Atividade extraclasse - PSPD

Curso: Engenharia de Software – Disciplina PSPD / Turma 2025.2
Integrantes: Artur Rodrigues Sousa Alves (211043638), Guilherme Soares Rocha (211039789), Pedro Augusto Dourado Izarias (200062620)

Prof. Fernando W. Cruz

Link para o vídeo de apresentação

https://www.youtube.com/watch?v=1sEO6BjSojc

Link para o repositório do github

https://github.com/Izarias/PSPD_Trabalho1

---

1. Introdução

Este relatório descreve o desenvolvimento de uma aplicação distribuída baseada em microserviços gRPC (Gateway P + Serviços A e B) implantada em um ambiente Kubernetes local (Minikube). O objetivo central foi: (i) estudar gRPC e seus padrões de comunicação; (ii) construir uma solução multi-linguagem (Node.js, Python e Go); (iii) containerizar e orquestrar os componentes em Kubernetes; (iv) demonstrar fluxos de invocação e streaming.

Tecnologias principais: gRPC, Protocol Buffers, HTTP/2, Node.js (gateway), Python (UserService), Go (StatsService), Docker, Kubernetes (Deployments, Services, Ingress), WebSocket (ponte para stream bidirecional), scripts de automação.

Visão geral sobre a estrutura do relatório: Seção 2 detalha o framework gRPC; Seção 3 descreve a aplicação (arquitetura, fluxos, execução e dificuldades); Seção 4 aborda a orquestração em Kubernetes; Seção 5 apresenta conclusões e autoavaliação;

---

2. Estudo do Framework gRPC

2.1 Componentes Principais

Componente	Descrição Resumida
Protocol Buffers	Linguagem de IDL e formato binário eficiente para serialização/deserialização.

Componente	Descrição Resumida
HTTP/2	Canal multiplexado, cabeçalhos comprimidos, suporte natural a streams, usado pelo gRPC.
Canal (Channel) & Stubs	O cliente cria canal para endereço do servidor; stubs encapsulam chamadas RPC.
Metadata	Metadados chave-valor para autenticação, tracing e contexto.
Deadlines & Cancelamento	Permitem limitar tempo de execução e liberar recursos.
Status Codes	Modelo padronizado de retorno (OK, NOT_FOUND, UNAVAILABLE, etc.).

2.2 Padrões de Comunicação e Testes

Padrão	Métodos Implementados -	Descrição do Cenário de Teste	Evidência (referência)
Unary	GetUser / GetScore	Requisição simples, retorno único.	Log smoke: GetUser / Score
Server Streaming	ListUsers	Gateway consome stream e agrega resposta JSON.	Log smoke: /users
Client Streaming	CreateUsers	Gateway envia vários usuários e recebe resumo final (count, ids).	Log smoke: bulk create
Bidirectional Streaming	UserChat (via WebSocket)	Cada mensagem WebSocket vira send() no stream gRPC; respostas eco + transformação.	Captura /chat-test

Observações de uso:

- Unary: CRUD típico, baixa complexidade.
- Server streaming: envio de listas graduais (lazy fetch / paginação incremental).
- Client streaming: upload em lote (agregação) antes de processar.
- Bidirectional: chat, telemetria, atualizações contínuas.

Arquivo `.proto` principal define as mensagens User, ChatMessage, ScoreRequest/Response e serviços UserService & StatsService com todos os padrões.

---

3. Aplicação Distribuída (Arquitetura P, A, B)

3.1 Descrição da Aplicação

Domínio escolhido: Catálogo simples de usuários (Serviço A) com cálculo de pontuação dinâmica (Serviço B). O Gateway oferece API HTTP/JSON para clientes externos e traduz chamadas para gRPC interno. Um canal WebSocket implementa uma ponte para o stream bidirecional UserChat.

3.2 Fluxos de Requisição

1. GET /users/:id → Gateway -> gRPC GetUser → retorno JSON.
2. GET /users → Gateway abre stream ListUsers, coleta itens e responde lista agregada.
3. POST /users/bulk (JSON array) → Gateway inicia stream CreateUsers, envia cada registro e recebe UsersSummary.
4. GET /scores/:userId → Gateway -> gRPC GetScore.
5. WebSocket /chat → Para cada mensagem JSON {user_id, text} o Gateway escreve no stream UserChat; respostas chegam em push para o cliente.

3.3 Passos de Instanciação / Execução (Local)

Preparação (primeira vez):

```
git clone <repo>
cd <repo>
./scripts/gen_protos.sh        # Gera stubs Go/Python se necessário
./scripts/run_all_local.sh     # Sobe serviços gRPC + gateway (portas dinâmicas)
```

Chat de teste: acessar http://localhost:8080/chat-test (ou porta configurada) em duas abas e trocar mensagens.

