UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

JANAINA LUDWIG

UMA ANÁLISE COMPARATIVA ENTRE GRPC E REST PARA A INTEGRAÇÃO DE SERVIÇOS WEB

DOIS VIZINHOS

2022

JANAINA LUDWIG

UMA ANÁLISE COMPARATIVA ENTRE GRPC E REST PARA A INTEGRAÇÃO DE SERVIÇOS WEB

A comparative analisys between gRPC and REST for Web services integration

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Software do Curso de Bacharelado em Engenharia de Software da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Rafael Alves Paes de Oliveira

DOIS VIZINHOS
2022

JANAINA LUDWIG

UMA ANÁLISE COMPARATIVA ENTRE GRPC E REST PARA A INTEGRAÇÃO DE SERVIÇOS WEB

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Software do Curso de Bacharelado em Engenharia de Software da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 24/junho/2022

Rafael Alves Paes de Oliveira
doutorado
Universidade Tecnológica Federal do Paraná

Francisco Carlos Monteiro Souza
doutorado
Universidade Tecnológica Federal do Paraná

Gustavo Jansen de Souza Santos
doutorado
Universidade Tecnológica Federal do Paraná

DOIS VIZINHOS
2022

Dedico este trabalho aos meus pais.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Sarajane de Bortoli Ludwig e Calisto Antônio Ludwig, que sempre me apoiaram e incentivaram meus estudos, desde minha infância e por todos estes anos. Também agradeço ao meu irmão André, minhas avós Luísa e Hilda, e meu namorado Daniel, por também me apoiarem e pela confiança depositada em mim sempre.

Ao meu orientador, Prof Dr. Rafael Paes de Oliveira, agradeço pela confiança, e por ter auxiliado e contribuído com este trabalho sempre que necessário. Também agradeço à banca, Prof. Dr. Gustavo Jansen de Souza Santos e Prof. Dr. Francisco Carlos Monteiro Souza por todas as contribuições e sugestões, me permitindo melhorar o trabalho.

Agradeço à UTFPR e todos os professores que me acompanharam nesta longa jornada na Universidade por todos os conhecimentos passados, que me auxiliaram a evoluir como pessoa e profissionalmente.

Agradeço também aos meus amigos, que estiveram do meu lado durante os momentos difíceis e felizes, compartilhando os desafios da Universidade, e tornando esta jornada mais fácil e divertida.

RESUMO

Com o surgimento de arquiteturas de software como microsserviços, surge a necessidade de Integração de Software, a fim de que os sistemas possam trocar dados entre si de modo eficiente e performático. E quando o arquiteto de software planeja o modelo de integração, é necessário levar em conta alguns fatores no contexto daquele software. Uma das maneiras de realizar integração é por meio do modelo *request/response*, e entre as tecnologias existentes estão REST (*REpresentational State Transfer*), que é muito utilizada, e gRPC (*Remote Procedure Call*), que é uma tecnologia relativamente nova, mas promissora. Diante disso, este estudo fornece informações no formato de uma comparação entre REST e gRPC para auxiliar nesta escolha. Para isso foi feito um estudo de caso, no qual foram construídos dois serviços utilizando ambas as tecnologias, além de pesquisas na literatura técnica, comparando estes dois estilos arquiteturais através da abordagem *Goal Question Metric*. Os resultados encontrados indicam que gRPC possui melhor performance em ambientes de alta demanda e REST possui melhor adequação a integrações externas entre serviços. As informações obtidas foram sumarizadas em uma comparação fornecendo insumos que auxiliam na decisão de qual tecnologia utilizar, e assim, contribuindo com a área de Integração de Software.

Palavras-chave: rest; grpc; comparação; integração web; serviço web.

ABSTRACT

With the emergence of software architectures such as microservices, there is a need for Software Integration, so that systems can exchange data with each other efficiently and performatively. And when the software architect plans the integration model, it is necessary to take into account some factors in the context of that software. One of the ways to integrate is through the request/response model, and among the existing technologies are REST, which is widely used, and gRPC, which is a relatively new but promising technology. Therefore, this study provides information by comparing REST and gRPC to assist in this choice. We conducted a case study, in which two services were built using both technologies and we researched the technical literature, comparing these two architectural styles through the Goal Question Metric approach. The outcome indicates that GRPC performs better in high-demand environments, and REST has better external integrations between services. The collected information was summarized in a comparison that assists the developer in the decision of which technology to use, thus contributing to the area of Software Integration.

Keywords: rest; grpc; comparision; web integration; web service.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTAGEM DE CÓDIGOS FONTE

SUMÁRIO

1 INTRODUÇÃO

Software está entre as as tecnologias mais importantes no mundo e a cada ano os requisitos dos sistemas ficam mais complexos. No século XXI, a sociedade como um todo depende da tecnologia (pessoas físicas, empresas e até governos), e utiliza software para tomadas de decisões e operação de suas atividades. Portanto, a construção de aplicações deve ter uma boa qualidade, e os sistemas de software devem ser projetados por meio de disciplinas recomendadas pela Engenharia de Software (PRESSMAN; MAXIM, 2019).

De acordo com Pressman e Maxim, existem sete categorias de software e uma delas são os aplicativos Web/móveis, o que contempla sistemas para navegadores, computação em nuvem, computação baseada em serviços e aplicativos móveis. Como explicado por Sommerville, no começo da Web, em 1990, tornou-se possível acessar informações fora de suas próprias organizações por meio de navegadores Web, porém era difícil acessar tais informações entre os próprios softwares. Então, surgiram serviços Web que permitiam manipular ou buscar os dados por meio da Web. Para isso as organizações podiam publicar suas interfaces de serviços definindo como seus dados podiam ser acessados ou usados, e assim, começaram as integrações de sistema de software.

Neste contexto, é importante analisar a arquitetura adotada para a construção dos sistemas. Arquitetura de software é o conjunto de estruturas necessárias para um sistema, que compreendem elementos do software, suas relações e propriedades. As estruturas arquiteturais não são fixas, ou seja, não seguem sempre o mesmo padrão para todo projeto, mas dependem do arquiteto de software analisar o que faz sentido para cada contexto, de forma a entregar os atributos de qualidade importantes do sistema (BASS; CLEMENTS; KAZMAN, 2021). No cenário de desenvolvimento Web, essas definições também se aplicam, e entre os possíveis modelos estão, por exemplo, monólitos, arquitetura orientada a serviços (SOA - do inglês, *Service-oriented Architecture*), e microsserviços. A principal diferença entre eles é que em monólitos toda a lógica de negócio e processamento são implantados em um único sistema (SOMMERVILLE, 2020), apresentando um maior acoplamento, mas em SOA e microsserviços a aplicação é dividida em serviços diferentes, tendo a necessidade de integração entre eles (JOSUTTIS, 2007; XIAO; WIJEGUNARATNE; QIANG, 2016)

Segundo Sommerville, em torno de 1990, o modelo tradicional para construção das aplicações era uma estrutura monolítica. O autor recomenda que protótipos ou mesmo a primeira versão das aplicações sejam construídas por meio de monólitos. Além disso, também aponta que desenvolvedores de microsserviços recomendam que a melhor maneira de identificar a necessidade de serviços separados é iniciando com monólitos, e depois refatorar em serviços menores.

Um monólito é uma escolha excelente para muitas empresas. No entanto, conforme a empresa e o sistema crescem, torna-se difícil gerenciar este desenvolvimento, pois vários desenvolvedores alteram o mesmo código, tentando implantar novas funcionalidades ou tendo que atrasá-las pela dependência das demais implantações. Outro problema é a falta de controle

de quem é dono de cada informação ou quem pode alterá-la (NEWMAN, 2019). Quando estes problemas surgem, é importante pensar na refatoração do software em diferentes serviços, mas com isso surgem vários desafios e escolhas, e uma delas é como fazer Integração de Software entre os diferentes serviços, o que deve ser planejado desde o começo da construção destes serviços.

As decisões arquiteturais devem levar em conta diversos fatores, como o contexto no qual o software está inserido e seus requisitos, como será a divisão em serviços, e também os requisitos não funcionais, como performance, escalabilidade, extensibilidade, entre outros. E nesse sentido, o método de integração de software pode causar muito impacto, positivamente ou negativamente. Por exemplo, se o serviço será disponibilizado publicamente, será necessário dar suporte às várias linguagens, mas se for um sistema que precisa de muita performance, talvez a mesma escolha não seja adequada. Portanto, é indispensável que o arquiteto possua informações sobre as diferentes tecnologias, a fim de fazer a escolha ideal para seu contexto em específico.

1.1 Problema de Pesquisa

No cenário atual, as empresas de tecnologia crescem cada vez mais, bem como seus sistemas. Assim, utilizam-se cada vez mais as arquiteturas de microsserviços, que precisam de decisões relacionadas à Integração de Software. Neste sentido, Newman classifica as decisões em um espectro de reversíveis e irreversíveis, e no contexto de software, é possível reverter uma parte delas, porém é importante considerar o custo da reversão caso se torne necessário. Por exemplo, a API (do inglês, *Application Programming Interface*) pública dos serviços poderia ser trocada, mas com um custo alto.

A Integração de Software é necessária, pois mesmo que cada serviço seja responsável por um domínio específico, é necessário compartilhar dados. Por exemplo, em um *e-commerce* poderia existir um serviço para gerenciamento de produtos e um serviço de pagamentos, o qual precisaria das informações dos produtos para poder relacionar com os pedidos. Por isso a integração é tão importante, e se bem feita, os serviços terão autonomia e poderão ser alterados e lançados de forma independente, mas se a integração não for bem pensada, os serviços serão acoplados e difíceis de gerenciar, o que reforça a importância de ter informações sobre as formas de fazer integração de serviços.

Existem diferentes maneiras para integrar serviços, e uma das primeiras decisões arquiteturais necessárias é a forma de comunicação, na qual o arquiteto de software pode escolher entre o modelo *request/response* e o modelo baseado em eventos. No modelo *request/response*, que é o foco deste estudo, um dos serviços faz uma requisição e espera por uma resposta do outro serviço, de maneira síncrona. Já no modelo baseado em eventos um dos serviços emite um evento, e os demais serviços executam alguma ação quando ele ocorrer, de forma assíncrona (NEWMAN, 2021).

Outra decisão é qual tecnologia utilizar, e especificamente no modelo de comunicação *request/response*, que é o foco do presente estudo, existem algumas opções como SOAP (do inglês, *Simple Object Access Protocol*), REST (do inglês, *Representational State Transfer*) e RPC (do inglês, *Remote Procedure Call*). É visto como uma evolução utilizar REST em comparação com RPC, apesar de ambos terem suas vantagens e desvantagens (VINOSKI, 2008; FENG; SHEN; FAN, 2009), principalmente considerando os métodos tradicionais para implementar RPC. Isso porque, por exemplo, algumas implementações de RPC são muito ligadas a uma única plataforma, como Java RMI (do inglês, *Remote Method Invocation*), o que impede a heterogeneidade dos diferentes serviços (NEWMAN, 2021). Outro motivo é a performance, já que um formato muito usado para as mensagens REST é JSON, que é um formato mais leve que XML, formato usado no SOAP (FENG; SHEN; FAN, 2009).

No entanto, quando considerado gRPC, que é um framework para RPC, há algumas mudanças nestes critérios, já que esta tecnologia é suportada por diferentes plataformas, faz uso de *Protocol Buffers* (formato binário para envio de mensagens), e estudos indicam que o uso de gRPC resulta em melhor performance (PERDANAPUTRA; KISTIJANTORO, 2020). O protocolo gRPC, criado pelo Google, é uma tecnologia relativamente nova e seu uso tem crescido como protocolo de comunicação (PERDANAPUTRA; KISTIJANTORO, 2020).

Contudo, ainda não existem muitas comparações entre REST e gRPC na literatura. E como abordado anteriormente, em uma migração para microsserviços as decisões devem ser bem pensadas, já que algumas delas são irreversíveis ou tem alto custo de mudança. Um dos aspectos que Newman cita como próximo do espectro irreversível é a mudança na API pública dos serviços. Uma API é a interface exposta por uma aplicação, ou seja, o conjunto de assinaturas exportadas por uma biblioteca/sistema e disponibilizadas aos usuários (BOURQUE; FAIRLEY, 2014), que pode ser local, mas também remota por meio da rede, permitindo integração com outros serviços (NEWMAN, 2021). Assim, mudá-la traria muito impacto aos sistemas e clientes, e por isso são necessárias informações a fim de tomar a decisão correta minimizando mudanças posteriores que sejam custosas.

Neste sentido, ao escolher entre REST - que é consolidado - ou gRPC - tecnologia nova e em crescimento - é importante notar que a principal consequência dessa escolha é o fato que não será simples ou rápido alterar a aplicação para o outro formato. Apesar de ambas utilizarem o protocolo HTTP (do inglês, *Hypertext Transfer Protocol*), a forma de consumo dos serviços é diferente, então não somente o código do servidor terá que ser alterado, mas também dos clientes, bem como os testes automatizados se existirem. Também não é simples porque podem existir muitos serviços consumindo aquela API, e fica ainda mais crítico quando os serviços são consumidos por softwares externos. Além disso, todo sistema possui características únicas, como necessidade de alta taxa de transferência ou flexibilidade para mudanças por exemplo, por isso é importante considerar qual modelo se adequará mais a cada cenário utilizando-se também de critérios quantitativos. Dependendo do software e seus requisitos, será necessário priorizar

um desenvolvimento rápido, dar suporte à integração com sistemas externos, ou construir serviços que suportem alta demanda e lidem com muitos dados.

Portanto, o problema a ser abordado neste trabalho é a falta de informações para auxiliar (de modo fundamentado e com evidências científicas) o arquiteto na escolha entre REST e gRPC como métodos para Integração de Software.

1.2 Objetivos

Este trabalho de conclusão de curso visa contribuir com a Engenharia de Software, especificamente na subárea de Integração de Software, por meio de uma comparação entre tecnologias utilizadas para integrar serviços Web. Nas seções a seguir são detalhados os objetivos gerais e específicos do projeto.

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é fornecer insumos para auxiliar na tomada de decisão de qual abordagem utilizar entre os métodos REST e gRPC para a integração de serviços Web, quando utilizado o modelo *request/response*.

1.2.2 Objetivos Específicos

Com o intuito de apoiar o presente trabalho, e para alcançar o objetivo geral, foram definidos os seguintes objetivos específicos:

- Estabelecer um protocolo para comparação entre tecnologias para Integração de Software;
- Construir serviços usando as duas tecnologias para possibilitar uma comparação técnica entre REST e gRPC;
- Analisar em quais ambientes (*mobile*, comunicação entre serviços e *Web*) as abordagens são recomendadas;
- Analisar como as duas abordagens se comportam durante períodos de alta demanda dos serviços; e
- Analisar prós e contras das duas abordagens em uma comparação.

1.2.3 Estrutura do Trabalho

No Capítulo 2 é feita a fundamentação teórica dos assuntos abordados no estudo. No Capítulo **??** são apresentados os estudos relacionados e algumas considerações sobre eles. A seguir, no Capítulo 3, é abordada a metodologia para o desenvolvimento do estudo, no Capítulo 4 são abordados os estudos empíricos realizados e no Capítulo 5 são apresentados os resultados e discussões. Por fim, o Capítulo 6 apresenta as conclusões deste estudo.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção são apresentados e explicados os conceitos necessários para entendimento do trabalho. Entre eles, está a Engenharia de Software relacionada com desenvolvimento Web, Arquitetura de Software no contexto de diferentes serviços, Integração de Software, além das duas tecnologias de foco do trabalho: gRPC e REST.

2.1 Engenharia de Software

Segundo Sommerville, a Engenharia de Software é a disciplina que se preocupa com todos os aspectos de produção de software desde seu planejamento inicial até a etapa de manutenção. O autor também aponta que os engenheiros aplicam teorias, métodos e ferramentas e buscam novas soluções se necessário. Eles devem se preocupar não somente com a parte técnica, mas também com gerenciamento e desenvolvimento de ferramentas para apoiar a construção de software.

Existem diferentes tipos de software, mas algumas questões são comuns a todos eles: *(i)* heterogeneidade, ou seja, é comum ter que interagir com outros softwares, legados ou não, que podem ser de plataformas distintas ou mesmo linguagens diferentes; *(ii)* mudanças de requisitos no negócio ou mudanças na sociedade, fazendo com que, para acompanhar as mudanças, softwares precisem ser desenvolvidos rapidamente para entregar valor aos clientes; *(iii)* segurança e confiança, necessários pois software é usado por toda a sociedade e é de extrema importância que as informações estejam seguras; e por fim, a *(iv)* escalabilidade, visto que alguns sistemas possuem uma escala muito pequena, como sistemas embarcados, ou grande, como sistemas na Internet usados por uma comunidade global (SOMMERVILLE, 2016).

Ainda de acordo com Sommerville, a Internet e a *World Wide Web* revolucionaram a forma de construir sistemas de software, pois em vez de desenvolver um sistema e instalá-lo no computador de cada usuário, passou-se a instalar o sistema em um servidor Web e acessá-lo por meio dos navegadores. Assim, diminuíram-se os custos de instalação, além de facilitar atualizações, manutenções corretivas e manutenções preventivas. Isso faz com que cada vez mais sejam construídos sistemas de software baseados em serviços. Ao contrário dos monólitos, os sistemas são distribuídos, chegando a rodar em servidores por todo o mundo, e há mais reuso de componentes ou outros sistemas de software. Portanto, sistemas Web ficam cada vez maiores, complexos e precisam lidar com problemas de complexidade e escala.

Neste contexto, são necessárias técnicas e planejamentos da Engenharia de Software, a fim de possibilitar que os sistemas atendam seus requisitos e tenham qualidade, e para isso a área de Arquitetura de Software tem muita importância. Na seção a seguir são apresentados alguns conceitos de Arquitetura de Software para contextualizar a necessidade de Integração de Software.

2.2 Arquitetura de Software

Arquitetura de Software é uma disciplina dentro da Engenharia de Software, e seu princípio básico é que um software deve atender às metas do negócio para o qual o software é projetado, sendo a arquitetura uma ponte para torná-las concretas em um software. Uma arquitetura pode ser projetada, analisada e documentada, é o conjunto de estruturas necessárias em um sistema, e compreende os elementos do software, suas relações e propriedades (BASS; CLEMENTS; KAZMAN, 2021).

Em arquitetura existem dois termos importantes de diferenciar: padrão arquitetural e estilo arquitetural. Um padrão arquitetural "expressa um esquema de organização estrutural fundamental para sistemas de software" (BUSCHMANN *et al.*, 1996 apud CLEMENTS *et al.*, 2010). No caso de um padrão, o contexto e o problema importam, ou seja, é um padrão recorrente que funciona em determinados contextos. Já um estilo arquitetural é mais voltado a uma especialização de um elemento e suas relações sem tanto foco no contexto do problema, mas ambos podem ser utilizados em conjunto e facilitam a comunicação a respeito da arquitetura do software (CLEMENTS *et al.*, 2010).

