Relatório

Contextualização

O presente relatório tem como objetivo apresentar e esclarecer os detalhes da primeira atividade extraclasse da disciplina Programação para Sistemas Paralelos e Distribuídos, ministrada pelo Dr. Fernando William Cruz na Universidade de Brasília.

A atividade propõe um aprofundamento nos conceitos de comunicação entre microsserviços, com foco na utilização do protocolo gRPC e da biblioteca Protobuf, estabelecendo uma comparação com o modelo tradicional de comunicação Rest-API/JSON.

Além disso, busca-se promover o desenvolvimento de conhecimentos sobre Docker e Kubernetes, com ênfase nos processos de conteinerização e orquestração dos microsserviços implementados.

Assim, os alunos da turma 01 responsáveis pela execução dassa atividade foram:

Nome Completo	Matrícula
FLÁVIO GUSTAVO ARAÚJO DE MELO	211030602
GUILHERME SILVA DUTRA	221021984
GUSTAVO FRANCA BOA SORTE	211030774
HARRYSON CAMPOS MARTINS	211039466

Introdução

A proposta da atividade solicita a construção de uma aplicação composta por três módulos interdependentes (P, A e B), além de um cliente web (Web Client). O módulo P atua como uma API Gateway, responsável por expor uma interface RESTful ao cliente final e intermediar as requisições HTTP externas enviadas pelo Web Client, traduzindo-as em chamadas gRPC internas aos demais módulos.

Adicionalmente, os módulos A e B são implementados como microserviços independentes, desenvolvidos em linguagens distintas do serviço P, com o objetivo de prover funcionalidades complementares e cooperar entre si para atender às demandas processadas por P. Dessa forma, essa arquitetura busca reproduzir cenários reais de sistemas distribuídos, nos quais diferentes serviços interagem de forma colaborativa para compor uma aplicação unificada, garantindo escalabilidade, modularidade e reutilização de componentes.

Além disso, a atividade envolve o estudo e a configuração de um ambiente de orquestração local utilizando o Minikube, ferramenta que per-

mite a execução de um cluster Kubernetes em host único. Nesse ambiente, os módulos P, A e B são implantados em contêineres Docker e gerenciados pelo Kubernetes, enquanto o Web Client atua como ponto de acesso externo à aplicação, simulando o comportamento de um usuário real interagindo com o sistema via navegador. Dessa maneira, é possível reproduzir de forma prática o ciclo completo de comunicação e deploy de uma aplicação distribuída em nuvem.

Outrossim, o restante do documento está organizado em seções que abordam, inicialmente, uma revisão conceitual sobre gRPC, seus componentes e modos de comunicação, com exemplos práticos. Em seguida, detalha o desenvolvimento da aplicação distribuída, incluindo a arquitetura adotada, as funcionalidades implementadas e um comparativo de desempenho entre uma versão baseada em gRPC e outra utilizando REST/JSON. Adicionalmente, também é descrito a configuração do ambiente Kubernetes, os arquivos de configuração utilizados e os passos para replicação da infraestrutura. Por fim, o relatório apresenta as conclusões do grupo e as considerações individuais sobre o aprendizado e as dificuldades enfrentadas durante a realização do projeto.

Entendendo o gRPC

O que é o gRPC?

O gRPC (Google Remote Procedure Calls) é um framework moderno e de código aberto criado pela Google para viabilizar chamadas de procedimentos remotos (RPC). Diferentemente das APIs REST tradicionais, que utilizam um modelo de requisição/resposta baseado em texto, como JSON, o gRPC foi concebido para oferecer alta performance, eficiência e confiabilidade na comunicação entre serviços, seja em arquiteturas de microserviços, na integração de aplicações móveis com backends ou em sistemas distribuídos complexos.

O objetivo central do gRPC é permitir que a interação entre cliente e servidor — mesmo que implementados em linguagens diferentes — seja tão simples quanto invocar uma função local. Essa abordagem abstrai a complexidade da comunicação de rede, facilitando o desenvolvimento e permitindo que os programadores concentrem seus esforços na lógica de negócio.

Protocol Buffers

No núcleo do gRPC está o Protocol Buffers (Protobuf), um método de serialização de dados independente de linguagem e plataforma. Diferente de formatos legíveis, como JSON ou XML, o Protobuf organiza os dados em um formato binário compacto. Esse formato é

definido em arquivos .proto, nos quais se descrevem as mensagens (estruturas de dados) e os serviços (interfaces com os métodos que podem ser invocados remotamente).

A principal vantagem do Protobuf é sua eficiência: os dados serializados ocupam muito menos espaço e são processados com maior rapidez do que os formatos de texto equivalentes. Isso reduz consideravelmente o consumo de largura de banda e de recursos de CPU, sendo especialmente importante para sistemas que exigem alta escalabilidade e baixa latência.

Abaixo estão os principais elementos que compõem um arquivo .proto:

1. **Sintaxe**: Define a versão do Protobuf utilizada (geralmente "proto2" ou "proto3").

```
syntax = "proto3";
```

2. **Pacote**: Especifica o namespace do Protobuf, ajudando a organizar os arquivos e evitar conflitos de nomes entre diferentes serviços.

```
package example;
```

3. **Mensagens**: Definem as estruturas de dados que serão trocadas entre cliente e servidor. Cada campo dentro de uma mensagem possui um tipo (como int32, string, bool) e um identificador numérico único.

```
message HelloRequest {
  string name = 1;
}
message HelloReply {
  string message = 1;
}
```

- Campos: Cada campo possui:
  - Tipo: Define o tipo de dado (ex.: string, int32, float, etc.).
  - Nome: Nome do campo.
  - Número: Identificador único do campo no formato binário.

4. **Serviços**: Define uma interface que contém os métodos (RPCs) que podem ser chamados remotamente. Cada método especifica os tipos de entrada e saída.

```
service Communicator {
  rpc UnaryHello(HelloRequest) returns (HelloReply);
}
```

- rpc: Representa uma chamada de procedimento remoto (Remote Procedure Call). Define:

- Nome do método: Nome da função remota.
- Entrada: Tipo de mensagem enviada pelo cliente.
- Saída: Tipo de mensagem retornada pelo servidor.

