

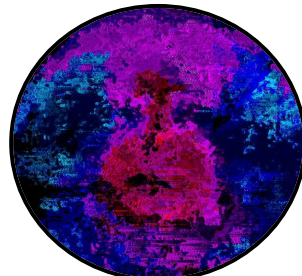
Grupo ProjectX - Gerenciador de Desempenho Distribuído



Bruno Lopes Santos - 201907448



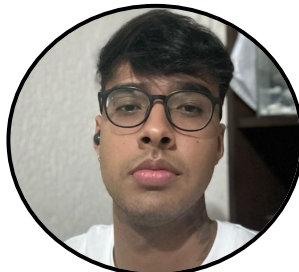
Eduardo Santos Santana - 202203503



Gabriel Silva Miranda - 201907464



Reydner Miranda Nunes - 201907501



Yatherson Lucas Teodoro Souza - 201912485



INSTITUTO DE
INFORMÁTICA
UFG



UFG
UNIVERSIDADE
FEDERAL DE GOIÁS

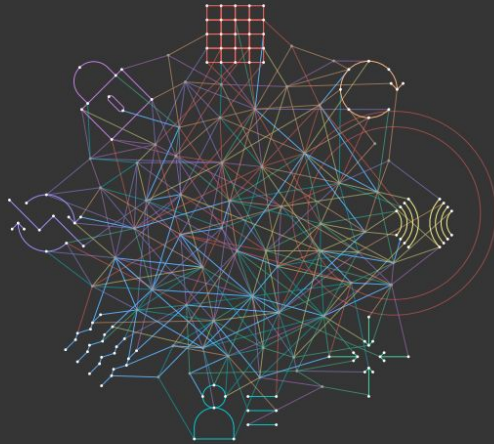


Objetivo

”

" Desenvolver um **Gerenciador de Desempenho Distribuído** capaz de coletar, agregar e analisar dados de desempenho de diversos computadores. O sistema permitirá o **monitoramento** em tempo real de **métricas** cruciais (CPU, RAM, rede, temperatura). Através de uma **interface** intuitiva, os usuários poderão visualizar gráficos e relatórios detalhados, identificando gargalos, otimizando recursos e garantindo a alta disponibilidade dos sistemas."