De acordo com Clements *et al.*, para grande parte dos sistemas, os atributos de qualidade são tão importantes quanto computar o resultado correto. Exemplos de atributos são performance, segurança e confiabilidade, e as decisões arquiteturais auxiliam a atingir estes requisitos de qualidade, bem como comportamentais. Não existe uma definição exata de até qual ponto vai a arquitetura de um sistema, no entanto, algumas das decisões arquiteturais podem ser bem detalhadas. Os autores explicam que, por exemplo, o tipo de interação entre serviços, ou formato dos dados, podem ser consideradas decisões arquiteturais se forem necessárias para atingir os requisitos, ou seja, depende de cada contexto.

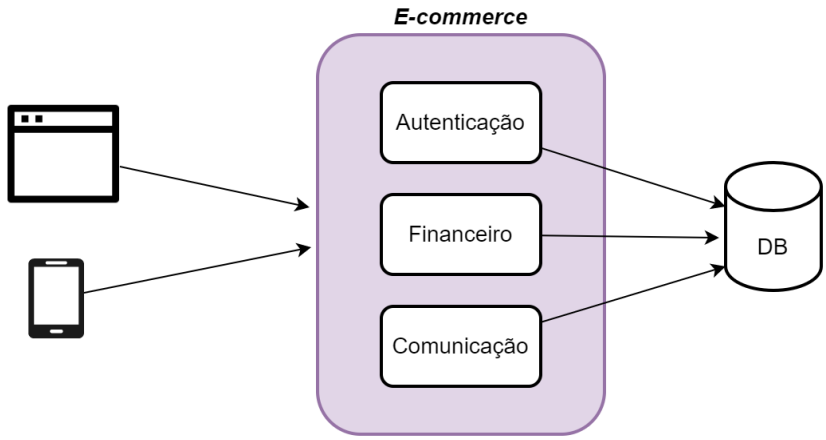
Um atributo de qualidade é uma propriedade do sistema que mede se o software possui qualidade além dos requisitos funcionais, porque não basta ter as funcionalidades se o software for muito lento ou difícil de manter, por exemplo. E as estruturas do software determinam se a arquitetura utilizada vai suportar os atributos de qualidade necessários. Um dos vários atributos de qualidade existentes é a integrabilidade, ou seja, define se é possível ou difícil integrar este software. A arquitetura de microsserviços, por exemplo, remove parcialmente o acoplamento, e os serviços só interagem por meio de suas interfaces (BASS; CLEMENTS; KAZMAN, 2021), porém traz a necessidade de fazer uma integração pela rede. Este atributo de qualidade é um exemplo a se ponderar ao definir a arquitetura dos sistemas, e dependendo do contexto, faz parte das decisões arquiteturais necessárias.

A seguir são apresentados algumas das possíveis arquiteturas para Web: monólitos, SOA e microsserviços.

2.2.1 Monólitos

Segundo Newman um monólito é uma estrutura de projetos de software no qual há uma única unidade de implantação, ou seja, todas as funcionalidades do sistema são implantadas juntas, como demonstrado pela Figura 1. Os monólitos podem ser classificados em três tipos, de acordo com o autor:

Figura 1 – Exemplo da arquitetura monolítica



Fonte: Autoria Própria.

- **Monólito de processo único:** Neste tipo existe um único processo, que pode ou não ter várias instâncias. Há uma variação, que é o monólito modular, no qual cada módulo funciona independentemente, mas ainda assim é necessário fazer uma implantação com todos os módulos juntos;
- **Monólito distribuído:** Neste outro modelo, existem diferentes serviços, mas são acoplados e precisam ser implantados em conjunto por algum motivo. O autor explica que este tipo de monólito traz as desvantagens de sistemas distribuídos (como a implantação de vários serviços) e as desvantagens do monólitos (como o acoplamento);
- **Sistemas caixa-preta de terceiros:** Neste caso, é utilizado algum software criado por terceiros, no qual não é possível fazer alterações de código. São considerados monólitos quando são utilizados para alguma decomposição durante migrações de sistemas.

Apesar de ser uma arquitetura adequada para algumas empresas, existem alguns desafios, principalmente o acoplamento. E quanto mais o sistema crescer, mais difícil de gerenciar, pois os desenvolvedores podem começar a alterar códigos que influenciam outras partes do sistema, e fica difícil saber quem é dono de cada domínio do software (NEWMAN, 2019).

2.2.2 Arquitetura orientada a Serviços

Arquitetura orientada a Serviços (SOA, do inglês, *Service-oriented architecture*) é um estilo arquitetural no qual serviços diferentes podem ser incluídos nas aplicações. Estes serviços devem ter interfaces bem definidas e seguir padrões consolidados ao invés de tecnologias proprietárias, de forma a não trazer incompatibilidades (SOMMERVILLE, 2016). O SOA surgiu como uma maneira de lidar com os desafios de monólitos. Um serviço, neste caso, fica em um processo totalmente separado do sistema operacional e a comunicação entre eles acontece através da rede (NEWMAN, 2021).

Os padrões do SOA são baseados em XML (do inglês, *eXtensible Markup Language*), uma linguagem de marcação que permite estruturar dados por meio de identificadores, de forma legível para humanos e máquinas. Outro padrão é a utilização de SOAP (*Simple Object Access Protocol*) como mecanismo de troca de mensagens entre os serviços, utilizando XML. Além disso, para definir a interface dos serviços, utiliza-se WSDL, ou *Web Service Description Language*, uma linguagem para descrição de serviços Web. O WSDL define quais operações, parâmetros e tipos aquele serviço possui. Outro padrão importante é o WS-BPEL, uma linguagem para definir processos que envolvem vários serviços diferentes (SOMMERVILLE, 2016).

Todas estas definições auxiliam os desenvolvedores a ter um padrão claro. No entanto, como apontado por Sommerville, conforme os serviços são desenvolvidos, eles acabam sendo serviços de uma única função e com interfaces simples. Isso fez com que a ideia de ter muitas definições acabava tendo o efeito oposto, já que se tornaram padrões pesados por serem muito gerais ou ineficientes. O tempo necessário para criar e processar XML desacelera a comunicação entre os serviços, principalmente quando são serviços que precisam de alta taxa de transferência. Além disso, Newman argumenta que além do problema do método de comunicação, faltava uma orientação a respeito da granularidade ideal dos serviços, ou mesmo a orientação errada sobre quais partes do sistema deviam ser divididas.

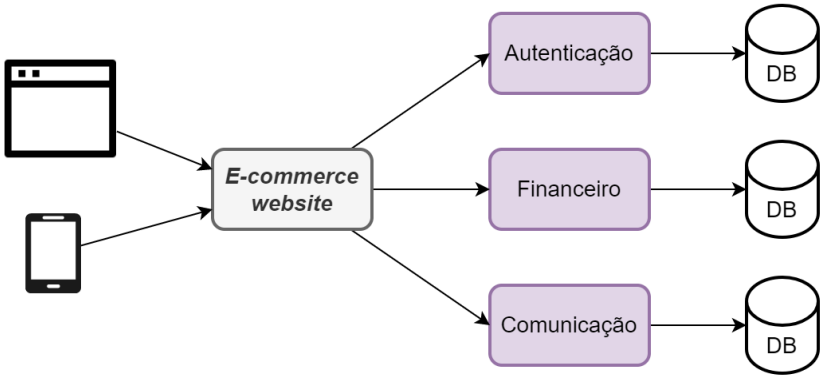
A partir desses problemas, surgiram outras alternativas: o estilo arquitetural REST (que será abordado nas próximas seções) como protocolo de comunicação mais leve que o SOAP (SOMMERVILLE, 2016), e em relação à arquitetura dos sistemas surgiu a abordagem de microsserviços, abordada na seção a seguir.

2.2.3 Microsserviços

Microsserviços são um tipo de arquitetura orientada a serviços (SOA), porém com algumas diferenças importantes das definições tradicionais de SOA. Em primeiro lugar, são serviços totalmente independentes entre si, e que podem ser implantados separadamente. Além disso, esta arquitetura é opinativa a respeito de como organizar os serviços: cada serviço deve ser responsável por um único domínio de negócio. São agnósticos em relação a quais tecnologias podem ser utilizadas, diferente do SOA tradicional. Outro ponto é que toda sua lógica interna é

abstraída - incluindo o banco de dados - e a comunicação é feita pela rede por meio de interfaces bem definidas (NEWMAN, 2019). A Figura 2 mostra um exemplo de microsserviços, que diferente da estrutura monolítica, nesta arquitetura cada serviço é independente, inclusive com cada serviço tendo seu próprio banco de dados.

Figura 2 – Exemplo da arquitetura de microsserviços



Fonte: Autoria Própria.

Neste estilo arquitetural, cada microsserviço é responsável por gerenciar seus próprios dados, e por isso, se os serviços forem muito específicos, haverá muita necessidade de replicação de dados, mas se forem organizados por meio de domínios lógicos de dados, utilizando abordagens como *Bounded Context*[1] por exemplo, a quantidade de replicação pode diminuir (SOMMERVILLE, 2020). Mas, nos dois casos, será necessário fazer Integração de Software.

Sommerville define algumas das principais características de microsserviços:

- **Independente:** Os serviços não devem ter dependências externas, mas sim ter seus próprios dados e suas próprias interfaces;
- **Leve:** A comunicação deve utilizar protocolos leves, a fim de não trazer sobrecargas para a comunicação entre os serviços;
- **Implementação independente:** Podem ser utilizadas diferentes linguagens de programação, e inclusive diferentes tipos de banco de dados;
- **Implantação independente:** Cada serviço tem seu próprio processo, deve ser possível lançar uma nova versão sem afetar os demais; e
- **Orientado a negócios:** Ao invés de fazer uma divisão de acordo com aspectos técnicos, deve-se levar em conta domínios de negócio.

Ao projetar uma arquitetura de microsserviços, uma das decisões arquiteturais consiste em como os serviços devem se comunicar. Os serviços se comunicam por meio de troca de mensagens, as quais incluem informações sobre a origem da mensagem, bem como os dados

[1] Padrão do *Domain-Driven Design* que divide os serviços de acordo com seu domínio lógico.

de *input* ou *output*. Estes dados devem ser estruturados de acordo com o protocolo do tipo de mensagem utilizada. Neste sentido, outras decisões devem ser tomadas: utilizar comunicação síncrona ou assíncrona; fazer uma comunicação direta ou por meio de um *middleware* ou *proxy*; e qual protocolo de mensagens utilizar. Em microsserviços, uma das abordagens mais utilizadas é o estilo arquitetural REST, com uso de mensagens no formato JSON (do inglês, *JavaScript Object Notation*) (SOMMERVILLE, 2020).

2.3 Integração de Software

Como abordado anteriormente, a arquitetura de microsserviços não define exatamente como deve ser feita a integração dos serviços, apesar de ter alguns protocolos mais comumente usados. Mas como essas decisões sempre dependem do sistema a ser construído, é importante entender sobre Integração de Software a fim de tomar a decisão ideal para cada contexto.

Alguns desafios são comuns para a atividade de integração. Hohpe *et al.* definem os seguintes desafios:

- **A rede é instável:** As integrações precisam enviar mensagens através da rede, e por isso podem ocorrer *delays* ou interrupções;
- **A rede é lenta:** É preciso considerar que transmitir dados através da rede é muito mais lento do que chamadas locais; assim, ao projetar os serviços, deve-se ter isto em mente, já que as soluções não podem seguir a mesma organização de um projeto local;
- **As aplicações possuem diferenças:** Uma solução de integração precisa considerar que muitas vezes serão integrados sistemas com linguagens de programação diferentes, ou formatos de dados diferentes; e
- **Mudança é inevitável:** As soluções devem ter baixo acoplamento e os serviços devem se adaptar a mudanças (que são naturais no ciclo de software) sem impactar os demais.

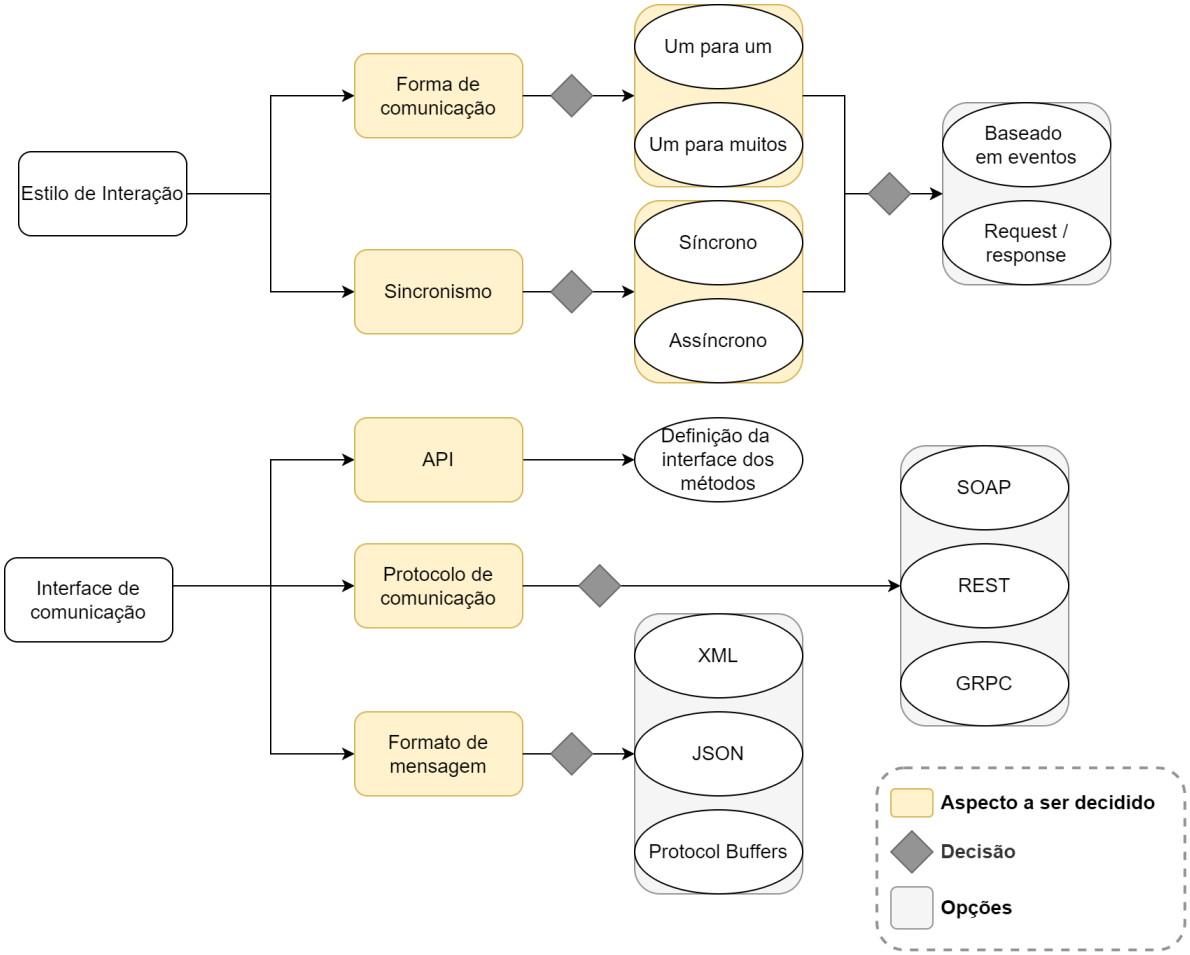
Para definir como fazer a integração dos serviços são necessárias algumas decisões, como demonstrado pela Figura 3, onde são mostrados alguns exemplos das alternativas existentes. Estes aspectos são explicados nas seções a seguir.

2.3.1 Estilo de interação

Entre os estilos para integrar serviços, de acordo com Newman, podem ser citados dois modelos importantes em relação à maneira de comunicação:

- **Baseado em eventos:** Naturalmente assíncrono, neste modelo um dos serviços emite um evento, e os demais serviços executam alguma ação quando ele ocorrer, como

Figura 3 – Decisões necessárias para integração de serviços Web



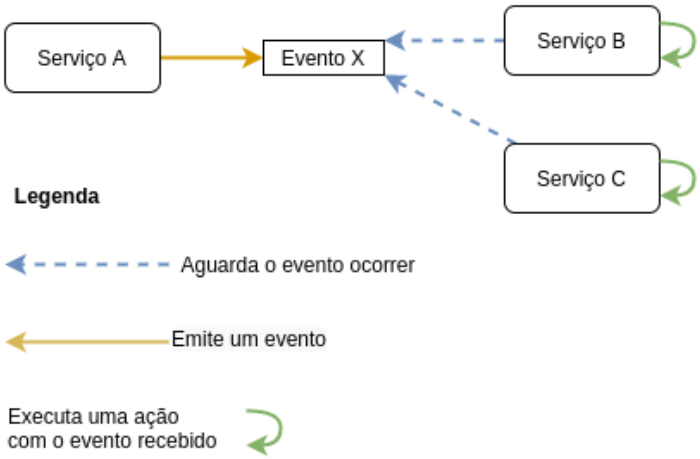
Fonte: Autoria Própria.

ilustrado pela Figura 4. Exemplos de tecnologias utilizadas para esta abordagem são RabbitMQ e Apache Kafka;

- ***Request/response*:** Neste modelo, um dos serviços faz uma requisição e espera por uma resposta, o que funciona de maneira síncrona, mas também pode resultar em modo assíncrono se, em vez de aguardar a ação ser finalizada, ocorrer por meio de *callbacks* quando a ação finalizar. Por exemplo, o serviço *A* faz uma requisição solicitando a geração de um documento para o serviço *B* e recebe uma resposta imediatamente informando que a solicitação foi recebida. Então, quando a geração do arquivo terminar, o serviço *B* faz outra requisição (o que é chamado de *callback*) para o serviço *A* informando a finalização e enviando o arquivo gerado.

Estes aspectos podem ser caracterizados em duas dimensões, de acordo com Richardson. A primeira delas é se a integração deve ser no modelo *um para um* ou *um para muitos*. No formato *um para um*, a requisição é processada por um único serviço. Um exemplo deste modelo é mostrado na Figura 5: uma requisição para validar permissão de acesso só precisa

Figura 4 – Modelo de comunicação baseado em eventos



Fonte: Autoria Própria.

