Implementação de Sistema de Venda de Passagens Aéreas para Companhias Low-Cost com Comunicação TCP/IP e Infraestrutura em Docker

Davi Jatobá Galdino, Gabriel Sena Barbosa

Universidade Estadual de Feira de Santana (UEFS)
Av. Transnordestina, s/n, Novo Horizonte - BA, 44036-900

ddavijatoba33@gmail.com, gabriel.sena.barbosa@gmail.com

Abstract: The low-cost carrier (LCC) sector has transformed the air transportation industry, making air travel more accessible to a larger audience. In this context, there is a growing need for a system that streamlines the ticket purchasing process, enabling users to select various flight segments. This work presents the development of a system that utilizes a TCP/IP-based API to facilitate ticket purchases. The system is designed with basic Socket API support and aims to manage multiple simultaneous requests, ensuring the integrity of reservations. Users will be able to purchase tickets efficiently, providing a seamless purchasing experience.

Resumo: O setor de aviação de baixo custo, ou low-cost carriers (LCCs), tem transformado a indústria do transporte aéreo, permitindo que mais pessoas tenham acesso a viagens a preços acessíveis. Nesse cenário, surge a necessidade de criar um sistema que facilite a compra de passagens, permitindo a seleção de diferentes trechos de voo. Este trabalho apresenta o desenvolvimento de um sistema que utiliza uma API baseada no subsistema de rede TCP/IP para permitir a compra de passagens. O sistema será implementado com suporte à API Socket básica e projetado para gerenciar múltiplas requisições simultâneas, garantindo a integridade das reservas. Os usuários poderão realizar compras de passagens de forma eficiente, proporcionando uma experiência de compra fluida.

1. Introdução

Nos últimos anos, o setor de aviação de baixo custo, ou low-cost carriers (LCCs), tem transformado a indústria do transporte aéreo, permitindo que um número maior de pessoas tenha acesso a viagens a preços acessíveis. A implementação de estratégias eficientes de redução de custos, como o uso de aeroportos secundários, maior utilização de aeronaves, tarifas sem serviços adicionais incluídos, e a automatização de operações simplificadas pela internet, não só democratizou o transporte aéreo como impulsionou o turismo e a conectividade global.

Nesse contexto, foi apresentado aos estudantes de TEC 502 (MI - Concorrência e Conectividade) o desafio de projetar e implementar um sistema que permita a compra de

passagens por meio da escolha de diferentes trechos de voo, considerando a preferência de quem realizar a compra primeiro. Ficou decidido que o sistema deve ser implementado com base no subsistema de rede TCP/IP, onde o protocolo de comunicação deve ser definido de acordo com uma API especificada pelas equipes de desenvolvimento. Esta API deverá ser programada em qualquer linguagem com suporte à API Socket básica do TCP/IP, visando oferecer as funcionalidades necessárias para a comunicação entre os clientes e o servidor.

Além de lidar com as questões técnicas da comunicação entre cliente e servidor, o projeto deve ser capaz de gerenciar múltiplas requisições simultâneas, assegurando que os primeiros clientes a realizarem suas compras tenham prioridade nos trechos selecionados, mantendo a integridade das suas reservas.

A abordagem adotada pelas equipes de desenvolvimento seguiu a metodologia Problem-Based Learning (PBL), que se concentra na resolução de problemas reais como forma de aprendizado. Este relatório deverá documentar o processo de desenvolvimento, apresentando as decisões tomadas e os conceitos teóricos que fundamentam a programação de um software desse tipo, evidenciando a conexão entre teoria e prática ao longo do projeto.

2. Metodologia

2.1 Interpretação do problema

Na primeira sessão, foi realizada a leitura inicial do problema fornecido, seguida por um levantamento de ideias e questões preliminares. Foram levantados questionamentos sobre o desenvolvimento de sistemas de comunicação cliente-servidor, com foco na utilização de sockets TCP/IP, para direcionar o estudo e a projeção do sistema.

Para nivelar as equipes e esclarecer o início do projeto, foi estabelecida uma meta de investigação sobre tópicos fundamentais, incluindo o funcionamento das companhias low-cost (LCC's) na compra e venda de passagens, os princípios da rede TCP/IP, a estrutura de APIs baseadas em sockets, o uso de Docker para contêinerização e as camadas do protocolo TCP/IP.

