UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO UNIVERSITÁRIO DO NORTE DO ESPÍRITO SANTO
DEPARTAMENTO DE COMPUTAÇÃO E ELETRÔNICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RAFAEL EMERY CADE POLONI

AVALIAÇÃO DE DESEMPENHO PARA DIFERENTES ABORDAGENS DE COMUNICAÇÃO EM SISTEMAS DISTRIBUÍDOS

SÃO MATEUS, ES
2023

RAFAEL EMERY CADE POLONI

AVALIAÇÃO DE DESEMPENHO PARA DIFERENTES ABORDAGENS DE COMUNICAÇÃO EM SISTEMAS DISTRIBUÍDOS

Trabalho de Conclusão de Curso apresentado ao Departamento de Computação e Eletrônica do Centro Universitário do Norte do Espírito Santo da Universidade Federal do Espírito Santo como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Pedro Felipe do Prado

SÃO MATEUS, ES

2023

RAFAEL EMERY CADE POLONI

AVALIAÇÃO DE DESEMPENHO PARA DIFERENTES ABORDAGENS DE COMUNICAÇÃO DIFERENTES ABORDAGENS DE COMUNICAÇÃO EM SISTEMAS DISTRIBUÍDOS

Trabalho de Conclusão de Curso apresentado ao Departamento de Computação e Eletrônica do Centro Universitário do Norte do Espírito Santo da Universidade Federal do Espírito Santo como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

COMISSÃO EXAMINADORA

Prof. Dr. Pedro Felipe do Prado
Universidade Federal do Espírito Santo
Orientador

Prof. Dra. Luciana Lee
Universidade Federal do Espírito Santo

Prof. Dr. Esequiel da Veiga Pereira
Universidade Federal do Espírito Santo

RESUMO

Conforme abordagens relacionadas a sistemas distribuídos e arquitetura de microsserviços possuem cada vez mais espaço para se construir aplicações escaláveis, existem alguns questionamentos válidos a serem feitos, caso contrário, a abordagem pode trazer problemas para os desenvolvedores ou para a organização. Considera-se a comunicação entre os diferentes microsserviços uma parte fundamental de se implementar uma arquitetura de microsserviços, visto que como serão diversos recursos separados, a todo momento eles devem se comunicar, e esta comunicação deve ser conveniente dado o contexto do negócio. O presente trabalho visa avaliar diferentes abordagens para comunicação em um sistema distribuído a fim de garantir o melhor desempenho e decisões embasadas.

A análise consistirá em uma avaliação de desempenho por aferição, utilizando um protótipo e metrificando fatores considerados relevantes, juntamente com uma metodologia que realiza dois tipos de experimentos que representam diversos contextos e circunstâncias. A partir disso, será verificado se de fato existe uma abordagem mais escalável, e caso sim, qual abordagem, em relação à avaliação de desempenho proposta pode ser considerada a mais conveniente e qual o seu contexto, podendo ter influência tanto nas discussões comparativas sobre o tema, até em decisões dentro de organizações que estão trabalhando neste processo de desenvolvimento envolvendo uma arquitetura distribuída buscando maior escalabilidade.

Palavras-chave: sistemas distribuídos; microsserviços; comunicação em microsserviços; análise de performance; REST; RPC; broker de mensagem; avaliação de desempenho computacional; redes de computadores.

ABSTRACT

As approaches related to distributed systems and microservices architecture become increasingly popular for building scalable applications, there are some valid questions to be asked, otherwise the approach could cause problems for the developers or the organization. Communication between the different microservices is considered to be a fundamental part of implementing a microservices architecture, since there will be several separate resources, at all times they must communicate, and this communication must be convenient given the business context. This work aims to evaluate different approaches to communication in a distributed system in order to guarantee the best performance and informed decisions.

The analysis will consist of a performance evaluation by benchmarking, using a prototype and metricizing factors considered relevant, together with a methodology that carries out two types of experiments that represent different contexts and circumstances. From this, it will be verified whether there is in fact a more scalable approach, and if so, which approach, in relation to the proposed performance evaluation, can be considered the most convenient and what its context is, which can have an influence both on comparative discussions on the subject, and on decisions within organizations that are working on this development process involving a distributed architecture seeking greater scalability.

Keywords: distributed systems; microservices; microservice communication; performance analysis; REST; RPC; message broker; computational performance evaluation; computer networks.

Lista de Abreviaturas e Siglas

API	Application Programming Interfaces
Iaas	Infrastructure as as Service
JSON	Javascript Object Notation
XML	Extensible Markup Language
REST	Representational State Transfer
RPC	Remote Procedure Call
gRPC	Google Remote Procedure Call
HTTP	Hypertext Transfer Protocol
AMQP	Advanced Message Queueing Protocol
AWS	Amazon Web Services
Amazon EC2	Amazon Elastic Compute Cloud
BFF	Backend-For-Frontend
SGBD	Sistema de Gerenciamento de Banco de Dados

Lista de Figuras

Lista de Tabelas

Sumário

1. INTRODUÇÃO

Atualmente, com o ritmo de evolução da indústria de desenvolvimento de *software* e as tendências de mercado gradualmente se direcionando para a área de tecnologia e *software*, a demanda por escalabilidade cresce em ritmo similar. Com a necessidade de escalar aplicações, novas abordagens relacionadas ao desenvolvimento estão tendo cada vez mais espaço, como a abordagem de microsserviços, a qual neste contexto mencionado se destaca em relação à tradicional abordagem de arquitetura monolítica.

Segundo (NEWMAN, 2015), microsserviços são pequenos e autônomos serviços que trabalham em conjunto, e de acordo com (FOWLER, 2017), com microsserviços, uma aplicação pode ser facilmente escalada tanto horizontalmente quanto verticalmente, a produtividade e a velocidade do desenvolvedor aumentam e tecnologias antigas podem facilmente ser trocadas pelas mais recentes. Ademais, deve-se ressaltar o aspecto de tolerância a falhas e observabilidade, porém, é importante ressaltar que, embora existam diversos aspectos positivos relacionados à esta abordagem, também existem alguns *trade-offs* associados e pontos em que a modelagem e o desenvolvimento podem se tornar exponencialmente mais complexos, podendo ser problemático para os desenvolvedores, as aplicações e o contexto inserido. O presente trabalho, tem como objetivo abordar um aspecto que, dependendo da decisão tomada e ferramentas utilizadas, trazem uma grande contribuição e facilidade para as aplicações ou então grandes dificuldades, e se constitui como um dos aspectos fundamentais de uma arquitetura baseada em microsserviços: a comunicação entre os diferentes microsserviços e suas diferentes abordagens.

Com uma premissa na qual representa um contexto de negócio em que as aplicações precisam se comunicar entre si (ou trocar mensagens) da forma mais rápida possível e com menos falhas, visto que podem existir diversas tratativas associadas, um protótipo será desenvolvido. A partir de uma avaliação de desempenho, utilizando a técnica de aferição por meio do protótipo construído e considerando métricas específicas, serão obtidas informações para entender, considerando os parâmetros e a metodologia utilizada, qual (ou quais) as abordagens mais convenientes para o contexto. Assim, existe a possibilidade de tomar decisões mais assertivas em relação a como os microsserviços devem se

comunicar e contribuir para a construção de uma aplicação de fato escalável e tolerante a falhas.

1.1 Contextualização

Com os recentes avanços na área de desenvolvimento e a crescente demanda por escalabilidade, uma abordagem focada em sistemas distribuídos para as soluções foi, gradualmente, obtendo mais visibilidade e uso no mercado e organizações e times de desenvolvimento optam pela mesma, mais especificamente a abordagem de microsserviços. Apesar de sistemas feitos utilizando esta abordagem estarem sendo desenvolvidos, não é possível considerá-la como algo absoluto e que não possui problemas. Deve-se considerar que a comunicação entre as partes de um sistema distribuído é um aspecto fundamental para o bom funcionamento do sistema e a falta de eficiência neste aspecto pode se mostrar um problema que impacta diretamente na escalabilidade e no cumprimento dos requisitos, ou seja, caso não seja pensado e desenvolvido de uma forma eficiente, podem ocorrer problemas de desempenho, persistência e confiabilidade nas aplicações, e até observabilidade.

Existem diversas abordagens e ferramentas para a comunicação entre microsserviços que servem a diferentes propósitos e trabalham especificamente em diversos pontos, e cabe à organização tomar decisões coerentes de quais utilizar e quando utilizá-las. Acredita-se que este momento de decisão em relação à, basicamente, como resolver este problema, pode trazer uma certa dificuldade às partes interessadas pois para uma decisão ser tomada, de forma que a solução seja de fato eficaz, devem ser levados em consideração parâmetros convenientes de acordo com o contexto em que o projeto se encontra. Dito isso, a partir do conhecimento de um conjunto de conceitos, métricas e práticas, considerando principalmente desempenho, a decisão pode ser tomada de forma que seja de fato eficiente.

1.2 Hipóteses

Para ter melhor orientação sobre as diretivas que o presente trabalho deve seguir, foram elaboradas hipóteses relacionadas com a proposta do presente trabalho, áreas de estudo, conceitos aplicados, e principalmente a metodologia utilizada. As hipóteses que irão nortear a metodologia são:

- Identificar qual abordagem de comunicação entre microsserviços, dentre as propostas, possui melhor desempenho de acordo com as métricas que foram adotadas e observadas.
- Dado o contexto apresentado e arquitetura proposta e desenvolvida, identificar qual método de comunicação pode ser considerado como mais eficaz e coerente para ser implementado.
- Experimentar e analisar o impacto que diferentes cargas de trabalho possuem no desempenho do sistema que está sendo aferido, e quais fatores de cargas de trabalho podem ser mais relevantes.
- Identificar e entender possíveis gargalos no sistema aferido, assim como estratégias para mitigá-los.

1.3 Objetivos

Analisar, por meio da metodologia e dos parâmetros definidos, as melhores abordagens para comunicação entre microsserviços, com a finalidade de auxiliar em decisões sobre onde e quando utilizá-las, e caso haja algum tipo de gargalo no sistema aferido, entender as razões e buscar formas de mitigá-lo. Para isso, será utilizado o método de aferição a partir de um protótipo que simula um contexto em que existe a necessidade de trocas de mensagens rápidas, ou seja, onde um fator decisivo para escalabilidade e cumprimento dos requisitos é o desempenho. O protótipo irá conter microsserviços que se comunicam entre si através de algumas interfaces que serão definidas posteriormente.

Como objetivos específicos, será analisada e avaliada a metodologia a partir de um modelo de aferição por protótipo, além da usabilidade deste protótipo e o quanto ele consegue refletir uma situação real, validando ferramentas de avaliação de desempenho e as métricas que irão ser utilizadas. Além disso, outro objetivo específico relevante a ser ressaltado é o levantamento bibliográfico da área de estudo e também projetos similares, o que permite maior embasamento para as

áreas nas quais o tema possui relação direta e indireta e melhor o entendimento sobre metodologias de avaliação de desempenho.

1.4. Organização

O presente documento é organizado em seções de maneira que descrevem o processo que foi adotado pelo autor para realizar a avaliação de desempenho. Primeiramente, foi realizada uma contextualização sobre o tema e os objetivos do presente trabalho, como evidenciado nas seções anteriores. A seção posterior representa o referencial teórico, que possui como objetivo um levantamento bibliográfico das áreas de estudo e de trabalhos relacionados, que possuem grande utilidade.

As três seções seguintes ao referencial teórico são referentes à metodologia, experimentos simples e experimentos completos, respectivamente. Entende-se que em uma avaliação de desempenho, a metodologia adotada é tida como fundamental e tem grande impacto nos resultados e nas conclusões obtidas. Por fim, as seções para conclusões obtidas e trabalhos futuros, as quais possuem os *insights* obtidos durante o processo de avaliação de desempenho e possíveis caminhos de pesquisa para trabalhos futuros.

2. REFERENCIAL TEÓRICO

2.1 Redes de computadores

Redes de computadores é considerada uma área de estudo essencial para o entendimento de sistemas distribuídos, já que ambas as áreas se relacionam em diferentes aspectos e pode-se afirmar que sistemas distribuídos geralmente são dependentes de uma rede de computadores para funcionar, pois como o próprio nome sugere, está distribuído e pode estar geograficamente distante, e portanto, uma rede possui um papel fundamental.

Inicialmente, será definido alguns pontos básicos de redes de computadores, o conceito de *internet* e seu funcionamento básico, o conceito de protocolo, e por fim, o modelo de cinco camadas da *internet*, que é amplamente utilizado para fins didáticos e ter um melhor entendimento sobre a rede, seus protocolos e camadas.

A *internet*, termo amplamente utilizado pela sociedade ao fazer referência a uma rede de computadores global, é uma rede de computadores que interconecta centenas de milhões de dispositivos de computadores ao redor do mundo. No jargão de redes, nomeamos esses dispositivos de computadores (TV, *laptops*, celulares, entre outros) como sistemas finais. Os sistemas finais são conectados entre si por meio de enlaces de comunicação e comutadores de pacotes, que podem ser de diferentes meios, como fios de cobre, fibra óptica, entre outros. Além disso, cada enlace possui uma taxa de transferência diferentes, medida em bits por segundo. Basicamente, os pacotes (informações que estão sendo transferidas) são enviados a um sistema final através dos enlaces, e também é importante ressaltar que existem os ISPs, que são os Provedores de Serviços de *Internet*, os quais são os meios que os sistemas finais usam para acessar a *internet* (KUROSE, 2012).

Além desta breve descrição em relação aos diferentes sistemas finais e sua conexão, outro conceito importante para redes são protocolos. Como uma definição informal, os protocolos são fundamentais para o funcionamento de redes pois definem regras específicas para a comunicação entre duas partes de uma rede, de acordo com o procedimento e a camada em específico. De acordo com (KUROSE, 2012) um protocolo define o formato e a ordem das mensagens trocadas entre duas ou mais entidades comunicantes, bem como as ações realizadas na transmissão e/ou recebimento de uma mensagem ou outro evento.

Para entender o melhor funcionamento da rede em um sentido geral e seus respectivos protocolos, deve-se considerar os modelos de camadas, e no presente trabalho será abordado o modelo de 5 camadas, ou como (KUROSE, 2012) define, a pilha de protocolos da *Internet* de cinco camadas, representados na Figura 1.

Figura 1 - Pilha de protocolos da Internet de cinco camadas.

Fonte: (KUROSE, 2012).

Dadas as camadas existentes, irão ter maior importância somente às camadas mais externas - sendo considerados as camadas mais acima na pilha - e seus respectivos protocolos, entendendo sua relevância para as abordagens de comunicação entre microsserviços, que acontecem nas camadas mais externas da rede.

A camada de transporte é a primeira camada na qual o desenvolvedor de aplicação pode utilizar de alguma forma, e tem como objetivo, resumidamente, lidar com a transferência de pacotes entre pontos de uma rede, com confiabilidade e tratamento de erros, ou seja, ela deve entregar uma mensagem sem perder nenhuma informação fundamental. Podem ser citados 3 protocolos para a camada de transporte: o TCP, o qual junto com o IP (formando o TCP/IP) é um padrão difundido de comunicação, o UDP e o RTP, que suporta transmissão em tempo real.

Por fim, a camada que possui um nível mais alto e que se mostra fundamental para o entendimento das abordagens do presente projeto, é a camada de apresentação. (KUROSE, 2012) cita que é a camada onde residem as aplicações de rede e seus protocolos. De fato, apresenta-se como a camada em que o desenvolvedor de aplicações tem maior contato e que possui grande relação com o tema do estudo, sistemas distribuídos. Alguns exemplos de protocolos de aplicação são: DNS (*Domain Name System*), que define um sistema para nomes de domínio na rede, HTTP e FTP. É importante ressaltar que o HTTP é o protocolo o qual faz parte do padrão REST, que será implementado e analisado.

2.2 Sistemas Distribuídos

De acordo com (COULOURIS, 2013) um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagem, e para (TANENBAUM, 2008), é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente.

Ambas as definições apresentadas refletem aspectos básicos de sistemas distribuídos, ou seja, diversos elementos computacionais, podendo ser uma parte do *hardware* ou um processo, interoperáveis se comunicando entre si para realizar uma determinada ação e estes diversos recursos sendo apresentados aos usuários como um único sistema, representando assim alguns aspectos básicos como interoperabilidade e transparência. Além disso, características como escalabilidade e recursos facilmente acessíveis são fundamentais para se constituir um sistema distribuído.

