Statera: Um balanceador de carga rápido e flexível para aplicações HTTP na nuvem

Matheus H. Freitas[1], **Vitor B. Souza**[1]

[1]Departamento de Informática – Universidade Federal de Viçosa (UFV)
Viçosa, MG – Brasil

{matheus.h.freitas,vitor.souza}@ufv.br

Resumo. *O balanceamento de carga é uma função essencial na computação em nuvem, e os balanceadores de carga de camada 4, notadamente, são amplamente utilizados para esse fim. Contudo, esse tipo de balanceador apresenta duas limitações típicas: 1) a impossibilidade de balanceamento baseado no conteúdo do tráfego; 2) generalização, não apresentando funções importantes para protocolos específicos. Neste trabalho nós apresentamos o Statera, um balanceador de carga flexível e específico para o protocolo HTTP, cujo objetivo é promover escalabilidade e sanar as limitações dos balanceadores de carga de camada 4. Ao longo do texto, nós expomos em detalhes a arquitetura do Statera e apresentamos resultados obtidos em diferentes cenários.*

1. Introdução

A explosão de popularidade da internet ao longo dos anos gerou um aumento massivo no número de acessos que os sistemas da internet recebem diariamente. Isso incentivou a criação e a migração dos mais variados tipos de sistemas para a nuvem, incorporando diversas características nessas aplicações, tais como a alta escalabilidade, elasticidade e o escalonamento dinâmico dos recursos [Qian et al. 2009].

Para que essas vantagens do ambiente em nuvem sejam plenamente atendidas, é fundamental que exista um balanceador de carga na operação dessas aplicações [Mishra et al. 2020]. O balanceador permite que o tráfego possa ser distribuído de forma eficiente, servidores possam ser adicionados e removidos sem interrupções, servidores com problemas sejam retirados de atividade de forma rápida, diferentes serviços sejam isolados em grupos de servidores, entre outras características importantes para o pleno aproveitamento da nuvem [Qian et al. 2009].

Balanceadores de carga de rede podem operar em diferentes camadas do modelo de referência OSI. Quanto mais elevada a camada, maior é a disponibilidade de dados que podem ser utilizados na decisão de roteamento. Os balanceadores de carga de camada 4 (camada de transporte) são amplamente utilizados e já foram discutidos em diferentes trabalhos [Patel et al. 2013]. Contudo, pouca atenção vem sendo dada para os balanceadores de carga de camada 7 (camada de aplicação).

Um balanceador de camada 7 pode trazer vantagens importantes para aplicações modernas. Assim, ao invés de prover o aumento de desempenho utilizando informações do cabeçalho TCP para processar uma solicitação, como acontece em balanceadores de camada 4, ao trabalhar na camada 7, o balanceador pode rotear o tráfego recebido com base em diversas informações contidas, por exemplo, no cabeçalho HTTP [Dymora et al. 2021].

Nesse trabalho nós apresentamos o Statera, um balanceador de carga em software para o protocolo HTTP. O Statera é um balanceador flexível, feito para a nuvem, que pode ser executado em hardwares comuns e em VMs. O objetivo principal da arquitetura proposta é definir o design de um sistema computacional cuja função primária é balancear carga de rede para o protocolo HTTP. Assim, o Statera é capaz de interceptar tráfego HTTP e de distribuí-lo da melhor maneira possível sobre um grupo de servidores, promovendo a escalabilidade, a disponibilidade e a redundância do sistema como um todo. Além disso, a partir da derivação das necessidades das aplicações, o balanceador incorpora outras funcionalidades e características que agregam um alto valor em sua utilização. São elas:

- Organização dos servidores sob o balanceador em grupos comuns, nos quais cada grupo tem parâmetros específicos de modo a permitir a coexistência de diferentes grupos de servidores fornecendo diferentes serviços.
- Checagens de saúde periódicas dos servidores, interrompendo o envio de requisições para servidores que se mostrarem sem saúde, causando disrupção mínima para o cliente.
- Possibilidade do roteamento baseado no conteúdo das requisições, através da definição de múltiplas regras de roteamento.
- Implementação de múltiplos algoritmos de distribuição de carga para o roteamento das requisições, atendendo a diferentes tipos de cargas de trabalho.

O restante do texto está organizado da seguinte forma. A seção 2 apresenta alguns trabalhos relacionados. A seção 3 apresenta a arquitetura do Statera. A seção 4 apresenta os experimentos realizados e discute os resultados obtidos. Finalmente, a seção 5 conclui o artigo e cita possíveis trabalhos futuros.