O planejamento da nossa equipe incluiu a criação de uma interface de navegador simples e intuitiva, que faz solicitações ao servidor seguindo um protocolo desenvolvido por nós. Optamos por uma aplicação stateless, onde o cliente gera telas com base nas respostas do servidor. A interface oferece conexão com o servidor e a opção de compra de rotas com base na origem e destino, permitindo ao cliente escolher entre os 10 melhores trechos (com base na distância). Após a confirmação da compra, o cliente é informado se a transação foi bem-sucedida ou se a rota está indisponível, com a opção de retornar ao menu para tentar novamente. Esse fluxo foi pensado para garantir uma linha de compra simplificada.

2.2 Desenvolvimento de protótipos de funcionamento

Para entender os problemas e solucioná-los em etapas simplificadas, foram estabelecidas metas de desenvolvimento para comunicações cliente-servidor. A primeira implementação

consistiu em uma comunicação com um cliente conectado, permitindo a troca de mensagens por meio de entradas do usuário, até que um dos lados decidisse encerrar a comunicação.

Em seguida, o sistema foi ampliado para suportar múltiplos clientes simultaneamente, utilizando sockets TCP/IP. Essa nova versão permite que o servidor aceite conexões de vários clientes, gerenciando cada um em uma thread separada. Os clientes podem interagir com uma lista compartilhada de cidades brasileiras, solicitando a remoção de itens por meio de índices. O servidor valida essas solicitações e atualiza todos os clientes com as alterações, garantindo que todos tenham acesso a informações atualizadas em tempo real.

3. Resultados

3.1. Cliente

O cliente é implementado com a biblioteca Tkinter, criando uma interface gráfica chamada "Navegador" que solicita ao usuário o IP e a porta do servidor para estabelecer a conexão. Após a conexão, o cliente solicita o menu principal ao servidor. O cliente envia requisições em formato JSON, contendo um método e os dados necessários. Com base na resposta do servidor, a interface se atualiza dinamicamente, renderizando layouts apropriados, como menus e opções de escolha, seguindo o protocolo proposto. Seguem as funções do cliente:

limpar_navegador(navegador): Remove todos os componentes visuais da interface gráfica para preparar a janela para uma nova renderização.

enviar_requisicao(method, data): Formata a requisição do cliente, enviando-a via socket e lidando com a resposta do servidor. A resposta é interpretada e os elementos de interface são renderizados na janela.

iniciar_conexao(): Coleta o IP e a porta inseridos pelo usuário, estabelece a conexão com o servidor e faz a solicitação para carregar o menu principal, com validações de entrada.

iniciar_navegador(): Configura a janela principal do Tkinter, permitindo ao usuário inserir os dados de conexão e iniciando o loop principal do Tkinter para interações.

3.2. Servidor

O servidor foi desenvolvido para processar múltiplas requisições de clientes simultaneamente, utilizando *threads*. Ele inicia ao criar um socket TCP/IP e escuta conexões em uma porta específica. A cada nova conexão, uma *thread* é gerada para tratar o cliente de forma independente, permitindo que o servidor atenda vários usuários ao mesmo tempo.

O servidor recebe requisições contendo o método a ser executado (ex: menu principal, escolha de destino, etc.) e dados adicionais, como origem e destino. Dependendo do método solicitado, uma função correspondente é chamada para processar a requisição.

As principais funções implementadas são:

retorna_menu_principal(): Retorna a interface inicial, com opções de navegação.

retorna_escolha_destino(): Oferece dois *dropdowns* para seleção de origem e destino.

retorna_trechos_disponiveis(data): Calcula e exibe as melhores rotas entre as cidades escolhidas, com base em distâncias e disponibilidade de passagens.

retorna_confirmacao_rota(data): Finaliza a compra, atualizando o estoque de passagens e enviando uma mensagem de confirmação.

Cada resposta do servidor é estruturada em formato JSON, contendo os elementos da interface e as opções de navegação. Após o envio da resposta, a *thread* encerra a conexão com o cliente.

3.3 Protocolo de Comunicação

O protocolo de comunicação entre o cliente e o servidor foi implementado utilizando o padrão TCP/IP. A troca de dados entre cliente e servidor segue um formato de requisição e resposta, onde o cliente envia uma requisição contendo um método e, opcionalmente, dados adicionais, enquanto o servidor responde com uma estrutura JSON que contém as informações necessárias para a interface do usuário.

