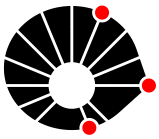

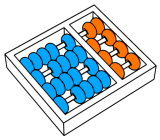

Algoritmos para Balanceamento de Carga no NGNIX

L. K. G. Tamanaha *G. L. Domingues* *L. F. Bittencourt*

Relatório Técnico - IC-PFG-21-49
Projeto Final de Graduação
2021 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Algoritmos para Balanceamento de Carga no NGINX

Lucas Koiti Geminiani Tamanaha Guilherme Luis Domingues
Luiz Fernando Bittencourt*

Resumo

Este trabalho tem por objetivo introduzir e avaliar a utilização da solução *NGINX*, em sua versão gratuita, como *load balancer*. O foco, portanto, será em trazer os conceitos relacionados à distribuição de carga em requisições HTTP com o intuito de analisar a performance da solução em cada algoritmo disponível em sua configuração: *Round Robin*, *Weighted Round Robin*, *IP Hash*, *Least Connection* e *Weighted Least Connection*.

1 Introdução

Cada dia mais há um alto crescimento no tráfego da internet. Com esse aumento no volume de requisições diferentes problemas e, consequentemente, soluções surgem. Nesse contexto, não apenas o hardware é desafiado a evoluir para atender e manter a qualidade dos sistemas, mas os softwares de servidores web também buscam uma maior flexibilidade e melhoria de performance constantemente.

Um dos principais pontos que podem afetar o desempenho de um servidor web é a quantidade de requisições que ele recebe em um espaço de tempo. Quanto maior a quantidade, mais processamento a máquina precisar realizar e responder para os clientes que estão acessando. Uma maneira de contornar esse problema é fazer o aumento vertical das máquinas, isto é, aumentar as especificações do hardware (mais memória RAM, um processador mais rápido, etc). Outra abordagem é fazer o aumento horizontal do servidor, isto é, aumentar o número de máquinas com as mesmas especificações e fazer com que, a partir de um único ponto de entrada, as requisições sejam distribuídas entre elas.

Assim, surgiu em 2004, o NGINX [1], uma ferramenta com inúmeras aplicações, tais como servidor web, servidor de cache, API gateway e, a utilizada neste trabalho, a capacidade de ser configurado como *load balancer* das requisições em um conjunto de servidores. Hoje o NGINX é utilizado por cerca de um terço do mercado de tecnologia, estando presente em empresas de grande porte como Adobe, Globo.com e BuzzFeed [6].

Contudo, o intuito deste trabalho é de analisar sua versão *open source* que contém disponível os seguintes algoritmos para distribuição de carga, os estáticos: *Round Robin*, *Weighted Round Robin* e *IP Hash*, os dinâmicos: *Least Connection* e *Weighted Least Connection*.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP

Assim, iremos observar a capacidade da solução no balanceamento de requisições HTTP. Ou seja, o comportamento do NGINX ao realizar o *proxy* das interações com um cliente. O qual será um teste de carga utilizando a ferramenta Locust [2]. Para, portanto, identificar sua performance em cada algoritmo citado.

2 Conceitos

2.1 Web Servers

Um *web server* [3] pode ser referência ao computador ou ao software que está sendo utilizado para essa função. A qual, no caso, é a capacidade de processar requisições HTTP/HTTPS. No momento, estamos nos referindo ao termo como o software utilizado para intermediar a comunicação entre o cliente e servidor. O qual também está em uma máquina intermediária. A comunicação entre cliente e servidor é feita através do protocolo REST, e utilizamos também o conceito de uma aplicação de interface, ou API.

2.2 NGINX

Este é o software *opensource* que nós utilizaremos como *Web Server*. No entanto, o NGINX [1] possui diversos recursos mais avançados que simplesmente redirecionar as requisições HTTP/HTTPS do cliente e devolver o retorno esperado. O principal, no caso deste trabalho, é a sua capacidade de agir como *load balancer*.

Sua arquitetura é formada por um processo principal e diversos processos menores denominados de workers. A quantidade desses processos menores varia de acordo com a quantidade de núcleos do processador. Se é um quad-core, serão criados quatro workers. Além disso, existe também um processo dedicado para o gerenciamento da cache e outro para seu carregamento.

O ciclo de vida de uma requisição HTTP para o NGINX começa com a leitura dos headers da requisição. Logo após, é feita a identificação da configuração do bloco recebido, aplicado os limites de concorrência e também a autenticação, verificando o controle de acesso interno e externo. Durante esse processo, são realizados os redirecionamentos internos e subrequests. Por fim, a requisição é redirecionada para o serviço de load balancing, que escolhe o servidor com base no algoritmo, e encaminha a requisição para aquele escolhido. Uma vez processada e retornada, o NGINX então, devolve para o cliente a resposta obtida do servidor [7].