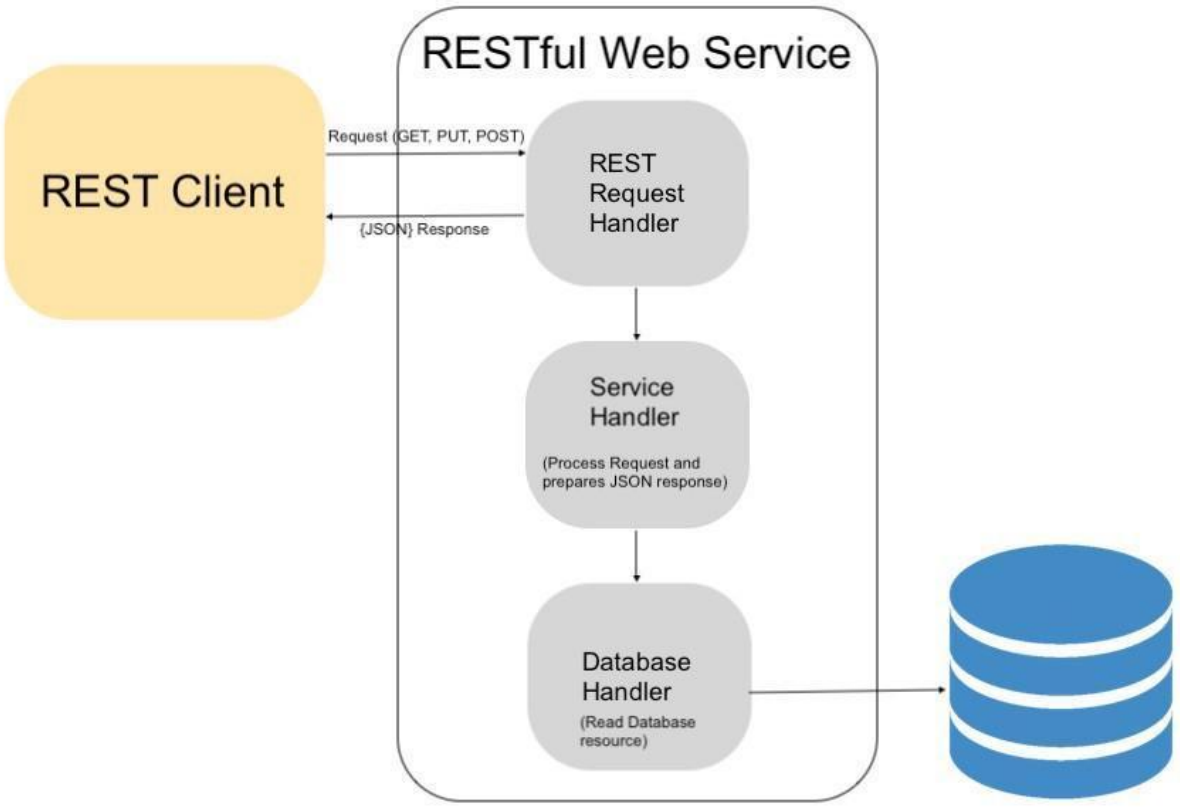
Por fim, como (COULOURIS, 2013) aponta, os computadores, ou recursos, de uma maneira geral, são interligados por meio de uma rede. Estabelecendo uma breve relação entre redes de computadores e sistemas distribuídos, entende-se que a área de redes de computadores engloba as cinco camadas citadas anteriormente no presente trabalho, além de todo o *software* e *hardware* de redes envolvidos, enquanto em sistemas distribuídos, o foco principal são as camadas de transporte e aplicação, como igualmente definido anteriormente, e seus respectivos protocolos. Desse modo, em redes de computadores existem estudos e conceitos que abrangem como todos os computadores irão ter suas interações, enquanto nos sistemas distribuídos o foco principal está na aplicação em questão que está sendo desenvolvida.

2.3 Comunicação entre sistemas distribuídos

A comunicação entre diferentes recursos, considerando recursos como máquinas, processos, *threads*, entre outros, está no coração de um sistema distribuído, e quando trata-se de aplicações que estão distribuídas dentro de uma rede, podem ser necessárias abordagens diferentes para trabalhar o fluxo de informações.

Inicialmente, e antes de maior aprofundamento em outras abordagens de comunicação, deve-se pensar no padrão convencional cliente-servidor. Nele, resumidamente, existe uma aplicação que atua como cliente e outra aplicação que atua como o servidor, na qual um cliente irá fazer uma requisição ao servidor - que geralmente é por meio do protocolo HTTP - e esperar uma resposta. O servidor irá processar a requisição, realizar as instruções necessárias, como validações, tratativas relacionadas a formatos, interação com banco de dados, e retornar ao cliente, como mostrado na Figura 2.

Figura 2 - Representação de uma arquitetura cliente-servidor.



Fonte: https://medium.com/@subhangdxt/beginners-guide-to-client-server-communication-8099cf0ac3af

A abordagem utilizando cliente-servidor é, de fato, muito difundida, principalmente na área de desenvolvimento *web*, porém em sistemas distribuídos, em muitos casos a comunicação não segue este padrão, visto que é restrito em relação a outros padrões e contextos. Antes de prosseguir para os próximos padrões

de comunicação, é importante considerarmos alguns conceitos fundamentais (TANENBAUM, 2008):

- Camadas de rede e protocolos: como mencionado anteriormente, através do modelo de cinco camada previamente apresentado, entende-se melhor a rede como um conjunto de camadas com diversos protocolos, e quando nos aprofundamos à comunicação dentro de um sistema distribuído, entendemos que um ponto, ou micro serviço, irá enviar alguma instrução ou informação a outro, e isso de fato acontecerá através dos determinados protocolos que estas camadas de redes possuem, como o HTTP, TCP/IP, entre outros. Deve-se ressaltar que, para cada abordagem de comunicação podem existir diversos protocolos sendo utilizados, como o padrão REST que utiliza o HTTP.
- Comunicação persistente e transiente: considerando um exemplo em que deve haver troca de mensagens entre dois clientes, passando por um servidor, como uma troca de *e-mails*. Na comunicação persistente, a mensagem que foi entregue ao servidor ficará armazenada em um *middleware* durante todo o processo até que o destinatário desta mensagem a receba, e este *middleware* de fato irá possuir a capacidade de armazenamento das mensagens trocadas, e caso aconteça alguma falha no processo, a mensagem e sua referência estarão guardadas. Na comunicação transiente, a mensagem permanece na aplicação durante o tempo em que as aplicações estiverem rodando, e caso aconteça alguma interrupção no processo, a mensagem será descartada. Os conceitos de persistente e transiente estão presentes em abordagens de comunicação entre microsserviços, visto que o padrão REST, por exemplo, se mostra transiente, enquanto uma abordagem focada em troca de mensagens por *broker*, utiliza-se da persistência, garantindo assim a tolerância a falhas.
- Comunicação síncrona e assíncrona: se tratando do mesmo contexto acima, existe que na comunicação síncrona, o remetente “aguarda” ou simplesmente é bloqueado até saber que sua requisição foi aceita e processada. No caso da comunicação assíncrona, não existe esta “espera” mencionada e ele continua seus processos até receber sua resposta. Assim como a comunicação persistente e transiente, os aspectos síncronos e assíncronos estão presentes em diversas abordagens de comunicação entre

microsserviços. Existem situações em que, para o desenvolvedor, um padrão pode funcionar de forma síncrona e assíncrona dependendo da sua implementação.

A partir dos aspectos apresentados, irá ser descrito de forma mais aprofundada as abordagens, seus respectivos protocolos e ferramentas relacionadas, para assim, entender melhor suas aplicações em um contexto de arquitetura distribuída baseada em microsserviços. Microsserviços interagem uns com os outros através da rede, via RPC ou troca de mensagens via *endpoints* (utilizando o padrão HTTP + REST) ou através de um *broker* de mensagens. As três abordagens definidas e descritas (FOWLER, 2017):

- RPC (*Remote Procedure Call*): resumidamente, se trata de um processo de uma máquina chamando o processo de outra máquina, onde são passados parâmetros e se obtém um resultado. A Figura 3 demonstra o funcionamento básico entre uma máquina cliente e uma máquina servidor. Atualmente, é muito utilizado no mercado o gRPC, uma implementação para o RPC desenvolvida pela Google.

Figura 3 - Comunicação via RPC entre um cliente e um servidor.

Fonte: (TANENBAUM, 2008)

- Troca de mensagens (HTTP + REST): utiliza o protocolo HTTP 1.1, que como descrito anteriormente, atua na camada de aplicação da rede, e irá enviar requisição e receber respostas por meio de *endpoints* que seguem o padrão

REST. Dado o contexto, é importante considerar quatro métodos feitos pelas requisições de um cliente: GET, POST, PUT/PATCH e DELETE. Outro aspecto importante é o envio e formatação através de formato JSON (JavaScript Object Notation). Embora seja considerado simples e convencional de ser utilizado, é síncrono e bloqueante - porém pode haver soluções paliativas durante a implementação.

- Troca de mensagens (*broker*): basicamente, um micro serviço envia dados (mensagem) pela rede para um *broker* que irá rotear para outros microsserviços. Pode ser *pub/sub*, no qual clientes assinam, ou “escutam”, um determinado fluxo de dados (ou tópico) e recebem mensagem sempre que algo é publicado no mesmo, ou solicitação-resposta, no qual o cliente envia uma solicitação para um *broker* que irá respondê-lo com informações. Nos estudos e desenvolvimento do presente trabalho, a abordagem *pub/sub* terá maior ênfase.

2.4 Avaliação de desempenho

O conceito de avaliação de desempenho está bastante presente no nosso dia a dia. Um exemplo da trivialidade mencionada nesta área seria avaliar a eficiência entre dois carros, considerando, como por exemplo, qual faz maior distância a cada litro de combustível, ou quantos clientes foram atendidos por hora em um restaurante. Naturalmente, a avaliação de desempenho em sistemas computacionais trabalha especificamente dentro do contexto de sistemas computacionais. Um aspecto importante que devemos ressaltar é que possui maior ênfase em mensurações quantitativas do que qualitativas. Para ilustrar o que representa uma medição de um parâmetro quantitativo e qualitativo, podemos pensar em tempo médio de resposta de uma requisição (quantitativa) e usabilidade de uma interface (qualitativa).

Sistemas computacionais de maneira geral realizam diversas tarefas, de diversas formas, como um SGBD (Sistema de Gerenciamento de Banco de Dados), que consulta e escreve dados, redes que transmitem e recebem informações, servidores, entre outros, e um sistema computacional recebe requisições (ou instruções) e realiza uma ação (também pode ser que retorne um *feedback* para alguma parte interessada).

De acordo com (JAIN, 1991), deve-se conhecer um sistema intimamente para avaliar seu desempenho, e é necessária em todos estágios do ciclo de vida de um sistema e/ou computador. Métricas são definidas e assim ter resultados aferidos. Existem três métodos principais para avaliação de desempenho, sendo aferição, modelagem e simulação, e portanto cada um possui seus respectivos contextos e *trade-offs* para serem aplicados da melhor maneira possível e ter maior valor para uma tomada de decisão mais correta.

A aferição, também definida como medição, terá maior foco pois foi o escolhido para ser utilizado no presente projeto. Consiste em medir, através de testes de carga e trabalho e a partir de métricas pré estabelecidas, um determinado sistema. Um ponto importante, que pode ser dito como negativo, a se considerar sobre o método de aferição é a necessidade de existência do sistema ou de um protótipo. Como pontos positivos, a quantidade de instrumentos para realizá-la, a facilidade de aprendizado e a qualidade dos resultados obtidos podem ser citados.

A modelagem consiste na retirada de aspectos que não são considerados importantes e posterior identificação dos componentes relevantes para a análise. Assim, é construído um modelo matemático que irá garantir previsibilidade no comportamento do sistema, baseando-se em componentes e entradas e saídas. A simulação é uma técnica que consiste em uma previsão de desempenho de um sistema que pode não existir, ao contrário da aferição, ou pode existir. É feita uma simulação para prever o comportamento do sistema. Embora seja a menos custosa das técnicas, seus resultados costumam ser menos confiáveis.

É importante ressaltar que podem ser realizadas combinações entre as técnicas de modelagem citadas anteriormente, o que pode garantir maior confiabilidade dos resultados obtidos, todavia o presente trabalho irá considerar somente a aferição a partir de um protótipo dada a sua aplicabilidade neste contexto.

2.5 Trabalhos relacionados

Como parte do levantamento bibliográfico relacionado à avaliação de desempenho e redes de computadores, sistemas distribuídos, arquitetura de microsserviços e comunicação entre microsserviços, foram encontrados diversos trabalhos. Os objetivos principais foram o entendimento de metodologias já existentes e utilizadas para o desenvolvimento e testagem, além de abordagens e

ferramentas específicas para comunicação entre microsserviços presentes na literatura acadêmica.

No trabalho de (OLIVEIRA E SAMPAIO, 2021) foi realizada uma análise da eficiência da comunicação de serviços utilizando REST ou gRPC. O projeto serviu para uma prova de conceito a um problema real e consistiu na comparação entre REST e gRPC através de testes de desempenho com diferentes cargas, considerando como métrica principal o tempo de resposta. Concluiu-se que o gRPC apresentou melhores resultados, e a melhora foi proporcional à quantidade de registros utilizados nos testes de desempenho propostos. É importante ressaltar que foi utilizado o conceito de cliente-servidor tanto para REST, quanto para gRPC.

Em (HONG, YANG E KIM, 2018), foi apresentado uma comparação entre as abordagens REST e o RabbitMQ (representando o *broker* de mensagens). Ao contrário do anterior, este estudo delimitou seu contexto a microsserviços e considerou o REST e o RabbitMQ como a camada de *middlewares* de comunicação (MOM). A avaliação do desempenho foi feita por meio de um teste de carga, onde um grande número de usuários enviando requisições simultâneas. Constatou-se que o RabbitMQ foi uma abordagem mais eficiente neste caso.

Na pesquisa de (BAGASKARA, SETYORINI E WARDANA, 2020) foi apresentada uma perspectiva interessante ao comparar diferentes *brokers* de mensageria em um ambiente de computação em névoa. Foram escolhidos o Apache Kafka e o RabbitMQ e ambos foram utilizados para testes em um ambiente de *Fog Computing* (computação em névoa). A partir de alguns parâmetros, foram feitas algumas considerações relativas a ambas ferramentas. Esta referência é relevante para o presente trabalho pois oferece algumas perspectivas em relação aos *brokers* de mensageria, tal qual sua aplicação em um ambiente diferente.

O estudo de trabalhos relacionados se mostrou fundamental pois mostrou referências e direcionamentos em relação ao que será feito, tanto em ferramentas e abordagens utilizadas para microsserviços, quanto em avaliações de desempenho, onde foi notado um padrão em testes de carga para medição (aferição) de desempenho. Além disso, fornecerá uma base para serem decididas as ferramentas para a construção do protótipo e dos testes, como por exemplo, softwares de análise de desempenho e *brokers* de mensageria.

3. METODOLOGIA

Nesta seção, será abordado o processo de desenvolvimento, arquitetura proposta e uma resumida descrição do método que foi utilizado no presente trabalho. É importante destacar que as escolhas de desenvolvimento e arquitetura foram norteadas pelo contexto em que o presente trabalho se coloca: uma arquitetura distribuída com alta granularidade, ou seja, microsserviços que sejam o mais objetivo possível, dados os requisitos de negócio. Além disso, deu-se preferência para ferramentas de código aberto, objetivas e que estão em alta para desenvolvimento de microsserviços.

3.1 Descrição da metodologia utilizada

Para realizar a avaliação de desempenho, adotou-se uma estratégia de aferição, e foi feita em um protótipo construído, que como mencionado, tem como direcionamento reproduzir o contexto proposto. Inicialmente, foram definidas as ferramentas e linguagem de programação a serem utilizadas, a definição da proposta de arquitetura e o desenvolvimento de fato do protótipo. Juntamente com isso, e partindo do referencial teórico acerca de avaliação de desempenho de sistemas computacionais proposto por (JAIN, 1991), foram definidos os fatores e níveis e as métricas que serão utilizadas.

Após as definições iniciais e o desenvolvimento, iniciaram-se os experimentos, que serão descritos nas duas seções seguintes, visto que possuem grande relevância para o valor científico do presente trabalho. Primeiramente, foi feito o planejamento fatorial simples (ou experimentos simples), que possui definida neste documento, a sua análise e respectivos resultados. Após os experimentos simples, ocorreu uma delimitação e alteração de fatores e níveis para se executar a etapa final de experimentos, a qual as descrições e justificativas das escolhas também estão explícitas. Por fim, foi executado o planejamento fatorial completo, com apenas dois fatores e dois níveis para cada fator, também nomeado como planejamento fatorial completo, com sua análise e respectivos resultados. O planejamento fatorial completo, como considerado como o conjunto de experimentos mais complexo executado, proveu os resultados necessários para as conclusões e trabalhos futuros.

De forma breve, entende-se o método utilizado como uma maneira de se avaliar o desempenho de *software*, considerando os cenários mais realistas possíveis (de acordo com o contexto proposto), e durante o processo, ir obtendo conhecimentos que agreguem e se complementam, para enfim obter conclusões mais objetivas e completas sobre o real desempenho de um determinado sistema. Conhecer e aplicar a metodologia de avaliação de desempenho proposta por Raj Jain e utilizada no presente trabalho, é entender de maneira extremamente coerente, onde e quando seu sistema pode ter um bom ou mal desempenho, e incluir esta prática no processo de desenvolvimento de *software* pode ser benéfico, visto que é possível fazer melhorias e mitigar erros e riscos de forma mais assertiva.