DISTRIBUTED SYSTEMS



MAARTEN VAN STEEN
ANDREW S. TANENBAUM

4TH EDITION

VERSION 01

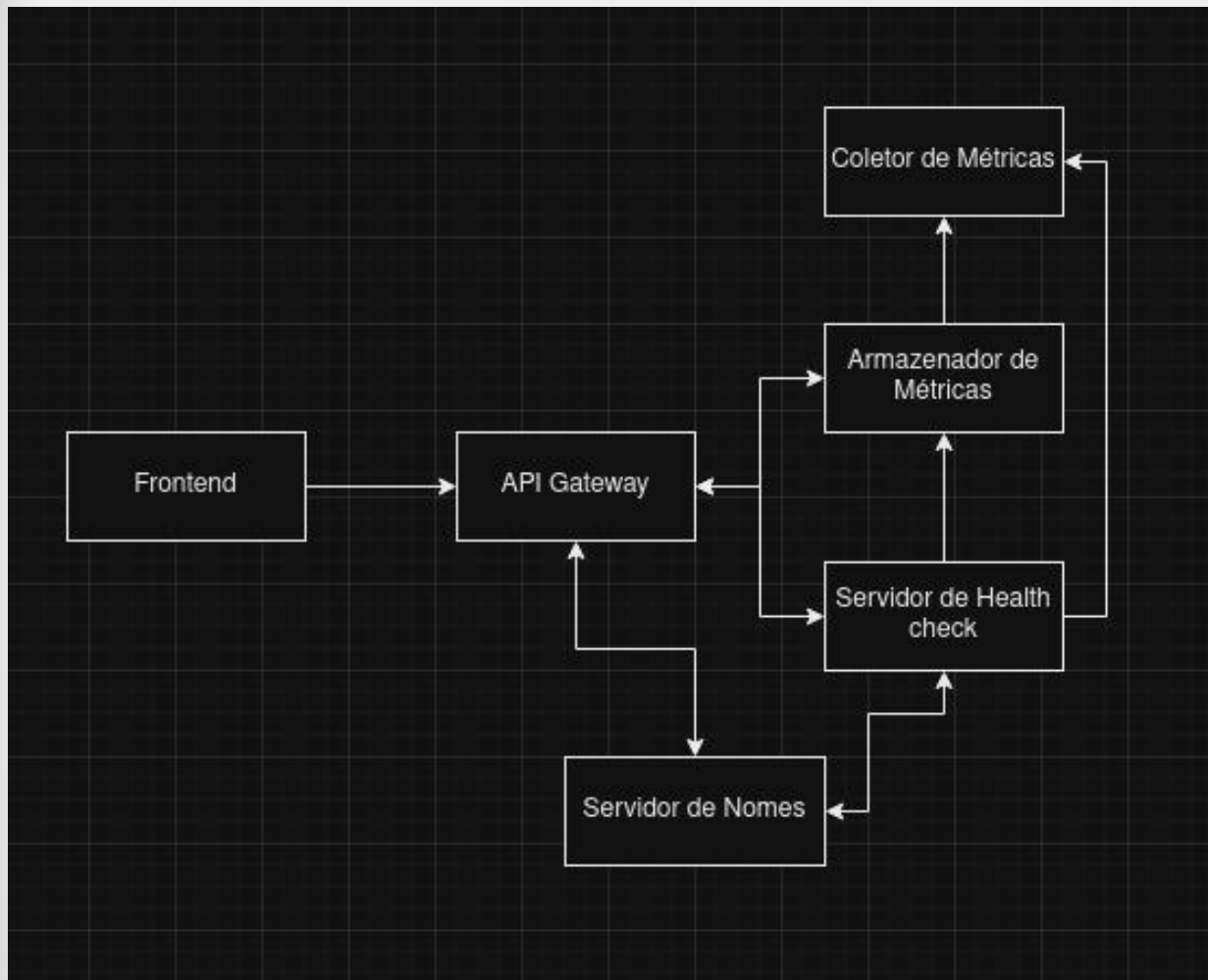
Fonte de estudo utilizada

Sistemas Distribuídos é uma obra fundamental que explora os conceitos e desafios inerentes à construção e ao gerenciamento de sistemas distribuídos. O livro busca fornecer aos seus leitores uma compreensão profunda dos princípios que regem todo contexto de sistemas distribuídos compartilhando conhecimento de Maarten van steen e Andrew S. Tanenbaum.

Indicação do Prof. Sergio

*“Sistema Distribuído: Um sistema de computador em rede onde processos e recursos estão **suficientemente** distribuídos em múltiplos computadores.”*