2.3 Balanceador de carga (load balancer)

O balanceador de carga é uma ferramenta responsável por administrar e controlar para qual servidor deve ir uma requisição. Ele é utilizado em aplicações que buscam uma maior confiança, já que é possível criar réplicas de todo o ambiente de um servidor. Dessa maneira, se uma máquina não estiver em seu pleno funcionamento, o cliente não deveria ser impactado, já que as demais estão funcionando corretamente. O load balancer distribui as requisições

para os servidores conectados à ele usando alguns algoritmos de balanceamento de carga, que são mostrados e definidos abaixo.

2.4 Algoritmos de balanceamento de carga

Os algoritmos de balanceamento de um load balancer podem ser de dois tipos: estáticos e dinâmicos.

2.4.1 Estáticos

Um algoritmo estático é definido por não utilizar informações referentes ao estado do sistema para tomar decisões de distribuição. Neste contexto, requisições HTTP serão distribuídas seguindo regras já estabelecidas que não verificam se os servidores de destino estão, por exemplo, sobrecarregados.

2.4.1.1 Round Robin

É um algoritmo estático bem simples. Sua decisão consiste em atribuir as requisições em sequência com base no grupo de servidores disponíveis. No caso do NGINX [1], esses servidores possuem seus endereços listados em ordem no arquivo de configuração. A Figura 1 ilustra uma configuração com 3 servidores. A primeira requisição é direcionada para o servidor 1, a subsequente para o 2 e a próxima para o servidor 3. Ao chegar no último servidor, volta-se para o início do grupo.

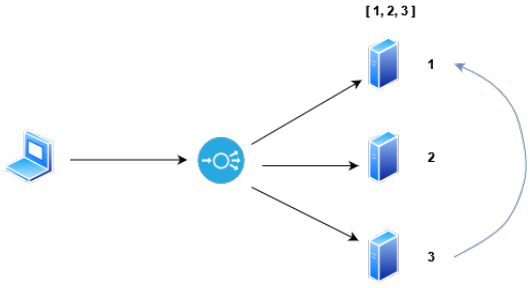

Figura 1 - Representação de roteamento utilizando o algoritmo Round Robin.

2.4.1.2 Weighted Round Robin

Weighted Round Robin é uma variação do Round Robin, porém os servidores recebem pesos para estabelecer uma proporção no número de requisições que recebem. Desta forma, em um ambiente com servidores não homogêneos, é possível direcionar mais carga para aqueles com maior capacidade computacional. A Figura 2 ilustra um cenário onde o servidor 1 recebe duas vezes mais requisições que os demais.

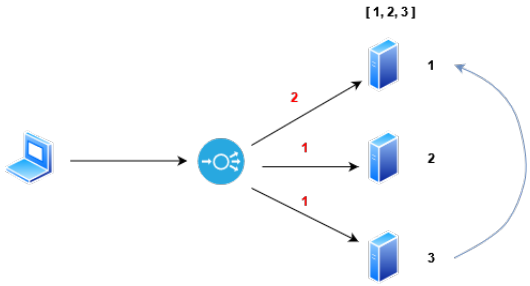

Figura 2- Representação do roteamento de Weighted Round Robin, na qual o servidor 1 possui peso 2.

2.4.1.3 IP Hash

O objetivo deste algoritmo é manter as requisições de um cliente direcionadas sempre para o mesmo servidor. Isso é feito através de um hash calculado com o endereço IP do cliente. Caso o servidor mapeado para um determinado IP se encontrar indisponível, a requisição é direcionada para o seguinte, seguindo a ordem na lista do grupo. A Figura 3 busca ilustrar como o balanceador de cargas direciona a requisição, na qual o cliente está mapeado para o servidor 1. No caso do NGINX [1], ele utiliza como chave hash os três primeiros *octets* do IPv4 ou o IPv6 todo que corresponde ao endereço do cliente.

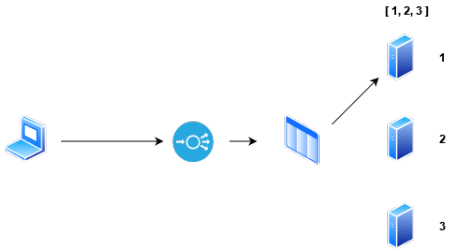

Figura 3 - Representação do algoritmo de IP hash, onde o load balancer direcionou o cliente para o primeiro server.

2.4.2 Dinâmicos

Os algoritmos dinâmicos, em oposição direta aos estáticos, consideram os estados das cargas no sistema para a tomada de decisão. A distribuição das requisições entre os servidores busca ser mais justa e evitando sobrecargas nos mesmos. Algumas aplicações que exigem maior processamento, podem sobrecarregar os servidores. Desta forma, o algoritmo busca direcionar as novas requisições para os servidores que estão menos sobrecarregados.

