Universidade Federal de Pernambuco

Departamento de Ciência da Computação

Curso de Ciência da Computação

Avaliação do Impacto de Balanceadores de Carga sobre o gRPC

Proposta de Trabalho de Conclusão de Curso de Graduação

por

João Filipe da Matta Ribeiro Moura

Orientador: Nelson Souto Rosa

Novembro / 2022

João Filipe da Matta Ribeiro Moura

Avaliação do Impacto de Balanceadores de Carga sobre o gRPC

Monografia apresentada ao Curso de Ciência da Computação, como requisito parcial para a obtenção do Título de Bacharel em Ciência da Computação, Centro de Informática da Universidade Federal de Pernambuco.

Orientador: Nelson Souto Rosa

Recife

2022

João Filipe da Matta Ribeiro Moura

Avaliação do Impacto de Balanceadores de Carga sobre o gRPC

Monografia apresentada ao Curso de Ciência da Computação, como requisito parcial para a obtenção do Título de Bacharel em Ciência da Computação, Centro de Informática da Universidade Federal de Pernambuco.

Aprovado em: 10 de Outubro de 2022

Banca Examinadora:

Prof. Dr. Nelson Souto Rosa (Orientador)
Centro de Informática da Universidade Federal de Pernambuco

Prof. Dr. Vinicius Cardoso Garcia
Centro de Informática da Universidade Federal de Pernambuco

Recife

2022

Agradecimentos

Eu gostaria de agradecer a todos vocês que me ajudaram durante esta jornada, especialmente para:

Jesus Cristo, meu Senhor e o único digno de todo mérito que a mim foi concedido; a meus pais que sempre incentivaram meu estudo, e através da criação construíram em mim um caráter forte, resiliente e disciplinado; aos meus avós, e melhores companheiros de quarto, Olivia e Adelino, que me acolheram como um filho; a minha esposa, minha maior companheira e maior apoiadora; ao meu irmão mais novo, Pedro, ser uma referência para você sempre foi uma das minhas maiores motivações; a toda minha família, que me acolheu em Recife, e mesmo quando não havia condições de me sustentar não me deixaram faltar nada.

Ao professor Nelson, por toda sua paciência durantes esses 2 anos de orientação e a tantos outros por aumentarem minha paixão pela computação.

Aos amigos e companheiros de curso, em especial Ramon, Tato, Rossi, Douglas, Nunes, Cunha, Lula por todas as noites em claro fazendo inúmeros projetos juntos.

Aos irmãos dos intervalos bíblicos, que foram uma fortaleza para minha vida, em especial, minha esposa Natália, Valter, Gabi, Douglas, Vitinho, Cesinha, Pedro, Renata, Joelma, Katia, vocês fortaleceram minha fé, junto vimos os milagres do Senhor dentro na universidade e vivemos a unidade da Igreja de maneira singular.

A Schub, e todos os seus integrantes, em especial, Luis, Gastón e Eduardo, por terem possibilitado a realização deste trabalho, arcando com os custos de infraestrutura, e com todo o apoio. Muito mais que companheiros de trabalho, vocês são família.

Deus joga dados onde ningém pode ver.

Stephen Hawking

RESUMO

Orquestradores de contêineres têm sido amplamente utilizados para melhorar a confiabilidade, desempenho, escalabilidade e gerenciar os sistemas distribuídos, bem como os recursos utilizados por eles, e.g., memória, processamento. A exemplo do Kubernetes que se tornou o padrão entre os orquestradores de contêineres utilizados pela indústria.
Para realizar a comunicação entre aplicações executadas em diferentes contêineres, configurando uma arquitetura de microsserviços, costuma-se utilizar *Google Remote Procedure Call* (gRPC) que é um *middleware* que facilita o uso de *Remote Procedure Call* (RPC). Ele é normalmente utilizado em aplicações que têm um requisito de latência muito rigoroso. Dessa forma, neste trabalho buscou-se realizar uma análise comparativa do impacto de latência de um balanceador de carga do tipo *Proxy* e *Client Side* na comunicação entre serviços através do gRPC. O objetivo foi gerar dados para auxiliar times responsáveis pelo *deploy* e arquitetura de microsserviços no design ou otimização dos seus sistemas levando em consideração a latência, visto que em casos extremos de otimização de performance mesmo pequenas adições de latência podem fazer toda a diferença. Para isso, os experimentos foram executados em um *cluster* Kubernetes em diferentes cenários de carga. Os experimentos apresentaram uma adição de latência média de 1 milissegundo do balanceador do tipo *Proxy* em relação ao *Client Side*, somado a uma utilização maior de recursos computacionais. Porém o *Proxy* traz mais funcionalidades, fazendo-se uma opção preferível caso os pontos negativos não sejam impeditivos para o contexto do sistema.

Keywords: gRPC, Balanceador de Carga, Proxy, Client Side, Kubernetes, Sistemas Distribuídos, Microsserviços.

ABSTRACT

Container orchestrators have been widely used to enhance the reliability, performance, scalability, and management of distributed systems, as well as the resources utilized by them, e.g., memory and processor like Kubernetes, that became the pattern between container orchestrators used in the industry.

To communicate between applications executed in different containers in a microservice architecture, usually, *Google Remote Procedure Call* gRPC is used, which is a middleware that makes the usage of Remote Procedure Call RPC more straightforward. It is mainly used in applications that have an extremely rigorous latency requirement. So, this paper has aimed to build an analysis on the difference of latency impact between load balancers of type Proxy and Client Side on gRPC communication between services. The goal was to generate reliable data to help teams responsible for deploying and architect microservices, or systems optimization, considering the latency, considering the fact that in extreme scenarios even small amounts of latency can make a big impact on the system. The experiments have been executed in a Kubernetes cluster in different load scenarios. The experiments revealed an average latency addition of 1 millisecond of the Proxy load balancer and increased the usage of computational resources compared to the Client Side load balancer. On the other hand, the Proxy brings more functionalities, which is preferable if the downsides are not blockers in the system context.

Keywords: gRPC, Load Balancer, Proxy, Client Side, Kubernetes, Distributed Systems, Microsservices

Lista de Figuras

Conteúdo

LISTA DE SIGLAS

gRPC *Google Remote Procedure Call*

RPC *Remote Procedure Call*

HTTP *Hiper Text Transfer Protocol*

DNS *Domain Name Service*

GCP *Google Cloud Platform*

AWS *Amazon Web Services*

1 INTRODUÇÃO

1.1 Motivação

Com o aumento da demanda e complexidade de soluções na área da computação e tecnologia da informação, cada vez mais sistemas distribuídos estão sendo implementados como uma necessidade e alternativa aos antigos servidores centralizados [8]. A distribuição de recursos computacionais entre vários servidores numa mesma rede, traz uma redução de riscos por não ter um único ponto de falha, escalabilidade automática de recursos computacionais e redução de custos em muitos casos [17]. E apesar de terem esses e muitos outros benefícios, um ponto negativo inerente à sua natureza é o aumento da latência do sistema quando comparado a uma aplicação monolítica, devido a todo o processamento extra adicionado pelos sistemas de *middleware* e a latência da rede [16]. Adicionalmente, essa característica pode ser ainda mais potencializada ao adicionar-se balanceadores de carga ao sistema.