2. Trabalhos Relacionados

Diferentes trabalhos apresentam arquiteturas de balanceadores de carga de rede que podem ser empregados em ambientes de nuvem. Em sua maioria, os trabalhos se particularizam em balanceadores de carga de camada 4. Um exemplo é o Ananta [Patel et al. 2013], um balanceador de carga em software que, em semelhança com o Statera, pode ser executado em hardwares comuns. O balanceador é baseado em um sistema distribuído e, devido a isso, pode ser escalado horizontalmente de forma nativa, atingindo grandes capacidades de taxa de transferência. No entanto, o trabalho não apresenta a possibilidade do balanceamento na camada 7, renunciando à funcionalidades específicas, como o roteamento baseado no conteúdo do tráfego.

Yoda [Gandhi et al. 2016] é um balanceador de carga de camada 7, que também opera sobre o protocolo HTTP. O trabalho se particulariza em atingir a alta disponibilidade do balanceador de carga, apresentando um mecanismo para o compartilhamento de estado das sessões TCP entre instâncias do Yoda, para evitar a perda de sessão no caso de falha. Contudo, o trabalho não se aprofunda no desenvolvimento de outros aspectos do balanceamento de carga, não apresentando, por exemplo, um algoritmo próprio de classificação de regras.

Os provedores de infraestrutura em nuvem, em sua maioria, oferecem serviços gerenciados de balanceamento de carga para o protocolo HTTP. A Amazon Web Services (AWS) oferece o *Application Load Balancer* [Amazon Web Services 2022], a Google

Cloud Plataform (GCP) oferece o *Cloud Load Balancing* [Google Cloud Plataform 2022] e a Azure oferece o *Application Gateway* [Azure 2022]. Todos eles têm características em comum com o Statera, apresentando mecanismos para o roteamento baseado em conteúdo, checagem de saúde, múltiplos algoritmos de roteamento, etc. Contudo, são soluções de código fechado, com o objetivo de exploração comercial e, dessa forma, os detalhes de suas arquiteturas não são publicados.

3. Arquitetura

Analisando os objetivos propostos, é possível perceber que as requisições precisarão passar por certas etapas em seu curso através do balanceador. Dessa forma, o balanceador deverá ser capaz de: escutar pelas requisições HTTP, avaliar as regras que foram definidas previamente e distribuir essas requisições para os servidores de forma eficiente.

Essas três etapas (escuta, avaliação de regras e encaminhamento) apresentam, cada uma, um domínio bem definido de responsabilidades e funções. Faz sentido então, na definição da arquitetura, agrupar cada uma dessas etapas em componentes distintos, de modo que a interação destes aconteça de maneira desacoplada e bem definida.

Para satisfazer essas necessidades, o Statera é dividido em três grandes componentes, chamados **Listener**, **Evaluator** e **Router**. O Listener é o componente responsável por escutar pelas conexões e se comunicar com os clientes. O Evaluator é o responsável por avaliar as regras definidas e decidir o caminho de cada requisição, podendo esta ser recusada e retornada ao cliente por meio do Listener, ou passada para o Router. O Router, por sua vez, é o responsável por distribuir as requisições sobre os servidores, realizando o balanceamento de fato. Os três componentes são organizados como uma *pipeline*, de forma que a requisição deve passar por cada uma das etapas. Essa *pipeline*, a organização dos componentes e o caminho de retorno são ilustrados na Figura 1.

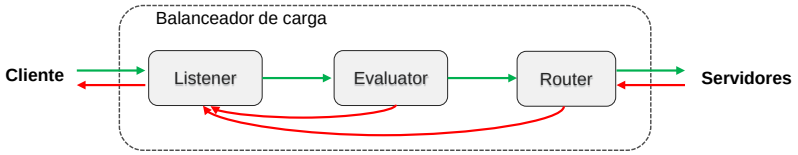


Figura 1. Organização dos componentes da arquitetura proposta.

3.1. Servidores e grupos de servidores

Em aplicações HTTP modernas, é comum que existam diferentes grupos de servidores, sendo cada um responsável por diferentes funções. Por exemplo, em uma mesma aplicação, pode haver um grupo de servidores responsável por servir arquivos estáticos e um outro grupo de servidores responsável por servir conteúdo dinâmico, como por exemplo, requisições de API.

Para refletir as diferentes partições lógicas observadas nesses grupos de servidores, a arquitetura proposta implementa de forma lógica o conceito de grupos. Cada grupo é composto por um conjunto de servidores, que geralmente, executam a mesma carga de trabalho.