2.4.2.1 Least Connection

Sua abordagem dinâmica consiste em consultar as informações sobre o número de conexões ativas em cada servidor disponível. Dessa forma, o balanceador de carga consegue distribuir a próxima requisição para o servidor com a menor carga naquele momento. A Figura

4 ilustra um cenário onde os servidores 1 e 2 estão com mais conexões ativas do que o servidor 3. O balanceador, para obter essa informação, consulta a quantidade de conexões em cada servidor antes de decidir para qual será enviado a requisição. No NGINX [1], caso todos possuam o mesmo número de conexões ativas, é utilizado o Round Robin para decidir.

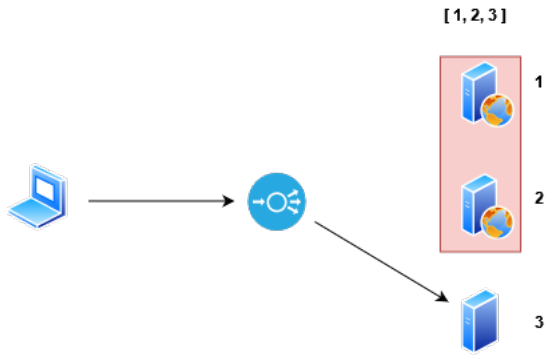

Figura 4 - Representação do algoritmo de balanceamento Least Connection, na qual os servidores 1 e 2 possuem mais conexões ativas que o 3.

2.4.2.2 Weighted Least Connection

Neste caso, o algoritmo dispõe de uma configuração inicial, a qual estabelece o quanto cada servidor disponível é proporcionalmente mais capaz de receber requisições. Dessa forma, não é escolhido o servidor com o menor número de conexões ativas no momento, necessariamente, mas aquele o qual possui menor valor em proporção. O cálculo desta proporção é a razão entre o número de conexões ativas e o peso atribuído. A Figura 5 mostra um cenário onde o servidor 1 tem peso 3, e os demais peso 1. O servidor 1 está com 3 conexões ativas, logo a proporção está em 1 (3/3), já os servidores 2 e 3 estão com 2 conexões ativas, fazendo que a proporção seja 2 (2/1).

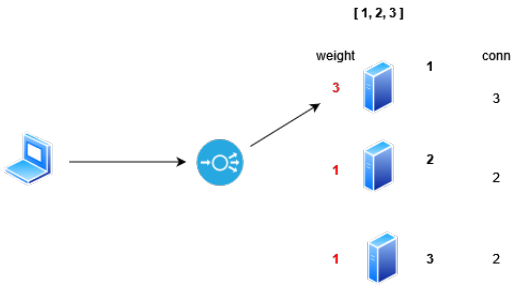

Figura 5 - Representação do algoritmo de Weighted Least Connection, na qual o servidor 1 possui peso 3 e está com 3 conexões ativas, enquanto os servidores 2 e 3 possuem peso 1 e estão com 2 conexões ativas cada.

2.5 Latência

Uma das métricas de desempenho a ser analisada e relacionada com a performance é a latência. A qual se refere ao *delay* entre a chamada feita pelo cliente e o retorno enviado pelo servidor. Ou seja, o tempo entre o cliente realizar a requisição e receber sua resposta.

2.6 Requisições por Segundo (RPS)

É a medida pela qual analisamos o *throughput* da aplicação. Consiste em analisar quantas requisições por segundo são recebidas, processadas e retornadas para o cliente.

2.7 Falha

Uma falha na resposta pode ser quando a latência aumentou e o servidor não conseguiu receber a requisição feita pelo cliente, tornando a resposta um timeout. Além da latência, o alto nível de processamento do servidor também pode acarretar em uma falha na resposta.

2.8 CPU credit balance

Nos servidores AWS existe o conceito de *CPU credit balance*, o qual é responsável por fornecer um ganho de processamento na utilização da CPU, com base no histórico de uso da mesma. Se o processo que está sendo executado fica abaixo da *baseline* de processamento para o tier (nível) da máquina, existe um acúmulo de créditos que são posteriormente utilizados. No cenário inverso, quando, esse baseline é ultrapassado, o servidor utiliza dos créditos acumulados para ter um ganho em seu processamento [8].

2.9 Locust

Ferramenta *open source* para testes de performance [2]. Nela é possível determinar a quantidade de usuários virtuais e o intervalo de subida entre eles para realizarem requisições em um dado host, essa configuração é feita via interface web ou linha de comando. Também podemos definir, via código Python, as *tasks* que os usuários vão executar em suas requisições.