5. Tipos de dados suportados: O Protobuf suporta diversos tipos de dados, incluindo tipos primitivos (int32, int64, float, double, bool, string, bytes) e tipos compostos (mensagens aninhadas, listas, mapas).

```
message Example {
  int32 id = 1;          // Número inteiro
  string name = 2;       // Cadeia de caracteres
  repeated string tags = 3; // Lista de strings
}
```

6. **enumerações** (enums): Permitem definir um conjunto fixo de valores nomeados, facilitando a representação de estados ou categorias.

```
enum Status {
    UNKNOWN = 0;
    ACTIVE = 1;
    INACTIVE = 2;
}
```

HTTP/2

Para transportar esses dados binários de forma eficiente, o gRPC se apoia no HTTP/2 como camada de transporte. Essa escolha vai além de simplesmente gerenciar requisições e respostas: o HTTP/2 oferece funcionalidades avançadas que o gRPC explora plenamente, como multiplexação (permitindo o envio de várias requisições e respostas simultaneamente em uma única conexão TCP, eliminando o bloqueio conhecido como Head-of-Line), compressão de cabeçalhos (reduzindo ainda mais a sobrecarga) e streams bidirecionais. Essa última funcionalidade é especialmente poderosa, pois possibilita que cliente e servidor mantenham uma conexão contínua e troquem múltiplas mensagens de forma assíncrona e em qualquer ordem. Isso habilita cenários complexos, como comunicação em tempo real e streaming de dados.

Tipos de comunicações suportadas pelo gRCP

Antes de propriamente avançarmos faz-se necessário especificar o arquivo .proto utilizado a fim de contribuir no esclarecimento dos exemplos que serão apresentados:

```
syntax = "proto3";

package example;

service Communicator {

  // Unary call
  rpc UnaryHello(HelloRequest) returns (HelloReply);

  // Server-streaming call
  rpc SplitWords(TextRequest) returns (stream WordReply);

  // Client-streaming call
  rpc Average(stream NumberRequest) returns (AverageReply);

  // Bidirectional streaming call
  rpc Chat(stream ChatMessage) returns (stream ChatMessage);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}

message TextRequest {
  string text = 1;
}

message WordReply {
  string word = 1;
}

message NumberRequest {
   float number = 1;
}

message AverageReply {
  float average = 1;
}

message ChatMessage {
  string sender = 1;
  string text = 2;
```

}

Chamda Unária (Unary Call)

A chamada unária é o modelo de comunicação mais simples e familiar, similar a uma requisição HTTP REST tradicional. Neste modelo, o cliente envia uma única requisição para o servidor e recebe uma única resposta de volta. É um paradigma síncrono e bloqueante, onde o cliente aguarda a resposta do servidor para dar continuidade ao processo. Este tipo é ideal para operações que são naturalmente baseadas em uma ação e uma reação imediata, como autenticação de um usuário (envia credenciais, recebe um token), consulta de um dado específico por ID ou processamento de um pagamento.

Para tanto, foi desenvolvido um exemplo simples que demonstra esse tipo de comunicação. Nele, podemos observar claramente o padrão request-response da chamada unária:

Lado cliente:

```
print("Unary:")
name = input("Enter your name: ")    # Cliente prepara a requisição
response = stub.UnaryHello(example_pb2.HelloRequest(name=name))          # Envia UMA requisição
print(response.message)        # Aguarda e recebe UMA resposta
```

Lado servidor:

```
def UnaryHello(self, request, context):
    return example_pb2.HelloReply(message=f"Hello, {request.name}!\nWelcome") # Processa e recebe uma resposta
```

Nesta implementação, fica evidente a natureza síncrona e bloqueante da chamada unária. O cliente envia uma única mensagem contendo o nome do usuário através do objeto HelloRequest e bloqueia a execução, aguardando necessariamente que o servidor processe a requisição e retorne uma única resposta do tipo HelloReply. O servidor, por sua vez, recebe a requisição completa, processa a lógica de negócio (nesse caso, a simples formatação de uma mensagem de saudação) e envia a resposta de volta de uma vez, finalizando a transação.

Streaming do Servidor (Server Streaming RPC)

No streaming do servidor, o cliente envia uma única requisição e, em contrapartida, recebe um fluxo (stream) de múltiplas mensagens como resposta.O servidor mantém a conexão aberta e pode enviar dados de forma contínua até que a transação seja finalizada. Este modelo é perfeito para cenários onde o servidor precisa transmitir uma grande quantidade de dados que são gerados ou buscados de forma progressiva. Exemplos clássicos incluem notificações em tempo real

(como alertas de sistema), o envio de um arquivo grande em pedaços (chunks) ou o recebimento de atualizações em tempo real de uma cotação de ações.

Em relação a essa comunicação, no exemplo desenvolvido o cliente envia uma única requisição contendo uma frase completa, e o servidor responde com um fluxo de mensagens individuais, onde cada palavra é enviada sequencialmente:

Lado cliente:

```
print("\nServer streaming:")
text_buffer = input("Enter a sentence: ")   # Cliente envia UMA requisição
print("Server response:")
for reply in stub.SplitWords(example_pb2.TextRequest(text=text_buffer)):
    print(reply.word)     # Recebe MÚLTIPLAS respostas em streaming
```

Lado servidor:

```
def SplitWords(self, request, context):
    words = request.text.split()
    for word in words:
        time.sleep(0.5)     # Simula processamento ou delay
        yield example_pb2.WordReply(word=word.upper())   # Retorna múltipla respostas via
        yield
```

Esta implementação demonstra elegantemente as características essenciais do server streaming. O cliente faz uma única chamada ao método SplitWords, mas em vez de receber uma resposta única, entra em um loop que itera sobre um fluxo contínuo de respostas. Cada palavra da frase original é processada e enviada individualmente pelo servidor, com um delay de 0.5 segundos entre elas, simulando um processamento progressivo ou a transmissão de dados que são gerados em tempo real.