3.2. Desenvolvimento

A seguir, serão descritas as escolhas técnicas realizadas para o desenvolvimento do projeto e execução dos testes.

3.2.1 SGBD: PostgreSQL

Para lidar com o banco de dados, optou-se por um banco de dados relacional, e o sistema gerenciador de banco de dados (SGBD) escolhido foi o PostgreSQL, que possui como características ser um SGBD popular, com excelente desempenho e de código aberto. O PostgreSQL possui muitas funcionalidades úteis para a modelagem e desenvolvimento, além de possuir diversas interfaces com diferentes ambientes, ferramentas e linguagens de programação que são validados e com grande confiabilidade.

3.2.2 *Message Broker*: RabbitMQ

Para implementar o método de *message broker* (ou broker de mensageria), optou-se pelo RabbitMQ. Assim como o SGBD adotado, é de código aberto, e, em resumo, possui o objetivo principal de atuar como um intermediário entre as aplicações que irão publicar a mensagem e as aplicações que irão consumi-la, que são comumente chamadas de *publisher* e *consumer*, respectivamente. Possui como características a simplicidade e objetividade, desempenho confiável, e facilidade de configuração, além de todas as funcionalidades básicas que um serviço de fila, ou *message broker*, deve possuir, como a definição de tópicos e filas.

É importante ressaltar que o RabbitMQ implementa o AMQP (*Advanced Message Queuing Protocol*) e pode ser estendido para implementar diversos outros protocolos, e também possui uma interface sólida e confiável com diversas linguagens de programação modernas.

3.2.3 Linguagem de programação: Golang

Golang, ou simplesmente Go, é uma linguagem de programação, desenvolvida dentro da Google em 2007, que, segundo (GO, 2023), permite ao desenvolvedor construir aplicações simples, seguras e escaláveis. Com Golang, podem ser construídas aplicações de nuvem e rede, interfaces de linha de comando, aplicações *web*, entre outros tipos.

As principais características do Go são a sua simplicidade e facilidade de uso, com uma sintaxe simples, objetiva e organizada, compatibilidade, de código aberto e compilada. Embora seja uma linguagem de programação relativamente nova, é uma grande tendência nos últimos anos, principalmente em desenvolvimento *web* e arquiteturas distribuídas, e possui um ecossistema de pacotes e comunidade coerente e constante. Como todas linguagens de programação e ferramentas, também existem características negativas, como o próprio fato de ser relativamente nova comparada com outras linguagens estabelecidas, como C, C++ e Java, o que resulta em algumas dificuldades e falta de abstrações que seriam comuns em outros ambientes, não implementa o paradigma orientado à objetos, embora possua algumas funcionalidades que façam ser parecido (no ponto de vista do desenvolvedor), e dificuldade de depuração e análise de problemas.

Para o desenvolvimento do protótipo, a linguagem se mostrou extremamente útil pela sua velocidade, simplicidade e característica escalável, além de bibliotecas confiáveis utilizadas para funcionalidades. É importante ressaltar que o fato de Golang ser uma tendência em desenvolvimento de arquiteturas distribuídas, teve uma contribuição relevante para o contexto, presente projeto e suas respectivas análises e propostas.

3.2.4 *Framework Web*: Fiber

Para o desenvolvimento de APIs, optou-se pelo Fiber, um Framework para Go, inspirado no Express (Node.js), que foi construído em cima de um pacote chamado Fast HTTP. O *Fast HTTP* é um pacote que implementa um servidor HTTP,

e segundo a documentação do repositório oficial (VALIALKIN, 2023), possui como característica principal o desempenho e escalabilidade, sendo até dez vezes mais rápido que o pacote básico *net/http* do Go, e com o menor gastos de recursos possíveis. Logo, como o Fiber também foi desenvolvido com este objetivo, além de um fator fundamental para o desenvolvimento e a aderência da comunidade: a facilidade de configuração e uso e simplicidade.

3.2.5 Pacote para implementar o gRPC: gRPC-Go

O gRPC é um *framework* extremamente popular e de código aberto com excelente desempenho para RPC, e o pacote gRPC-Go trata desta implementação em Golang. Como mencionado, é extremamente popular, sendo considerado praticamente uma convenção de uso no que tange ao RPC em Go, ou até em outras linguagens em ambiente. Com o pacote, é possível definir um arquivo *.proto*, uma extensão para arquivos que contém *Protocol Buffers* (GOOGLE, 2023), e que irá conter todas as informações necessárias e os contratos (fortemente tipados) das mensagens, e partindo deste arquivo, são gerados os clientes e servidores, cabendo ao desenvolvedor somente implementá-las de fato e executar as chamadas.

3.2.6 Pacote para interface com o RabbitMQ: amqp

O amqp, é um cliente em Go para o RabbitMQ e que é mantido pela sua própria equipe. Segundo seu repositório oficial no GitHub, possui como objetivo prover uma interface objetiva e funcional que representa o protocolo AMQP implementado no RabbitMQ. A sua implementação é simples, porém extremamente útil, e é utilizado nas aplicações que publicam mensagens e que consomem. É importante ressaltar que o pacote utilizado se tornou depreciado em 2022, tendo como alternativa (pelos próprios desenvolvedores) o amqp091-go, porém o protótipo não foi impactado negativamente por isso, além do pacote depreciado possuir mais documentação e artigos disponíveis.

3.2.7 Pacote para interface com o PostgreSQL: database/sql e pq

Para lidar com o SGBD adotado e o banco de dados foi utilizado o pacote básico para interface com banco de dados *database/sql* com o adaptador *pq* para o PostgreSQL. O pacote *database/sql* provê uma interface genérica para banco de dados relacionais (GO, 2023), sendo necessário um pacote “auxiliar” para interagir

com o PostgreSQL, logo, é importante ressaltar que, a nível de código, o pacote mais relevante (que mais foi importado) é o *database/sql,* sendo o *pq* importado no momento de se conectar com o PostgreSQL dentro dos servidores.

3.2.8 Ferramenta utilizada para executar testes de carga: Locust

Para serem realizados os testes de carga (ou testes de stress), optou-se pelo Locust, uma ferramenta de código aberto que possui como objetivo prover uma interface simples e adaptável para testes de carga. Para isso, o Locust possui como característica ser executado em *scripts* em Python, e só é necessário ter instalado o python 3 e o pip (gerenciador de pacotes) no python. Isso permite que usuários tenham maior liberdade para definir os testes e não ficarem restritos a uma determinada interface ou arquivo XML. Com o Locust, foi possível executar testes de carga no sistema (protótipo) aferido e obter métricas relevantes, alterando, por exemplo, a quantidade de requisições feitas e a quantidade de usuários simultâneos.

3.2.9 Ferramenta utilizada para executar as aplicações: Docker

Para garantir maior isolamento e flexibilidade para as aplicações no ambiente de testes, optou-se por utilizar a ferramenta Docker, uma plataforma de código aberto para desenvolvimento e execução de aplicações em contêineres. Segundo (DOCKER, 2023), um contêiner é uma unidade padrão de *software* que empacota o código e todas as suas dependências para que o aplicativo seja executado de forma rápida e confiável de um ambiente de computação para outro. Juntamente com o Docker, foi utilizado o Docker Compose, o qual é uma ferramenta que permite criar, configurar e gerenciar múltiplos contêineres através de arquivos YAML.

Os contêineres e as máquinas virtuais têm benefícios semelhantes de isolamento e alocação de recursos, mas funcionam de maneira diferente porque os contêineres virtualizam o sistema operacional em vez do *hardware*. Os contêineres são mais portáteis e eficientes (DOCKER, 2023).

3.3. Arquitetura e características do sistema

A arquitetura proposta consiste em representar o contexto do problema em que o presente trabalho se propõe a avaliar, o qual para se executar os fluxos de negócios e implementar as funcionalidades necessárias, podem existir diversas

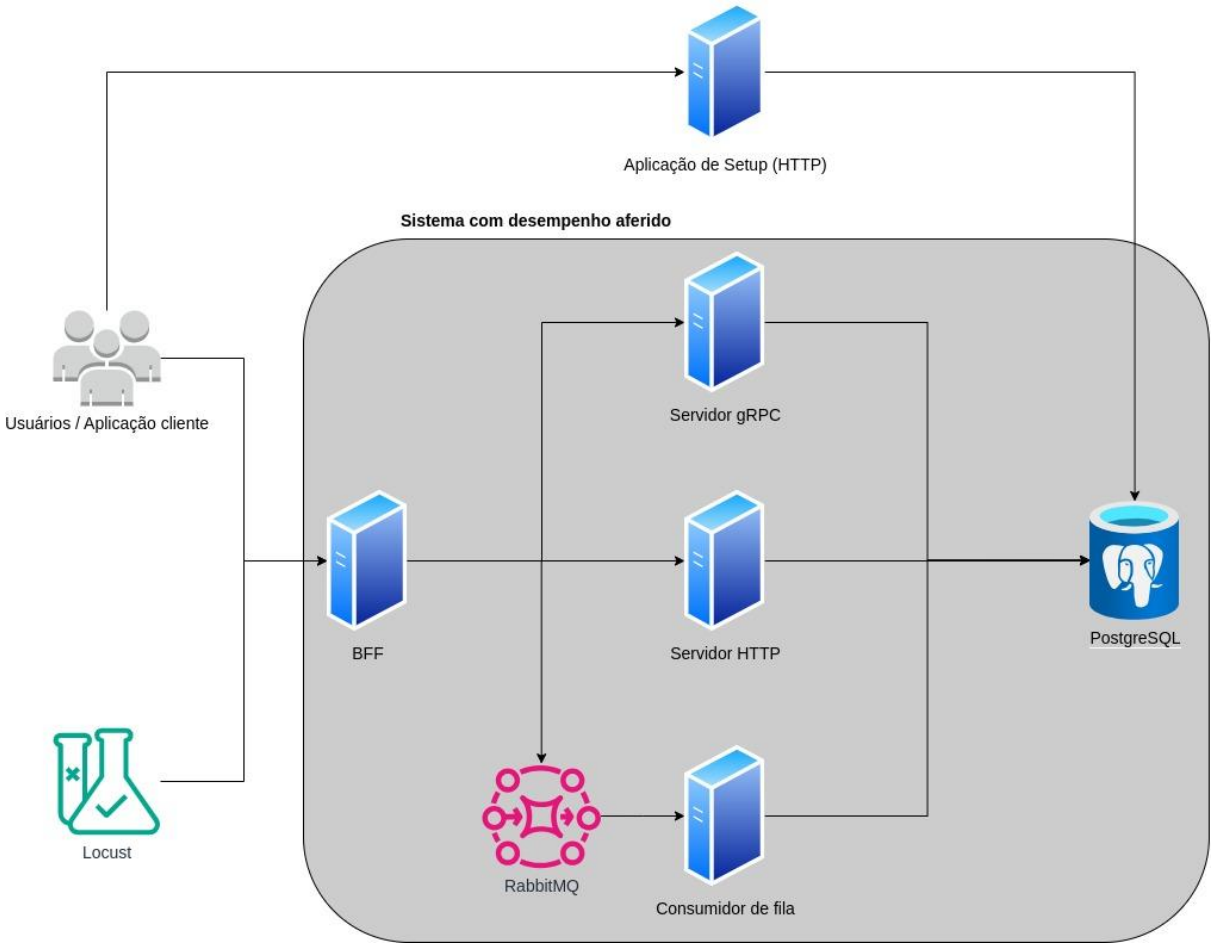
aplicações, como APIs ou servidores que se comunicam entre si, com o banco de dados ou aplicações externas, ou consumidores, que escutam uma determinada fila de eventos e executam ações.

3.3.1 Utilização do padrão de projeto BFF

É importante ressaltar que utilizou-se o padrão de BFF (*Backend-For-Frontend*), um padrão de projeto que se utiliza de um serviço chamado BFF para cada tipo de cliente, adaptando as requisições entre os diferentes tipos de clientes e esconde a complexidade do *backend* dos clientes em questão (ALKHODARY. 2022). Funciona de maneira semelhante ao *API Gateway*, que consiste em um único ponto de entrada para o sistema, porém optou-se pelo BFF pois pode funcionar de maneira mais flexível de acordo com múltiplos clientes e serviços que interagem. O padrão *API Gateway* comumente é utilizado para realizar a comunicação entre um único cliente com diversas outras APIs, e no protótipo apresentado, as possibilidades de comunicação são maiores.

3.3.2 Arquitetura

Figura 4 - Arquitetura proposta para representar o sistema distribuído.



Fonte: elaborada pelo autor.

A Figura 4 ilustra a arquitetura do protótipo e destaca o sistema com desempenho aferido. Primeiramente, foi desenvolvido um BFF, que consiste em uma *API RESTful* desenvolvida utilizando o Fiber, que possui *endpoints* que podem ser utilizados pelos usuários e principalmente uma aplicação *client* (*frontend*), ou pela ferramenta de testes de carga adotada, Locust, que simula múltiplos usuários em aplicações Web realizando requisições de acordo com os parâmetros. No BFF foram implementadas três formas de comunicação distintas, para cada um dos servidores avaliados.

Foram desenvolvidas três aplicações, que rodam em portas e separadas e de maneira diferente, para receber as interações realizadas pelo BFF - e consequentemente pelos usuários. O “servidor HTTP” implementa a abordagem de comunicação REST, que representa uma API RESTful, o “servidor gRPC”

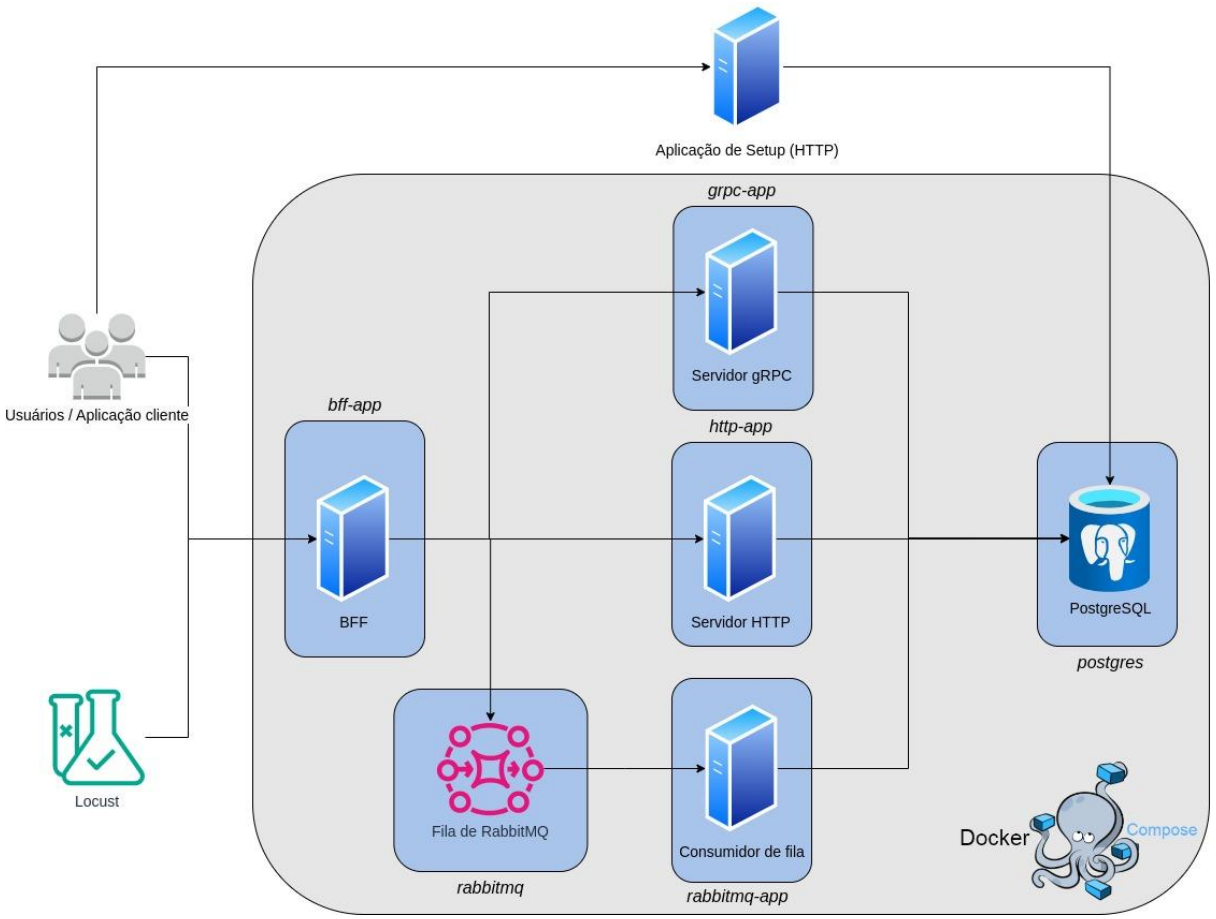
implementa a abordagem de gRPC, que expõe as chamadas necessárias e o consumidor de fila implementa a abordagem de serviço de mensageria. É relevante entender que esta última abordagem implementada não segue o mesmo padrão das outras duas, tendo necessidade de um serviço de fila (RabbitMQ) agindo como intermediário, o que se mostrará relevante para as análises do presente trabalho. Por fim, todas as aplicações interagem com um banco de dados PostgreSQL.