Arquitetura



Armazenador de métricas



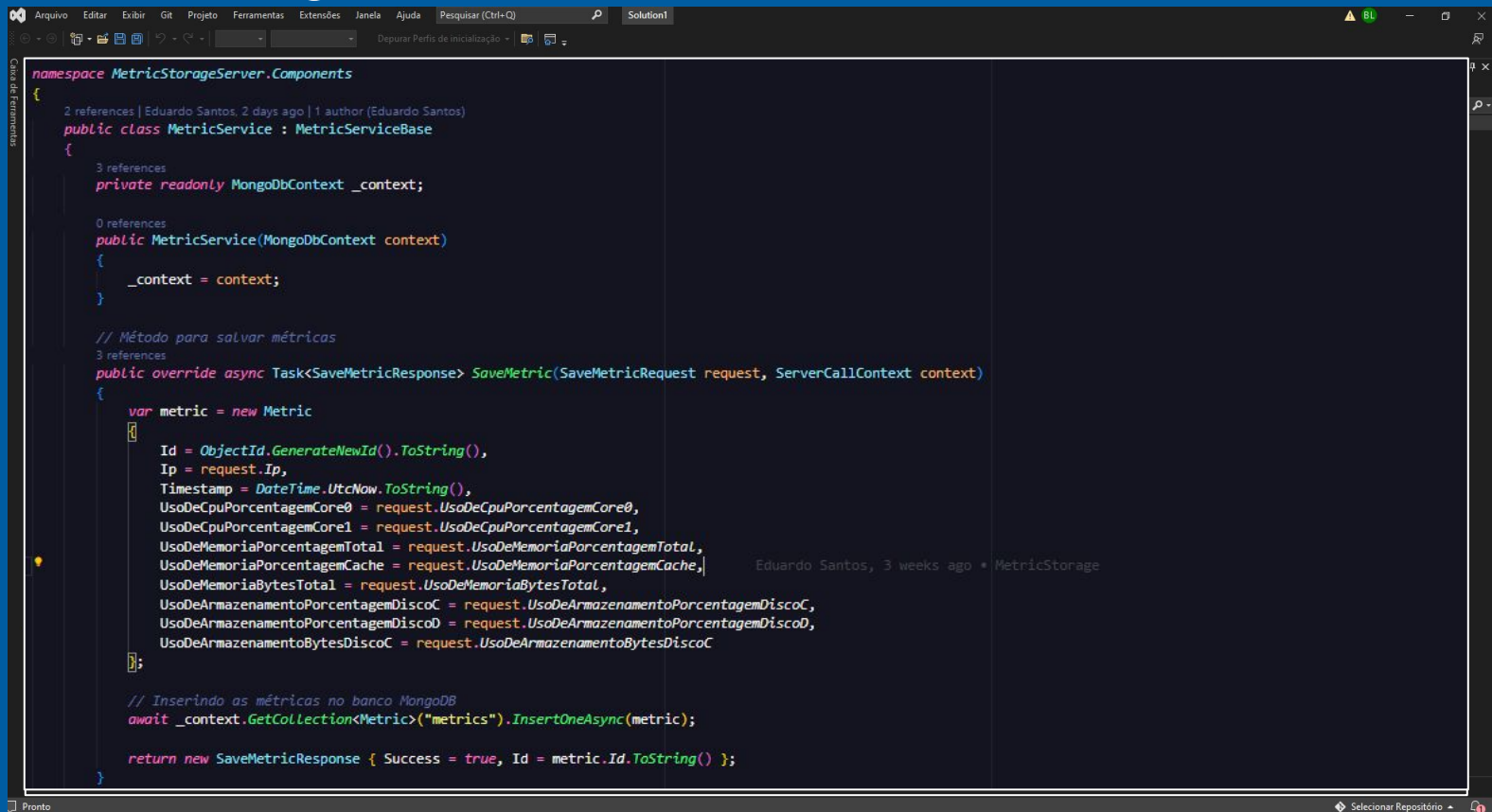
Conceito Literário: Algumas técnicas/conceitos estudadas para tentar aplicar no projeto:

- Sharding: A divisão dos dados em partes menores permite escalar o armazenamento de métricas horizontalmente.
- Protocolos Leves: Protocolos como HTTP facilitam a comunicação entre os agentes de coleta e o armazenador de métricas.
- Multithreading: O processamento concorrente de métricas através de múltiplas threads aumenta a eficiência do armazenador.

Descrição do Componente: Servidor de armazenamento de métricas que utiliza MongoDB e gRPC para receber e fornecer métricas sobre o uso de recursos como CPU, memória e armazenamento.

Linguagem: MongoDB e gRPC em .NET

Print do código



```
namespace MetricStorageServer.Components
{
    2 references | Eduardo Santos, 2 days ago | 1 author (Eduardo Santos)
    public class MetricService : MetricServiceBase
    {
        3 references
        private readonly MongoDBContext _context;

        0 references
        public MetricService(MongoDbContext context)
        {
            _context = context;
        }

        // Método para salvar métricas
        3 references
        public override async Task<SaveMetricResponse> SaveMetric(SaveMetricRequest request, ServerCallContext context)
        {
            var metric = new Metric
            {
                Id = ObjectId.GenerateNewId().ToString(),
                Ip = request.Ip,
                Timestamp = DateTime.UtcNow.ToString(),
                UsoDeCpuPorcentagemCore0 = request.UsoDeCpuPorcentagemCore0,
                UsoDeCpuPorcentagemCore1 = request.UsoDeCpuPorcentagemCore1,
                UsoDeMemoriaPorcentagemTotal = request.UsoDeMemoriaPorcentagemTotal,
                UsoDeMemoriaPorcentagemCache = request.UsoDeMemoriaPorcentagemCache,
                UsoDeMemoriaBytesTotal = request.UsoDeMemoriaBytesTotal,
                UsoDeArmazenamentoPorcentagemDiscoC = request.UsoDeArmazenamentoPorcentagemDiscoC,
                UsoDeArmazenamentoPorcentagemDiscoD = request.UsoDeArmazenamentoPorcentagemDiscoD,
                UsoDeArmazenamentoBytesDiscoC = request.UsoDeArmazenamentoBytesDiscoC
            };

            // Inserindo as métricas no banco MongoDB
            await _context.GetCollection<Metric>("metrics").InsertOneAsync(metric);

            return new SaveMetricResponse { Success = true, Id = metric.Id.ToString() };
        }
    }
}
```