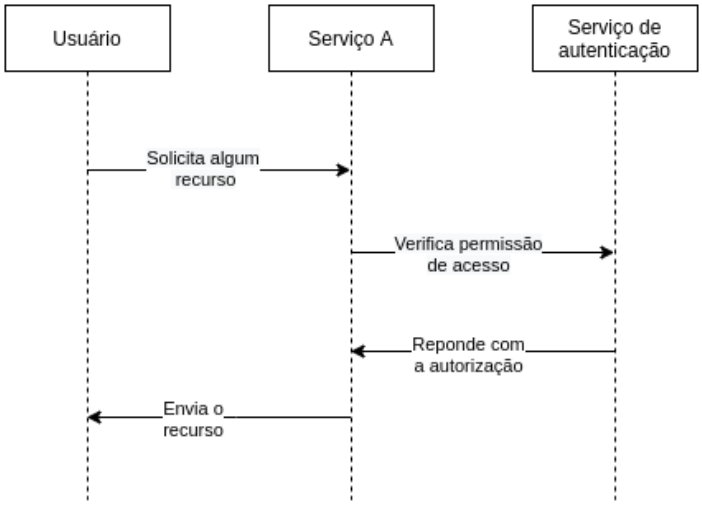
ser processada no serviço de autenticação. Já no formato *um para muitos*, cada requisição precisa ser processada por diferentes serviços, conforme a Figura 4, na qual é ilustrado o modelo de comunicação baseado em eventos, que é uma forma de comunicação *um para muitos*. Um exemplo deste último modelo seria em um *e-commerce*, quando um pagamento é finalizado é necessário que o serviço financeiro emita a nota fiscal, mas além disso o serviço de entregas precisará iniciar o envio do produto, e ainda, o serviço de comunicação com o cliente precisará enviar um email ao cliente com uma confirmação.

A segunda dimensão explicada pelo autor é a respeito de sincronismo. No modelo *síncrono*, o serviço que fez a requisição (cliente) espera por uma resposta, podendo inclusive ficar bloqueado enquanto não receber algum retorno. Já no modelo *assíncrono*, o cliente não fica bloqueado esperando por uma resposta, que não é enviada imediatamente. Por exemplo, em um processamento que leva vários segundos para ser concluído, haveria vantagem no modelo assíncrono, por não bloquear o cliente. Mas se for um cenário onde uma resposta é necessária imediatamente, será melhor o modelo síncrono, como exemplificado na Figura 5. Neste caso, o *Serviço A* precisa ficar bloqueado até verificar que o usuário está autenticado corretamente para poder responder com o recurso solicitado.

2.3.2 Definição das Interfaces

Além da escolha do estilo de interação, é preciso definir a interface de comunicação, ou seja, a API. As interfaces definem quais operações estão disponíveis, seus parâmetros e tipos de retorno, como um contrato. Em aplicações monolíticas, a interface é construída na própria linguagem de programação do software, então caso seja uma linguagem compilada, se alguma mudança incompatível na sua interface ocorrer, o código nem irá compilar. No entanto, ao definir a API de um serviço, se for feita alguma mudança incompatível nela, o erro só será disparado

Figura 5 – Exemplo de requisição síncrona, no modelo *um para um*



Fonte: Autoria Própria..

em tempo de execução (RICHARDSON, 2018). Por isso é recomendado evitar mudanças incompatíveis na API dos serviços (NEWMAN, 2021) ou versionar a API quando for necessário.

Para criar a API de um serviço é preciso definir o protocolo de comunicação. SOAP é um tipo de protocolo que faz uso de XML para definir as mensagens, que são transportadas pela rede. No entanto, conforme abordado anteriormente, ele não é o modelo ideal em alguns casos. Dois modelos importantes que utilizam o conceito de *resquest/response*, no contexto de microsserviços, são REST e gRPC. Dependendo do protocolo utilizado, o formato das mensagens já é definido, ou pode ser uma decisão a parte.

As mensagens (dados de entrada e saída da API), abstraem os dados internos da linguagem específica de cada serviço para que possam ser interpretados por linguagens diferentes. Essa conversão dos dados internos para externos é comumente chamado de serialização, tradução ou *marshalling* (empacotamento, em português). O formato de mensagem escolhido deve ser capaz de expressar diferentes estruturas de dados, deve ser um formato que diferentes tecnologias possam interpretar, além de ter boa performance (BASS; CLEMENTS; KAZMAN, 2021).

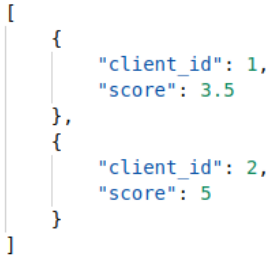
Os tipos de mensagem são divididos em duas categorias principais, como explicado por Richardson:

- **Formato baseado em texto:** a vantagem deste formato é que as mensagens são legíveis também para humanos, além de serem auto-descritivas. JSON e XML são exemplos, e em ambos, os valores ficam em propriedades nomeadas. Os consumidores das mensagens podem usar somente os valores necessários. A Figura 6 mostra um exemplo do formato JSON.

- **Formato binário:** as mensagens também podem ser no formato binário, e dois exemplos deste tipo são Apache Avro e *Protocol Buffers*, que definem o formato das men-

sagens de forma tipada. Neste modelo, o compilador gera um código responsável por serializar ou desserializar os dados de acordo com o formato definido pelo programador. Utilizando *Protocol Buffers*, por exemplo, será gerado um código específico para a linguagem sendo utilizada, de acordo com o formato definido das mensagens, sendo que este código gerado não deve ser alterado.

Figura 6 – Exemplo do formato de mensagens JSON



Fonte: Autoria Própria.

2.3.3 Métricas para análise de serviços

Para verificar se um método de integração atende às necessidades de algum software, podem ser utilizadas algumas métricas para avaliar performance:

- **Latência:** o tempo mínimo para ter alguma resposta. Esta medida torna-se relevante em chamadas remotas, pois é o tempo necessário para transitar as mensagens pela rede (FOWLER, 2012) ou por sistemas intermediários (BRAJESH, 2017);
- **Tempo de resposta:** o tempo que um sistema demora a processar e enviar a resposta para alguma chamada de fora do sistema, como uma requisição para a API no contexto de integração (FOWLER, 2012);
- ***Throughput* (taxa de transferência):** o quanto um sistema consegue fazer em um determinado período de tempo. O que é medido varia de acordo com o tipo do software e a complexidade da operação sendo realizada, mas em geral, para uma API considera-se o número de transações por segundo ou minuto (FOWLER, 2012);
- **Taxa de erro e sucesso:** mede o quanto das requisições foram executadas com sucesso ou erro durante uma carga no sistema (BRAJESH, 2017);
- **Carga:** indica sob quanto estresse um sistema está sendo executado. Uma forma de medir é quantas conexões ativas um servidor está processando. Uma alta carga pode impactar as demais métricas (FOWLER, 2012);

- **Sensibilidade de carga:** indica quanto o tempo de resposta varia se a carga aumentar. Também pode ser utilizado o termo *degradação* para indicar em uma comparação qual sistema é mais sensível a cargas (FOWLER, 2012); e

- **Apdex:** abreviação para *Application Performance Index* (Índice de Performance da Aplicação), é uma métrica criada para analisar a satisfação do usuário final, de ponta-a-ponta. Esta métrica varia de 0 a 1, sendo 0 uma satisfação baixa, e 1 a melhor satisfação possível. Existem algumas ferramentas para fazer este cálculo (MOLYNEAUX, 2014).

É importante pontuar que, segundo Fowler, a forma de medir performance depende do contexto. Por exemplo, para um determinado sistema o tempo de resposta pode ser o mais importante, mas para outro é a taxa de transferência. Além destas métricas, outro grupo que pode ser analisado é o consumo de recursos do servidor:

- **Consumo de CPU (do inglês, *Central Processing Unit*):** mede a capacidade do sistema durante uma carga. Esta métrica é importante pois, se mesmo com muitas requisições o consumo de CPU for baixo, é um indicativo de que o sistema consegue receber uma carga ainda mais alta (BRAJESH, 2017); e

- **Consumo de memória:** indica quanta memória está sendo utilizada pelo servidor. Se pouca memória estiver disponível, a performance das requisições poderá ser impactada (BRAJESH, 2017).

Estas métricas não devem ser analisadas somente para comparar duas formas de integração, mas também devem ser analisadas nos serviços em produção, para que se tenha informações sobre o estado da API e assim poder tomar decisões de melhorias ou mesmo saber se a API está de acordo com o esperado (BRAJESH, 2017). Para coletar as métricas são utilizadas ferramentas para observabilidade, a fim de ter um monitoramento dos serviços. Observabilidade significa coletar as métricas, erros, consumo de memória e CPU, entre outros, e existem ferramentas que automatizam a coleta e visualização destes dados. Exemplos de ferramentas de mercado são New Relic, Datadog e Sysdig, mas também existem alternativas *open-source*, como Prometheus (para monitoramento) e Grafana (para visualização dos dados), que podem ser utilizadas em conjunto (PAWLIKOWSKI, 2021).

A seguir são explicados REST e gRPC, bem como o formato de mensagens utilizados por ambos em mais detalhes.

2.4 REST

REST, ou *Representational State Transfer* (Transferência Representacional de Estado) é um estilo arquitetural baseado na transferência de recursos entre um servidor e o cliente. O estilo REST é um conjunto de convenções para comunicação de serviços através do protocolo

HTTP e que faz uso de XML e JSON para envio de mensagens. Quando um serviço segue as especificações REST, ele é chamado de API *RESTful* (SOMMERVILLE, 2020). O estilo de interação de REST é *request/response*, de *um para um*.

Antes de ir aos padrões definidos no REST, é preciso entender o protocolo HTTP, que é um protocolo de comunicação que envia dados estruturados por texto. As requisições HTTP possuem um método ou verbo HTTP (como GET, POST, PUT e DELETE), seu endereço (ou caminho), cabeçalho e corpo da requisição. A resposta HTTP possui um código de status (entre as faixas de 100 a 500), além do cabeçalho e corpo. E ambos possuem a versão do HTTP utilizada (SOMMERVILLE, 2020; RICHARDSON, 2018).

Os princípios mais importantes do estilo REST, segundo Sommerville, são:

- Uso dos verbos HTTP: as operações disponíveis nos serviços são acessadas através dos verbos `GET`, `POST`, `PUT` e `DELETE`;
- Serviços sem estado: os serviços não mantêm um estado interno, assim como microsserviços. Então por exemplo, não será mantida uma sessão autenticada, mas será necessário enviar os dados de autenticação em toda requisição;
- Endereçável por meio de URIs (do inglês, *Uniform Resource Identifier*): os recursos fornecidos por um serviço REST devem seguir endereços lógicos, bem como seus subrecursos, cada um tendo seu próprio conjunto de verbo mais o URI; e
- Mensagens XML ou JSON: as mensagens devem normalmente utilizar um destes formatos, sendo mais comum o uso de JSON, visto que pode ser processado de forma mais eficiente e assim reduzir a sobrecarga de uma requisição.

Além destes princípios, é definido o verbo HTTP para as operações básicas, também de acordo com Sommerville:

- Ler: para busca de um recurso, utiliza-se o verbo `GET`, que não pode criar ou atualizar recursos;
- Criar: para criação de um recurso, utiliza-se o verbo `POST`;
- Atualizar: para atualização de um recurso, utiliza-se o verbo `PUT`, que por sua vez, não deve ser utilizado para criação de recursos;
- Excluir: para exclusão de um recurso, utiliza-se o verbo `DELETE`. Pode ser feita exclusão física ou lógica, mas em ambos os casos aquele recurso torna-se inacessível através de seu `GET`.

Para diferenciar o retorno das requisições são utilizados os status HTTP e cada faixa possui um significado diferente. A faixa de 100 a 199 é relacionada a informações do protocolo de comunicação; De 200 a 299 indica que a operação obteve sucesso; a faixa de 300 a 399

é relacionada a redirecionamentos; de 400 a 499 indica erro na requisição do cliente; e 500 a 599 indica algum erro no servidor. Por exemplo, o status 201 indica que um recurso foi criado; 202 significa que o servidor aceitou a requisição e vai iniciar um processamento assíncrono; 401 indica que o cliente não enviou credenciais de autorização corretas; e existem vários outros status que podem ser utilizados, cada um com seu significado seguindo as especificações do HTTP (MASSE, 2011).

Outro ponto importante são os cabeçalhos das requisições e respostas. Por exemplo, o cabeçalho *Content-Type* indica o formato da mensagem; *Content-Length* informa o tamanho do corpo da mensagem; *Last-Modified* indica a data da última atualização daquele recurso (MASSE, 2011). Mas além de alguns cabeçalhos padrão, também podem ser utilizados outros, como *Authorization*, para enviar credenciais de autenticação.

2.4.1 Especificações de uma API REST

Diferente de outros protocolos de comunicação, como SOAP, que usa WSDL para especificar sua interface, não existe uma IDL (do inglês, *interface definition language* - linguagem para definição de interface) padrão definida para o REST originalmente. Apesar disso, a comunidade criou algumas formas para especificar as interfaces, sendo a mais popular o Open API, que foi evoluído a partir do projeto *open-source* Swagger (RICHARDSON, 2018). Open API é uma forma de especificar uma API REST utilizando JSON ou YAML (que é outro formato para serialização de dados), na qual são definidos os métodos disponíveis, formas de autenticação, modelos de requisição e resposta, e todas as informações necessárias para consumir determinada API [2]. A partir desta definição é possível convertê-la para um formato de documentação visual, e existem diferentes ferramentas para isso, como o Swagger UI (SWAGGER, 2022).

2.4.2 Exemplo de uma API REST

Para exemplificar os conceitos apresentados sobre o estilo REST, foi construída uma API REST com três métodos, utilizando a linguagem de programação Go. O Algoritmo 1 mostra os *endpoints* criados em código. Para este exemplo, foi criado um serviço simples que gerencia registros de estudantes em um banco de dados.

Para consumir o serviço, foi utilizado o software Postman[3], no entanto, para fazer a integração com serviços deve-se utilizar um cliente HTTP em código e fazer as mesmas requisições mostradas a seguir. O primeiro método é utilizado para buscar a lista de estudantes, basta fazer uma chamada para o *endpoint* "*/students*" usando o verbo HTTP GET, como mostrado na Fi-

[2] Um exemplo de definição de API utilizando Open API pode ser encontrado na documentação oficial da ferramenta, através deste endereço: https://petstore.swagger.io/v2/swagger.json

[3] Software Postman disponível em https://www.postman.com/

Listagem 1 – Exemplo de *endpoints* no estilo REST utilizando a linguagem Go

```
1  package router
2
3  import (
4    "github.com/julienschmidt/httprouter"
5    "grpc-rest/api/handlers"
6    "net/http"
7  )
8
9  func Routes() http.Handler {
10   router := httprouter.New()
11
12   router.GET("/", handlers.Index)
13   router.GET("/students", handlers.GetStudents)
14   router.POST("/students", handlers.CreateStudent)
15   router.DELETE("/students/:id", handlers.DeleteStudent)
16
17   return router
18 }
```

Fonte: Autoria Própria.

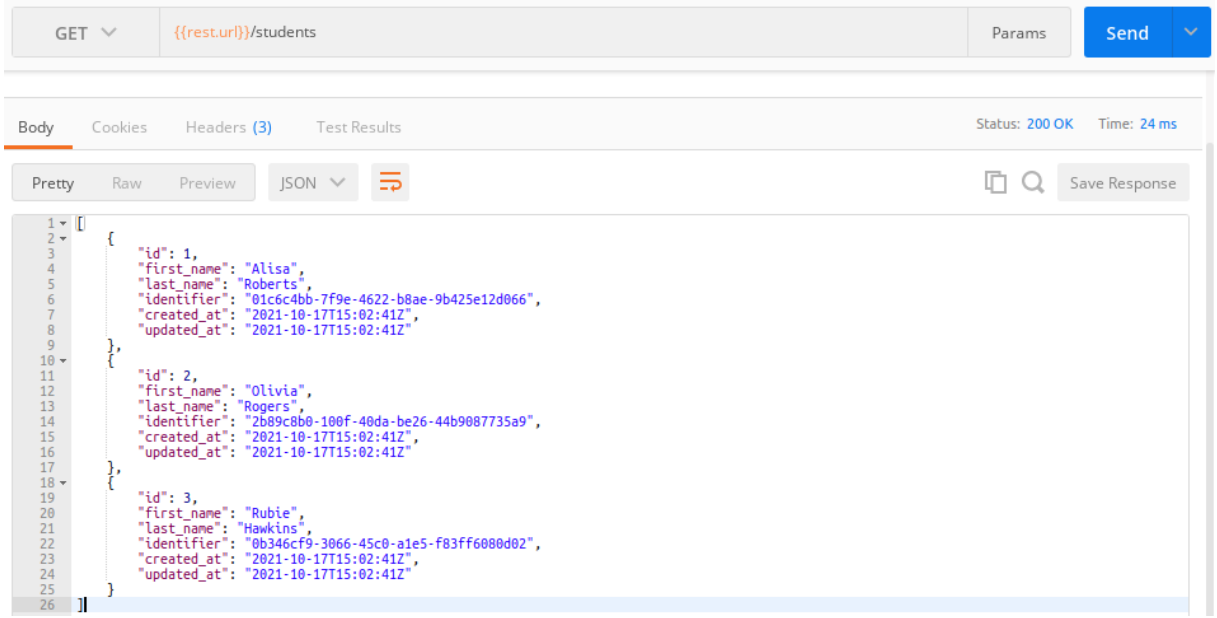
gura 7. Pode-se observar que o *status* retornado foi 200, indicando sucesso na requisição, e que o formato de mensagens utilizado foi JSON.

Na Figura 8 foi feita uma requisição ao serviço para excluir um estudante. Neste caso utiliza-se o *endpoint* "*students/:id*" com o verbo DELETE. Porém como a requisição solicitou a exclusão de um recurso inexistente, a API retornou o status 404, indicando que o recurso não existe. Na Figura 9 pode-se observar o cabeçalho da resposta, com informações a respeito do tamanho do corpo da mensagem, bem como o tipo de dados.

2.4.3 Considerações sobre a utilização de REST

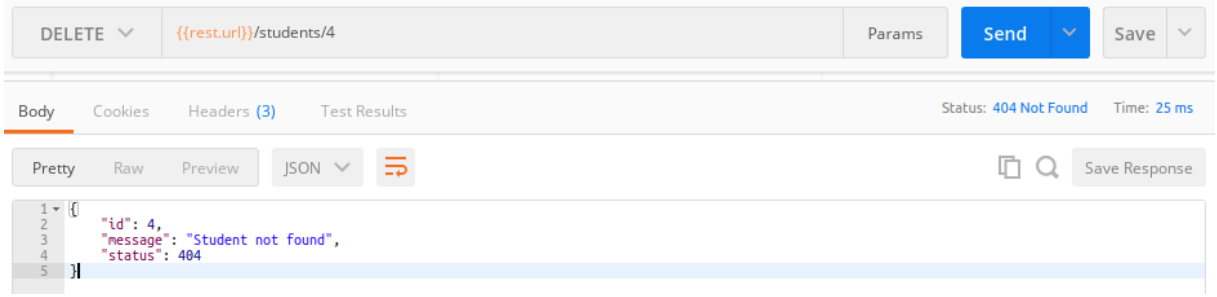
Existem alguns pontos negativos de utilizar REST, como não ser possível gerar automaticamente um cliente para consumir um serviço, apesar de existirem muitas bibliotecas para HTTP. Ou seja, em vez de simplesmente utilizar um método como se fosse uma chamada local, é preciso interagir diretamente com uma biblioteca HTTP, implementando as especificidades de

Figura 7 – Exemplo de uma requisição no estilo REST para buscar um recurso



Fonte: Autoria Própria.