O balanceamento de carga nesta arquitetura acontece a nível do grupo, ou seja, cada grupo pode utilizar um algoritmo de balanceamento diferente. Essa divisão do tra-

balho da escolha do servidor em múltiplos grupos, além do benefício supracitado, proporciona uma menor contenção no acesso concorrente às estruturas de dados utilizadas pelos algoritmos de balanceamento, e que precisam ser acessadas de forma sequencial.

3.2. Caminho de retorno

O caminho de retorno do tráfego balanceado de uma arquitetura que utiliza o Statera é através do próprio balanceador. Como, desde o princípio, a proposta é desenvolver um balanceador que irá operar na camada de aplicação, especificamente para o protocolo HTTP, esse é o único caminho de retorno possível, não sendo viável a utilização de DSR (Direct Server Return) [Bourke 2001].

Essa imposição traz consigo vantagens e desvantagens. A desvantagem mais considerável é o grande overhead gerado no balanceador no processamento do tráfego de resposta, já que é usual que o tráfego de resposta HTTP tenha múltiplas vezes o tamanho da requisição [Bourke 2001]. Uma solução que vem sendo utilizada pela indústria para contornar essa situação, sem perder os benefícios do balanceamento de camada 7, é utilizar uma arquitetura multi-camadas. Nessa arquitetura, utiliza-se balanceadores de camada 4 configurados com DSR para balancear o tráfego sobre balanceadores de camada 7, obtendo as características da atuação em ambas camadas [Naseer et al. 2020].

Uma vantagem importante do caminho de retorno que é adotado no projeto é o chamado SSL *Offloading*, que pode ser usado caso a rede entre o balanceador e os servidores seja privada, segura e confiável. Como o balanceador realiza a terminação TLS com o cliente (o balanceador é um ponto de criptografia/descriptografia da comunicação), a comunicação do balanceador até os servidores pode acontecer em forma de texto puro (não criptografado). Dessa maneira, o trabalho de criptografia que deveria ser realizado pelos servidores é transferido para o balanceador, fazendo com que os servidores possam dedicar mais recursos à sua função principal.

3.3. O componente *Listener*

A função do componente Listener é gerenciar as conexões dos clientes ao balanceador. O componente anuncia o balanceador no endereço de rede da máquina e realiza todo o trabalho necessário de gerência das conexões TCP estabelecidas entre os clientes e o balanceador.

Para atuar na camada de aplicação, o componente implementa um servidor HTTP de alto desempenho. Essa implementação segue todas as especificações das RFCs que definem o protocolo, garantindo sua máxima compatibilidade com os diferentes clientes. O servidor fornece o suporte a múltiplas versões do HTTP (versões 1.0, 1.1 e 2) e do TLS (versões 1.0 a 1.3), de forma que essas versões podem ser configuradas pelo administrador do balanceador.

O Listener dá suporte à escuta de conexões em múltiplas interfaces e portas de forma simultânea, inclusive em endereços IPv4 e IPv6. Por exemplo, é comum que servidores HTTP que atendem requisições da internet escutarem simultaneamente nas portas 80 e 443, as portas padrão do HTTP e HTTPS, respectivamente. Dessa maneira, o suporte à escuta simultânea em múltiplas portas traz flexibilidade para as diferentes possibilidades de uso do balanceador.

Em conexões HTTPS, o componente é o responsável por realizar o trabalho de criptografia da comunicação com cliente. O Listener permite a configuração de múltiplos certificados. Como, em geral, um mesmo balanceador encapsula aplicações distintas, é corriqueiro que essas aplicações tenham certificados distintos, fazendo-se necessário essa funcionalidade.

3.4. O componente *Evaluator*

O componente Evaluator é o responsável por decidir o caminho que as requisições irão seguir, realizando esse trabalho a partir da avaliação das regras de roteamento definidas previamente. Para cada requisição que chega ao balanceador, o componente realiza uma avaliação. Com base na regra que for considerada satisfeita, o Evaluator toma uma das três decisões possíveis: 1) encaminhar a requisição para um grupo de servidores; 2) rejeitar a requisição; 3) redirecionar o cliente.

Como apresentado na Figura 1, o Evaluator é o elo entre o Listener e o Router e, devido a isso, tem como uma de suas principais funções a desambiguação do caminho que a requisição deve seguir. Isso se deve ao fato de que a requisição pode chegar por meio de múltiplos endereços/portas e existem múltiplos grupos de servidores, ou seja, dado o caminho pelo qual a requisição chegou, é necessário uma forma de decidir para qual grupo de servidores ela será encaminhada.

