UNIVERSIDADE DE BRASÍLIA

Faculdade do Gama

Documentação Laboratório de Virtualização

ACADÊMICO:

Arthur Grandão de Mello

Bruno Martins Valério Bomfim

Douglas Alves dos Santos

Gabriel Campello Marques

Geovanna Maciel Avelino da Costa

DOCENTE:

Fernando William Cruz

Brasília - DF

2025

Estudo do GPC

O gRPC é um framework de comunicação criado pelo Google que permite que aplicações distribuídas se comuniquem de forma rápida e eficiente. Ele utiliza o protocolo HTTP/2 para transportar mensagens e o Protocol Buffers (Protobuf) como linguagem de definição de serviços e estruturas de dados.O HTTP/2 fornece funcionalidades como comunicação multiplexada, compressão de cabeçalhos e suporte a comunicação bidirecional, o que o torna ideal para aplicações modernas e em tempo real. Já o Protobuf define como as mensagens são organizadas e fornece as instruções para gerar o código de cliente e servidor em várias linguagens.

O gRPC oferece vantagens significativas em relação a abordagens tradicionais como Web Services baseados em SOAP e REST APIs. Enquanto os Web Services costumam usar XML e protocolos como SOAP sobre HTTP, o que pode gerar comunicações mais pesadas e verbosas, o gRPC utiliza Protobuf (Protocol Buffers), um formato binário mais compacto e eficiente. Em relação às REST APIs — que normalmente usam JSON sobre HTTP/1.1 —, o gRPC adota o HTTP/2, que possibilita comunicação multiplexada, streaming bidirecional e compressão de cabeçalhos, resultando em melhor desempenho, especialmente em sistemas de tempo real ou de alta frequência de chamadas.

Além disso, o gRPC oferece geração automática de código para diversas linguagens a partir dos arquivos `.proto`, promovendo forte tipagem, padronização e agilidade no desenvolvimento. Em contrapartida, REST APIs são mais simples de consumir em navegadores e amplamente suportadas, o que ainda as torna uma boa escolha para sistemas web convencionais. Assim, a escolha entre gRPC, REST ou Web Services depende dos requisitos específicos do sistema, como desempenho, compatibilidade, facilidade de desenvolvimento e suporte a diferentes padrões de comunicação.

Em `.proto,` por exemplo, o serviço abaixo define um método chamado `SayHello`, que recebe um nome e responde com uma saudação:

```
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
```

```
}
```

Em Python essse programa seria:

```python
import grpc
from concurrent import futures
import hello_pb2
import hello_pb2_grpc

class GreeterServicer(hello_pb2_grpc.GreeterServicer):
      def SayHello(self, request, context):
      return hello_pb2.HelloReply(message=f"Olá, {request.name}!")
server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
hello_pb2_grpc.add_GreeterServicer_to_server(GreeterServicer(),
server)
server.add_insecure_port('[::]:50051')
server.start()
server.wait_for_termination()
```

E o cliente correspondente seria:

```python
import grpc
import hello_pb2
import hello_pb2_grpc

channel = grpc.insecure_channel('localhost:50051')
stub = hello_pb2_grpc.GreeterStub(channel)
response = stub.SayHello(hello_pb2.HelloRequest(name='Alice'))
print(response.message)
```

Esse é um exemplo de chamada **unária**, onde o cliente envia uma única requisição e recebe uma única resposta. Além disso, o gRPC permite outros tipos de comunicação:

- **Server-streaming**: o cliente envia uma requisição e o servidor envia várias respostas em sequência. Ideal para relatórios ou atualizações em tempo real. No exemplo abaixo, o servidor envia a hora atual várias vezes, e o cliente vai recebendo os dados em fluxo.

```
#Servidor:
import grpc
from concurrent import futures
import time
from datetime import datetime
import stream_pb2
import stream_pb2_grpc

class StreamerServicer(stream_pb2_grpc.StreamerServicer):
      def StreamTime(self, request, context):
      for _ in range(request.count):
            now = datetime.now().strftime("%H:%M:%S")
            yield stream_pb2.TimeReply(time=now)
            time.sleep(1)

server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
stream_pb2_grpc.add_StreamerServicer_to_server(StreamerServicer(),
server)
server.add_insecure_port('[::]:50051')
server.start()
print("Server-streaming iniciado.")
server.wait_for_termination()

#Cliente:

import grpc
import stream_pb2
import stream_pb2_grpc

channel = grpc.insecure_channel('localhost:50051')
stub = stream_pb2_grpc.StreamerStub(channel)

request = stream_pb2.TimeRequest(count=5)
response = stub.StreamTime(request)

for reply in response:
      print("Hora:", reply.time)
```