Mensurar esse impacto de latência é algo extremamente difícil de se realizar, principalmente nas fases iniciais de definição da arquitetura do sistema por conta da grande quantidade de parâmetros que podem interferir no funcionamento do sistema. Ademais, ainda há a escassez de dados sólidos de desempenho das tecnologias mais usadas em ambientes de produção. Por isso, faz-se necessário haver mais trabalhos e documentações no auxílio desses tipos de mensurações.

1.2 Problema

Em um contexto de sistemas distribuídos, dentre inúmeros requisitos, dois se destacam: escalabilidade horizontal[11] e tolerância à falha[10]. A escalabilidade horizontal consiste na habilidade de aumentar a capacidade de resposta do sistema através da adição de réplicas, enquanto que a tolerância à falha é a habilidade do sistema de continuar funcionando ainda que parte de suas réplicas estejam danificadas. Para alcançar esses dois requisitos, várias técnicas são utilizadas, dentre elas uma das principais são os balanceadores de carga, que são aplicações especializadas em distribuir requisições entre múltiplas réplicas de um servidor. Porém essas funcionalidades possuem como consequência um aumento de latência.

Apesar desse aspecto a princípio negativo, os balanceadores de carga ainda assim são necessários na implementação de sistemas distribuídos, e para isso existem dois métodos principais para fazer o balanceamento de carga do *gRPC*: o balanceamento *Client Side* e o do tipo *Proxy*. O primeiro é implementado no próprio cliente. O segundo trata-se de uma aplicação especializada que intermedeia os clientes e as réplicas servidoras envolvidas, cuja natureza implica em um aumento ainda maior de latência devido ao passo de rede extra necessário.

1.3 Soluções Atuais

O trabalho de Soares (2021) [14] teve como objetivo realizar uma análise de desempenho comparativa entre diferentes *middleweres*. Foi realizado uma avaliação de desempenho comparativa entre *RPC-QUIC* e *gRPC*, concluindo que o *RPC-QUIC* reduz o tempo de latência no estabelecimento de conexões, porém se mostra menos eficiente no transporte de dados grandes.

Trabalhos existentes sobre o desempenho de balanceadores de carga encontram-se majoritariamente em classificar e comparar algoritmos de balanceamento, como Mishra (2020) [12] que explorou a importância de algoritmos de balanceamento de carga, bem como realizou uma avaliação de desempenho e eficiência de diversos algoritmos de balanceamento de carga utilizados em ambientes de nuvem. Mas não foram encontrados trabalhos trazendo dados experimentais relacionados a casos de uso comuns na indústria.

1.4 Objetivo

Este trabalho visa realizar uma avaliação de desempenho comparativa entre dois tipos de balanceadores em relação a adição de latência que o *Proxy* implica em relação ao *Client Side*, Espera-se que os resultados da avaliação possam auxiliar no projeto e deploy de aplicações que usam balanceadores de carga.

1.5 Estrutura do Documento

Este trabalho está estruturado em 4 capítulos, incluindo este capítulo inicial. O capítulo 1 contempla a introdução do trabalho. O capítulo 2 introduz os conceitos necessários para o entendimento do trabalho. O capítulo 3 apresenta os experimentos e os

resultados. Finalmente, o Capítulo 4 apresenta as conclusões e os trabalhos futuros.

2 CONCEITOS BÁSICOS

Neste capítulo serão descritos todos os conceitos necessários para entender este trabalho.

2.1 Balanceamento de Carga *Client Side*

Nesta estratégia de balanceamento cada réplica do cliente irá se responsabilizar por balancear a carga entre as réplicas do servidor. Para isso, o cliente abre um conjunto de conexões com cada réplica do servidor e a cada operação, o algoritmo de balanceamento irá escolher uma conexão para utilizar.

Existe ainda um observador, que em um intervalo determinado ou na ocasião de um erro de conexão, atualiza o conjunto de conexões para garantir que ele esteja sempre atualizado. Essa atualização ocorre quando uma réplica do servidor é encerrada, por exemplo com uma falha no servidor, ou mais réplicas são criadas. Sem esse observador, poderiam haver sucessivos erros de conexão ou haver um desbalanceamento de carga, já que as novas réplicas do servidor nunca receberiam requisições. A arquitetura quando um balanceador *Client Side* é utilizado pode ser observada na Figura 1.

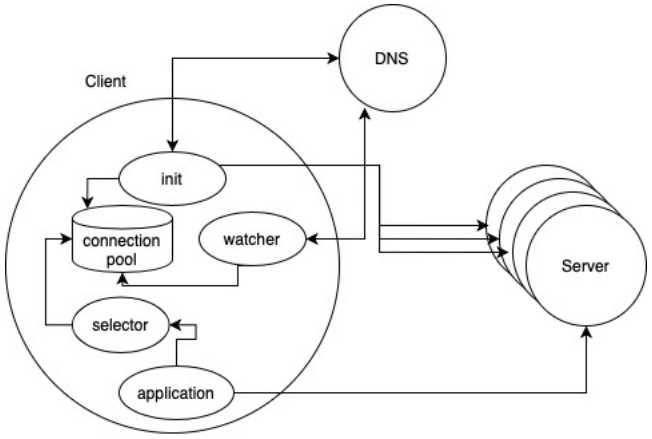

Figura 1: Balanceamento *Client Side*

Quando comparado a balanceadores de carga do tipo *Proxy*, o *Client Side* não tem a necessidade de fazer um passo extra de rede, visto que não passa por uma aplicação intermediária para se comunicar com o servidor. Essa característica traz uma vantagem em termos de latência. Entretanto, o seu principal ponto negativo é o custo de implementação e manutenção, onde cada cliente precisará implementar o seu método de balanceamento.

Esta característica aumenta o número de pontos de falha, podendo causar sobrecargas nos servidores em caso de interrupções no serviço. Além disso, os clientes precisam ser confiáveis, visto que a responsabilidade de balancear a carga será de cada um deles.

2.2 Balanceador *Proxy*

Existem vários tipos de balanceadores de carga do tipo *Proxy*. Suas categorias são divididas pela camada de rede em que é implementado. Neste trabalho, o balanceador de carga utilizado para os experimentos é o da camada de aplicação, visto que o gRPC[6] é baseado em *Hiper Text Transfer Protocol* (HTTP) 2.0, um protocolo da camada de aplicação. Todas as referências ao *Proxy* aqui estarão se referindo ao *Proxy* da camada de aplicação.

Balanceadores do tipo *Proxy* funcionam como um ponto central que tem um conjunto de conexões para cada réplica do servidor. Desta forma, como o balanceador *Client Side*, eles também possuem um observador que procura por mudanças no conjunto de servidores. A grande diferença é que no caso do *Proxy* toda essa complexidade fica concentrada em si mesmo, que dessa forma abstrai a responsabilidade de balanceamento do cliente. Tudo que um cliente precisa fazer nesse caso é abrir uma conexão (ou um conjunto de conexões) com o *Proxy*, e ele lida com o restante. Essa arquitetura está representada na Figura 2.