Streaming do Cliente (Client Streaming RPC)

No streaming do cliente, a dinâmica se inverte: o cliente envia um fluxo (stream) de múltiplas mensagens para o servidor e, ao finalizar o envio, recebe uma única resposta. Isso permite que o cliente faça um upload de dados de forma eficiente e contínua, sem precisar acumular tudo na memória antes de enviar. O servidor, por sua vez, processa todo o fluxo de dados recebido e retorna uma única resposta de confirmação ou agregação. Um uso comum é o upload de um arquivo muito grande, onde o cliente envia os pedaços sequencialmente. Outro exemplo é a agregação de métricas de diversos dispositivos IoT, onde o servidor consolida todos os dados recebidos em um único relatório.

Assim, no exemplo proposto, temos uma implementação clássica de client streaming que demonstra exatamente a inversão de papéis de-

scrita conceitualmente. O cliente envia um fluxo contínuo de números para que o servidor calcule e retorne uma única resposta com a média aritmética:

Lado cliente:

```
print("\nClient streaming:")
print("Enter numbers (or 'end' to finish):")

def generate_numbers():
    while True:
        user_input = input("> ")   # Cliente coleta dados continuamente
        if user_input.lower() in ["end", "exit", "quit"]:
            break
        try:
            num = float(user_input)
            yield example_pb2.NumberRequest(number=num)   # Envia múltiplas requisições via yield
        except ValueError:
            print("Invalid input! Please enter a valid number.")

response = stub.Average(generate_numbers())      # Recebe UMA resposta final
print(f"\nServer calculated average: {response.average:.2f}")
```

Lado Servirdor:

```
def Average(self, request_iterator, context):
    total = 0
    count = 0
    for req in request_iterator:     # Processa o fluxo de requisições
        total += req.number
        count += 1
        print(f"Received: {req.number}")
    avg = total / count if count > 0 else 0
    print(f"Final average: {avg:.2f}")
    return example_pb2.AverageReply(average=avg)       # Retorna UMA resposta consolidada
```

Esta implementação captura perfeitamente a essência do client streaming. O cliente utiliza um generator (generate_numbers()) para enviar números de forma assíncrona e contínua, mantendo o controle sobre quando iniciar e finalizar o envio dos dados. O servidor, por sua vez, recebe um iterador (request_iterator) que lhe permite processar cada número individualmente à medida que chega, acumulando os valores para o cálculo final.

Streaming Bidirecional (Bidirectional Streaming RPC)

Já o streaming bidirecional é o modelo mais flexível e complexo, permitindo que cliente e servidor enviem um fluxo de mensagens de

forma independente e assíncrona. As duas pontas da comunicação podem ler, escrever e operar simultaneamente, sem uma ordem predeterminada. Essa capacidade é habilitada pelo HTTP/2, que permite a multiplexação em uma única conexão. Este padrão é extremamente poderoso para aplicações que exigem interação em tempo real e comunicação contínua. Casos de uso notáveis incluem sistemas de chat em grupo, jogos multiplayer onde a posição de vários jogadores é atualizada constantemente, ou aplicações de negociação em bolsa, onde ambas as partes estão constantemente enviando e recebendo ordens de compra e venda.

Logo, para tal comunicação o exemplo desenvolvido foi um sistema de chat em tempo real que demonstra toda a potência e complexidade do streaming bidirecional. A implementação cria um canal de comunicação totalmente assíncrono onde cliente e servidor podem trocar mensagens livremente e simultaneamente:

Lado cliente:

```
def generate_messages():
    while not closed.is_set():
        with lock:
            while messages:
                yield messages.pop(0)  # Envia mensagens do cliente

def read_input():
    while not closed.is_set():
        text = input()  # Captura input do usuário em tempo real
        messages.append(example_pb2.ChatMessage(sender=name, text=text))

def receive(responses):
    for response in responses:
        print(f"({response.sender}): {response.text}")  # Recebe mensagens do servidor

# Execução paralela dos fluxos
threading.Thread(target=read_input, daemon=True).start()
receive(responses)
```

Lado servidor:

```
def Chat(self, request_iterator, context):
    def receive():
        for msg in request_iterator:  # Recebe mensagens do cliente
            print(f"({msg.sender}) {msg.text}")

    def send():
        while not closed.is_set():
            text = input("")  # Captura input do servidor
```

```
                messages_to_send.append(example_pb2.ChatMessage(sender="Server", text=text))

    # Threads para receber e enviar simultaneamente
    threading.Thread(target=receive, daemon=True).start()
    threading.Thread(target=send, daemon=True).start()

    while not closed.is_set():
        with lock:
            while messages_to_send:
                yield messages_to_send.pop(0)    # Envia mensagens do servidor
```

Nesta implementação através do uso de threads paralelas tanto no cliente quanto no servidor, conseguimos a verdadeira independência dos fluxos de comunicação - o cliente pode digitar e enviar mensagens enquanto simultaneamente recebe e exibe mensagens do servidor, e vice-versa.

A chave está na multiplexação do HTTP/2 que permite que ambos os streams (cliente→servidor e servidor→cliente) coexistam na mesma conexão TCP sem bloqueio mútuo. O generator generate_messages() no cliente e o loop while com yield no servidor mantêm os canais abertos continuamente, permitindo que mensagens sejam enviadas e recebidas em qualquer ordem, a qualquer momento.

OBS: Os arquivos utilizados nessa apresentação podem ser controntrados na pasta *conhecendo_grpc*

Detalhes da Aplicação Desenvolvida

A atividade consistiu na implementação de um sistema de microsserviços para a compra de peças de carros, utilizando tanto o protocolo gRPC com Protobuf quanto o modelo HTTP/REST. O sistema é composto por três microsserviços principais: o Microsserviço A, responsável por fornecer as opções de peças disponíveis para cada modelo de carro; o Microsserviço B, encarregado do cálculo do preço total e no frete com base nas peças selecionadas; e o Microsserviço P, que atua como intermediário entre os clientes (Web Client) e os outros dois microsserviços.

O Microsserviço P é o ponto de entrada para os clientes, recebendo requisições HTTP/REST e convertendo-as para o formato Protobuf antes de encaminhá-las aos Microsserviços A e B. Após receber as respostas em Protobuf, o Microsserviço P converte os dados de volta para HTTP/REST e os envia aos clientes.