- **Client-streaming**: o cliente envia vários dados, e o servidor responde apenas no final. Útil quando se quer processar um lote de informações de uma vez, como uma média de valores.

```
import grpc
import average_pb2
import average_pb2_grpc

channel = grpc.insecure_channel('localhost:50052')
stub = average_pb2_grpc.AveragerStub(channel)

def generate_numbers():
      for n in [10.0, 20.0, 30.0, 40.0]:
      yield average_pb2.Number(value=n)

response = stub.ComputeAverage(generate_numbers())
print("Média calculada:", response.result)
```

- **Bidirectional-streaming**: cliente e servidor trocam dados livremente, útil para chats, jogos, ou serviços de monitoramento contínuo. Esse tipo de comunicação permite que as duas partes conversem em tempo real, sem precisar encerrar a comunicação para aguardar uma resposta.

```
#Servidor:
import grpc
from concurrent import futures
import chat_pb2
import chat_pb2_grpc

class ChatServicer(chat_pb2_grpc.ChatServicer):
      def ChatStream(self, request_iterator, context):
      for msg in request_iterator:
            print(f"{msg.user} diz: {msg.message}")
            yield chat_pb2.ChatMessage(user="Servidor",
message=f"Recebido: {msg.message}")

server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
chat_pb2_grpc.add_ChatServicer_to_server(ChatServicer(), server)
server.add_insecure_port('[::]:50053')
server.start()
print("Chat bidirecional iniciado.")
server.wait_for_termination()


#Cliente:
```

```python
import grpc
import chat_pb2
import chat_pb2_grpc
import time

channel = grpc.insecure_channel('localhost:50053')
stub = chat_pb2_grpc.ChatStub(channel)

def send_messages():
      mensagens = ["Oi", "Como vai?", "Até logo"]
      for msg in mensagens:
      yield chat_pb2.ChatMessage(user="Alice", message=msg)
      time.sleep(1)

responses = stub.ChatStream(send_messages())

for response in responses:
      print(f"{response.user}: {response.message}")
```

Conclui-se que cada tipo de chamada do gRPC é mais adequado a determinados cenários. As chamadas do tipo **unária** são ideais para operações simples e pontuais, como consultas ou requisições únicas em que o cliente envia uma mensagem e recebe uma resposta direta. O modelo **server-streaming** é mais apropriado quando o cliente precisa receber uma sequência de dados do servidor, como no caso de logs contínuos ou atualizações em tempo real. Já o **client-streaming** é útil quando o cliente precisa enviar múltiplas mensagens ao servidor e aguarda apenas uma resposta consolidada ao final, como no envio de um conjunto de valores para o cálculo de uma média. Por fim, a comunicação **bidirecional** é essencial em aplicações que demandam interação constante e simultânea entre cliente e servidor, como em sistemas de chat, jogos ou monitoramento contínuo de eventos.

Relatório Parte Prática

Metodologia

A metodologia adotada no desenvolvimento do projeto foi uma adaptação do Scrum, combinada com a prática de *pair programming*. Todos os integrantes atuaram em duplas, promovendo colaboração contínua e troca de conhecimento entre os membros do grupo. Essa abordagem permitiu não apenas uma divisão equilibrada das tarefas, mas também um suporte mútuo, contribuindo para a superação de dificuldades técnicas ao longo do processo. Como ponto de partida, foi elaborado o diagrama apresentado na Imagem 1, que serviu como guia estrutural para as etapas seguintes do projeto, orientando a implementação das funcionalidades e a organização dos componentes do sistema distribuído.

