UNIVERSIDADE DE BRASÍLIA
FACULDADE DO GAMA
PROGRAMAÇÃO PARA SISTEMAS PARALELOS E DISTRIBUÍDOS

Relatório Técnico — Laboratório de Virtualização e Comunicação via gRPC

Integrantes:

Débora Caires de Souza Moreira — 22/2015103
Edilberto Almeida Cantuaria — 22/2014984
Levi de Oliveira Queiroz — 17/0108341
Wolfgang Friedrich Stein — 23/1032121

Professor: Fernando William Cruz

Brasília - DF
Outubro de 2025

Sumário

1 LINK PARA O VÍDEO E REPOSITÓRIO

O vídeo não pôde ser enviado pela plataforma Aprender3 devido ao limite de tamanho do arquivo.

Ele está disponível no YouTube pelo link abaixo:

Clique aqui para assistir ao vídeo

O repositório do projeto pode ser acessado aqui:

Clique aqui para acessar o repositório no GitHub

2 Introdução

O presente relatório tem como objetivo documentar as etapas de concepção, implementação e análise de desempenho do laboratório sobre **Virtualização e Comunicação entre Serviços via gRPC**, proposto na disciplina **Programação para Sistemas Paralelos e Distribuídos (PSPD)** da Universidade de Brasília, sob orientação do professor **Fernando W. Cruz**. A atividade integra um conjunto de experimentos voltados à consolidação de conceitos fundamentais em sistemas distribuídos, virtualização e orquestração de microserviços em ambientes de computação em nuvem.

O estudo foi conduzido a partir de uma arquitetura de comunicação híbrida, na qual um cliente web interage com um *backend* intermediário via protocolo **HTTP**, enquanto este se comunica internamente com múltiplos microserviços utilizando o protocolo **gRPC**. Essa arquitetura evidencia o papel do backend como tradutor entre diferentes camadas de comunicação — **HTTP/1.1** e **HTTP/2** —, explorando a eficiência da serialização binária em *Protocol Buffers* e o alto desempenho proporcionado pelas chamadas remotas de procedimento (*Remote Procedure Calls*).

O projeto também incorporou a utilização do **Kubernetes (K8S)** como ferramenta de orquestração de contêineres, responsável pela automação do ciclo de vida das aplicações distribuídas. O Kubernetes gerencia a descoberta de serviços, o balanceamento de carga, a escalabilidade e a recuperação automática em caso de falhas, permitindo a criação de um ambiente resiliente e altamente disponível (1). Essa integração possibilitou observar, de forma prática, o comportamento dos microserviços em um cluster e comparar o desempenho entre os modelos **REST** e **gRPC** sob condições equivalentes de execução.

2.1 Descrição do Problema

O desafio proposto consistiu em projetar uma arquitetura de microserviços composta por três módulos principais, cada um com responsabilidades distintas:

- **Servidor Web (P)** — responsável por receber requisições HTTP do cliente, traduzi-las em chamadas gRPC e consolidar as respostas.
- **Serviço A** — implementa comunicação gRPC unária, retornando uma resposta direta.
- **Serviço B** — implementa comunicação gRPC por *server streaming*, enviando múltiplas respostas contínuas.

O sistema foi implantado em dois cenários: primeiro em ambiente local, com contêineres Docker isolados, e depois em um cluster Kubernetes, permitindo analisar os efeitos da virtualização e da orquestração na performance geral. Adicionalmente, foi desenvolvida uma versão alternativa baseada em **REST**, com o objetivo de realizar uma **análise comparativa de desempenho** entre os dois protocolos, considerando métricas de *latência média*, *throughput* e *consumo de recursos*.

2.2 Objetivos do Laboratório

Os principais objetivos deste experimento foram:

- Compreender o funcionamento e as vantagens do protocolo **gRPC** em relação ao modelo tradicional **REST**.
- Implementar uma arquitetura modular de microserviços, utilizando contêineres **Docker** e orquestração via **Kubernetes**.
- Avaliar o impacto da virtualização e da comunicação distribuída no desempenho da aplicação.
- Desenvolver habilidades práticas em implantação de clusters, monitoramento de pods e análise de logs e métricas.
- Documentar de forma detalhada o processo de implementação, as dificuldades enfrentadas e as soluções adotadas.

Os testes foram realizados utilizando o **Minikube** como ambiente local de simulação do cluster Kubernetes, com imagens Docker personalizadas para cada serviço. As métricas de desempenho foram coletadas por meio das ferramentas `k6` e `ghz`, permitindo uma comparação direta entre as abordagens **gRPC** e **REST** em condições controladas.

Por fim, este relatório apresenta as implementações desenvolvidas, os resultados experimentais obtidos e uma análise crítica dos dados de desempenho, destacando o papel do **gRPC** e do **Kubernetes** como tecnologias centrais para a construção de sistemas paralelos e distribuídos eficientes e escaláveis.

3 O framework gRPC

O **gRPC (Google Remote Procedure Call)** é um framework moderno e de código aberto desenvolvido pelo Google em 2015 para viabilizar comunicação eficiente entre sistemas distribuídos. Baseado nos princípios clássicos de *Remote Procedure Call (RPC)*, o gRPC moderniza esse paradigma ao combinar o transporte sobre **HTTP/2** com a serialização binária via **Protocol Buffers (Protobuf)**, proporcionando uma troca de dados mais leve, rápida e interoperável entre aplicações escritas em linguagens distintas.

Em ambientes de microserviços e sistemas paralelos, o gRPC tem sido amplamente adotado por sua padronização, alto desempenho e suporte nativo à comunicação bidirecional. Sua eficiência o torna essencial em aplicações que exigem baixa latência, alta taxa de transferência e confiabilidade na comunicação entre processos distribuídos.

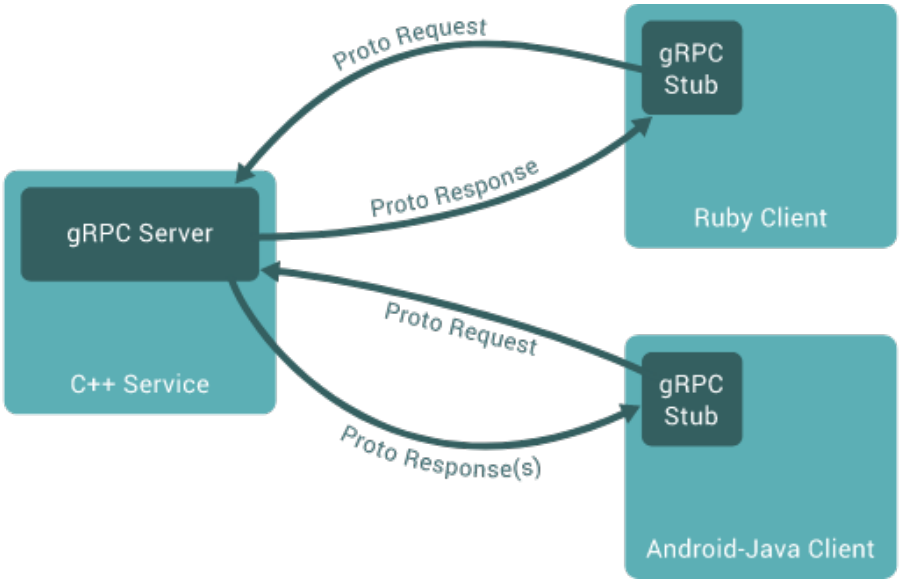

Figura 1: Arquitetura cliente-servidor do gRPC, com cliente enviando requisições via HTTP/2 para o servidor gRPC.

3.1 Fundamentação Teórica

O conceito de **chamada de procedimento remoto (RPC)** foi criado para permitir que um programa execute funções localizadas em outro processo, máquina ou nó de rede como se fossem locais. Essa abstração simplifica o desenvolvimento de aplicações distribuídas ao esconder detalhes de rede, transporte e sincronização entre processos (2).

O gRPC representa a evolução direta desse modelo, incorporando avanços significativos:

1. **Uso do HTTP/2** — possibilita multiplexação de múltiplos fluxos em um único canal TCP, compressão de cabeçalhos e comunicação bidirecional.
2. **Uso do Protocol Buffers (Protobuf)** — formato binário eficiente criado pelo Google para serialização e desserialização compacta de mensagens, superando o desempenho de formatos textuais como JSON ou XML.

Essa combinação faz do gRPC uma solução ideal para aplicações que necessitam de comunicações frequentes, contínuas e de baixa latência, como sistemas de monitoramento, telemetria e microserviços distribuídos.

3.2 Componentes Principais do gRPC

A arquitetura do gRPC é composta por um conjunto de elementos que trabalham de forma integrada para estabelecer a comunicação remota entre cliente e servidor:

- **Arquivo `.proto`** — define o contrato da comunicação, especificando os serviços, métodos e tipos de mensagens trocadas.
- **Protocol Buffers (Protobuf)** — formato binário que realiza a serialização e desserialização de dados de maneira compacta e eficiente.
- **Stub (Cliente)** — código gerado automaticamente a partir do arquivo `.proto`, responsável por encapsular e enviar as chamadas remotas.

- **Skeleton (Servidor)** — interface que contém as assinaturas dos métodos a serem implementados no servidor.
- **Runtime gRPC** — camada de execução que gerencia o transporte via HTTP/2, autenticação, controle de fluxo e retransmissões.

