Relatório do Problema 1: Recarga de carros elétricos inteligente

Lucas Carneiro de Araújo Lima, Daniel Santa Rosa Santos, Antonio Wellington Santana Sousa

[1]Universidade Estadual de Feira de Santana (UEFS)
Feira de Santana – Bahia – Brasil
Engenharia de Computação

lucascarneiro3301@gmail.com, daniel_srs_@hotmail.com,

awssousa2001@gmail.com

Resumo. *Este relatório apresenta o desenvolvimento de um sistema de recarga inteligente, projetado para otimizar o uso das estações de carregamento por meio de uma distribuição eficiente dos usuários, considerando disponibilidade e localização. O sistema foi implementado com base na arquitetura TCP/IP e no modelo de comunicação cliente-servidor.*

1. Introdução

Com o aquecimento global assolando o planeta, soluções para a substituição de combustíveis fósseis aceleraram-se e alternativas tecnológicas para isso são criadas dia após dia. Nesse contexto, destaca-se o projeto da criação dos carros elétricos na tentativa de diminuir a liberação de gases do efeito estufa.

No entanto, a utilização de carros elétricos ainda é um pouco negligenciada, pelo fato de seus preços serem exorbitantes. Além disso, há a escassez de postos para sua recarga, principalmente em cidades pequenas.

Um sistema de recarga inteligente pode contribuir significativamente para otimizar o uso das estações, distribuindo melhor os usuários conforme a disponibilidade e localização, reduzindo filas e o tempo de espera.

Este relatório tem como objetivo explicar e analisar como foi planejado e produzido o sistema para a recarga desses automóveis, utilizando a arquitetura TCP/IP e baseado no modelo de comunicação cliente-servidor para permitir o envio e recebimento de requisições entre os usuários e o servidor central do sistema.

O restante deste relatório está disposto da seguinte forma: Seção 2 apresentando os fundamentos teóricos; Seção 3 a metodologia utilizada, a implementação e os testes; Seção 4 os resultados e discussões; e, por fim, a Seção 5 com as conclusões obtidas.

2. Fundamentação Teórica

2.1. Arquitetura TCP/IP

O modelo de referência TCP/IP surgiu a partir do desenvolvimento da ARPANET, uma rede de pesquisa financiada pelo Departamento de Defesa dos Estados Unidos. Inicialmente voltada à conexão entre universidades e órgãos públicos, a ARPANET enfrentou,

com o tempo, desafios relacionados à interligação com novas tecnologias emergentes, como redes de rádio e satélite [Tanenbaum and Wetherall 2011].

Para resolver essas dificuldades, foi necessário criar uma nova arquitetura capaz de integrar diferentes tipos de redes de maneira uniforme. Dessa necessidade nasceu o modelo TCP/IP, projetado com dois principais objetivos: garantir a **interoperabilidade** entre redes heterogêneas e assegurar a **resiliência**, ou seja, a continuidade da comunicação mesmo diante da falha de partes da rede [Tanenbaum and Wetherall 2011].

A arquitetura TCP/IP pode ser compreendida como a junção de protocolos que servem como base para a internet e para redes locais: o Protocolo de Controle de Transmissão (TCP) e o Protocolo de Internet (IP), em que ambos trabalham em conjunto para possibilitar a comunicação entre dispositivos em redes distintas [ESR]. Essa arquitetura é composta por quatro camadas principais, conforme a Figura 1:

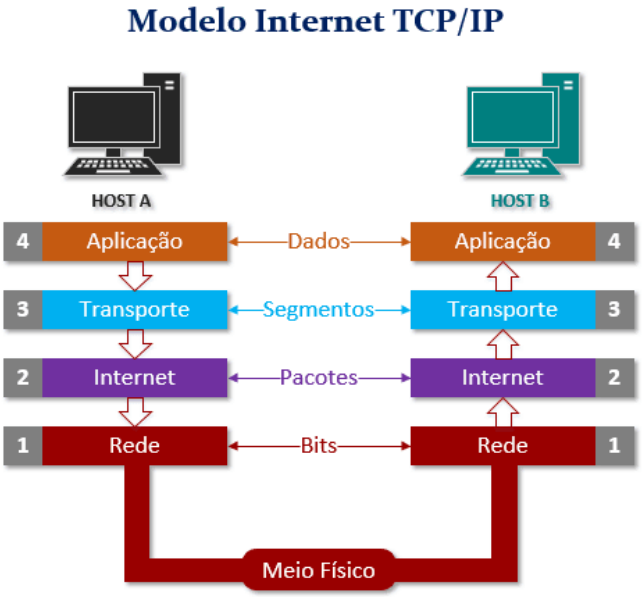

Figura 1. Arquitetura TCP/IP. Fonte: [Passei Direto]