[Link Github Armazenador de Métricas](#)

Coletor de métricas



Conceito Literário:

- Centralização/Descentralização: A decisão sobre onde armazenar as métricas influencia a arquitetura do sistema.
- Comunicação: Protocolos leves como HTTP ou gRPC facilitam a comunicação entre agentes e armazenador.
- Mensageria: Sistemas de mensageria como Kafka ou RabbitMQ permitem o envio de métricas em tempo real.

Descrição do Componente: Instalado nas máquinas monitoradas, coleta e expõe métricas do sistema (uso de memória, CPU, etc.), atualizando-as periodicamente.

Linguagem: Python 3.10

Funcionalidades/Técnicas: A aplicação funciona de forma assíncrona, servindo um servidor TCP na porta 9090 e coletando métricas de sistema de tempos em tempos. As métricas são inseridas no arquivo metrics.html, que é lido e enviado a qualquer cliente que se conecta à porta exposta.

- uso_de_cpu_porcentagem(total): porcentagem de uso de CPU total (todos os núcleos)
- uso_de_memoria_porcentagem(total): porcentagem de uso de memória RAM total

Print do código

```
Arquivo  Editar  Exibir  Git  Projeto  Ferramentas  Extensões  Janela  Ajuda  Pesquisar (Ctrl+Q)  Solution1
Depurar Perfil de inicialização

Casa de Ferramentas

async def send_metrics_to_grpc(metrics):
    # Carregar o certificado do servidor
    with open(CERTIFICATE_PATH, "rb") as cert_file:
        certificate = cert_file.read()

    # Criar credenciais SSL
    credentials = grpc.ssl_channel_credentials(certificate)

    # Conectar ao servidor gRPC usando HTTPS
    channel = grpc.aio.secure_channel("localhost:5001", credentials)
    stub = metric_service_pb2_grpc.MetricServiceStub(channel)

    # Preparar os dados da métrica
    request = metric_service_pb2.SaveMetricRequest(
        ip=metrics["Ip"],
        uso_de_cpu_porcentagem_core0=metrics["UsoDeCpuPorcentagemCore0"],
        uso_de_cpu_porcentagem_core1=metrics["UsoDeCpuPorcentagemCore1"],
        uso_de_memoria_porcentagem_total=metrics["UsoDeMemoriaPorcentagemTotal"],
        uso_de_memoria_porcentagem_cache=metrics["UsoDeMemoriaPorcentagemCache"],
        uso_de_memoria_bytes_total=metrics["UsoDeMemoriaBytesTotal"],
        uso_de_armazenamento_porcentagem_discoC=metrics["UsoDeArmazenamentoPorcentagemDiscoC"],
        uso_de_armazenamento_porcentagem_discoD=metrics["UsoDeArmazenamentoPorcentagemDiscoD"],
        uso_de_armazenamento_bytes_discoC=metrics["UsoDeArmazenamentoBytesDiscoC"],
    )

    # Enviar a métrica para o servidor gRPC
    response = await stub.SaveMetric(request)
    print("Métrica enviada com sucesso. ID:", response.id)

    await channel.close()
```

[Link Github Coletor de Métricas](#)

Servidor Health Check



Conceito Literário: Algumas técnicas/conceitos estudadas para tentar aplicar no projeto:

- Concorrência: Multithreading no servidor health check permite o processamento concorrente das informações.
- Gerenciamento de Falhas: Replicação e failover garantem a continuidade do serviço em caso de falhas.
- RMI: uso de invocação de método remoto para acessar informações do nameserver.