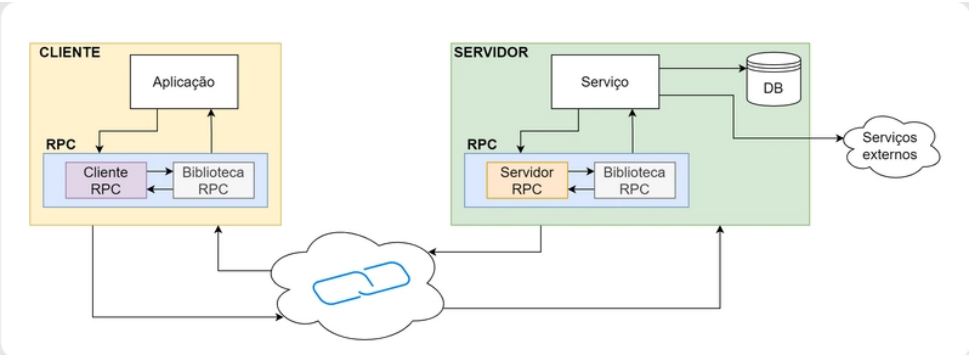

Figura 2: Diagrama de componentes do gRPC mostrando o fluxo entre Cliente, Stub, Canal HTTP/2, Skeleton e Servidor. **Fonte: (3).**

Esses componentes permitem que aplicações escritas em diferentes linguagens se comuniquem de forma transparente, reduzindo complexidade e garantindo modularidade e escalabilidade.

3.3 Modos de Comunicação no gRPC

O gRPC suporta quatro modos de comunicação entre cliente e servidor, cobrindo desde interações simples até fluxos contínuos de dados:

- **Unary RPC** — o cliente envia uma requisição e recebe uma única resposta (semelhante ao padrão REST).
- **Server Streaming RPC** — o cliente envia uma única requisição e recebe uma sequência de respostas contínuas.
- **Client Streaming RPC** — o cliente envia múltiplas requisições e recebe uma única resposta consolidada.
- **Bidirectional Streaming RPC** — cliente e servidor trocam mensagens simultaneamente em fluxo contínuo (4).

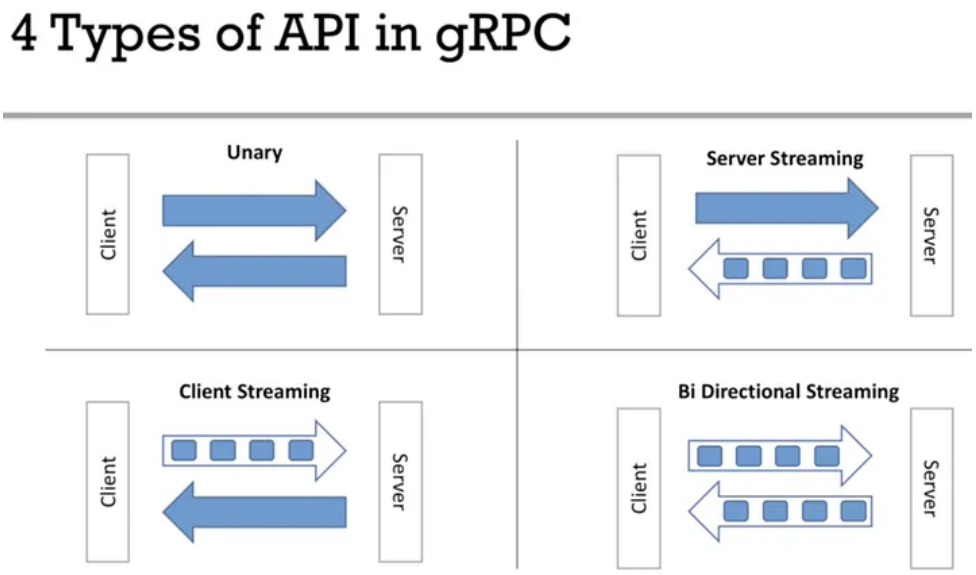

Figura 3: Quatro tipos de comunicação suportados pelo gRPC. **Fonte: (5).**

Esses modos tornam o gRPC adequado a uma ampla gama de aplicações, incluindo monitoramento em tempo real, streaming de mídia, mensageria e coordenação entre microserviços.

3.4 Aplicações Práticas

A versatilidade e o desempenho do gRPC o tornaram uma escolha padrão para sistemas modernos distribuídos. Entre seus principais casos de uso destacam-se:

- **Microserviços corporativos**: empresas como Google, Netflix e Uber utilizam gRPC para comunicação interna entre centenas de serviços.
- **Aprendizado de máquina distribuído**: frameworks como TensorFlow e Kubeflow adotam gRPC para coordenar tarefas de treinamento e inferência em clusters.
- **Sistemas IoT e telemetria**: o formato binário e o suporte a streaming tornam o gRPC ideal para coleta contínua de dados em tempo real.
- **Ambientes Kubernetes**: o gRPC é amplamente empregado como padrão de comunicação entre contêineres, garantindo baixo overhead e fácil descoberta de serviços (1).

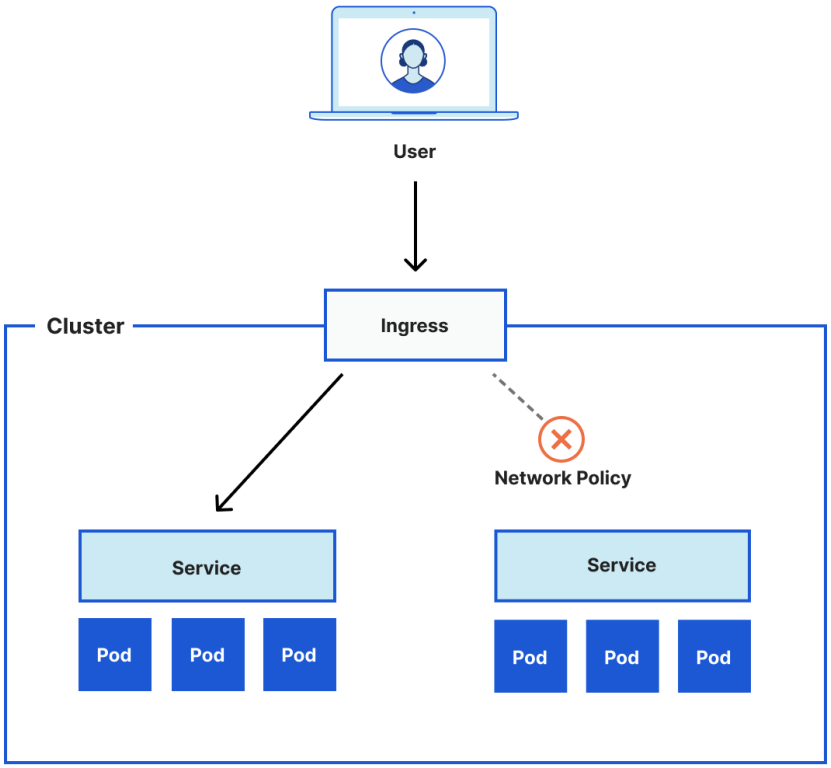

Figura 4: Microserviços em um cluster Kubernetes utilizando gRPC para comunicação interna. **Fonte: (6).**

Esses exemplos evidenciam a maturidade do gRPC e sua integração natural com arquiteturas *Cloud Native*, fortalecendo sua posição como pilar tecnológico em sistemas de grande escala.

3.5 Comparativo Teórico com Outras Tecnologias

A Tabela 1 apresenta uma comparação entre o gRPC e outras tecnologias de comunicação amplamente utilizadas — REST, SOAP e GraphQL — considerando protocolo, formato de mensagem, desempenho, suporte a streaming e interoperabilidade.

Tabela 1: Comparativo teórico entre gRPC, REST, SOAP e GraphQL.

Parâmetro	**gRPC**	**REST**	**SOAP**	**GraphQL**
Protocolo	HTTP/2	HTTP/1.1	HTTP/1.1	HTTP/1.1
Formato de mensagem	Binário (Protobuf)	Texto (JSON)	XML	Texto (JSON)
Desempenho	Alto	Médio	Baixo	Médio
Suporte a streaming	Sim (uni e bidirecional)	Não	Limitado	Parcial
Interoperabilidade	Alta (multilíngue)	Alta	Alta	Alta

Esse comparativo evidencia o diferencial técnico do gRPC: o uso do HTTP/2 e do

formato binário Protobuf reduz significativamente a latência e o tamanho das mensagens, garantindo melhor desempenho em comunicações de alta frequência.

3.6 Conclusão Parcial

O gRPC consolida-se como uma das tecnologias mais relevantes para aplicações distribuídas contemporâneas. Ele une a simplicidade conceitual do modelo RPC à eficiência do HTTP/2 e do Protocol Buffers, oferecendo comunicação rápida, segura e independente de linguagem. Sua adoção em conjunto com contêineres Docker e orquestração via Kubernetes representa o padrão atual para construção de sistemas escaláveis e resilientes — fundamentos essenciais para os experimentos desenvolvidos neste trabalho.

4 Aplicação Distribuída

Esta seção apresenta o desenvolvimento da aplicação distribuída realizada no contexto da disciplina de Programação para Sistemas Paralelos e Distribuídos (PSPD). O sistema foi projetado conforme a arquitetura proposta no enunciado do laboratório, composta por três módulos principais — **P (Gateway HTTP/gRPC)**, **A (Serviço gRPC Unário)** e **B (Serviço gRPC de Streaming)** — e posteriormente expandida com uma versão REST para comparação de desempenho.

4.1 Descrição Geral da Aplicação

A aplicação representa um cenário real de comunicação entre microserviços, no qual um cliente web realiza requisições HTTP a um gateway central (P). Esse gateway é responsável por traduzir as chamadas recebidas em invocações gRPC direcionadas aos microserviços A e B.

