Associação Propagadora Esdeva
Centro Universitário Academia – UniAcademia
Curso de Bacharelado em Engenharia de Software
Trabalho de Conclusão de Curso – Artigo

Análise da Eficiência da Transferência de Dados em uma Rede de Microserviços – Proposta de Comparação de Desempenho entre REST E GRPC

Maicon Alcântara de Oliveira[1]
Centro Universitário Academia - UniAcademia, Juiz de Fora, MG
Romualdo Monteiro de Resende Costa[2]
Centro Universitário Academia - UniAcademia, Juiz de Fora, MG

RESUMO

A arquitetura de microsserviços é uma proposta que visa facilitar a criação de soluções distribuídas, onde cada serviço é implementado com o objetivo de resolver um problema específico do negócio. Nesse contexto, cada serviço é, normalmente, independente e pouco acoplado aos demais e, por isso, é preciso que exista um meio de comunicação eficiente entre eles. Assim, problemas com a comunicação podem inviabilizar o projeto, pois ela desempenha um papel essencial para essa arquitetura, ligando um serviço ao outro. A partir dessa perspectiva, este artigo apresenta duas aplicações, cada uma usando uma tecnologia diferente de comunicação entre microserviços, com o objetivo de comparar o desempenho da comunicação entre os serviços usando *REST* e *gRPC*. Nos testes realizados, a aplicação que utilizou gRPC, obteve um melhor desempenho para trafegar os dados em uma rede de microserviços.

Palavras-chave: Microsserviços, Comunicação, *gRCP*, *Protobubf*, *REST*, *JSON*, Métricas de performance.

[1] Discente do curso de Engenharia de Software do Centro Universitário Academia – UniAcademia. e-mail: maicon.soft.eng@gmail.com

[2] Docente do curso de Engenharia de Software do Centro Universitário Academia. UniAcademia. e-mail: romualdocosta@uniacademia.edu.br

ABSTRACT

The microservices architecture is a proposal that aims to facilitate the creation of distributed solutions where each service is implemented with the objective of solving a specific business problem. Each service is normally independent and little coupled to the others, so there must be an efficient means of communication between them. Problems with communication can make the project unfeasible, as it plays an essential role in microservices architecture, linking one service to another when necessary, allowing them to work as if they were a single application like the many we are used to using, such as Netflix, Uber, Spotify. This article presents two applications, each using a different communication technology between microservices, REST and gRPC, with the objective of comparing the performance of the communication between the services, for this purpose, metrics such as throughput analysis, latency analysis and execution time. of applications were used to measure performance in each application. In the tests carried out, the application that used gRPC, obtained a better performance to travel the data in a network of microservices.

Keywords: *Microservices, Communication, gRCP, Protobubf, REST, JSON, Performance metrics.*

1. INTRODUÇÃO

Constantemente, engenheiros de software buscam encontrar novas formas e métodos eficazes para desenvolver sistemas. Em razão disso, a Arquitetura de Microsserviços tem sido um assunto que vêm despertando o interesse de pesquisas, tanto no meio acadêmico quanto na indústria (SANTOS, 2019). Empresas como *Netflix*, *Amazon*, *Uber*, *eBay* e *Twiter* estão deixando suas aplicações monolíticas e adotando a arquitetura de microsserviços (FOWLER J., 2017. P18). Com microsserviços é possível construir de forma eficaz aplicativos e sistemas de informação, uma vez que eles definem entidades pequenas, fáceis de manter e com um baixo acoplamento entre si. Eles desempenham apenas tarefas limitadas e bem definidas, por isso, é necessário combinar vários microsserviços para alcançar a entidade desejada, sendo fundamental a comunicação entre eles.

Considere, como exemplo, um *e-commerce*, formado por vários microsserviços que fornecem os dados necessários para a visualização do produto, como o seu

nome, preço e imagem. Para o fornecimento desses dados, chamadas de *Application Programming Interface (APIs*), implementadas de maneira tradicional, através do modelo arquitetural chamado Transferência de Estado Representativa (*REST*), do inglês *Representational State Transfer,* são usualmente oferecidas (TORRE, 2022). No entanto, a quantidade de dados enviadas através da rede nessa solução pode causar um aumento do *payload*, que é a carga útil de informação a ser transmitida. Por sua vez, esse aumento das informações que trafegam em uma rede de serviços APIs pode produzir outros problemas, como o aumento da latência na comunicação, sendo esse o tempo que leva para uma solicitação chegar a um servidor e para a resposta viajar de volta (HALILI, 2008, p121).

Segundo Gonzaga (2019), em uma arquitetura de microsserviços dois princípios devem ser respeitados, que são a escalabilidade e a resiliência. Assim, considerando que o modelo arquitetural *REST* pode, tradicionalmente, apresentar problemas como o elevado consumo de recursos e a baixa capacidade de transferência de dados, outras opções devem ser consideradas como, por exemplo, o *Google Remote Procedure Call* (*gRPC*) (RYAN, 2015). Essa alternativa é um *framework* de alta performance baseado em *Remote Procedures Call* (*RPC*), onde você só precisa definir as requisições e respostas e ele cuida restante. Ele usa um formato de mensagem chamado *Protocol Buffers* (KHARE, 2018), que busca melhorar a transferência de dados, tornando as cargas de mensagens menores, consumindo poucos recursos e transferindo uma maior quantidade de dados em um tamanho menor de mensagem.

Devido às características mencionadas do *gRPC* e a ampla utilização do *REST*, este trabalho busca analisar a transferência de dados entre microserviços, comparando a eficiência da arquitetura de comunicação entre essas duas soluções, buscando responder à seguinte questão: Dentre os modelos arquiteturais de serviço *Web*, *REST* e *gRPC*, qual apresenta melhor desempenho para a transferência de dados entre microserviços?

Para tratar dessa questão, as próximas seções apresentam os detalhes deste trabalho, iniciando pela Seção 2, onde são apresentados os trabalhos correlatos, os recursos e benefícios da arquitetura de microsserviços, exibindo um paralelo entre sistemas monolítico e microsserviços. A seguir, a Seção 3 apresenta as distintas

arquiteturas e mensagens, *REST*, com o formato de mensagem *JSON* e o *gRPC*, com o formato de mensagem *Protobuf*. A Seção 4 apresenta o desenvolvimento deste trabalho, onde foi utilizada a linguagem C# para criação dos microsserviços usando diferentes tecnologias para comunicação entre si, e realizada a comparação do desempenho entre os microsserviços, onde cada senário de teste utiliza a arquitetura cliente/servidor para se comunicarem através de uma rede local. Finalmente, a quinta, e última seção, apresenta a conclusão, as considerações finais e os possíveis trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 TRABALHOS CORRELATOS

BACK (2016) realizou uma análise comparativa de técnicas de integração entre microsserviço, o trabalho avalia, de forma analítica, REST e AMQP como diferentes técnicas de integração entre microsserviços, mostrando as vantagens e desvantagens encontradas em cada uma delas. Por fim ele chegou-se à conclusão de que a existência de diferentes técnicas de integração entre serviços é justificada com os testes realizados em seu artigo, pois cada técnica possui aplicações conforme suas vantagens e desvantagens que devem ser ponderadas de acordo com cada tipo de sistema que se deseja implementar.