A “Aplicação de *Setup* (HTTP)” representada na imagem, foi adotada simplesmente uma facilidade no desenvolvimento que foi adotada, que oferece algumas funcionalidades principalmente em relação ao banco. Resumidamente, é uma API RESTful com alguns *endpoints* para auxiliar nos experimentos e garantir resultados melhores, com três funcionalidades: retornar quantos registros existem no banco, criar registros em conjunto e deletar todos os registros de uma determinada tabela. Na descrição dos experimentos, será mencionado o fato do banco de dados estar em cenários semelhantes em todos os testes, logo, foi de grande ajuda. Ademais, durante a execução dos testes, esta aplicação não teve impacto no desempenho do sistema aferido.

3.3.3 Utilização de Docker

Para a execução do processo de testes do presente trabalho, optou-se pela estratégia de separar todas as aplicações em contêineres utilizando a ferramenta Docker, e realizar o gerenciamento de instrumentação dos mesmos com Docker Compose. Ao todo, foram utilizados seis contêineres, nos quais os quatro implementados tiveram suas imagens criadas a partir de um *Dockerfile* específico para cada uma, enquanto para os outros dois (correspondentes ao serviço de fila e SGBD) foram utilizadas as imagens padrão. A Figura 5 ilustra a estratégia adotada, com os respectivos nomes dos serviços próximos às caixas azuis:

Figura 5 - Arquitetura proposta com a utilização de Docker e Docker Compose.



Fonte: elaborada pelo autor.

A utilização de Docker se mostrou relevante para a delimitação do ambiente de testes e os experimentos por conta das facilidades de configuração e execução de diversas ferramentas, a qual não existe a necessidade de serem instaladas localmente (como RabbitMQ e PostgreSQL), e com um simples comando é possível rodar todas as aplicações ou somente a desejada. Além disso, o Docker Compose permite a delimitação de memória e CPU que cada serviço irá utilizar, o que foi fundamental para mitigar o risco de serviços estarem disputando recursos entre si.

3.3.4 Repositório do projeto

Durante o processo de desenvolvimento, foi utilizado Git para versionamento e GitHub para o repositório remoto, como um repositório público[1].

[1] O repositório está acessível em *https://github.com/RafaelEmery/performance-analysis-server.*

3.4. Ambiente de testes

Primeiro, é importante mencionar as configurações do computador em que os experimentos foram realizados:

- CPU: Intel Core i5-8265U CPU @ 1.60GHz x 8; possui quatro núcleos físicos e oito threads.
- Memória RAM: 16,0 GB DDR4; 2666 MHz de frequência.
- Disco: 256GB SSD PCIe NVMe M.2
- Sistema Operacional: Linux Ubuntu 22.04.3 LTS
- Fabricante: Dell

Com a estratégia de executar as diferentes aplicações do sistema com desempenho aferido, optou-se por utilizar o recurso de Docker Compose de limitar a quantidade de memória e CPU. Logo, para os contêineres das aplicações criadas, foi utilizado:

- Número de núcleos da CPU: 2
- Memória RAM máxima: 4 GB

Portanto, as quatro aplicações essenciais para o sistema que possui o desempenho aferido utilizam uma quantidade reduzida de núcleos de CPU e memória máxima. A ideia de limitação de recursos por contêiner foi conveniente pois contribui para um maior isolamento das aplicações e realiza uma simulação do que seriam os recursos de uma instância simples de um serviço de nuvem, como a *t2.medium* do serviço EC2 da AWS como demonstrado no guia de tipos de instâncias (AWS, 2023), dado que o contexto da avaliação envolve arquiteturas distribuídas com alta granularidade e serviços que sejam relativamente pequenos (em complexidade e em gasto de recursos).

É importante ressaltar que os serviços de PostgreSQL e RabbitMQ não tiveram nenhuma limitação em recursos, pois entende-se que, nos experimentos do presente trabalho, tais serviços devem ser executados sempre com o melhor desempenho possível para evitar de serem gargalos de início. Então, a estratégia foi incluir limitações somente nas aplicações desenvolvidas para o protótipo e isolá-las, e manter os serviços considerados auxiliares ou secundários, com a melhor performance possível.

Por fim, optou-se por manter somente duas aplicações rodando no Docker (com exceção dos serviços de PostgreSQL e RabbitMQ, que ficaram constantemente ligados e consumindo recursos), sendo o BFF e o servidor que representa o método de comunicação testado no momento. Entendeu-se que foi importante para garantir que não houvesse disputa de recursos. Logo, sempre que uma aferição é realizada sobre uma aplicação de um método de comunicação, os outros estão inativos.

3.4.1 Modelagem do banco de dados

Como evidenciado na arquitetura proposta, todas as aplicações interagem com o banco, e para isso, o que será relevante para os fatores e níveis definidos a seguir. Para isso, foi pensado em um cenário que o banco possui somente uma tabela, que representa produtos. Os produtos possuem nome, SKU (código de produto), nome de vendedor, preço, desconto, entre outros campos relevantes. A temática de produtos foi meramente uma escolha do autor, para ser mais coerente com um determinado contexto real. Além disso, a tabela foi criada uma única vez dentro do contêiner do PostgreSQL e mantida por meio da declaração *volumes* presentes no arquivo *docker-compose.yml*. A Figura 6 evidencia as colunas e seus respectivos tipos, além de outras informações:

Figura 6 - Visualização da tabela de produtos.

Name	Type	Nullable	Default Value	Primary
id	uuid	☐	(NULL)	✓
name	text	☐	(NULL)	
sku	varchar(64)	☐	(NULL)	
seller_name	varchar(64)	☐	(NULL)	
price	float8(53,)	☐	(NULL)	
available_discount	float8(53,)	☐	(NULL)	
available_quantity	int4(32,0)	☐	(NULL)	
sales_quantity	int4(32,0)	☐	(NULL)	
active	bool	☐	(NULL)	
created_at	timestamptz(6)	☐	now()	
updated_at	timestamptz(6)	☐	now()	

Fonte: elaborada pelo autor.

A chave primária é nomeada de *id* e é do tipo *uuid*, uma *string* que representa um identificador único universal. Adotou-se o *id* com este tipo, ao invés de um *unsigned integer* convencional pois é uma prática comum em microsserviços, na qual muitas vezes existem diversos bancos - para diversas aplicações e tipos - e existe a necessidade de identificar um mesmo registro comum entre eles. Além disso, não possui impacto relevante na performance de consultas e interações com o banco. Por fim, as outras colunas e tipos são mais comuns com as práticas convencionais de banco de dados, como por exemplo, *timestamps* para obter registro de criação e atualização.

3.5. Definição de fatores e níveis

Os fatores e seus respectivos níveis foram definidos de forma que representem o método de comunicação, que é o objetivo do presente trabalho, funcionalidades diversas que as aplicações podem executar, e cargas de trabalhos diferentes. Para fins de melhor entendimento, os fatores são separados (por tipo) de forma que sejam relacionados ao sistema, e relacionados à carga de trabalho. Evidentemente, entende-se que os fatores e níveis devem representar diferentes cenários que podem ou não ser relevantes para o desempenho do sistema aferido. Além disso, inicialmente optou-se por três níveis para cada fator. A Tabela 1 demonstra os fatores e seus níveis e posteriormente será feita uma análise por trás dos fatores, níveis e respectivas escolhas:

Tabela 1 - Fatores e níveis definidos.

Fator	Tipo	Níveis
Método de comunicação	Relacionado ao sistema	REST, gRPC e um *message broker*[2]
Funcionalidade no servidor	Relacionado ao sistema	Criação, relatório e consulta com ordenação
Quantidade de requisições feitas	Carga de trabalho	1.000, 5.000 e 10.000

[2] É referenciado no presente trabalho como *message broker* ou broker de mensageria.

Quantidade de usuários simultâneos	Carga de trabalho	1, 10 e 100

Fonte: elaborada pelo autor.

3.5.1 Método de comunicação

É considerado a essência do trabalho e o fator pelo qual as métricas são mais direcionadas. Representam os três métodos (ou abordagens)[3] de comunicação descritos e que terão seu desempenho aferido.

É importante mencionar, que como a arquitetura demonstra, foram implementadas aplicações específicas para cada método de comunicação, nas quais a aplicação para o nível REST possui *endpoints*, a aplicação gRPC possui os métodos implementados e a aplicação de RabbitMQ (consumidor de fila) consome mensagens publicadas na fila. Além disso, o RabbitMQ só possui uma fila, com um nome genérico, então as informações que são passadas via *endpoint* ou métodos em REST e gRPC respectivamente, são passadas via metadados e no *body* da mensagem publicada, o que garante que todos as três aplicações consigam sempre receber as mesmas informações durante a interação com a aplicação BFF. Como exemplo, a criação de um produto:

- No REST, seria um *endpoint* chamado */create*
- No gRPC, seria um método chamado *Create*
- No RabbitMQ, um metadado passado no cabeçalho da mensagem chamado *create*.

3.5.2 Funcionalidade no servidor

É a funcionalidade que o servidor implementa e que será executada durante o cenário de teste. Foi um fator pensado com o objetivo de aproximar o sistema em questão com um contexto mais próximo da realidade, executando fluxos lógicos que façam sentido para os usuários e que interajam com o contexto e modelagem proposta. É importante mencionar que todos os servidores (que representam os métodos de comunicação) implementam os três níveis de funcionalidade no servidor. Ademais, as funcionalidades foram pensadas para se utilizar diferentes recursos, representando características de ser *CPU bound, memory bound e IO bound*, ou seja, interagem com CPU, memória ou IO. Os níveis escolhidos:

[3] Pode ser chamado por método ou abordagem de comunicação. Como fator, foi chamado de método.

- **Criação**: O nome da funcionalidade em *endpoint*, método ou metadado na mensagem é *create.* O servidor realiza a criação de um produto no banco de dados. O Golang utiliza o pacote database/sql para executar uma *query* de *insert*, de acordo com parâmetros "*fake*" que são gerados na aplicação BFF - não foi visto necessidade de fazer a ferramenta de teste de carga enviar estes dados. É majoritariamente *IO bound*, pois realiza conexão com o banco de dados, entre outras interações.
- **Relatório**: O nome da funcionalidade em *endpoint*, método ou metadado na mensagem é *report.* O servidor realiza a obtenção de 100 registros da tabela de produtos no banco de dados e gera um relatório em PDF com estes valores. Foi utilizado o pacote *maroto* para geração leve de arquivos e os arquivos criados são salvos em uma pasta de arquivos temporários, e existe um comando no *Makefile* para remover estes arquivos. É majoritariamente *IO bound* e *memory bound*, pois interage com o sistema de arquivos e mantém os 100 produtos em memória, entre outras alocações.
- **Consulta com ordenação**: O nome da funcionalidade em *endpoint*, método ou metadado na mensagem é *getByDiscount.* Realiza a obtenção de 100 produtos da tabela de produtos, e em resumo, consiste em uma lógica que aplica um desconto no produto (por meio do valor da coluna *available_discount*) e realiza a ordenação dos produtos para retornar ao BFF. A ordenação é realizada por meio do pacote *sort* do Go, que implementa por padrão o algoritmo QuickSort, e foi adotado o mesmo pois é um algoritmo considerado em média eficiente. Consiste em ter as características de *IO bound* e *memory bound*, em conjunto de *CPU bound* pois a ordenação exige mais recurso da CPU.

É relevante citar que nas funcionalidades que envolvem consulta ao banco de dados, o banco já está "populado" com dados *"fake"*[4] de produtos, criados com a aplicação de *Setup* mencionada anteriormente. Outro ponto importante, é que cada funcionalidade pode exigir parâmetros diferentes, e como mencionado na descrição do fator anterior, todos os três níveis de métodos de comunicação conseguem implementar as funcionalidades de forma semelhante, sem haver diferenças em parâmetros e retorno.

[4] Produtos criados com o auxílio de pacotes que geram dados falsos e aleatórios.

3.5.3 Quantidade de requisições

É um fator de carga de trabalho que representa quantas requisições serão realizadas. Os valores para os níveis 1.000, 5.000 e 10.000 tiveram como embasamento principal a ideia de se realizar o teste de carga simulando um “ambiente de stress” para aplicação que seja muito rápido (1.000 requisições), intermediário (5.000 requisições) e mais longo (10.000 requisições). Entende-se que o tempo em que o teste de carga é aplicado pode ser um fator que tenha influência sobre as métricas e resultados obtidos, e que, por vezes, é extremamente útil projetar testes com um número alto de requisições pois indica que o tempo de stress da aplicação será mais longo. Entretanto, o tempo em que se executam estas quantidades de requisições pode estar relacionado com o próximo fator e seus níveis - quantidade de usuários simultâneos - pois, por exemplo, 1000 requisições podem ser atingidas mais rapidamente com 100 usuários simultâneos, do que com 1.

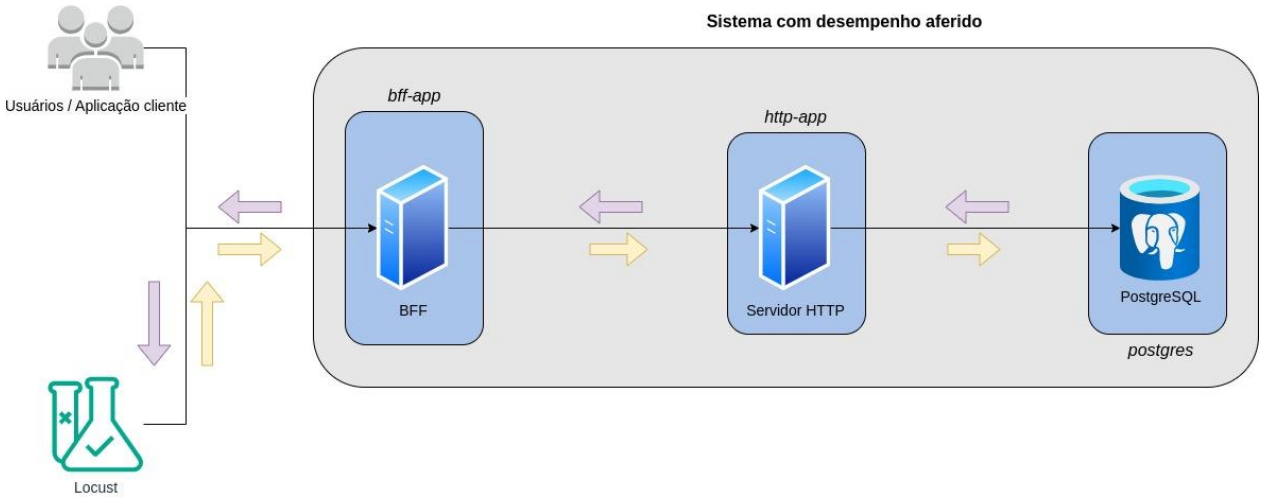
3.5.4 Quantidade de usuários simultâneos

É um fator que representa quantos usuários estão realizando requisições simultaneamente. A ferramenta adotada para os testes de carga já trás a possibilidade de definição como padrão e, como mencionado, pode influenciar a quantidade de requisições executadas.

3.6. Considerações sobre a aferição

Partindo da premissa que não é simplesmente uma aplicação que está tendo seu desempenho aferido e sim um sistema, é fundamental o entendimento sobre em que partes do fluxo de comunicação entre as diversas aplicações apresentado na arquitetura, evidenciado como “sistema com desempenho aferido na Figura 5. A Figura 7 demonstra visualmente o fluxo de uma requisição feita pelo Locust - ferramenta para testes de carga - ou o usuários no sistema com desempenho aferido, por meios das setas amarelas e roxas, representando a “ida e a volta” de uma requisição, respectivamente:

Figura 7 - Fluxo executado por uma requisição.



Fonte: elaborada pelo autor.

Tomando como exemplo o método de REST com a *http-app*, será:

1. Locust realiza a requisição a um *endpoint* simples do BFF chamado */interact/http* (podendo ser */interact/grpc* ou */interact/rabbitmq*) e passando no corpo da requisição a funcionalidade que deve ser executada.
2. O BFF recebe a requisição do Locust e chama o *endpoint* relacionado a funcionalidade na aplicação REST (*http-app).*
3. A aplicação REST recebe a requisição e executa a lógica referente ao fator de funcionalidade (criação, relatório ou consulta com ordenação).
4. Como todas as funcionalidades interagem com o banco, executa uma *query* no banco de dados.
5. O banco de dados retorna o resultado da *query.*
6. A aplicação REST repassa a resposta para o BFF.
7. A aplicação REST repassa para o Locust.
8. O Locust processa os resultados.