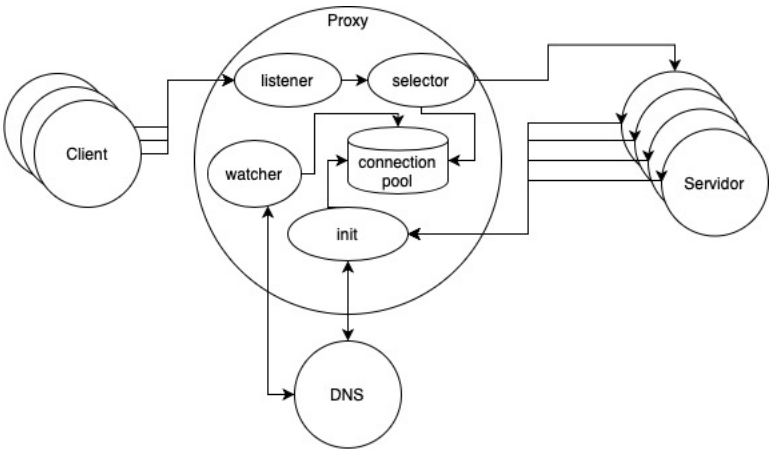

Figura 2: Balanceador Proxy

As principais vantagens desse método são a maior simplicidade de implementação e manutenção, além de facilitar a observabilidade, cuja característica ajuda na inves-

tigação durante interrupções no serviço. Já as principais desvantagens são adição de latência devido ao processamento extra. Este processamento adicional ocorre por conta da necessidade de desempacotar e reempacotar a requisição, e também por se tornar um único ponto de falha, podendo comprometer a disponibilidade do servidor se não for bem mantido.

2.3 *gRPC*

gRPC[6] é um middleware que utiliza o HTTP2.0 e com o codificador *Protocol Buffer*. Pelo fato de utilizar o HTTP2.0 implica em conexões TCP persistentes, que idealmente devem durar o tempo de vida da aplicação. Esta característica reduz significativamente o tempo gasto com aberturas de conexão, como acontece no HTTP1.1.

O *Protocol Buffer* também contribui para a eficiência do *gRPC* visto que seu processo de serialização é extremamente eficiente e a mensagem final é muito pequena.

2.4 Contêiner

Contêiner é uma máquina virtual minimalista, que faz uso de uma tecnologia de virtualização mais leve e imagens com o mínimo de recursos instalados para executar uma aplicação.

Apesar de ter vantagens em relação às máquinas virtuais, uma imagem de contêiner não pode ser instalado direto em máquinas físicas, visto que é uma tecnologia de virtualização, logo necessita de máquinas hospedeiras para ser executado. Esta limitação torna necessária uma forma automatizada de orquestrar contêineres em máquinas hospedeiras. O Kubernetes[9] é justamente um orquestrador de contêineres e se tornou a principal ferramenta na indústria.

3 AVALIAÇÃO COMPARATIVA

3.1 Objetivos

Este trabalho tem como objetivo fazer uma análise comparativa sobre o impacto de latência entre as estratégias de balanceamento de carga *Proxy* e *Client Side*. Dessa forma, busca-se fornecer informação para auxiliar no processo de tomada de decisão de times responsáveis pelo *deploy* e arquitetura de microsserviços possibilitando-os estimar de modo mais preciso a latência geral do seu sistema.

Esse objetivo será alcançado a partir dos seguintes passos[7]:

1. Design dos experimentos.
2. Definição das métricas a serem observadas.
3. Definição de fatores e parâmetros.
4. Definição de cenários que simulem um ambiente de produção.
5. *Deploy* de um cluster de Kubernetes em alta disponibilidade.
6. Desenvolvimento e instrumentação de uma aplicação servidora simples.
7. Configuração de um sistema de monitoramento.
8. Execução dos experimentos.
9. Análise comparativa das métricas coletadas.

3.2 Métricas, Parâmetros e Fatores

A métrica a ser observada em cada componente dos experimentos (Cliente, *Proxy*, Servidor) é a latência, que é o tempo decorrido entre o início e o fim de cada requisição. Além disso, no Cliente e no *Proxy* serão observados o uso de CPU, e no Servidor a taxa de requisições por réplica. Por sua vez, o uso de CPU é medido como o tempo que cada aplicação ocupa o processador por minuto, e a taxa de requisições por réplica do servidor é a quantidade de requisições por segundo recebida por cada réplica.

As métricas serão coletadas usando o sistema de monitoramento *Prometheus*[13] e serão visualizadas com o *Grafana*[5], com exceção da latência do Cliente. Esta métrica será monitorada com o gerador de carga utilizado. A coleta da latência necessita de instrumentação do código da aplicação. No caso do Cliente e do *Proxy* a instrumentação já está inclusa no código da ferramenta, no Servidor, por sua vez, será necessário implementá-la. A métrica de CPU será coletada pelo Prometheus, por um exportador chamado *cAdvisor*[2], que monitora e consolida métricas de recursos utilizados por contêineres. Para coletar a taxa de requisições por réplica foi necessário fazer a instrumentação do servidor usando a biblioteca do Prometheus.

Uma vez definidas as métricas é necessário definir tudo que pode afetá-las, ou seja definir os parâmetros da avaliação:

1. Taxa de requisições, sendo a quantidade de requisições por segundo executadas pelo cliente;
2. Número de requisições concorrentes, ou seja a quantidade de operações sendo realizadas em paralelo;
3. Número de conexões abertas pelo cliente.
4. Assinatura da carga, sendo a variação da taxa de requisições em um determinado período de tempo;
5. Estratégia de balanceamento;
6. Largura de banda da rede.

Alguns destes parâmetros serão variados durante os experimentos (fatores):

1. Assinatura da carga (crescente, decrescente e constante);
2. Estratégia de balanceamento (*Proxy L7* e *Client Side*);
3. Taxa de requisições, medida em requisições por segundo (r/s). (500/s, 1000/s, 1500/s).

Os parâmetros escolhidos são os que tem relação mais direta com a efetividade do balanceamento de carga em si, visto que o objetivo do trabalho é comparar duas

estratégias de balanceamento. Os outros parâmetros poderiam ser também variados, mas expandiriam o escopo da análise e não adicionariam informação relevante ao objeto de análise do trabalho.

Com relação aos níveis considerados para os fatores no caso da assinatura de carga, os níveis escolhidos são suficientes para trazer representatividade sobre os casos mais difíceis para um balanceador de carga, e.g., manter o balanceamento mesmo em cenários de mudança na taxa de requisição. Considerar outros níveis não adicionaria cenários representativos para justificar o aumento de complexidade dos experimentos. As taxas de requisições foram definidas após um experimento inicial que definiu o nível em que o proxy começaria a degradar seu desempenho sem escalabilidade horizontal. A partir disso foram definidos 3 níveis de carga: leve, moderado e pesado.

3.3 **Experimentos**

Um *cluster* Kubernetes[9] foi configurado com 3 máquinas com 2 CPUs e 8GB de memória RAM, cada uma em uma Zona de Disponibilidade diferente afim de representar um cenário de rede habitual em ambientes de produção representado na Figura 3.