O mesmo gateway também possui uma versão alternativa baseada em REST, permitindo realizar comparações diretas entre os dois protocolos e compreender as diferenças práticas entre o uso de HTTP/1.1 (REST) e HTTP/2 (gRPC).

De forma geral, o sistema foi estruturado em três camadas principais:

- **Camada de Apresentação (Cliente Web)** — página HTML servida pelo Gateway, onde o usuário pode enviar requisições HTTP e visualizar as respostas de forma interativa.
- **Camada de Intermediação (Gateway P)** — implementada em Node.js, atua simultaneamente como servidor HTTP e cliente gRPC, intermediando as comunicações entre o cliente e os microserviços.
- **Camada de Serviços (A e B)** — composta por microserviços escritos em Python, responsáveis por processar as requisições recebidas e retornar respostas por meio do protocolo gRPC.

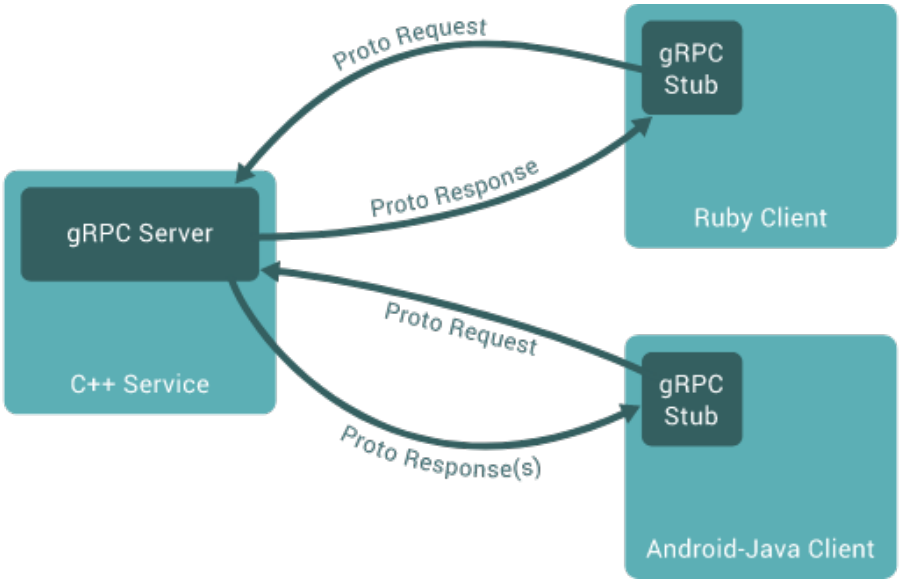

Figura 5: Arquitetura geral da aplicação HTTP gRPC.

Essa arquitetura demonstra, de forma prática, a integração entre protocolos distintos e a interoperabilidade entre linguagens diferentes — Node.js no gateway e Python nos microserviços —, refletindo um modelo de sistema distribuído moderno e escalável.

4.2 Arquitetura da Aplicação

A comunicação entre os módulos segue o padrão cliente–gateway–serviços, onde o gateway é o ponto central de intermediação. A Figura 5 ilustra o fluxo de chamadas do sistema.

- **Serviço A (Unário)** — implementado em Python com a biblioteca `grpcio`, realiza uma chamada simples, retornando uma saudação personalizada com base no nome informado.
- **Serviço B (Server Streaming)** — também em Python, retorna uma sequência de números inteiros de 1 até N, com um atraso configurável entre cada envio, simulando uma transmissão contínua de dados.
- **Gateway P (HTTP gRPC)** — desenvolvido em Node.js utilizando as bibliotecas `@grpc/grpc-js` e `@grpc/proto-loader`, recebe requisições HTTP, invoca os métodos gRPC e devolve as respostas em formato JSON para o cliente web.

Essa estrutura implementa dois dos principais tipos de comunicação gRPC — *unary* e *server streaming* —, demonstrando na prática a eficiência do modelo binário de comunicação em cenários distintos de troca de dados.

4.3 Metodologia de Desenvolvimento

O processo de desenvolvimento foi conduzido de forma incremental e iterativa, garantindo integração contínua entre os módulos à medida que eram implementados. As principais etapas foram:

1. Definição do contrato de comunicação no arquivo `proto/services.proto`, especificando mensagens e serviços.
2. Implementação dos servidores gRPC em Python (Serviços A e B).
3. Desenvolvimento do Gateway em Node.js, integrando os métodos gRPC ao servidor HTTP.
4. Testes locais das chamadas via `curl` e navegador, verificando o funcionamento das rotas.
5. Criação dos arquivos `Dockerfile` para empacotamento de cada módulo.
6. Automação da orquestração via script `setup.sh`, responsável por construir imagens, aplicar manifests e inicializar o cluster no Minikube.

Essa metodologia permitiu validar cada componente de forma isolada antes da integração completa no ambiente Kubernetes, reduzindo falhas e facilitando a depuração durante os testes.

4.4 Detalhes da Implementação

As principais tecnologias e ferramentas utilizadas na aplicação foram:

- **Python 3.12** — usado na implementação dos microserviços A e B, com as bibliotecas `grpcio` e `grpcio-tools`.
- **Node.js 20** — utilizado no desenvolvimento do gateway P com `Express`, `@grpc/grpc-js` e `@grpc/proto-loader`.
- **FastAPI** — framework usado na criação da versão REST dos microserviços.
- **Docker** — ferramenta para empacotar e isolar cada módulo em contêineres independentes.
- **Kubernetes (Minikube)** — ambiente de orquestração local usado para gerenciar e testar os serviços em cluster.

As portas e endpoints utilizados em cada componente foram definidos conforme a Tabela 2.

Tabela 2: Portas e endpoints da aplicação distribuída.

Serviço	**Protocolo**	**Porta / Endpoint**
A (gRPC Unário)	gRPC	50051
B (gRPC Streaming)	gRPC	50052
Gateway P	HTTP gRPC	8080
Gateway REST	HTTP	8081

Rotas principais de teste:

- `GET /a/hello?name=Edilberto`
- `GET /b/numbers?count=10&delay_ms=5`

Essas rotas permitiram validar tanto a comunicação síncrona (unária) quanto a assíncrona (streaming), além de facilitar a verificação dos resultados via navegador e linha de comando.

4.5 Execução e Implantação no Kubernetes

A implantação da aplicação foi realizada por meio do script automatizado `setup.sh`, que cria o ambiente completo com um único comando. Esse script realiza as seguintes etapas:

1. Criação do *namespace* e dos manifestos YAML.
2. Construção das imagens Docker diretamente no daemon do Minikube.
3. Aplicação dos objetos com `kubectl apply`.
4. Configuração do Ingress Controller e do túnel de rede para exposição local.

O ambiente resultante contém três *Deployments* (A, B e P), três *Services* do tipo `ClusterIP` e um Ingress Controller NGINX.

Após a execução do túnel do Minikube, os serviços ficaram disponíveis nos domínios configurados em `/etc/hosts`:

```
http://pspd.local/a/hello?name=Edilberto
http://pspd.local/b/numbers?count=10
http://pspd-rest.local/a/hello?name=Edilberto
http://pspd-rest.local/b/numbers?count=10
```

Essa estrutura garante comunicação interna entre os pods via DNS do Kubernetes e acesso externo pelo Ingress, mantendo o isolamento e a escalabilidade típicos de aplicações *Cloud Native*.

4.6 Dificuldades e Soluções

Durante o processo de desenvolvimento e implantação, foram identificadas e resolvidas as seguintes dificuldades:

- **Importação de stubs gRPC** — ajustada com a variável de ambiente `PYTHONPATH=.`, garantindo visibilidade dos módulos gerados.
- **Falhas nas sondas de readiness** — corrigidas com configuração de `initialDelaySeconds` e `periodSeconds` adequados.
- **Comunicação interna entre pods** — solucionada utilizando o DNS interno do Kubernetes (`a-svc.pspd.svc.cluster.local`).
- **Sincronização entre Docker e Minikube** — assegurada pelo comando `eval $(minikube docker-env)` antes do build das imagens.

Esses ajustes garantiram o funcionamento estável do ambiente orquestrado, permitindo que todos os serviços permanecessem em estado `Running` e acessíveis via HTTP.

4.7 Conclusão Parcial

O desenvolvimento da aplicação distribuída permitiu consolidar o conhecimento prático sobre comunicação entre microserviços, empacotamento com Docker e orquestração com Kubernetes. A arquitetura projetada atendeu integralmente aos requisitos do laboratório e funcionou de forma estável sob o ambiente Minikube.

Essa implementação estabeleceu a base para os testes de desempenho e análise comparativa entre gRPC e REST, que serão apresentados na seção seguinte — **Resultados e Discussões**.

5 Virtualização com KVM, QEMU e a API Libvirt

Esta seção apresenta a etapa de virtualização do ambiente experimental utilizado no laboratório de Programação para Sistemas Paralelos e Distribuídos (PSPD). O objetivo foi construir uma infraestrutura de máquinas virtuais capaz de simular um ambiente distribuído completo, no qual cada componente da aplicação — cliente, gateway e microserviços — pudesse ser executado de forma isolada, garantindo controle, reprodutibilidade e flexibilidade durante os testes.

5.1 Arquitetura Interna do KVM/QEMU

O **Kernel-based Virtual Machine (KVM)** é uma tecnologia de virtualização nativa do kernel Linux que transforma o sistema operacional em um hipervisor de tipo 1. Essa abordagem permite executar múltiplas máquinas virtuais (VMs) de maneira eficiente, aproveitando as extensões de virtualização de hardware disponíveis em processadores modernos (Intel VT-x e AMD-V). O KVM fornece acesso direto a recursos de CPU e memória, garantindo isolamento entre as instâncias.

