UNIVERSIDADE DE BRASÍLIA

Faculdade de Ciências e Engenharias em Tecnologia

FGA0244 - T02

PROGRAMAÇÃO PARA SISTEMAS PARALELOS E DISTRIBUÍDOS

Programação para Sistemas Paralelos e Distribuídos

Estudos extraclasse – gRPC

Valderson Pontes da Silva Junior – 190020521

Renan Rodrigues Lacerda – 190048191

João Victor Max Bisinotti de Oliveira – 170069991

Brasília, DF
2025

1. INTRODUÇÃO

Este relatório apresenta o desenvolvimento de uma aplicação distribuída composta por microserviços integrados por meio do framework gRPC, executando sobre uma infraestrutura virtualizada. A arquitetura da aplicação é dividida em três módulos: um gateway HTTP (módulo P), que atua como cliente gRPC, e dois servidores gRPC (módulos A e B), implementados em linguagens diferentes e hospedados em máquinas virtuais distintas.

O ambiente foi configurado utilizando o Virt-Manager como ferramenta principal de criação e gerenciamento das máquinas virtuais Ubuntu, com suporte das tecnologias QEMU, KVM e Libvirt. Cada máquina foi configurada com endereçamento de rede fixo e conectividade direta via macvlan.

O gRPC (Remote Procedure Call) é um framework moderno de comunicação entre serviços, baseado em HTTP/2 e Protobuf, que permite a troca eficiente de dados entre aplicações distribuídas. Seu suporte a diferentes padrões de chamada, como chamadas unárias e streaming bidirecional, oferece flexibilidade para diversos tipos de integração entre serviços. A combinação dessa tecnologia com a virtualização permitiu a simulação de um ambiente distribuído real, onde os serviços operam de forma isolada, mas cooperativa.

Além da aplicação, o relatório apresenta os passos necessários para a criação e configuração das máquinas virtuais, bem como o funcionamento da arquitetura distribuída proposta.

2.ESTUDO DO FRAMEWORK GRPC

gRPC (gRPC Remote Procedure Calls) é um framework open-source de alta performance desenvolvido inicialmente pelo Google. Ele permite que uma aplicação cliente chame métodos em uma aplicação servidora localizada em outra máquina (ou no mesmo processo) como se fosse um objeto local, facilitando a criação de sistemas distribuídos e microserviços. Sua eficiência, suporte a múltiplas linguagens e recursos avançados o tornam uma alternativa poderosa aos modelos tradicionais de comunicação, como REST sobre HTTP/1.1.

2.1. Elementos Constituintes: Protobuf e HTTP/2

O poder e a eficiência do gRPC derivam fundamentalmente de dois pilares tecnológicos: Protocol Buffers (Protobuf) e o protocolo HTTP/2.

a) Protocol Buffers (Protobuf):

Protocol Buffers é um mecanismo extensível, eficiente e neutro em termos de linguagem e plataforma, desenvolvido pelo Google, para serializar dados estruturados. Pense nele como um XML ou JSON, mas menor, mais rápido e mais simples. No contexto do gRPC, o Protobuf desempenha duas funções cruciais:

Interface Definition Language (IDL): Desenvolvedores definem a interface do serviço em um arquivo .proto. Este arquivo descreve os métodos que o serviço expõe (incluindo seus parâmetros e tipos de retorno) e as estruturas de dados (mensagens) que serão trocadas. Essa definição de contrato é clara, fortemente tipada e serve como única fonte da verdade para a comunicação entre cliente e servidor.