MENDONÇA (2022), em seu trabalho realizou a avaliação experimental de uma arquitetura de microsserviços para o gerenciamento de notas fiscais eletrônicas, em uma arquitetura microsserviços, usando um do ponto de vista qualitativo, através dos critérios de manutenibilidade e portabilidade, e quantitativo, em que os aspectos de desempenho e escalabilidade são mensurados através de uma experimentação fazendo uso das tecnologias como *gRPC* e *REST*. O experimento realizado consistiu em se executar as operações de inserção e consultas em subconjuntos progressivamente maiores das notas fiscais geradas, com cada operação sendo executada cinco vezes, e mensurar os tempos de execução. As médias dos tempos para cada operação e conjunto de dados foram então extraídas e comparadas entre uma arquitetura tradicional e a arquitetura de ControleNF criada pelo autor, que utilizou tecnologias como REST API para se comunicar com aplicações externas e gRPC para a comunicação interna entre os microsserviços. Assim, concluiu-se que a arquitetura, da maneira que foi proposta não reúne condições para substituir a

arquitetura tradicional mesmo que haja potenciais benefícios em relação aos requisitos de manutenibilidade e portabilidade.

CRUZ (2021) realizou uma modelagem de microsserviços e arquiteturas de apis, ele utilizou modelos de *APIs* como *gRPC*, *REST*, *SOAP* e *GraphQL* para a construção de aplicativos em uma arquitetura de microserviços, trazendo uma nova visão ao desenvolvedor, onde é necessário conhecer esses padrões para saber onde utilizá-los e como utilizá-los, pois cada implementação traz características diferentes que são mais vantajosas em alguns cenários do que em outros. No entanto concluiu que existem benefícios para as aplicações de grande porte, como a facilidade de divisão em times que, por fim, agiliza o processo de produção, a confiabilidade gerada com base no fator de que o serviço que apresentar falha não impactará a aplicação como um todo, além de que para a comunicação dos serviços os padrões de modelagem de *APIs* como *REST*, *gRPC*, *SOAP* e *GraphQL* são mais performáticos, contrário ao *SOAP* que é um padrão mais antigo.

Nas literaturas citadas nesse trabalho o *REST* em sua essência é usado para a transferência de dados entre serviços e ou sistemas, também encontrei trabalhos onde o *REST* é usando para comunicação entre os serviços internos e o *gRPC* para comunicação com serviços externos. Neste trabalho a abordagem adotada, diferente da maioria, é usar gRPC tanto para a integração interna quanto para externa de microsserviços, buscando a interoperabilidade entre sistemas, um aumento de performance na comunicação entre serviços e consequentemente a diminuição de gastos com escalonamento vertical devido ao baixo custo da transferência de dados ao fazer uso do *gRPC*.

2.2 ARQUITETURA DE MICROSSERVIÇOS

Os Microsserviços surgiram como um estilo importante da arquitetura de *software*. A primeira definição considerada para microsserviços foi em 2014 em um artigo que descrevia uma arquitetura para desenvolver sistemas através de pequenas coleções de serviços que operavam seus próprios processos e se comunicavam entre si por meios de mecanismos leves. Em relação a esse termo podemos afirmar que:

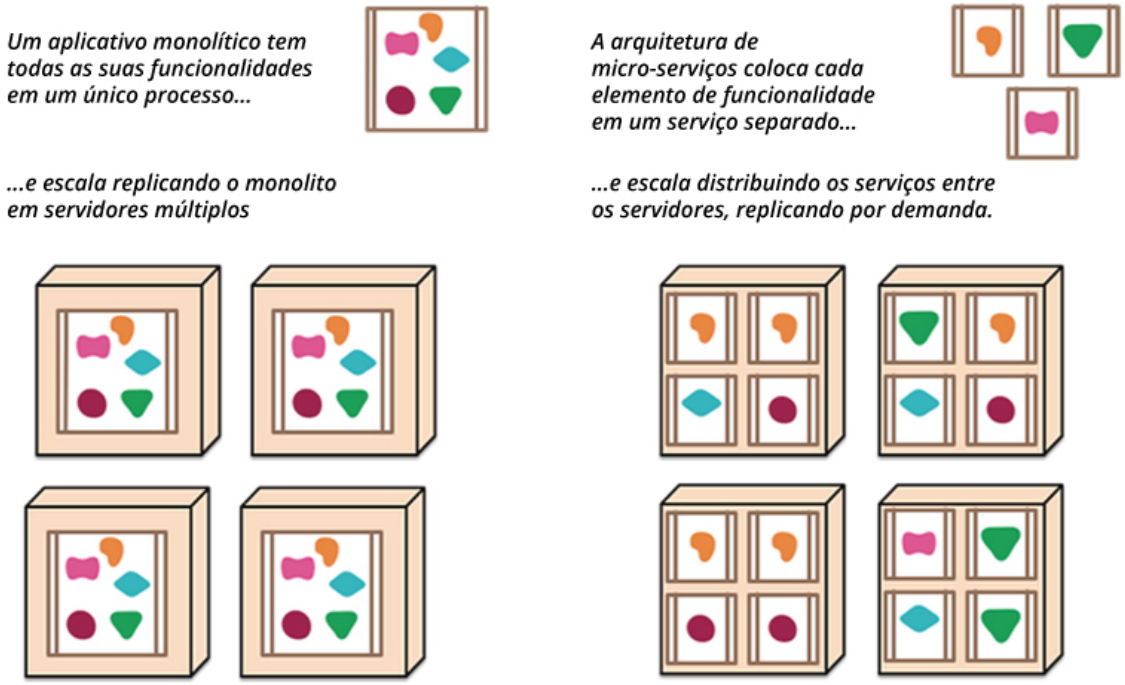
> *"Arquitetura de Microsserviços surgiu nos últimos anos para descrever uma maneira particular de projetar aplicativos de software como conjuntos de*

> *serviços implantáveis independentemente. Embora não haja uma definição precisa desse estilo arquitetônico, existem certas características comuns em torno da capacidade de negócios, implantação automatizada, inteligência nos terminais e controle descentralizado de linguagens e dados." (*FOWLER e LEWIS, 2014).