A requisição do cliente segue um padrão simples, onde são enviados dois campos principais:

- method: Define a ação que o cliente deseja realizar, como acessar o menu principal, escolher uma rota, ou finalizar uma compra (para chamar o método correspondente no servidor).
- data: Um dicionário opcional que contém os dados necessários para o método escolhido, como as cidades de origem e destino ou a rota selecionada para compra.

Exemplo de requisição:

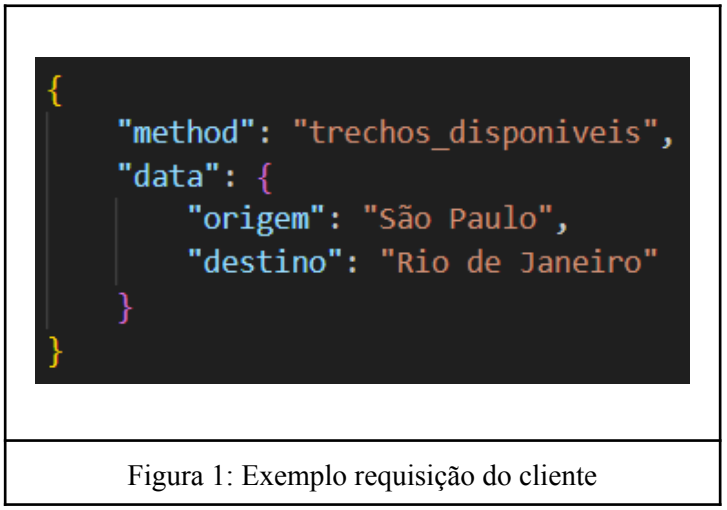

Figura 1: Exemplo requisição do cliente

Figura 1 mostra uma requisição pedindo que o servidor retorne as rotas disponíveis entre as cidades de São Paulo e Rio de Janeiro.

Resposta do Servidor

O servidor responde com uma estrutura JSON que descreve a interface do usuário e as informações solicitadas. A resposta é composta por uma lista de elementos de interface, como botões, *dropdowns*, e mensagens, que são exibidos no lado do cliente.

Principais componentes da resposta:

- **page_layout:** Uma lista de componentes da interface, como botões, *dropdowns*, e mensagens. Cada item possui informações como rótulos e o método a ser executado ao interagir com o elemento.

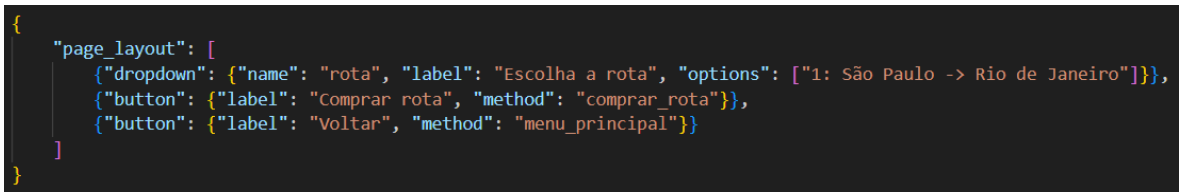

Figura 2: Exemplo resposta do servidor

3.4. Testes

Nos primeiros testes realizados com os protótipos, descritos no tópico 2.2 *Desenvolvimento de protótipos de funcionamento*, foi possível conectar dois clientes simultaneamente ao servidor, cada um em máquinas diferentes, simulando cenários reais de compra de passagens. Esses testes iniciais tinham como objetivo validar a capacidade do sistema de gerenciar múltiplas conexões, verificar a integridade das informações compartilhadas e garantir que atualizações, como a redução de passagens disponíveis, fossem refletidas em tempo real para todos os clientes conectados.

Após a finalização do projeto, enfrentamos problemas para rodar o cliente no container Docker, pois era necessária uma autorização no laboratório para que o Linux pudesse gerar a interface gráfica utilizando a biblioteca Tkinter. Para contornar essa situação, criamos um arquivo simulador_cliente, que automatiza as requisições e realiza a compra de uma rota sem a necessidade da interface gráfica. Dessa forma, conseguimos executar a compra de múltiplos clientes no servidor simultaneamente, utilizando containers Docker para simular diferentes máquinas conectadas.