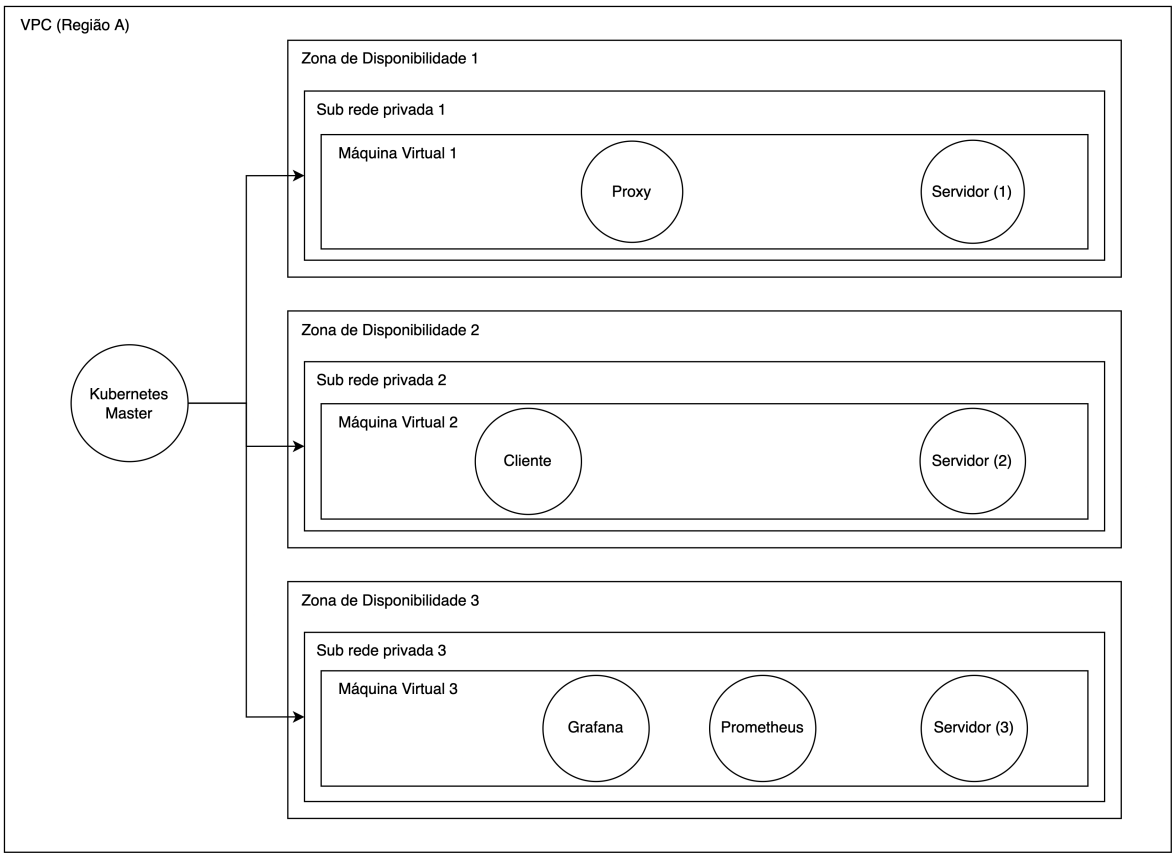

Figura 3: Configuração do Ambiente

Uma aplicação servidora que responde aos clientes com uma mensagem estática foi implementada. O servidor foi executado com 3 réplicas de modo que cada réplica seja executada em uma máquina diferente. O *Deploy* foi configurado para ter CPU e memória garantidos, para minimizar problemas de competição de CPU com as outras aplicações.

A carga foi gerada por uma ferramenta de teste de carga chamada *GHZ*[3]. Ela nos permite parametrizar o teste de carga de diversas formas, incluindo todos os parâmetros definidos para esse experimento, e gera como saída um arquivo com medições de latência de todas as requisições realizadas no experimento. Esses dados foram importados para um banco de dados através da ferramenta *Apache Superset*[1] para serem analisados e gerar visualizações.

Para o *Proxy* foi escolhido o *Traefik*[15] por possuir suporte avançado para HTTP2.0. Além disso, é escrito em *Go*[4] bem como a aplicação servidora e a ferramenta de balanceamento de carga, facilitando assim a análise dos resultados e configurações. O *Proxy* foi executado com apenas uma réplica para diminuir a quantidade de tráfego necessário para saturar sua capacidade.

Cada experimento teve duração de 6 minutos e havia apenas um cliente executando por vez.

No total foram realizados 10 experimentos, 5 utilizando o método *Client Side* e 5 utilizando o *Proxy* com correspondência de 1 pra 1 afim de comparar diretamente o comportamento da latência nos 2 cenários.

O balanceamento utilizando o *Client Side* foi escolhido como valor de referência por ser a forma de balanceamento de carga mais direta possível, visto que não é necessário nenhum intermediário.

3.4 **Resultados e Análise**

Nessa análise os resultados sempre serão apresentados primeiro os resultados dos experimentos *Client Side* e depois *Proxy*, e em cada gráfico a ordem de execução será:

1. Carga Constante - 500r/s.
2. Carga Constante - 1000r/s.
3. Carga Constante - 1500r/s.
4. Carga Decrescente.
5. Carga Crescente.

3.4.1 **Análise da Latência**

A Figura 4 e Figura 5 mostram a latência média quando os dois métodos de balanceamento são utilizados. Observa-se nestas figuras que ambos possuem comportamentos parecidos com a variação de taxa de requisição, aumentando levemente com o aumento dessa taxa. A diferença entre os dois métodos é a ordem de grandeza da escala de tempo; o *Proxy* em milissegundos, e *Client Side* em microssegundos. Esse era exatamente o comportamento esperado, visto que o *Proxy* adiciona um passo a mais de rede e processamento ao sistema. Porém, para confirmar a estes resultados, métricas adicionais foram também observadas.

Figura 4: Média de Latência Experimentos *Client Side* - Cliente

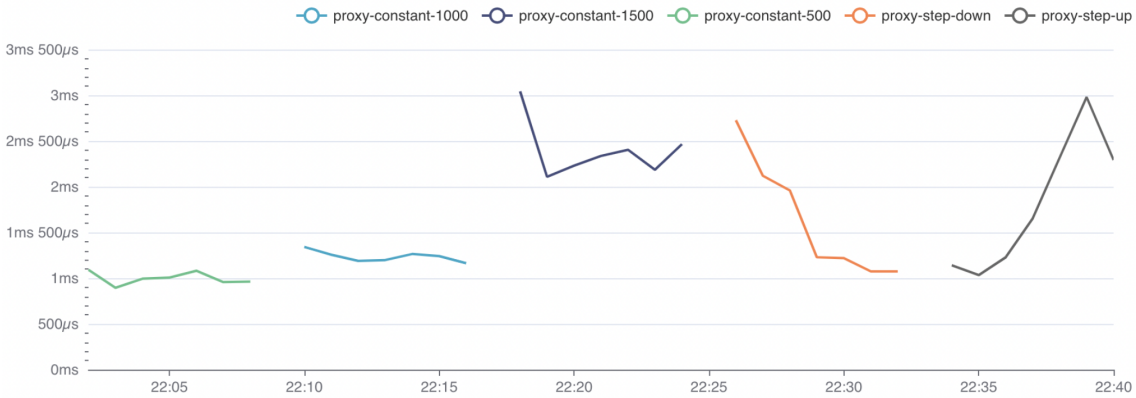

Figura 5: Média de Latência Experimentos *Proxy* - Cliente

3.4.2 **Análise das métricas do servidor**

A Figura 6 e a Figura 7 mostram as taxas de requisição que chegam em cada uma das três réplicas do servidor. Nestes gráficos é possível observar um balanceamento quase perfeito entre as três réplicas. Em cada figura há três curvas, cada uma representa a quantidade de requisições que cada servidor recebeu por segundo, e elas apresentaram um comportamento muito semelhante ao longo do tempo, significando que cada réplica está recebendo a mesma quantidade de requisições.