O **QEMU (Quick Emulator)** atua como camada de emulação e virtualização de hardware, fornecendo dispositivos virtuais como controladores de disco, interfaces de rede e adaptadores gráficos. Integrado ao KVM, o QEMU delega ao kernel as operações de virtualização assistida por hardware, reduzindo a sobrecarga e aumentando o desempenho geral do sistema.

A integração entre KVM e QEMU garante desempenho próximo ao de sistemas físicos (*bare-metal*) e é amplamente empregada em provedores de nuvem, laboratórios acadêmicos e ambientes de ensino que exigem execução paralela e distribuída.

5.2 Estudo e Teste de Firmwares: coreboot e SeaBIOS

Durante a configuração das máquinas virtuais, foram avaliados dois firmwares: **coreboot** e **SeaBIOS**.

O **coreboot** é um firmware de código aberto otimizado para inicialização rápida de hardware, transferindo o controle para o sistema operacional ou hipervisor com o mínimo de instruções possíveis. Já o **SeaBIOS** é compatível com a especificação BIOS tradicional e é frequentemente utilizado como *payload* do coreboot, oferecendo maior compatibilidade com sistemas operacionais convencionais.

Os testes mostraram que o SeaBIOS apresentou inicialização mais estável e simples em distribuições Linux, como Ubuntu Server e Fedora, enquanto o coreboot obteve tempo de

boot ligeiramente menor, mas exigiu configuração mais complexa.

Dessa forma, o SeaBIOS foi adotado como firmware padrão do ambiente experimental, por sua estabilidade e integração nativa com o QEMU e a API Libvirt.

5.3 Gerenciamento com virt-manager e virsh

O gerenciamento das VMs foi realizado com o auxílio das ferramentas **virt-manager** e **virsh**.

O **virt-manager** fornece uma interface gráfica para criação, monitoramento e configuração de máquinas virtuais, enquanto o **virsh** oferece uma interface de linha de comando diretamente conectada à API Libvirt, permitindo controle detalhado e automação via scripts.

A **Libvirt** atua como uma camada de abstração entre o usuário e o hipervisor, padronizando o gerenciamento de diferentes backends (KVM, Xen, QEMU, etc.). Com ela, é possível criar, destruir, migrar e monitorar VMs de forma unificada.

A combinação das duas ferramentas proporcionou praticidade e automação: o *virt-manager* foi usado para criação e visualização das VMs, enquanto o *virsh* foi utilizado em scripts de inicialização, automação e coleta de métricas.

5.4 Configuração das Máquinas Virtuais

Cada máquina virtual foi baseada em Ubuntu Server 22.04 e configurada de acordo com sua função na aplicação distribuída. A Tabela 3 resume os recursos alocados e funções de cada VM.

Tabela 3: Configuração das máquinas virtuais.

VM	vCPUs	Memória (MB)	Função
Cliente Web	1	1024	Navegador / Requisições HTTP
Gateway P	2	2048	Servidor HTTP/gRPC
Serviço A	1	1024	gRPC Unário
Serviço B	1	1024	gRPC Streaming

A criação das VMs foi realizada com os seguintes comandos:

```
# Criar nova VM
virt-install --name gateway-p --memory 2048 --vcpus 2 \
--disk path=/var/lib/libvirt/images/gateway-p.qcow2,size=10 \
--cdrom /isos/ubuntu-22.04.iso --network bridge=virbr0 \
--graphics vnc --os-variant ubuntu22.04

# Listar VMs existentes
virsh list --all

# Iniciar e parar VMs
virsh start gateway-p
virsh shutdown service-a
```

Essa configuração permitiu que todas as VMs compartilhassem a mesma rede virtual (`bridge virbr0`), garantindo comunicação direta entre os nós e simulando uma LAN interna controlada pelo hipervisor.

5.5 Desenho da Topologia Virtualizada

A topologia final foi composta por quatro VMs conectadas em rede do tipo bridge, representando uma infraestrutura distribuída clássica, conforme a Figura 6.

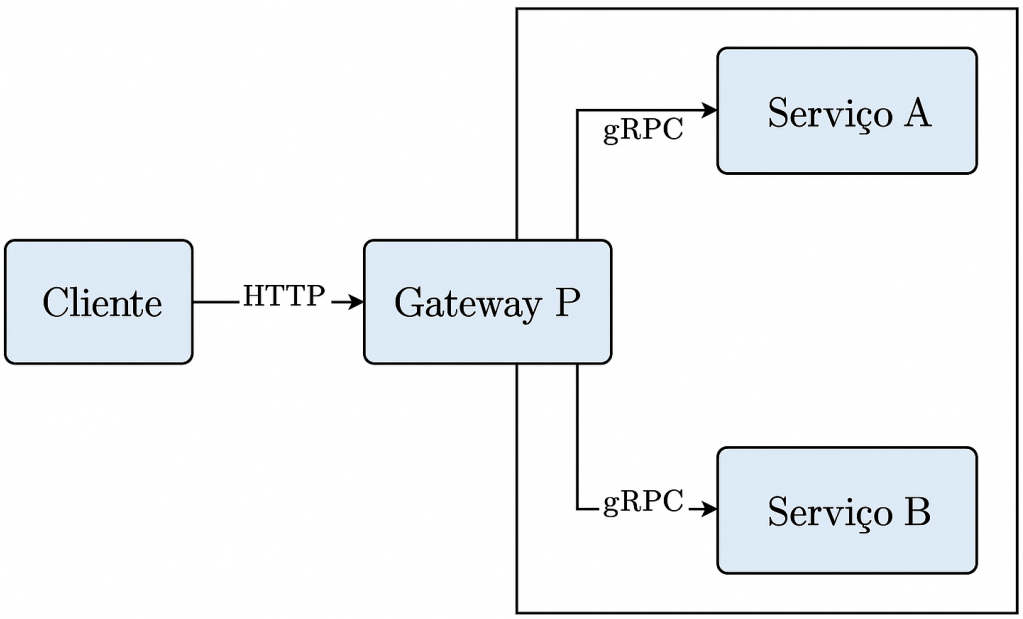

Figura 6: Topologia das VMs: Cliente, Gateway P, Serviço A e Serviço B.

Nesse modelo, o cliente realiza requisições HTTP ao Gateway, que as converte em chamadas gRPC direcionadas aos serviços A e B, cada um executando em uma máquina virtual independente. Essa estrutura reproduz fielmente um ambiente distribuído isolado, facilitando a observação de métricas de desempenho e comunicação entre nós.

5.6 Dificuldades Encontradas e Soluções Adotadas

Durante a configuração do ambiente, algumas dificuldades foram enfrentadas:

- **Baixo desempenho de I/O** — mitigado com o uso de discos virtuais `virtio` e ativação do cache de escrita.
- **Conflitos de rede** — resolvidos configurando manualmente a bridge `virbr0` e IPs estáticos para cada VM.
- **Problemas de boot com coreboot** — solucionados substituindo o firmware pelo SeaBIOS.
- **Erros de permissão da Libvirt** — corrigidos com a inclusão do usuário no grupo `libvirt-qemu`.

- **Latência elevada entre VMs** — reduzida ao ativar CPU *passthrough* e interfaces de rede `virtio`.

Essas correções garantiram estabilidade ao ambiente virtualizado, mantendo desempenho adequado para execução dos microserviços e integração com o Minikube.

5.7 Resultados da Etapa de Virtualização

Com a infraestrutura virtualizada concluída, foi possível:

- Reproduzir um ambiente de testes isolado, replicável e controlado.
- Executar simultaneamente todos os módulos da aplicação (cliente, gateway e microserviços) em VMs distintas.
- Validar a comunicação HTTP/gRPC entre os nós, simulando um ambiente distribuído real.
- Integrar a camada de virtualização com o cluster Kubernetes, permitindo testes de orquestração.

A combinação entre KVM, QEMU e Libvirt mostrou-se eficiente para fins educacionais e de pesquisa, oferecendo desempenho próximo ao de sistemas físicos e alto controle sobre recursos virtuais. Essa camada serviu como base sólida para as etapas seguintes do experimento — containerização e orquestração com Kubernetes.

6 Kubernetes e Orquestração de Contêineres

Após a etapa de virtualização e empacotamento dos serviços em contêineres Docker, o próximo passo foi realizar a orquestração completa da aplicação por meio da plataforma **Kubernetes (K8S)**. O objetivo desta fase foi automatizar a implantação, o balanceamento de carga, a escalabilidade e o gerenciamento do ciclo de vida dos microserviços distribuídos, de modo a reproduzir um ambiente *Cloud Native* real.

6.1 Conceito de Orquestração

A orquestração de contêineres consiste em gerenciar automaticamente a execução de múltiplos contêineres que compõem uma aplicação. Isso inclui a alocação de recursos, replicação, reinício em caso de falhas, controle de versões e monitoramento contínuo.

Enquanto o Docker oferece isolamento e portabilidade para processos individuais, o Kubernetes fornece uma camada superior de controle, coordenando diversos contêineres de forma integrada em um cluster. Essa camada de automação elimina a necessidade de gerenciamento manual, aumenta a disponibilidade dos serviços e garante que o sistema permaneça operacional mesmo diante de falhas.