Contudo, o endereço/porta pelo qual a requisição chegou não é a única informação que pode ser utilizada para decidir para qual grupo encaminhá-la. Como o Statera opera na camada de aplicação, todo o conteúdo da requisição está disponível ao balanceador e pode ser usado para enriquecer as avaliações de regras. Dessa forma, as regras podem dispor de partes selecionadas do conteúdo da requisição.

Para atender às demandas típicas de balanceadores de carga de aplicações em nuvem, a regra do Statera tem uma estrutura bem definida que atende a diferentes casos de uso. A estrutura completa da regra está descrita em detalhes na Tabela 1.

Cada regra tem uma prioridade (*Priority*), que é um inteiro positivo definido no momento da criação da regra. A cada requisição recebida, o Evaluator avalia cada uma das regras, em ordem crescente de prioridade. Assim que uma regra é satisfeita, a avaliação é considerada terminada e a ação (*Action*) constante nessa regra é tomada. Uma regra é considerada satisfeita quando todas as suas condições (*Conditions*) são satisfeitas. É importante destacar que cada regra precisa estar associada a um endereço/porta.

Na Tabela 2 é exibido, como exemplo, um conjunto de regras. O conjunto é composto por quatro regras que refletem, de forma simplificada, um caso comum de aplicações HTTP em nuvem. A primeira regra possui menor valor para o campo prioridade e, com isso, deve ser avaliada primeiro. Ela está associada à escuta na porta 80 do balanceador. Como essa regra tem um conjunto vazio de condições, ao ser avaliada ela é sempre satisfeita e, além disso, tem como ação o redirecionamento do cliente para um endereço do balanceador que utiliza HTTPS. A segunda e terceira regra são análogas entre si. Essas duas regras estão associadas à escuta na porta 443 e têm uma única condição: que o caminho da URL da requisição comece com **/api** ou **/static**, com a requisição sendo encaminhada para os grupos **api-servers** e **static-servers**, respectivamente. Por fim, a última regra e a com maior valor para o campo prioridade, é sempre satisfeita quando avaliada e encaminha a requisição para o grupo **default-servers**.

Tabela 1. Estrutura da regra

Priority	Um inteiro positivo que define a ordem de avaliação das regras.	
Listener	Um identificador que define o endereço/porta ao qual essa regra está associada.	
Conditions	Um conjunto de condições que precisam ser satisfeitas para que a regra seja considerada satisfeita. Cada condição possui a seguinte estrutura:	
	Not	Um booleano que caso tenha o valor verdadeiro nega a condição.
	Type	Um identificador que define qual será o tipo de avaliação realizada na condição. Isso define, por exemplo, qual campo/chave/informação será avaliada. Os tipos possíveis são:
		Path: Realiza a comparação no Path da URL da requisição. Esse tipo não admite chave.
		Query: Realiza a comparação na Query da URL da requisição. Esse tipo admite chave.
		Body (String): Realiza a comparação no corpo da requisição, tratando-o como uma string (texto plano). Esse tipo não admite chave.
		Body (X-www-form-urlencoded): Realiza a comparação no corpo da requisição, tratando-o como um conjunto de chaves-valores, como os enviados em formulários da WWW. Esse tipo admite chave.
		Header: Realiza a comparação no cabeçalho HTTP da requisição. Esse tipo admite chave.
		IP: Realiza a comparação sobre o IP do cliente que enviou a requisição, verificando se o IP se encontra dentro de um intervalo. Esse tipo não admite chave.
	Key	Uma string que define qual a chave que será avaliada, se o tipo escolhido acima permitir chaves. Caso o tipo escolhido não suporte chaves, esse campo deve ser deixado em branco.
	Operation	Um identificador que define qual a operação de avaliação que deverá ser executada. Pode ser: 1) igualdade 2) começa com 3) regex 4) range (apenas pode ser usado em IPs).
	Value	Uma string que define o valor que será usado na avaliação.
Action	Indica a ação que deverá ser tomada caso essa regra seja satisfeita. Esse campo possui a seguinte estrutura:	
	NodeGroup	Caso escolhido, indica que a requisição deve ser encaminhada para este grupo de servidores.
	Reject	Caso escolhido, indica que essa requisição deve ser negada. É composto por um par descrevendo o status HTTP e a mensagem que deve ser exibida ao cliente pelo balanceador.
	Redirect	Caso escolhido, indica que o cliente deve ser redirecionado o endereço definido neste campo.