Figura 8 – Exemplo de uma requisição no estilo REST para excluir um recurso

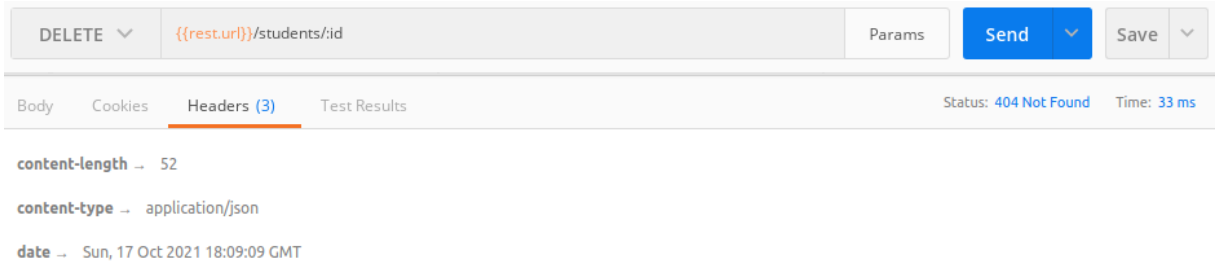


Fonte: Autoria Própria.

cada API. Assim, será necessário implementar a autenticação em cada chamada, por exemplo, bem como sempre conhecer e especificar um *endpoint*, além do corpo da requisição de acordo com o formato que a API espera receber.

Outro ponto negativo é que dependendo do *framework* utilizado, nem todos os verbos HTTP são suportados. Além disso, apesar da vantagem de JSON em relação ao XML, JSON é maior em número de bytes em comparação a protocolos binários, e é necessário mais trabalho para processar as mensagens em comparação com algumas tecnologias de RPC (NEWMAN, 2021). Apesar disso, REST é atualmente um padrão muito utilizado para APIs, como pelo Facebook (META, 2022) e OpenWeather (OPENWEATHER, 2022), além de ser simples de utilizar e testar. Mas existem alternativas, como gRPC, abordado na próxima seção.

Figura 9 – Exemplo de cabeçalho em uma resposta REST



Fonte: Autoria Própria.

2.5 GRPC

GRPC é um framework para chamadas de procedimento remoto (RPC), disponível para diferentes linguagens, faz uso de mensagens no formato binário e se comunica através da rede usando HTTP/2 no formato *request/response*, funcionando de maneira síncrona (RICHARDSON, 2018), e no estilo de interação *um para um*. As mensagens no formato binário fazem uso de *Protocol Buffers*, que também é utilizado como uma IDL para especificar a interface do serviço. *Protocol Buffers* é um mecanismo *open-source* para serializar dados estruturados, criado pelo Google (GRPC, 2018).

Ao criar um serviço usando gRPC, o desenvolvedor é forçado a utilizar o conceito de *API-First* (RICHARDSON, 2018), pois inicia-se definindo a interface do serviço usando *Protocol Buffers*, no qual são especificados os métodos disponíveis para chamadas remotas, seus parâmetros e formatos de mensagem, como mostrado no Algoritmo 2. A partir desta interface definida, o compilador gera alguns códigos. Estes códigos podem ser gerados para todas as linguagens suportadas, e servem para serializar e desserializar as mensagens. Outro código gerado será a interface do serviço na linguagem escolhida, que deverá ser implementada por alguma classe do sistema, e então utilizada para iniciar o servidor.

Além disso, também será gerado um cliente, que conecta no servidor de forma abstraída para utilizar os seus métodos, ou seja, para fazer as chamadas de procedimento remoto. Um aspecto importante é que podem ser gerados os clientes para diferentes linguagens, e, por exemplo, um cliente na linguagem Go consegue se conectar em um servidor que utiliza Java, não existindo dependência entre as linguagens. Dessa forma, os detalhes de baixo nível para a comunicação são abstraídos, e para os desenvolvedores a chamada dos métodos é simples, como se estivessem fazendo uma chamada de função local (INDRASIRI; KURUPPU, 2020).

Além do modelo tradicional de *request/response* (chamado de RPC unário), que é o mais parecido com o modelo REST, existem outras formas de comunicação no gRPC. É possível fazer *stream* do servidor (*Server-Streaming RPC*), no qual o servidor envia uma sequência de mensagens e só ao final envia o status da resposta. Também é possível o contrário, ou seja, *stream* a partir do cliente (*Client-Streaming RPC*), no qual são enviadas várias mensagens e é re-

cebida uma única resposta do servidor. Por fim, também é possível enviar *streams* bidirecionais (*Bidirectional-Streaming RPC*), no qual ambos o cliente e servidor podem enviar sequências de mensagens (INDRASIRI; KURUPPU, 2020).

Outro conceito importante são os canais (*channels*), que são conexões para um *endpoint* através do protocolo HTTP/2. A criação desta conexão é abstraída pelo cliente gRPC, e assim que o canal foi criado, poderá ser reutilizado para múltiplas chamadas ao servidor, trazendo vantagens de performance, já que não é necessário criar uma nova conexão HTTP em cada requisição para o mesmo servidor. As mensagens são enviadas através de streams do HTTP/2 e divididas em *frames* (INDRASIRI; KURUPPU, 2020).

2.5.1 *Protocol Buffers*

Protocol Buffers é o formato utilizado para envio das mensagens, além de especificar a interface do serviço. As mensagens e métodos ficam definidos em um arquivo com a extensão *.proto* (INDRASIRI; KURUPPU, 2020), como mostrado pelo Algoritmo 2. Na linha 8 é definido um serviço, e dentro dele, os seus métodos (listagem e criação). Para isso, é preciso especificar todos os parâmetros e retornos, de forma tipada. Algumas mensagens, neste exemplo, são *GetStudentsRequest* (linha 16) e *GetStudentsResponse* (linha 21), a qual possui uma lista da mensagem *Student* (linha 25). Esta, por sua vez, possui vários campos, como *id*, *first_name*, entre outros.

Cada campo é constituído por seu tipo e nome, além de um número único, que é utilizado para identificar aquele campo no formato binário no momento da serialização ou desserialização, então em vez de utilizar o formato de "chave/valor" no dado que é transportado pela rede, é utilizado somente este número único, a fim de otimizar o tamanho da mensagem. Por exemplo, na linha 27 do Algoritmo 2, o tipo do campo é *string*, o nome do campo é *first_name* e seu número único é 2. O número único não deve ser alterado para manter a compatibilidade da mensagem, justamente por ser a forma de mapear os campos na mensagem binária. Outro conceito importante é que cada campo pode ser singular (que é o padrão) ou repetido usando a palavra chave *repeated*, como na linha 22, ou seja, representando uma lista de valores assim como um *array* no formato JSON (GOOGLE, 2021).

Como pode-se observar no exemplo, todos os campos são tipados. Isso traz vantagens pois a interface dos serviços é bem definida e clara, o que torna os serviços gRPC mais estáveis. E é a partir desta definição nos arquivos *.proto* que o compilador de *Protocol Buffers* gera o código para consumir os serviços e interpretar as mensagens (INDRASIRI; KURUPPU, 2020).

Listagem 2 – Exemplo de definição de um serviço usando *Protocol Buffers*

```
syntax="proto3";

import "google/protobuf/timestamp.proto";

package grpc;
option go_package = "./proto";

service StudentsService {
  rpc GetStudents (GetStudentsRequest) returns (GetStudentsResponse) {}
  rpc CreateStudent (CreateStudentRequest) returns (Student) {}
}

message GetStudentsRequest {
}

message CreateStudentRequest {
  string first_name = 1;
  string last_name = 2;
}

message GetStudentsResponse {
  repeated Student students = 1;
}

message Student {
  int64 id = 1;
  string first_name = 2;
  string last_name = 3;
  string identifier = 4;
  google.protobuf.Timestamp created_at = 5;
  google.protobuf.Timestamp updated_at = 6;
}
```

Fonte: Autoria Própria.

2.5.2 Metadados

Como abordado anteriormente, no estilo REST é possível enviar cabeçalhos nas requisições e um conceito parecido com esta ideia no gRPC são os metadados, que permitem enviar informações que não são diretamente relacionadas com o domínio das chamadas de procedimento remoto. Assim, não faria sentido ser um argumento dos próprios métodos. Através dos metadados é possível compartilhar informações entre cliente e servidor, na forma de chave-valor. Um caso comum de uso de metadados é para transmitir informações de autenticação, que é essencial para construir uma comunicação segura entre serviços, independente do protocolo utilizado (INDRASIRI; KURUPPU, 2020).

2.5.3 Tratamento de erros

Assim como o estilo REST trata erros através dos códigos de status do HTTP, também é possível tratar erros usando gRPC. No entanto, isso é feito através de códigos específicos para gRPC. Além do código do erro, é possível enviar uma mensagem de erro, no formato de *string*, que é opcional e pode ser usada para detalhar o erro (GRPC, 2018).

Em caso de sucesso na requisição, é recebido um código *OK*, e existem outros códigos para variados erros, como cancelamento, ou parâmetros errados na requisição, por exemplo (INDRASIRI; KURUPPU, 2020). A documentação oficial do gRPC (GRPC, 2018) divide os possíveis erros em algumas categorias, como mostrado abaixo, com alguns exemplos de possíveis códigos:

- Erros gerais:
  - cancelamento pelo cliente (*GRPC_STATUS_CANCELLED*);
  - método não implementado (*GRPC_STATUS_UNIMPLEMENTED*);
  - servidor não disponível (*GRPC_STATUS_UNAVAILABLE*).
- Falhas de rede:
  - prazo final da requisição atingido (*GRPC_STATUS_DEADLINE_EXCEEDED*).
- Erros de protocolo:
  - falha na autenticação (*GRPC_STATUS_UNAUTHENTICATED*);
  - erro ao ler *protocol buffers* (*GRPC_STATUS_INTERNAL*).

Dessa forma é possível tratar possíveis erros na aplicação, pois na integração de serviços é possível, e até mesmo provável que ocorram erros. Este tratamento padrão de erros funciona para todas as linguagens que gRPC tem suporte, porém existem alternativas para detalhar ainda mais os erros, mas que não são suportadas por todas as linguagens (GRPC, 2018).

2.5.4 Exemplo de uma API gRPC

Para exemplificar uma API utilizando gRPC, foi construído o mesmo serviço do exemplo de REST, mas em gRPC e também utilizando a linguagem Go. Primeiro, foi definida a interface usando *Protocol Buffers*, como mostrado no Algoritmo 2. Em seguida, utilizando o compilador de *Protocol Buffers* e as bibliotecas necessárias para a linguagem Go, são gerados dois arquivos, com o cliente gRPC e o esqueleto do servidor, além do mecanismo necessário para serializar e desserializar as mensagens. Estes arquivos gerados são na linguagem utilizada no projeto e não devem ser alterados manualmente.

O próximo passo é implementar todos os métodos do servidor, o que varia de acordo com cada linguagem. Na linguagem Go, é necessário implementar a interface gerada pelo compilador, registrar os serviços concretos e então iniciar o servidor. O Algoritmo Listagem 3 mostra parte da implementação, na qual são recebidos os parâmetros definidos no *Protocol Buffer* e retornados a mensagem e o erro se ocorrer, exatamente de acordo com a definição da interface. A linha 11 cria a estrutura para o serviço e nas linhas 19 e 34 são implementados os dois métodos definidos nas linhas 9 e 10 do Algoritmo 2. Por fim, para criar o servidor, conforme mostrado no Algoritmo 4, é necessário instanciar um servidor gRPC (linha 18), registrar a implementação *StudentsService* (linhas 20 e 22), e então iniciá-lo (linha 28).

Desta forma, a etapa de construção do serviço foi finalizada. Agora basta consumir o serviço usando o código gerado automaticamente pelo compilador de *Protocol Buffers*. O Algoritmo Listagem 5 mostra como consumir o serviço.

Para isso, é iniciada uma conexão (canal) com o *endpoint* do serviço (linha 19), que abstrai a conexão HTTP. Então, na linha 24 é iniciado um cliente para consumir o serviço *StudentsService*, criado anteriormente. Todos os métodos definidos no *Protocol Buffers* ficam disponíveis neste cliente, e assim, basta chamar o método desejado, como na linha 25, onde é feita uma requisição para o método *GetStudents*. Como a mensagem é no formato binário, foi feita uma conversão para JSON simplesmente para ser visualizada, mas o código gerado pelo compilador transforma a mensagem binária na estrutura específica da linguagem, sem a necessidade de converter para JSON ou qualquer outro formato. A Figura 10 mostra o resultado da execução do cliente.

2.5.5 Considerações sobre a utilização de gRPC

GRPC possui diversos pontos positivos, como ter interfaces bem definidas, e mensagens mais leves por serem no formato binário. No entanto, também possui pontos negativos. Não possui suporte para todas as linguagens. Além disso, não é a tecnologia mais comum de ser utilizada e por utilizar HTTP/2, deve-se considerar que nem sempre será suportado. E apesar de ter as interfaces bem definidas, é preciso considerar que o código dos clientes terá que ser regerado para receber atualizações como novos campos das mensagens.

2.6 Considerações Finais

Neste capítulo foram abordados os conceitos fundamentais relacionados a este estudo, como Engenharia de Software, Arquitetura, Integração, bem com as duas tecnologias de foco. REST é um estilo muito usado, com suas vantagens e desvantagens, bem como gRPC, que é uma tecnologia relativamente nova e em crescimento. Ambas podem ser utilizadas para integração de serviços, sendo necessário considerar o contexto do software a ser construído ao tomar as

Listagem 3 – Implementação de um serviço usando gRPC

```
1  package server
2
3  import (
4    "context"
5    "google.golang.org/protobuf/types/known/timestamppb"
6    "grpc-rest/grpc/proto"
7    "grpc-rest/models/student"
8    "log"
9  )
10
11 type StudentsService struct {
12   proto.UnimplementedStudentsServiceServer
13 }
14
15 func NewStudentsServiceController() *StudentsService {
16   return &StudentsService{}
17 }
18
19 func (s *StudentsService)
20 GetStudents(ctx context.Context, req *proto.GetStudentsRequest)
21 (*proto.GetStudentsResponse, error) {
22   students, err := student.FetchAll(ctx)
23   if err != nil {
24     log.Println(err)
25     return nil, err
26   }
27
28   resp := &proto.GetStudentsResponse{}
29   for _, u := range students {
30     resp.Students = append(resp.Students, studentToProto(&u))
31   }
32
33   return resp, nil
34 }
35
36 func (s *StudentsService)
37 CreateStudent(ctx context.Context, request *proto.CreateStudentRequest)
38 (*proto.Student, error) {
39   // ...
40 }
```

Fonte: Autoria Própria.

decisões arquiteturais relacionadas com integração, que é um cenário comum principalmente ao utilizar o estilo arquitetural de microsserviços.

Listagem 4 – Registro de um serviço usando gRPC

```
1  package main
2
3  import (
4    "google.golang.org/grpc"
5    "google.golang.org/grpc/reflection"
6    "grpc-rest/config"
7    "grpc-rest/core"
8    "grpc-rest/grpc/proto"
9    services "grpc-rest/grpc/server"
10   "log"
11   "net"
12 )
13
14 func main() {
15   config.LoadEnv(config.RootPath() + "/config/.env")
16   core.StartApp()
17
18   server := grpc.NewServer()
19
20   proto.RegisterStudentsServiceServer(server,
21  \\ services.NewStudentsServiceController())
22
23   reflection.Register(server)
24
25   con, err := net.Listen("tcp", ":" + config.App.GrpcPort)
26   if err != nil {
27     log.Fatalln(err)
28   }
29   err = server.Serve(con)
30   if err != nil {
31     log.Fatalln(err)
32   }
33 }
```

Fonte: Autoria Própria.

Listagem 5 – Exemplo de consumo de um serviço usando gRPC

```
1  package main
2
3  import (
4    "context"
5    "encoding/json"
6    "google.golang.org/grpc"
7    "grpc-rest/config"
8    "grpc-rest/core"
9    "grpc-rest/grpc/proto"
10   "log"
11 )
12
13 func main() {
14   config.LoadEnv(config.RootPath() + "/config/.env")
15   core.StartApp()
16   ctx := context.Background()
17
18   serverAddress := "localhost:" + config.App.GrpcPort
19   conn, e := grpc.DialContext(ctx, serverAddress, grpc.WithInsecure())
20   if e != nil {
21     log.Fatal(e)
22   }
23
24   client := proto.NewStudentsServiceClient(conn)
25   students, e := client.GetStudents(ctx, &proto.GetStudentsRequest{})
26   if e != nil {
27     log.Fatal(e)
28     return
29   }
30
31   data, err := json.MarshalIndent(students.GetStudents(), "", " ")
32   log.Println(string(data), err)
33 }
```

Fonte: Autoria Própria.

Figura 10 – Resultado de consumo de um serviço usando gRPC


```
2021/10/17 17:48:59 [
 {
  "id": 1,
  "first_name": "Alisa",
  "last_name": "Roberts",
  "identifier": "01c6c4bb-7f9e-4622-b8ae-9b425e12d066",
  "created_at": {
   "seconds": 1634482961
  },
  "updated_at": {
   "seconds": 1634482961
  }
 },
 {
  "id": 3,
  "first_name": "Rubie",
  "last_name": "Hawkins",
  "identifier": "0b346cf9-3066-45c0-a1e5-f83ff6080d02",
  "created_at": {
   "seconds": 1634482961
  },
  "updated_at": {
   "seconds": 1634482961
  }
 }
] <nil>
```


Fonte: Autoria Própria.

