha TCP/IP são utilizados [Forouzan 2009]. TCP oferece comunicação confiável e orientada à conexão, adequada para comandos críticos. UDP oferece comunicação rápida e sem conexão, adequada para mensagens de status frequentes onde a perda ocasional é tolerável.
- **Serialização de Dados**: Para a troca de informações estruturadas entre componentes de software é necessária a serialização de dados. JSON (JavaScript Object Notation) é um formato leve e textual amplamente utilizado para este fim [Ecma International 2017].
- **Concorrência em Sistemas Servidores**: Servidores tipicamente precisam atender múltiplas requisições de clientes simultaneamente. Modelos de concorrência, como o uso de threads ou processos leves (Goroutines em Go), são essenciais para garantir a responsividade e escalabilidade do servidor [Pike 2012]
- **Sincronização**: Em ambientes concorrentes, o acesso a recursos compartilhados (como dados sobre o estado dos pontos de recarga) deve ser coordenado para evitar inconsistências. Mecanismos de sincronização, como mutexes (Mutual Exclusion locks), são usados para garantir acesso exclusivo a seções críticas do código [Silberschatz et al. 2018].
- **Conteinerização**: É uma tecnologia de virtualização no nível do sistema operacional que permite empacotar uma aplicação e suas dependências em uma unidade isolada chamada contêiner. Isso promove portabilidade e consistência entre ambientes de desenvolvimento, teste e produção [Docker Inc. 2024].

2.2. Tecnologias Empregadas

Go (Golang): Linguagem de programação compilada e estaticamente tipada, conhecida por seu suporte nativo à concorrência (Goroutines e canais) e bibliotecas padrão robustas para desenvolvimento de sistemas de rede. Foi a linguagem escolhida para implementar todos os componentes do protótipo.

Docker: Plataforma de conteinerização utilizada para empacotar cada componente do sistema (servidor, cliente, ponto) em imagens Docker.

Docker Compose: Ferramenta para definir e executar aplicações Docker multi-contêiner. Foi utilizada para orquestrar a execução conjunta dos três componentes do sistema em uma rede virtual definida.

3. Metodologia

Esta seção descreve a implementação prática do protótipo, detalhando como os conceitos e tecnologias apresentados na Seção 2 foram aplicados para solucionar os requisitos do sistema de gerenciamento de recarga.

3.1. Implementação da Arquitetura

A arquitetura cliente-servidor foi implementada com três programas distintos em Go: server, cliente e ponto de recarga. Como demonstrado na figura 1, o server atua como processo central, escutando por conexões e mantendo o estado global. O cliente inicia conexões TCP com o servidor para enviar requisições e receber respostas. O ponto de recarga recebe requisições do servidor via TCP e envia atualizações de estado via UDP.

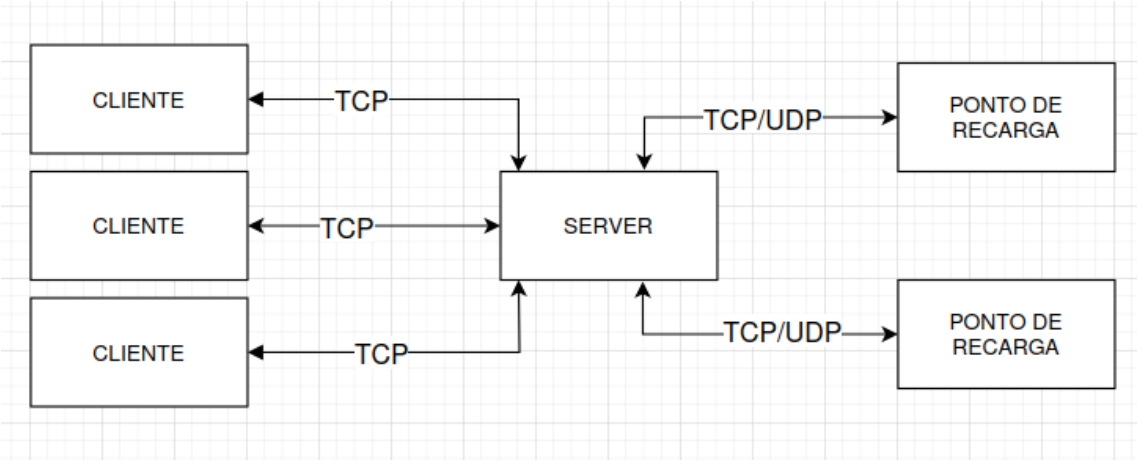

Figura 1. Esquema representando a comunicação do projeto

3.2. Aplicação dos Protocolos de Comunicação

A biblioteca net do Go foi utilizada para estabelecer a comunicação via sockets, representada na figura 1.