3.5. O componente *Router*

O Router é o componente responsável pelo roteamento das requisições HTTP para os servidores. Ele é o encarregado pela comunicação com os servidores, se conectando a cada um e encaminhando as requisições de acordo com a decisão tomada pelo Evaluator. Além disso, o Router realiza o balanceamento das requisições entre os servidores de um

Tabela 2. Exemplo de um conjunto de regras

Priority	1	2	3	4
Listener	0.0.0.0:80	0.0.0.0:443	0.0.0.0:443	0.0.0.0:443
Conditions	*Vazio*	**Not** False **Type** Path **Key** - **Operation** Começa com **Value** /api	**Not** False **Type** Path **Key** - **Operation** Começa com **Value** /static	*Vazio*
Action	**Redirect** | https://example.com	**NodeGroup** | api-servers	**NodeGroup** | static-servers	**NodeGroup** | default-servers

mesmo grupo, utilizando o algoritmo de balanceamento escolhido para o grupo.

3.5.1. Algoritmos de balanceamento

O Statera implementa três algoritmos de balanceamento [Lee and Jeng 2011]: Round-Robin, Weighted Round-Robin e Least-Requests. O **Round-Robin (RR)** é um algoritmo amplamente conhecido por sua simplicidade. No caso do Statera, as requisições são distribuídas de forma circular sobre cada servidor do grupo, de modo que cada um recebe a mesma quantidade proporcional de requisições. A vantagem no uso desse algoritmo é a sua simplicidade de execução e de implementação. A sua desvantagem é que não leva em conta a especificidade de cada tipo de trabalho e nem a capacidade dos servidores, podendo levar, muitas vezes, a sobrecarga de alguns nós.

O **Weighted Round-Robin (WRR)** possui um mecanismo de funcionamento muito semelhante ao RR, com a diferença de que permite que sejam definidos pesos para cada um dos servidores. Dessa maneira, o algoritmo distribui as requisições de forma que cada servidor receba uma quantidade proporcional ao seu peso. A vantagem dessa opção é que caso se saiba a capacidade de cada um dos servidores, é possível distribuir a carga de acordo. A desvantagem vem do fato de que definir esses pesos previamente pode ser um grande desafio.

O **Least-Requests (LR)** é uma adaptação do algoritmo Least-Connections [Chen and Chen 2004]. Ao se utilizar LR, o Statera monitora constantemente o número de requisições HTTP em andamento para cada um dos servidores e envia a próxima requisição para o servidor com o menor número de requisições em andamento. A vantagem desse algoritmo é que ele ajusta dinamicamente o tráfego para corrigir disparidades de capacidades de servidores e de cargas de trabalhos. A desvantagem se refere a maior demanda de trabalho sobre o balanceador ao ter que monitorar e contar cada uma das requisições.

Esses algoritmos foram escolhidos devido ao seu longo histórico de uso e bons resultados práticos [Lee and Jeng 2011, Mesbahi and Rahmani 2016]. Além disso, foi levado em conta o fácil entendimento do funcionamento e os claros cenários ideais de aplicação de cada um, diminuindo o esforço cognitivo necessário no momento da escolha.

3.5.2. Checagem de saúde

O Router também realiza a checagem de saúde dos servidores. Essa checagem é importante para que apenas servidores saudáveis recebam requisições. A checagem acontece

de maneira periódica no decorrer do tempo, sendo realizada através de uma requisição HTTP que é enviada a cada servidor. Caso a requisição seja completada dentro do tempo máximo definido e retorne o código de status 200 (Ok), o servidor é considerado saudável. Caso não seja possível completar a requisição dentro do tempo máximo definido ou o servidor retorne um código de erro, o servidor é considerado não saudável.

O administrador do Statera pode definir três parâmetros, a nível de grupo, relacionados à verificação de saúde. O **Path** define o caminho da URL da requisição. O **Timeout** define o prazo de resposta em segundos, que, se excedido, indica que o servidor está inoperante. O **Interval** define o intervalo em segundos entre cada uma das requisições de checagem. Quanto menor esse valor, mais rápido uma disrupção é detectada. Por outro lado, valores muito pequenos combinados com um alto número de servidores resultam em uma maior carga de trabalho para o balanceador e aumento de carga na rede.

3.6. Escalabilidade e redundância do balanceador

Apesar do Statera promover maior escalabilidade dos servidores, ele ainda não apresenta mecanismos que permitam que ele próprio seja escalado horizontalmente de forma nativa. Como medida paliativa, uma possibilidade é a distribuição dos servidores sobre cada instância do balanceador. Assim, em um grupo com 500 servidores e duas instâncias do Statera, cada instância poderia ficar responsável por 250 servidores, por exemplo. Além disso, por se tratar de um balanceador de carga HTTP, uma outra possibilidade para escalar o balanceador horizontalmente seria aproveitar o fato de as requisições HTTP não terem estado e distribuir a carga sobre múltiplos balanceadores através, por exemplo, da técnica conhecida como DNS Round-Robin.