Formato de Serialização Binária: Uma vez definida a estrutura dos dados no arquivo .proto, o compilador do Protobuf (protoc) pode gerar código fonte (em diversas linguagens como Java, Python, Go, C++, C#, Node.js, etc.) para criar ou parsear um fluxo de bytes que representa as mensagens estruturadas. Essa serialização é binária, resultando em payloads muito menores e mais rápidos de serializar/desserializar em comparação com formatos baseados em texto como JSON ou XML.

b) HTTP/2:

gRPC foi projetado para operar sobre o protocolo HTTP/2, que oferece vantagens significativas sobre o HTTP/1.1, essenciais para a performance e as funcionalidades do gRPC:

Multiplexing: HTTP/2 permite que múltiplas requisições e respostas sejam enviadas e recebidas concorrentemente sobre uma única conexão TCP, sem bloqueio "head-of-line". Isso é fundamental para a eficiência do gRPC, especialmente para chamadas de longa duração ou streaming, pois evita a sobrecarga de estabelecer múltiplas conexões TCP.

Streaming Bidirecional: HTTP/2 suporta nativamente streams bidirecionais full-duplex, onde cliente e servidor podem enviar mensagens de forma independente e simultânea sobre a mesma conexão. gRPC utiliza essa capacidade para implementar seus modelos de comunicação de streaming (server-streaming, client-streaming e bidirectional-streaming).

Framing Binário: Diferente do HTTP/1.1 que é baseado em texto, HTTP/2 usa um protocolo de framing binário. Isso torna o parsing mais eficiente, menos propenso a erros e mais compacto na rede.

Compressão de Cabeçalhos (HPACK): HTTP/2 usa o algoritmo HPACK para comprimir cabeçalhos HTTP, reduzindo significativamente o overhead, especialmente em chamadas RPC frequentes com cabeçalhos repetitivos.

Ao combinar a definição de contrato clara e a serialização eficiente do Protobuf com as capacidades avançadas de transporte do HTTP/2, o gRPC oferece uma base robusta e performática para comunicação entre microserviços.

2.2. Tipos de Comunicação gRPC

gRPC define quatro tipos de métodos de serviço (ou padrões de comunicação), aproveitando as capacidades do HTTP/2:

- Unary RPC: É o modelo mais simples e tradicional de RPC. O cliente envia uma única requisição ao servidor e espera por uma única resposta, de forma similar a uma chamada de função normal.
    - Uso Típico: Consultas simples, comandos que executam uma ação e retornam sucesso/falha ou um resultado único. Ex: Obter detalhes de um usuário, criar um novo pedido.
- Server Streaming RPC: O cliente envia uma única requisição ao servidor, mas recebe de volta um stream de mensagens. O cliente lê do stream retornado até que não haja mais mensagens.
    - Uso Típico: Servidor enviando notificações, retornando grandes conjuntos de dados em partes (chunks), subscrição de eventos. Ex: Receber cotações de ações em tempo real, baixar um arquivo grande em partes.
- Client Streaming RPC: O cliente envia um stream de mensagens para o servidor. Uma vez que o cliente termina de enviar as mensagens, ele espera o servidor processá-las e retornar uma única resposta.
    - Uso Típico: Cliente enviando grandes volumes de dados para o servidor, agregação de dados. Ex: Fazer upload de um arquivo grande ou de dados de telemetria em partes.
- Bidirectional Streaming RPC: Ambos, cliente e servidor, enviam um stream de mensagens um para o outro usando uma única conexão HTTP/2. As duas streams operam independentemente, permitindo que cliente e servidor leiam e escrevam em qualquer ordem.

- Uso Típico: Comunicação interativa em tempo real, sessões de longa duração. Ex: Aplicações de chat, sessões de edição colaborativa, jogos multiplayer.

A flexibilidade oferecida por esses quatro tipos de comunicação permite que o gRPC seja adaptado a uma vasta gama de cenários de comunicação distribuída.

2.3. gRPC vs. Alternativas (REST, Web Services, etc.)

Ao construir aplicações distribuídas, gRPC não é a única opção. É útil compará-lo com alternativas comuns:

2.3.1. gRPC vs. REST (JSON/HTTP/1.1 ou HTTP/2):

- Contrato: gRPC usa Protobuf para um contrato forte e geração de código, enquanto REST geralmente depende de convenções e documentação externa (como OpenAPI/Swagger), sendo mais flexível mas potencialmente mais propenso a erros de integração.

- Payload: Protobuf (binário) é tipicamente menor e mais rápido de processar que JSON (texto).