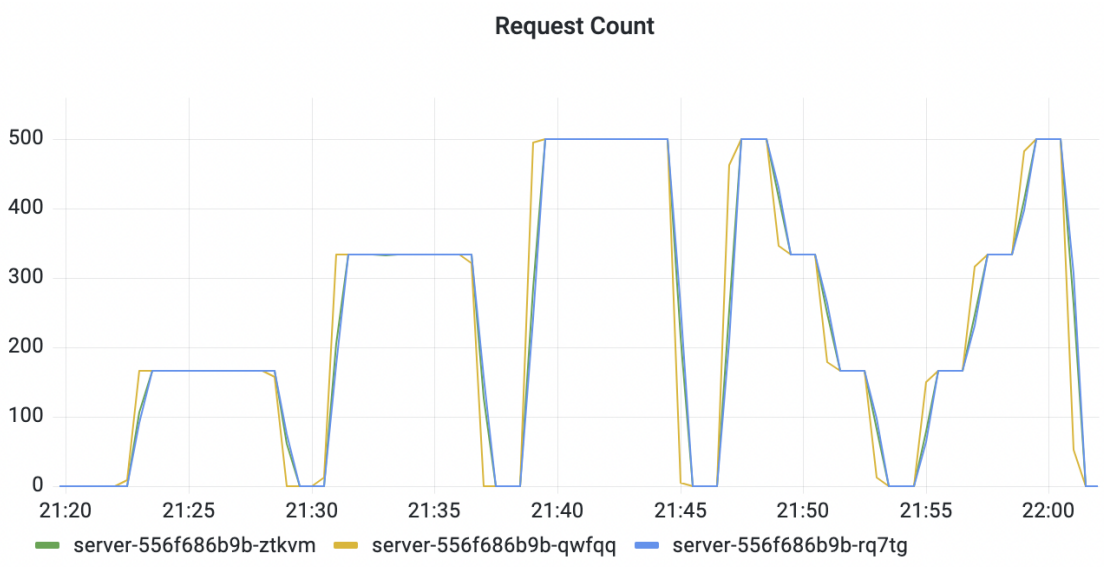

Figura 6: Distribuição de Carga - *Client Side*

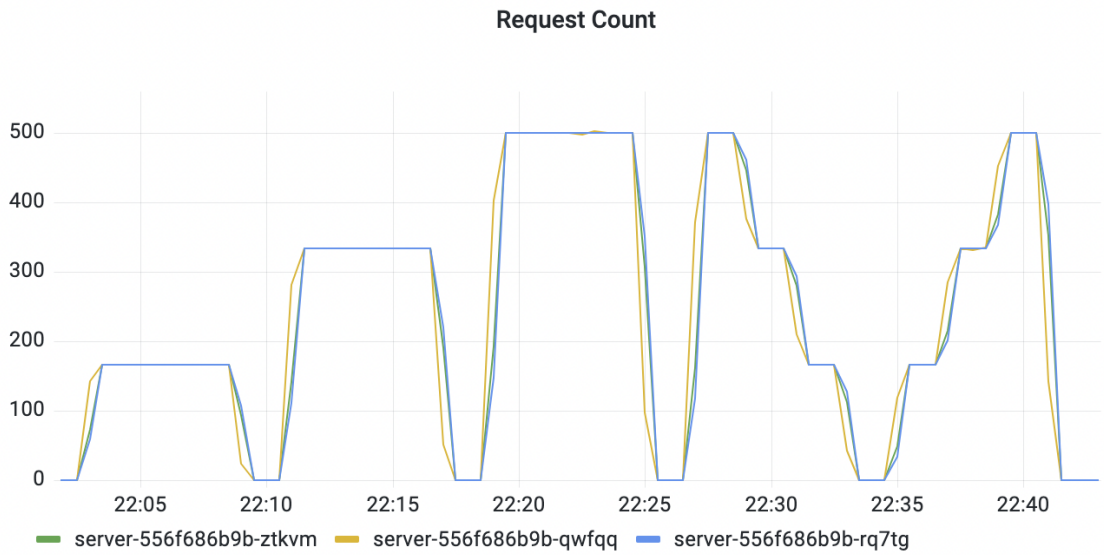

Figura 7: Distribuição de Carga - *Proxy*

A Figura 8 e a Figura 9 confirmam a efetividade do balanceamento ao observar ainda o tempo de processamento percebido no servidor. Este é o tempo gasto pelo Servidor para processar as requisições, sem contar o tempo decorrido no servidor *Web*. Nota-se que não há mudança significativa com a variação de assinatura de carga, taxa de requisições e nem método de balanceamento. Na verdade esse tempo é praticamente desprezível para esse contexto, o que faz sentido, visto que a aplicação se trata de um método que retorna um valor estático.