- **TCP**: Usado na comunicação Cliente-Servidor. O servidor escuta com net.Listen e aceita conexões com Accept, cada uma tratada em uma Goroutine. O cliente usa net.Dial para se conectar.
- **UDP**: Usado para atualizações de status do Ponto ao Servidor. O servidor usa net.ListenPacket e uma Goroutine para processar os datagramas. O ponto envia dados com net.Dial (UDP) e conn.Write, priorizando eficiência em comunicações frequentes.

3.3. Troca de Dados via JSON

As mensagens entre os componentes são definidas como 'structs' em Go, serializadas em JSON com 'json.Marshal' antes de serem enviadas via socket. No receptor, os dados são lidos e desserializados com 'json.Unmarshal', garantindo uma representação padronizada e fácil de interpretar.

3.4. Implementação da Concorrência e Sincronização

O servidor lida com múltiplas conexões simultâneas usando Goroutines, iniciando uma nova para cada conexão TCP. O acesso concorrente aos mapas compartilhados (pontos de carga e contas de usuários) é protegido com ‘sync.Mutex‘, garantindo que operações críticas sejam feitas de forma segura com ‘mu.Lock()‘ e ‘mu.Unlock()‘.

3.5. Gerenciamento de Estado Distribuído

O estado instantâneo de cada ponto (ex: ’Disponível’, ’Carregando’) é gerenciado localmente pelo processo ”Charging Point”e reportado periodicamente ao servidor via UDP. O Servidor Central, além de agregar este estado, implementa e gerencia uma fila de reservas para cada ponto de recarga.

Quando um cliente solicita uma reserva, o servidor verifica a ocupação atual desta fila específica. A reserva é aceita e adicionada à fila se a capacidade máxima da fila não foi atingida. Desta forma, um veículo pode reservar um ponto mesmo que ele esteja atualmente no estado ’Carregando’, garantindo seu lugar na espera.

A reserva é rejeitada apenas quando a fila específica do ponto está cheia. O acesso concorrente tanto ao estado instantâneo dos pontos quanto às suas filas de reserva é protegido no servidor utilizando um mutex.

Quando o processo ”Charging Point”sinaliza via UDP que uma sessão de carregamento terminou (voltando ao estado ’Disponível’ localmente), o servidor processa o próximo cliente da fila daquele ponto, se houver, ou marca o ponto como totalmente disponível.

3.6. Tratamento de Erros e Confiabilidade

A implementação trata erros básicos de rede e parsing, verificando retornos de funções como ‘net.Dial‘, ‘conn.Read‘, ‘conn.Write‘ e ‘json.Unmarshal‘. Quando há erro, ele é registrado com o pacote ‘log‘. Conexões TCP encerradas ou com erro terminam a goroutine do servidor. A lógica de reserva verifica o estado do ponto antes de confirmar, e uma duração de reserva é usada para liberar pontos automaticamente após um tempo.

3.7. Metodologia de Empacotamento e Execução com Docker

Cada componente (servidor, cliente e ponto) foi conteinerizado com um Dockerfile específico em formato multiestágio: a aplicação é compilada em uma imagem Go e o executável é copiado para uma imagem mínima, reduzindo o tamanho final. O ‘docker-compose.yml‘ define os três serviços, mapeia portas e configura uma rede Docker para permitir comunicação entre contêineres por nome. A aplicação é executada com ‘docker-compose up‘, que constrói e inicia os contêineres na ordem correta.

4. Resultados e Discussão

Esta seção apresenta os resultados obtidos com a execução do protótipo do sistema de gerenciamento de recarga e discute suas características, limitações e o alcance dos objetivos propostos.

4.1. Resultados Obtidos

A execução do protótipo foi realizada utilizando Docker Compose, conforme documentado no arquivo README.md. Os três componentes (Servidor Central, Cliente Veículo, Ponto de Recarga) foram iniciados como contêineres Docker em uma rede virtual isolada, demonstrando a viabilidade da metodologia de empacotamento e orquestração escolhida.