3 Metodologia

Como o objetivo do trabalho é avaliar o desempenho dos diferentes tipos de algoritmos de roteamento de um load balancer, nosso ambiente conta com três servidores, um para o load balancer e outros dois que hospedam a API desenvolvida.

3.1 Topologia do sistema

Toda nossa infraesturtura está hospedada na AWS, pois trata-se de um ambiente muito utilizado pelo mercado, e também com uma alta confiabilidade. Todas as máquinas mencionadas a seguir seguem a especificação da instância *t2.micro*. A Figura 6 ilustra como as máquinas estão divididas. O load balancer é o ponto de entrada da requisição, nele é

determinado para qual dos dois servidores a requisição deverá ser atendida, com base no algoritmo de roteamento. Uma vez que o servidor processa e retorna para o load balancer a reposta, a qual é direcionada ao cliente.

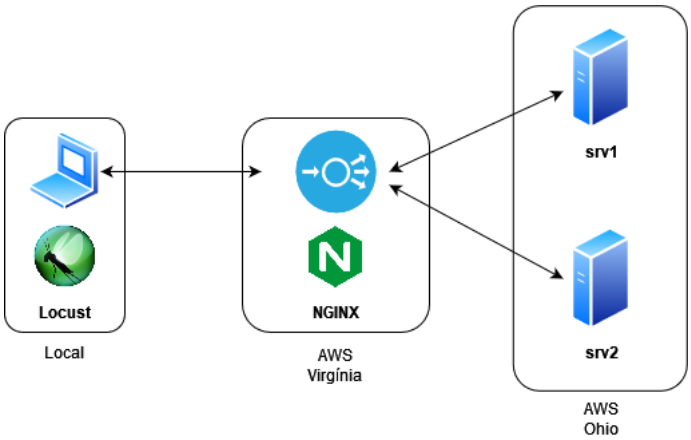

Figura 6 - Representação da topologia do sistema.

3.2 Cliente e servidor

Para o cliente, utilizamos a ferramenta Locust para fazer as requisições do teste de carga e salvar as métricas utilizadas para a comparação. O Locust disponibiliza os dados de cada teste em formato csv, facilitando assim a manipulação dos mesmos.

O servidor, por sua vez, realiza uma multiplicação de duas matrizes 20x20, preenchidas com números gerados aleatoriamente, a fim de simular o processamento de uma aplicação real. Desta forma, conseguimos deixar um servidor ocupado por um tempo, buscando avaliar o real impacto do uso de um load balancer. Tanto o algoritmo do cliente quanto do servidor foram desenvolvidos utilizando o Python 3.7.

3.3 Teste de Carga

Para facilitar a coleta de dados e uma melhor análise estatística, como desvio padrão, média e mediana, criamos um script em Python para rodar os testes da maneira automática. O qual consiste em executar o teste do Locust três vezes, por 60 segundos e uma quantidade de usuários simultâneos. Definimos, para compreender o comportamento dos algoritmos em diferentes cenários, as quantidades de 10, 100, 250 e 500 usuários simultâneos.

4 Resultados

O script é executado, para cada algoritmo, 3 vezes. Dessa forma, conseguimos eliminar eventuais erros sistemáticos e também calcular o desvio padrão das medidas, a fim de uma análise dos dados mais completa.

4.1 Round Robin

A Tabela 1 mostra os resultados referentes à quantidade de requisições, falhas e RPS para o algoritmo Round Robin e a Tabela 2 contém os dados obtidos para o tempo da latência média, mínima, máxima e também sua mediana.

Usuários Concorrentes	**Total de Requisições**	σ	**RPS**	σ	**Falhas/s**	σ
10	2641	61,45	44,67	1,07	0	0
100	5021	47,51	84,80	0,83	0	0
250	5173	616,24	87,36	10,37	0	0
500	22532	1172,59	379,99	19,93	234,62	18,98

Tabela 1: Resultados de total de requisições, RPS e falhas por segundo para o algoritmo de Round Robin

Usuários Concorrentes	**Mediana Latência (ms)**	σ	**Média Latência (ms)**	σ	**Latência Mínima (ms)**	σ	**Latência Máxima (ms)**	σ
10	200	0	223,10	5,46	171,91	0,83	1.370,47	415,51
100	710	75,06	1.160,26	11,74	171,32	0,10	3.399,41	284,40
250	720	235,16	2.359,35	152,75	172,38	0,59	38.023,90	2580,06
500	370	10,00	1.221,55	61,92	0,84	0,04	29.553,42	20601,59

Tabela 2: Resultados da mediana, média, máximos e mínimos da latência.