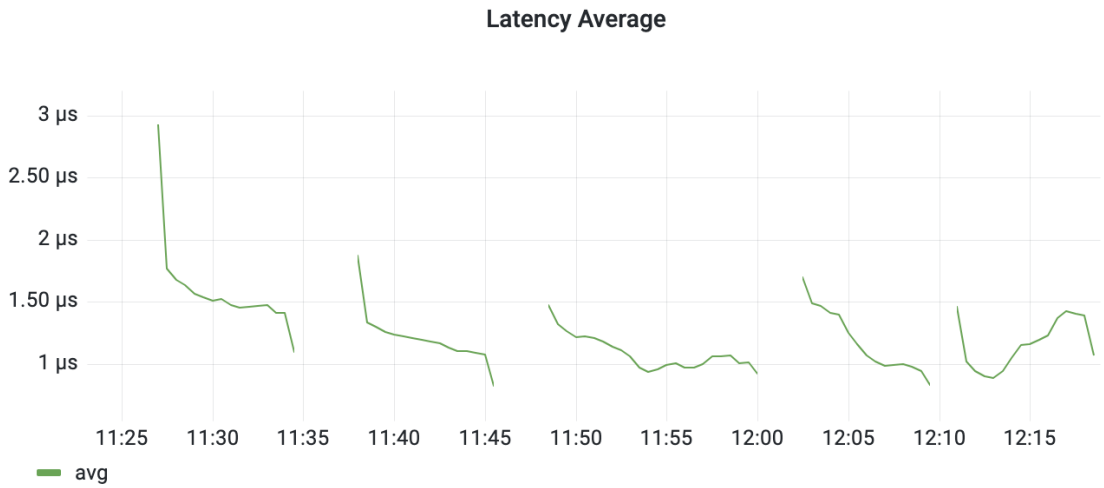

Figura 8: Tempo de Processamento do Servidor *Client Side* - *Servidor*

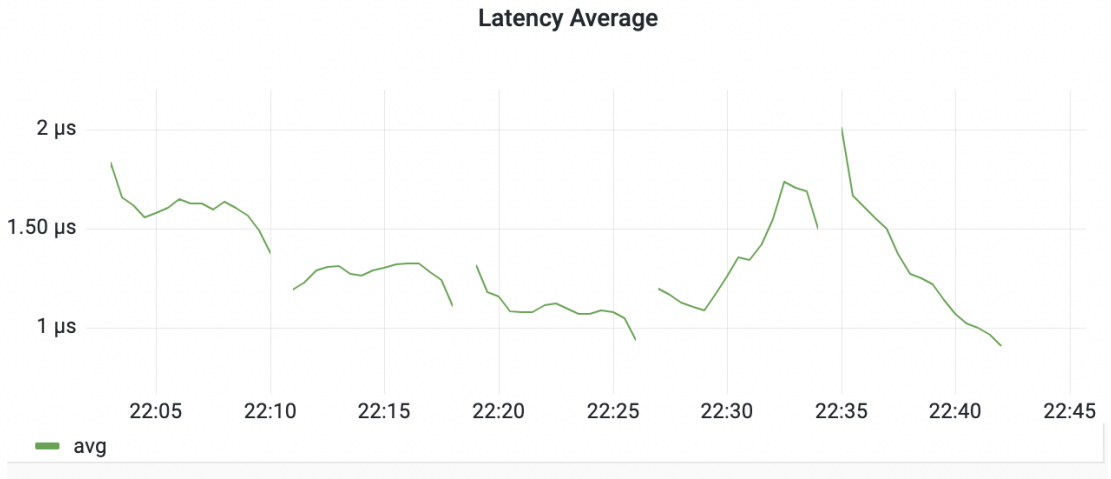

Figura 9: Tempo de Processamento do Servidor *Proxy* - *Servidor*

Observa-se o mesmo na Figura 10 e Figura 11 que representam o tempo total de resposta do processador desde a chegada da requisição no servidor *Web* até o retorno da resposta para a rede.

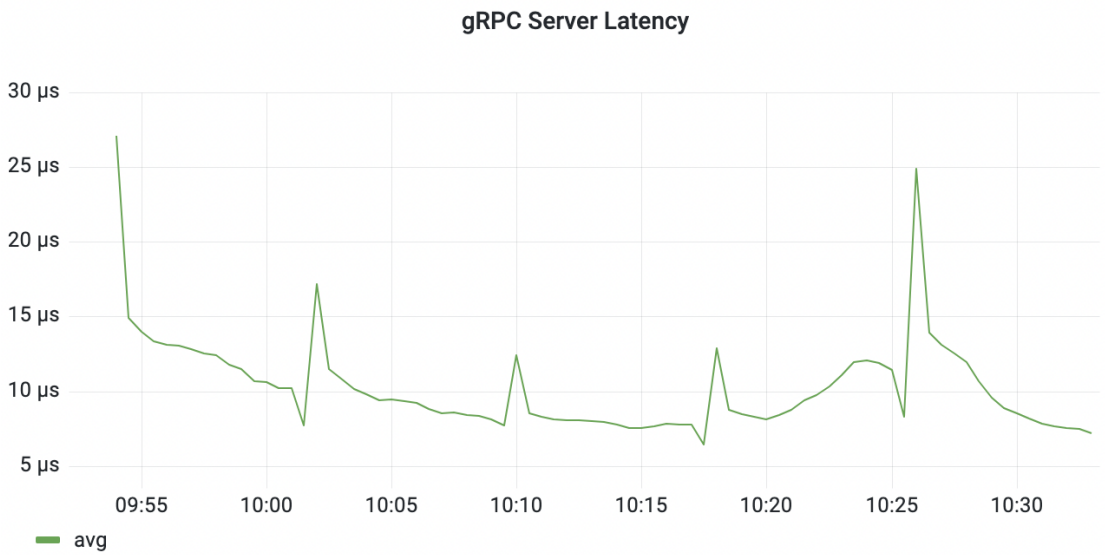

Figura 10: Latência Servidor gRPC *Client Side* - *Servidor*

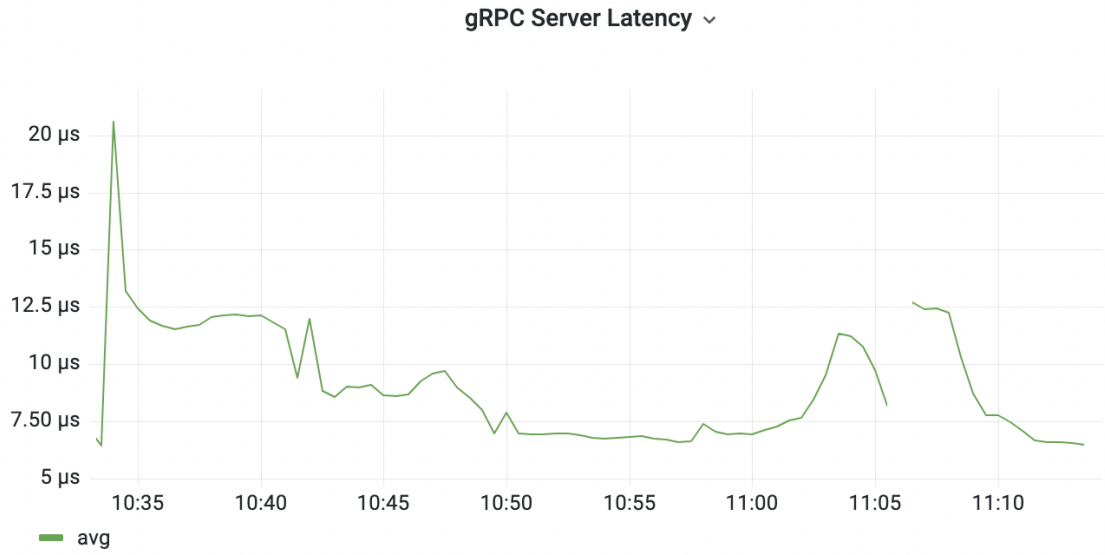

Figura 11: Latência Servidor gRPC *Proxy* - *Servidor*

Vale ainda observar o uso de CPU do servidor para verificar se há limitação de recursos e se as réplicas estão no mesmo nível de saturação.

Na Figura 12 e na Figura 13 o uso de CPU de cada réplica é acumulado, mostrando um somatório que nos picos se aproxima de 0,4 CPUs, com cada um chegando no máximo a 0,17 CPUs. Considerando que cada réplica tem 1 CPU completa para si, há recurso de sobra, além de estarem em valores extremamente próximos confirmando um balanceamento de carga muito bom.