Sobre o gRPC

A seguir, detalhamos as funcionalidades e a comunicação entre os microsserviços com ênfase no uso do gRPC:

Microsserviço A

O Microsserviço A é responsável por fornecer as opções de peças disponíveis para cada modelo de carro. Ele expõe um endpoint gRPC que recebe o modelo do carro como parâmetro e retorna uma lista de peças compatíveis. A comunicação entre o Microsserviço P e o Microsserviço A é realizada através de chamadas gRPC unárias, onde o Microsserviço P envia uma requisição com o modelo do carro e aguarda a resposta com as peças disponíveis.

Microsserviço B

O Microsserviço B é encarregado do cálculo do preço total e do frete com base nas peças selecionadas pelo usuário. Ele também expõe um endpoint gRPC que recebe uma lista de peças e retorna o preço total calculado. A comunicação entre o Microsserviço P e o Microsserviço B é realizada através de chamadas gRPC unárias, onde o Microsserviço P envia a lista de peças selecionadas e aguarda a resposta com o preço total.

Microsserviço P

O Microsserviço P atua como intermediário entre os clientes e os outros dois microsserviços. Ele recebe requisições HTTP/REST dos clientes, converte os dados para o formato Protobuf e encaminha as requisições para os Microsserviços A e B. Após receber as respostas em Protobuf, o Microsserviço P converte os dados de volta para HTTP/REST e os envia aos clientes. O Microsserviço P também é responsável por gerenciar o estado da sessão do usuário, garantindo que as peças selecionadas sejam corretamente associadas ao modelo de carro escolhido.

Comparativo com REST/JSON

Para avaliar o desempenho e a eficiência do gRPC em comparação com o modelo tradicional de comunicação REST/JSON, foi implementada uma versão alternativa do sistema utilizando apenas HTTP/REST. Nessa versão, o Microsserviço P se comunica com os Microsserviços A e B através de requisições HTTP/REST, enviando e recebendo dados no formato JSON.

Metodologia de Teste

Para garantir uma comparação justa entre as duas tecnologias, foram desenvolvidos cenários de teste idênticos para ambas as versões da aplicação. Os testes foram executados sob as mesmas condições de infraestrutura e carga, medindo o tempo de resposta de três operações principais:

1. **GetPeças**: Operação que consulta as peças disponíveis para um modelo de carro específico
2. **Calcular**: Operação que calcula o preço total das peças selecionadas
3. **Pagar**: Operação que processa o pagamento e finaliza a compra

Cada operação foi executada 10 vezes consecutivas para ambas as versões (gRPC e REST/JSON), registrando o tempo de resposta em milissegundos. Os dados de tempo de resposta foram coletados através das ferramentas de desenvolvedor do navegador famigerado inspecionar, especificamente na aba Network, que permite monitorar e medir com precisão o tempo de resposta das requisições HTTP. Os testes foram realizados em ambiente controlado, utilizando a mesma infraestrutura e configurações de rede.

Resultados dos Testes de Performance

Versão gRPC/Protobuf

Execução	GetPeças (ms)	Calcular (ms)	Pagar (ms)
1	11	3	3
2	10	4	3
3	21	6	5
4	9	4	4
5	11	3	3
6	12	3	7
7	9	5	3
8	5	3	4
9	10	4	7
10	12	3	3
Média	**11,0**	**3,8**	**4,2**

Versão REST/JSON

Execução	GetPeças (ms)	Calcular (ms)	Pagar (ms)
1	5	4	77

Execução	GetPeças (ms)	Calcular (ms)	Pagar (ms)
2	6	4	37
3	5	5	34
4	6	5	24
5	4	5	19
6	5	5	19
7	4	5	18
8	8	5	42
9	4	6	22
10	4	5	22
Média	**5,1**	**4,9**	**31,4**

Análise Comparativa

Operação	gRPC/Protobuf (ms)	REST/JSON (ms)	Diferença (%)	Vencedor
GetPeças	11,0	5,1	+115,7%	REST
Calcular	3,8	4,9	+28,9%	gRPC
Pagar	4,2	31,4	+647,6%	gRPC

Conclusões do Comparativo de Performance

Os resultados dos testes revelam um panorama interessante sobre o desempenho das duas tecnologias:

Vantagens do REST/JSON: - **Operação GetPeças**: O REST/JSON demonstrou superioridade significativa nesta operação, sendo aproximadamente 2,15 vezes mais rápido que o gRPC. Esta diferença pode ser atribuída à simplicidade da operação e ao overhead inicial do gRPC para estabelecer conexões HTTP/2.

Vantagens do gRPC/Protobuf: - **Operação Calcular**: O gRPC apresentou desempenho ligeiramente superior (22,4% mais rápido), demonstrando a eficiência da serialização binária do Protobuf para operações de processamento de dados. - **Operação Pagar**: Aqui o gRPC mostrou sua maior vantagem, sendo cerca de 7,5 vezes mais rápido que o REST/JSON. Esta diferença substancial sugere que o gRPC é particularmente eficiente em operações mais complexas que envolvem múltiplas etapas de processamento.

Considerações Técnicas:

1. **Sobrecarga Inicial**: O gRPC possui uma sobrecarga inicial maior devido ao estabelecimento de conexões HTTP/2 e à

negociação de protocolos, o que explica o desempenho inferior em operações simples como GetPeças.

2. **Eficiência em Operações Complexas**: Para operações mais elaboradas (como Pagar), o gRPC demonstra sua superioridade através da serialização binária eficiente e do reuso de conexões HTTP/2.
3. **Consistência de Performance**: O gRPC apresentou maior consistência nos tempos de resposta, especialmente nas operações Calcular e Pagar, enquanto o REST/JSON mostrou maior variabilidade, particularmente na operação Pagar.

Recomendações:

- **Para operações simples e isoladas**: REST/JSON pode ser mais adequado devido à sua simplicidade e menor overhead inicial.
- **Para sistemas com múltiplas operações complexas**: gRPC oferece vantagens significativas, especialmente quando há necessidade de comunicação frequente entre microsserviços.
- **Para arquiteturas de microsserviços de alta performance**: gRPC é recomendado devido à sua eficiência em cenários de comunicação intensiva e operações complexas.