Descrição do Componente: Verifica a conectividade dos serviços do backend e informa o servidor de nomes quando um IP registrado não está funcionando para manter a lista atualizada.

Linguagem: Lua

Funcionalidades/Técnicas:

- **RPC/RMI com o servidor de nomes para acessar os dados**
- **Checagem de saúde concorrente de todos os IPs de um serviço**

Status: Não finalizado

Print do código - main loop

```
--  
-- HEALTH CHECKER SERVICE  
--  
local checker = require("checker")  
local names = require("names")  
local utils = require("utils")  
local tango = require("tango")  
  
local nameServerIp = os.getenv("NAME_SERVER_IP") or "127.0.0.1"  
local nameServerPort = tonumber(os.getenv("NAME_SERVER_PORT")) or 3001  
local delay = os.getenv("DELAY") or "5"  
  
local name_list = names.new_names()  
local check = checker.new_checker()  
local util = utils.new_utils()  
  
while true do  
    print("Criando RPC")  
    local connection = require("tango.client.socket").connect({address=nameServerIp, port=nameServerPort})  
    local namecache = tango.ref(connection.names)  
    print("Populando lista de hosts")  
    name_list:getNamesFromNameServer(namecache,util)  
    name_list:getIpsForEntries(namecache,util)  
    local entries = {}  
    print(name_list:getEntry({"name1"}))  
    for i, entry in ipairs(name_list.entries) do  
        print(i)  
        print(entry[i])  
        entries[i] = entry  
    end  
    print("Preparando checagem de hosts")  
    check:prepareCheck(entries)  
  
    print("Rodando checagem de hosts")  
    check:runCheck()  
  
    print("Checagem finalizada, esperando delay: "..delay.." segundos")  
    os.execute("sleep "..delay)  
end
```

[Link Github Health Check](#)

Print do código - RMI

```
-- Utilitários
--

return {
  new_utils = function()
    local Utils = {}
    local tango = require("tango")

    function Utils:getIpList(namecache,entry)
      local _, ip_list = namecache:get_all_entries(entry[1])
      if ip_list == 404 then
        return 404, "No names in cache"
      end
      return ip_list
    end

    function Utils:getNameList(namecache)
      local _, name_list = namecache:get_all_names()
      local entry_list = {}
      if name_list == 404 then
        return 404, "No names in cache"
      end
      for _, name in pairs(name_list) do
        table.insert(entry_list,{name,ips={}})
      end
      return entry_list
    end

    return Utils
  end
}
```

Print do código - função de checagem de saúde

```
function checker:checkHost(host, port)
  local client_get = assert(socket.udp4())
  local data = ""
  local status
  local s
  print("Enviando request GET para "..host..":"..port)
  local _, message = client_get:sendto("GET /health", host, port)
  if message ~= nil then print("Erro recebido ao enviar request: "..message) end
  while true do
    s, status = checker:Receive(client_get)
    if status ~= nil then
      print("status: "..status)
    end
    if s ~= nil then
      data = data..s
    end
    if string.find(data, "\n") then
      break
    end
  end
  client_get:close()
end
```

Servidor de nomes



Conceito Literário: Algumas técnicas/conceitos estudadas para tentar aplicar no projeto:

- Distribuição: A hierarquia DNS distribui a responsabilidade pela resolução de nomes entre diversos servidores.
- Replicação: A replicação de zonas DNS em múltiplos servidores aumenta a redundância e a disponibilidade.
- Comunicação: O protocolo DNS define a comunicação entre servidores e clientes para resolução de nomes.

Descrição do Componente: Mantém listas de nomes e IPs dos serviços, consultado por outros serviços como fonte de verdade para IPs, como um servidor DNS, porém simplificado;

Linguagem: Lua;

Funcionalidades/Técnicas:

- Adicionar um registro Para adicionar um registro, é necessário enviar, via UDP
- Recuperar um registro Para recuperar um registro, é necessário enviar, via UDP
- Remover um registro Para remover um registro, é necessário enviar, via UDP
- Sincronização de réplicas O Nameserver faz uma sincronização com suas réplicas ao receber um novo registro ou ter um registro deletado.