Durante a execução, observou-se o funcionamento das seguintes funcionalidades principais:

- **Comunicação entre Componentes**: O Servidor Central estabeleceu com sucesso sockets de escuta TCP e UDP. Clientes puderam conectar-se ao servidor via TCP para enviar comandos. Pontos de Recarga puderam enviar mensagens de status via UDP para o servidor sempre que ocorria uma atualização em algum status do ponto de recarga. Logs gerados pelos componentes confirmaram o recebimento e envio de mensagens nos formatos esperados (JSON).
- **Funcionalidade do Cliente Veículo**: O cliente simulado permitiu ao usuário (via terminal) com o auxílio de uma biblioteca para melhor navegação do usuário, executar as seguintes ações:
  - Enviar o comando 'Notificação de bateria' com coordenadas simuladas.
  - Receber do servidor uma lista de pontos de recarga próximos, incluindo seus IDs e status (Disponível, Reservado, Carregando).
  - Enviar o comando 'Reservar ponto' para um ponto disponível.
  - Receber confirmação ou falha da reserva do servidor.
  - Enviar o comando 'Chegada' para iniciar o carregamento e liberar o posto.
- **Funcionalidade do Ponto de Recarga**: O ponto de recarga simulado demonstrou as seguintes capacidades
  - Inicialização e registro (implícito pelo envio de status) junto ao servidor.
  - Envio periódico de mensagens de status via UDP, refletindo seu estado atual (Disponível, Reservado, Carregando).
  - Transição automática de estados (ex: de Disponível para Carregando, ou quando a fila atinge o limite de reservas ficando Reservado).

- **Funcionalidade do Servidor Central**: O servidor demonstrou gerenciar o estado do sistema e processar requisições:

- Recebimento e **processamento concorrente de conexões** TCP de múltiplos clientes (através de Goroutines).
- Recebimento e **processamento de datagramas UDP de múltiplos pontos.**
- **Atualização do estado dos pontos de recarga** (mapa chargingPoints) com base nas reservas e nos status recebidos. O uso de sync.Mutex garantiu o acesso seguro a este estado compartilhado.
- **Gerenciamento básico de contas de usuário** (mapa cars), incluindo consulta de custos de recarga.
- **Geração de logs** indicando as operações realizadas e potenciais erros.

Os resultados demonstram que a arquitetura e a metodologia implementadas permitem a comunicação e a coordenação básicas entre os componentes para realizar as operações fundamentais de localização, reserva e atualização de status dos pontos de recarga.

4.2. Discussão e Avaliação Crítica

A avaliação do protótipo considera tanto os objetivos alcançados quanto as limitações inerentes à sua natureza e escopo.

Objetivos Atingidos:

- **Localização e Disponibilidade**: O sistema cumpre o objetivo central de permitir que um cliente (veículo) localize pontos de recarga próximos e verifique sua disponibilidade em tempo real.
- **Reserva**: A funcionalidade de reserva de pontos foi implementada, permitindo ao usuário garantir um ponto antes de se deslocar. O controle de concorrência com mutex evita reservas conflitantes do mesmo ponto.
- **Infraestrutura Cliente-Servidor**: A arquitetura foi implementada com sucesso usando Go e Docker, provendo uma base funcional.

Limitações e Aspectos Não Cobertos:

- **Escalabilidade e Desempenho**: O uso de um mutex global pode limitar o desempenho sob alta carga. Estratégias como sharding ou locks mais granulares não foram exploradas.
- **Robustez e Tolerância a Falhas**:O tratamento de erros é básico, sem persistência de dados ou tolerância a falhas. Desconexões e falhas não são bem tratadas.
- **Monitoramento**: Limitado a logs no console, sem dashboards ou alertas automatizados.

Em suma, o protótipo demonstra a viabilidade dos conceitos e tecnologias escolhidos para as funcionalidades básicas de gerenciamento de recarga. Contudo, ele representa uma prova de conceito inicial. Uma solução pronta para produção exigiria desenvolvimento adicional significativo nas áreas de robustez, escalabilidade, segurança, realismo da simulação e algoritmos de otimização.

Referências

Docker Inc. (2024). Docker documentation. `https://docs.docker.com/`. Accessed: 2025-04-07.

Ecma International (2017). Ecma-404 the json data interchange syntax. `https://www.ecma-international.org/publications-and-standards/standards/ecma-404/`. Standard ECMA-404, 2nd Edition.

Forouzan, B. A. (2009). *Comunicação de dados e redes de computadores*. AMGH Editora.

Pike, R. (2012). Go concurrency patterns. `https://blog.golang.org/concurrency-patterns`. Accessed: 2025-04-07.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2018). *Operating System Concepts*. John Wiley & Sons, 10th edition.

Tanenbaum, A. and Van Steen, M. (2007). *Sistemas distribuídos: princípios e paradgmas*. Pearson Educación.