A arquitetura de microsserviços é melhor explicada quando comparada a um estilo tradicional, usualmente chamado de monolito, que são aplicações construídas em um único bloco de código responsável por todos os processos, da interação com o usuário até o acesso ao banco de dados (SANTOS, 2019, p15). No *software* desenvolvido de forma tradicional, os recursos funcionam em um mesmo processo, acomodando, assim, as várias necessidades de diferentes negócios em um só lugar. Por outro lado, em uma arquitetura de microsserviços, o desenvolvimento do software é dividido em partes que operam de forma independente, desenvolvidas para executar tarefas especificas *(*FOWLER e LEWIS, 2014).

Como pode ser visto na Figura 1, sistemas monolíticos apresentam toda a sua base de código contidas em um só lugar.

Figura 1 – Monólitos e microsserviços



Fonte: Fowler e Lewis (2014).

Como podemos observar na Figura 1 todas as funcionalidades se concentram em um mesmo bloco e os dados que servem à aplicação ficam em uma base de dados, compartilhados. Por sua vez, em microsserviços, cada processo da aplicação

se torna um único serviço. Nessa outra proposta, cada processo é independente, e eles se comunicam usando chamadas em uma rede. Cada pequeno serviço tem uma responsabilidade única e a sua própria base de dados. Eles são autônomos, podem ser desenvolvidos e aperfeiçoados, sem afetar a aplicação como um todo. Para que um *software* baseado em microsserviços funcione, deve ser criado um canal de comunicação entre os microsserviços. Um *e-commerce* é um exemplo de um sistema que requer várias partes para funcionar. Nesse exemplo, microsserviços separados podem ser desenvolvidos para implementar um módulo de pagamento, uma caixa de pesquisa, um módulo de entrega ou de gerenciamento de pedidos. Como existem muitos componentes separados, a comunicação entre eles é um dos aspectos mais importantes do seu desenvolvimento. De acordo com RICHARDSON (2019), a comunicação entre microsserviços é frequentemente implementada através de um modelo de solicitação-resposta (*request-response*) síncrono ou assíncrono. Esses mecanismos de comunicação são baseados em trocas de mensagens, que podem ser de diversos formatos, alguns legíveis por humanos como os baseados em texto, *JSON* ou *XML*, e outros não legíveis, como o *Protocol Buffers*, que é binário. Na Seção 3 as interfaces de *softwares* e suas adaptações para microsserviços serão apresentadas com mais detalhes.

2.3 BENEFÍCIOS E DESAFIOS DA ARQUITETURA DE MICROSSERVIÇOS

Empresas que optam por implantar a arquitetura de microserviços no desenvolvimento de *software*, muitas vezes, o fazem após terem construído uma aplicação e encontrado desafios organizacionais e de escalabilidade. Elas iniciam com aplicação monolítica e depois migram para microserviços (FOWLER J., 2017). Sistemas monolíticos possuem vantagens como a inteligibilidade arquitetural, onde há poucas camadas para se preocupar. A solução é criada em uma mesma tecnologia, os times se tornam mais próximos, gerando um conhecimento homogêneo devido a simplicidade da arquitetura, tornando a implantação mais fácil e o desenvolvimento a mais ágil. Porém, um dos maiores perigos apresentados por uma arquitetura monolítica, é que, caso ocorra falha em um processo do sistema, o mesmo poderia ficar inativo, além do fato de que adicionar novas funcionalidades e melhorias se torna mais complexo à medida que o código cresce (ALMEIDA, 2015). O uso de uma única tecnologia também pode ser considerado uma desvantagem, uma vez que um

problema poderia ser resolvido facilmente com outra tecnologia existente (RICHARDSON, 2014).

Microsserviços, por serem serviços leves e independentes, realizam funções únicas, cooperando com outros serviços similares através de uma interface de comunicação bem definida. Devido sua natureza minimalista, a arquitetura de microsserviços cada vez ganha mais espaço na área da engenharia de software, resolvendo problemas inerentes a aplicações monolíticas que se utilizam de serviços altamente acoplados, executados como um único serviço. Na arquitetura de microsserviços, os processos da aplicação se tornam componentes independentes como apresentado na Figura 1, tendo como principais características a autonomia e a singularidade. Isso significa que cada serviço pode ser desenvolvido, implementado, operado e dimensionado de forma independente, sem afetar a aplicação como um todo. Uma vez que microsserviços executam funções específicas, caso o código base cresça, ele pode ser dividido em outros serviços menores, diminuindo, assim, a sua complexidade. Pelo fato de serem pequenos, os microsserviços geralmente são mantidos por equipes menores, que ficam focadas em um contexto restrito e bem compreendido. Segundo Almeida (2015), uma base de código menor facilita a curva de aprendizado do projeto para um novo desenvolvedor do time. De acordo com NEWMAN (2015), essa abordagem permite a escolha das melhores ferramentas para cada projeto, possibilitando o dimensionamento deles de acordo com a necessidade de cada sistema. Esse benefício permite uma melhor gestão dos recursos de infraestrutura, de acordo com a exigência de cada programa.

Uma vez que cada serviço possua pouca responsabilidade, de acordo com o *SRP*, *Single Responsibility Principle* ou Princípio de Responsabilidade Única (TORRE, 2022), é possível isolar as falhas para realizar correções e ainda manter os outros serviços em execução normalmente. Sendo fundamental para sistemas críticos, onde a habilidade de recuperação de falhas é extremamente importante, permitindo realizar o gerenciamento de falhas de forma otimizada. De acordo com LEWIS e FOWLER (2014), a arquitetura de microsserviços nem sempre é a mais ideal para todas as abordagens. Em alguns casos existem desvantagens, como o aumento da complexidade operacional, sendo que para gerenciar sistemas com uma estrutura criada em microsserviços é necessária uma equipe de Desenvolvimento e Operações

(*DevOps)* madura, capaz de trabalhar de forma colaborativa, analisar dados e levar o produto do desenvolvimento à implantação no ambiente de produção. Com uma extensa rede de microsserviços implementados em uma empresa qualquer, a governança deles se torna um desafio com centenas de componentes distribuídos que precisam trabalhar como se fossem uma única aplicação, muitas das vezes, o trabalho de governança pode deixar a desejar. Além disso um problema recorrente é uma possível maior lentidão do sistema, devido à falta de recurso para manter inúmeros serviços comunicando entre si através de uma cadeia de redes, gerando inúmeras interdependências, comprometendo, assim, o funcionamento da infraestrutura como um todo.