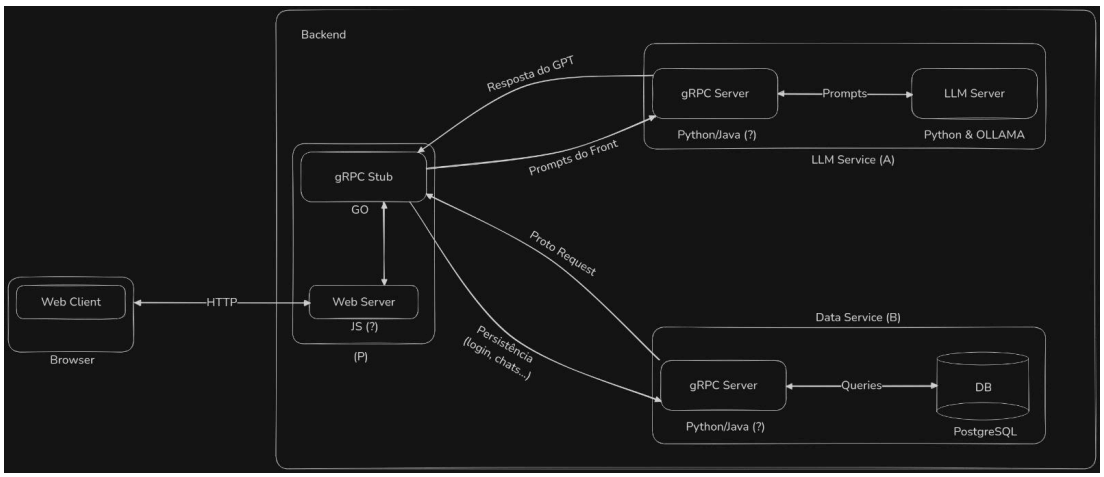

Imagem 1: Esquemático do projeto

Virtualização

A virtualização e o uso do Docker foram realizados em paralelo. Inicialmente, utilizou-se o Docker para executar todos os serviços de forma integrada. Houve uma tentativa de rodar o ambiente localmente na máquina virtual, instalando manualmente as dependências, mas, devido à falta de tempo, optou-se por passar o docker-file para dentro da VM, sem realizar uma separação completa dos serviços dentro da mesma.A virtualização de sistemas computacionais permite a criação de múltiplos ambientes isolados sobre um único hardware físico, promovendo melhor aproveitamento de recursos, flexibilidade e segurança.

No contexto de QEMU, libvirt, virt-manager e virsh, o QEMU atua como emulador de hardware e, quando acelerado pelo KVM, fornece virtualização de alta performance, enquanto o libvirt oferece uma camada de abstração e API unificada para controlar diferentes hipervisores. O virt-manager provê interface gráfica para orquestração de máquinas virtuais, e o virsh, ferramenta de linha de comando, possibilita automação e scripting de operações como criação, snapshot e migração de domínios. O monitor do QEMU (QEMU-Monitor ou QMP) expõe comandos avançados para inspeção e ajuste em tempo real das VMs, completando um ecossistema que atende tanto administradores gráficos quanto engenheiros de automação.

Num exemplo prático, um administrador pode, via virt-manager, criar uma nova VM selecionando uma imagem ISO de instalação, configurar CPU, memória e disco, e, após a instalação do sistema convidado, utilizar o virsh para gerar um snapshot antes de aplicar atualizações críticas. Caso uma atualização cause instabilidade, basta executar “virsh snapshot-revert” para restaurar o estado anterior. Para ajustes dinâmicos sem desligar a VM, conecta-se ao QEMU-Monitor e emite comandos QMP para adicionar CPUs ou discos em tempo real.

Referencial teórico

A emulação de hardware pelo QEMU permite executar sistemas convidados de arquiteturas diversas sem alterações no código-fonte, sendo acelerada pelo KVM para

desempenho quase nativo (FLINT, 2005)[1]. O libvirt padroniza o gerenciamento de hipervisores distintos por meio de uma API única, suportando linguagens como C, Python e Go (KHALID; SINGH, 2018)[2]. O virt-manager, cliente gráfico do libvirt, facilita criação e monitoramento de domínios com dashboards de CPU, memória e I/O (SMITH; JOHNSON, 2020)[3]. Já o virsh, utilitário CLI, permite automação via scripts, incluindo operações de lifecycle e snapshots (DOE; RICHARDS, 2019)[4]. O QEMU-Monitor implementa o Virtual Machine Monitor (VMM), responsável por mediar todas as operações entre hardware real e sistemas convidados, conforme modelo de Popek e Goldberg (POPEK; GOLDBERG, 1974)[5].

Funcionamento integrado

A orquestração inicia-se com o virt-manager, que, conectado ao demon libvirtd, cria definições XML de domínios. Essas definições são interpretadas pelo libvirt, que invoca o QEMU/KVM para instanciar o hardware virtual. O virsh pode manipular essas mesmas definições via linha de comando, permitindo integração a pipelines CI/CD. O QEMU-Monitor (acessível por socket QMP) fornece interface de baixo nível para ajustes em tempo de execução, como hot-plug de dispositivos e inspeção de registradores. Essa arquitetura em camadas garante modularidade, pois atualizações no libvirt não afetam diretamente o QEMU, e ferramentas de GUI ou CLI podem ser desenvolvidas independentemente (LEE; KIM, 2021)[6].