6.2 Arquitetura do Kubernetes

A arquitetura do Kubernetes é composta por um conjunto de componentes que trabalham em conjunto para garantir o funcionamento autônomo do cluster. Os principais são:

- **Control Plane (Master Node)** — responsável por orquestrar e supervisionar o cluster.
- **Kube-API Server** — ponto central de controle que processa comandos e atualizações.
- **Scheduler** — decide em qual nó cada *pod* será executado, com base na disponibilidade de recursos.
- **Controller Manager** — garante que o estado atual do cluster corresponda ao estado desejado.
- **Kubelet** — agente instalado em cada nó, responsável por executar e monitorar os contêineres.
- **Etcd** — banco de dados chave-valor que armazena o estado global do cluster.
- **Kube-Proxy** — gerencia a comunicação de rede e o balanceamento de carga entre os serviços.

O Kubernetes abstrai completamente a complexidade da infraestrutura, permitindo que o desenvolvedor foque na aplicação, enquanto o sistema garante a disponibilidade, resiliência e escalabilidade automaticamente.

6.3 Orquestração dos Microserviços da Aplicação

Para o experimento, foi utilizado o **Minikube**, uma distribuição leve do Kubernetes que executa um cluster local em um único nó. Cada componente da aplicação — Serviços A, B e Gateway P — foi empacotado em uma imagem Docker e implantado como um *Deployment*, sendo exposto por meio de um *Service*.

O Gateway P foi publicado externamente através de um *Ingress*, acessível pelos domínios locais configurados em `/etc/hosts`.

Os principais objetos definidos para o cluster foram:

- **Namespace:** `pspd`, responsável por isolar os recursos da aplicação.
- **Deployments:** definem as réplicas e as imagens dos serviços A, B e P.
- **Services:** expõem os pods internamente via DNS do Kubernetes, possibilitando comunicação direta.
- **Ingress:** expõe o Gateway P externamente sob os domínios `pspd.local` e `pspd-rest.local`.

A implantação do cluster foi realizada automaticamente pelo script `setup.sh`, que executa os seguintes comandos:

```
kubectl apply -f k8s/namespace.yaml
kubectl apply -f k8s/a.yaml
kubectl apply -f k8s/b.yaml
kubectl apply -f k8s/p.yaml
kubectl apply -f k8s/ingress.yaml
```

Cada `.yaml` define as especificações de imagem, portas, variáveis de ambiente, políticas de reinício e sondas de disponibilidade.

6.4 Gerenciamento e Monitoramento do Cluster

Durante os testes, foram utilizados comandos nativos do Kubernetes para inspeção e acompanhamento do estado do cluster:

```
kubectl get pods -n pspd
kubectl get svc -n pspd
kubectl logs deploy/p-deploy -n pspd
```

Esses comandos permitiram validar a criação correta dos pods, verificar o balanceamento interno e acompanhar os logs de execução dos microserviços.

Além disso, foram configuradas as sondas **livenessProbe** e **readinessProbe**, que garantiram a reinicialização automática dos pods em caso de falha e impediram o roteamento de requisições antes da inicialização completa dos serviços.

6.5 Dificuldades e Ajustes Necessários

Durante a configuração e execução do cluster, foram enfrentados alguns desafios técnicos:

- **Falhas no Ingress Controller** — corrigidas com a reinstalação do complemento via `minikube addons enable ingress`.
- **Erros de resolução DNS interna** — solucionados reiniciando o pod `coredns`.
- **Latência inicial entre serviços A e B** — ajustada com parâmetros adequados nas sondas de readiness.
- **Tempo elevado de build das imagens** — otimizado ao utilizar o Docker interno do Minikube.

Esses ajustes garantiram estabilidade ao cluster e execução contínua dos pods durante os testes de carga e comparação entre protocolos.

6.6 Resultados e Observações Finais

A aplicação orquestrada apresentou comportamento estável e resiliente durante todos os testes realizados. O Kubernetes garantiu isolamento entre os pods, balanceamento automático de requisições e recuperação imediata em caso de falha de algum serviço.

A integração entre **Minikube**, **Docker** e **gRPC** demonstrou o funcionamento prático de um ecossistema distribuído completo, reforçando os conceitos de modularidade, escalabilidade e automação. Essa infraestrutura serviu como base para a análise de desempenho detalhada apresentada na próxima seção — **Resultados e Discussões**.

7 Resultados e Discussões

Esta seção apresenta a metodologia de testes de carga, os resultados experimentais obtidos e a análise comparativa entre os protocolos **gRPC** e **REST**. Os experimentos foram conduzidos no ambiente Kubernetes configurado previamente, permitindo avaliar o impacto da comunicação binária e do transporte HTTP/2 sobre o desempenho da aplicação distribuída.

7.1 Metodologia dos Testes

Os testes de desempenho foram realizados utilizando a ferramenta **k6**, com o objetivo de medir o tempo médio de resposta e o *throughput* (requisições por segundo) sob diferentes níveis de carga.

Cada execução simulou um número crescente de *usuários virtuais* (VUs), variando de 25 até 3200, durante 30 segundos de teste contínuo. As mesmas requisições foram realizadas tanto na versão gRPC quanto na versão REST, garantindo condições idênticas de ambiente e carga.

O fluxo geral dos testes foi:

1. Inicialização do cluster Kubernetes via Minikube e verificação do status dos pods.
2. Execução dos testes gRPC e REST em paralelo, utilizando os scripts:

```
k6 run load/load_grpc_http.js
k6 run load/load_rest_http.js
```

3. Registro dos resultados em arquivos `.txt` dentro do diretório `test_results/`.
4. Processamento automático dos dados e geração dos gráficos comparativos pelo script `plot_results.py`.

As métricas coletadas pelo k6 foram:

- **http_req_duration**: tempo médio de resposta de cada requisição;
- **http_reqs**: quantidade total de requisições processadas por segundo;
- **vus**: número de usuários virtuais simulados simultaneamente.

Essas métricas permitem observar a variação de latência e throughput de forma quantitativa, comparando o desempenho dos dois protocolos sob carga crescente.

7.2 Resultados Experimentais

A Tabela 4 resume os resultados obtidos para cada nível de carga testado.

Tabela 4: Comparativo de desempenho entre gRPC e REST sob diferentes cargas de usuários.

VUs	Tempo médio gRPC (ms)	Tempo médio REST (s)	Throughput gRPC (req/s)	Throughput REST (req/s)
25	76.13	28.21	342.29	0.44
50	150.9	28.43	351.05	0.88
100	292.57	28.41	348.52	1.76
200	549.61	28.24	362.04	3.53
400	1.07	28.37	359.29	7.05
800	2.14	28.35	345.54	14.12
1600	5.18	28.23	269.23	28.24
3200	10.48	28.26	242.82	56.76

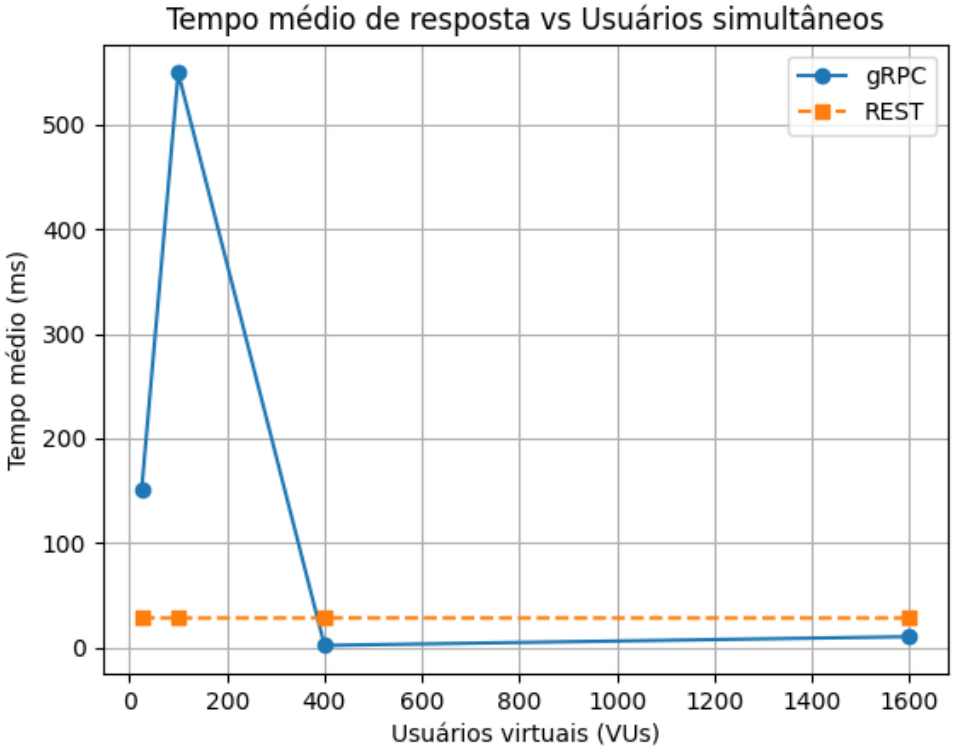

Figura 7: Tempo médio de resposta — comparação entre gRPC e REST.