3 ARQUITETURAS E SEUS FORMATOS DE MENSAGENS

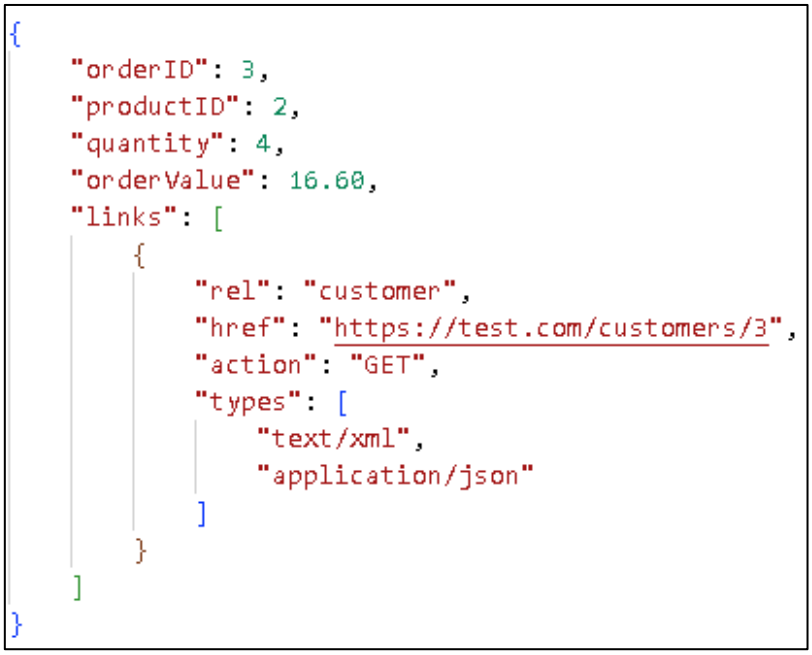
As subseções a seguir apresentam as arquiteturas *REST* e *gRPC*, bem como os seus respectivos formatos de mensagens *JSON* e *Protocol Buffer*.

3.1 *REST e JSON*

REST é um estilo de desenvolvimento de *web services* que se originou com a tese de doutorado de Roy Fielding em 2000 (SAUDATE, 2014). Segundo FOWLER (2010), o modelo *REST* é baseado em *HTTP* e possui funções que representam ações que permitem a manipulação de um sistema. Sua arquitetura pode ser dividida nas seguintes abstrações: recursos, representações, URIs e em solicitação *HTTP*. Essas abstrações formam a interface de comunicação entre o cliente e o serviço. A URI ou Identificador de Recurso Uniforme é um hiperlink distinto que aponta para um recurso e transmite suas representações. O *HTTP* inclui vários métodos para trabalhar com os recursos, são eles o *GET* que retorna um recurso, o *POST* que cria um recurso, o *PUT* que seve para atualizar um recurso e o *DELETE* que exclui um recurso. NEWMAN (2015) expões alguns pontos negativos para o uso do *REST* com *HTTP*, como o fato de estar sujeito a problemas de performance e latência de rede e que nem todos os navegadores suportavam os verbos. Um recurso é um objeto que pode ser apontado ou transferido pela *Web*, como um artigo acadêmico. Uma representação de um recurso pode ser por exemplo uma imagem, uma página *HTML* ou um arquivo *JSON*.

O *JSON* é um padrão aberto para formato de arquivo usado para transferência de dados. Seu formato de texto é legível, sendo os objetos de dados compostos de pares de valor de atributo, bem como tipos de dados tabulares chamados chave-valor. Como seu próprio nome indica, *JSON* é originado da linguagem de programação *ECMAScript*, ele define um pequeno conjunto de regras estruturadas para a representação de dados (ECMA, 2017). A figura 2 fornece um exemplo de dados no formato *JSON* que apresenta informações de um pedido.

Figura 2 – Exemplo de dados formato JSON.



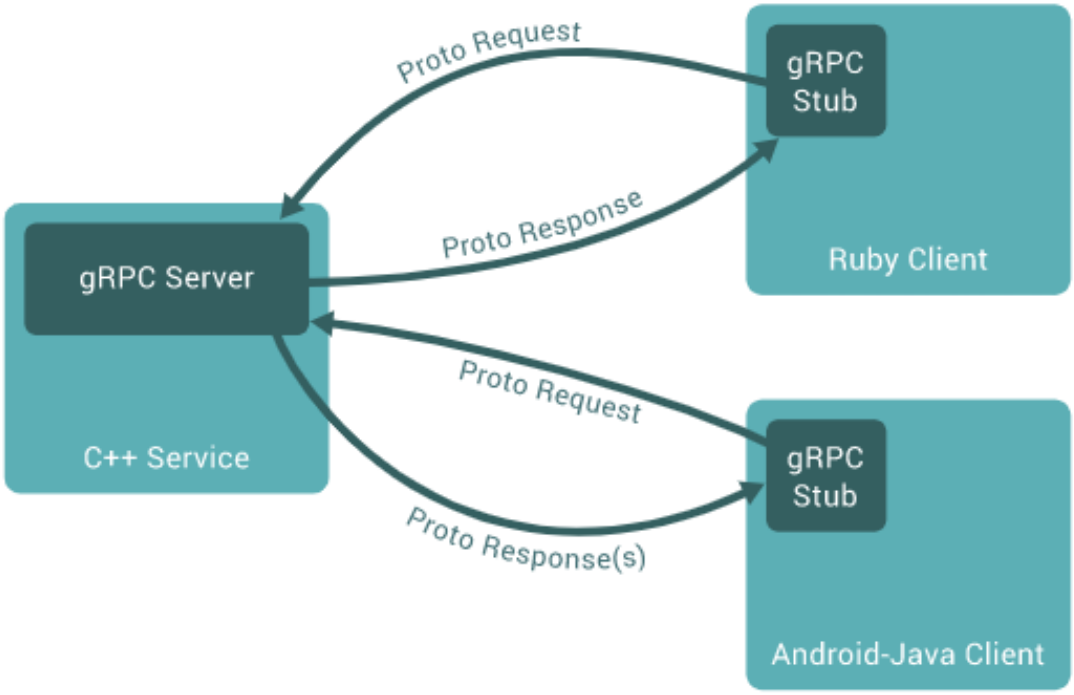
Na Figura 2, além dos dados no formato *JSON*, contendo informações de um pedido, como a quantidade (*quantity*) e o valor (*orderValue*), a representação inclui links que identificam as operações disponíveis para o cliente do pedido. No caso apresentado na Figura 2, o link corresponde a uma ação para retornar um cliente e vinculá-lo ao pedido que pode conter um ou mais produtos, a quantidade de cada produto e valor total do pedido.