- **Camada de Acesso à Rede:** Também chamada de *link* de dados, é responsável pela parte física da infraestrutura, possibilitando a comunicação entre computadores pela internet. Exemplos dessa parte física incluem redes sem fio, ethernet, entre outras.
- **Camada de Internet:** É responsável por interligar diferentes redes e garantir que os pacotes de dados sejam encaminhados do host de origem até o host de destino, independentemente das redes envolvidas no trajeto.
- **Camada de Transporte:** Situando-se logo acima da camada internet, a camada de transporte tem como principal função possibilitar a comunicação direta entre processos em dispositivos diferentes, garantindo que os dados transmitidos de um host de origem possam ser corretamente recebidos pelo host de destino. Nessa camada são utilizados diferentes protocolos, sendo os principais o TCP e o UDP (Protocolo de Datagrama de Usuário).
- **Camada de Aplicação:** É o topo da arquitetura e apresenta os protocolos de nível mais alto. Esta camada é composta por aplicativos que possibilitam o acesso

do usuário à rede, pelos quais são realizados a maior parte das requisições para execução de tarefas na rede.

2.2. Modelo Cliente-Servidor

O conceito de **cliente-servidor** refere-se a um modelo de comunicação que vincula vários dispositivos informáticos através de uma rede. O **cliente**, nesse contexto, faz solicitações de serviços ao **servidor**, responsável por atender a esses requisitos. Com esse modelo, as tarefas são distribuídas entre os servidores (que fornecem os serviços) e os clientes (que exigem esses serviços). Em outras palavras, o cliente solicita ao servidor um recurso, e este fornece uma resposta [DIO].

Com isso, é possível distribuir a capacidade de processamento. O servidor pode ser executado em mais de um equipamento e também ser representado por mais de um programa. Esse tipo de arquitetura é utilizado por grande parte dos serviços da internet. Os conceitos de cliente e servidor são definidos da seguinte forma [DIO]:

- **Cliente:** Refere-se a qualquer dispositivo (como computador, celular etc.) que solicita e inicia uma comunicação com o servidor. Seu papel é apenas enviar uma solicitação e receber uma resposta, sem precisar saber como os dados são processados.
- **Servidor:** É o sistema centralizado capaz de receber as solicitações dos clientes, processar os pedidos e enviar uma resposta adequada. Ele pode hospedar dados, gerenciar transações e aplicações.

Alem disso, o processo cliente-servidor segue o seguinte ciclo [DIO]:

1. **Cliente envia uma solicitação:** Pode ser, por exemplo, uma interação com um site;
2. **O servidor processa a solicitação:** O servidor busca os dados solicitados, como uma música, imagem etc.;
3. **O servidor envia a resposta:** Após processar os dados, o servidor os envia ao cliente;
4. **O cliente exibe os dados:** Os dados são apresentados ao usuário final.

3. Metodologia, Implementação e Testes

3.1. Configuração do Docker

O código abaixo ilustra como cada Dockerfile foi configurado neste projeto:

```
FROM node:23-alpine
WORKDIR /app
COPY package.json ./
COPY yarn.lock ./
RUN corepack enable
RUN yarn
COPY . .
CMD ["yarn", "..."]
```

Dessa forma, o Docker foi configurado para executar uma aplicação Node.js. A imagem é baseada no Alpine Linux, o que reduz o tamanho do container, e o gerenciador Yarn cuida da instalação das dependências.

Além disso, foi utilizado a rede no modo *host*, fazendo com que cada conteiner utilize diretamente a pilha de rede da máquina hospedeira. Isso permite que a aplicação acesse e ofereça serviços nas mesmas portas do sistema local, sem a necessidade de redirecionamento ou mapeamento de portas [Docker, Inc. 2024].

3.2. Comunicação TCP/IP

O código abaixo ilustra como a comunicação TCP/IP foi configurada neste projeto:

```
const server = net.createServer(socket => {
  socket.on('data', d => {...});

  socket.on('end', () => {...});

  socket.on('error', err => {...});
});

server.listen(PORT, HOST, () => {...});

server.on('error', err => {...});
```

A comunicação TCP/IP foi construída utilizando o módulo *net* do Node.js, que facilita a criação de um servidor TCP. Dessa forma, o servidor fica "ouvindo" conexões em uma porta e endereço definidos pelas variáveis PORT (8080) e HOST (*localhost*). Quando um cliente se conecta, o servidor identifica o IP e a porta de origem e começa a escutar os dados enviados por esse cliente por meio de um socket, que funciona como um canal de comunicação entre os dois lados.