No ambiente da AWS, como descrito na Figura 6, identificamos um comportamento diferente do esperado para uma aplicação com recursos homogêneos, no caso do Round Robin. Seu desempenho está inferior ao esperado, atingindo números muito baixos de Requisições por Segundo. O algoritmo está, como exemplificado na Figura 1, distribuindo as requisições de forma homogênea aos servidores. Cada um recebe a próxima sucessivamente. Com isso, utilizamos o teste com o Weighted Round Robin para investigar e notamos que uma das máquinas utilizadas como servidor da aplicação estava com um desempenho bem abaixo da outra. Fato que pode ser explicado pela utilização de crédito de cada CPU. Assim, seu desempenho pode ser analisado identificando a aplicação como heterogênea. Dessa forma, seu comportamento esperado seria uma baixa performance, pois distribui a carga de forma igual aos servidores, tendo como gargalo o com pior desempenho.

4.2 Weighted Round Robin

Para o Weighted Round Robin existe dois cenários: o primeiro no qual o servidor 1 recebe peso 3 e o segundo onde o servidor 2 recebe o peso 3. Desta forma, temos os resultados para o primeiro cenário na Tabela 3 e Tabela 4. Já os resultados do segundo são apresentados na Tabela 5 e Tabela 6.

4.2.1 Weighted Round Robin com peso 3 no servidor 1

Usuários Concorrentes	**Total de Requisições**	σ	**RPS**	σ	**Falhas/s**	σ
10	2934	63,91	49,56	1,09	0	0
100	10.183	88,83	172,07	1,46	0	0
250	10574	237,42	178,57	4,01	0	0
500	25532	538,64	430,36	9,01	247,73	7,43

Tabela 3: Resultados para o Weighted RR, com peso 3 no servidor 1.

Usuários Concorrentes	**Mediana Latência (ms)**	σ	**Média Latência (ms)**	σ	**Latência Mínima (ms)**	σ	**Latência Máxima (ms)**	σ
10	190	0	201,17	4,50	171,74	1,56	1.024,95	260,54
100	220	5,77	574,07	4,40	171,38	0,44	2.275,00	225,94
250	400	393,23	1.298,29	12,44	173,29	1,76	35.776,81	4397,76
500	400	10	1.107,53	54,70	0,92	0,08	58.754,32	691,06

Tabela 4: Mediana, média, latência miníma e máxima para o WRR.

Como mencionado no Round Robin, conseguimos notar aqui que com o servidor 1 configurado com peso 3 no balanceamento, o desempenho da aplicação aumenta consideravelmente. Em todos os casos com 10 usuários concorrentes, a escolha do algoritmo não traz muito impacto. Mas, a partir de 100, conseguimos notar os resultados esperados. No caso, considerando o sistema como heterogêneo, temos a informação que o servidor 1 possui melhor performance. Assim, como algoritmo estático, apenas configuramos para que ele possua peso 3. Logo, nota-se que diminuímos o gargalo criado pelo servidor 2.

4.2.2 Weighted Round Robin com peso 3 no servidor 2

Usuários Concorrentes	**Total de Requisições**	σ	**RPS**	σ	**Falhas/s**	σ
10	2547	17,52	43,04	0,29	0	0
100	3338	35,16	56,42	0,60	0	0
250	4905	311,90	82,80	5,16	0	0
500	22903	527,67	386,20	8,84	267,39	15,93

Tabela 5: Total de requisições, RPS e falhas por segundo para o WRR, com peso 3 no servidor 2.

Usuários Concorrentes	**Mediana Latência (ms)**	σ	**Média Latência (ms)**	σ	**Latência Mínima (ms)**	σ	**Latência Máxima (ms)**	σ
10	210	5,77	231,60	1,59	172,49	0,42	1.146,14	177,39
100	2.200	0	1.738,79	18,40	172,45	0,51	4.046,64	477,40
250	1500	305,51	2.716,89	183,26	172,64	1,29	31.676,93	3556,79
500	350	5,77	1.187,67	7,04	0,87	0,04	39.149,78	14560,90

Tabela 6: Mediana, média, latência máxima e mínima para o WRR com peso 3 no servidor 2.

Com a finalidade de ter clareza que o servidor 2 estava apresentando um desempenho inferior do que o esperado, realizamos o teste de carga com os pesos dos servidores invertidos. Assim, agora, com o servidor 2 configurado com peso 3, identificamos a queda de performance. Em relação ao Round Robin, identificamos algumas semelhanças nos números de RPS, no entanto, como mais requisições estão sendo balanceadas para o servidor 2, nota-se que o número total de requisições foi um pouco menor.

4.3 IP Hash

Neste caso, os clientes do teste de carga sempre terão o mesmo IP, então observamos a distribuição de carga sendo direcionada sempre para o mesmo servidor. A fim de apresentar os diferentes cenários, configuramos o IP Hash apontando apenas para um dos servidores. Os resultados do servidor 1 estão nas Tabelas 7 e 8 e do servidor 2 estão nas Tabelas 9 e 10.