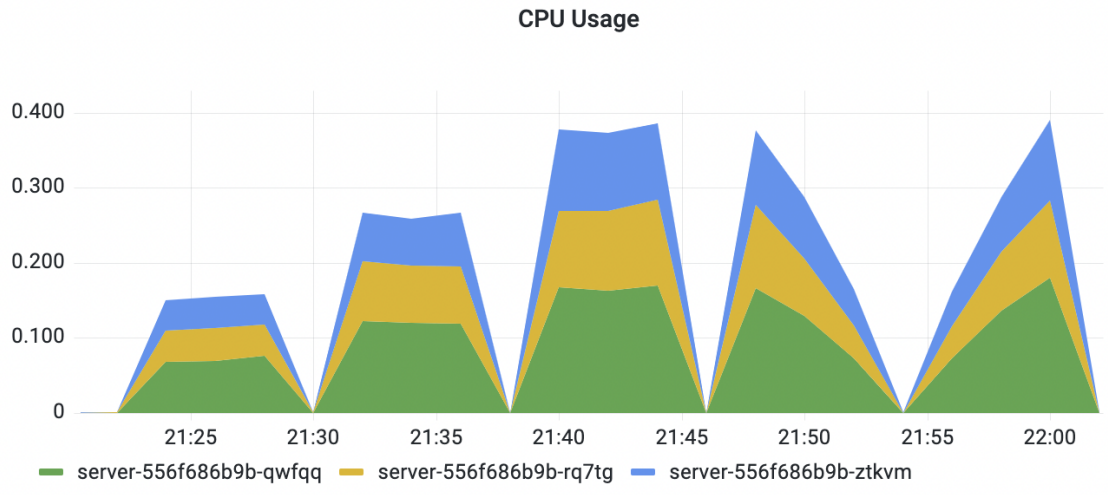

Figura 12: Uso de CPU *Client Side* - Servidor

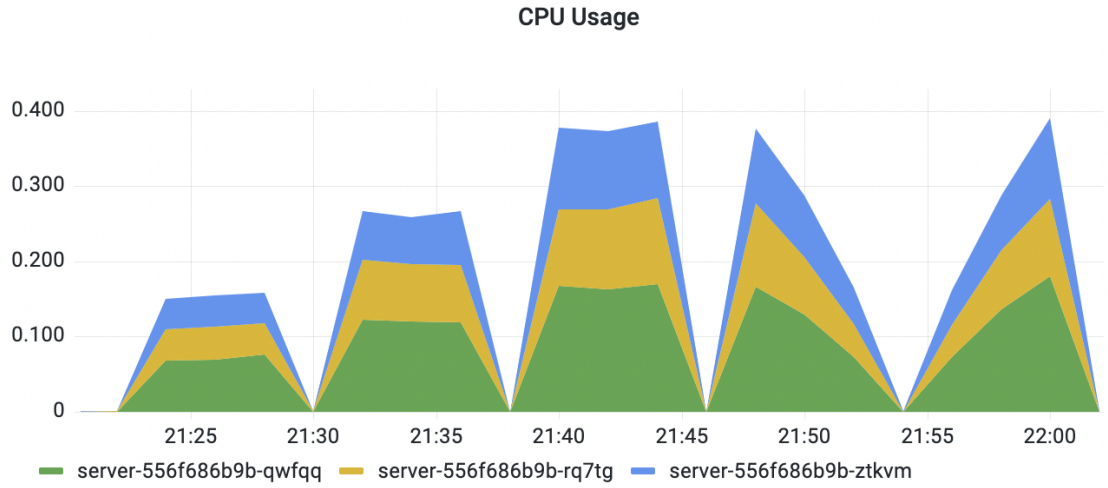

Figura 13: Uso de CPU *Proxy* - Servidor

3.4.3 **Análise das métricas do *Proxy***

A Figura 14 apresenta a latência medida pelo próprio *Proxy*. Nela que o comportamento observado pelo Cliente se mantém com leves aumentos de acordo com o aumento da taxa de requisições. Esse comportamento é mais acentuado nas duas últimas linhas do gráfico, que representam respectivamente os experimentos de carga crescente e decrescente.

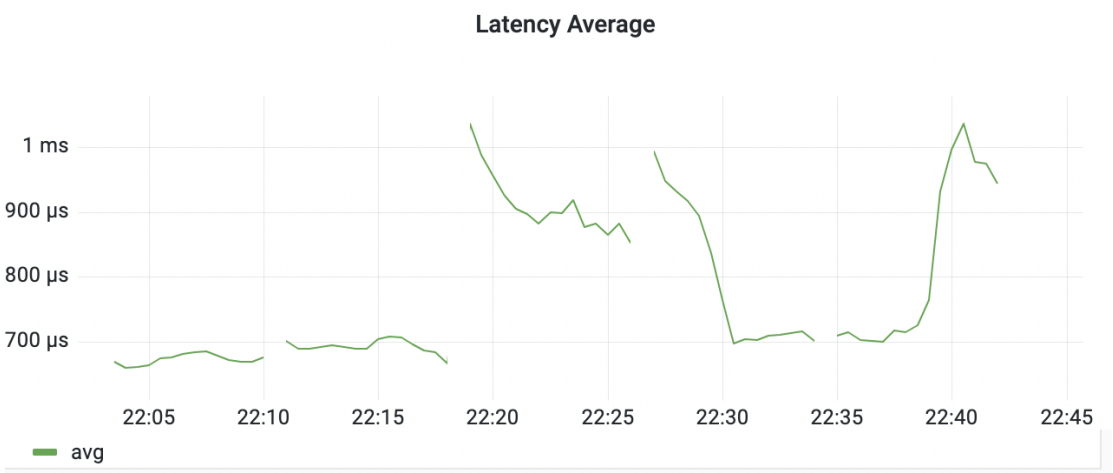

Figura 14: Média Latência - *Proxy*

Nesse caso, essa degradação pode ser explicada pela saturação de CPU do *Proxy*. Como pode ser observado na Figura 15, os aumentos de latência representados na Figura 14 se assemelham com o aumento no uso de CPU.

Nota-se ainda, ao observar mais criteriosamente o experimento de carga crescente (representado pela última curva da Figura 14, e a última área da Figura 15), que a degradação se acentua quando o proxy ultrapassa 80% de sua capacidade total de CPU, até pouco antes desse ponto a latência segue sem alterações consideráveis.

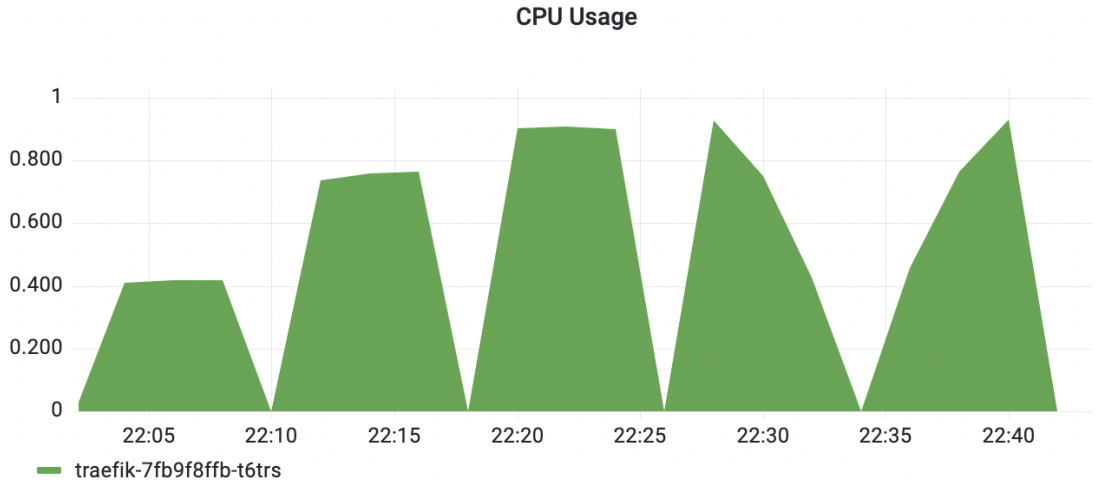

Figura 15: Uso de CPU - *Proxy*

3.5 **Considerações Finais**

Foi visto a partir dos resultados, que quando operando próximo de 70% de sua capacidade de CPU o *Proxy* adiciona em média cerca de 1 milissegundo de latência ao sistema, quando comparado ao balanceamento *Client Side*. Além disso há um consumo extra considerável de CPU e memória para manter o *Proxy* responsivo.