3 METODOLOGIA

3.1 *Goal Question Metric*

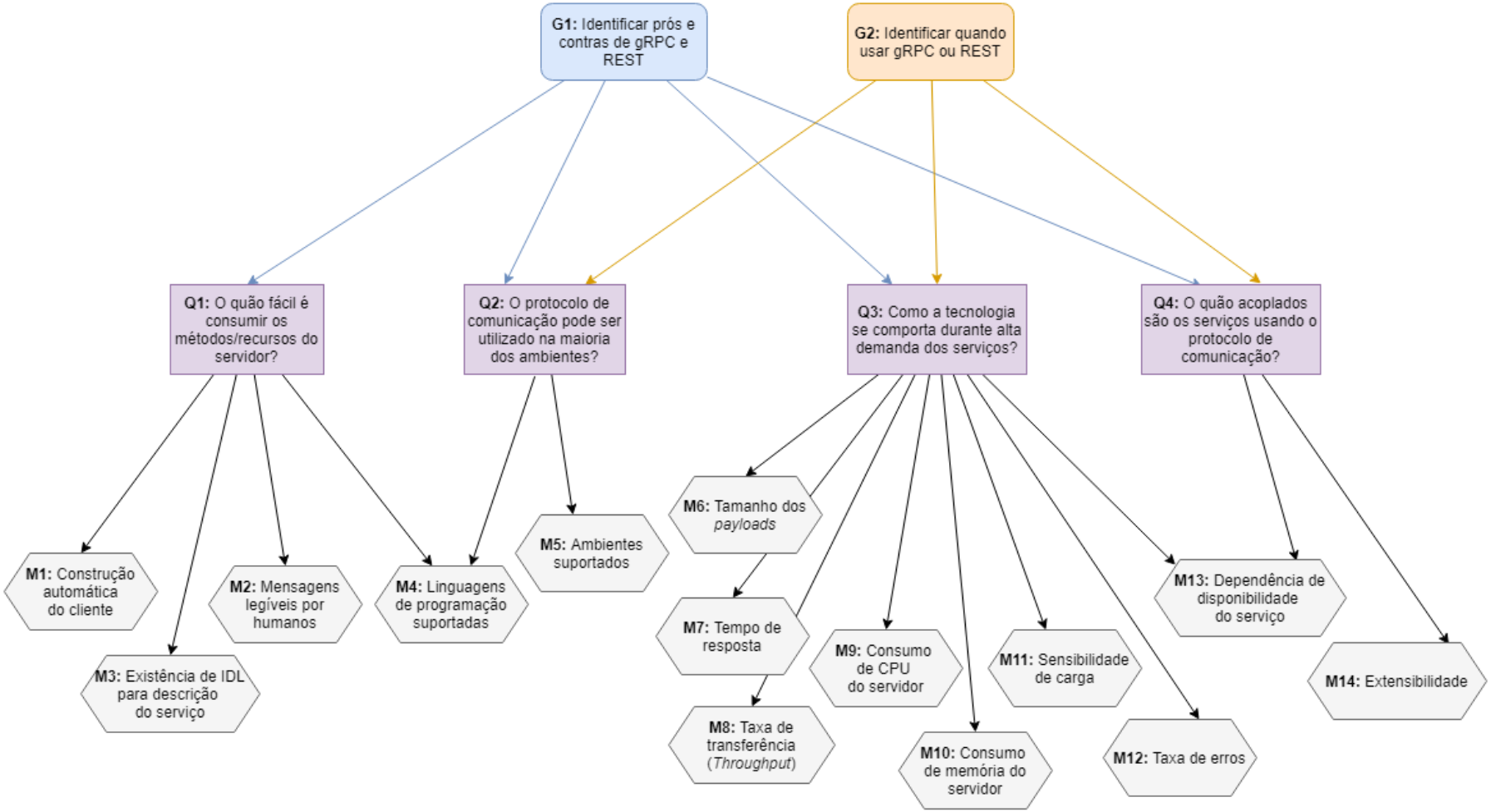
Para a realização deste trabalho é utilizada como base a abordagem *Goal Question Metric* (GQM), na qual as métricas são definidas de forma *top-down*, ou seja, inicia-se definindo os objetivos gerais e depois as métricas específicas. Esta abordagem foi originalmente criada para avaliar defeitos em um conjunto de projetos da NASA (do inglês *National Aeronautics and Space Administration*)[1], e depois foi expandida para um contexto maior. O resultado esperado da aplicação do modelo GQM é um modelo de medição específico para um problema, e é dividido em três níveis: *(i)* Conceitual, no qual são definidos os objetivos; *(ii)* Operacional, com a definição das questões que caracterizam como será avaliado um objetivo em específico; e *(iii)* Quantitativo, que define quais dados serão medidos para avaliar cada questão (BASILI; CALDIERA; ROMBACH, 1994).

O modelo GQM construído é ilustrado na Figura 11. Foram definidos dois objetivos principais para o aspecto conceitual:

- **G1:** Identificar prós e contras de gRPC e REST;
- **G2:** Identificar quando usar gRPC ou REST.

[1] https://www.nasa.gov/

Figura 11 – Modelo GQM criado para a comparação dos estilos arquiteturais REST e gRPC



Fonte: Autoria própria.

Para estes objetivos, no nível operacional foram definidas as seguintes questões:

- **Q1: O quão fácil é consumir os métodos/recursos do servidor?** - espera-se fazer uma análise da facilidade em consumir os recursos disponibilizados pelo servidor, visto que isso causa impacto na etapa de desenvolvimento das integrações (G1);
- **Q2: O protocolo de comunicação pode ser utilizado na maioria dos ambientes?** - neste contexto, espera-se que seja analisado, por exemplo, quais linguagens possuem suporte a cada um dos estilos arquiteturais, já que no contexto de microsserviços é comum ter que lidar com heterogeneidade dos sistemas (G1 e G2);
- **Q3: Como a tecnologia se comporta durante alta demanda dos serviços?** - nesta questão será analisada a performance dos serviços, pois pode ser um aspecto crítico para a decisão dependendo do contexto do software (G1 e G2); e
- **Q4: O quão acoplados são os serviços usando o protocolo de comunicação?** - será analisado o acoplamento que o estilo arquitetural insere nos serviços, pois acoplamento é um aspecto indesejável na arquitetura de microsserviços (G1 e G2).

Em seguida, para cada uma destas questões foi definido o nível quantitativo, com as métricas apresentadas a seguir:

- **M1: Construção automática do cliente** - será analisado se é possível construir o código do cliente a partir da IDL do serviço. Possíveis valores: “sim” e “não” (Q1);
- **M2: Mensagens legíveis por humanos** - será analisado se o tipo de mensagem utilizada pelo estilo arquitetural é legível. Uma mensagem binária não é considerada legível e uma mensagem textual é legível. Possíveis valores: “sim” e “não” (Q1);
- **M3: Existência de IDL para descrição do serviço** - nesta métrica será analisado se o estilo arquitetural possui uma IDL padrão definida, se existem alternativas não oficiais ou se não existe. Possíveis valores: possui IDL padrão (2), possui IDL não oficial (1) e não possui (0) (Q1);
- **M4: Linguagens de programação suportadas** - será analisado se o estilo arquitetural possui suporte geral a qualquer linguagem ou somente algumas linguagens. Possíveis valores: possui suporte geral das linguagens de programação (2); não possui suporte abrangente, mas é suportado por mais de uma linguagem (1); suporte de apenas uma linguagem (0) (Q1 e Q2);
- **M5: Ambientes suportados** - será analisado onde o protocolo de comunicação pode ser usado nos ambientes comuns de uso de serviços, entre Web, mobile e comunicação entre processos (entre os próprios serviços da arquitetura de microsserviços). Possíveis valores: todos os ambientes (2); somente alguns destes ambientes (1); nenhum (0) (Q2);

- **M6: Tamanho dos *payloads*** - o tamanho das mensagens transmitidas pela rede para avaliar a largura de banda utilizada. Medido em kilobytes, sendo que um valor menor é melhor (Q3);
- **M7: Tempo de resposta** - o tempo que o serviço leva para responder uma requisição. Medido em milissegundos, sendo que um valor menor é melhor (Q3);
- **M8: Taxa de transferência** - *throughput* dos serviços. Medido em requisições por minuto, sendo que um valor maior é melhor (Q3);
- **M9: Consumo de CPU do servidor** - será avaliado o quanto da CPU do servidor é utilizada de acordo com o aumento de carga. Medido em percentual de utilização, sendo que um valor menor é melhor (Q3);
- **M10: Consumo de memória do servidor** - será avaliado o quanto da memória do servidor é utilizada de acordo com o aumento de carga. Medido em megabytes, sendo que um valor menor é melhor (Q3);
- **M11: Sensibilidade de carga** - será avaliada a degradação do servidor de acordo com o aumento de carga. Será calculado o desvio padrão em relação ao tempo de resposta médio de acordo com cargas crescentes, sendo que um valor menor indica que o servidor se degrada mais lentamente com aumento da carga (Q3);
- **M12: Taxa de erros** - será avaliado o percentual de erros de resposta de acordo com o aumento de carga, sendo que um valor menor é menor (Q3);
- **M13: Dependência de disponibilidade do serviço** - será avaliado se há dependência do serviço estar disponível para fazer comunicação ou troca de mensagem entre serviços. Possíveis valores: não há dependência (1); há dependência (0) (Q3 e Q4);
- **M14: Extensibilidade** - será avaliado se é possível alterar as respostas de um método do serviço sem prejudicar as integrações já existentes. Possíveis valores: possível alterar (1); não é possível alterar (0) (Q4).

3.2 Abordagem Experimental

A abordagem experimental utilizada foi um estudo de caso, que é um método observacional utilizado para estudos empíricos em diversas áreas de conhecimento, inclusive na Engenharia de Software, e pode ser usado para avaliar diferenças entre dois métodos avaliando qual é mais adequado em determinada situação. Um dos métodos é considerado a base para a comparação, e são coletados seus dados. Então, é utilizado o método alternativo e coletados seus dados também, sendo que os dois projetos devem ter as mesmas características, ou seja, devem ser comparáveis (WOHLIN *et al.*, 2012).

Um estudo de caso é uma abordagem flexível e as conclusões são feitas a partir de um conjunto de evidências, podendo ser qualitativas e quantitativas. O processo para sua execução envolve as seguintes etapas: *(i)* definição de objetivos e planejamento; *(ii)* preparação para a coleta dos dados, com a definição do protocolo para coleta; *(iii)* coleta dos dados; *(iv)* análise dos dados; e *(v)* relato dos resultados. Uma forma de coletar dados quantitativos nesta abordagem é através do modelo GQM, que foi apresentado na seção anterior (WOHLIN *et al.*, 2012). O estudo de caso comparativo foi realizado nos serviços desenvolvidos, sendo o serviço em REST o modelo base, e o serviço em gRPC a comparação.

4 ESTUDOS EMPÍRICOS

Neste capítulo é explicado como foram realizados os estudos empíricos, desde a preparação para realização dos testes (construção dos serviços e ferramentas utilizadas), até a condução de cada teste individual para coletar as métricas definidas na metodologia do trabalho.

4.1 Procedimento Experimental

Para extrair as métricas definidas no modelo GQM apresentado anteriormente, foram coletadas informações na literatura disponível sobre ambas as tecnologias, bem como documentações técnicas. Além disso, algumas das métricas (principalmente as relacionadas com a Q3, ou seja, relacionadas com a alta demanda) foram coletadas a partir de uma análise empírica, na qual foram construídos dois serviços Web, um para cada estilo arquitetural (REST e GRPC).

O procedimento experimental utilizado, conforme mostrado na Figura 12 foi feito em três etapas:

- Preparação: foram implementados os dois serviços Web, e instrumentados (ou seja, os serviços foram configurados com uma ferramenta de monitoramento). Além disso, foram construídos os clientes para consumir os serviços, e criado um *script* para automatizar os testes de carga, tornando, assim, possível de reproduzir os testes e coletar métricas automaticamente;
- Execução: os serviços foram iniciados e então o *script* de teste foi executado, coletando as métricas;
- Resultados: os dados coletados foram analisados e as métricas faltantes foram coletadas na literatura e documentações.

4.2 *Setup*

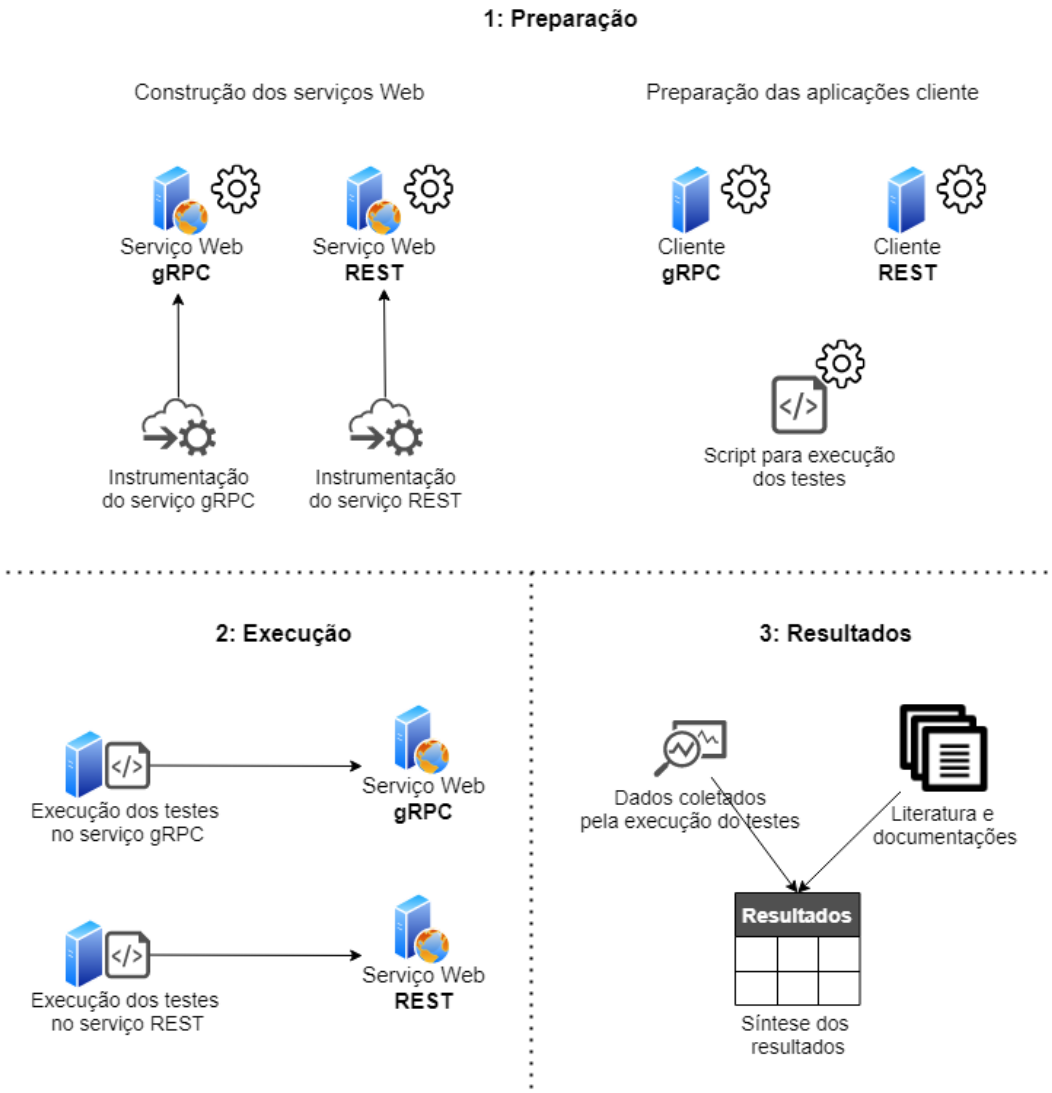
Nesta seção é apresentado como foram implementados os serviços, bem como o *script* de teste, e as tecnologias e ferramentas utilizadas.

4.2.1 Construção dos serviços

Para a construção dos serviços foi utilizada a linguagem de programação Go, visto que é uma linguagem com suporte às duas tecnologias deste estudo. Foi criado um sistema *open-source*[1] com as operações de CRUD. Os métodos disponíveis são no contexto de um sistema de

[1] O sistema *open-source* construído encontra-se disponível no seguinte repositório público: https://github.com/JanainaLudwig/go-rest-grpc

Figura 12 – Ilustração do desenvolvimento do estudo



Fonte: Autoria própria.

gerenciamento de alunos de uma universidade, seguindo o exemplo do capítulo de Fundamentação Teórica, e foram criados métodos que permitem avaliar a troca de mensagens com tamanhos variados. O desenvolvimento foi de forma modular, a fim de possibilitar a criação de duas interfaces de comunicação de maneira independente. Ou seja, a camada de comunicação foi isolada, com a responsabilidade de serializar e desserializar os dados apenas, mas executando a mesma regra de negócio. Isto foi importante para que os resultados representem somente a diferença entre os dois protocolos de comunicação, e não exista interferência por conta de implementações diferentes entre os dois serviços. Para o serviço REST foi utilizado o formato de mensagens JSON por ser o formato mais utilizado em microsserviços, e para o serviço gRPC foi utilizada a forma de comunicação unária, que corresponde e pode ser comparado ao REST.

Os métodos dos serviços se comunicam com um banco de dados MySQL[2], no entanto, os métodos utilizados nos testes retornam dados fixos, sem consulta a banco de dados. Isto foi feito para que não ocorram influências externas nos testes de carga, e para que os resultados reflitam somente a diferença dos dois protocolos. Além disso, utilizar banco de dados poderia fazer com que mais recursos de hardware fossem utilizados, diminuindo a capacidade dos testes de carga.

Outro aspecto importante da implementação dos serviços é que ambos os servidores foram instrumentados com uma ferramenta para observabilidade, que coleta os dados de transações (chamadas para a API), como tempo de resposta, utilização de memória e CPU, para o servidor. A ferramenta escolhida foi *NewRelic*, devido a ter uma integração simples para aplicações na linguagem Go, para ambos os protocolos de comunicação REST e gRPC. Apesar de ser uma ferramenta de mercado, existe uma versão gratuita para um usuário, que foi utilizada para monitorar os serviços durante os testes. Na Figura 13 é mostrado um exemplo do monitoramento através da *NewRelic*, na qual é possível observar como o aumento de carga influencia no aumento do tempo de resposta do servidor.

4.2.2 *Script* de teste

Para execução dos testes nos serviços foram utilizados dois clientes, também na linguagem Go, e um *script* para fazer requisições às APIs de maneira automatizada. Para consumir o serviço REST foi utilizado o cliente HTTP padrão da linguagem, e para consumir o serviço gRPC foi utilizado o cliente gerado pelo compilador de *Protocol Buffer*, que gera um cliente a partir da definição da interface no arquivo "*.proto*".