O comparativo demonstra que a escolha entre gRPC e REST/JSON deve considerar o contexto específico da aplicação, com gRPC sendo mais vantajoso para sistemas que demandam alta performance em operações complexas e comunicação frequente entre serviços.

Kubernets

Nesta aplicação, utilizamos o Minikube, uma ferramenta que cria e gerencia um cluster Kubernetes completo, porém simplificado, que roda localmente na máquina.

O diferencial do Minikube, é que ele é um cluster que o Minikube cria é um cluster de nó único (host unico), enquanto um cluster de produção geralmente é composto por múltiplos nós (computadores)

1.1 Etapa Inicial - Containerização com Docker O desenvolvimento iniciou com a criação de **Dockerfiles individuais** para cada microserviço:

- **Microserviço A (Catálogo)**: ./Microservices/serverA-microsservice/Dockerfile
- **Microserviço B (Pricing)**: ./Microservices/serverB-microsservice/Dockerfile
- **API Gateway (P-API)**: ./P-Api/Dockerfile

Após a containerização individual, foi criado um **docker-compose.yml** para:
- **Orquestrar todos os serviços** em uma única rede (car-build-network)
- **Definir dependências** entre serviços (healthchecks)
- **Gerenciar volumes** para persistência do Post- greSQL
- **Testar a comunicação** entre containers antes da migração

---

2. Arquitetura Kubernetes Implementada

2.1 Minikube Escolha Tecnológica: Minikube foi selecionado por ser: - **Ambiente Local**: Ideal para desenvolvimento e testes - **Cluster Completo**: Simula um cluster Kubernetes real em nó único - **Facilidade de Uso**: Setup rápido sem configurações complexas

2.2 Componentes de Infraestrutura

A) Persistent Volume Claims (PVC)

```yaml
# postgres-pvc.yaml
kind: PersistentVolumeClaim
spec:
  resources:
    requests:
      storage: 1Gi
```

Justificativa de Uso:
- **Persistência de Dados**: Garante que dados PostgreSQL sobrevivam a reinicializações de pods
- **Desacoplamento**: Separa o lifecycle do storage do lifecycle do container
- **Portabilidade**: Permite migração entre nodes sem perda de dados

B) ConfigMaps

```yaml
# postgres-configmap.yaml
kind: ConfigMap
data:
  init.sql: |
    CREATE TABLE carros...
    CREATE TABLE pecas...
```

Justificativa de Uso:
- **Separação de Configuração**: Remove scripts SQL do código da aplicação

- **Versionamento**: Permite controle de versão das configurações
- **Reutilização**: Configurações podem ser compartilhadas entre ambientes

2.3 Camada de Deployments

A) PostgreSQL Deployment

```yaml
# postgres-deployment.yaml
kind: Deployment
spec:
  replicas: 1
  template:
    spec:
      containers:
      - image: postgres:15-alpine
        volumeMounts:
        - name: postgres-storage
          mountPath: /var/lib/postgresql/data
        - name: init-script
          mountPath: /docker-entrypoint-initdb.d
```

Nesta aplicação, utilizamos o Minikube, uma ferramenta que cria e gerencia um cluster Kubernetes completo, porém simplificado, que roda localmente na máquina.

O diferencial do Minikube, é que ele é um cluster que o Minikube cria é um cluster de nó único (host unico), enquanto um cluster de produção geralmente é composto por múltiplos nós (computadores)

Componentes e Manifestos

1. Camada de Dados (PostgreSQL)

postgres-pvc.yaml - Armazenamento Persistente

```yaml
kind: PersistentVolumeClaim
```

Função: Reserva 1GB de disco persistente no cluster
- **Por que?**: Dados do PostgreSQL precisam sobreviver a reinicializações
- **Com- portamento**: Mesmo se o pod morrer, os dados permanecem salvos

postgres-configmap.yaml - Script de Inicialização

```yaml
kind: ConfigMap
data:
  init.sql: |
    CREATE TABLE carros...
    CREATE TABLE pecas...
```

Função: Armazena o script SQL que cria tabelas e popula dados - **Execução**: Roda automaticamente na primeira inicialização do Post-

greSQL - **Conteúdo**: Tabelas carros e pecas + dados iniciais (Fusca, Civic, Corolla)

postgres-deployment.yaml - Container do Banco

```
kind: Deployment
image: postgres:15-alpine
```

Função: Executa o PostgreSQL no cluster
- **Volumes**: Conecta PVC criado (dados) + ConfigMap (script SQL)
- **Ambiente**: Define as var- iáveis de ambiente do banco. Essas credenciais são utilizadas tanto pelo contêiner do banco quanto pelos serviços que irão se conectar a ele.
- **Porta**: 5432 (exposta internamente no cluster para acesso pelas aplicações).

postgres-service.yaml - Rede Interna do Banco

```
kind: Service
type: ClusterIP
```

Função: Cria um serviço interno para expor o PostgreSQL dentro do cluster.
- **DNS interno**: Gera automaticamente o nome postgres-service, que pode ser usado por outros Pods para se conectar ao banco sem precisar do IP direto.
- **Acesso**: Direciona o tráfego recebido na porta 5432 para os Pods do Deployment do PostgreSQL.
- **Isolamento**: Por ser do tipo ClusterIP, o serviço só é acessível de dentro do cluster, garantindo que o banco não fique exposto externamente.