Algumas considerações sobre o exemplo e passo-a-passo apresentado:

- No método de gRPC o fluxo é similar, e no *message broker* existem algumas diferenças que serão abordadas posteriormente.
- Este fluxo é mais relevante à métricas relativas ao sistema inteiro.
- Neste fluxo, está incluso o tratamento de erros, e caso ocorra alguma falha em alguma etapa do processo no sistema com desempenho aferido, o Locust irá interpretar como falha.

3.7. Definição de métricas

Foram definidas as seguintes métricas:

Tabela 2 - Definição das métricas.

Métrica	Unidade de medida
Tempo médio de resposta	ms (milissegundos)
Taxa de falhas	% (porcentagem)
Gasto médio de memória RAM por requisição na aplicação BFF	KB (kilobyte)
Gasto médio de memória RAM por requisição na aplicação servidor	KB (kilobyte)

Fonte: elaborada pelo autor.

Como citado e exemplificado anteriormente, as métricas de tempo médio de resposta e taxa de falhas se referem ao sistema com desempenho como um todo, como resultados da “ida e volta” das requisições realizadas durante o teste de carga, e são relevantes para se entender o desempenho em tempo do sistema e os seus possíveis gargalos. A própria ferramenta de teste de carga provém estas métricas. É importante destacar que uma requisição com falha, é que possui o código *5xx* (do tipo 500) no cabeçalho da resposta.

As métricas de gasto médio de memória RAM por requisição nas aplicações (BFF e servidor que interage com o BFF) referem-se a quantidade média de memória alocada para cada requisição recebida e alocada dentro das aplicações, sendo medidas com o pacote *runtime* nativo do Go, e exibidas no log dos contêineres, para ao final do experimento, ser executado um *script* (também escrito em Go) que calcula as médias em *kilobytes* em cada aplicação. São métricas que, ao contrário das duas primeiras apresentadas, não são relativas ao sistema como um todo. Foram adotadas para ter visibilidade do quanto de memória RAM uma aplicação gasta em média para receber e processar uma requisição executada, e como pode divergir entre os diversos cenários. É importante ressaltar que adotou-se a memória RAM como recurso principal a ser observada ao invés de CPU ou disco

devido à proposta de diversas implementações do Go que tem como objetivo a baixa alocação de memória RAM, como por exemplo o *framework* Fiber e sua implementação sobre o *Fast HTTP*. Logo, metrificar a memória RAM como recurso principal foi considerada mais promissora para obter-se avaliações e resultados mais valiosos.

4. EXPERIMENTOS SIMPLES

A partir do que foi obtido na seção anterior referente à metodologia e definições iniciais, iniciou-se o processo de experimentação seguindo o planejamento fatorial simples. Primeiro, foi definida uma configuração inicial para os experimentos, e a partir desta configuração inicial, os experimentos foram executados dentro da quantidade considerada confiável - entre 10 e 32 repetições - variando um nível de cada fator por vez. Durante o processo foram colhidas todas as métricas (citadas na Tabela 2) e foi certificado que o ambiente de testes estava coerente com as propostas iniciais. A partir dos resultados, foram feitas análises com gráficos para o experimento poder prosseguir, pois o planejamento fatorial simples é o início das etapas de teste.

4.1. Configuração inicial e testes realizados

A configuração inicial adotada foi:

Tabela 3 - Configuração inicial.

Fator	Nível
Método de comunicação	REST
Funcionalidade no servidor	Criação
Quantidade de requisições feitas	5.000
Quantidade de usuários simultâneos	10

Fonte: elaborada pelo autor.

Como justificativas para esta configuração inicial, entende-se que o REST é um excelente ponto de partida para os experimentos, já que é o método mais popular dentre os apresentados, e as análises iniciais podem ser bem elucidativas, a criação corresponde à uma funcionalidade simples e objetiva, e os fatores de carga de trabalho tiveram valores intermediários para obter-se resultados que também possam ser elucidativos em relação aos níveis definidos.

Ao todo, foram nove configurações testadas[5] e as métricas definidas anteriormente foram devidamente observadas e documentadas. A Tabela 4 demonstra as configurações testadas e as células com a fonte em negrito representam os níveis variados:

Tabela 4 - Cenários de teste executados.

Método de comunicação	Funcionalidade no servidor	Quantidade de requisições realizadas	Quantidade de usuários simultâneos
REST	Criação	5.000	10
gRPC	Criação	5.000	10
Message broker	Criação	5.000	10
REST	**Relatório**	5.000	10
REST	**Consulta com ordenação**	5.000	10
REST	Criação	**1.000**	10
REST	Criação	**10.000**	10
REST	Criação	5.000	**1**
REST	Criação	5.000	**100**

Fonte: elaborada pelo autor.

[5] O registro (notas) dos experimentos simples estão disponíveis em: *https://docs.google.com/spreadsheets/d/1qFv7o7qm5hqKyFtetdzRL9AcyNoJA56subbf5QbTjLY/edit?usp=sharing*

É importante ressaltar que como se tratava de uma funcionalidade de criação, registros eram criados no banco de dados, e mesmo que no presente trabalho possa não ser uma escala que faça diferença para o tempo que o banco de dados leva para processar consultas, foi garantido que, a cada experimento que se iniciava, a tabela de produtos estava com zero registros. De forma similar, as funcionalidades que realizavam consultas no banco de dados (relatório e consulta ordenada) tiveram os testes sendo executados de forma que o banco só iria possuir 100 registros. Foi pensado em uma quantidade zerada ou pequena de registros no banco com a finalidade de manter o banco o mais leve possível para cada nível poder ser mais relevante, além de garantir a uniformidade do contexto em cada teste. Foi utilizada a Aplicação de *Setup* descrita na arquitetura para realizar estas operações.

Por fim, a cada teste executado, eram documentadas as métricas de tempo médio de resposta e taxa de falhas, que a ferramenta de teste de carga disponibiliza, e eram obtidos os valores referentes às médias de memória RAM no BFF e servidor em questão através do *script* construído. Após a obtenção de todas as métricas, os contêineres eram reiniciados através de comandos do Docker Compose, e os *logs* que continham as informações do experimento anterior, foram apagados. A reinicialização dos contêineres foi relevante pois é uma maneira fácil de reiniciar toda a aplicação e remover, por exemplo, algum recurso que ainda esteja sendo alocado, e por limpar os *logs* que o contêiner possui, já que a obtenção de duas métricas era realizada nas informações ali contidas, e é necessário que esteja somente os registros gravados do teste em questão.

4.2. Resultados obtidos

4.2.1 Considerações iniciais

Para a análise dos resultados, é necessário mencionar que nos testes executados, a métrica de taxa de falhas foi nula (zero) em todos os cenários de teste, o que já entende-se como um ponto relevante, visto que o sistema com desempenho aferido apresentou indícios de ser relativamente escalável, o que será um fator relevante abordado posteriormente. Portanto, a métrica de taxa de falhas não será mostrada na análise de influências dos fatores e níveis a seguir.

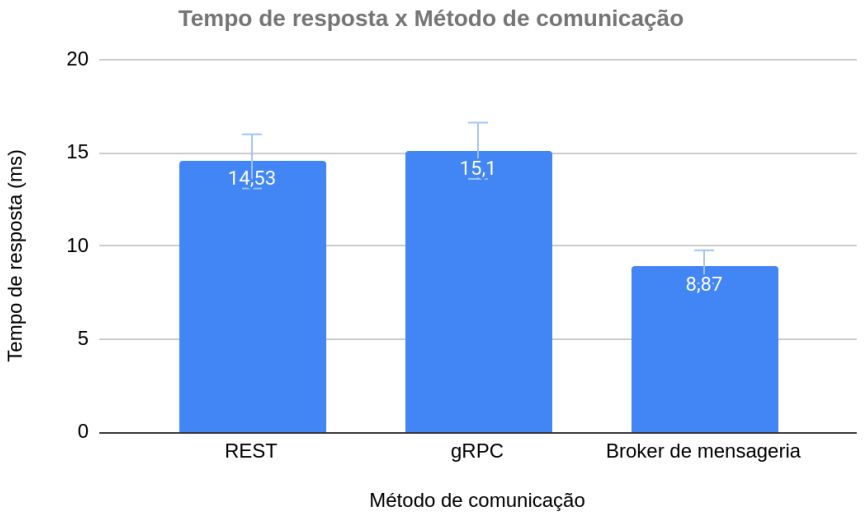
Ademais, para fins de entendimento e melhor visualização, a métrica “tempo médio de resposta” foi referenciada como “tempo de resposta” e as métricas de

“gasto médio de memória RAM por requisição na aplicação BFF” e “gasto médio de memória RAM por requisição na aplicação servidor” serão mostradas em conjunto e de forma abreviada por “Gasto médio (RAM)” e respectivamente “Gasto médio (KB) - BFF” e “Gasto médio (KB) - Servidor” nos gráficos a seguir. Por fim, os gráficos possuem um intervalo de confiança de 10%.

4.2.2 Influência do método de comunicação

Variando o nível do método de comunicação e observando o gráfico da Figura 8, é seguro afirmar que o tempo de resposta foi uma métrica que, de uma maneira geral, não foi muito afetada, dada a carga de trabalho definida. Os métodos REST e gRPC demonstraram desempenho similar, mas o *message broker* apresentou um melhor desempenho de pouco mais da metade do tempo dos outros níveis. Contudo, o *message broker* funciona de maneira de que o tempo de resposta metrificado somente até o momento em que a mensagem é publicada, logo, não é possível obter uma perspectiva real sobre o desempenho nesta métrica, e tal característica será abordado posteriormente no início do planejamento fatorial completo.

Figura 8 - Gráfico de tempo de resposta por método de comunicação.

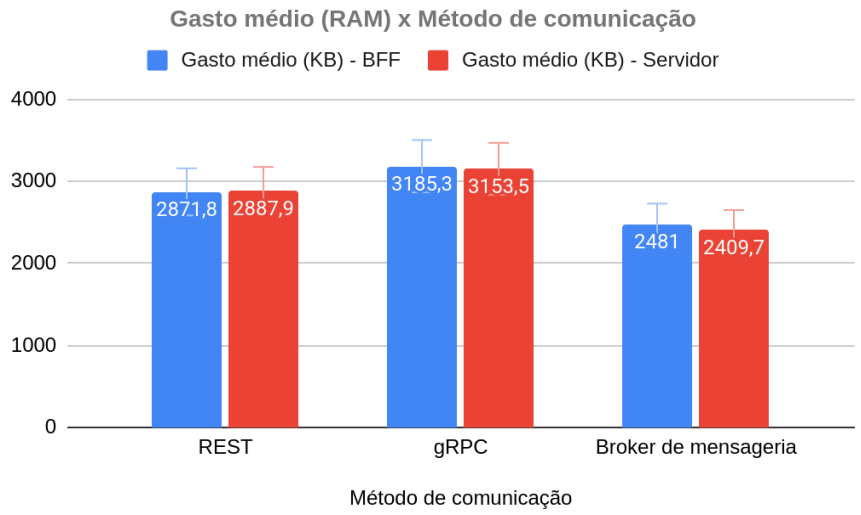


Fonte: elaborada pelo autor.

Como o gráfico da Figura 9 demonstra, em relação ao gasto médio de memória RAM nas aplicações em questão, nota-se que o gRPC consome mais memória que o método REST, porém a diferença neste cenário não se mostra relevante. O *message broker* foi o mais performático dos métodos, consumindo

menos memória no momento em que a aplicação BFF publica a mensagem e que a aplicação de servidor (neste caso, o consumidor de fila) consome a mensagem.

Figura 9 - Gráfico de gasto médio (RAM) por método de comunicação.

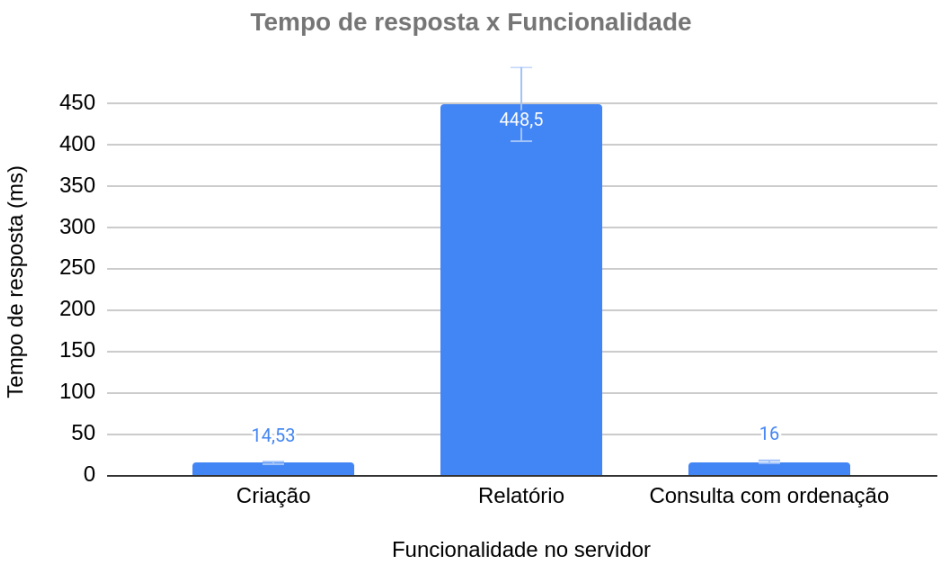


Fonte: elaborada pelo autor.

4.2.3 Influência da funcionalidade no servidor

Assim como explícito no gráfico da Figura 10, as alterações na funcionalidade do servidor resultaram na funcionalidade de relatório com um tempo de resposta extremamente maior, e as outras funcionalidades em cerca de 3,4% de seu tempo. Nota-se que a funcionalidade de relatório possui grande impacto no tempo, pela geração do documento em PDF e posterior interação com o *filesystem,* visto que salva o arquivo em uma pasta. A influência do tempo da consulta do banco no tempo de resposta pode ser descartada pois a funcionalidade de consulta com ordenação realiza a mesma consulta no banco, e não onera o desempenho.

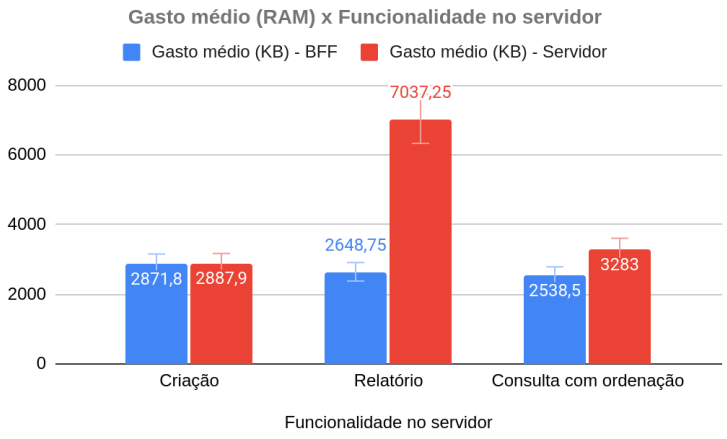
Figura 10 - Gráfico do tempo de resposta por funcionalidade no servidor.



Fonte: elaborada pelo autor.

Assim como a métrica anterior e observando o gráfico da Figura 11, nota-se que a funcionalidade de relatório realiza maior alocação de memória RAM especificamente no servidor (no BFF não diferenças consideráveis), sendo 7037,25 KB em média, cerca de 244% da criação (2887,9 KB) e 214% da consulta com ordenação (3283 KB). Nota-se também uma diferença entre a criação e consulta com ordenação, evidenciando que a maior complexidade lógica da consulta com ordenação tem impacto, mesmo que no presente cenário irrelevante.

Figura 11 - Gráfico de gasto médio (RAM) por funcionalidade no servidor.

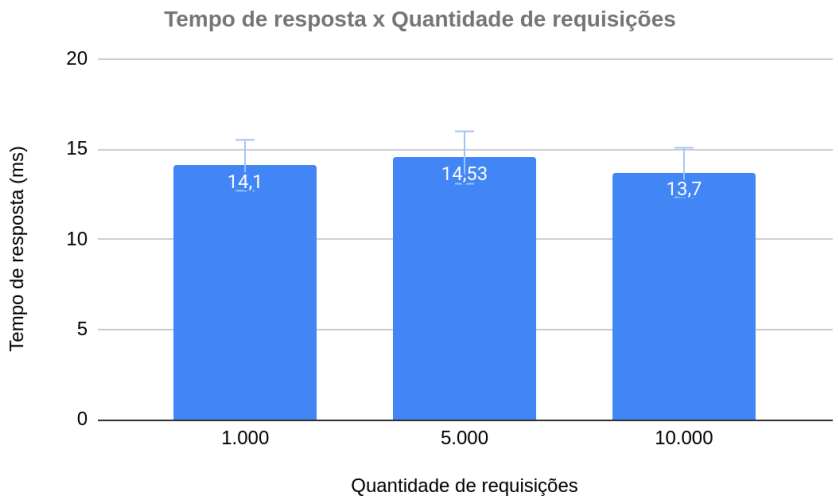


Fonte: elaborada pelo autor

4.2.4 Influência da quantidade de requisições

A partir do gráfico da Figura 12, nota-se que a variação da quantidade de requisições não demonstrou ter impacto relevante no tempo de resposta, na qual com 1.000, 5.000 e 10.000 requisições os valores foram 14,1 ms, 14,53 ms e 13,7 ms respectivamente. Mesmo que o último nível (mais alto) tenha resultado em menos tempo, entende-se que não é impactante pois a diferença é insignificante e dentro do intervalo de confiança.