O *script* de teste foi implementado com o objetivo de automatizar as chamadas para os serviços, permitindo executar cargas variadas e incrementais, além de coletar dados como tempo de resposta e se ocorreu algum erro na requisição, de cada uma das transações. O *script* também foi implementado na linguagem Go, pois um dos recursos da linguagem é a implementação de execuções concorrentes, permitindo assim a execução de várias chamadas às APIs de maneira concorrente, independentemente de esperar que cada requisição fosse finalizada. Assim foi possível simular um ambiente real onde são feitas várias chamadas ao serviço ao mesmo tempo. Então foi especificado no *script* uma lista de quantas requisições deveriam ser feitas por segundo, e por quantos segundos, por exemplo:

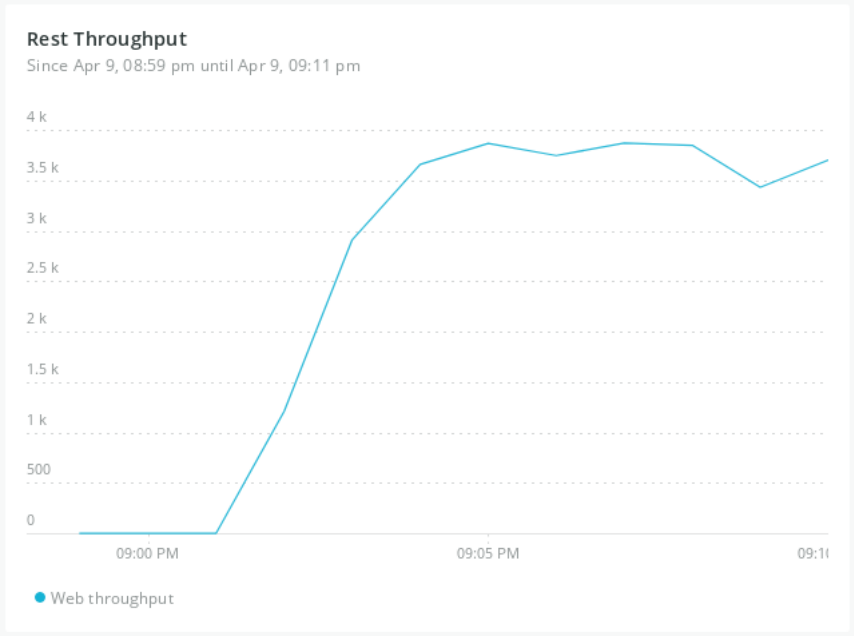
1. Dez requisições por segundo, por dois segundos;
2. Vinte requisições por segundo, por dois segundos.

A saída do *script* com a configuração descrita acima é exibida na Figura 14, na qual pode-se acompanhar a execução das chamadas. Além disso, no final são exibidos o total de

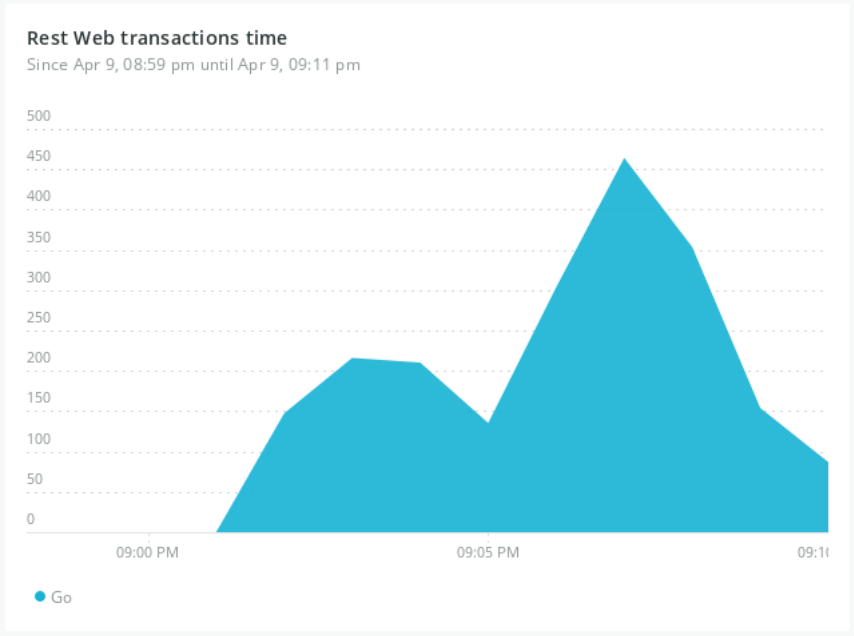
[2] https://www.mysql.com/

Figura 13 – Exemplo de gráficos gerados pela ferramenta *NewRelic* após teste de carga no serviço REST

(a) *Throughput* do servidor, medido em requisições por minuto



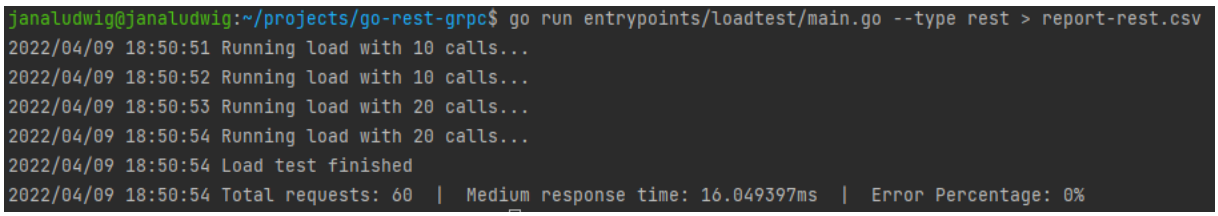
(b) Tempo de resposta do servidor, medido em milissegundos



Fonte: Autoria Própria.

requisições feitas, o tempo médio de resposta em milissegundos e o percentual de erro das requisições. A diferença do *script* desenvolvido para o monitoramento da *NewRelic* adicionado nos serviços é que no *script* as métricas são coletadas a partir da aplicação cliente, e não do servidor.

Figura 14 – Saída do *script* de teste de carga



Fonte: Autoria própria.

Por fim, é gerado um relatório no formato CSV com a configuração utilizada para rodar o teste, o cálculo das métricas coletadas de maneira geral (as mesmas exibidas na saída do terminal), além das métricas individuais para cada uma das requisições, permitindo assim a coleta de resultados precisos. Um exemplo de relatório pode ser observado na Figura 15

Figura 15 – Relatório gerado pelo *script* de teste de carga

	C1	C2	C3	C4
1	Number of requests	Medium response time	Error percentage	
2	5	3.901341ms	0	
3	Type	Requests per second	Duration	
4	rest	2	1s	
5	rest	3	1s	
6	Response time	Success	Error	End time
7	6.457139ms	true	-	Apr 23 10:46:23.319208381
8	7.103134ms	true	-	Apr 23 10:46:23.319782170
9	1.866798ms	true	-	Apr 23 10:46:24.314993905
10	1.818226ms	true	-	Apr 23 10:46:24.315019197
11	2.261409ms	true	-	Apr 23 10:46:24.315495107

Fonte: Autoria própria.

4.3 Condução e Resultados quantitativos

Nesta seção são apresentados os resultados para cada uma das métricas definidas no modelo GQM apresentado na metodologia, e como foi a condução de cada teste para obter a resposta. A partir destes resultados as questões de pesquisa foram respondidas, no Capítulo 5.

4.3.1 Coleta de métricas com base na literatura

A seguir são apresentados os resultados para as métricas que foram pesquisadas na literatura existente, bem como em documentações técnicas. Não foram realizados experimentos práticos para elas.

4.3.1.1 M1: Construção automática do cliente

REST: Apesar de alguns serviços (como a Amazon, por exemplo) fornecerem bibliotecas oficiais, ou existirem bibliotecas criadas pela comunidade para serviços muito utilizados, não há construção automática do cliente. Assim, as chamadas à API devem ser implementadas para a linguagem desejada, utilizando um cliente HTTP (RICHARDSON; RUBY, 2008).

GRPC: O compilador gera o cliente automaticamente a partir da interface definida no *Protocol Buffer*, sendo necessário somente utilizar o código gerado, como se o programador fizesse uma invocação de função local, já que a comunicação por HTTP fica abstraída (GRPC, 2018).

4.3.1.2 M2: Mensagens legíveis por humanos

REST: As mensagens utilizam o formato JSON, que é um formato textual, e portanto, legível.

GRPC: As mensagens em *Protocol Buffer* são no formato binário, portanto não são legíveis por humanos, sendo necessário desserializar as mensagens para ler seu conteúdo.

4.3.1.3 M3: Existência de IDL para descrição do serviço

REST: Não há um único padrão definido para definir a interface de comunicação, e as especificações do REST não definem qual formato deve ser utilizado como IDL. No entanto, existem alguns formatos criados pela comunidade e empresas que utilizam REST. Por exemplo: WADL (*Web Application Description Language*), que adicionou suporte ao REST na versão 2.0 e é baseado em XML; RSDL (*RESTful Service Definition Language*), também baseado em XML; RAML (*RESTful API Modeling Language*), que utiliza YAML para descrever a interface; e Swagger, também conhecido como *OpenAPI Specification*, que provavelmente é a IDL mais madura com suporte a JSON, e possui várias ferramentas de suporte, como para construir uma visualização da documentação, utilizando o Swagger UI (CARNEIRO; SCHMELMER, 2016; SWAGGER, 2022).

GRPC: GRPC utiliza como formato de mensagens os *Protocol Buffers*, que são também uma IDL, já que são definidas as assinaturas dos métodos e todas as possíveis mensagens. E como o uso e definição de arquivos "*.proto*" é necessário para construir um cliente ou servidor em GRPC, necessariamente será utilizado e definido o serviço com uma IDL, oficial do protocolo de comunicação (GRPC, 2018; CARNEIRO; SCHMELMER, 2016).

4.3.1.4 M4: Linguagens de programação suportadas

REST: APIs *Restful* são implementadas baseadas no protocolo de comunicação HTTP, e portanto, não dependem de uma linguagem em específico (CARNEIRO; SCHMELMER, 2016). Assim, qualquer linguagem com suporte ao protocolo HTTP pode ser utilizada. Dessa forma, considera-se que REST possui suporte geral de linguagens de programação.

GRPC: As linguagens suportadas por GRPC são C#, C++, Dart, Go, Java, Kotlin, Node, Objective-C, PHP, Python e Ruby (GRPC, 2018). Portanto, não há suporte geral de linguagens, mas não é restrito apenas a uma linguagem de programação.

4.3.1.5 M5: Ambientes suportados

REST: Da mesma forma que existe suporte para qualquer linguagem de programação que tenha suporte a chamadas HTTP, o mesmo ocorre com os ambientes suportados. Assim, há suporte para Web (navegadores), comunicação entre serviços e mobile.

GRPC: GRPC é suportado de maneira oficial para comunicação entre serviços e mobile (GRPC, 2018). No entanto, até o momento não existe como implementar GRPC com HTTP/2 nos navegadores, mas existe uma biblioteca chamada GRPC-Web que atua como um *proxy* entre o navegador e o servidor, ou seja, traduz as chamadas do navegador para HTTP/2 e encaminha as respostas. Dessa forma é possível utilizar GRPC na Web sem a necessidade de implementar outra API para traduzir as chamadas. Porém, nem todas as operações de GRPC são suportadas através deste *proxy*, sendo possível utilizar somente a comunicação unária e *Server-Streaming*. Não é possível utilizar *Client-Streaming* ou *Bidirectional-Streaming* (BRANDHORST, 2019), que apesar de não serem o foco deste estudo, fazem que seja considerado, então, que GRPC não é totalmente compatível com o ambiente Web (navegadores *client-side*). E além disso, mesmo que seja facilitado o uso neste ambiente através da biblioteca, é necessário utilizar este *proxy* para a comunicação ser possível.

4.3.1.6 M13: Dependência de disponibilidade do serviço

REST: REST funciona no modelo *request/response*, portanto, há dependência de disponibilidade dos serviços (NEWMAN, 2021), já que se o servidor não estiver disponível no momento da requisição, a integração vai falhar.

GRPC: Da mesma forma, GRPC utiliza o modelo *request/response*, tendo também dependência de disponibilidade dos serviços.

4.3.1.7 M14: Extensibilidade

REST: Deve-se evitar mudanças incompatíveis na API dos serviços (NEWMAN, 2021), mas é muitas vezes necessário, por exemplo, adicionar novos campos nas respostas do servidor. Utilizando REST com o formato JSON, é possível adicionar novos campos nas mensagens normalmente, ou seja, é possível alterar as mensagens e estender os serviços.

GRPC: GRPC utiliza os arquivos "*.proto*" para definir as interfaces. Caso o arquivo de definição das interfaces seja alterado, não será necessário atualizar imediatamente os clientes, desde que sejam seguidas algumas regras para manter a compatibilidade. Por exemplo, podem ser adicionados novos campos com identificadores diferentes, que serão ignorados pelos clientes que estão com a definição antiga das mensagens, mas isso não irá causar incompatibilidades nem quebrar integrações (GOOGLE, 2021). Portanto, GRPC também foi considerado como extensível.

4.3.2 Coleta de métricas com base no estudo de caso

A seguir são apresentados a condução e resultados encontrados para as métricas nas quais foram realizados testes práticos nos serviços desenvolvidos, utilizando as ferramentas de monitoramento apresentadas anteriormente, bem como o *script* de teste.

Para a execução dos testes foi utilizado o seguinte hardware: Intel® Core™ i7-11800H @ 2.30GHz, com 16GB de memória RAM (do inglês, *Random Access Memory*), e com o sistema operacional Windows 10, mas executando os testes no subsistema Windows para Linux (WSL2, do inglês, *Windows Subsystem for Linux*), utilizando Ubuntu 20.04 LTS dentro do WSL2. Para rodar os servidores REST e GRPC, utilizou-se a ferramenta *Docker* para criar os contêineres, isolando assim o ambiente de cada processo. Além disso, cada um dos contêineres foi configurado com 1 CPU e 1 GB de memória RAM, que é uma configuração comum para servidores Web.

4.3.2.1 M6: Tamanho dos *payloads*

Para analisar a diferença de tamanho dos *payloads*, foram utilizados dois métodos de cada protocolo e verificado também a diferença utilizando a compressão GZIP ao enviar as respostas HTTP. GZIP é um método de compressão de dados que pode ser utilizado para diminuir a largura de banda utilizada em chamadas HTTP (TSAI *et al.*, 2011). O primeiro método testado foi uma busca pelo *id* do estudante, e o segundo foi listagem de estudantes, variando a quantidade de registros retornados. O resultado mostrado no Quadro 1 demonstra que GRPC possui vantagens, pois em ambos os casos (com e sem compressão), o tamanho dos *payloads* foi menor em comparação a REST, consumindo menos largura de banda já que é um formato binário.

Quadro 1 – Comparação dos tamanhos dos payloads

Registros	Tamanho dos payloads (ms)			
	REST		GRPC	
	Padrão	GZIP	Padrão	GZIP
1	173 B	134 B	51 B	65 B
300	50,7 KB	360 B	15,53 KB	148 B
500	84,47 KB	495 B	24.9 KB	179 B
1000	167 KB	829 B	51,76 KB	256 B

Fonte: Autoria própria.

4.3.2.2 M7: Tempo de resposta

Foi utilizado o *script* de teste para fazer várias requisições de maneira concorrente aos serviços. O *script* foi executado cinco vezes utilizando o método de listar estudantes para cada um dos protocolos de comunicação a fim de observar se alguma das execuções apresentava um resultado muito diferente do padrão. Em seguida, foi executado o teste utilizando o método de criar estudantes, com a mesma configuração de carga. A execução dos dois métodos foi importante para testar tanto a entrada quanto a saída de dados do servidor, visto que o método de listagem retorna dados para o cliente e a criação recebe os dados enviados pelo cliente. Dessa maneira, analisou-se a serialização e desserialização de dados. A configuração utilizada para ambos os métodos foi:

- 10000 requisições por segundo, por 3 segundos;
- 12000 requisições por segundo, por 3 segundos.

Esta configuração apresentada resultou em 66000 requisições feitas para cada serviço em cada uma das cinco execuções dos testes de carga. O resultado obtido, como mostrado no Quadro 2, mostrou que GRPC teve um melhor desempenho em ambos os métodos testados (listagem e criação de recursos). GRPC obteve um tempo de resposta médio de 617ms para listagem e 36ms para criação. Já o protocolo REST obteve uma média de 792ms para listagem e 90ms para criação.

4.3.2.3 M8: Taxa de transferência

Para a métrica de taxa de transferência foi utilizada a mesma execução da métrica M7 (carga, número de execuções e métodos testados). O resultado é apresentado no Quadro 3, utilizando a medida de requisições por segundo. Em ambos os casos, GRPC também apresentou um melhor resultado: obteve 848 e 6267 requisições por segundo, para listagem e criação de recursos, respectivamente. Já REST obteve 673 e 3980 para estes mesmos métodos. No caso da

Quadro 2 – Comparação dos tempos de resposta

Execução	Tempo médio de resposta (ms)			
	REST		GRPC	
	Listar	Criar	Listar	Criar
1	777,355956	93,951196	625,539461	34,010099
2	790,979415	87,603146	609,77922	42,787764
3	797,078867	90,10314	630,459507	36,750807
4	801,266677	88,769139	617,051307	36,534865
5	796,042385	90,096819	605,485049	31,524332
Média	792,54466	90,104688	617,6629088	36,3215734

Fonte: Autoria própria.

taxa de transferência, quanto maior o valor obtido, melhor, já que o servidor consegue atender a mais requisições durante o mesmo período de tempo.

Quadro 3 – Comparação da taxa de transferência

Execução	Taxa de Transferência (req/s)			
	REST		GRPC	
	Listar	Criar	Listar	Criar
1	685,9160	3876,0930	840,8873	6371,6933
2	673,4698	4061,1504	854,3012	5863,1511
3	670,2213	3990,3004	832,4605	6202,7842
4	666,8273	4033,0983	849,7804	6354,1078
5	672,6082	3943,3078	865,2303	6545,4158
Média	673,80852	3980,78998	848,53194	6267,43044

Fonte: Autoria própria.

4.3.2.4 M9: Consumo de CPU do servidor

Para o teste de consumo de CPU do servidor, também foi utilizado o *script* de teste desenvolvido, porém com outras configurações de carga. Neste caso, também foram testados os métodos de listagem e de criação de estudantes, utilizando uma carga crescente. Para o método de criação, foi iniciado em 100 requisições por segundo (req/s) durante 2 segundos, e aumentando linearmente de 200 em 200 requisições até chegar em 45000 req/s, pelos mesmos 2 segundos. Isso resultou em um total de 105800 requisições para cada um dos dois serviços. Já o método de listagem, utilizando esta mesma carga, chegou muito rapidamente em 100% de

uso da CPU em ambos os protocolos, por isso foi utilizada uma carga mais baixa, mas por um período de tempo maior. No caso do método de listagem, então, foi utilizada a carga de 100 req/s durante 10 segundos, aumentando 200 requisições até chegar em 1100 req/s, resultando em 36000 requisições para cada servidor. Estas mesmas execuções foram utilizadas para coleta da métrica 10 (consumo de memória).

Para coletar as informações de consumo de CPU e memória do servidor, foi utilizado um comando do Docker[3] em conjunto com alguns comandos em shell para gravar os dados em um arquivo. O comando completo é mostrado no Algoritmo 6. Nele, é feito uma execução a cada 0.3 segundo. Cada ciclo executa o comando "*docker stats*" (com alguns parâmetros, como o ID do *container* do servidor e formato de saída), que retorna o status da utilização dos recursos reservados para o *container* especificado - configurado com 1 CPU e 1GB de memória. Depois, é utilizado o comando "*grep*" para remover o cabeçalho da saída do comando anterior, e por fim é utilizado o comando "*tee*" para gravar os dados em um arquivo CSV.