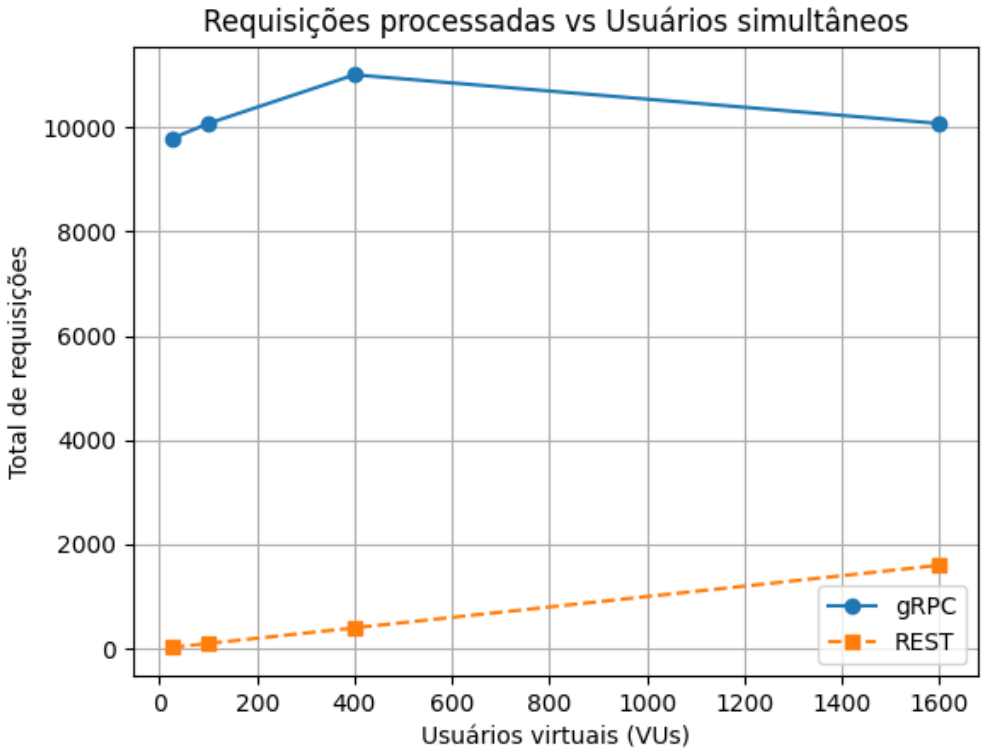

Figura 8: Requisições processadas por segundo — comparação entre gRPC e REST.

Os gráficos evidenciam um contraste nítido entre os protocolos: o gRPC manteve tempos de resposta na ordem de milissegundos mesmo sob carga intensa, enquanto o REST permaneceu estável, porém com lat

8 Conclusão

O desenvolvimento deste laboratório proporcionou uma compreensão prática e integrada dos principais pilares que sustentam a computação paralela e distribuída: **virtualização**, **comunicação entre microserviços**, **containerização** e **orquestração**. Por meio da combinação de tecnologias como **KVM/QEMU**, **Libvirt**, **gRPC**, **Docker** e **Kubernetes**, foi possível compreender de forma completa o ciclo de implantação e gerenciamento de uma aplicação distribuída moderna — desde a infraestrutura virtual até a comunicação eficiente entre processos em um cluster orquestrado.

A etapa de **virtualização com KVM e QEMU** foi fundamental para simular uma infraestrutura real de múltiplos nós, permitindo o isolamento de ambientes e o controle preciso de recursos computacionais. O uso da API **Libvirt** e das ferramentas **virt-manager** e **virsh** facilitou a criação e o monitoramento das máquinas virtuais, enquanto o estudo comparativo entre os firmwares **SeaBIOS** e **coreboot** proporcionou um entendimento mais profundo do processo de inicialização dos sistemas convidados.

Na sequência, a **implementação da aplicação distribuída baseada em gRPC** demonstrou, na prática, como a comunicação binária de alta performance supera as abordagens REST tradicionais. O **Gateway P**, atuando como intermediário entre o cliente web e os microserviços A e B, evidenciou os benefícios do uso do **Protocol Buffers** e do **HTTP/2**, reduzindo latência e melhorando o aproveitamento de rede. A versão alternativa em REST serviu como base comparativa, revelando limitações de desempenho e reforçando a superioridade técnica do gRPC em cenários de alta concorrência.

A posterior **containerização com Docker** trouxe portabilidade e modularidade, permitindo empacotar e executar cada serviço de forma isolada e reprodutível. Com a **orquestração via Kubernetes (Minikube)**, o sistema atingiu um novo patamar de automação e escalabilidade. O uso de *Deployments*, *Services* e *Ingress Controllers* garantiu alta disponibilidade e possibilitou observar o comportamento autônomo e resiliente do cluster em situações de falha.

8.1 Principais Dificuldades e Soluções

Durante o desenvolvimento, foram enfrentados desafios em diferentes camadas do sistema, desde a infraestrutura até a comunicação entre serviços. Os principais foram:

- **Configuração de rede entre VMs** — solucionada com a criação manual da bridge `virbr0` e atribuição de endereços IP estáticos.
- **Erros na compilação dos stubs gRPC** — resolvidos com o uso da variável de ambiente `PYTHONPATH=.` e a recompilação dos arquivos `.proto`.
- **Integração entre o Gateway HTTP e os microserviços gRPC** — corrigida com o uso das bibliotecas `@grpc/grpc-js` e `@grpc/proto-loader`.
- **Instabilidade do Ingress Controller no Minikube** — solucionada com a reinstalação dos complementos e ajustes nas sondas de `readiness` e `liveness`.

Cada dificuldade representou uma oportunidade de aprofundar o entendimento sobre o funcionamento interno de sistemas distribuídos — desde o hipervisor até as camadas de comunicação entre processos — fortalecendo o aprendizado prático do grupo.

8.2 Pontos de Melhoria e Perspectivas Futuras

Embora os resultados obtidos tenham sido satisfatórios, há possibilidades de aprimoramento que podem aproximar o projeto de um ambiente real de produção:

- **Automatização do pipeline de implantação**, utilizando ferramentas de integração e entrega contínua (CI/CD), como GitLab CI ou Jenkins.
- **Monitoramento e observabilidade** com **Prometheus** e **Grafana**, permitindo a coleta de métricas em tempo real e a análise de desempenho dos pods e serviços.
- **Implementação de segurança avançada**, com autenticação mútua e criptografia TLS nas comunicações gRPC.
- **Exploração de autoescalonamento**, via *Horizontal Pod Autoscaler*, para adaptação dinâmica da carga de trabalho.
- **Integração de um banco de dados distribuído**, como MongoDB ou CockroachDB, para persistência de dados e expansão da arquitetura para um sistema backend completo.

Essas melhorias aumentariam a robustez e a escalabilidade da aplicação, tornando o ambiente experimental mais próximo de uma solução corporativa moderna.

8.3 Considerações Finais

O trabalho atingiu plenamente seus objetivos ao integrar teoria e prática na construção de um ecossistema distribuído completo — desde a virtualização de servidores até a orquestração de microserviços em cluster. Os conhecimentos adquiridos abrangeram múltiplas áreas da Engenharia de Software: redes, sistemas operacionais, virtualização, comunicação entre processos, containerização, orquestração e análise de desempenho.

Mais do que um exercício técnico, este projeto configurou-se como um verdadeiro **laboratório integrado de engenharia de sistemas distribuídos**, no qual cada tecnologia — **KVM**, **gRPC**, **Docker** e **Kubernetes** — contribuiu para consolidar os princípios de escalabilidade, modularidade e resiliência. Ao final, a equipe alcançou uma compreensão sólida e aplicada sobre como essas ferramentas se integram para sustentar infraestruturas de software modernas, caracterizadas por eficiência, automação e confiabilidade — pilares centrais da computação distribuída contemporânea.

9 Aprendizados e Contribuições Individuais

O desenvolvimento deste trabalho exigiu cooperação contínua entre os integrantes da equipe, tanto no planejamento quanto na execução das etapas técnicas e na elaboração deste relatório. Cada membro contribuiu com conhecimentos específicos e desempenhou um papel essencial na implementação, análise e documentação da aplicação distribuída. A seguir, são apresentadas as principais responsabilidades e os aprendizados adquiridos por cada participante do grupo.

Débora Caires de Souza Moreira — Matrícula 22/2015103

Débora teve papel fundamental na **configuração do ambiente de virtualização**, atuando diretamente na criação e gerenciamento das máquinas virtuais por meio das ferramentas **KVM**, **QEMU** e **virt-manager**. Foi responsável pela realização e documentação dos testes de firmware, comparando o desempenho e a compatibilidade entre **SeaBIOS** e **coreboot**. Além disso, contribuiu para a elaboração dos diagramas de topologia e para a descrição detalhada das etapas de rede e orquestração no relatório.

Durante o desenvolvimento, aprofundou seus conhecimentos sobre:

- O funcionamento interno do hipervisor **KVM** e sua integração com a API **Libvirt**.
- A criação, monitoramento e manutenção de VMs utilizando interfaces gráficas e linha de comando.
- A documentação técnica de ambientes virtualizados e topologias de rede.

Edilberto Almeida Cantuaria — Matrícula 22/2014984

Edilberto foi o principal responsável pelo **desenvolvimento da aplicação distribuída** e pela integração entre os módulos do sistema. Implementou o **Gateway P** em Node.js, responsável por interligar o cliente web aos microserviços via gRPC, além de desenvolver os serviços A e B em Python, com chamadas unárias e de *server streaming*. Também foi autor dos **Dockerfiles**, dos manifestos **Kubernetes (YAML)** e do script de automação `setup.sh`, que realiza o build e o deploy completo no Minikube. Contribuiu ainda para a redação das seções teóricas sobre gRPC, Kubernetes e Conclusão.

Os principais aprendizados obtidos incluem:

- Domínio prático da comunicação gRPC e integração entre linguagens e protocolos distintos.
- Experiência completa no ciclo de desenvolvimento distribuído: implementação, empacotamento, orquestração e deploy.
- Compreensão aprofundada do funcionamento interno do Kubernetes e sua aplicação em microserviços escaláveis.
- Aprimoramento das habilidades em documentação técnica, padronização e formatação de relatórios acadêmicos.