Exemplo descritivo de uso: imagine um laboratório de testes de kernels Linux. O pesquisador inicia o virt-manager, clica em “Create VM”, aponta para a ISO do novo kernel, define 4 GB de RAM e 2 vCPUs. Após a instalação, abre um terminal e executa:

```
virsh snapshot-create-as teste-kernel antes-patch "Snapshot antes de patch"
```

```
virsh list --all (confirma o snapshot)
```

Ao aplicar o patch, o kernel trava. Então, reverte com:

```
virsh snapshot-revert teste-kernel antes-patch
```

Para estudar comportamento de I/O, conecta-se ao QEMU-Monitor e emite:

```
device_add virtio-blk-pci... (adiciona disco virtual em tempo real)
```

Tudo isso ocorre sem desligar a VM, permitindo iterações rápidas e seguras.

Implementação no projeto

Durante o projeto de rede virtual com QEMU, criei três máquinas virtuais (VM1, VM2 e VM3) com o objetivo de simular um ambiente de rede segmentado em duas LANs distintas (LAN1 e LAN2). A VM1 atua como um API Gateway, responsável por fazer o roteamento entre as duas redes, enquanto as VMs 2 e 3 representam dois servidores backend que se comunicam entre si pela LAN2.

Primeiramente, foram criadas duas bridges virtuais com o comando brctl, chamadas br0 e br1. A bridge br0 representa a LAN1 e a br1, a LAN2. Após isso, levantei essas interfaces com ip link set br0 up e ip link set br1 up.

Para conectar as VMs às redes, foi necessário criar interfaces TAP com o comando ip tuntap add dev tapX mode tap, onde X varia de 0 a 3, correspondendo às VMs e suas interfaces. Cada TAP foi associada à sua respectiva bridge usando brctl addif e também ativada com ip link set.

A VM1 foi criada com duas interfaces de rede: uma ligada à tap0 (conectada à br0/LAN1) e outra à tap1 (conectada à br1/LAN2), permitindo que ela atue como roteador entre as duas redes. Já as VMs 2 e 3 foram configuradas com uma única interface cada, ligadas às TAPs tap2 e tap3, ambas conectadas à br1 (LAN2), para que pudessem se comunicar entre si.

Inicialmente, para instalar o Alpine Linux nas VMs 2 e 3, foi necessário conectá-las temporariamente a uma rede com acesso à internet, utilizando uma bridge que estivesse associada à interface real do host. Após a instalação, elas foram conectadas à br1.

Cada VM foi configurada manualmente com IP fixo no arquivo /etc/network/interfaces, atribuindo endereços como 10.0.0.1 para a interface eth1 da VM1, 10.0.0.2 para a VM2, e 10.0.0.3 para a VM3, todas com máscara 255.255.255.0. A gateway das VMs 2 e 3 foi definida como a interface da VM1 (10.0.0.1), permitindo que o roteamento funcione corretamente.

Finalizada a configuração, os testes de rede utilizando o comando ping confirmaram que as máquinas estavam corretamente configuradas e podiam se comunicar entre si pela LAN2, e que a VM1 era capaz de acessar ambas as redes.

Banco de Dados

O sistema foi desenvolvido com uma arquitetura distribuída baseada em microsserviços, onde cada componente possui responsabilidades bem definidas, garantindo escalabilidade e baixo acoplamento. A comunicação entre os módulos é realizada via gRPC, um framework de chamada de procedimento remoto (RPC) que utiliza HTTP/2 e Protocol

Buffers para troca eficiente de mensagens em formato binário. Essa abordagem permite alto desempenho e baixa latência nas operações do sistema.

O backend, implementado em Go, atua como um gateway entre os clientes (aplicações web ou mobile) e o servidor gRPC. Ele é responsável por receber requisições, aplicar a lógica de negócio e se comunicar com o serviço de persistência em Python por meio de stubs gRPC gerados a partir de arquivos .proto. Essa separação de camadas garante que o backend não precise conhecer detalhes da implementação do banco de dados, apenas se comunicar via contratos RPC bem definidos.