4.3.1 IP Hash com conexão no servidor 1

Usuários Concorrentes	**Total de Requisições**	σ	**RPS**	σ	**Falhas/s**	σ
10	3046	38,42	51,46	0,67	0	0
100	8743	11,53	147,70	0,13	0	0
250	8696	17,50	146,79	0,27	0	0,01
500	35594	9338,63	599,84	157,52	475,89	179,59

Tabela 7: Total de requisições, RPS e falhas por segundo para o IP hash, apenas com o servidor 1.

Usuários Concorrentes	**Mediana Latência (ms)**	σ	**Média Latência (ms)**	σ	**Latência Mínima (ms)**	σ	**Latência Máxima (ms)**	σ
10	190,00	0	193,62	2,58	171,21	0,33	594,31	31,12
100	670,00	0	672,17	0,41	340,67	33,97	1.805,16	559,43
250	1.100,00	0	1.601,99	60,20	587,46	191,33	38.074,98	13502,63
500	320,00	37,86	739,02	177,96	0,84	0,00	58.489,18	9255,75

Tabela 8: Mediana, média, latência máxima e mínima para o IP hash para o servidor 1.

Como temos uma aplicação com comportamento heterogêneo, foram feitos experimentos com a carga sendo direcionada para cada servidor. Neste momento, sendo balanceada para

o servidor 1, notamos seu comportamento semelhante ao Weighted Round Robin com peso 3 nele. Mas, como no IP Hash, estamos sobrecarregando um único servidor, isso impacta e notamos nos valores de requisições por segundo e na proporção de falhas ao compará-las ao WRR referido, pois estão menores e maiores respectivamente.

4.3.2 IP Hash com conexão no servidor 2

Usuários Concorrentes	**Total de Requisições**	σ	**RPS**	σ	**Falhas/s**	σ
10	2257	54,31	38,13	0,92	0	0
100	2421	24,43	40,90	0,41	0	0
250	2380	43,41	40,15	0,73	0,41	0
500	19878	559,96	335,25	9,55	295,70	9,41

Tabela 9: Total de requests, RPS e falhas por segundo para o IP hash e servidor 2.

Usuários Concorrentes	**Mediana Latência (ms)**	σ	**Média Latência (ms)**	σ	**Latência Mínima (ms)**	σ	**Latência Máxima (ms)**	σ
10	240	0	261,46	6,27	191,95	2,07	1.408,37	185,81
100	2400	0	2.395,83	24,03	1.961,18	647,59	4.269,74	654,78
250	3400	100	4.236,18	299,01	2.974,58	592,35	58.248,56	14958,77
500	340	5,77	898,52	110,22	0,99	0,15	57.888,57	557,34

Tabela 10: Mediana, média, latência máxima e mínima para o IP hash e servidor 2.

Este caso deixa claro o comportamento mencionado anteriormente sobre a performance do servidor 2. Como em nosso teste de carga todos os clientes possuem o mesmo ip, colocamos o servidor 2 como o primeiro a ser direcionado às requisições. Assim, notamos que apenas sobrecarregando ele evidenciamos seu desempenho. O qual atinge valores de RPS menores que o Round Robin, como esperado. E, notamos que começam a acontecer falhas antes que os outros algoritmos, já com 250 usuários concorrentes. E, com 500, o valor de falhas por segundo é muito perto do RPS.

4.4 Least Connection

Usuários Concorrentes	**Total de Requisições**	σ	**RPS**	σ	**Falhas/s**	σ
10	2899	80,56	48,96	1,36	0	0
100	11196	31,66	189,23	1,54	0	0
250	11173	13,58	188,86	0,20	0	0
500	25721	327,33	433,66	5,49	251,57	9,11

Tabela 11: Total de requests, RPS e falhas por segundo para o Least Connection.

Usuários Concorrentes	**Mediana Latência (ms)**	σ	**Média Latência (ms)**	σ	**Latência Mínima (ms)**	σ	**Latência Máxima (ms)**	σ
10	190	0	203,62	5,69	171,61	1,45	823,12	367,89
100	390	0	524,80	1,56	306,03	80,79	1.800,20	225,76
250	890	5,77	1.302,19	1,31	451,50	68,11	5.136,12	429,83
500	390	5,77	1.055,92	22,34	1,08	0,14	33.135,34	13871,00

Tabela 12: Mediana, média, latência máxima e mínima para o Least connection.