3.2 *gRPC* e *Protocol Buffers*

O *gRPC* é um *software* de código aberto desenvolvido pelo *Google*, que usa uma única infraestrutura *RPC* para conectar um grande número de microsserviços em execução em seus data centers (RYAN, 2015).

A Figura 3 representa como um aplicativo pode chamar diretamente um método de um servidor de forma distribuída usando *gRPC*. Nessa proposta, a primeira coisa que precisa ser feita é a descrição da interface de serviço. Ela contém informações de como consumir o serviço, quais métodos serão usados remotamente e quais parâmetros e tamanhos de mensagens podem ser usados para chamar o método. (Core concepts, architecture and lifecycle, 2021). Um aplicativo *gRPC* pode chamar diretamente um método no servidor, mesmo estando em máquinas diferentes, essa chamada ocorre como se fosse um objeto local, isso facilita a criação de aplicativos e serviços distribuídos (*Introduction* to *gRPC*, 2020).

Figura 3 – Chamada de método Remota gRPC.



Fonte*: Introduction to gRPC* (2020).

Por padrão, o *gRPC* usa o *protocol buffers* (*protobuf*), um formato de mensagens binária. O *Google* desenvolveu o *protocol buffers* para usar em seus serviços internos. Ele é um formato de codificação binário que permite a especificação de esquemas para os dados usando uma linguagem de especificação.

Por mais que *JSON* tenha um formato simples, legível e seja amplamente usado com *REST* como afirma REZVINA (2015), por ser um formato de mensagem baseado em texto, quando comparado a formatos binários, ele tende a ocupar mais espaço e largura de banda para a transferência de mensagens. Diferentemente, o

protocol buffers, tende a requerer menos espaço e largura de banda para transferir mensagens devido a sua natureza binária, sendo, assim, uma alternativa melhor para transferência de dados. A Figura 4 mostra a mesma mensagem em dois formatos: *JSON* e *Protobuf*.

Figura 4 – Estrutura de mensagem JSON e Binária

```
/* Mensagem JSON */
{
    "orderId": 1,
    "customerId": 123,
    "items": [987, 988],
    "couponCode": "ALLFREE",
    "paymentMode": "CASH",
    "shippingAddress": {
        "name": "Alice",
        "address": "xyz street",
        "pincode": "111111"
    }
}

/* Mensagem binária */
8,1,16,123,26,4,219,7,220,7,34,7,65,76,76,70,82,69,69,40,1,50,27,10,5,
65,108,105,99,101,18,
10,120,121,122,32,115,116,114,101,101,116,26,6,49,49,49,49,49,49
```

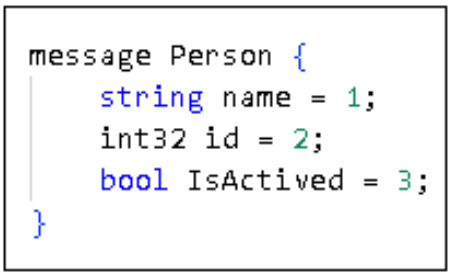
Fonte: *Protobuf — What & Why?* (2021).

Para NEWTON (2019), o *protobuf* como frequentemente é chamado, é usado para serializar mensagens *gRPC*, por ser um formato binário, ele serializa muito rapidamente no servidor e no cliente, resultando em cargas de mensagens pequenas que são excelentes para o cenário de largura de banda limitada, como, por exemplo os aplicativos moveis. Segundo KHARE, (2018) os buffers de protocolo são o mecanismo extensível de linguagem neutra para serializar dados estruturados, são pequenos, rápido e simples. *Protobuf* é definido por uma estrutura de dados que será serializada através de um arquivo chamado *proto*.

A Figura 5 representa um formato de mensagem contendo uma série de pares nome-valor, neste exemplo, o *protobuf* usa mensagens para estruturar dados, onde cada dado está escrito em formato de pares nome-valor, chamados de campos. É possível também definir serviços gRPC e especificar seus métodos que podem ser

chamados remotamente através de parâmetros. O *Protobuf* é a linguagem mais usada para descrever um serviço *gRPC*.

Figura 5 – Estrutura de mensagem *protobuf* (*Introduction to gRPC*, 2020).

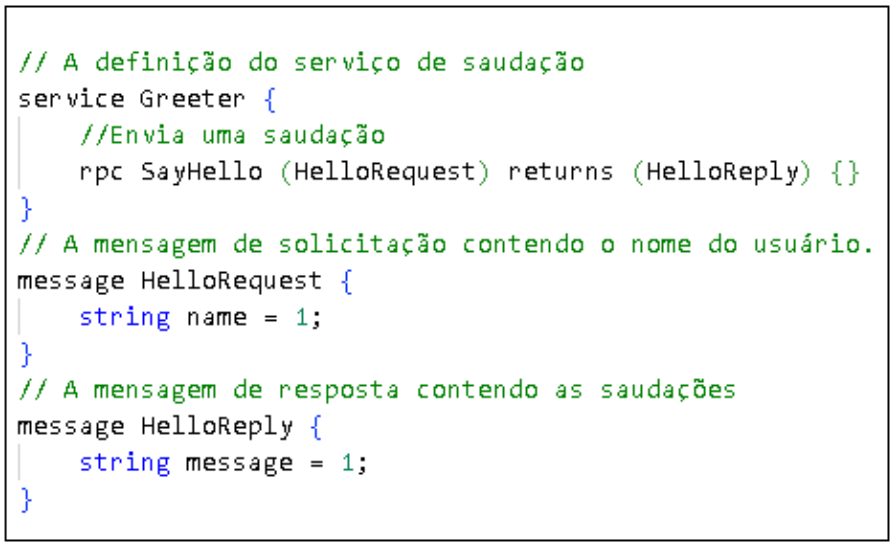


```
message Person {
    string name = 1;
    int32 id = 2;
    bool IsActived = 3;
}
```

Fonte: *Introduction to gRPC* (2020).

A Figura 6 demonstra a definição de um serviço de saudação *gRPC* com mensagens de solicitação contendo o nome do usuário e uma mensagem de resposta contendo a saudação para usuário.

Figura 6 – Serviço *gRPC* de *buffer* de protocolo.



```
// A definição do serviço de saudação
service Greeter {
    //Envia uma saudação
    rpc SayHello (HelloRequest) returns (HelloReply) {}
}
// A mensagem de solicitação contendo o nome do usuário.
message HelloRequest {
    string name = 1;
}
// A mensagem de resposta contendo as saudações
message HelloReply {
    string message = 1;
}
```

Fonte*: Introduction to gRPC* (2020).