3.4 Dificuldades e Soluções (Parte de gRPC)

Durante o desenvolvimento da aplicação, enfrentamos diversos desafios técnicos que exigiram soluções específicas. Um dos primeiros problemas encontrados foi o alinhamento entre diferentes versões do Protocol Buffers: código gerado com a versão 6.x, enquanto o runtime era 5.x, causando incompatibilidades. Para resolver essa situação, foi necessário atualizar explicitamente para o protobuf 6.31.1 em todos os componentes.

A instalação do pacote grpcio em Python 3.13 também se mostrou desafiadora, pois o processo caía para compilação a partir do código fonte, resultando em builds

extremamente demoradas. A solução foi adotar a versão 1.75.1, que oferecia wheels pré-compilados compatíveis, acelerando significativamente o processo de instalação.

A orquestração local de múltiplas portas entre os serviços apresentava conflitos frequentes e dificuldades no gerenciamento de variáveis de ambiente. Para contornar esse problema, implementamos o script `run_all_local.sh`, que automaticamente gera um arquivo `ports.env` com as configurações necessárias, simplificando a inicialização dos serviços.

Por fim, enfrentamos complicações para implementar o streaming bidirecional acessível via navegador, já que era necessário criar uma ponte entre WebSockets e gRPC. Nossa solução foi implementar um endpoint WebSocket (`wss /chat`) que estabelece um stream gRPC para cada conexão de cliente, permitindo a comunicação em tempo real.

3.5 Comparativo de Performance (gRPC vs REST)

Foi construída uma variante REST (serviços Users e Stats em FastAPI e Go) e realizado um microteste comparando latência de chamadas simples (unary) entre a rota exposta via gateway gRPC e as rotas REST equivalentes acessadas diretamente.

Metodologia

- Ambiente: execução local
- Iterações: 40 requisições por cenário (script `scripts/quick_compare.sh`).
- Métricas coletadas: média (Avg_ms), p95 (p95_ms), menor valor observado (Min_ms).
- Endpoints testados:
  - gRPC (via Gateway P): `/users/1` (GetUser), `/scores/1` (GetScore)
  - REST direto: `/users/1` (FastAPI), `/scores/1` (Go REST)

Resultados

Cenário	Tecnologia	Tempo Médio (ms)	P95 (ms)	Tempo Mín. (ms)	Amostras
GetUser	gRPC (via Gateway)	38.73	41.00	37	40
GetUser	REST (direto)	38.12	40.00	36	40

Cenário	Tecnologia	Tempo Médio (ms)	P95 (ms)	Tempo Mín. (ms)	Amostras
GetScore	gRPC (via Gateway)	37.92	39.00	37	40
GetScore	REST (direto)	37.33	40.00	36	40

Resumo Comparativo por Operação

Operação	Diferença Tempo Médio	Diferença P95	Observação
GetUser	gRPC +0.61ms (+1.6%)	gRPC +1.00ms (+2.5%)	Praticamente equivalente
GetScore	gRPC +0.59ms (+1.6%)	gRPC -1.00ms (-2.5%)	Praticamente equivalente

Análise

Os valores ficaram praticamente equivalentes para ambos os estilos em ambiente local com payload mínimo. A ausência de diferença significativa decorre de:

1. Payloads muito pequenos (mensagens simples) — o benefício de serialização binária (Protobuf) fica marginal frente a JSON.
2. Cada invocação REST e cada chamada gRPC via gateway foi independente; não se explorou reutilização prolongada de canal HTTP/2 ou multiplexação de múltiplas RPCs.
3. O gateway adiciona um salto extra também às chamadas gRPC, aproximando o custo do caminho REST direto.

Limitações do Experimento

- Não houve medição de rotas de lista (`/users`) onde o server streaming gRPC libera itens incrementalmente, enquanto a versão REST agrega tudo antes de responder.
- Não foi aferida mediana ou desvio padrão.

---

4. Kubernetes (Minikube)

4.1 Arquitetura e Conceitos

Componentes utilizados: Namespace dedicado (`pspd-lab`), Deployments (1 réplica cada), Services (ClusterIP internos para A e B, NodePort/Ingress para Gateway), Ingress para roteamento HTTP externo, uso de DNS interno (service-a, service-b).