O servidor gRPC, escrito em Python, realiza todas as operações no banco PostgreSQL. Ele expõe endpoints como criação de usuários, envio de mensagens e recuperação de histórico de chats, seguindo um contrato estabelecido no arquivo .proto. Além disso, gerencia transações, concorrência e erros, garantindo a consistência dos dados.

O banco de dados PostgreSQL armazena as informações de forma relacional, com tabelas para usuários, chats e mensagens. Para otimizar consultas frequentes, como busca de mensagens ou validação de login, são utilizados índices. A arquitetura ainda permite escalabilidade horizontal, com possibilidade de adicionar réplicas de leitura conforme a demanda cresce.

O fluxo de comunicação segue uma sequência clara: o cliente envia uma requisição ao backend em Go, que a processa e aciona o servidor gRPC em Python. Este, por sua vez, executa a operação no banco de dados e retorna a resposta ao backend, que finalmente a repassa ao cliente. Essa estrutura traz vantagens como baixo acoplamento, alta performance e facilidade de manutenção, sendo ideal para aplicações que exigem comunicação rápida e gerenciamento eficiente de dados, como sistemas de chat em tempo real.

Conclusão

O desenvolvimento do projeto possibilitou a compreensão prática de conceitos essenciais relacionados à virtualização, redes virtuais, containers, arquitetura de microsserviços e comunicação eficiente entre aplicações distribuídas por meio do gRPC. A experiência abrangeu desde a configuração de ambientes virtualizados com QEMU/KVM e ferramentas correlatas, até a implementação de serviços com comunicação binária otimizada utilizando Protocol Buffers.

A aplicação dos diferentes tipos de chamadas gRPC — unária, server-streaming, client-streaming e bidirecional — evidenciou como cada abordagem pode ser adequadamente empregada em distintos cenários de sistemas distribuídos. Simultaneamente, a utilização do Docker e a integração com o banco de dados PostgreSQL proporcionaram uma visão abrangente do funcionamento de arquiteturas modernas, destacando a relevância da modularização, escalabilidade e eficiência na troca de informações.

Durante o desenvolvimento, foram enfrentados diversos desafios técnicos que demandaram pesquisa, adaptação e cooperação entre os integrantes do grupo. A configuração da rede virtual com múltiplas máquinas virtuais e a implementação de um sistema de mensagens com backend distribuído representaram marcos significativos na consolidação do conhecimento adquirido.

Conclusões Pessoais

Arthur Grandão de Mello: Apesar dos prazos apertados e da dificuldade técnica enfrentada, acredito que conseguimos fazer um projeto interessante. Minha participação se deu principalmente na parte da comunicação gRPC entre o servidor web e o banco de dados, além de contribuir pontualmente com as dificuldades dos companheiros.

Geovanna Maciel Avelino da Costa: considero o trabalho particularmente desafiador, pois exigiu o domínio de diversas ferramentas e conceitos técnicos em paralelo, como QEMU, libvirt, gRPC, Docker e PostgreSQL. Apesar das dificuldades, a experiência foi enriquecedora. Contribuí realizando a documentação completa do projeto, o que me ajudou a consolidar os conhecimentos adquiridos e entender a importância de registrar cada etapa do desenvolvimento de forma clara e estruturada.

Referências

GOOGLE. *gRPC: A high-performance, open-source universal RPC framework*. Disponível em: https://grpc.io. Acesso em: 30 maio 2025.

GOOGLE. *gRPC GitHub repository*. Disponível em: https://github.com/grpc/grpc. Acesso em: 30 maio 2025.

FLINT, J. QEMU: A Fast and Portable Dynamic Translator. 2005. Disponível em: https://www.qemu.org. Acesso em: 1 maio 2025.

KHALID, A.; SINGH, R. libvirt: Managing Virtualization Platforms. Journal of Cloud Computing, v. 7, p. 45–60, 2018.

SMITH, L.; JOHNSON, M. Graphical Virtualization with virt-manager. Virtualization Today, v. 12, n. 3, p. 22–30, 2020.

DOE, J.; RICHARDS, P. Automation of VM Lifecycles Using virsh. SysAdmin Magazine, v. 15, n. 2, p. 10–18, 2019.

POPEK, G. J.; GOLDBERG, R. H. Formal Requirements for Virtualizable Third Generation Architectures. Communications of the ACM, v. 17, n. 7, p. 412–421, 1974.

LEE, S.; KIM, H. Modular Virtualization Architectures. International Conference on Systems, 2021.