- Protocolo: gRPC exige HTTP/2, aproveitando multiplexing e streaming nativo. REST pode usar HTTP/1.1 (menos eficiente) ou HTTP/2, mas o streaming não é tão padronizado quanto no gRPC.

- Casos de Uso: gRPC brilha na comunicação interna entre microserviços (performance, contrato forte, streaming). REST é excelente para APIs públicas (simplicidade, legibilidade humana do JSON, vasto suporte de ferramentas e browsers).

    - Browser Support: Chamar gRPC diretamente de um browser é complexo e geralmente requer um proxy (como gRPC-Web), enquanto REST/JSON é nativamente suportado.

2.3.2. gRPC vs. Web Services (SOAP/XML/HTTP):

- Performance: gRPC (Protobuf/HTTP/2) é significativamente mais performático que SOAP (XML/HTTP/1.1) devido à serialização binária e ao transporte mais eficiente.

- Complexidade: SOAP é frequentemente visto como mais complexo devido à verbosidade do XML e ao extenso conjunto de padrões WS-*. gRPC tende a ser mais simples de usar para casos comuns.

- Contrato: Ambos oferecem contratos fortes (WSDL para SOAP, Protobuf para gRPC), mas a abordagem do Protobuf é geralmente considerada mais moderna e leve.

- Popularidade: SOAP ainda é usado em sistemas legados e enterprise, mas gRPC e REST são mais populares para novos desenvolvimentos de microserviços.

2.3.3. Outras Formas (Message Queues - RabbitMQ, Kafka):

Estes não são substitutos diretos do RPC, mas sim padrões de comunicação assíncrona baseados em mensagens/eventos. Eles são frequentemente usados em conjunto com gRPC ou REST para desacoplar serviços, lidar com picos de carga ou implementar arquiteturas orientadas a eventos. A escolha depende do padrão de comunicação desejado (síncrono vs. assíncrono). Exemplos incluem RabbitMQ e Kafka.

Em suma, gRPC oferece uma solução moderna, performática e fortemente tipada para comunicação RPC, particularmente adequada para ambientes de microserviços internos onde a eficiência e contratos claros são prioritários, complementando ou substituindo outras abordagens como REST dependendo dos requisitos específicos do sistema.

3.DESENVOLVIMENTO DA APLICACÃO DISTRIBUÍDA

3.1. VISÃO GERAL DA APLICAÇÃO

A aplicação distribuída implementada tem como finalidade executar operações matemáticas simples e avançadas por meio de chamadas gRPC entre módulos interligados. A estrutura é composta por três módulos principais:

- Módulo P: Web server com API HTTP (interface para o usuário), implementado em Go, atuando como cliente gRPC.
- Módulo A: Servidor gRPC em Node.js responsável por operações como soma, subtração, multiplicação e divisão.
- Módulo B: Servidor gRPC em Python responsável por operações como cálculo do quadrado, raiz quadrada, potência e fatorial.

Os módulos comunicam-se exclusivamente por gRPC, respeitando uma arquitetura distribuída, onde cada serviço está isolado em uma máquina virtual distinta.

3.2. ARQUITETURA

A arquitetura da aplicação distribuída foi planejada com base em três máquinas virtuais isoladas, cada uma executando um módulo específico do sistema: P, A e B. Esses módulos se comunicam entre si exclusivamente por meio do protocolo gRPC, possibilitando a separação de responsabilidades, facilidade de manutenção e testes independentes.

A infraestrutura foi criada utilizando o Virt-Manager para virtualização e o QEMU-KVM como hypervisor. No host (Linux Mint), foi configurada uma interface virtual macvlan0 para permitir a comunicação entre as VMs e o sistema hospedeiro, utilizando uma ponte baseada na interface Wi-Fi (wlp0s20f3). Cada máquina recebeu um IP fixo dentro da mesma sub-rede para facilitar a comunicação direta:

- Host: 192.168.1.100
- VM1 (módulo P): 192.168.1.201
- VM2 (módulo A): 192.168.1.202
- VM3 (módulo B): 192.168.1.203

As VMs também foram configuradas para acessar a internet por meio de uma interface NAT secundária (enp2s0). A configuração de rede manual, incluindo o uso da interface macvtap no Virt-Manager, foi essencial para garantir a comunicação direta entre os módulos.