4. Desenvolvimento do trabalho

As aplicações criadas como objeto de estudo deste trabalho caracterizam-se como dois microsserviços, um usando o conjunto *REST* e *JSON* e o outro usando o conjunto *gRPC* e *Protobuf* como tecnologias de transferência de dados entre seus serviços. Essas aplicações têm por objetivo enviar e receber um grande de fluxo de dados, para que com isso seja possível a análise do tráfego e, assim, observar qual das duas tecnologias apresentam a melhor eficiência na transferência de seus dados.

No desenvolvimento dos microserviços, optou-se pela linguagem de programação C#[1] e para a persistência a base de dados utilizada foi o *SQL Server*[2], também foram utilizadas ferramentas para coleta e visualização de dados. O Prometheus[3] é uma ferramenta usada para coletar dados gerados pelo tráfego entre os serviços de cada aplicação, usando métricas como análise de latência e análise de vazão. Grafana[4] é uma ferramenta usada para transformar os dados coletados pelo Prometheus em informações que podem ser visualizadas e analisadas através de dashboards, que são painéis que exibem indicadores metrificados de forma visual (NASCIMENTO, 2017). Todos os testes foram executados no mesmo computador com as seguintes configurações: processador Intel(R) Core (TM) i7-8665U CPU @ 1.90GHz 2.11 GHz, RAM instalada: de 16,0 GB (utilizável: 15,8 GB) e sistema operacional de 64 bits.

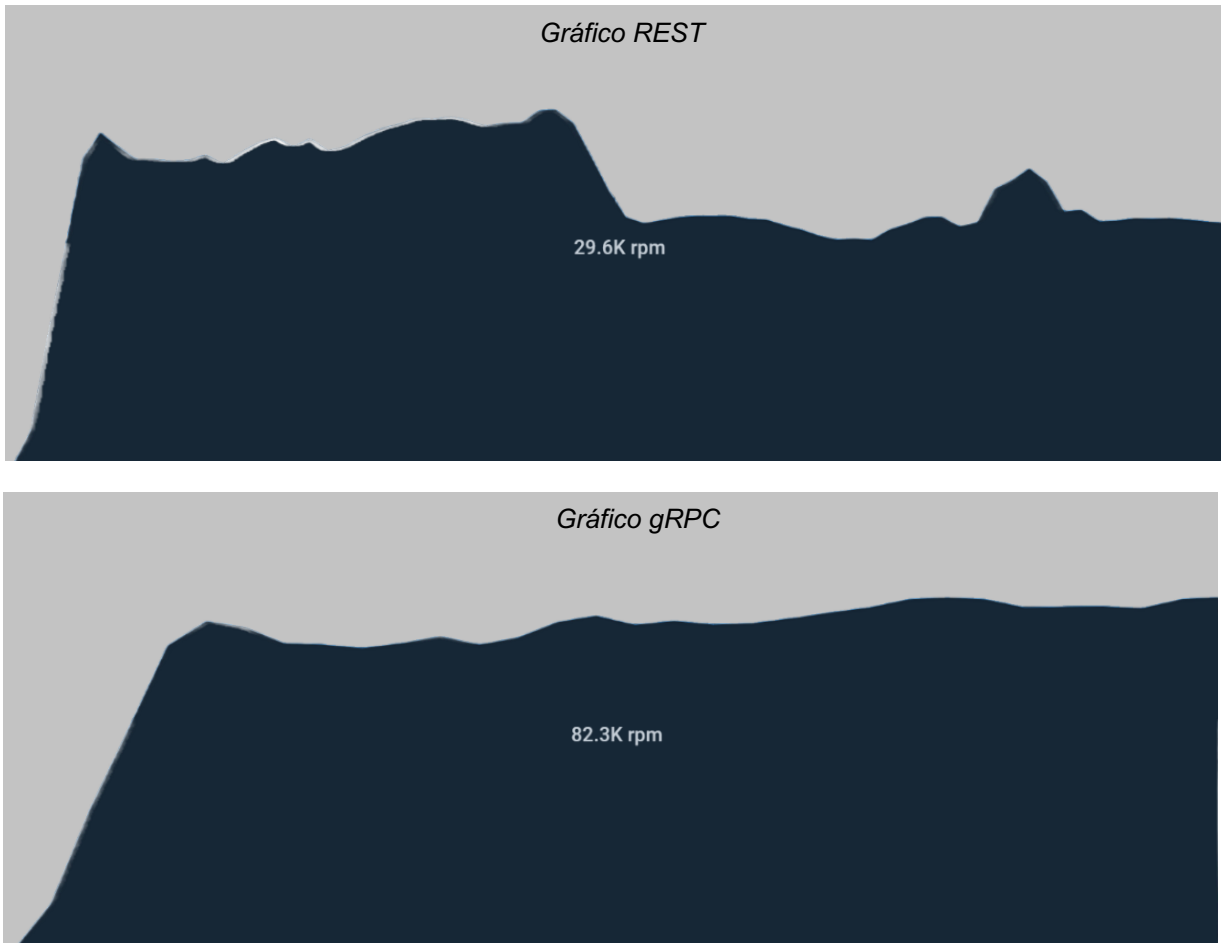
4.2 Resultados

Segundo BOARGES (2021) as métricas são medições importantes que ajudam a analisar os resultados obtidos durante um teste, sendo assim possível identificar os pontos positivos e negativos no sistema. Para gerar um volume de dados passível de análise, uma carga de 200.000 (duzentas mil) mensagens foi enviada entre os serviços de uma mesma aplicação, no primeiro experimento foram usados REST com *JSON* e no segundo experimento *gRPC* com *Protobuf*.

A primeira métrica escolhida foi de análise da vazão, que segundo SALES (2000) é a taxa temporal na qual os pacotes de pedidos são atendidos pelo sistema, ou seja, a quantidade máxima de requisições feita para o servidor durante a execução da aplicação. O primeiro experimento (*REST* com *JSON*) obteve um resultado médio de 29.6 mil requisições por minuto. Já o segundo experimento (*gRPC* com *Protobuf*) a taxa de transferência média por minuto foi de 82.2 mil requisições por minuto, como pode ser observado na Figura 7, onde é apresentado o gráfico gerado pela ferramenta Grafana.

1 learn.microsoft.com/pt-br/dotnet/csharp
2 www.microsoft.com/pt-br/sql-server/sql-server-2019
3 Prometheus.io
4 Grafana.io

Figura 7 – Gráfico da análise da vazão (do autor).