Para isso, o método `listen` é utilizado, pois ele inicia o servidor e o deixa pronto para receber conexões. Já o método `on` é usado para registrar os eventos que o servidor deve tratar, como a chegada de dados ('data'), a finalização da conexão ('end') e eventuais erros de comunicação ('error').

3.3. Interface de Comunicação

O código abaixo ilustra como as mensagens de requisição são definidas neste projeto:

```
type ApiRequest = {
  type: DefinedEndpoints;
  data: RequestResponseMap[DefinedEndpoints]['input'];
};

export type Request = ApiRequest;
```

Dessa forma, toda requisição deve conter um `type` e um `data`. O primeiro, como o próprio nome sugere, se refere ao tipo de requisição que se deseja realizar, a qual estará associada a um *endpoint* do servidor. Já o segundo representa os dados de

entrada necessários para essa operação, que, neste projeto, podem incluir identificadores de pontos de recarga, informações dos carros, nível da bateria, etc.

Além da requisição, a resposta gerada pelo servidor também segue um formato definido, conforme o código abaixo:

```
export type Response<T> = {
  message: string;
  success: true;
  data: T;
};

export type ErrorResponse<T> = {
  message: string;
  success: false;
  error: T;
};
```

Dessa forma, a resposta do servidor, tanto em caso de sucesso (`Response`), quanto em caso de erro (`ErrorResponse`), contém os atributos `message` e `success`. O primeiro é uma *string* usada para explicar o resultado da requisição, enquanto que o segundo indica se houve sucesso ou não. Em caso de sucesso, é enviado ao cliente, através do campo `data`, os dados requeridos (como uma lista de pontos de recarga, por exemplo). Caso contrário, a causa do erro é informada no campo `error`.

3.4. Validação

Para validar as mensagens recebidas, o servidor utiliza a função `connectionSchema`. Essa função descreve todas as estruturas de requisição válidas que o servidor pode receber, garantindo que apenas mensagens corretamente formatadas sejam processadas, evitando erros causados por dados malformados ou incorretos. O código abaixo ilustra como a validação de mensagens foi definida neste projeto:

```
export const connectionSchema = z.discriminatedUnion('type', [
  z.object({
    type: z.literal(...),
    data: z.object({...}),
  }),
]);
```

3.5. Diagrama do Projeto

Conforme a Figura 2, o diagrama mostra a interação entre três entidades principais: **Carro**, **Ponto de Recarga** e **Servidor**. Neste sistema, tanto o Carro quanto o Ponto de Recarga atuam como clientes, enquanto o Servidor realiza o processamento e armazenamento das informações conforme as solicitações dos clientes.

3.6. Pontos de Recarga

O código abaixo ilustra como o Ponto de Recarga foi definido nese projeto:

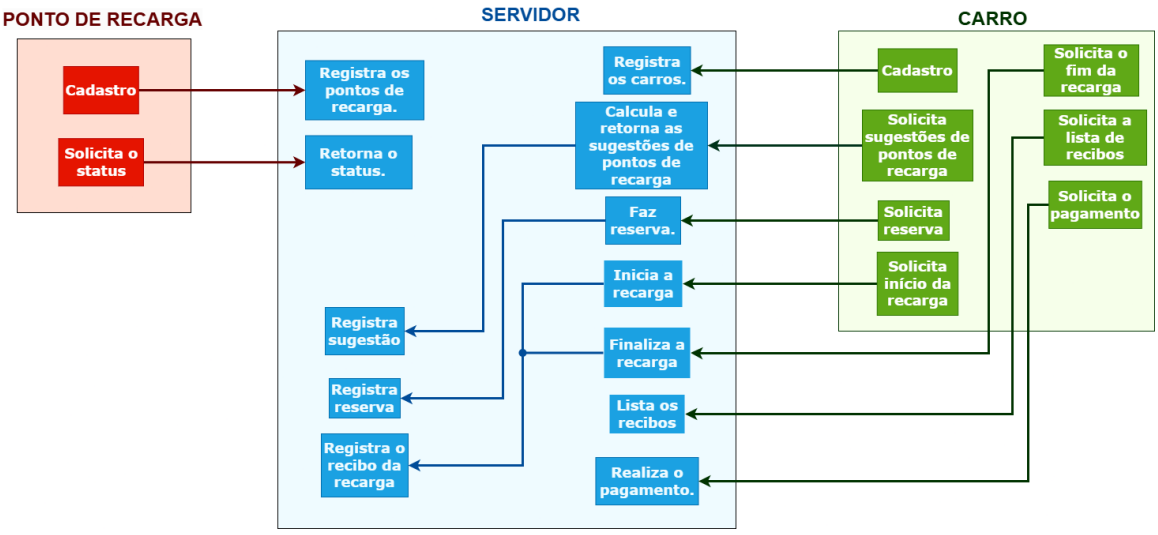

Figura 2. Diagrama do Projeto. Fonte: Autor

```
export type Station = {
  id: number;
  location: Position;
  reservations: number[];
  suggestions: number[];
} & STATION_STATE;
```

Logo, considerou-se que cada ponto de recarga possui os seguintes atributos: `id`, utilizado para identificação unequívoca, `location`, que salva a posição (definida pelas coordenadas X e Y), `reservations`, uma fila que registra os *ids* de todos os carros reservados, `suggestions`, uma lista que salva os *ids* dos carros que receberam o ponto de recarga em questão como primeira sugestão, e `STATION_STATE`, que salva os possíveis estados que os ponto de recarga pode assumir.

Quando um ponto de recarga não possui reservas, ele está no estado *avaliable* (disponível), o que significa que qualquer carro tem a permissão de iniciar uma recarga, mesmo sem possuir uma reserva. Quando há uma ou mais reservas, o ponto de recarga entra no estado *reserved* (reservado), ou seja, apenas o carro que está em primeiro na fila de reservas pode iniciar uma recarga. Por fim, há o estado de *charging-car*, o que implica que há, atualmente, um carro recarregando no posto. Uma vez no estado *charging-car*, o ponto de recarga conta com o atributo `onUse`, que salva o *id* da última recarga realizada, permitindo que apenas o último carro a iniciar a recarga possa finalizá-la.

3.7. Carro

O código abaixo ilustra como o Carro foi definido nese projeto:

```
export interface Car {
  id: number;
  location: Position;
  batteryLevel: number;
}
```

Ou seja, o Carro possui os atributos `id`, `location` e `batteryLevel`. Além disso, a entidade Carro também é tratada como o usuário do sistema (o qual compartilha o mesmo número de identificação do carro).

3.8. Servidor

O papel do Servidor é atuar como intermediário centralizado entre os clientes (Carros ou Pontos de Recarga) e os dados/ações do sistema (como reservas, recargas, registros, etc.). Os tópicos abaixo descrevem o que o Servidor faz, passo a passo:

- Escuta conexões TCP na porta `8080`.
- Recebe mensagens JSON que indicam uma ação (`type`) e seus dados (`data`).
- Valida a estrutura da mensagem.
- Seleciona uma rota apropriada (como `startCharging`, para iniciar uma recarga, por exemplo.) com base no campo `type`.
- Executa a função associada àquela rota, passando os dados necessários.
- Envia uma resposta ao cliente com o resultado (sucesso, erro, dados).

O Servidor funciona por meio de um roteador interno baseado em mensagens, onde cada tipo de requisição (fazer uma reserva, por exemplo) é tratada por uma função específica, chamada de rota. Cada rota é usada para definir o comportamento do Servidor diante de diferentes tipos de requisições que chegam via socket. O código abaixo ilustra como este sistema de roteamento foi definido neste projeto:

```
const router = createRouter()
      .add('reserve', reserve(...))
      .add('getSuggestions', getSuggestions(...))
      .add('registerStation', registerStation(...))
      .add('registerUser', registerUser(...))
      .add('startCharging', startCharging(...))
      .add('endCharging', endCharging(...))
      .add('rechargeList', rechargeList(...))
      .add('payment', payment(...));
```

3.9. Cadastro e Registro de Pontos de Recarga

Para efetuar o cadastro no sistema, requerem-se de um ponto de recarga apenas três informações: O número de identificação (*id*) e as coordenadas (eixo X e Y). Este processo só é feito com sucesso se o *id* solicitado não pertence a nenhum ponto de recarga já cadastrado.

3.10. Cadastro e Registro de Carros

Ao contrário do ponto de recarga, o cadastro dos carros exige apenas o número de identificação. Porém, da mesma forma, esta operação só é corretamente realizada se o *id* solicitado não pertence a nenhum outro carro já cadastrado.