4 CONCLUSÃO E TRABALHOS FUTUROS

De modo geral, o gRPC tem sido utilizado no desenvolvimento de aplicações que estão sujeitas a cargas de trabalho normalmente muito elevadas, ou que possuem requisito de latência muito estrito, tornando necessário otimizar ao máximo, até mesmo no método de comunicação entre serviços.

Balanceadores de carga *Proxy* possuem funcionalidades que abstraem uma série de implementações de código dos clientes e dos servidores. No caso de uma implementação *Client Side*, o gRPC suporta diversas linguagens através de bibliotecas, as quais implementam boa parte das necessidades de um balanceador desse tipo. Porém, cada linguagem possui suas peculiaridades, e como o gRPC é um projeto de código aberto é possível encontrar comportamentos destoantes entre as implementações de cada linguagem. Por exemplo, na linguagem *Go* é feita uma atualização no registro de *Domain Name Service* (DNS) de tempos em tempos para atualizar o conjunto de conexões, já na implementação de Java essa operação só é realizada quando há falha em uma requisição. Apesar de serem pequenas estas diferenças, elas são pouco documentadas, e podem causar anomalias em um ambiente de produção, por exemplo se uma aplicação servidora (S) tem vários clientes (A e B) que geram carga de modo distinto um dos outros, pode ser que em determinado momento A dobre a quantidade de requisições geradas em S, fazendo com que ele tenha que escalar horizontalmente, enquanto B continua com a mesma taxa de requisições, se não houver nenhuma requisição falha, B não irá atualizar seu conjunto de conexões causando assim um desbalanceamento no sistema.

Este trabalho mostrou a diferença de latência entre as duas estratégias de balanceamento. Porém não é possível afirmar de maneira absoluta qual é a estratégia que produz o melhor resultado. De fato, cada uma das estratégias produz um melhor ou pior resultado, dependendo do contexto de uso.

Se para um determinado sistema 1 milissegundo em média não faz diferença e todos os outros pontos negativos listados nesse trabalho também são irrelevantes para o seu contexto, o *Proxy* será uma opção preferível devido à sua maior quantidade de abstrações e facilidade de implementação. Porém, caso essa latência adicional possa causar problemas ou interferir na responsividade do sistema, o *Client Side* apresenta-se como uma opção muito razoável para se trabalhar considerando que possui um excelente suporte das

bibliotecas do gRPC.

4.1 Limitações

Devido as limitações no *Proxy* e na ferramenta de geração de carga, não foi possível extrair estatísticas mais avançadas como os percentis, que ajudariam a entender o comportamento dos balanceadores nos casos mais extremos.

Além disso, o custo com a infraestrutura impossibilitou uma geração de carga mais robusta para testar o comportamento do *Proxy* com escalabilidade horizontal automática.

4.2 Trabalhos Futuros

Ainda há pouca informação de qualidade sobre esse tipo de ferramenta e muitas empresas gastam bastante tempo realizando suas próprias análises de desempenho, o que acaba atrasando o desenvolvimento da parte central que interessa ao seu negócio. Dessa forma, a análise de desempenho de ferramentas para sistemas distribuídos em ambientes de nuvem é um tema com grande potencial de pesquisa.

Os experimentos e análises realizados nesse trabalho poderiam continuar com muitas outras variações.

1. Adicionar aos fatores o tamanho do corpo da requisição, visto que o *Proxy* necessita realizar certo processamento que envolvem esses dados;

2. Comparar o desempenho do mesmo ambiente executando em provedores de nuvem diferentes. Durante a realização das execuções notou-se uma grande diferença de resultados entre a *Digital Ocean* e à *Google Cloud Platform* (GCP), sendo o segundo muito mais performática;

3. Realizar uma comparação entre outros *Proxies* de código aberto, como *Envoy* e *NGINX*;

4. Comparar o desempenho entre *Proxies* gerenciados como o *Amazon Web Services* (AWS) *Application Load Balancer* e os de código aberto citados a cima;

5. Analisar a latência do sistema executado nesse trabalho em um *cluster* multi-regiões e multi-nuvem;

6. Incluir na comparação *Service Meshes* como *Istio* e *Consul*.

REFERÊNCIAS

[1] *Apache Superset*. URL: https://superset.apache.org/.

[2] *cAdvisor*. URL: https://github.com/google/cadvisor.

[3] *GHZ*. URL: https://ghz.sh/.

[4] *Go*. URL: https://go.dev/.

[5] *Grafana*. URL: https://grafana.com/.

[6] *gRPC*. URL: https://grpc.io/.

[7] Raj Jain. *The Art Of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. Cambridge University Press, 1991.

[8] Marx Kanovich. "On The Complexity of Verification of Time Sensitive Distributed Systems". Em: (2021). URL: https://books.google.com.br/books?hl=pt-BR&lr=&id=dvFPEAAAQBAJ&oi=fnd&pg=PA251&dq=complexity+of+distributed+systems&ots=yIjODIGZm6&sig=H8z8va0ggTytCeZXcl6948y30n0&redir_esc=y#v=onepage&q=complexity%5C%20of%5C%20distributed%5C%20systems&f=false.

[9] *Kubernetes*. URL: https://kubernetes.io/.

[10] Priti Kumari e Parmeet Kaur. "A survey of fault tolerance in cloud computing". Em: *Journal of King Saud University - Computer and Information Sciences* 33.10 (2021), pp. 1159–1176. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2018.09.021. URL: https://www.sciencedirect.com/science/article/pii/S1319157818306438.

[11] Alex Magalhaes et al. "REPO: A Microservices Elastic Management System for Cost Reduction in the Cloud". Em: *2018 IEEE Symposium on Computers and Communications (ISCC)*. 2018, pp. 00328–00333. DOI: 10.1109/ISCC.2018.8538453.

[12] Sambit Kumar Mishra, Bibhudatta Sahoo e Priti Paramita Parida. "Load balancing in cloud computing: A big picture". Em: *Journal of King Saud University - Computer and Information Sciences* 32.2 (2020), pp. 149–158. ISSN: 1319-1578.

DOI: https://doi.org/10.1016/j.jksuci.2018.01.003. URL: https://www.sciencedirect.com/science/article/pii/S1319157817303361.

[13] *Prometheus*. URL: https://prometheus.io/.

[14] Douglas Soares. "RPC-QUIC: Middleware baseado em RPC utilizando protocolo QUIC". Em: (2021).

[15] *Traefik*. URL: https://traefik.io/.

[16] Robert Underwood, Jason Anderson e Amy Apon. "Measuring Network Latency Variation Impacts to High Performance Computing Application Performance". Em: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 68–79. ISBN: 9781450350952. DOI: 10.1145/3184407.3184427. URL: https://doi.org/10.1145/3184407.3184427.

[17] Zhiheng Zhong e Rajkumar Buyya. "A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources". Em: *ACM Trans. Internet Technol.* 20.2 (abr. de 2020). ISSN: 1533-5399. DOI: 10.1145/3378447. URL: https://doi.org/10.1145/3378447.

AVALIADORES

Prof. Vinicius Cardoso Garcia

ASSINATURAS

Recife, ____ de _____ de _____

João Filipe da Matta Ribeiro Moura

(Aluno)

Nelson Souto Rosa

(Orientador)