Levi de Oliveira Queiroz — Matrícula 17/0108341

Levi foi responsável pela **configuração e validação da infraestrutura de rede** das máquinas virtuais e dos contêineres. Garantiu a comunicação correta entre os serviços A, B e o Gateway P, ajustando parâmetros de portas, interfaces e endereços IP. Contribuiu para o desenho do diagrama de topologia de rede e auxiliou na depuração de falhas de comunicação entre pods e VMs. Durante a fase de testes, apoiou a execução das requisições HTTP e gRPC, bem como a análise de logs e tempos de resposta.

Entre os aprendizados adquiridos, destacam-se:

- Entendimento detalhado da configuração de redes virtuais (*bridge* e NAT) em ambientes KVM/QEMU.

- Integração entre múltiplas camadas de rede — virtual, de contêineres e de cluster Kubernetes.
- Aplicação de técnicas de troubleshooting e análise de desempenho em sistemas distribuídos.
- Valorização da importância da padronização e sincronização entre serviços para o funcionamento estável do sistema.

Wolfgang Friedrich Stein — Matrícula 23/1032121

Wolfgang concentrou-se na **análise e validação dos resultados experimentais**. Foi responsável pela execução dos testes de desempenho e pela comparação entre os protocolos **gRPC** e **REST**, utilizando a ferramenta `k6` para medir latência, throughput e consumo de recursos. Contribuiu também para a revisão técnica e textual do relatório final, garantindo consistência entre as seções e clareza na apresentação dos resultados.

Durante sua atuação, desenvolveu aprendizados importantes relacionados a:

- Interpretação de métricas de desempenho e identificação de gargalos em sistemas distribuídos.
- Análise comparativa entre diferentes arquiteturas de comunicação (HTTP/REST e HTTP/2/gRPC).
- Boas práticas de escrita técnica e estruturação de relatórios científicos.
- Colaboração e versionamento em ambiente técnico multidisciplinar.

10 APÊNDICE - CÓDIGOS FONTE

Nesta seção, estão apresentados os códigos fonte utilizados para implementação do Jogo da Vida em diferentes paradigmas de paralelismo.

10.1 gateway_p_node

10.1.1 index.html

```
<!doctype html>
<html>
  <head><meta charset="utf-8"><title>PSPD gRPC Gateway</title></head>
  <body style="font-family:system-ui;margin:2rem">
    <h1>PSPD    HTTP -> gRPC</h1>
    <div style="border:1px solid ...
        #333;padding:1rem;border-radius:12px;margin-bottom:1rem">
      <h2>Service A    unary</h2>
      <input id="name" placeholder="Seu nome" />
      <button onclick="hello()">Chamar /a/hello</button>
      <pre id="outA"></pre>
    </div>
    <div style="border:1px solid #333;padding:1rem;border-radius:12px">
      <h2>Service B    server-stream</h2>
      <input id="count" type="number" value="5" />
      <input id="delay" type="number" value="0" />
      <button onclick="nums()">Chamar /b/numbers</button>
      <pre id="outB"></pre>
    </div>
    <script>
      async function hello() {
        const name = document.getElementById('name').value || 'mundo';
        const r = await ...
            fetch(`/a/hello?name=${encodeURIComponent(name)}`);
        document.getElementById('outA').textContent = ...
            JSON.stringify(await r.json(), null, 2);
      }
      async function nums() {
        const count = document.getElementById('count').value || 5;
        const delay_ms = document.getElementById('delay').value || 0;
        const r = await ...
            fetch(`/b/numbers?count=${count}&delay_ms=${delay_ms}`);
        document.getElementById('outB').textContent = ...
            JSON.stringify(await r.json(), null, 2);
      }
    </script>
  </body>
</html>
```

10.1.2 Dockerfile

```
FROM node:20-slim
WORKDIR /app

# 1) Instalar deps com cache
```

```
COPY gateway_p_node/package*.json ./
RUN npm install --omit=dev

# 2) Copiar protos e c digo do gateway
COPY proto/ ./proto/
COPY gateway_p_node/ .

ENV PORT=8080
EXPOSE 8080
CMD ["node","server.js"]
```

10.1.3 package.json

```
{
  "name": "gateway-p-node",
  "version": "1.0.0",
  "main": "server.js",
  "type": "module",
  "dependencies": {
    "@grpc/grpc-js": "^1.11.3",
    "@grpc/proto-loader": "^0.7.13",
    "express": "^4.19.2",
    "morgan": "^1.10.0",
    "cors": "^2.8.5"
  }
}
```

10.1.4 server.js

```
import express from "express";
import cors from "cors";
import morgan from "morgan";
import * as grpc from "@grpc/grpc-js";
import * as protoLoader from "@grpc/proto-loader";
import path from "path";
import { fileURLToPath } from "url";

const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

const PORT = process.env.PORT || 8080;
const A_ADDR = process.env.A_ADDR || "localhost:50051";
const B_ADDR = process.env.B_ADDR || "localhost:50052";

const PROTO_PATH = path.join(__dirname, "proto/services.proto");
const packageDefinition = protoLoader.loadSync(PROTO_PATH, { ...
    keepCase: true, longs: String, enums: String, defaults: true, ...
    oneofs: true });
const proto = grpc.loadPackageDefinition(packageDefinition).pspd;

const clientA = new proto.ServiceA(A_ADDR, ...
    grpc.credentials.createInsecure());
const clientB = new proto.ServiceB(B_ADDR, ...
    grpc.credentials.createInsecure());
```

```
22
23 const app = express();
24 app.use(cors());
25 app.use(morgan("dev"));
26 app.use(express.json());
27
28 app.get("/", (req, res) => res.sendFile(path.join(__dirname, ...
       "public/index.html")));
29
30 app.get("/a/hello", (req, res) => {
31   const name = req.query.name || "mundo";
32   clientA.SayHello({ name }, (err, reply) => {
33     if (err) return res.status(500).json({ error: err.message });
34     res.json({ from: "A", message: reply.message });
35   });
36 });
37
38 app.get("/b/numbers", (req, res) => {
39   const count = parseInt(req.query.count || "5", 10);
40   const delay_ms = parseInt(req.query.delay_ms || "0", 10);
41   const call = clientB.StreamNumbers({ count, delay_ms });
42   const values = [];
43   call.on("data", (chunk) => values.push(chunk.value));
44   call.on("error", (err) => res.status(500).json({ error: err.message ...
       }));
45   call.on("end", () => res.json({ from: "B", values }));
46 });
47
48 app.get("/healthz", (_, res) => res.send("ok"));
49
50 app.listen(PORT, () => {
51   console.log(`Gateway P listening on :${PORT}`);
52   console.log(`Using A at ${A_ADDR} and B at ${B_ADDR}`);
53 });
```

10.2 gateway_p_rest_node

10.2.1 index.html

```
1  <!doctype html>
2  <html>
3    <head>
4      <meta charset="utf-8">
5      <title>PSPD REST Gateway</title>
6      <meta name="viewport" content="width=device-width,initial-scale=1">
7      <style>
8        body { font-family: system-ui, -apple-system, Segoe UI, Roboto, ...
             Arial, sans-serif; margin: 2rem; }
9        h1 { margin-bottom: 0.5rem; }
10       .card { border: 1px solid #333; padding: 1rem; border-radius: ...
             12px; margin-bottom: 1rem; }
11       input, button { font-size: 1rem; padding: .5rem; }
12       pre { background: #111; color: #eee; padding: 1rem; ...
             border-radius: 10px; overflow:auto; }
13     </style>
14   </head>
```

```
15    <body>
16      <h1>PSPD    HTTP     REST</h1>
17      <div class="card">
18        <h2>Service A-REST</h2>
19        <input id="name" placeholder="Seu nome" />
20        <button onclick="hello()">Chamar /a/hello</button>
21        <pre id="outA"></pre>
22      </div>
23      <div class="card">
24        <h2>Service B-REST</h2>
25        <input id="count" type="number" value="5" />
26        <input id="delay" type="number" value="0" />
27        <button onclick="nums()">Chamar /b/numbers</button>
28        <pre id="outB"></pre>
29      </div>
30      <script>
31        async function hello() {
32          const name = document.getElementById('name').value || 'mundo';
33          const r = await ...
                fetch(`/a/hello?name=${encodeURIComponent(name)}`);
34          const j = await r.json();
35          document.getElementById('outA').textContent = ...
                JSON.stringify(j, null, 2);
36        }
37        async function nums() {
38          const count = document.getElementById('count').value || 5;
39          const delay_ms = document.getElementById('delay').value || 0;
40          const r = await ...
                fetch(`/b/numbers?count=${count}&delay_ms=${delay_ms}`);
41          const j = await r.json();
42          document.getElementById('outB').textContent = ...
                JSON.stringify(j, null, 2);
43        }
44      </script>
45    </body>
46  </html>
```

10.2.2 Dockerfile

```
1  FROM node:20-slim
2  WORKDIR /app
3
4  # 1) Instalar deps com cache
5  COPY gateway_p_rest_node/package*.json ./
6  RUN npm install --omit=dev
7
8  # 2) Copiar c digo do servi o
9  COPY gateway_p_rest_node/ .
10
11 ENV PORT=8081
12 EXPOSE 8081
13 CMD ["node", "server.js"]
```

10.2.3 package.json

```
{
  "name": "gateway-p-rest-node",
  "version": "1.0.0",
  "main": "server.js",
  "type": "module",
  "dependencies": {
    "express": "^4.19.2",
    "morgan": "^1.10.0",
    "cors": "^2.8.5",
    "node-fetch": "^3.3.2"
  }
}
```