Neste caso, espera-se um bom desempenho entre os algoritmos no cenário descrito. Pois, dada sua característica dinâmica, espera-se que o servidor com menor desempenho impacte de forma que as requisições sejam distribuídas de maneira otimizada em relação às conexões. De fato, vemos que são os dados que atingem um maior valor de RPS, mas que não são tão elevados quanto o WRR com peso 3 no servidor 1. Mas, esse comportamento é compreensível, pois o WRR também distribui entre ambos os servidores mas não de forma tão otimizada quanto o Least Connection. E, analisando as falhas com 500 usuários concorrentes, vemos que sua performance é umas das melhores comparando os valores de requisição por segundo com falhas por segundo.

4.5 Weighted Least Connection

Usuários Concorrentes	**Total de Requisições**	σ	**RPS**	σ	**Falhas/s**	σ
10	3038	40,15	51,35	0,73	0	0
100	11260	26,41	190,32	0,32	0	0
250	11220	25,71	189,68	0,42	0	0
500	26048	687,64	439,86	11,97	252,11	11,11

Tabela 13: Total de requests, RPS e falhas por segundo para o Weighted Least Connection.

Usuários Concorrentes	**Mediana Latência (ms)**	σ	**Média Latência (ms)**	σ	**Latência Mínima (ms)**	σ	**Latência Máxima (ms)**	σ
10	190	0	194,17	2,71	172,70	2,24	777,97	218,17
100	510	0	522,32	1,08	364,73	117,75	1.019,70	485,66
250	1.100	0	1.299,55	3,48	651,49	121,01	16.415,41	23960,02
500	380	0	1.057,77	37,97	0,92	0,03	58.024,61	588,23

Tabela 14: Mediana, média, latência máxima e mínima para o Weighted Least Conn

Para o Weighted Least Connection, apenas foi avaliado com o peso no servidor de melhor desempenho. Pois, seu comportamento dinâmico de balanceamento, implica que sua performance será otimizada de acordo com a capacidade dos servidores. Assim, percebe-se que ele manteve o desempenho visto no Least Connection.

4.6 Comparação final

Dada a análise individual para cada algoritmo realizada anteriormente, percebe-se que em 250 usuários concorrentes há uma boa caracterização das performances dos algoritmos sem a ocorrência significativa de falhas. Pois, a única visualização de falhas ocorre no caso do IP Hash direcionado ao servidor 2, o qual apresentou 0,41 falhas por segundo. Dessa forma, apresenta-se nas Tabelas 15 e 16 uma comparação final sobre as métricas obtidas.

Assim, pode-se tirar as seguintes observações. Na aplicação com comportamento heterogêneo utilizada, os algoritmos de WRR, com peso no servidor 1, LC e WLC tiveram as melhores performances em relação à latência, RPS e falhas/s. No entanto, entre eles, destaca-se que a utilização do estático teve uma performance melhor ao comparar às latências medianas. Mas, ao analisar as latências máximas, as quais impactam a experiência do usuário significativamente, o LC mostra-se melhor. Nos demais, podemos visualizar que, apesar de uma latência semelhante aos LC e WLC, o Round Robin teve um número de RPS baixo.

	Usuários Concorrentes	**Total de Requisições**	σ	**RPS**	σ	**Falhas/s**	σ
RR	250	5173	616,24	87,36	10,37	0,00	0,00
WRR-srv1	250	10574	237,42	178,57	4,01	0,00	0,00
WRR-srv2	250	4905	311,90	82,80	5,16	0,00	0,00
IPH-srv1	250	8696	17,50	146,79	0,27	0,00	0,01
IPH-srv2	250	2380	43,41	40,15	0,73	0,41	0,00
LC	250	11173	13,58	188,86	0,20	0,00	0,00
WLC	250	11220	25,71	189,68	0,42	0,00	0,00

Tabela 15: Total de requests, RPS e falhas por segundo

	Usuários Concorrentes	**Mediana Latência (ms)**	σ	**Média Latência (ms)**	σ	**Latência Mínima (ms)**	σ	**Latência Máxima (ms)**	σ
RR	250	720	235,16	2359,35	152,75	172,38	0,59	38023,90	2580,06
WRR-srv1	250	400	393,23	1298,29	12,44	173,29	1,76	35776,81	4397,76
WRR-srv2	250	1500	305,51	2716,89	183,26	172,64	1,29	31676,93	3556,79
IPH-srv1	250	1100	0,00	1601,99	60,20	587,46	191,33	38074,98	13502,63
IPH-srv2	250	3400	100,00	4236,18	299,01	2974,58	592,35	58248,56	14958,77
LC	250	890	5,77	1302,19	1,31	451,50	68,11	5136,12	429,83
WLC	250	1100	0,00	1299,55	3,48	651,49	121,01	16415,41	23960,02

Tabela 16: Latência: Mediana, média, máxima e mínima

4.6.1 Latência

Para uma melhor compreensão, consolidamos os valores das latências para cada algoritmo, com base no número de requisições simultâneas.