---

2. Camada de Microserviços (Containers A & B)

server-a-deployment.yaml - Microserviço de Catálogo

```
kind: Deployment
image: car_build-server-a:latest
env:
  - name: DB_HOST
    value: "postgres-service"
```

Função: Container A - Microserviço gRPC de catálogo
- **Responsabilidade**: Buscar peças por modelo de carro
- **Tecnologia**: Node.js + gRPC
- **Banco**: Conecta no PostgreSQL via DNS interno
- **Porta**: 50051 (gRPC)

server-a-service.yaml - DNS do Catálogo

```
kind: Service
type: ClusterIP
```

Função: Cria DNS server-a-service para acesso interno

server-b-deployment.yaml - Microserviço de Pricing

```
kind: Deployment
image: car_build-server-b:latest
```

Função: Container B
- Microserviço gRPC de cálculo de preços
- **Responsabilidade**: Calcular preços totais dos orçamentos
- **Tecnologia**: Node.js + gRPC
- **Porta**: 50052 (gRPC)

server-b-service.yaml - DNS do Pricing

```
kind: Service
type: ClusterIP
```

Função: Cria DNS server-b-service para acesso interno

---

3. Camada de API Gateway (Container P)

p-a pi-deployment.yaml - API Gateway

```
kind: Deployment
image: car_build-p-api:latest
env:
  - name: SERVER_A_HOST
    value: "server-a-service"
  - name: SERVER_B_HOST
    value: "server-b-service"
```

Função: Executa o API Gateway (Container P), que serve como ponte entre o frontend e os serviços internos.
- **Rede Externa**: Recebe requisições HTTP do frontend
- **Rede Interna**: Converte para gRPC e chama Server A/B
- **Ambiente**: Define variáveis (SERVER_A_HOST e SERVER_B_HOST) que apontam para os serviços internos no cluster, garantindo que o Gateway saiba como se comunicar com eles via DNS do Kubernetes.
- **Tecnologia**: Construído em FastAPI com clientes gRPC para orquestrar as chamadas. - **CORS**: Configurado para per- mitir que o frontend (mesmo rodando em localhost) consiga acessar a API sem bloqueios de navegador.

p-api-service.yaml - Exposição Externa

kind: Service
type: LoadBalancer

Função: **ÚNICA** Torna o API Gateway acessível de fora do cluster.
- **Tipo**: LoadBalancer abre uma tunel no nó do cluster, permitindo que usuários externos (como o frontend) façam requisições HTTP.
- **Comando**: minikube tunnel cria túnel
- **Resultado**: O frontend pode se conectar ao cluster chamando o Gateway diretamente, sem precisar conhecer os serviços internos.

Fluxo Completo de uma Requisição

1. Buscar Peças de um Carro:

1. [Frontend React]
   POST http://127.0.0.1:57153/get-pecas
   Body: {"modelo": "fusca", "ano": 2014}
   ↓

2. [minikube tunnel]
   Encaminha para cluster Kubernetes
   ↓

3. [Container P - FastAPI]
   - Recebe HTTP POST
   - Converte para gRPC
   - Chama server-a-service via DNS interno
   ↓

4. [Container A - Server gRPC]
   - Recebe gRPC GetPecas()
   - Conecta postgres-service via DNS
   - Query SQL: SELECT * FROM pecas WHERE modelo_fk = 'fusca'
   ↓

5. [PostgreSQL]
   - Busca dados no PVC persistente
   - Retorna: Chassi R$5000, Motor 1.6 R$3500, etc.
   ↓

6. [Resposta volta pelo caminho inverso]
   PostgreSQL → Server A → P-API → Tunnel → Frontend

2. Calcular Preço Total:

1. [Frontend] Envia peças selecionadas para /calcular
2. [Container P] Converte HTTP → gRPC
3. [Container B] Recebe lista de peças e quantidades
4. [Container B] Calcula: (quantidade × valor) para cada peça
5. [Resposta] Preço total retorna para frontend

Tipos de Services e Redes

Service	Tipo	Acesso	Função
postgres-service	ClusterIP	Apenas interno	DNS do banco
server-a-service	ClusterIP	Apenas interno	DNS do catálogo
server-b-service	ClusterIP	Apenas interno	DNS do pricing
p-api-service	LoadBalancer	Externo	Gateway público

Redes Configuradas:

- **Rede Externa**: Frontend ↔ P-API (HTTP tradicional)
- **Rede Interna**: P-API ↔ Server A/B (HTTP/2 gRPC)

Seção Completa sobre Kubernetes - Arquitetura e Funcionalidades

1. Processo de Desenvolvimento e Migração para Kubernetes

1.1 Etapa Inicial - Containerização com Docker
O desenvolvimento iniciou com a criação de **Dockerfiles individuais** para cada microserviço:

- **Microserviço A (Catálogo)**: ./Microservices/serverA-microsservice/Dockerfile
- **Microserviço B (Pricing)**: ./Microservices/serverB-microsservice/Dockerfile
- **API Gateway (P-API)**: ./P-Api/Dockerfile

Após a containerização individual, foi criado um **docker-compose.yml** para:
- **Orquestrar todos os serviços** em uma única rede (car-build-network) -**Definir dependências** entre serviços (healthchecks)
- **Gerenciar volumes** para persistência do Post- greSQL
- **Testar a comunicação** entre containers antes da migração

---

2. Arquitetura Kubernetes Implementada

2.1 Minikube Escolha Tecnológica: Minikube foi selecionado por ser:
- **Ambiente Local**: Ideal para desenvolvimento e testes
- **Cluster Completo**: Simula um cluster Kubernetes real em nó único
- **Facilidade de Uso**: Setup rápido sem configurações complexas

2.2 Componentes de Infraestrutura

A) Persistent Volume Claims (PVC)

```yaml
# postgres-pvc.yaml
kind: PersistentVolumeClaim
spec:
  resources:
    requests:
      storage: 1Gi
```

Justificativa de Uso:
- **Persistência de Dados**: Garante que dados PostgreSQL sobrevivam a reinicializações de pods
- **Desacoplamento**: Separa o lifecycle do storage do lifecycle do container
- **Portabilidade**: Permite migração entre nodes sem perda de dados

B) ConfigMaps

```yaml
# postgres-configmap.yaml
kind: ConfigMap
data:
  init.sql: |
    CREATE TABLE carros...
    CREATE TABLE pecas...
```

Justificativa de Uso:
- **Separação de Configuração**: Remove scripts SQL do código da aplicação
- **Versionamento**: Permite controle de versão das configurações
- **Reutilização**: Configurações podem ser compartilhadas entre ambientes