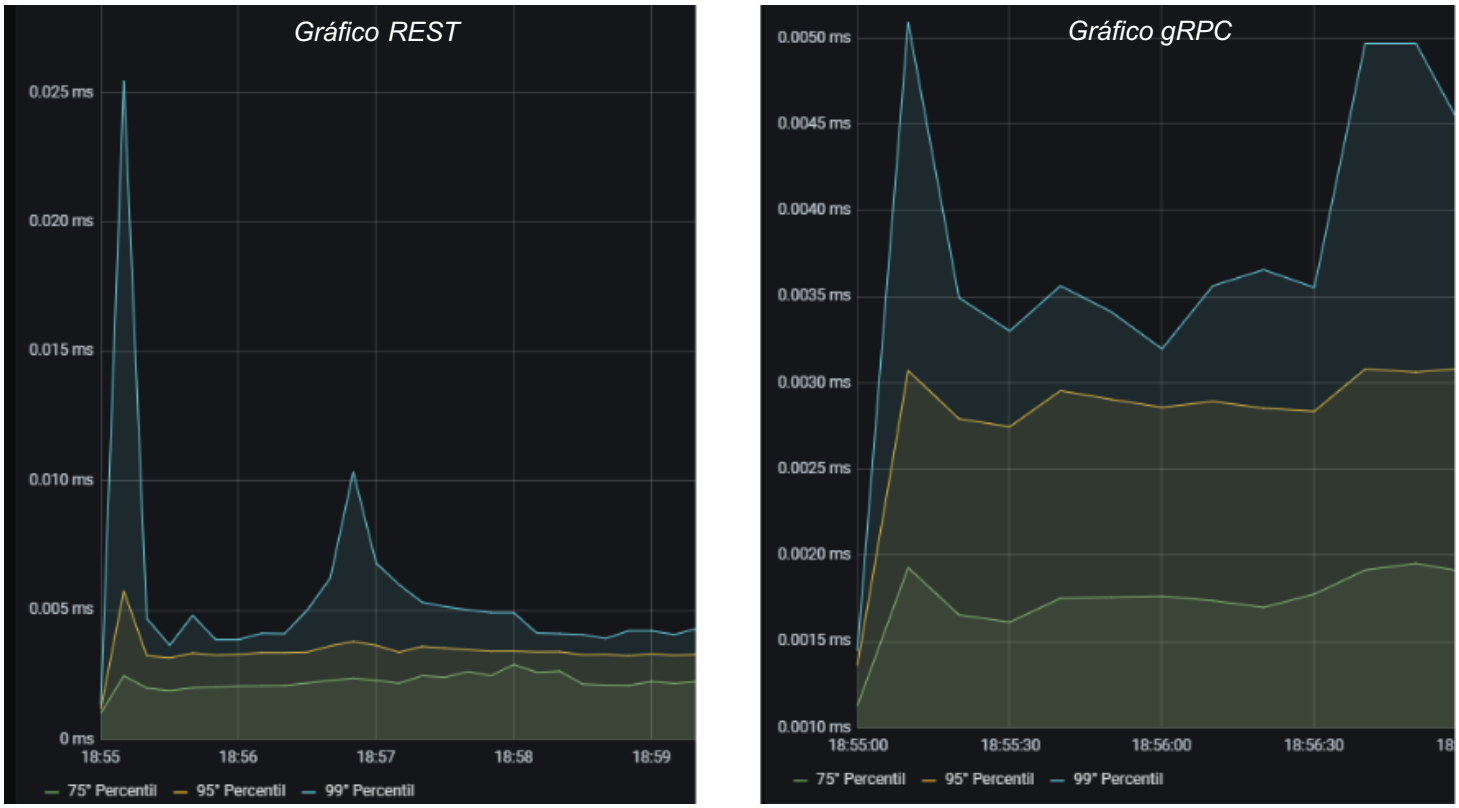
Como pode ser observado na Figura 7, o segundo experimento obteve uma máxima de 82.2 mil requisições, equivalendo ao aumento de 52.6 mil transferências a mais que o primeiro, devido ao uso do *gRPC* com o *Protobuf*, tecnologia usada pelo segundo experimento.

A próxima métrica utilizada foi a de análise da latência, para SALES (2000), a latência é o tempo que um pacote de dados trafega de um ponto ao outro, fim-a-fim, em uma rede. Essa métrica pode expressar resultados obtidos em milissegundos para os seguintes percentis de 75%, 95% e 99%. De acordo com LEAL (2022), percentis são um meio de popular e especificar a dispersão máxima e mínima de uma amostra.

No primeiro experimento usando o *REST* com *JSON*, em 75% das chamadas o tempo médio de respostas foi de 0,00257 milissegundos, em 95% delas o tempo médio de resposta foi de 0,00334 milissegundos e em 99% das chamadas o tempo médio de resposta foi de 0,00465 milissegundos. No experimento 2, usando *gRPC* com *Protobuf* foram obtidos, para 75% das chamadas a média de 0,00181

milissegundos, para 95% foi de 0,00285 milissegundos e, em 99% das chamadas, o tempo médio das respostas foi de 0,00370 milissegundos, conforme apresentado na Figura 8.

Figura 8 – Gráfico da análise da Latência (do autor)



Pode ser observado que em 75% das requisições o experimento 2 foi 0,00076 milissegundos mais rápidos que o primeiro, na média, em 95% das requisições o experimento 2 superou o primeiro em 0,00049 milissegundos e em 99% das requisições o experimento 2 obteve um tempo de resposta de 0,00095 milissegundos a mais quando em comparação com o primeiro experimento. Desta forma, é possível observar que a aplicação usando *gRPC* com *Protobuf* mostrou-se mais performática quanto ao tempo de resposta.

A última métrica analisada foi a do tempo total gasto para que as aplicações concluíssem a tarefa, o primeiro experimento levou um total de 00:17:19 (dezessete minutos e dezenove segundos) para transmitir as mensagens usando o método *REST* com *JSON*, em comparação, o segundo experimento que usa *gRPC* com *Protobuf* executou a tarefa em um tempo total de 00:7:46 (sete minutos e quarenta e seis segundos). Para calcular a diferença percentual do tempo total obtido entre os

experimentos, considera-se a variação percentual dada por *((Tb – Ta) / Tb) x 100*. Como pode ser observado na Figura 10, onde T_a e T_b correspondem, respectivamente, ao maior e o menor tempo de execução das aplicações.

Figura10 – Fórmula da variação de percentual

$$Variação\ percentual = \frac{(Valor\ maior - Valor\ menor)}{Valor\ Maior} x100$$

Para este cálculo foi preciso converter o tempo total de cada experimento para uma única unidade tempo, então, o primeiro experimento levou 1339 segundos em tempo de execução e o segundo experimento 466 segundos. A variação do tempo de execução entre eles foi de -55,14%. A redução do tempo do segundo experimento para o primeiro representa o ganho de desempenho obtido entre o microsserviço usando *gRPC* em comparação ao que usa *REST*.