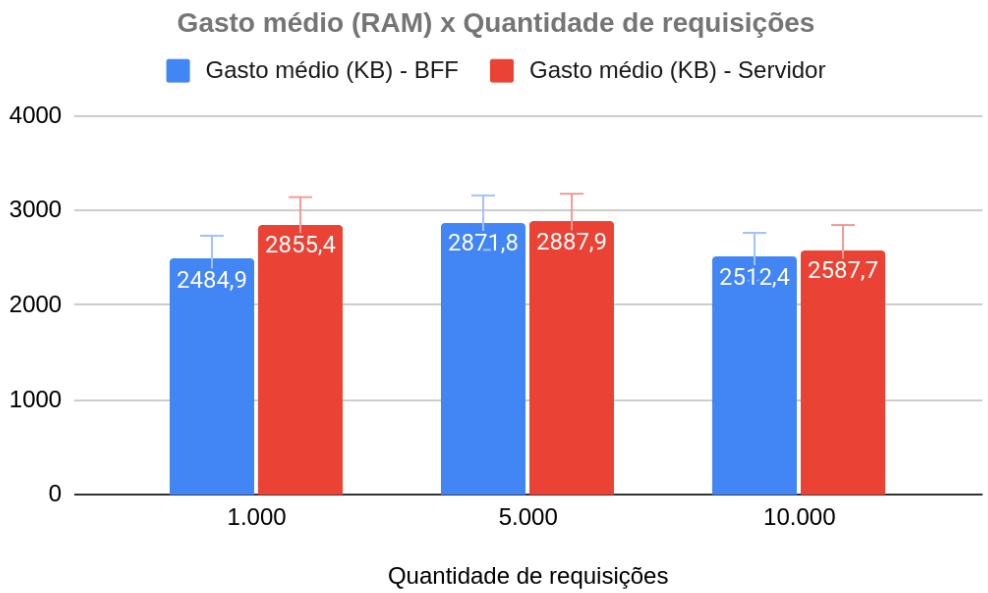
Figura 12 - Gráfico de tempo de resposta por quantidade de requisições.



Fonte: elaborada pelo autor.

Assim como a métrica de tempo de resposta, nota-se pelo gráfico da Figura 13 que a variação da quantidade de requisições não apresentou impacto significativo em relação ao gasto médio de memória por aplicações. Os valores ficaram entre 2484,9 KB e 2887,9 KB e assim como os valores de tempo de resposta, as diferenças são pequenas e dentro do intervalo de confiança.

Figura 13 - Gráfico de gasto médio (RAM) por quantidade de requisições.

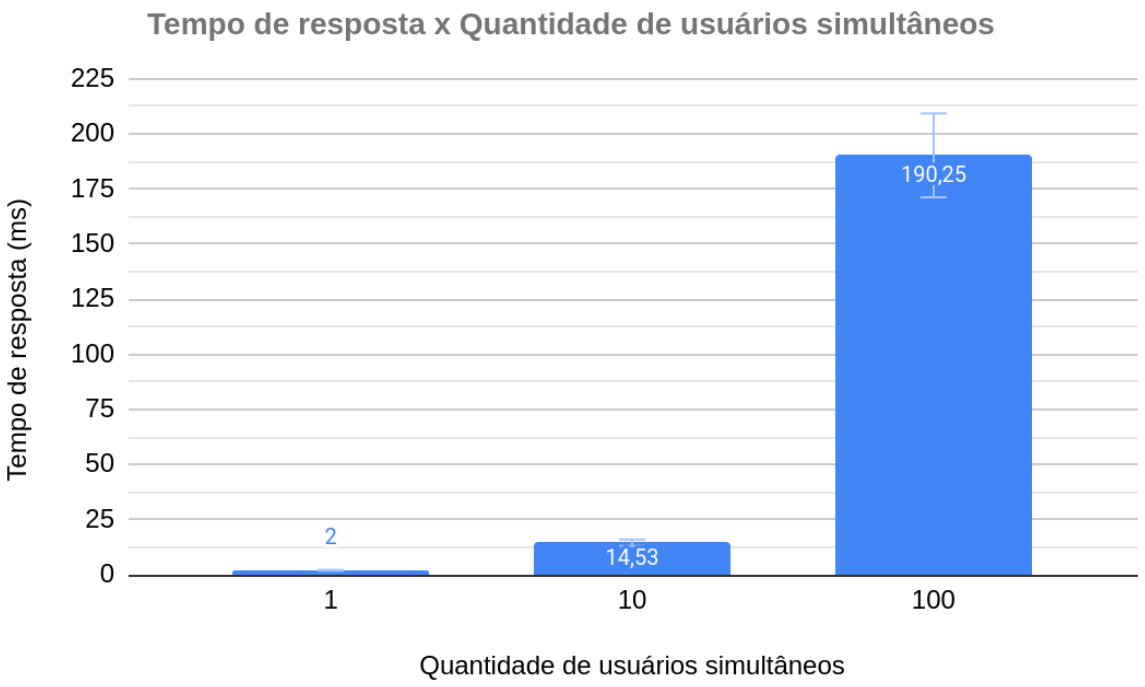


Fonte: elaborada pelo autor.

4.2.5 Influência da quantidade de usuários simultâneos

A alternância entre os níveis da quantidade de usuários simultâneos mostrou-se significativa em relação ao tempo de resposta, como demonstra o gráfico da Figura 14. Nota-se que, com 1 usuário simultâneo, o sistema aferido foi extremamente rápido (2 ms), com 10 usuários simultâneos, ainda se mostrou eficiente em termos de velocidade, entretanto com 100 usuários simultâneos o tempo de resposta aumentou significativamente para 190,25 ms, representando um aumento de cerca de 1309%. Como fator de carga de trabalho, entende-se a quantidade de usuários simultâneos como o mais influente para esta métrica.

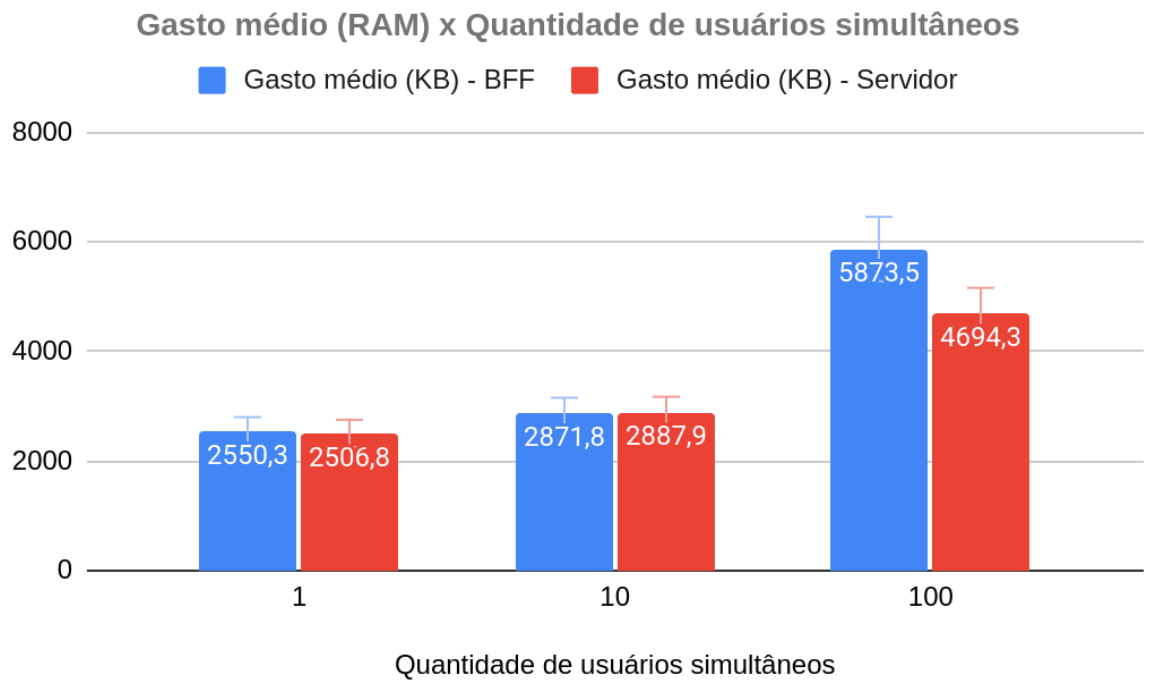
Figura 14 - Gráfico de tempo de resposta por quantidade de usuários simultâneos.



Fonte: elaborada pelo autor.

Observando o gráfico da Figura 15, nota-se que entre 1 e 10 usuários simultâneos, não houveram diferenças relevantes de gasto médio de memória RAM (entre 300 e 400 KB de diferença em ambas aplicações), contudo, com 100 usuários simultâneos o gasto médio de memória RAM ficou próximo do dobro, com 5873,5 KB no BFF e 4694,3 KB no servidor. Nota-se um maior consumo na aplicação BFF para lidar com as requisições simultâneas. Assim como para a métrica de tempo de resposta, entende-se este fator de carga de trabalho como o mais impactante para esta métrica.

Figura 15 - Gráfico de gasto médio (RAM) por quantidade de usuários simultâneos.



Fonte: elaborada pelo autor.

5. EXPERIMENTOS COMPLETOS

A partir dos resultados dos experimentos simples, houve uma melhor compreensão em relação aos fatores, níveis e métricas propostos inicialmente e qual sua relevância para o presente estudo e possíveis conclusões. O planejamento fatorial completo consiste em testar todas as combinações de fatores com seus respectivos níveis com a finalidade de entender de maneira mais objetiva e detalhada o funcionamento do desempenho aferido, com base nas métricas definidas. No presente trabalho, será adotada a estratégia de realizar a redução da quantidade de níveis dos fatores para 2 níveis, considerando somente dois níveis verdadeiramente relevantes para o estudo e mantendo a objetividade para que os testes possam ser executadas da maneira mais completa possível - já que com uma grande combinação de fatores e níveis, o número de experimentos aumenta significativamente e existe o risco do foco do presente trabalho ser perdido.

Logo, percebeu-se a necessidade de reavaliar níveis, fatores e até métricas, para assim executar os testes. Por fim, os testes foram executados - de maneira semelhante aos experimentos simples - com todas as combinações possíveis e os resultados foram obtidos.

5.1. Alterando níveis, fatores e métricas

Para adaptar o presente experimento para o planejamento fatorial completo com 2 níveis para cada fator, foram necessárias adaptações em fatores e níveis. Além disso, as métricas sofreram alterações com a finalidade de serem obtidos os resultados que melhor possam fornecer informações para conclusões serem atingidas.

5.1.1 Mudanças em fatores e níveis

Em relação ao fator de método de comunicação, foram mantidos os níveis REST e gRPC, e removido o nível de *message broker*. Embora tenha obtido resultados melhores no planejamento simples, assim como foi mencionado nos resultados obtidos, o *message broker* é inadequado para a aferição da métrica de tempo de resposta, considerada uma métrica extremamente relevante para o presente trabalho. De acordo com a Figura 7 do presente trabalho e sua exemplificação, métricas como o tempo médio de resposta e taxa de erros são aferidas de acordo com o desempenho do sistema, e da “ida e volta” da requisição feita pela ferramenta de teste de carga, e como o *message broker* possui a característica de ser uma comunicação assíncrona (ao contrário de REST e gRPC que são síncronas), não é possível se obter referência de, por exemplo, o tempo de “volta” da requisição feita ou se ocorreu alguma falha durante o processo. Por fim, existe uma dificuldade de metrificar exatamente o tempo em que o RabbitMQ (serviço de fila) repassa as mensagens para serem consumidas pelo servidor, o que adiciona mais uma camada de complexidade para essa análise, já que o BFF não se comunica diretamente com o servidor como nos outros níveis. Logo, dadas as divergências com os outros dois níveis e suas respectivas formas de analisar o desempenho, a escolha foi removê-lo.

No fator de funcionalidade no servidor, optou-se por manter criação e consulta com ordenação e a remoção do fator de relatório. A funcionalidade no servidor de relatório se mostrou muito divergente em relação às outras duas e com grande influência nas métricas, o que ocorreu devido ao que é executado e como é implementado. Entende-se que o foco do presente trabalho é demonstrar as influências do método de comunicação em diferentes cenários, então optou-se por manter este direcionamento. Ademais, é importante ressaltar que os níveis de

criação e consulta de ordenação são promissores no tocante à quantidade de dados enviados na resposta - em que a criação retorna um objeto de produto e a consulta com ordenação 100 objetos de produto (no caso do REST em formato JSON) - e que esta diferença entre as respostas pode ser um aspecto relevante à se avaliar, pois existem diferenças em como os métodos de comunicação enviam as respostas ao cliente que realizou a requisição.

Por fim, em relação a quantidade de usuários simultâneos, o único nível mantido foi de 100 usuários simultâneos, e os níveis de 1 e 10 foram removidos, e então mais um nível foi adicionado: 200 usuários simultâneos. Mesmo com os resultados em relação aos níveis de 1 e 10 usuários simultâneos, optou-se por removê-los e incluir um nível mais elevado de carga de trabalho pois o sistema com desempenho aferido se mostrou promissor em termos de escalabilidade, então entendeu-se que seria relevante adicionar uma carga de trabalho melhor para verificar seu comportamento. Além disso, a proposta de se adicionar um nível de carga de trabalho maior pode ser útil para verificar a taxa de falhas e identificar possíveis gargalos no sistema.

A Tabela 5 mostra os fatores e níveis para o experimento completo:

Tabela 5 - Fatores e níveis após alterações.

Fator	Níveis
Método de comunicação	REST e gRPC
Funcionalidade no servidor	Criação e consulta com ordenação
Quantidade de usuários simultâneos	100 e 200

Fonte: elaborada pelo autor.

5.1.2 Fator removido

Diante da pouca relevância que o fator de quantidade de requisições teve nos experimentos simples, compreendeu-se que não haveria necessidade de considerar tal fator e seus respectivos níveis, onde o fator de carga de trabalho mais influente foi a quantidade de usuários simultâneos. Com isso, a quantidade de requisições foi mantida como um valor fixo em 10.000, correspondente ao terceiro nível do fator

definido inicialmente. Como pretexto para a decisão, entende-se que executar os testes de forma que sejam feitas mais requisições como uma potencial vantagem, garantindo maior confiança e coerência em relação ao desempenho do sistema aferido, já que estará submetido ao teste de carga durante mais tempo.

5.1.3 Métricas alteradas

Optou-se por manter a métrica de tempo médio de resposta, de acordo com a sua relevância para o que é tido como bom desempenho, e a métrica de taxa de falhas, embora irrelevante nos experimentos simples (com resultados sendo 0%), ainda foi mantida para caso ocorram requisições com falhas e identificação de gargalos, visto que a carga de trabalho teve um nível maior (200 usuários simultâneos). Além disso, a métrica de vazão foi inserida para ter melhor entendimento de como maiores tempos de resposta, falhas e possíveis gargalos podem influenciar em quantas requisições são realizadas por segundo.

Em relação ao gasto de recurso, foi mantida a métrica de gasto médio de memória RAM por requisição na aplicação servidor e removida à métrica similar na aplicação BFF. Entendeu-se que é mais vantajoso monitorar o gasto de recurso no servidor, pois em um contexto mais complexo, o mesmo pode receber requisições e cargas de diversos outros clientes, sendo outros BFFs ou outros servidores.

Por fim, foi adicionada uma métrica de tempo médio de interação com banco de dados por requisição, pois poderia ser valioso a metrificação e posterior entendimento de se o banco de dados possui influência no desempenho, visto que ele faz parte do sistema como um todo. É importante destacar que o tempo de interação só é registrado se ocorre sem erros, portanto não é possível subtrair este valor com os tempos de respostas - visto que o tempo de resposta considera o tempo caso haja falha. Por fim, foi medida de forma similar ao gasto médio de memória RAM por requisição no servidor, imprimindo os valores nos *logs* do contêiner e utilizando um *script* para calcular a média.

A Tabela 6 demonstra todas as métricas e unidades de medida:

Tabela 6 - Métricas após alterações.

Métrica	Unidade de medida
Tempo médio de resposta	ms (milissegundos)
Taxa de falhas	% (porcentagem)
Gasto médio de memória RAM por requisição na aplicação servidor	KB (kilobyte)
Tempo médio de interação com banco de dados por requisição	ms (milissegundos)
Vazão	req / s (requisições por segundo)

Fonte: elaborada pelo autor

5.2. Testes realizados

Ao todo, foram nove configurações testadas e as aferidas[6], como demonstra a Tabela 7. Assim como descrito e executado nos experimentos simples, o banco de dados era limpo (quando a funcionalidade no servidor foi criação), os resultados das métricas obtidos e containers reiniciados.