A seguir, são descritos os papéis e tecnologias utilizadas em cada um dos módulos:

- Módulo P (VM1 - Go): Implementado em Go, este módulo atua como um gateway HTTP. Ele escuta requisições externas na porta 8080 e repassa chamadas via gRPC para os módulos A e B conforme o tipo de operação solicitada (/sum ou /square).
- Módulo A (VM2 - Node.js): Este módulo expõe um servidor gRPC na porta 50051. Ele realiza operações matemáticas básicas como soma,

subtração, multiplicação e divisão, utilizando a biblioteca @grpc/grpc-js.

- Módulo B (VM3 - Python): Implementado em Python, o módulo B oferece um servidor gRPC que realiza operações como quadrado, raiz quadrada, potência e fatorial. Ele escuta na porta 50052 e utiliza grpcio e grpcio-tools.

3.3. IMPLEMENTAÇÃO

A implementação da aplicação distribuída foi dividida entre três VMs, cada uma com um papel específico. A comunicação entre os módulos foi realizada por gRPC, com bibliotecas específicas para cada linguagem.

Módulo A – gRPC Server em Node.js (VM2)

- Bibliotecas utilizadas:
  - @grpc/grpc-js
  - @grpc/proto-loader
- Instalação:
  - sudo apt update
  - sudo apt install nodejs npm -y
  - npm install @grpc/grpc-js @grpc/proto-loader
- Como rodar
  - Node server.js
- Definição do serviço (calculator.proto):

```
syntax = "proto3";

option go_package = "./sum";

service Calculator {
  rpc Sum (SumRequest) returns (SumResponse);
  rpc Subtract (SumRequest) returns (SumResponse);
  rpc Multiply (SumRequest) returns (SumResponse);
  rpc Divide (SumRequest) returns (DivisionResponse);
}

message SumRequest {
  int32 num1 = 1;
  int32 num2 = 2;
}

message SumResponse {
  int32 result = 1;
}

message DivisionResponse {
  double result = 1;
}
```

- Criar o servidor server.js

JavaScript

```javascript
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');

const PROTO_PATH = './calculator.proto';

const packageDefinition = protoLoader.loadSync(PROTO_PATH);
const calculatorProto = grpc.loadPackageDefinition(packageDefinition).Calculator;

function sum(call, callback) {
  const { num1, num2 } = call.request;
  const result = num1 + num2;
  console.log(`Sum: ${num1} + ${num2} = ${result}`);
  callback(null, { result });
}

function subtract(call, callback) {
  const { num1, num2 } = call.request;
  const result = num1 - num2;
  console.log(`Subtract: ${num1} - ${num2} = ${result}`);
  callback(null, { result });
}

function multiply(call, callback) {
  const { num1, num2 } = call.request;
  const result = num1 * num2;
  console.log(`Multiply: ${num1} * ${num2} = ${result}`);
  callback(null, { result });
}
```

```
function divide(call, callback) {
  const { num1, num2 } = call.request;
  if (num2 === 0) {
    return callback({
      code: grpc.status.INVALID_ARGUMENT,
      message: 'Division by zero is not allowed',
    });
  }
  const result = num1 / num2;
  console.log(`Divide: ${num1} / ${num2} = ${result}`);
  callback(null, { result });
}

const server = new grpc.Server();
server.addService(calculatorProto.service, {
  Sum: sum,
  Subtract: subtract,
  Multiply: multiply,
  Divide: divide,
});

const address = '0.0.0.0:50051';
server.bindAsync(address, grpc.ServerCredentials.createInsecure(), () => {
  console.log(`gRPC Server running at ${address}`);
  server.start();
});
```