O simulador realiza as mesmas operações que o cliente tradicional, porém executa as requisições de forma sequencial com dados previamente configurados.

3.5 Instruções para execução do sistema

Construção da Imagem Docker do Servidor: No terminal, navegue até o diretório onde o Dockerfile está localizado e execute o comando para criar a imagem: **docker build -t servidor_vendepass-app .**

Construção da Imagem Docker do Cliente: No terminal, navegue até o diretório onde o Dockerfile do cliente está localizado e execute o comando para criar a imagem: **docker build -t cliente_vendepass-app .**

Execução Manual do Servidor: Após a imagem ser construída, o servidor pode ser iniciado manualmente com o seguinte comando: **docker run --name servidor -p 61582:61582 servidor_vendepass-app**

Exercução Manual do Cliente: Após a imagem ser construída, o cliente pode ser iniciado manualmente com o seguinte comando: **docker run --name cliente1 -e SERVER_IP= servidor -e SERVER_PORT= 61582 -p 61583:61583 cliente_vendepass-app**

Execução múltiplos clientes: Para executar mais clientes, repita o comando anterior, mudando o nome do container e a porta, por exemplo: **docker run --name cliente2 -e SERVER_IP= servidor -e SERVER_PORT=61582 -p 61584:61584 cliente_vendepass-app**

Uso do Docker Compose: Para iniciar todos os serviços (servidor e 6 clientes), execute o comando a seguir no terminal, dentro do diretório que contém o docker-compose.yml: **docker-compose up**

3.6 Armazenamento das rotas do servidor

O sistema utiliza um arquivo local, chamado "distancias.plk", para armazenar as distâncias entre as cidades e a quantidade de passagens disponíveis para cada rota. O armazenamento é feito com a biblioteca pickle, que salva e carrega os dados em formato binário.

Quando o servidor inicia, ele tenta carregar as informações do arquivo. Se o arquivo não existir, um conjunto padrão de rotas e passagens é usado, com as distâncias entre as cidades e as quantidades iniciais de bilhetes disponíveis. Em seguida, os dados são salvos no arquivo para garantir que a próxima execução do servidor mantenha as informações atualizadas.

4. Conclusão

O sistema desenvolvido para a venda de passagens aéreas de companhias low-cost foi implementado com uma arquitetura em sockets TCP/IP, oferecendo uma interface gráfica simples para interação do usuário. Foram criadas funcionalidades que permitem a gestão de rotas e passagens, além da compra de bilhetes. O uso de JSON como protocolo de

comunicação trouxe a vantagem de compatibilidade ampla, já que é um formato muito utilizado atualmente.

Embora o sistema esteja funcional, algumas funcionalidades mais avançadas, como o aprimoramento da interface gráfica e validações detalhadas nas operações de compra, ainda podem ser implementadas. Durante o processo, o aprendizado incluiu a implementação de sistemas de comunicação em rede, gerenciamento de dados, criação de uma interface gráfica e uso de estruturas de dados para garantir o bom funcionamento do sistema. Esse conhecimento pode ser aplicado em outros sistemas distribuídos e em projetos que exigem comunicação cliente-servidor.

5. Referências

PYTHON Software Foundation. Python documentation: socket — Low-level networking interface. Disponível em: https://docs.python.org/3/library/socket.html. Acesso em: set. 2024.

MOZILLA DEVELOPER NETWORK. HTTP - Visão geral. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/HTTP. Acesso em: set. 2024.

DOCKER INC. O que é Docker?. Disponível em: https://www.docker.com/what-docker. Acesso em: set. 2024.

BÓSON TREINAMENTOS. Cliente e Servidor com Módulo Socket em Python - Exemplo. Disponível em: https://www.youtube.com/watch?v=vbUuJ2_6wqs. Acesso em: set. 2024.

MATEUS MULLER. SOCKET: O que é um SOCKET DE REDE? SOCKETS com PYTHON! Disponível em: https://www.youtube.com/watch?v=aV4p6f2MuJc. Acesso em: set. 2024.

FABRICIO VERONEZ. Docker do zero ao compose: Parte 01. Disponível em: https://www.youtube.com/watch?v=GkMJJkWRgBQ. Acesso em: set. 2024.

PYTHON Software Foundation. *pickle — Python object serialization*. Disponível em: https://docs.python.org/3/library/pickle.html. Acesso em: set. 2024.