Tabela 7 - Cenários de teste executados nos experimentos completos.

Método de comunicação	Funcionalidade no servidor	Quantidade de usuários simultâneos
REST	Criação	100
REST	Criação	200
REST	Consulta com ordenação	100
REST	Consulta com ordenação	200

[6] O registro (notas) dos experimentos simples estão disponíveis em: *https://docs.google.com/spreadsheets/d/1qFv7o7qm5hqKyFtetdzRL9AcyNoJA56subbf5QbTjLY/edit?usp=sharing*

gRPC	Criação	100
gRPC	Criação	200
gRPC	Consulta com ordenação	100
gRPC	Consulta com ordenação	200

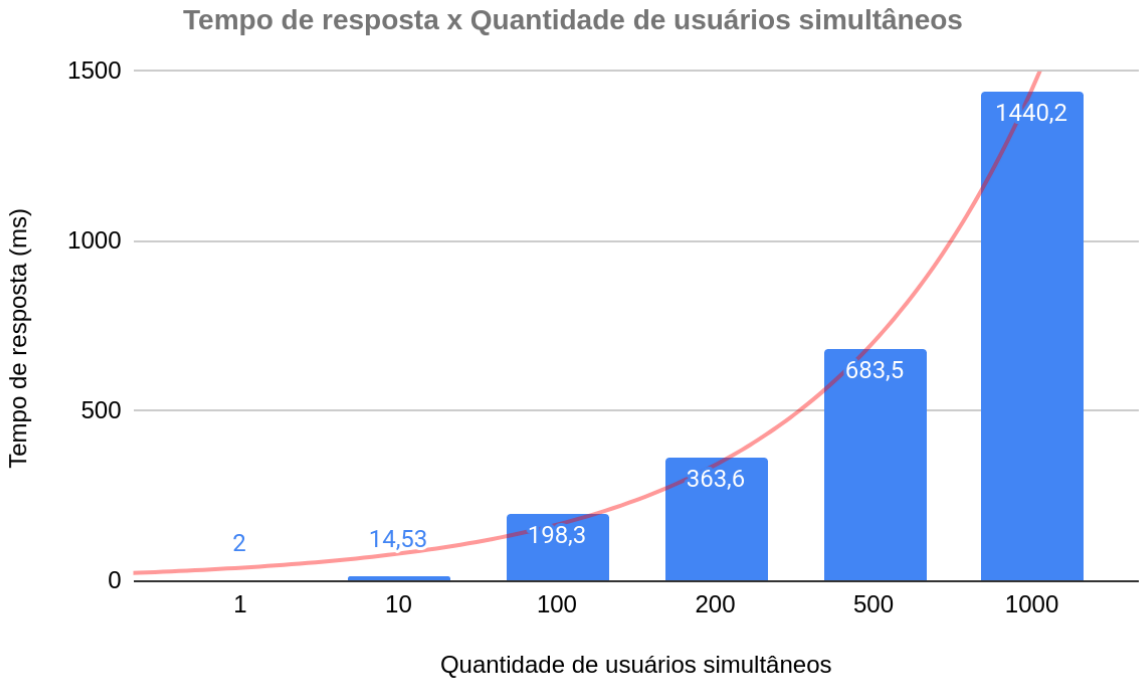
Fonte: elaborada pelo autor.

5.3. Resultados obtidos

A partir dos resultados obtidos nos experimentos simples e as consequentes modificações em fatores, níveis e métricas, adotou-se a estratégia de análise dos resultados pelo parâmetro de carga de trabalho, já que a quantidade de usuários simultâneos exerceu grande influência nos resultados obtidos. A abordagem adotada consiste em representar o desempenho das combinações da Tabela 7. Além disso, optou-se por visualizar e analisar, primeiramente por tempo médio de resposta (“tempo médio de resposta”), tempo médio de interação com banco de dados por requisição (“tempo de interação com BD”) e vazão, e posteriormente por taxa de falhas (caso não seja nulas) e recursos (gasto médio de memória RAM por requisição na aplicação servidor).

Em relação à influência da carga de trabalho em métricas, o gráfico da Figura 16 demonstra um exemplo de aferição com diferentes quantidades de usuários simultâneos e como que impactou o tempo de resposta. O exemplo se trata da combinação REST, criação e 10.000 requisições variando-se a quantidade de usuários simultâneos, e foram aferidos os valores de 500 e 1.000 para fins de visualização do comportamento.

Figura 16 - Aumento do tempo de resposta com base no aumento de usuários simultâneos.



Fonte: elaborada pelo autor.

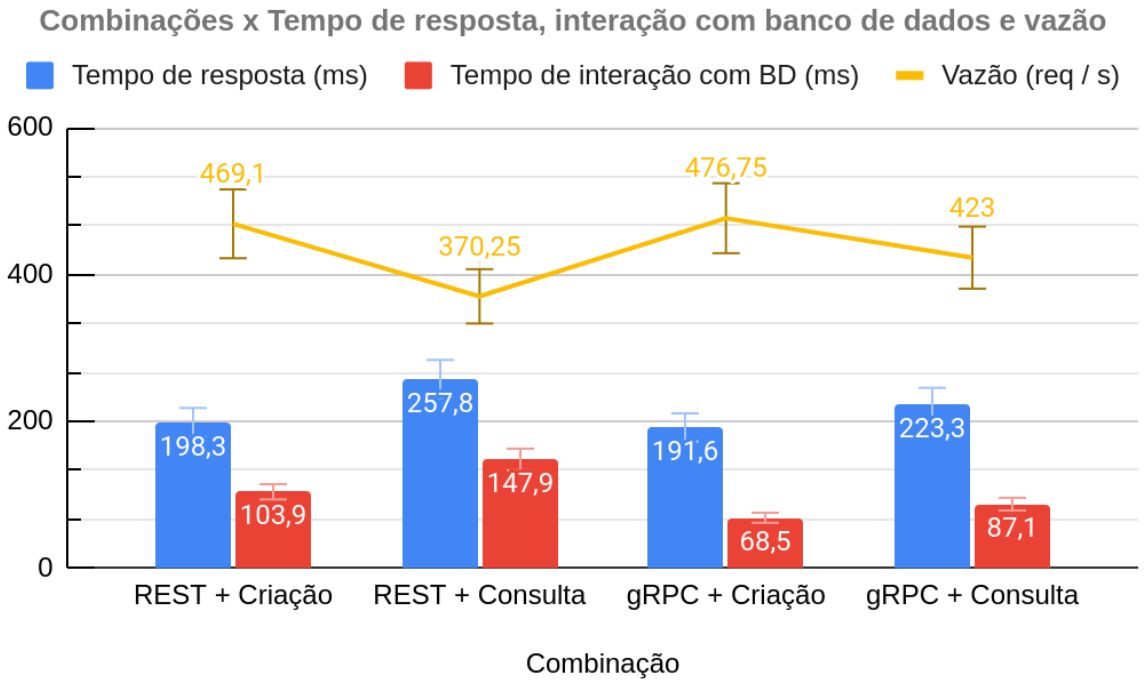
5.3.1 Com 100 usuários simultâneos

Com a carga de trabalho de 100 usuários simultâneos, a taxa de falhas foi zero para todas as combinações. O gráfico da Figura 17 demonstra o tempo de resposta, tempo de interação e a vazão no momento em que o teste de carga atingiu 10.000 requisições realizadas. Nota-se que o REST e gRPC demonstraram pouca diferença no tempo de resposta, com resultados de 198,3 ms, 257,8 ms para REST e 191,6 ms e 223,3 ms para gRPC, com as funcionalidades de criação e consulta com ordenação respectivamente. Contudo, houve uma diferença considerável no tempo de interação com o banco de dados, na qual com o método REST foi medido 103,9 ms e 147,9 ms e com o método gRPC foi medido 68,5 ms e 87,1 ms com as funcionalidades de criação e consulta com ordenação respectivamente, o que demonstra que o servidor com o método gRPC, embora minimamente mais lento em alguns casos, interage com o banco de dados de uma forma menos onerosa.

Na funcionalidade de consulta, nota-se que o gRPC, pela primeira vez nos experimentos até o presente estágio, obteve um desempenho melhor do que o

REST, o que pode estar relacionado com a melhor interação com o banco de dados juntamente com o formato no qual o gRPC realiza a serialização da resposta e a envia (e como o BFF a recebe). Por fim, a vazão se mostrou coerente com as respectivas combinações, visto que segue o padrão dos tempos de resposta - um tempo de resposta maior, resulta em uma vazão menor.

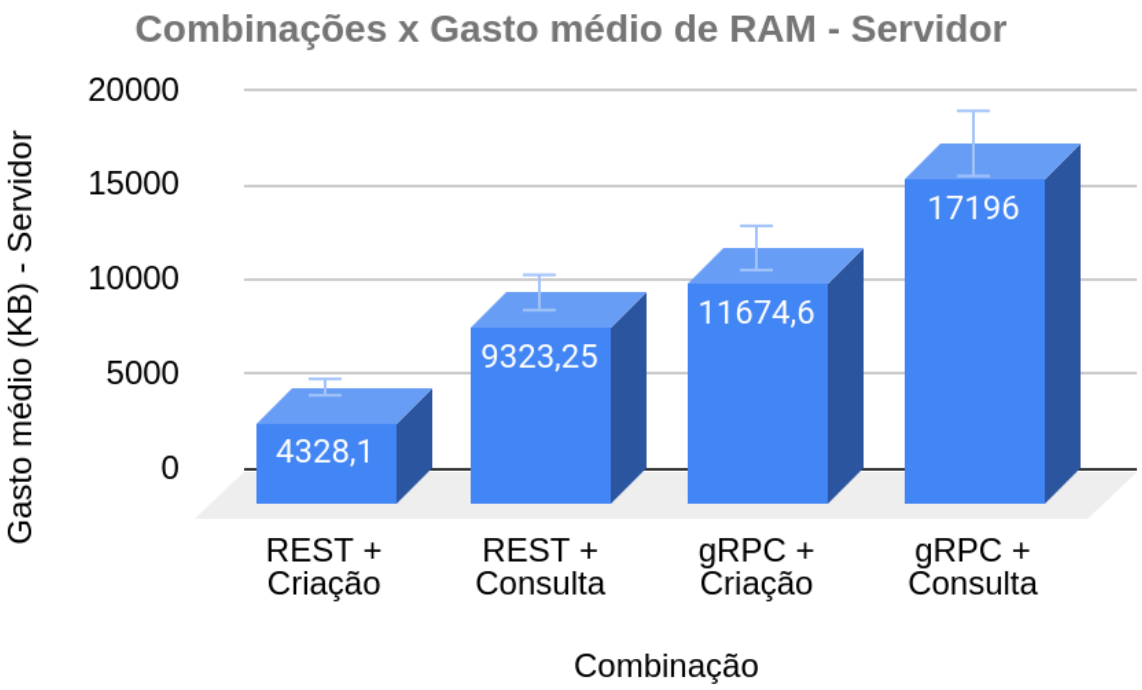
Figura 17 - Métricas relacionadas ao tempo por combinações (100).



Fonte: elaborada pelo autor.

Os resultados relativos ao gasto de recursos (memória RAM) médio por requisição podem ser visualizados no gráfico da Figura 18. Nota-se a tendência que o método gRPC possui de alocar mais memória, assim como nos experimentos simples, com o servidor REST alocando cerca de 37% da alocação de memória do servidor gRPC na funcionalidade de criação, e de cerca de 54% da alocação de memória do servidor gRPC na funcionalidade de consulta com ordenação. Outro fator importante a ser considerado, é que a funcionalidade de consulta consumiu mais recursos, assim como teve um tempo médio de interação com o banco de dados maior (na Figura 17), o que reforça a hipótese de que a funcionalidade de consulta possui uma influência considerável no sistema e as diferentes implementações dos métodos possuem relação com isso.

Figura 18 - Gasto de RAM médio por combinações (100).

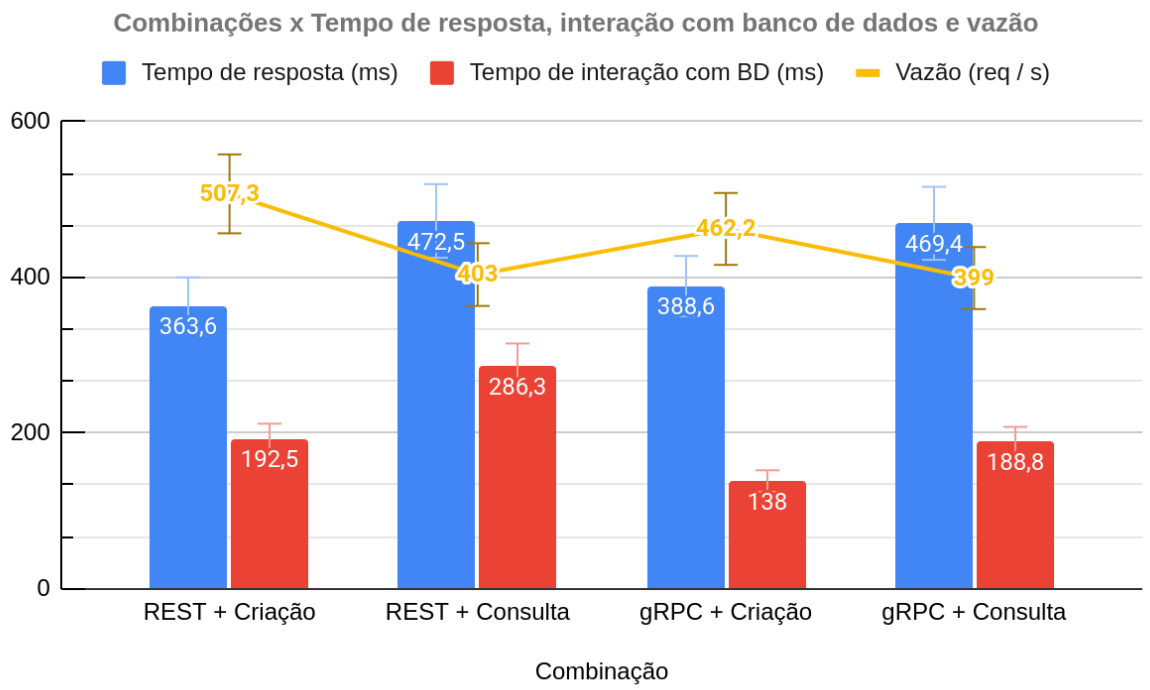


Fonte: elaborada pelo autor

5.3.2 Com 200 usuários simultâneos

Atuando com uma carga de trabalho de 200 usuários, foi possível entender melhor o desempenho do sistema e a aparição de possíveis gargalos, já que a taxa de falhas foi afetada neste cenário. Como evidenciado no gráfico da Figura 19, com uma carga de trabalho mais - a maior testada com todos os cenários até o momento - o REST só obteve desempenho superior ao gRPC na funcionalidade de criação, com 363,6 ms e 388,6 ms, respectivamente, enquanto na consulta, o gRPC se mostrou superior, reforçando que o gRPC possui desempenho que o REST em casos de consulta e envio de muitos dados na resposta. Assim como no cenário anterior, nos servidores gRPC, o tempo médio de interação com o banco de dados foi consideravelmente menor, confirmando a hipótese de uma interação menos onerosa com o banco de dados.

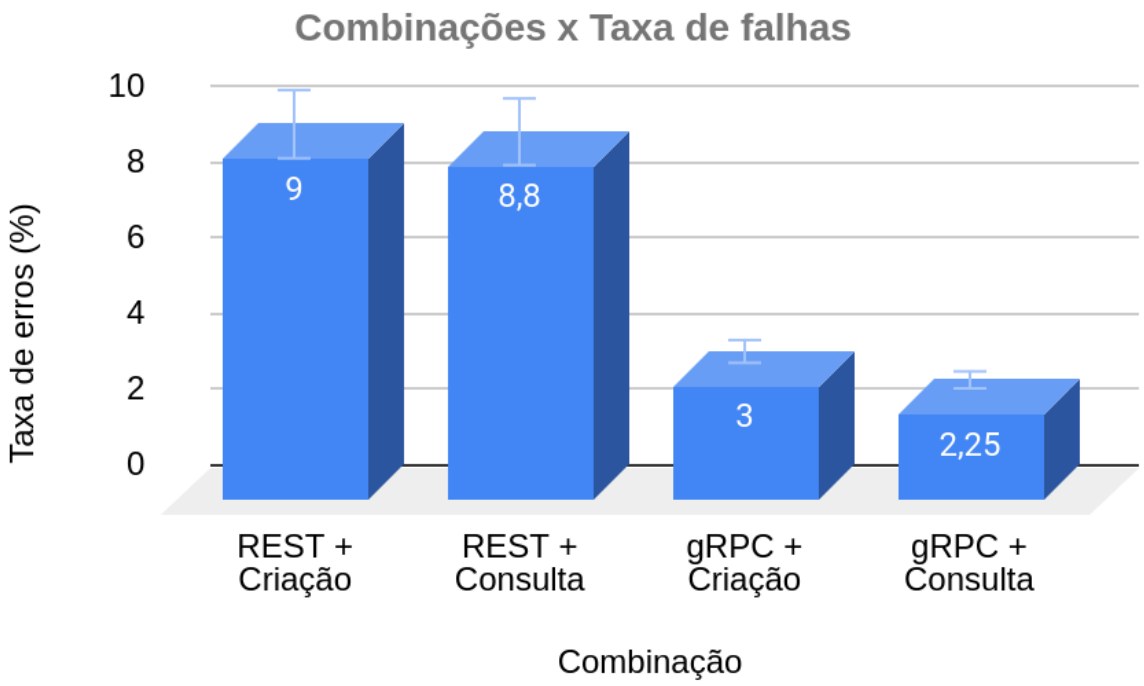
Figura 19 - Métricas relacionadas ao tempo por combinações (200).