5. Conclusão

Neste trabalho foram analisados dois microserviços, cada um usando uma tecnologia distinta para comunicação, sendo elas o *REST/JSON* e *gRPC/Protobuf*, com o intuito de comparar o desempenho dessas tecnologias através da análise de métricas de performance. Com base nos resultados obtidos, pode-se afirmar que a tecnologia *REST* funciona bem com a interface *JSON* para trafegar informações entre uma rede de microserviços. Porém o *gRPC* juntamente com Protobuf, como observado neste trabalho provou ser mais performático no que diz respeito ao tempo de resposta da transferência de dados entre os serviços, quanto a carga máxima de dados transferidos por minuto, além de obter o menor tempo de latência na comunicação entre as requisições enviadas pelo microsserviço.

Nos experimentos realizados neste trabalho não foram utilizadas técnicas como compressão de *HTTP*, isso talvez aceleraria a transferência de dados para o experimento que usou *REST* e *JSON*. Também não foi possível usar chamadas multiplexadas, onde várias mensagens são compactadas na mesma transmissão, usando somente um canal para enviarem e receberem pacotes de dados, o que pode melhorar ainda mais a velocidade de transmissão do experimento usando *gRPC* e *Protobuf*. Ficando assim como ideia de um trabalho futuro, pois possivelmente

utilizando-se dessas técnicas, aumentaria a performance de cada experimento, podendo ou não levar a um resultado diferente dos observados.

REFERÊNCIAS

ALMEIDA, Adriano. **Arquitetura de microserviços ou monolítica?** 2015. Disponível em <https://blog.caelum.com.br/arquitetura-de-microservicos-ou-monolitica/> Acesso 23 de ago. de 2020.

BACK, Renato Pereira. **Análise Comparativa de Técnicas de Integração entre Microsserviços**. Universidade Federal de Santa Catarina, 2016.

Core concepts, **architecture and lifecycle**. 2021. Disponível em <https://grpc.io/docs/what-is-grpc/core-concepts/>. Acessado em 17 fev. 2021.

CRUZ, Julio da Silva. **MODELAGEM DE MICROSSERVIÇOS E ARQUITETURAS DE APIS**. Faculdade de tecnologia de São Paulo, 2021.

ECMA, International. **The JSON Data Interchange Syntax**. 2 Edition/December, 2017. 16 p.

FIELDING, Roy. **Architectural Style and Design of Network-based Software Architectures**. California, 2000. 162 p.

FOWLER, J. Susan. **Microsserviços prontos para a produção: Construindo sistemas padronizados em uma organização de engenharia de software**. São Paulo: Novatec Editora Ltda, 2017. 16p.

FOWLER, Martin. **Richardson Maturity Model: steps toward the glory of REST**. 2010. Disponível em <https://martinfowler.com/articles/richardsonMaturityModel.html>. Acesso 10 nov. de 2020.

FOWLER, Martin; LEWIS, James. **Microsserviços** (2014). Disponível em: <https://martinfowler.com/articles/microservices.html> Acessado em 20 de jul de 2020.

FOWLER, Martin; LEWIS, James. **James Microsserviços em poucas palavras** (2015). Disponível em:

<https://www.thoughtworks.com/pt/insights/blog/microservices-nutshell> Acessado em 20 de jul de 2020.

GONZAGA, Rafael. **Comunicação entre microsserviços: Async**, 2019. Disponível em <https://medium.com/@rafaelgss/comunicacao-entre-microservices-async-ed3e5897ba6> Acessado em 27 ago. 2021.

HALILI, Emily H. **Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites**. Birmingham, 2008. 121p.

Introduction to gRPC. 2020. Disponível em <https://grpc.io/docs/what-is-grpc/introduction>. Acessado em 17 fev. 2021.

KHARE, Kartik. **JSON vs Protocol Buffers vs FlatBuffers**. 2018. Disponível em <https://codeburst.io/json-vs-protocol-buffers-vs-flatbuffers-a4247f8bda6f>. Acessado em 01 mar. 2021.

LEAL, Braulio G. **Avaliação de Desempenho de Sistemas**. 2022. Universidade Federal do Vale do São Francisco Avenida Antônio Carlos Magalhães, 510 Santo Antônio Juazeiro/BA – Brasil.

MENDONÇA, Arthur do Rego Barros. **Avaliação Experimental de uma Arquitetura de Microsserviços para o Gerenciamento de Notas Fiscais Eletrônicas**. Universidade Federal de Pernambuco Centro de Informática, 2022.

NASCIMENTO, Rodrigo. **O que é dashboard?.** 2017. Disponível em <http://marketingpordados.com/analise-de-dados/o-que-e-dashboard-%f0%9f%93%8a/> Acessado em 28 mar. 2021.

NEWMAN, Sam. **Building Microservices: Designing fine-grained systems**. Sebastopol, Ca, States Of America: O'reilly Media, Inc., 2015. 473 p.

NEWTON, James. **Comparar serviços gRPC com APIs HTTP**. 2019. Disponível em <https://docs.microsoft.com/pt-br/aspnet/core/grpc/comparison?view=aspnetcore-5.0#performance> Acessado em 01 mar. 2021.

Protobuf — What & Why?. 2021. Disponível em <medium.com/nerd-for-tech/protobuf-what-why-fcb324a64564>. Acessado em 04 set. 2022.

REZVINA, Sasha. **5 Reasons to Use Protocol Buffers Instead of JSON for Your Next Service**. 2015. Disponível em <https://codeclimate.com/blog/choose-protocol-buffers/>. Acesso 14 fev. de 2021.

RICHARDSON, Chris. **A pattern language for microservices** (2014). Disponível em: <http://microservices.io/patterns/>. Acesso em 24 ago. de 2020.

RICHARDSON, Chris. **Microservices Pattern.** Shelter Island, Ny 11964: Manning Shelter Island, 2019. 490 p.

RYAN, Louis. **gRPC Motivation and Design Principles.** 2015. Disponível em <https://grpc.io/blog/principles/>. Acessado em 17 fev. 2021.

SALES, André Barros de. **MEDIDAS DE LATÊNCIA EM AMBIENTES DE PROCESSAMENTO DE ALTO DESEMPENHO**. Universidade Federal de Santa Catarina Programa de Pós-Graduação Em Ciência da Computação, 2000.

SANTOS, Thiago Aquino dos. **Um estudo do stackoverflow sobre perguntas e soluções sobre microsserviços.** Recife, 2019. 14p.

SAUDATE, Alexandre. **REST: Construa Api's inteligentes de maneira simples.** São Paulo, Casa do Código, 2014. 101 p.

TORRE, Cesar de La. **.NET Microservices: Architecture for Containerized .NET Applications**. Redmond, Washington, 2022 35p.