4.2 Passos de Instalação (Exemplo)

```
minikube start
minikube addons enable ingress
eval "$(minikube docker-env)"
./scripts/build_local_images.sh
kubectl apply -f k8s/namespace.yaml
kubectl apply -f k8s/service-a-deployment.yaml -f k8s/service-a-service.yaml
kubectl apply -f k8s/service-b-deployment.yaml -f k8s/service-b-service.yaml
kubectl apply -f k8s/gateway-deployment.yaml -f k8s/gateway-service.yaml -f
k8s/ingress.yaml
kubectl get pods -n pspd-lab
minikube ip   # adicionar em /etc/hosts -> pspd.local
curl http://pspd.local/users/1
```

4.3 Arquivos de Configuração e Processo de Implementação

Para a implantação da nossa aplicação em Kubernetes, criamos um conjunto de arquivos de configuração YAML na pasta `k8s/`, cada um responsável por um aspecto específico da infraestrutura. O processo de implementação seguiu uma abordagem incremental, partindo da definição do namespace até a configuração do acesso externo.

Inicialmente, criamos o arquivo `namespace.yaml` que define o ambiente isolado `pspd-lab` para todos os recursos do projeto. Em seguida, desenvolvemos os arquivos de deployment para cada serviço: `service-a-deployment.yaml` para o serviço Python, `service-b-deployment.yaml` para o serviço Go, e `gateway-deployment.yaml` para o gateway Node.js. Estes manifestos incluem configurações de recursos computacionais, estratégias de implantação, políticas de imagem e verificações de saúde.

Complementando os deployments, criamos os arquivos de serviço correspondentes: `service-a-service.yaml` e `service-b-service.yaml` como ClusterIP para comunicação interna, e `gateway-service.yaml` como o ponto de entrada para requisições externas. Os serviços internos foram configurados para expor as portas gRPC com nomes apropriados para facilitar a descoberta de serviço.

Finalmente, para permitir o acesso externo via URL, criamos o arquivo `ingress.yaml`, definindo regras de roteamento para o host `pspd.local` que direcionam todo o tráfego HTTP para o serviço gateway na porta 80. Esta configuração exigiu a ativação do controlador Ingress no Minikube e a adição de uma entrada no arquivo `/etc/hosts` local.

5. Conclusão e Dificuldades

O projeto demonstrou integração multi-linguagem via gRPC, explorando todos os padrões de comunicação, e implantação consistente em Kubernetes. Observou-se redução de boilerplate em contratos (proto) e facilidade de expansão de métodos. A arquitetura separou responsabilidades: Gateway (tradução/entrada), UserService (dados de usuários), StatsService (cálculo de score). O Kubernetes facilitou o isolamento, escalabilidade futura e health management.

Os benefícios percebidos foram o desempenho potencial (serialização binária), streaming nativo, contratos fortes, facilidade de gerar stubs. Já quanto às dificuldades enfrentadas pela nossa equipe, durante o desenvolvimento do projeto, enfrentamos diversos desafios técnicos que exigiram soluções criativas. Inicialmente, encontramos problemas com a visibilidade das imagens Docker no ambiente Minikube.

O cluster Kubernetes não conseguia acessar as imagens construídas localmente, resultando em erros frequentes de "ImagePullBackOff". Após algumas investigações, descobrimos que era necessário executar `eval $(minikube docker-env)` antes de construir as imagens, o que efetivamente conecta o daemon Docker local ao ambiente do Minikube.O acesso à aplicação também apresentou desafios. Em vez de utilizar endereços IP diretos ou portas NodePort, que são pouco práticos para desenvolvimento, implementamos um Ingress e configuramos uma entrada no arquivo `/etc/hosts` apontando `pspd.local` para o IP do Minikube, facilitando o acesso e compartilhamento entre a equipe.

Para otimizar o ciclo de desenvolvimento, evitamos o envio constante de imagens para registries externos configurando os deployments com política `imagePullPolicy: IfNotPresent` e reconstruindo as imagens localmente quando necessário. Esta estratégia acelerou consideravelmente nosso processo de desenvolvimento e testes.

Sobre o comparativo, o microteste local indicou latências equivalentes entre gRPC (via gateway) e REST direto em cenário de baixa complexidade, reforçando que o ganho de gRPC se manifesta sobretudo em mensagens maiores, canais persistentes e padrões de streaming. Extensões propostas podem realçar diferenças em trabalhos futuros.

6 Contribuição Individual e Autoavaliação

Integrante	Contribuições Principais	Autoavaliação (0-10)
Pedro Augusto, Artur Rodrigues e Guilherme Soares	Proto, Gateway e WebSocket	8.5
Artur Rodrigues	Serviço A (Python), scripts	8
Pedro Augusto	Serviço B (Go), Kubernetes	8

Guilherme Soares	Automação e smoke tests	8
Artur Rodrigues, Pedro Augusto e Guilherme Soares	Documentação e testes	8.5