- Comando para a geração dos stubs grpc_tools_node_protoc \ --js_out=import_style=commonjs:. \ --grpc_out=grpc_js:. \ --proto_path=. \ calculator.proto

Módulo B – gRPC Server em Python (VM3)

- Bibliotecas utilizadas
  - grpcio
  - grpcio-tools
- Instalação
  - sudo apt update
  - sudo apt install python3-pip –y
  - pip3 install grpcio grpcio-tools
- Como rodar
  - python3 server.py
  - Utilizar o ambiente virtual gerado na pasta anterior a do servidor
  - source ../env/bin/activate
- Definição do serviço

```
syntax = "proto3";

option go_package = "./square";

service SquareService {
  rpc Square (SquareRequest) returns (SquareResponse);
  rpc Sqrt (SqrtRequest) returns (SqrtResponse);
  rpc Power (PowerRequest) returns (PowerResponse);
  rpc Factorial (FactorialRequest) returns (FactorialResponse);
}

message SquareRequest {
  int32 number = 1;
}
message SquareResponse {
  int32 result = 1;
}

message SqrtRequest {
  double number = 1;
}
message SqrtResponse {
  double result = 1;
}

message PowerRequest {
  int32 base = 1;
  int32 exponent = 2;
}
message PowerResponse {
  int64 result = 1;
}

message FactorialRequest {
  int32 number = 1;
}
message FactorialResponse {
  int64 result = 1;
}
```

- Criar o servidor

```python
from concurrent import futures
import grpc
import math
import square_pb2
import square_pb2_grpc

class SquareService(square_pb2_grpc.SquareServiceServicer):
    def Square(self, request, context):
        result = request.number * request.number
        return square_pb2.SquareResponse(result=result)

    def Sqrt(self, request, context):
        number = request.number
        if number < 0:
            context.set_code(grpc.StatusCode.INVALID_ARGUMENT)
            context.set_details("Negative number cannot have a real square
root.")
            return square_pb2.SqrtResponse()
        result = math.sqrt(number)
        return square_pb2.SqrtResponse(result=result)

    def Power(self, request, context):
        result = int(math.pow(request.base, request.exponent))
        return square_pb2.PowerResponse(result=result)

    def Factorial(self, request, context):
        n = request.number
        if n < 0:
            context.set_code(grpc.StatusCode.INVALID_ARGUMENT)
            context.set_details("Cannot calculate factorial of a negative
number.")
            return square_pb2.FactorialResponse()
        result = math.factorial(n)
        return square_pb2.FactorialResponse(result=result)

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    square_pb2_grpc.add_SquareServiceServicer_to_server(SquareService(), server)
    server.add_insecure_port('[::]:50052')
    server.start()
    print("gRPC Server B running at 0.0.0.0:50052")
    server.wait_for_termination()

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

- Comando para a geração dos stubs
    - python3 -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. Square.proto

Módulo P – Cliente gRPC + HTTP Server em Go (VM1)

- Bibliotecas utilizadas:
    - google.golang.org/grpc
    - net/http

- Instalação
    - sudo apt update
    - sudo apt install golang-go –y
    - go install google.golang.org/protobuf/cmd/protoc-gen-go@latest
    - go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest
- Como rodar
    - go run main.go
- Definição dos serviços
    - Os arquivos .proto são os mesmos dos criados para os microsserviços
- Comando para a geração dos stubs
    - protoc --go_out=. --go-grpc_out=. sum.proto
    - protoc --go_out=. --go-grpc_out=. Square.proto
- Definição do webservice

```go
package main

import (
    "context"
    "fmt"
    "log"
    "net/http"
    "strconv"
    "time"

    pbSquare "grpc-client-p/square"
    pbCalc "grpc-client-p/sum"