Por outro lado, por ser um balanceador de camada 7, o Statera realiza a terminação das conexões TCP. Num caso de eventual falha de uma instância do balanceador, o estado seria perdido e as conexões estabelecidas precisariam ser reiniciadas. Contudo, esse problema é amenizado pelo caráter das requisições HTTP. A operação básica do protocolo (requisição-resposta) possui um tempo de vida relativamente curto. Assim, no caso da disrupção de uma conexão TCP, a requisição seguinte iniciará uma nova conexão [Nielsen et al. 1999].

Como o projeto atual não apresenta mecanismos que controlem o roteamento antes do tráfego chegar a uma instância do balanceador, em um caso de falha, o administrador fica responsável por implementar mecanismos que monitorem e direcionem as novas conexões a uma instância saudável. O desenvolvimento de estratégias para permitir a escalabilidade e redundância no Statera de forma nativa ficam como trabalhos futuros.

4. Experimentos e resultados

Com o objetivo de comprovar a efetividade da arquitetura proposta, o Statera foi implementado utilizando a linguagem de programação **Go**, que é uma linguagem de código aberto, compilada, sintaticamente semelhante ao C, com suporte nativo à concorrência, e que tem como propósito, desde sua concepção, a programação de sistemas em rede para máquinas com múltiplos núcleos[1]. Além disso, foi planejado e executado um experimento para simular um ambiente de trabalho do balanceador similar ao real. O objetivo principal foi verificar a efetiva capacidade do Statera de receber uma quantidade significativa

[1]Implementação do Statera disponível no repositório: `https://github.com/mhef/statera`

de tráfego dos clientes e de distribuir esse tráfego sobre múltiplos servidores maximizando sua utilização. Em outro experimento, foi comparada a capacidade dos algoritmos de balanceamento implementados. Nesta seção, esses experimentos e seus resultados são detalhados.

4.1. Agentes

Para simular os clientes foi utilizada a ferramenta geradora de carga **k6**. Essa é uma ferramenta de código aberto, implementada em Go, especializada em testes de carga para o protocolo HTTP [k6 2022]. A principal funcionalidade que difere essa ferramenta de outras tradicionais (como o JMeter, por exemplo) é a definição total do teste por meio da linguagem de programação *JavaScript*, o que traz maior flexibilidade e facilidade para automação. A ferramenta introduz o conceito de *Virtual Users* (VUs) que executam paralelamente o script do teste.

Para simular os servidores, foi implementado um servidor HTTP próprio que realiza dois tipos de operações. O primeiro tipo tem o objetivo de simular operações *I/O bound*, nas quais a implementação faz com que a corrotina responsável durma por um tempo aleatório que varia de 45ms a 200ms antes de transmitir a resposta, simulando uma espera por *I/O*. Esse intervalo de tempo foi escolhido devido a sua similaridade com o tempo médio de resposta de múltiplas operações em disco. O segundo tipo de operação tem o objetivo de simular operações *CPU bound*, nas quais a implementação realiza um teste de primalidade (Baillie–PSW) de um número primo de 512 bits enviado pelo cliente. Esse tamanho foi escolhido a partir de experimentos que mostraram um balanço adequado entre a utilização dos recursos do servidor e a quantidade de requisições.

4.2. Ambiente de execução

A execução do teste foi realizada em VMs na infraestrutura de nuvem do provedor Amazon Web Services (AWS). Foram utilizadas instâncias EC2 e *containers* executados no motor Fargate, um ambiente de execução de *containers* sem servidor [Amazon Web Services 2022].

A distribuição dos recursos computacionais e dos agentes foi feita da seguinte maneira: **k6** implantado em *1 x* instância [*48 vCPU / 96 GiB RAM*]; **Statera** implantado em *1 x* instância [*32 vCPU / 64 GiB RAM*]; **Servidor** implantado em *65 x container* [*2 vCPU / 4 GiB RAM*]. Para evitar interferências causadas pelo roteamento na internet pública, todo o teste foi realizado na rede privada do provedor.

4.3. Configurações dos agentes

O script do VU foi implementado de forma a aleatorizar a ordem de envio dos dois tipos de requisição, simulando o comportamento do tráfego real. Além disso, a quantidade de requisições *CPU bound* e *I/O bound* foi aproximada para 20% e 80% do tráfego total, respectivamente. Essa razão foi escolhida a partir de experimentos que mostraram um balanço adequado entre a utilização dos recursos do servidor e a quantidade de requisições por segundo. O comportamento do VU está descrito no Algoritmo 1.