Listagem 6 – Exemplo de *endpoints* no estilo REST utilizando a linguagem Go

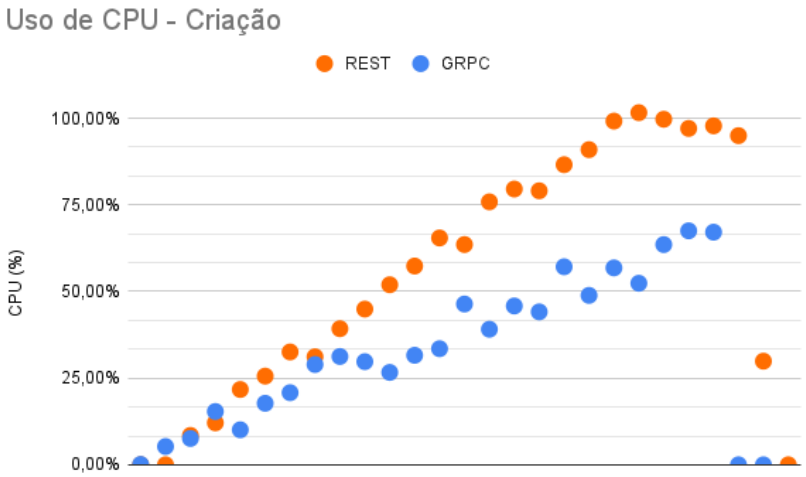
```
1 while true; do \
2 docker stats {CONTAINER_ID} --no-stream --format "table {{.Name}},
3 {{.CPUPerc}},{{.MemUsage}},{{.NetIO}},{{.BlockIO}},{{.PIDs}}" | \
4 grep 'NAME' -v | tee --append stats.csv; \
5 sleep 0.3; done
```

Fonte: Autoria Própria.

O resultado da utilização de CPU para o método de criação é mostrado da Figura 16. Observa-se que REST obteve uma utilização muito maior da CPU conforme a carga aumentava, tendo atingido 100% de uso, enquanto o servidor GRPC, apesar de também ter aumentado o consumo da CPU, ficou abaixo de 75% de uso. Já para o método de listagem, como mostrado na Figura 17, o uso da CPU foi muito parecido, apesar de que com as cargas inciais GRPC teve um uso ligeiramente menor, com cargas mais altas não houveram diferenças significativas. Portanto, GRPC foi considerado melhor em relação ao consumo de CPU do servidor, especialmente em métodos onde há maior entrada de dados ao invés de saída.

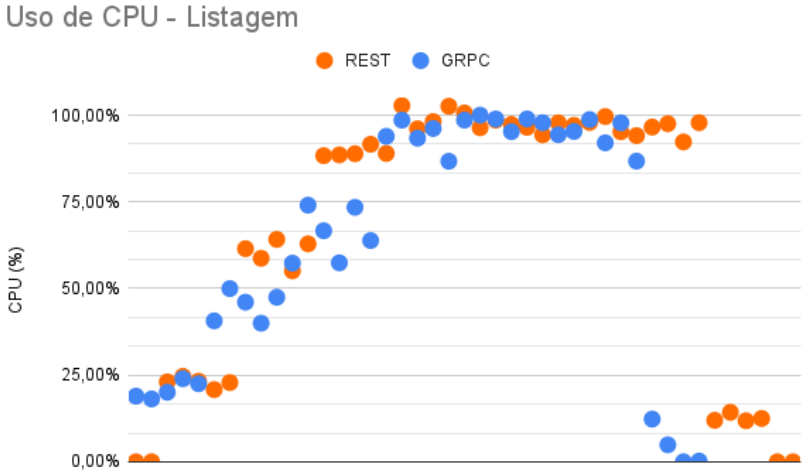
[3] https://www.docker.com/

Figura 16 – Comparação do uso de CPU do servidor (método de criação)



Fonte: Autoria própria.

Figura 17 – Comparação do uso de CPU do servidor (método de listagem)

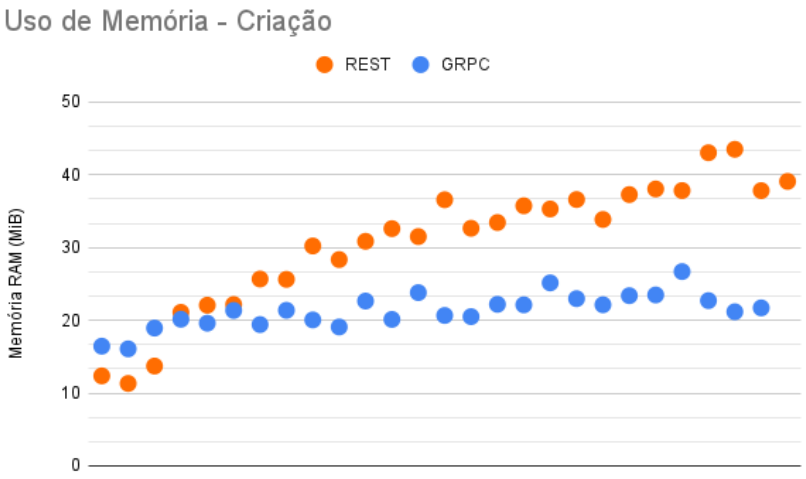


Fonte: Autoria própria.

4.3.2.5 M10: Consumo de memória do servidor

Para a métrica de consumo de memória do servidor foi utilizada a mesma execução e método de coleta de dados da métrica M9. O resultado do método de criação é apresentado na Figura 18, utilizando a medida de MiB (MebiBytes). É possível observar que ambos os protocolos aumentaram o uso da memória conforme a carga aumentou, porém REST obteve um aumento maior em comparação a GRPC. O servidor REST teve um pico de 43,49 MiB enquanto o servidor GRPC teve um pico de 26,69 MiB. No método de listagem, mostrado na Figura 19, observa-se o mesmo comportamento: o servidor GRPC teve menor uso, atingindo um pico de 51,32 MiB enquanto REST obteve um pico de 93,47 MiB. Um consumo de memória mais baixo é melhor por indicar que o serviço é capaz de lidar com mais requisições, visto que demanda menos do hardware. Portanto, GRPC também foi considerado melhor em relação ao consumo de memória.

Figura 18 – Comparação do uso de memória do servidor (método de criação)



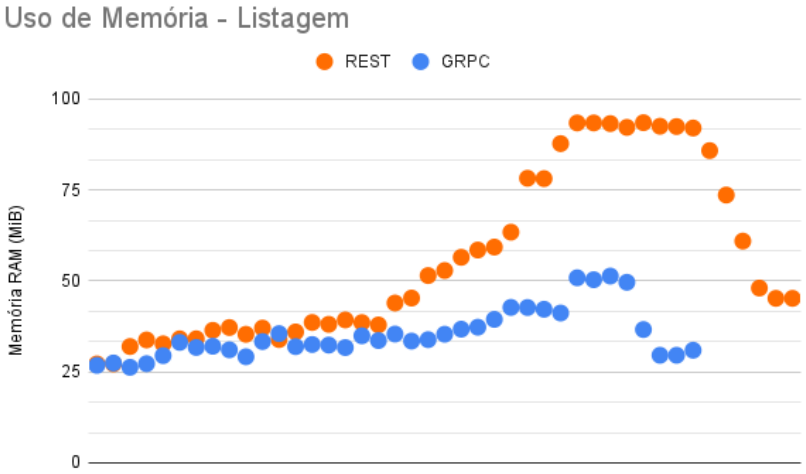
Fonte: Autoria própria.

4.3.2.6 M11: Sensibilidade de carga

A sensibilidade de carga foi medida utilizando o *script* de teste com o método de listagem, com cargas incrementais, utilizando a seguinte configuração:

- 50 req/s por 30s;
- 100 req/s por 30s;
- 200 req/s por 30s;
- 300 req/s por 30s;

Figura 19 – Comparação do uso de memória do servidor (método de listagem)



Fonte: Autoria própria.

- 400 req/s por 30s;
- 500 req/s por 30s;
- 600 req/s por 30s.

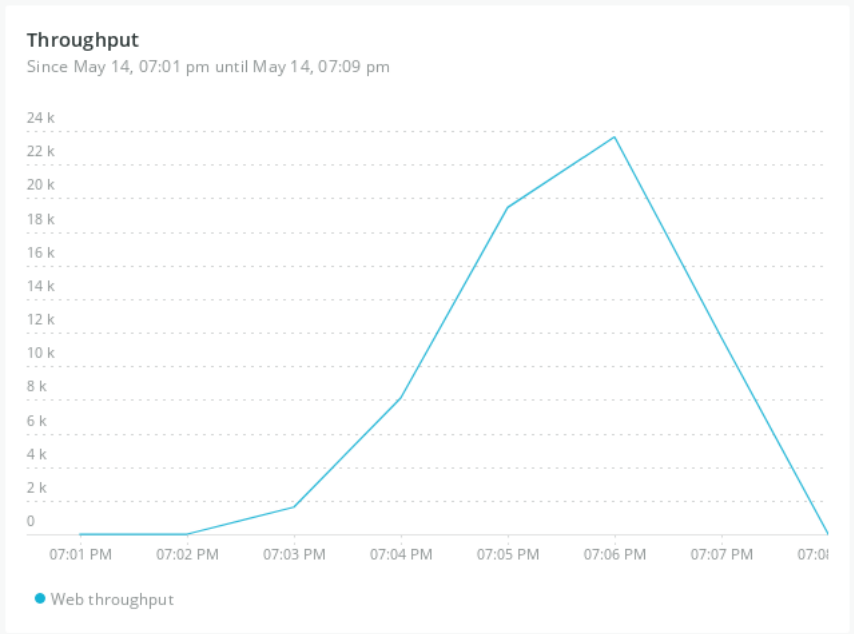
Para capturar os dados foi utilizada a ferramenta NewRelic e os relatórios gerados pelo *script*. Utilizando os gráficos gerados pela NewRelic, é possível visualizar o resultado, como mostrado nas Figuras 20 e 21. Nas subfiguras *(a)* é possível verificar quantidade de chamadas feitas ao servidor. Já nas subfiguras *(b)* são mostrados os tempos de resposta para o mesmo período de tempo. Observa-se que, conforme as cargas foram aumentando, o tempo de resposta também aumentou para ambos os protocolos.

No entanto, REST teve um desvio padrão maior, ou seja, com a mesma carga, teve um aumento médio maior no seu tempo de resposta. Através do relatório do script, foi possível calcular que REST obteve um desvio padrão de 89,98, enquanto GRPC obteve um desvio padrão de 52,34.

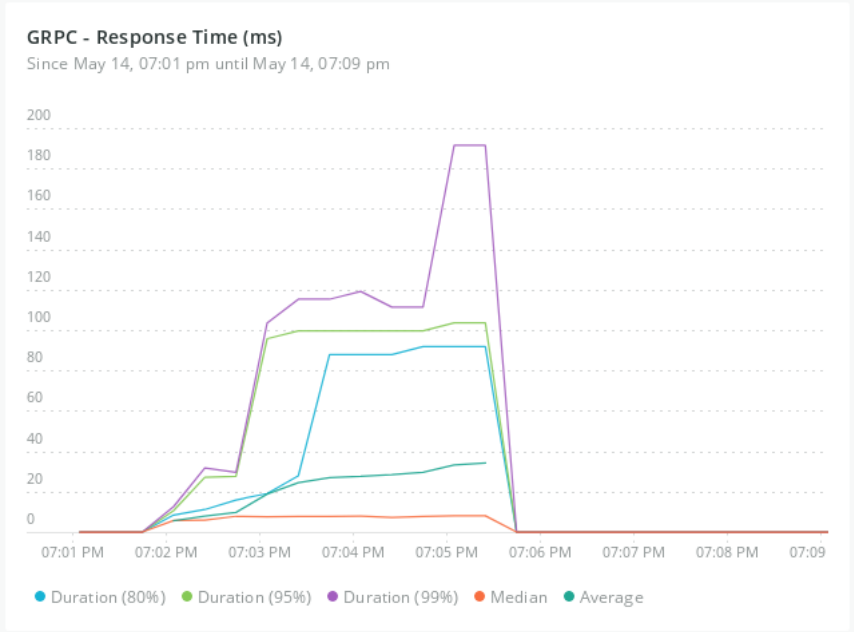
Esta tendência pode ser observada na Figura 22, que mostra os tempos médios de resposta de ambos os protocolos, na qual verifica-se que o servidor GRPC manteve seu tempo de resposta mais regular em comparação com o servidor REST. Portanto, GRPC apresentou menor sensibilidade de carga, o que indica que mesmo aumentando a carga, o servidor não se degradará tão rapidamente quando comparado a REST.

Figura 20 – Resultados do teste de sensibilidade de carga capturado na ferramenta NewRelic para o servidor GRPC.

(a) *Throughput* do servidor GRPC, medido em requisições por minuto



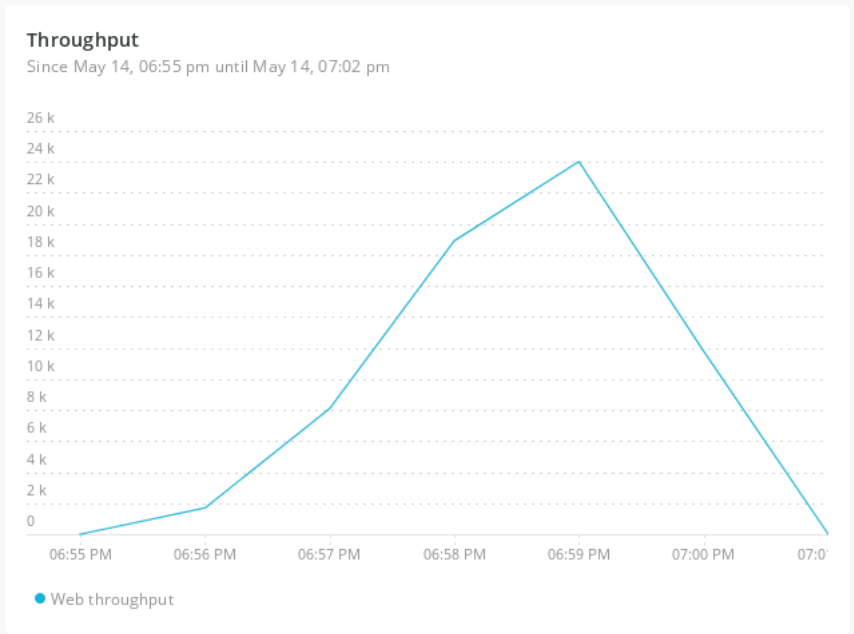
(b) Percentis de tempo de resposta do servidor GRPC, medido em milissegundos



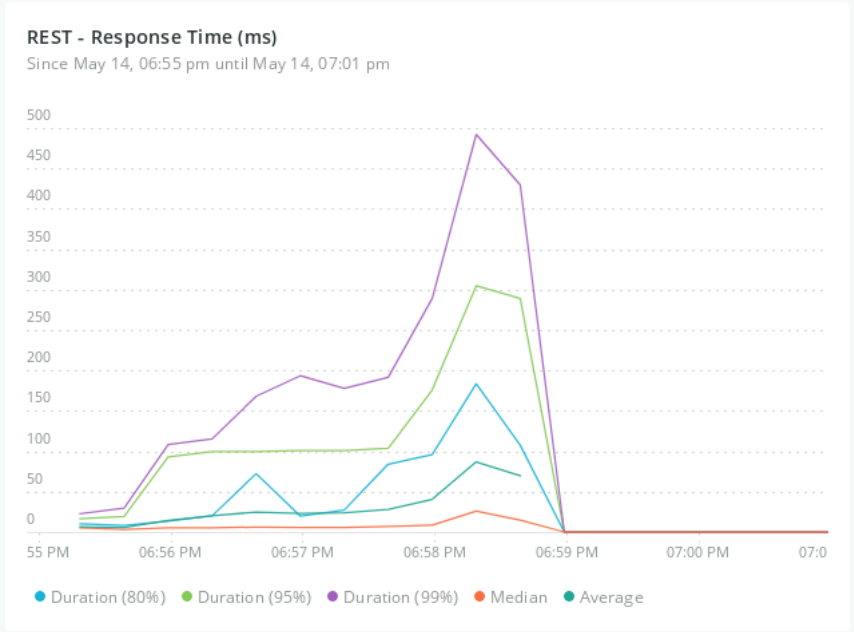
Fonte: Autoria Própria.

Figura 21 – Resultados do teste de sensibilidade de carga capturado na ferramenta NewRelic para o servidor GRPC

(a) *Throughput* do servidor REST, medido em requisições por minuto

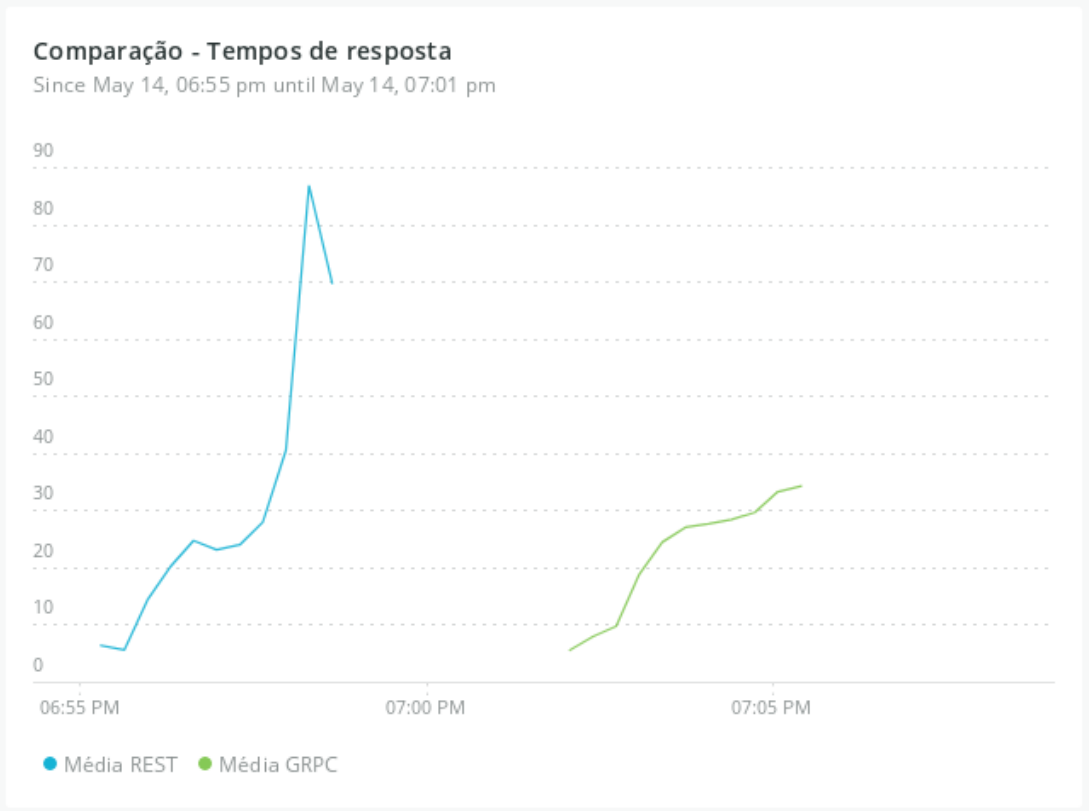


(b) ercentis de tempo de resposta do servidor REST, medido em milissegundos



Fonte: Autoria Própria.