    "google.golang.org/grpc"
)

func sumHandler(w http.ResponseWriter, r *http.Request) {
    aStr := r.URL.Query().Get("a")
    bStr := r.URL.Query().Get("b")

    a, err1 := strconv.Atoi(aStr)
    b, err2 := strconv.Atoi(bStr)

    if err1 != nil || err2 != nil {
        http.Error(w, "Missing or invalid query parameters: ?a=5&b=3",
http.StatusBadRequest)
        return
    }

    conn, err := grpc.Dial("192.168.1.202:50051", grpc.WithInsecure())
    if err != nil {
        http.Error(w, "Could not connect to gRPC Server A (Calculator)",
http.StatusInternalServerError)
        return
    }
    defer conn.Close()

    client := pbCalc.NewCalculatorClient(conn)
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()

    resp, err := client.Sum(ctx, &pbCalc.SumRequest{Num1: int32(a), Num2:
int32(b)})
    if err != nil {
        http.Error(w, "Error calling gRPC Sum", http.StatusInternalServerError)
        return
    }

    fmt.Fprintf(w, "Soma de %d + %d = %d\n", a, b, resp.Result)
}
```

```go
func squareHandler(w http.ResponseWriter, r *http.Request) {
    numStr := r.URL.Query().Get("number")
    num, err := strconv.Atoi(numStr)
    if err != nil {
        http.Error(w, "Missing or invalid query parameter: ?number=7",
http.StatusBadRequest)
        return
    }

    conn, err := grpc.Dial("192.168.1.203:50052", grpc.WithInsecure())
    if err != nil {
        http.Error(w, "Could not connect to gRPC Server B (SquareService)",
http.StatusInternalServerError)
        return
    }
    defer conn.Close()

    client := pbSquare.NewSquareServiceClient(conn)
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()

    resp, err := client.Square(ctx, &pbSquare.SquareRequest{Number: int32(num)})
    if err != nil {
        http.Error(w, "Error calling gRPC Square",
http.StatusInternalServerError)
        return
    }

    fmt.Fprintf(w, "Quadrado de %d = %d\n", num, resp.Result)
}

func main() {
    http.HandleFunc("/sum", sumHandler)
    http.HandleFunc("/square", squareHandler)

    fmt.Println("HTTP Server running on :8080")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

3.4. TESTE DOS ENDPOINTS

Para executar a aplicação distribuída como um todo, é necessário iniciar todos os serviços em suas respectivas máquinas virtuais. Após isso, acesse a VM1, onde há uma pasta chamada frontend localizada em /home. Dentro dessa pasta, execute o seguinte comando para iniciar um servidor HTTP simples:

python2 -m http.server 3000

Em seguida, abra o navegador na VM1 e acesse http://localhost:3000. A interface disponibilizada permite testar os endpoints expostos pelo WebServer Go. Esses endpoints realizam chamadas gRPC para os microsserviços hospedados na VM2 (Node.js) e na VM3 (Python), possibilitando validar toda a comunicação da aplicação distribuída.

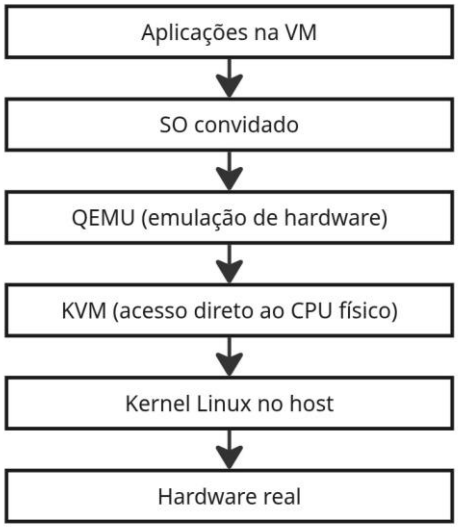
4. ESTUDO E USO DA VIRTUALIZAÇÃO

4.1 Arquitetura interna do kvm/qemu.

O KVM (Kernel-based Virtual Machine) e o QEMU operam juntos para fornecer uma solução de virtualização eficiente em sistemas Linux. O KVM é um módulo do kernel que transforma o Linux em um hipervisor tipo 1, permitindo que o sistema operacional hospede múltiplas máquinas virtuais (VMs) com acesso direto ao hardware, usando extensões como Intel VT-x ou AMD-V. O QEMU, por sua vez, é um emulador de hardware que provê todo o ambiente virtual da VM (CPU, memória, dispositivos de entrada/saída, rede, etc.). Quando combinado com o KVM, ele consegue virtualizar sistemas operacionais de forma muito mais rápida, pois delega a execução das instruções diretamente para o hardware físico via KVM.