2.3 Camada de Deployments

A) PostgreSQL Deployment

```yaml
# postgres-deployment.yaml
kind: Deployment
```

```
spec:
  replicas: 1
```

```
  template:
    spec:
      containers:
      - image: postgres:15-alpine
        volumeMounts:
        - name: postgres-storage
          mountPath: /var/lib/postgresql/data
        - name: init-script
          mountPath:  /docker-entrypoint-initdb.d
```

Estratégia Adotada:
- **Imagem Oficial**: postgres:15-alpine para compatibilidade e segurança
- **Volume Mounting**: PVC para dados + ConfigMap para inicialização
- **Single Replica**: Para bancos estateful, uma réplica evita problemas de sincronização

B) Microserviços Deployments

```
# server-a-deployment.yaml / server-b-deployment.yaml
kind: Deployment
spec:
  replicas: 1
  template:
    spec:
      containers:
      - image: car_build-server-a:latest
        imagePullPolicy: IfNotPresent
        env:
        - name: DB_HOST
          value: "postgres-service"
```

Decisões de Design:
- **ImagePullPolicy: IfNotPresent**: Otimiza build local com minikube
- **Variáveis de Ambiente**: Injeção de con- figuração via Kubernetes
- **Service Discovery**: Uso de DNS interno (postgres-service)

C) API Gateway Deployment

```
# p-api-deployment.yaml
kind: Deployment
spec:
  template:
    spec:
      containers:
      - image: car_build-p-api:latest
        env:
        - name: SERVER_A_HOST
```

```
      value: "server-a-service"
```

```
    - name: SERVER_B_HOST
      value: "server-b-service"
```

Arquitetura Gateway Pattern:
- **Centralização**: Único ponto de entrada para requisições externas
- **Conversão de Protocolo**: HTTP REST para gRPC internamente
- **Service Mesh**: Comunicação com microserviços via DNS interno

---

3. Configuração de Rede e Services

3.1 Services Internos (ClusterIP) Postgres Service:

```
# postgres-service.yaml
kind: Service
spec:
  type: ClusterIP
  ports:
  - port: 5432
    targetPort: 5432
```

Microserviços Services:

```
# server-a-service.yaml / server-b-service.yaml
kind: Service
spec:
  type: ClusterIP
  ports:
  - port: 50051/50052
    targetPort: 50051/50052
```

Justificativa ClusterIP:
- **Segurança**: Serviços não expostos externamente
- **Performance**: Comunicação direta entre pods
- **Service Discovery**: DNS automático (service-name.namespace.svc.cluster.local)

---

4. Comandos de Implementação e Dificuldades

4.1 Sequência de Comandos Utilizados

```
# 1. Preparação do Ambiente
minikube start
minikube dashboard # Opcional - monitoramento visual

# 2. Build das imagens Docker
cd Car_Build
docker build -f ./Microservices/serverA-microsservice/Dockerfile -t car_build-server-
```

a:latest

```
docker build -f ./Microservices/serverB-microsservice/Dockerfile -t car_build-server-b:latest
docker build -f ./P-Api/Dockerfile -t car_build-p-api:latest ./P-Api

# 3. Carregar imagens no minikube

# Depois que já tiver as imagens, rode o comando para carregar as imagens no minikube, não
# precisa ta com o docker rodando, só precisava das imagens:

minikube image load car_build-p-api:latest
minikube image load car_build-server-a:latest
minikube image load car_build-server-b:latest

# 4. Aplicar os manifests

# Depois, entre na pasta /manifests e rode o comando para dar apply no Kubernetes:

cd manifests
kubectl apply -f .

# 5. Verificar deployments

# Deve demorar um pouco para subir o banco, mas confirma com os comandos:

kubectl get deployments
kubectl get pods

# 6. Expor o serviço com Minikube Tunnel

# Criar túnel (manter rodando em background):
bash
minikube tunnel

# Verificar se funcionou:

kubectl get service p-api-service

# Deve mostrar algo como:
# NAME            TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)      AGE
# p-api-service   LoadBalancer   10.96.X.X       localhost       8000:XXX     1m


# > *Resultado*: P-API sempre disponível em http://localhost:8000

# 7. Iniciar o frontend
```

```
# O frontend usa URL
fixa configurada:
        cd WebClient
        npm start
```

4.2 Dificuldades Encontradas

A) Gerenciamento de Imagens
Problema: Kubernetes não encontrava imagens Docker locais **Solução**: Uso do `minikube image load` para transferir imagens para o cluster interno **Lição**: Minikube possui registry interno separado do Docker local

B) Service Discovery e DNS
Problema: Microserviços não conseguiam se conectar usando hostnames definidos nos dockerfiles **Solução**: Migração para DNS interno do Kubernetes (`service-name`) **Configuração**: Variáveis de ambiente no gateway apontando para services (`postgres-service`, `server-a-service`)

C) Persistência de Dados
Problema: Dados PostgreSQL eram perdidos a cada restart do pod **Solução**: Implementação de Persis- tentVolumeClaim **Configuração**: Volume mounting correto nos De- ployments

D) Exposição Externa
Problema: Frontend não conseguia aces- sar API Gateway **Solução**: Service tipo LoadBalancing **Resultado**: Túnel e porta fixa entre localhost e cluster

---

5. Resultados Alcançados

5.1 Arquitetura Final Funcional

Infraestrutura Obtida:
- **Cluster Kubernetes** funcional com minikube
- **4 Deployments** indepen- dentes e escaláveis
- **5 Services** com DNS interno funcional
- **Persistent Volume** para dados PostgreSQL
- **ConfigMap** para scripts de inicialização
- **Rede Externa** via LoadBalancing funcional
- **Comunicação gRPC** interna entre microserviços

5.2 Benefícios Obtidos

Aspecto	Docker Compose	Kubernetes
Escalabilidade	Manual, limitada	Automática por deployment
Alta Disponibilidade	Restart simples	Redistribuição automática
Service Discovery	Hostnames fixos	DNS dinâmico
Monitoramento	Logs básicos	Dashboard + métricas
Configuração	Environment files	ConfigMaps + Secrets
Networking	Bridge network	Service mesh