Fonte: elaborada pelo autor.

Nota-se que a vazão também se mostrou coerente em relação aos respectivos tempos de resposta das combinações. Entretanto, com este cenário falhas no sistema foram identificadas, como demonstra o gráfico da Figura 20. Como mais um fator de diferença entre os métodos de comunicação, o REST apresentou cerca de três vezes mais falhas na criação quando comparado ao gRPC, e na consulta, foram 8,8% de falhas no REST e apenas 2,25% no gRPC.

Figura 20 - Taxa de falhas por combinações (200).



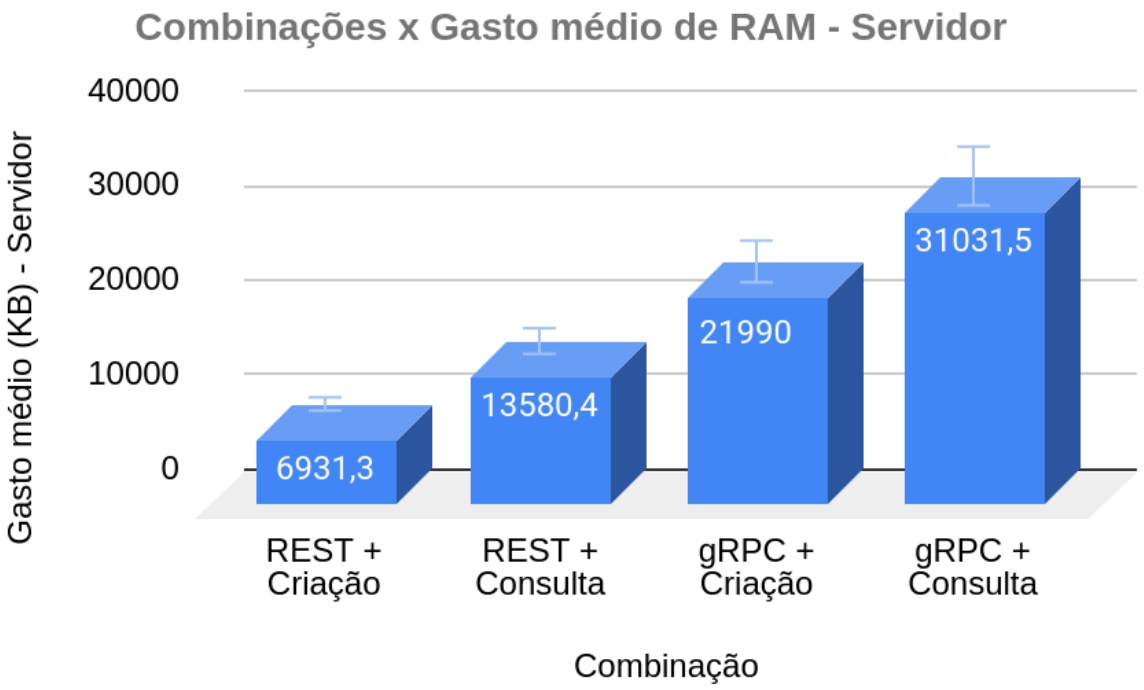
Fonte: elaborada pelo autor.

É fundamental destacar que todas as falhas que ocorreram foram no banco de dados PostgreSQL, e devido ao fato do limite de conexões abertas ser excedido. Com isso, entende-se que nesta condição o banco de dados foi um gargalo, e que é mais benéfico adotar um método de comunicação no qual o servidor lida com as requisições de forma que seja menos oneroso (como evidenciado no tempo médio de interação) e gere menos falhas: o gRPC. Ainda sobre o percentual de falhas que cada método e funcionalidade apresentou, entende-se que a vazão e o tempo médio de resposta foram afetados, pois uma requisição executada com falha idealmente se mostra mais rápida do que uma requisição executada sem erros, já que o tempo médio de interação com o banco de dados (que se mostrou relevante pelas análises) é praticamente nulo nestes casos. Logo, estima-se que o método com mais falhas (REST) pode ter tido tempos de respostas maiores ainda em comparação com os apresentados, e consequentemente uma vazão menor. Os resultados para gRPC também seguem esta lógica porém numa escala menor, visto que a taxa de falhas foi menor.

Como é possível visualizar no gráfico da Figura 21, os gastos médios de RAM nas aplicações servidores foram significativamente maiores no servidor gRPC,

sendo 21990 KB (cerca de 22 MB) na criação e 31031,5 KB (cerca de 31 MB) na consulta, enquanto o gasto médio no servidor REST foi de 6931,3 KB (cerca de 6,9 MB) e 13580,4 KB (cerca de 13,5 MB), logo, o REST gastou em média 31,5% e 43,76% dos recursos gastos pelo gRPC na criação e consulta, respectivamente. Conclui-se que o gRPC realmente é inferior ao REST no aspecto de gasto de recurso, e a proposta de mínima alocação de memória possível da implementação do *Fast HTTP* e Fiber para Golang se mostra verdadeira em relação a outro método de comunicação.

Figura 21 - Gasto de RAM médio por combinações (200).



Fonte: elaborada pelo autor.

6. CONSIDERAÇÕES FINAIS

6.1. Conclusões

Nos experimentos simples descritos no presente trabalho, nota-se que a abordagem de comunicação *message broker* se destacou de acordo com o tempo médio de resposta e recursos obtidos, entretanto, devido à divergência de tipo de comunicação, entendeu-se que a metodologia utilizada e poderia gerar resultados

divergentes e com pouca confiança para esta abordagem. Dada a complexidade de metrificação e aferição inerente à comunicação síncrona e assíncrona, a qual traria dificuldades em relação à arquitetura do sistema e parâmetros utilizados, optou-se por não prosseguir com a avaliação de desempenho para o mesmo. Por fim, não seria possível obter a taxa de falhas e a identificação de gargalos de desempenho seria prejudicada. Portanto, embora tenha tido resultados promissores, foi desconsiderada na análise final.

Entre as duas abordagens síncronas testadas, é importante evidenciar que ambas tiveram um desempenho consideravelmente bom, dado o contexto e os parâmetros de carga de trabalho que foram relevantes. Em relação aos tempos médios calculados, o REST apresentou melhor desempenho em cargas de trabalho menor ou desempenho similar em alguns cenários com as maiores cargas de trabalho, porém foi inferior em cenários que envolviam consultas e envio de muitas informações em resposta, e também com uma interação mais onerosa com o banco de dados. O ponto mais significativo positivamente para o REST foi o consumo médio de memória RAM por requisição, o qual foi consideravelmente menor do que em relação ao gRPC.

Por sua vez, em relação às métricas relacionadas à tempo, o gRPC apresentou desempenho vagarosamente inferior ou similar em alguns cenários, apresentando um desempenho melhor principalmente no cenários que envolvem consultas e envio de muitas informações, além de ter interagido com o banco de dados de forma menos onerosa e com menos falhas, o que foi demonstrado pelos valores menores de tempo médio de interação com o banco de dados por requisição e taxa de falhas (%). O ponto mais significativo negativamente para o gRPC foi o consumo médio de RAM por requisição, o qual foi consideravelmente maior que o REST, chegando a duas ou três vezes a alocação do outro nível.

É fundamental compreender que a principal evidência de que ambas abordagens tiveram um bom desempenho e suportam a carga de trabalho proposta é o fato de que, considerando o contexto simplificado e além dos valores de tempo e gasto de recursos, o gargalo identificado estava em outro componente do sistema distribuído: o banco de dados. Partindo da premissa do presente estudo e que o gargalo do banco não será tratado individualmente no momento, um bom caminho para se concluir sobre a eficiência das abordagens é priorizar a abordagem que menos onera o banco: o gRPC. O gRPC apresentou desempenho similar ou um

pouco pior que o REST, porém com menos falhas e menos tempo de interação com o banco de dados, indicando que o mesmo estava menos sobrecarregado. Outra vantagem evidente do gRPC em relação ao REST é a forma que se transmite os dados, relevantes em consultas, pois como define (HILLPOT, 2023), o gRPC utiliza *Protocol Buffers*, um formato de serialização binário que resulta em uma resposta menor e comunicação mais rápida, enquanto REST utiliza o JSON ou XML, que são formatos baseados em textos. Embora tenha um gasto de memória RAM significativamente maior, dadas as configurações do contêiner utilizado, não foi constatado um impacto negativo. De qualquer forma, é uma característica do Go a utilização de poucos recursos.

Por fim, é importante entender a relevância dos resultados do REST neste caso, apresentando um desempenho excelente, lidando com muitas requisições simultaneamente e sendo extremamente otimizado em relação à recursos. Entende-se que, caso o gargalo do banco possa seja tratado individualmente, aumentando o número de conexões simultâneas ou adotando estratégias para interagir de forma menos onerosa, a utilização de REST se mostra uma excelente opção, sendo até superior ao gRPC, dado que conseguiu lidar de forma mais performática com a carga de trabalho. A velocidade aliada à otimização na alocação de memória pode permitir ambientes com menos recursos (sem comprometer o desempenho) e com menos custo, por exemplo, instâncias menores e mais baratas no EC2 da AWS. Logo, o REST também é uma opção viável caso o gargalo com o banco de dados for sanado.

6.2. Comparação com trabalhos relacionados

Em (OLIVEIRA E SAMPAIO, 2021) foi realizada uma análise de eficiência entre REST e gRPC, e os casos de testes tinham como fator a quantidade de registros retornados nas chamadas de ambos os métodos, e de acordo com os cenários propostos, foi concluído que o gRPC apresenta melhor desempenho, principalmente ao aumentar a quantidade de registros retornados pelas chamadas, e assim como no presente trabalho, entende-se como uma possível justificativa o formato de serialização para binário feito pela utilização dos Protocol Buffers. É importante ressaltar que, além dos fatores considerados, uma diferença relevante entre o presente trabalho e o citado é o ambiente que os testes rodam e a

implementação do projeto, pois o trabalho citado teve como objeto de análise uma API RESTful robusta desenvolvida utilizando .NET Core (*framework*) e C#, e o presente trabalho utiliza um protótipo simples em Golang e Fiber (para implementar as APIs RESTful). A diferença entre linguagens de programação e *frameworks* podem ser um fator que resulte em diferenças, visto que recursos e características da linguagem possuem impactos diretos em desempenho.

Em (HONG, YANG E KIM, 2018), como mencionado no referencial teórico, foram testadas as abordagens REST e *message broker* (RabbitMQ), e foi constatado o RabbitMQ como mais eficiente e estável do que o REST, principalmente quando as quantidades de usuários simultâneos realizando requisição eram grandes. A principal diferença entre o presente trabalho foi a metodologia e a metrificação, pois os experimentos foram executados durante um tempo de 15 minutos, foram comparados uma *API Gateway* com um serviço de RabbitMQ, o qual o *API Gateway* distribui as requisições para as aplicações e o RabbitMQ para as filas e serviços.

6.3. Trabalhos futuros

Como trabalhos futuros, compreende-se que realizar o cálculo de influência dos fatores nas métricas com base nos resultados de experimentos completos pode ser importante para a visualização de padrões entre os diferentes cenários e efeitos. Ainda dentro da área de avaliação de desempenho, pode ser vantajoso conduzir testes com o contexto, fatores e níveis propostos em outros ambientes, como máquinas virtuais, e hospedados em servidores na internet de serviços de nuvem como AWS (ou outro IaaS), pois assim poderiam ser considerados fatores como maior isolamento de recursos e latência de rede.

Além disso, como foi evidenciado um gargalo no banco de dados com a aplicação de uma determinada carga de trabalho, é relevante entender os motivos e ser apto a realizar uma avaliação de desempenho específica para banco de dados, com diferentes fatores, níveis, carga de trabalhos e métricas, ou então ser conduzido um estudo sobre conceitos e práticas que possam mitigar o risco de erros em bancos de dados em cargas altas de usuários simultâneos. Por fim, entende-se que é relevante um estudo em relação ao REST e ao gRPC, visto que são duas abordagens de comunicação síncronas que utilizam diferentes versões do protocolo

HTTP (1.1 e 2, respectivamente), além de outras diferenças. Portanto, pode ser extremamente valioso um estudo mais detalhista e que pode considerar fatores de baixo nível.

REFERÊNCIAS

TANENBAUM, Andrew S et al. Sistemas distribuídos : princípios e paradigmas. Brasil: Pearson Education do Brasil, 2008.

COULOURIS, George et al. Sistemas Distribuídos - Conceito e Projeto. 5. ed. [S.l.]: bookman, 2013.

KUROSE, James F; ROSS, Keith W. Computer networking. Kurose, Keith W. Ross. Harlow: Pearson Education, 2012.

RAJ JAIN. The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling. New York: Wiley, 1991.

FOWLER, Susan. Microsserviços prontos para a produção: construindo sistemas padronizados em uma organização de engenharia de software. Brasil: Novatec, 2017.

NEWMAN, Sam. Building microservices: designing fine-grained systems. O'Reilly Media, Inc., 2015.

A. E. Bagaskara, S. Setyorini and A. A. Wardana, Performance Analysis of Message Broker for Communication in Fog Computing. 2020 12th International Conference on Information Technology and Electrical Engineering (ICITEE), Yogyakarta, Indonesia, 2020, pp. 98-103, doi: 10.1109/ICITEE49829.2020.9271733.

X. J. Hong, H. Sik Yang and Y. H. Kim, Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application. 2018 International Conference on

Information and Communication Technology Convergence (ICTC), Jeju, Korea (South), 2018, pp. 257-259, doi: 10.1109/ICTC.2018.8539409.

OLIVEIRA, João Victor Marques Oliveira. Analisando a eficiência da comunicação entre serviços utilizando REST ou gRPC. 2021. 17 f. TCC (Especialização) - Residência em Tecnologia da Informação, Departamento de Ciência da Informação, Universidade Federal do Rio Grande do Norte, Natal, 2021.

ALKHODARY, SAMER, The Evaluation of Using Backend-For-Frontend in a Microservices Environment. Department of Computer Science, LTH, Lund University, 2022, ISSN 1650-2884.

W3SCHOOLS. W3Schools, S.I. What is JSON. <https://www.w3schools.com/whatis/whatis_json.asp>. Acesso em: 18 jan. 2023.

GRPC AUTHORS. gRPC, S.I. Introduction to gRPC. Disponível em: <https://grpc.io/docs/what-is-grpc/introduction/>. Acesso em: 18 jan. 2023.

GOOGLE LLC. Google, S.I. Protocol Buffers Documentation . Disponível em: <https://protobuf.dev/overview/>. Acesso em: 25 set. 2023.

AMAZON WEB SERVICES INC. AWS, 2023. Tipos de instâncias. Disponível em: <https://aws.amazon.com/pt/ec2/instance-types/>. Acesso em: 02 out. 2023.

DOCKER INC. Docker, 2023. Docker overview. Disponível em: <https://www.docker.com/resources/what-container/>. Acesso em: 26 set. 2023.

POSTGRESQL. 2023. About. Disponível em: <https://www.postgresql.org/about/>. Acesso em: 04 set. 2023.

LOCUST. 2023. What is Locust. Disponível em: <https://docs.locust.io/en/stable/what-is-locust.html>. Acesso em: 04 set. 2023.

GO. 2023. Use cases. Disponível em: <https://go.dev/solutions/use-cases>. Acesso em: 01 set. 2023.

GO. 2023. database/sql. Disponível em: <https://pkg.go.dev/database/sql>. Acesso em: 01 set. 2023.

HILLPOT, JEREMY. DreamFactory, 2023. gRPC vs REST: Key Similarities and Differences. Disponível em: <https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/>. Acesso em: 28 nov. 2023.

LIB. lib/pq. Pure Postgres driver for database/sql. Repositório no GitHub. Disponível em <https://github.com/lib/pq>. Acesso em: 28 set. 2023.

GOFIBER. Fiber. Express inspired web framework written in go. Disponível em <https://gofiber.io/>. Acesso em 28 set. 2023.

VALIALKIN, ALIAKSANDR. Fast HTTP. Repositório no GitHub. Disponível em <https://github.com/valyala/fasthttp>.. Acesso em: 28 set. 2023.

STREADWAY, SEAN. amqp. Go client for AMQP 0.9.1. Repositório no GitHub. Disponível em <https://github.com/streadway/amqp>. Acesso em: 28 set. 2023.