O QEMU, por sua vez, é um emulador de hardware que provê todo o ambiente virtual da VM (CPU, memória, dispositivos de entrada/saída, rede, etc.). Quando combinado com o KVM, ele consegue virtualizar sistemas operacionais de forma muito mais rápida, pois delega a execução das instruções diretamente para o hardware físico via KVM.

A arquitetura pode ser representada dessa forma:



4.2 Estudo e testes de recurso de firmware em máquinas virtuais (firmware coreboot e BIOS SeaBIOS).

Durante a inicialização de uma máquina virtual, o firmware é o primeiro componente executado, responsável por preparar o ambiente para carregar o sistema operacional.

4.2.1 SeaBIOS

SeaBIOS é um firmware compatível com a especificação tradicional da BIOS. Ele é o firmware padrão utilizado em VMs do QEMU/KVM, e foi usado em todas as VMs criadas neste experimento. SeaBIOS permite boot com discos virtuais, suporte a teclados, rede PXE e dispositivos IDE/SATA emulando um comportamento semelhante ao de BIOSs convencionais.

4.2.2 Coreboot

Coreboot é um firmware de código aberto que visa substituir firmwares proprietários, oferecendo inicialização extremamente rápida e modular. Foi realizado um estudo teórico e um teste de VM utilizando uma imagem compatível com coreboot como firmware, através da flag -bios no QEMU.

4.3 Comandos Utilizados

virsh list --all (Listar VMs)

virsh start vm1 (Iniciar VM)

virsh shutdown vm2 (Desligar VM)

virsh destroy vm3 (Desligar à força)

virsh console vm1 (Abrir terminal da VM)

4.3 Dificuldades Encontradas

- Configuração de rede virtual: foi necessário configurar corretamente as interfaces e as conexões LAN interna entre as VMs e roteamento para o host (gateway/API).
- Exportação/importação de VMs.

4.4 Resultados Alcançados

- As VMs foram criadas com sucesso utilizando virt-manager e virsh, sendo exportadas e testadas em outra máquina com sucesso.

- A estrutura final implementada seguiu o projeto proposto com três VMs (gateway + dois servidores gRPC) conectadas por redes virtuais distintas e integradas com sucesso.

5.CONCLUSÃO

A construção desta aplicação distribuída permitiu colocar em prática diversos conceitos fundamentais de sistemas paralelos e distribuídos, com foco no uso do gRPC como mecanismo de comunicação entre microserviços. A escolha por utilizar três máquinas virtuais independentes, cada uma com um módulo específico e bem definido, proporcionou uma simulação realista de ambientes distribuídos utilizados em contextos profissionais.

Durante o desenvolvimento, foi possível explorar os principais recursos do gRPC, como a definição de contratos com arquivos .proto, a comunicação eficiente baseada em HTTP/2 e a compatibilidade com múltiplas linguagens. A interoperabilidade entre serviços escritos em Node.js, Python e Go demonstrou, na prática, a flexibilidade da tecnologia para integrações heterogêneas.

A configuração manual das redes virtuais, utilizando Virt-Manager e interfaces macvlan, exigiu atenção especial, mas foi fundamental para garantir a comunicação entre os serviços e o host. Essa etapa contribuiu significativamente para o entendimento prático sobre a criação de redes isoladas e roteamento entre máquinas virtuais.

Em resumo, mesmo com algumas dificuldades, o projeto alcançou os objetivos propostos, integrando de forma prática os conhecimentos sobre virtualização, redes, comunicação gRPC e organização de aplicações distribuídas. A experiência contribuiu para o amadurecimento técnico e aprofundamento nos temas abordados na disciplina.