10.2.4 server.js

```
import express from "express";
import cors from "cors";
import morgan from "morgan";
import fetch from "node-fetch";
import path from "path";
import { fileURLToPath } from "url";

const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

const PORT = process.env.PORT || 8081;
const A_REST = process.env.A_REST || "http://localhost:50061";
const B_REST = process.env.B_REST || "http://localhost:50062";

const app = express();
app.use(cors());
app.use(morgan("dev"));
app.use(express.json());

app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "public/index.html"));
});

app.get("/a/hello", async (req, res) => {
  const name = encodeURIComponent(req.query.name || "mundo");
  const r = await fetch(`${A_REST}/a/hello?name=${name}`);
  const j = await r.json();
  res.json(j);
});

app.get("/b/numbers", async (req, res) => {
  const count = encodeURIComponent(req.query.count || "5");
  const delay_ms = encodeURIComponent(req.query.delay_ms || "0");
  const r = await ...
      fetch(`${B_REST}/b/numbers?count=${count}&delay_ms=${delay_ms}`);
  const j = await r.json();
  res.json(j);
});

app.get("/healthz", (_, res) => res.send("ok"));
```

```
40
41 app.listen(PORT, () => {
42   console.log(`Gateway P-REST listening on :${PORT}`);
43   console.log(`Using A-REST at ${A_REST} and B-REST at ${B_REST}`);
44 });
```

10.3 load

10.3.1 load_grpc_http.js

```
1 import http from 'k6/http';
2 export const options = { vus: 100, duration: '30s' };
3 export default function () {
4   http.get('http://localhost:8080/a/hello?name=pspd');
5   http.get('http://localhost:8080/b/numbers?count=10&delay_ms=5');
6 }
```

10.3.2 load_rest_http.js

```
1 import http from 'k6/http';
2 export const options = { vus: 100, duration: '30s' };
3 export default function () {
4   http.get('http://localhost:8081/a/hello?name=pspd');
5   http.get('http://localhost:8081/b/numbers?count=10&delay_ms=5');
6 }
```

10.4 proto

10.4.1 services.proto

```
1 syntax = "proto3";
2
3 package pspd;
4
5 message HelloRequest { string name = 1; }
6 message HelloReply { string message = 1; }
7
8 service ServiceA {
9   rpc SayHello(HelloRequest) returns (HelloReply);
10 }
11
12 message StreamRequest { int32 count = 1; int32 delay_ms = 2; }
13 message NumberReply { int32 value = 1; }
14
15 service ServiceB {
16   rpc StreamNumbers(StreamRequest) returns (stream NumberReply);
17 }
```

10.5 services

10.5.1 a_py

Dockerfile

```
FROM python:3.12-slim
WORKDIR /app

# Dependências do gRPC (cache melhor)
RUN pip install --no-cache-dir grpcio==1.63.0 grpcio-tools==1.63.0

# Copia proto e código do serviço a partir da RAIZ do repo
COPY proto/ /app/proto/
COPY services/a_py/ /app/a_py/

# Gera stubs para /app/proto
RUN python -m grpc_tools.protoc -Iproto \
    --python_out=./proto \
    --grpc_python_out=./proto \
    proto/services.proto

# Torna importável
RUN touch /app/proto/__init__.py
ENV PYTHONPATH=/app:/app/proto

EXPOSE 50051
CMD ["python","a_py/server.py"]
```

requirements.txt

```

grpcio==1.63.0
grpcio-tools==1.63.0
```

server.py

```

import grpc
from concurrent import futures
import time, os

from proto import services_pb2, services_pb2_grpc

class ServiceAImpl(services_pb2_grpc.ServiceAServicer):
    def SayHello(self, request, context):
        name = request.name or "world"
        return services_pb2.HelloReply(message=f"Olá, {name}! [A]")

def serve():
    port = int(os.environ.get("PORT", "50051"))
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    services_pb2_grpc.add_ServiceAServicer_to_server(ServiceAImpl(), ...
        server)
    server.add_insecure_port(f"[::]:{port}")
```

```
    server.start()
    print(f"Service A listening on :{port}", flush=True)
    try:
        while True:
            time.sleep(86400)
    except KeyboardInterrupt:
        server.stop(0)

if __name__ == "__main__":
    serve()
```

10.5.2 a_rest

Dockerfile

```
FROM python:3.12-slim
WORKDIR /app

# Copia s   o requirements do servi o (cache)
COPY services/a_rest/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Agora o c digo do servi o
COPY services/a_rest/ .

# Se usar os protos no REST, descomente:
# COPY proto/ ./proto/
# RUN touch /app/proto/__init__.py
# ENV PYTHONPATH=/app:/app/proto

EXPOSE 8000
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

requirements.txt

```

fastapi==0.115.5
uvicorn==0.32.0
```

main.py

```

from fastapi import FastAPI
from fastapi.responses import JSONResponse

app = FastAPI(title="A-REST")

@app.get("/a/hello")
def hello(name: str = "mundo"):
    return JSONResponse({"message": f"Ol  , {name}! [A-REST]"})
```

10.5.3 b_py

Dockerfile

```
FROM python:3.12-slim
WORKDIR /app

# 1) Dependncias (melhor cache)
RUN pip install --no-cache-dir grpcio==1.63.0 grpcio-tools==1.63.0

# 2) Copia proto e o cdigo do servio a partir da RAIZ do repo
COPY proto/ /app/proto/
COPY services/b_py/ /app/b_py/

# 3) Gera stubs em /app/proto
RUN python -m grpc_tools.protoc -Iproto \
    --python_out=./proto \
    --grpc_python_out=./proto \
    proto/services.proto

# 4) Torna importvel
RUN touch /app/proto/__init__.py
ENV PYTHONPATH=/app:/app/proto

EXPOSE 50052
CMD ["python","b_py/server.py"]
```

requirements.txt

```

grpcio==1.63.0
grpcio-tools==1.63.0
```

server.py

```

import grpc
from concurrent import futures
import time, os

from proto import services_pb2, services_pb2_grpc

class ServiceBImpl(services_pb2_grpc.ServiceBServicer):
    def StreamNumbers(self, request, context):
        count = request.count if request.count > 0 else 5
        delay_ms = request.delay_ms if request.delay_ms > 0 else 0
        for i in range(1, count + 1):
            yield services_pb2.NumberReply(value=i)
            if delay_ms > 0:
                time.sleep(delay_ms/1000.0)

def serve():
    port = int(os.environ.get("PORT", "50052"))
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
```

```
20      services_pb2_grpc.add_ServiceBServicer_to_server(ServiceBImpl(), ...
            server)
21      server.add_insecure_port(f"[::]:{port}")
22      server.start()
23      print(f"Service B listening on :{port}", flush=True)
24      try:
25          while True:
26              time.sleep(86400)
27      except KeyboardInterrupt:
28          server.stop(0)
29
30  if __name__ == "__main__":
31      serve()
```

10.5.4 b_rest

Dockerfile

```
1  FROM python:3.12-slim
2  WORKDIR /app
3
4  # 1) Instala deps espec ficas do servi o REST B
5  COPY services/b_rest/requirements.txt .
6  RUN pip install --no-cache-dir -r requirements.txt
7
8  # 2) Copia  s   o c digo do servi o
9  COPY services/b_rest/ .
10
11 # (Opcional) Se o REST B usar os protos, descomente:
12 # COPY proto/ ./proto/
13 # RUN touch /app/proto/__init__.py
14 # ENV PYTHONPATH=/app:/app/proto
15
16 EXPOSE 8000
17 CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

requirements.txt

```
1
2  fastapi==0.115.5
3  uvicorn==0.32.0
```

main.py

```
1
2  from fastapi import FastAPI
3  from fastapi.responses import JSONResponse
4  import time
5
6  app = FastAPI(title="B-REST")
7
8  @app.get("/b/numbers")
9  def numbers(count: int = 5, delay_ms: int = 0):
```

```
10      count = max(0, int(count))
11      delay_ms = max(0, int(delay_ms))
12      out = []
13      for i in range(1, count + 1):
14          out.append(i)
15          if delay_ms > 0:
16              time.sleep(delay_ms/1000.0)
17      return JSONResponse({"values": out})
```

Referências

[1] BURNS, B.; BEDA, J.; HIGHTOWER, K. *Kubernetes: Up and Running*. 3rd. ed. [S.l.]: O'Reilly Media, 2023.

[2] TANENBAUM, A. S.; STEEN, M. van. *Distributed Systems: Principles and Paradigms*. 2nd. ed. [S.l.]: Pearson Prentice Hall, 2007. ISBN 9780132392273.

[3] Expertos Tech. *O que é gRPC? Seus componentes, RPC e HTTP/2 (Parte 1)*. 2022. Acessado em: 7 de outubro de 2025. Disponível em: <https://dev.to/expertostech/o-que-e-grpc-seus-componentes-rpc-e-http2-parte-1-nnm>.

[4] BHANDARI, P. Performance comparison between rest and grpc in microservices communication. *International Journal of Computer Applications*, v. 183, n. 32, p. 15–21, 2021.

[5] BHADORIYA, A. *Understanding gRPC*. Medium, 2020. Acessado em: 7 de outubro de 2025. Disponível em: <https://medium.com/geekculture/understanding-grpc-a142b1a7a401>.

[6] The Kubernetes Authors. *Connecting Applications with Services*. 2024. `https://kubernetes.io/docs/concepts/services-networking/service/`. Acessado em: 7 de outubro de 2025.