Figura 22 – Sensibilidade de carga dos servidores



Fonte: Autoria própria.

4.3.2.7 M12: Taxa de erros

Para a métrica de taxa de erros, foi utilizada a mesma execução da métrica M7 (carga, número de execuções e métodos testados), e ambos os protocolos obtiveram 0% de erro. Por conta disso, em todas as demais execuções foi observada esta métrica, mas manteve-se o mesmo resultado. Portanto, nos testes realizados tanto REST e GRPC apresentaram um resultado satisfatório, não ocorrendo nenhum erro no servidor.

5 RESULTADOS E DISCUSSÕES

Nesta seção é apresentada uma sumarização dos resultados, a resposta para as questões de pesquisa, bem como as ameaças à validade identificadas para este estudo.

As métricas coletadas foram sumarizadas no Quadro 4, no qual foi registrado o valor numérico de cada uma das métricas de acordo com o modelo GQM definido na metodologia. Ao lado de cada uma das métricas, uma seta indica qual o melhor resultado (seta para cima indica que quanto maior o resultado, melhor), com um destaque por cores de qual protocolo resultou melhor (na cor roxa para GRPC e laranja para REST). Por exemplo, para a métrica M7 quanto menor o valor, melhor o resultado.

Quadro 4 – Resultados coletados para as métricas do modelo GQM

Métrica	**REST**		**GRPC**	
↑ M1	0	Não automático	1	Automático
↑ M2	1	Legível	0	Não legível
↑ M3	1	IDL não oficial	2	Possui IDL
↑ M4	2	Suporte geral	1	Suporte parcial
↑ M5	2	Todos	1	Parcial
↓ M6	50,7 KB	300 registros	15,53 KB	300 registros
↓ M7	∼792ms/∼90ms	Listagem/Criação	∼617ms/∼36ms	Listagem/Criação
↑ M8	∼673/∼3980	Listagem/Criação	∼848/∼6267	Listagem/Criação
↓ M9	∼73%/∼61%	Listagem/Criação	∼69%/∼36%	Listagem/Criação
↓ M10	∼57/∼32MiB	Listagem/Criação	∼35/∼21MiB	Listagem/Criação
↓ M11	89,98	Desvio padrão	52,34	Desvio padrão
↓ M12	0%	Erros	0%	Erros
↑ M13	0	Dependente	0	Dependente
↑ M14	1	Extensível	1	Extensível
Legenda (melhor resultado)		Mesmo resultado	REST	GRPC

Fonte: Autoria Própria.

5.1 Questões de pesquisa

A partir das métricas coletadas, foi possível responder às questões de pesquisa do modelo GQM. Em relação à Questão 1, "**O quão fácil é consumir os métodos/recursos do servidor?**", o resultado encontrado indica que ambos os protocolos tem vantagens e desvantagens equilibradas. Por um lado, GRPC possui sua interface bem definida através de uma IDL, além

de ser possível gerar o cliente automaticamente, o que torna fácil o consumo de métodos do servidor, e traz rapidez ao desenvolvimento já que as chamadas ocorrem como se fosse uma chamada local. No entanto, o tipo de mensagem utilizada (*Protocol Buffers*) não é legível, e nem todas as linguagens são suportadas, mas estes são pontos de vantagem para REST, já que a linguagem legível pode fazer o *debug* de aplicações ser mais fácil, e consequentemente auxiliando na manutenção. Além disso, REST é disponível para qualquer linguagem com suporte a chamadas HTTP, e assim, não há dependência de algumas linguagens específicas, o que poderia ser um impeditivo em integrações de software caso fosse necessário consumir um servidor em GRPC utilizando uma linguagem sem suporte do protocolo.

Na Questão 2 "**O protocolo de comunicação pode ser utilizado na maioria dos ambientes?**", REST sim, já que pode ser utilizado em qualquer ambiente e linguagem com suporte às chamadas HTTP, mas GRPC não pode ser utilizado em todos. Utilizando GRPC, apesar de existirem alternativas como o uso de um proxy, não pode ser utilizado em navegadores Web nativamente, por exemplo, e este é um uso comum de APIs Web. Para isso seria necessário a utilização de uma API intermediária, o que pode aumentar a latência da resposta. Outro ponto negativo, é que APIs públicas, ou seja, com intenção de serem disponibilizadas para consumo de terceiros, construídas com GRPC ficam limitadas às linguagens que possuem suporte, o que não acontece com REST que é totalmente independente da linguagem de programação.

A Questão 3 "**Como a tecnologia se comporta durante alta demanda dos serviços?**" foi respondida pelas métricas do estudo de caso, onde percebe-se que GRPC obteve vantagem na maioria das métricas. A taxa de erros apresentou o mesmo resultado em ambos os protocolos. As demais podem ser relacionadas para explicar o resultado obtido. Por exemplo, houve menor sensibilidade de carga em GRPC, o que pode ser explicado pelo menor consumo de CPU e memória. Por sua vez, a menor utilização de CPU e memória pode estar relacionada ao tamanho dos payloads, já que para transportar exatamente os mesmos dados, os payloads no protocolo GRPC são menores, bem como a uma serialização mais eficiente dos dados por utilizar o formato binário para as mensagens. Além disso, esta menor utilização do hardware em conjunto com o menor tempo de resposta pode estar relacionada com uma maior taxa de transferência. Portanto, GRPC comporta-se muito bem durante alta demanda, podendo ser uma boa escolha quando é necessário alta taxa de transferência ou alta velocidade de resposta. Já REST, apesar de também se comportar bem durante alta demanda (pois o serviço manteve-se respondendo sem erros), ficou abaixo de GRPC nesta questão. Na métrica de dependência do serviço, ambos apresentam dependência por serem um método de integração síncrona, o que poderia ser um ponto negativo caso o serviço fique indisponível.

Na Questão 4: "**O quão acoplados são os serviços usando o protocolo de comunicação?**" não houveram diferenças entre ambos. REST e GRPC podem ter seu contrato da API evoluídos desde que não ocorram mudanças incompatíveis, mas ambos possuem dependência do serviço estar disponível durante a integração de software. No entanto, isso é esperado para integrações síncronas, mas faz com que exista certo acoplamento entre os serviços.

Assim, as vantagens e desvantagens de cada um dos protocolos foram elencadas e são mostradas na Figura 23.

Figura 23 – Vantagens e desvantagens de GRPC e REST

Fonte: Autoria própria.

A partir destes resultados é possível pensar em diferentes estratégias de integração dependendo da necessidade do sistema. Por exemplo, em um software que precisa de uma API pública disponibilizada a terceiros, seria uma boa escolha a utilização de REST, porque que não há nenhuma dependência de linguagem ou plataforma como existe em GRPC. Já em serviços que precisam de muita *performance*, haveria vantagem em utilizar GRPC por utilizar menos hardware, ter menor tempo de resposta e taxa de transferência mais alta. Além disso, há a possibilidade de pensar em uma estratégia mista: utilizar GRPC para a comunicação de serviços internos que não são disponíveis em uma rede pública, portanto utilizados exclusivamente para integração dos serviços *backend*, e disponibilizar os métodos públicos em uma API REST, que poderia ser escalada conforme a demanda, mas sem necessidade de escalar todos os outros serviços. Nesta estratégia, também há o benefício de manter as interfaces dos servidores muito bem definidas, o que torna a integração mais robusta, e a API em REST permite a comunicação com serviços externos, como a disponibilização dos métodos para o ambiente Web (navegadores) e integrações com terceiros.

Por fim, é necessário que o arquiteto de software analise as vantagens e desvantagens de cada modelo de integração em conjunto com as necessidades e atributos de qualidade do software que será construído, a fim de tomar uma decisão sobre o modelo de integração a ser adotado. Desta forma, com a utilização das métricas coletadas, é possível chegar a uma decisão assertiva para cada contexto individualmente.

5.2 Ameaças à validade

As ameaças à validade podem ser divididas em ameaças de conclusão, internas, de construção e externas. Ameaças de conclusão estão relacionadas a quão certo pode-se estar de que o resultado realmente corresponde ao experimento realizado. Ameaças internas analisam se podem ter outros fatores não medidos ou dos quais não houve controle que podem ter causado o resultado final. Já ameaças de construção focam na relação entre o experimento e a observação, analisando se realmente estão relacionados. Por fim, ameaças externas verificam se os resultados podem ser aplicados fora do escopo do estudo em questão. (FELDT; MAGAZINIUS, 2010)

Não foram identificadas ameaças de conclusão nem de construção pois ambos os serviços testados executavam a mesma ação, sendo isolada para teste somente a diferença entre os dois protocolos testados, além do isolamento do ambiente de execução dos serviços, diminuindo fatores que poderiam influenciar no resultado final.

A ameaça externa identificada foi a possibilidade dos métodos dos serviços construídos não serem próximos de serviços do mundo real, ou mesmo ocorrer diferenças dependendo do tipo de sistema utilizado em diferentes contextos. Por exemplo, se retornando tipos de dados muito diferentes dos testados, haveria diferenças no resultado final.

Uma ameaça interna identificada foi a dependência de serviços externos ao realizar o teste nos serviços desenvolvidos, especialmente o banco de dados. Apesar do uso de um banco local, a alta carga aplicada nos serviços ocupava recursos do banco de dados também, diminuindo o total de carga que ambos os serviços suportavam. Foi observado um grande aumento no consumo de CPU do hardware utilizado. Porém, testar o acesso ao banco foi considerado fator irrelevante para as diferenças encontradas nos dois protocolos, visto que independentemente do método utilizado (REST ou GRPC) o consumo ao banco era feito exatamente da mesma maneira. Então, para prevenir influências externas e possibilitar um teste com carga maior, foi construído um método que retornava dados sem fazer nenhuma consulta ao banco, isolando assim a diferença entre os dois protocolos de comunicação testados, e sem comprometer o resultado final.

6 CONCLUSÃO

6.1 Conclusão

Com o avanço da tecnologia, requisitos cada vez mais complexos e surgimento de novos estilos arquiteturais de software como microsserviços, surgem também desafios para integrar os sistemas, mas também novas tecnologias para Integração de Software. Nesse sentido é necessário avaliar os requisitos específicos do sistema e as diferentes abordagens de integração para que se utilize a mais adequada no contexto do software a ser construído. Entre os estilos arquiteturais para integração no modelo *request/response*, que é uma das formas de fazer integração, estão REST, que é muito utilizado, e gRPC, que é uma tecnologia criada recentemente. Assim, são necessárias informações sobre estes dois estilos arquiteturais para auxiliar na construção de software.

Por isso, com o objetivo de obter dados que auxiliem na decisão de qual protocolo utilizar entre REST e GRPC para integração de serviços, foi construído um protocolo para comparação de estilos arquiteturais de integração de software utilizando o modelo GQM. Esta contribuição pode também ser utilizada para comparar outras tecnologias no futuro. Além disso, foram criados serviços open-source para ambos os modelos de integração, bem como um script para automatização dos testes nestes serviços, que também podem ser expandidos para análise de outros protocolos de integração.

Por meio desta análise empírica e da pesquisa na literatura e documentações técnicas, foi possível extrair as métricas desejadas e analisar o comportamento dos serviços durante alta demanda, bem como em quais ambientes cada protocolo se comporta melhor. REST possui independência da linguagem utilizada, e portanto, é ideal para integrações externas e integrações entre diferentes plataformas. GRPC possui alta performance, mas por haver limitação de linguagens e ambientes, é indicado para sistemas internos com alta demanda.

Por fim, estas métricas coletadas permitiram analisar as duas abordagens e levantar vantagens e desvantagens para cada uma delas. Com estes resultados foi feita a principal contribuição deste estudo, fornecendo informações a arquitetos e desenvolvedores de software para auxiliar nas decisões arquiteturais de integração de software ao escolher entre REST e GRPC, visto que a decisão depende do contexto de cada sistema.

6.2 Trabalhos futuros

Em pesquisas futuras, pretende-se implementar GRPC em outras linguagens de programação para comparar se há diferenças relevantes de performance entre as linguagem suportadas para este protocolo ou se os resultados não dependem da linguagem utilizada. Além disso, outro aspecto a ser estudado é como a experiência de desenvolvimento difere ao utilizar estes

protocolos, ou seja, entender se os desenvolvedores possuem uma melhor experiência de desenvolvimento dependendo do protocolo utilizado.

REFERÊNCIAS

BASILI, V. R.; CALDIERA, G.; ROMBACH, D. H. **The Goal Question Metric Approach**. [*S.l.*]: John Wiley & Sons, 1994. I.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice, 4th Edition**. [*S.l.*]: Addison-Wesley Professional, 2021. ISBN 9780136885979; 0136885977.

BOURQUE, P.; FAIRLEY, R. E. (Ed.). **SWEBOK: Guide to the Software Engineering Body of Knowledge**. Version 3.0. IEEE Computer Society, 2014. ISBN 978-0-7695-5166-1. Disponível em: http://www.swebok.org/.

BRAJESH, D. **API Management. An Architect's Guide to Developing and Managing APIs for Your Organization**. [*S.l.*]: Apress, Berkeley, CA., 2017.

BRANDHORST, J. **The state of gRPC in the browser**. 2019. Acesso em 23 abr. de 2022. Disponível em: https://grpc.io/blog/state-of-grpc-web/.

CARNEIRO, C.; SCHMELMER, T. Microservices from day one. **Apress. Berkeley, CA**, Springer, 2016.

CLEMENTS, P. *et al.* **Documenting Software Architectures: Views and Beyond, Second Edition**. [*S.l.*]: Addison-Wesley Professional, 2010.

FELDT, R.; MAGAZINIUS, A. Validity threats in empirical software engineering research-an initial survey. *In*: **Proceedings of the 22nd International Conference on Software Engineering Knowledge Engineering (Seke)**. [*S.l.*]: Knowledge Systems Institute Graduate School, 2010. p. 374–379.

FENG, X.; SHEN, J.; FAN, Y. Rest: An alternative to rpc for web services architecture. *In*: **Proceedings of the First International Conference on Future Information Networks**. [*S.l.*]: IEEE, 2009. p. 7–10.

FOWLER, M. **Patterns of Enterprise Application Architecture**. [*S.l.*]: Addison-Wesley, 2012.

GOOGLE. **Language Guide (proto3) | Protocol Buffers | Google Developers**. Google, 2021. Acesso em 17 de out. de 2021. Disponível em: https://developers.google.com/protocol-buffers/docs/proto3.

GRPC, A. **High performance, open-source universal RPC framework**. 2018. Acesso em 25 de set. de 2021. Disponível em: https://grpc.io/.

HOHPE, G. *et al.* **Enterprise Integration Patterns: building, and deploying messaging solutions**. [*S.l.*]: Addison-Wesley Pearson-Education Boston, 2003.

INDRASIRI, K.; KURUPPU, D. **gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes**. [*S.l.*]: O'Reilly Media, 2020. ISBN 9781492058281.

JOSUTTIS, N. M. **SOA in practice: the art of distributed system design**. [*S.l.*]: "O'Reilly Media, Inc.", 2007.

MASSE, M. **REST API Design Rulebook**. [*S.l.*]: O'Reilly Media, 2011. (Oreilly and Associate Series). ISBN 9781449310509.

META. **Graph API**. 2022. Acesso em 28 de mar. de 2022. Disponível em: https://developers.facebook.com/docs/graph-api.

MOLYNEAUX, I. **The Art of Application Performance Testing, 2nd Edition**. [*S.l.*]: O'Reilly Media, Inc., 2014.

NEWMAN, S. **Monolith to microservices: evolutionary patterns to transform your monolith**. [*S.l.*]: O'Reilly Media, 2019.

NEWMAN, S. **Building microservices**. [*S.l.*]: "O'Reilly Media, Inc.", 2021.

OPENWEATHER. **Open Weather API**. 2022. Acesso em 28 de mar. de 2022. Disponível em: https://openweathermap.org/api.

PAWLIKOWSKI, M. **Chaos Engineering: Site reliability through controlled disruption**. [*S.l.*]: Manning, 2021. ISBN 9781617297755.

PERDANAPUTRA, A.; KISTIJANTORO, A. I. Transparent tracing system on grpc based microservice applications running on kubernetes. *In*: **Proceedings of the 7th International Conference on Advance Informatics: Concepts, Theory and Applications (ICAICTA)**. [*S.l.*]: IEEE, 2020. p. 1–5.

PRESSMAN, R.; MAXIM, B. **Software Engineering: A Practitioner's Approach 9th Edition**. [*S.l.*: *s.n.*], 2019. ISBN 9781259872976.

RICHARDSON, C. **Microservices patterns: with examples in Java**. [*S.l.*]: Simon and Schuster, 2018.

RICHARDSON, L.; RUBY, S. **RESTful web services**. [*S.l.*]: O'Reilly Media, Inc., 2008.

SOMMERVILLE, I. **Software Engineering**. 10. ed. [*S.l.*]: Pearson Australia Pty Limited, 2016.

SOMMERVILLE, I. **Engineering Software Products: An Introduction to Modern Software Engineering**. [*S.l.*]: Pearson, 2020.

SWAGGER. **About Swagger Specification**. 2022. Acesso em 02 de abr. de 2022. Disponível em: https://swagger.io/docs/specification/about/.

TSAI, C.-L. *et al.* Transmission reduction between mobile phone applications and restful apis. *In*: **Proceedings of the ACM Symposium on Applied Computing**. [*S.l.*]: Association for Computing Machinery, 2011. p. 445–450.

VINOSKI, S. Rpc and rest: Dilemma, disruption, and displacement. **IEEE Internet Computing**, IEEE, v. 12, n. 5, p. 92–95, 2008.

WOHLIN, C. *et al.* **Experimentation in software engineering**. [*S.l.*]: Springer Science & Business Media, 2012.

XIAO, Z.; WIJEGUNARATNE, I.; QIANG, X. Reflections on soa and microservices. *In*: **Proceedings of the 4th International Conference on Enterprise Systems (ES)**. [*S.l.*: *s.n.*], 2016. p. 60–67.