Algoritmo 1: Script do VU (Virtual User).

```
while true do
    if random() < 0.8 then
        doIORequest();
    else
        doCPURequest();
    end
    sleep(1); // sleep 1 sec
end
```

Para simular diferentes cargas de trabalho, o teste foi segmentado em 3 fases: uma fase inicial de aceleração gradual (*ramp up*), uma fase de sustentação e uma última fase de desaceleração gradual (*ramp down*). O tempo de duração e a quantidade alvo de VUs de cada fase estão descritos na Tabela 3. Essa quantidade alvo de VUs foi definida através de experimentos prévios que demonstraram que essa quantidade é suficiente para atingir o intervalo de 70%-80% de utilização máxima de CPU no *cluster* de servidores.

Tabela 3. Fases do teste de carga do experimento

Fase	1	2	3
Duração	15 min	10 min	15 min
Quantidade alvo de VUs	de 0 a 10.000	10.000	de 10.000 a 0

O balanceador foi configurado com apenas uma regra de roteamento, a qual encaminha todas requisições para um único grupo englobando todos os 65 servidores.

4.4. Experimentos

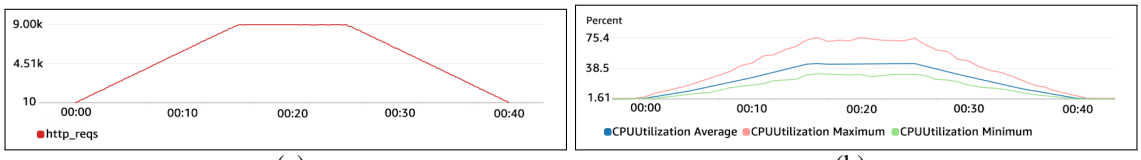
Dois experimentos foram realizados com o objetivo de avaliar a capacidade do Statera de comportar o grande número de requisições realizadas pelos VUs, bem como a eficácia dos algoritmos de balanceamento. Ambos experimentos utilizaram os mesmos parâmetros, sendo variado apenas o algoritmo de balanceamento. Assim, no **experimento 1**, foi adotado o algoritmo Round-Robin, enquanto que, no **experimento 2**, foi adotado o algoritmo Least-Requests. O algoritmo Weighted Round-Robin não foi contemplado nos experimentos devido a sua semelhança com o Round-Robin.

4.5. Resultados experimentais

Durante a execução dos experimentos foram observadas duas principais métricas para verificar a eficácia do balanceador. São elas: 1) quantidade de requisições por segundo; 2) utilização média de CPU pelo *cluster* de servidores. A hipótese era de que a utilização média de CPU pelo *cluster* seria proporcional a quantidade de requisições por segundo. Isso demonstraria a capacidade do balanceador de distribuir a carga de forma efetiva.

No primeiro experimento, onde o balanceador foi configurado para utilizar o algoritmo Round-Robin, foram executadas 13 milhões de requisições HTTP, com uma média de 5633 RPS (requisições por segundo), sendo que 90% das requisições tiveram um tempo de resposta menor que 186.39ms. A quantidade de RPS e a utilização de CPU média dos servidores durante o decorrer do experimento estão expressos nos gráficos da Figura 2.

É possível notar por meio do gráfico (a) da Figura 2 que a quantidade de RPS não apresenta flutuações bruscas no transcorrer do teste, demonstrando a plena capacidade do



(a) (b)

Figura 2. (a) Quantidade de RPS processadas pelo balanceador (b) Utilização de CPU em porcentagem pelo *cluster* de servidores.

balanceador de absorver a carga. Além disso, durante todo o teste, o pico de utilização de CPU pelo balanceador foi de apenas 6%, sendo o gargalo ocasionado pela utilização de CPU do *cluster* de servidores. Isso mostra a capacidade do Statera de atender a um conjunto de servidores ainda maior.

Através da análise do gráfico (b) da Figura 2 de utilização de CPU pelo servidores, é possível notar que houve uma distribuição satisfatória de carga, com a utilização de CPU sendo proporcional à quantidade de RPS. A utilização *média* teve o pico em aproximadamente 44%, enquanto a utilização *máxima* observada teve o pico em aproximadamente 75%. Essa disparidade entre a utilização média e máxima pode ser explicada pelo algoritmo utilizado nesse experimento. Como o Round-Robin não leva em conta a carga de cada servidor, acontece uma distribuição menos efetiva. No experimento 2, foi avaliado o algoritmo Least-Requests. A utilização de CPU pelos servidores nesse cenário está descrita na Figura 3.