3.11. Cálculo de Sugestões de Pontos de Recarga

Implementada pela rota `stationSuggestions`, essa funcionalidade retorna uma lista de sugestões conforme solicitação do Carro. Para obter a lista de sugestões de pontos de recarga, é necessário, inicialmente, ordenar os pontos de recarga conforme diferentes parâmetros com diferentes prioridades, sendo eles:

1. **Disponibilidade**: Pontos de recarga disponíveis (sem reserva) têm prioridade sobre pontos de recarga reservados.
2. **Quantidade de reservas**: Quanto menos reservas, mais prioridade o ponto de recarga terá.
3. **Distância**: Quanto mais próximo ao carro, mais prioridade o ponto de recarga terá.
4. **Quantidade de sugestões**: Quanto menos sugestões a outros carros o ponto tiver, maior será sua prioridade.

Após isso, a lista ordenada de pontos de recarga é devolvida pelo servidor ao carro que fez a solicitação. Além disso, o ponto de recarga com maior nível de recomendação (o primeiro da ordenação) recebe o *id* do carro na sua lista de sugestões (`suggestions`). Caso o carro que solicitou as recomendações já esteja na lista de sugestões de outro ponto de recarga, seu *id* é removido desta lista. Esse controle de sugestões (que constitui a última etapa de ordenação) serve para recomendar pontos de recargas que menos foram sugeridos a outro carros, uma vez que a probabilidade do ponto de recarga estar livre ou pouco concorrido aumenta

3.12. Reservas

O ato de fazer uma reserva equivale, na prática, a ocupar a fila de reservas (`reservations`) de um ponto de recarga. Para isso, o Carro solicita ao servidor uma reserva, informando seu *id* e o *id* do ponto de recarga desejado. Inicialmente, o servidor verifica se o ponto de recarga escolhido existe no sistema. Caso não exista, retorna uma mensagem de erro. Em seguida, verifica se o usuário já possui uma reserva em outro ponto de recarga; se sim, impede uma nova reserva e retorna uma mensagem de erro. Caso contrário, o identificador do usuário é adicionado à fila de reservas do ponto de recarga. Por fim, o servidor retorna uma resposta informando que a reserva foi feita com sucesso.

3.13. Recibo de recarga

Neste sistema, cada recarga feita por um carro é salva em um Recibo de recarga. Um Recibo de recarga é definido da seguinte forma neste projeto:

```
export type Charge = {
  chargeId: number;
  userId: number;
  stationId: number;
  startTime: Date;
  endTime: Date;
  cost: number;
  hasPaid: boolean;
};
```

O atributo `chargeId` é um identificador único da recarga, enquanto `userId` e `stationId` indicam, respectivamente, o usuário responsável pela recarga e o posto utilizado. Os campos `startTime` e `endTime` registram o momento em que a recarga começou e terminou, servindo apenas como histórico para consulta. O atributo `cost` representa o valor cobrado pela sessão de recarga, e `hasPaid` indica se esse valor já foi quitado.

3.14. Iniciar uma recarga

A operação de iniciar uma recarga exige três parâmetros: o *id* do usuário, o *id* do ponto de recarga e o nível de bateria atual do carro (utilizado para calcular o custo da recarga). Inicialmente, o servidor verifica se tanto o ponto de recarga quanto o usuário existem. Em seguida, avalia se o ponto está disponível ou, caso esteja reservado, se a reserva pertence ao usuário informado. Se essas condições forem atendidas, a reserva é removida (se houver) para que o processo de recarga possa prosseguir.

Após essa validação, um novo recibo de recarga é criado contendo as informações da sessão, como o tempo de início da recarga e o custo baseado no nível de bateria. Esse recibo é então armazenado no registro de recargas. Por fim, o ponto de recarga é marcado como em uso, passando para o estado *charging-car*, e o campo `onUse` da estação é atualizado para referenciar o *id* da nova recarga.

3.15. Finalizar uma recarga

Para finalizar uma recarga, é necessário passar os mesmos parâmetros citados anteriormente. Assim, o servidor valida a existência do ponto de recarga e do usuário. Em seguida, verifica se o ponto de recarga está no estado `charging-car` e se há um registro de recarga válido associado ao ponto de recarga por meio do campo `onUse`. Caso o *id* do usuário informado coincida com o *id* do usuário registrado na recarga em andamento, o campo `endTime` é atualizado e o custo final é ajustado com base no nível atual da bateria, subtraindo o valor previamente calculado no início da recarga. Por fim, o ponto de recarga é liberado e seu status é atualizado para *avaliable*, se não houver mais reservas, ou para *reserved*, caso existam usuários na fila.