Prints do código - main loop

Code

Blame

25 lines (21 loc) · 687 Bytes

```
1  --
2  -- NAMESERVER SERVICE
3  --
4  local socket = require("socket")
5  local namecache = require("namecache")
6  local serv_hand = require("server_handler")
7
8  local host = ""
9  local port = 3000
10
11 local server = assert(socket.udp())
12 assert(server:setsockname(host,port))
13
14 local names = namecache.new_namecache()
15 local handler = serv_hand.new_server_handler()
16
17 assert(server:settimeout(0.01))
18 print("Start listening to requests:")
19 while true do
20     local data, client_ip, client_port = server:receivefrom(1024);
21     if data then
22         print("Received request from "..client_ip..":"..client_port.." with data: "..data)
23         handler:handle_request(data,client_ip,client_port,server,names)
24     end
25 end
```

[Link Github Servidor de Nomes](#)

Prints do código - rotina de testes

Code

Blame

22 lines (17 loc) · 362 Bytes

```
1  #!/usr/bin/lua
2
3  local testing = require("testing")
4
5  local ip = "127.0.0.1"
6  local port = 3000
7  local ip_quant = 50
8  local name_quant = 20
9  local names = {}
10 local ips = {}
11 for i=1,ip_quant do
12     ips[i] = "192.168.0..."i
13 end
14
15 for i=1,name_quant do
16     names[i] = "teste"..i
17 end
18
19 local test = testing.new_testing()
20 test:create_test(ips,ip,port,names)
21
22 test:run_test()
```

[Link Github Servidor de Nomes](#)

Prints do código - servidor com RPC (extra)

```
--  
-- NAMESERVER SERVICE  
--  
local socket = require("socket")  
local namecache = require("namecache")  
local serv_hand = require("server_handler")  
local copas = require("copas")  
  
names = namecache.new_namecache()      ■ Global variable in lowercase initial, Did you miss `local`  
local handler = serv_hand.new_server_handler()  
  
function receive_request(skt)          ■ Global variable in lowercase initial, Did you miss `local`  
    skt = copas.wrap(skt)  
    while true do  
        local data, err, port = skt:receivefrom(2048)  
        if data then  
            print("Received request from "..err..":"..port.." with data: "..data)  
            handler:handle_request(data,err,port,skt,names)  
        end  
    end  
end  
end  
  
local host = "*"   
local port = os.getenv("PORT")  
local rpc_port = os.getenv("RPC_PORT")  
local tango_socket, tango_request = require('tango.server.copas_socket').new{port=rpc_port}  
  
local server = assert(socket.udp())  
assert(server:setsockname(host,port))    ■ Undefined field `setsockname`.  
  
copas.addserver(server, receive_request, 1)  
copas.addserver(tango_socket, tango_request)  
  
print("Start listening to requests:")  
copas()
```


API Gateway



Conceito Literário: Algumas técnicas/conceitos estudadas para tentar aplicar no projeto:

- RPC: Para invocar serviços nos microsserviços.
- Mensageria: Para comunicação assíncrona entre a API Gateway e outros sistemas.
- Circuit Breaker: Para proteger a API Gateway de falhas em serviços downstream.
- Sharding: Para escalar a API Gateway horizontalmente, distribuindo a carga entre múltiplos nós.
- Autenticação e Autorização: Para controlar o acesso à API Gateway e aos serviços protegidos.