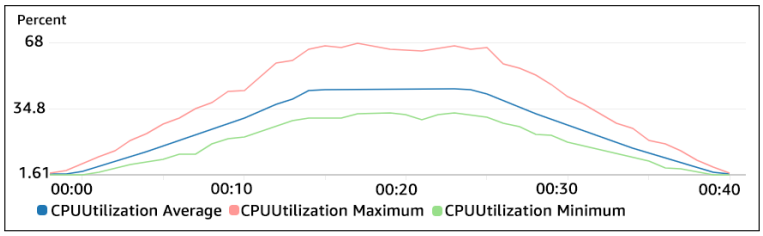


Figura 3. Utilização de CPU em porcentagem pelo *cluster* de servidores no cenário 2.

É possível notar através do gráfico da Figura 3, que a utilização média de CPU se manteve muito semelhante ao cenário anterior, mantendo-se na fase de sustentação, próxima a 44%. Já a utilização máxima de CPU apresentou, na fase de sustentação, uma queda de aproximadamente 7%. Isso pode ser explicado pela avaliação em tempo-real feita pelo algoritmo, que encaminha as requisições para os servidores com menor carga no instante.

5. Conclusão

Esse artigo apresentou o Statera, um balanceador de carga de camada 7 rápido e flexível para aplicações HTTP na nuvem. Por se especializar no protocolo HTTP, o Statera é capaz de utilizar o conteúdo do tráfego para tomar as decisões de roteamento. Para isso, a arquitetura proposta permite a criação de regras baseadas em diversos campos constantes nas requisições recebidas pelo balanceador. Os resultados preliminares, obtidos através de experimentos, mostraram que o Statera permite que o tráfego HTTP seja distribuído

de forma eficiente sobre múltiplos servidores, maximizando a utilização dos recursos disponíveis. Como trabalho futuro, planejamos: 1) estender o componente Evaluator para permitir a criação de regras baseadas em scripts, aumentando seu poder e flexibilidade; 2) introduzir mecanismos que possibilitem distribuir o Statera de forma nativa, promovendo a escalabilidade e redundância do balanceador.

Referências

Amazon Web Services (2022). Compute on aws. [Online. Acesso em 7 jul. 2022].

Azure (2022). Application gateway. [Online. Acesso em 7 jul. 2022].

Bourke, T. (2001). *Server Load Balancing*. Help for network administrators. O'Reilly Media, Incorporated.

Chen, T.-S. and Chen, K.-L. (2004). Balancing workload based on content types for scalable web server clusters. In *18th International Conference on Advanced Information Networking and Applications, 2004. AINA 2004.*, volume 2, pages 321–325. IEEE.

Dymora, P., Mazurek, M., and Sudek, B. (2021). Comparative analysis of selected open-source solutions for traffic balancing in server infrastructures providing www service. *Energies*, 14(22):7719.

Gandhi, R., Hu, Y. C., and Zhang, M. (2016). Yoda: A highly available layer-7 load balancer. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16.

Google Cloud Plataform (2022). Cloud load balancer. [Online. Acesso em 7 jul. 2022].

k6 (2022). k6 documentation. [Online. Acesso em 11 jul. 2022].

Lee, R. and Jeng, B. (2011). Load-balancing tactics in cloud. In *2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 447–454. IEEE.

Mesbahi, M. and Rahmani, A. M. (2016). Load balancing in cloud computing: a state of the art survey. *Int. J. Mod. Educ. Comput. Sci*, 8(3):64.

Mishra, S. K., Sahoo, B., and Parida, P. P. (2020). Load balancing in cloud computing: a big picture. *Journal of King Saud University-Computer and Information Sciences*, 32(2):149–158.

Naseer, U., Niccolini, L., Pant, U., Frindell, A., Dasineni, R., and Benson, T. A. (2020). Zero downtime release: Disruption-free load balancing of a multi-billion user website. In *ACM SIG on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 529–541.

Nielsen, H., Mogul, J., Masinter, L. M., Fielding, R. T., Gettys, J., Leach, P. J., and Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1. RFC 2616.

Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D. A., Kern, R., Kumar, H., Zikos, M., Wu, H., et al. (2013). Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218.

Qian, L., Luo, Z., Du, Y., and Guo, L. (2009). Cloud computing: An overview. In *IEEE international conference on cloud computing*, pages 626–631. Springer.