5.3 Funcionalidades Implementadas Orquestração Completa:
- **Auto-restart** de containers com falhas
- **Load balancing** interno entre réplicas
- **Service discovery** automático via DNS
- **Volume management** para persistência
- **Configuration management** via ConfigMaps

Monitoramento e Observabilidade:
- Dashboard visual via `minikube dashboard`
- Logs centralizados via `kubectl logs`
- Status de health via `kubectl get pods` - Métricas de recursos via Kubernetes API

5.4 Vantagens da Migração

Técnicas:
- **Isolamento**: Cada serviço em namespace próprio
- **Escalabilidade**: Réplicas indepen- dentes por deployment
-**Resiliência**: Auto-recovery e redistribuição
- **Flexibilidade**: Configuração declarativa via YAML

Operacionais:

- **Padronização**: Mesma interface para todos os ambientes
- **Versionamento**: Controle de versão de toda infraestrutura
- **Debugging**: Ferramentas avançadas de troubleshooting

Esta implementação demonstra uma migração bem-sucedida de Docker Compose para Kubernetes, evidenciando as vantagens de orquestração, escalabilidade e gerenciamento de uma arquitetura de microserviços em ambiente cloud-native.

5.5 Conformidade com Especificação

Esta implementação atende perfeitamente aos requisitos definidos inicialmente:

- **HServ**: Kubernetes cluster (minikube)
- **HClient**: Browser com frontend React
- **Container P**: API Gateway (HTTP ↔ gRPC)

- **Container A**: Microserviço catálogo gRPC
- **Container B**: Microserviço pricing gRPC
- **Rede Externa**: HTTP tradicional via LoadBalancer
- **Rede Interna**: HTTP/2 gRPC entre containers
- **Persistência**: PostgreSQL com PVC

Resultado Esperado

- **Backend**: Disponível via túnel do minikube
- **Frontend**: http://localhost:3000
- **Dashboard**: `minikube dashboard` para monitoramento

Conclusão

A atividade extraclasse proporcionou uma experiência prática e enriquecedora no desenvolvimento de sistemas distribuídos utilizando o protocolo gRPC com Protobuf, bem como a comparação com o modelo tradicional de comunicação REST/JSON. A implementação dos microsserviços A, B e P permitiu compreender os desafios e as vantagens de cada abordagem, destacando a eficiência e a performance do gRPC em cenários que exigem alta escalabilidade e baixa latência. Além do mais, a utilização do Minikube para orquestração local com Kubernetes também foi fundamental para entender os conceitos de conteinerização e gerenciamento de microsserviços em um ambiente controlado.

Nesse sentido, a atividade contribuiu significativamente para o desenvolvimento de habilidades técnicas e a compreensão dos princípios fundamentais da arquitetura de microsserviços e da comunicação entre sistemas distribuídos, elementos tão importantes no contexto atual do desenvolvimento de software em que mutas empresas estão migrando para a nuvem e adotando arquiteturas baseadas em microsserviços. Assim, aprender tais conceitos e tecnologias é essencial para qualquer profissional que deseja se destacar no mercado.

Feedback dos integrantes do grupo

Nome: FLÁVIO GUSTAVO ARAÚJO DE MELO

Feedback:
A atividade foi de extrema importância para o aprimoramento dos meus conhecimentos prévios. Eu já possuía experiência com Docker e Docker Compose, porém ainda não havia tido a oportunidade de trabalhar com Kubernetes e com o protocolo gRPC. Como o trabalho envolveu a comunicação entre microserviços e a conteinerização, pude aplicar conceitos valiosos, especialmente o uso do gRPC em um exemplo prático e pronto para deploy. Dessa forma, acredito que minha contribuição foi significativa para o desenvolvimento do projeto como um todo, com destaque para a parte de Kubernetes, onde optei por explorar a tecnologia e realizar a implementação.

Autoavaliação: 9/10

Nome: GUILHERME SILVA DUTRA

Feedback:
Considero esta atividade de grande valia para o meu desenvolvimento profissional. Ela proporcionou o exercício de conhecimentos avançados em microsserviços e em tecnologias valorizadas pelo mercado. Na minha avaliação, essa prática é fundamental, uma vez que a graduação em Engenharia de Software prioriza os fundamentos, um conhecimento indispensável, porém, por vezes, distante das aplicações imediatas no ambiente de trabalho. Dessa forma, a execução de tarefas como esta contribui significativamente para nossa familiarização com as tecnologias demandadas pela indústria. Assim, o trabalho se mostrou, portanto, muito relevante, permitindo-me aplicar conceitos alinhados à realidade profissional e aprofundar-me no gRPC, tecnologia à qual dediquei especial atenção

Autoavaliação: 7/10

Nome: GUSTAVO FRANCA BOA SORTE

Feddback:
Esta atividade foi fundamental para meu desenvolvimento técnico, especialmente na implementação prática de sistemas distribuídos. A criação em determinadas funções me permitiu compreender os desafios reais da comunicação entre microsserviços via gRPC e Protocol Buffers. O trabalho com diferentes linguagen integradas através de gRPC demonstrou as vantagens desta tecnologia sobre REST tradicional. A experiência com dockerização e debugging distribuído me permitiu desenvolver habilidades essenciais para o mercado atual. Considero que contribuí de forma integral, implementando as funcionalidades designadas e auxiliando na validação da infraestrutura para Kubernetes.

Autoavaliação: 8/10

Nome: HARRYSON CAMPOS MARTINS

Feddback:
Este execício foi essencial para o meu aprimoramento técnico, especialmente no que diz respeito à aplicação prática de conceitos de sistemas distribuídos. Ao participar do desenvolvimento de determinadas funcionalidades, pude compreender as tecnologias gRPC e Protocol Buffers, além de microsserviços .Além disso me proporcionou entender as diferenças entre o REST tradiocional e o grpc. Também tive uma participação ativa na implementação das funcionalidades atribuídas e no suporte à validação da infraestrutura voltada ao Kubernetes, o que me proporcionou uma visão mais ampla sobre orquestração e escalabilidade de serviços. De modo geral, considero que contribui de forma integral e essa prática representou um avanço marcante tanto no meu domínio técnico quanto na minha capacidade de colaboração em projetos complexos.

Autoavaliação: 9/10