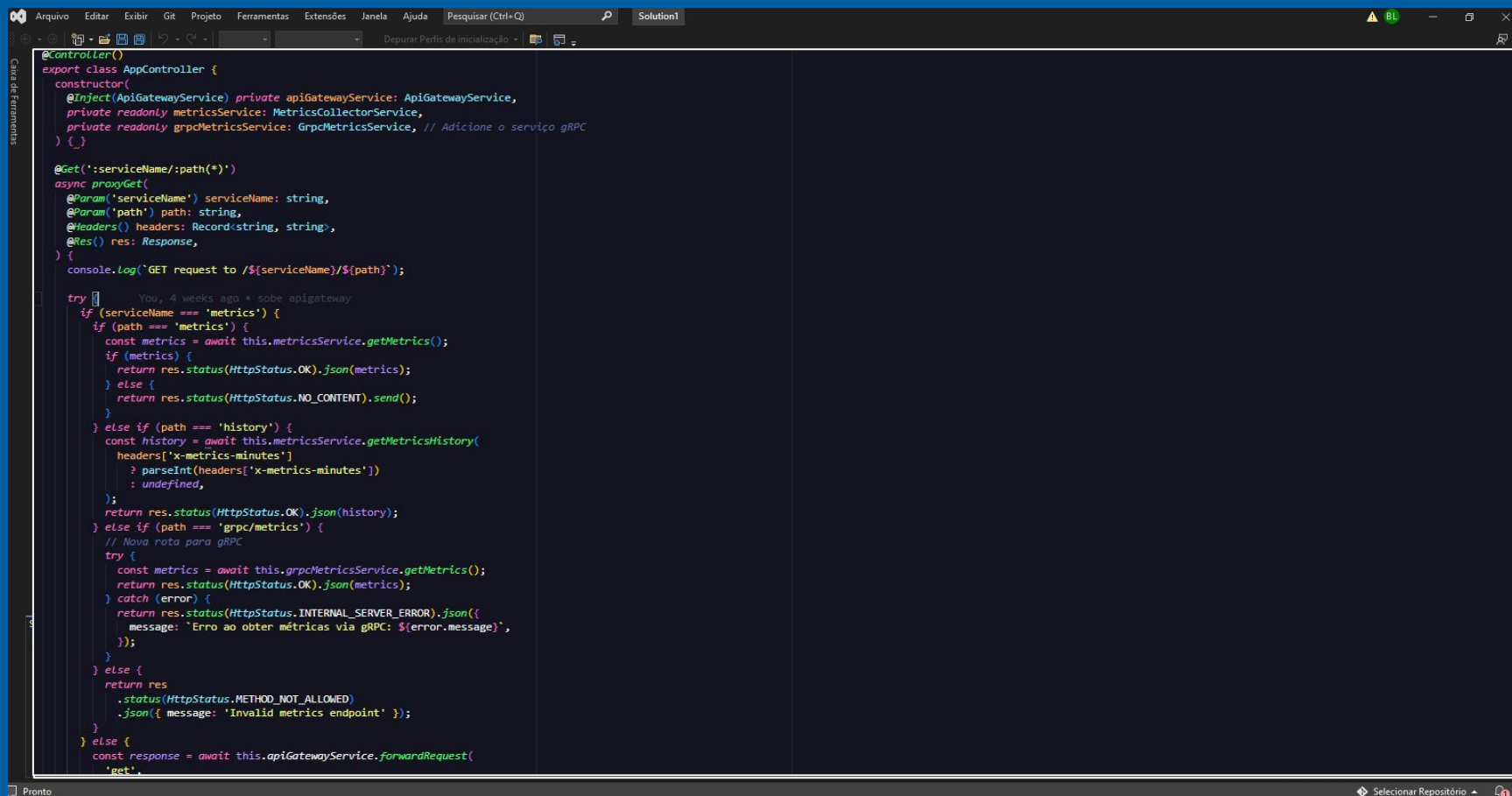
Descrição do Componente: Implementa um API Gateway usando o framework NestJS. O objetivo é fornecer uma camada de entrada unificada para múltiplos serviços, facilitando a gestão de requisições e respostas.

Linguagem: NestJS com REST

Funcionalidades/Técnicas:

- Descoberta de Serviços: Utiliza um serviço de descoberta para localizar os endereços IP dos serviços downstream.
- Encaminhamento de Requisições: Encaminha requisições HTTP para os serviços correspondentes com base no nome do serviço e no caminho da requisição.
- Suporte a Métodos HTTP: Suporta os métodos GET, POST e DELETE.
- Gestão de Erros: Trata erros e fornece respostas adequadas em caso de falhas.
- Coleta de Métricas: Coleta e armazena métricas de uso de CPU e memória em intervalos regulares.
- Histórico de Métricas: Permite recuperar o histórico de métricas por um período de tempo específico.
- Resiliência na Coleta de Métricas: O serviço de coleta de métricas é resiliente a falhas temporárias de conexão, tentando se reconectar de forma automática antes de reportar um erro.