3.16. Listar recibos de recarga

A funcionalidade de listar recargas tem como objetivo recuperar todos os recibos de recarga associados a um determinado usuário. Para isso, ela exige que o identificador do usuário seja informado na requisição. Dessa forma, o sistema verifica se esse usuário existe e, caso exista, retorna a lista de recargas vinculadas a ele.

3.17. Pagamento

Para realizar um pagamento de uma recarga, é necessário informar o *id* da recarga e do usuário. Ao receber a solicitação, o servidor primeiro verifica se o cliente existe no sistema. Em seguida, verifica se a recarga informada existe e se ela realmente pertence ao cliente. Se a recarga for válida e estiver associada ao cliente, o sistema então verifica se ela já foi paga. Se o pagamento já tiver sido realizado anteriormente, o servidor emite um aviso e impede um novo pagamento. Caso contrário, o pagamento é confirmado com sucesso, e a recarga é atualizada para refletir esse novo estado (`hasPaid` é definido como verdadeiro).

4. Resultados e Discussões

Como demonstrado nas seções anteriores, temos que a solução criada satisfaz, a priori, os requisitos e restrições do problema proposto. Temos o servidor central que concentra todas as regras, dados e usuários e as aplicações que consomem (carro e estação) as funcionalidades disponibilizadas, reduzindo assim o nível de complexidade dos sistemas,

que mesmo funcionando de maneira distribuída, têm uma arquitetura menos complexa, facilitando tanto o desenvolvimento quanto a escalabilidade.

A comunicação entre os sistemas existe de forma padronizada e bem definida, seja em operações que resultaram em sucesso ou erro, refletindo as alterações de forma imediata em todas as operações disponíveis: sugestão de pontos de recarga, reservas, ocupação das estações, disponibilidade e pagamento.

Por fim, destaca-se o algoritmo de distribuição de demanda pelas estações de recarga, pensado para direcionar os carros sempre para os pontos disponíveis e mais próximos ou para os que estão com menores filas de espera, reduzindo tanto a demanda por cada uma das estações como o tempo de espera para atendimento.

5. Conclusão

De modo geral, a temática e especificação do problema foram bastante interessantes, pois, com um tema bastante atual no cenário mundial de transição energética, foi possível trabalhar com os tópicos mais essenciais de sistemas distribuídos. Poder lidar e se familiarizar com a camada mais básica do sistema operacional para trabalhar com redes, entender tópicos como endereçamento, envio de pacotes, o protocolo TCP, como processos se comunicam via rede e a importância das portas para identificar e encaminhar corretamente essa comunicação.

Houve também a oportunidade de trabalhar com Docker para encapsular a aplicação em um ambiente controlável e reproduzível, de modo a facilitar tanto o desenvolvimento quanto a distribuição dos sistemas. É por sistemas, as três aplicações desenvolvidas precisavam se comunicar, criando desafios para manter a confiabilidade do sistema, lidar com erros que possam acontecer em diversos cenários.

5.1. Melhorias

A UX da aplicação poderia ser melhorada, nossa percepção foi que, em alguns cenários, a usabilidade fica um pouco confusa, seja por opções que não ficaram tão claras nas interfaces ou em situações em que um erro tenha acontecido. Pois, embora sempre haja tratamento dos possíveis erros durante a execução da aplicação, nem sempre esse tratamento é informado ao usuário em forma de feedback, novamente causando problemas de usabilidade.

Referências

DIO. Arquitetura cliente-servidor: fundamentos e aplicações. https://www.dio.me/articles/arquitetura-cliente-servidor-fundamentos-e-aplicacoes/. Acesso em: abril de 2025.

Docker, Inc. (2024). Networking using the host network. https://docs.docker.com/engine/network/drivers/host/. Acesso em: abril de 2025.

ESR. Arquitetura tcp/ip: conceitos básicos. https://esr.rnp.br/administracao-e-projeto-de-redes/arquitetura-tcp-ip/. Acesso em: abril de 2025.

Passei Direto. Modelo tcp/ip. https://www.passeidireto.com/arquivo/122833487/tcp-ip. Acesso em: abril de 2025.

Tanenbaum, A. S. and Wetherall, D. J. (2011). *Redes de Computadores*. Pearson Prentice Hall, São Paulo, 5 edition.