Print do código



```
@Controller()
export class ApiController {
  constructor(
    @Inject(ApiGatewayService) private apiGatewayService: ApiGatewayService,
    private readonly metricsService: MetricsCollectorService,
    private readonly grpcMetricsService: GrpcMetricsService, // Adicione o serviço gRPC
  ) {}

  @Get('/:serviceName/:path(*)')
  async proxyGet(
    @Param('serviceName') serviceName: string,
    @Param('path') path: string,
    @Headers() headers: Record<string, string>,
    @Res() res: Response,
  ) {
    console.log(`GET request to /${serviceName}/${path}`);

    try {
      // You, 4 weeks ago + sobre apigateway
      if (serviceName === 'metrics') {
        if (path === 'metrics') {
          const metrics = await this.metricsService.getMetrics();
          if (metrics) {
            return res.status(HttpStatus.OK).json(metrics);
          } else {
            return res.status(HttpStatus.NO_CONTENT).send();
          }
        } else if (path === 'history') {
          const history = await this.metricsService.getMetricsHistory(
            headers['x-metrics-minutes']
              ? parseInt(headers['x-metrics-minutes'])
              : undefined,
          );
          return res.status(HttpStatus.OK).json(history);
        } else if (path === 'grpc/metrics') {
          // Nova rota para gRPC
          try {
            const metrics = await this.grpcMetricsService.getMetrics();
            return res.status(HttpStatus.OK).json(metrics);
          } catch (error) {
            return res.status(HttpStatus.INTERNAL_SERVER_ERROR).json({
              message: 'Erro ao obter métricas via gRPC: ${error.message}',
            });
          }
        } else {
          return res
            .status(HttpStatus.METHOD_NOT_ALLOWED)
            .json({ message: 'Invalid metrics endpoint' });
        }
      } else {
        const response = await this.apiGatewayService.forwardRequest(
          'get',
        );
      }
    }
  }
}
```

Frontend



Conceito Literário: Algumas técnicas/conceitos estudadas para tentar aplicar no projeto:

- WebSockets: Comunicação full-duplex para atualizações em tempo real entre cliente e servidor.
- Flux: Gestão de estado unidirecional e imutável, inspirada em sistemas distribuídos.
- Gerenciamento de Cache: Melhora de desempenho e experiência do usuário com armazenamento local de dados.

Descrição do Componente: Interface acessada pelos clientes, consome dados de métricas da API Gateway e constrói cards dinâmicos

Linguagem: ReactJS e css

Print do código

```
// Função para determinar o estado baseado nas métricas
const determineState = (metric) => {
  const cpuUsage = Math.max(metric.uso_de_cpu_porcentagem_core0, metric.uso_de_cpu_porcentagem_core1);
  const memoryUsage = metric.uso_de_memoria_porcentagem_total;
  const storageUsage = metric.uso_de_armazenamento_porcentagem_discoC;

  if (cpuUsage > 90 || memoryUsage > 90 || storageUsage > 90) return '#Crítico';
  if (cpuUsage > 70 || memoryUsage > 70 || storageUsage > 70) return '#Alerta';
  return '#Normal';
};

// Filtragem única combinando busca por IP e estado
const filteredData = Object.entries(metricsData)
  .filter(([ip, metrics]) => {
    if (!ip || ip === "" || !metrics || metrics.length === 0) return false;

    const state = determineState(metrics[metrics.length - 1]);
    const searchLower = searchQuery.toLowerCase();

    // Busca por IP ou estado
    const matchesIP = ip.toLowerCase().includes(searchLower);
    const matchesState = searchLower
      ? state.toLowerCase().includes(searchLower) ||
        state.toLowerCase().replace('#', '').includes(searchLower)
      : true;

    // Verificação dos filtros selecionados
    const matchesFilters = selectedFilters.length === 0 ||
      selectedFilters.includes(ip) ||
      selectedFilters.includes(state);

    return (matchesIP || matchesState) && matchesFilters;
  });

return (
```

Building the code...



Obrigado!

Dúvidas ou sugestões?