4.6.1.1 Mediana

A Figura 7 ilustra o gráfico com os resultados dos algoritmos. Temos que a mediana da latência segue o esperado. O algoritmo que resultou na melhor performance foi o Weighted

Round Robin, com o peso 3 no servidor 1. O pior algoritmo foi o IP Hash que utilizou o servidor 2, já que esse estava com o desempenho comprometido por falta de CPU credits, como explicado acima.

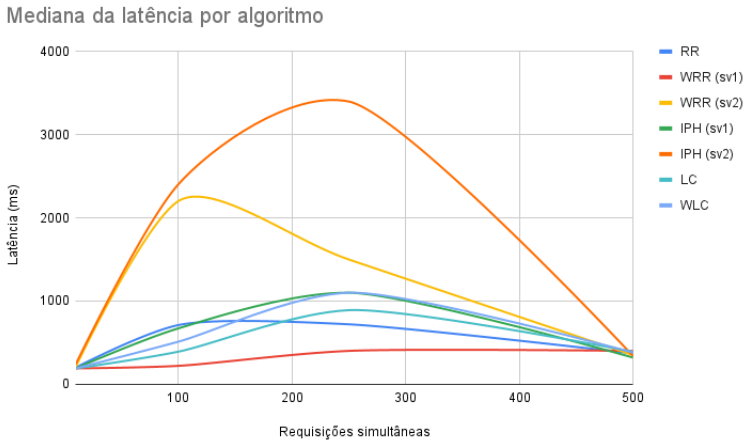

Figura 7 - Valor da mediana para cada algoritmo de acordo com as requisições simultâneas.

4.6.1.2 Média

A Figura 8 mostra os resultados para a média. Para ela, temos que os algoritmos Weighted Round Robin (sv1), Least Conn e Weighted Least Conn tiverem praticamente a mesma performance, ficando um pouco acima de 1 segundo de latência quando as requisições simultâneas foram maiores que 200. O IP Hash com o servidor 2 mostra-se, mais uma vez, estar com o pior desempenho dentre os algoritmos.

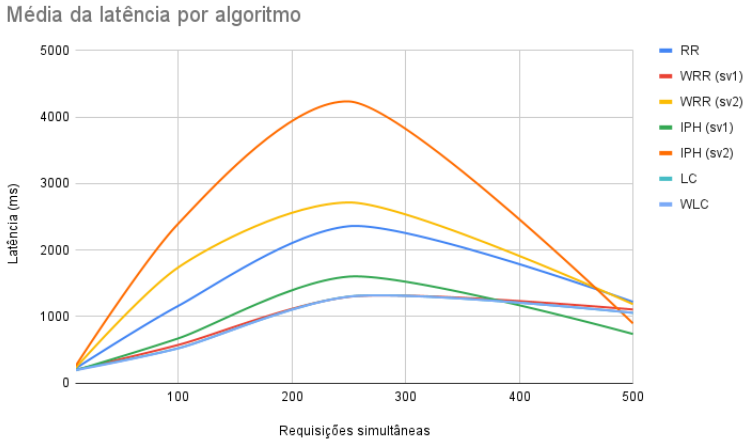

Figura 8 - Média da latência de retorno para cada algoritmo.

4.6.1.3 Mínima

A Figura 9 disponibiliza a visualização com as curvas de todos os algoritmos. Para os valores mínimos, os algoritmos mostram uma performance parecida, com exceção do IP Hash com o servidor 2. Esse resultado é esperado pois, como dito anteriormente, a performance inferior do servidor 2 é evidenciada pela do IP Hash (sv2) chegando aos 3s de latência mínima, enquanto os demais algoritmos ficaram abaixo do 1s.

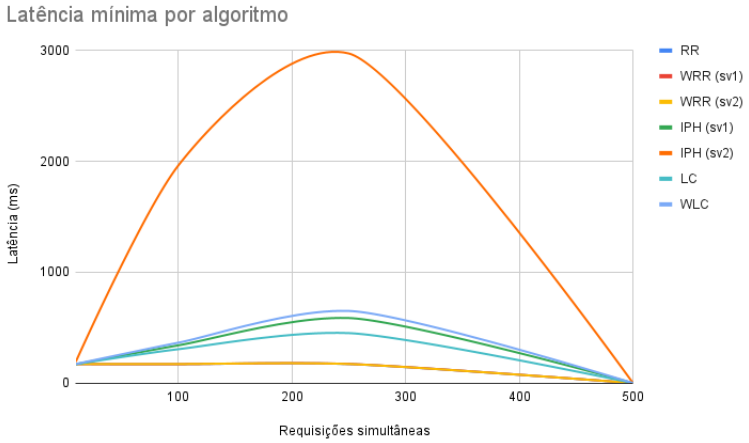

Figura 9 - Valor da latência mínima para cada algoritmo.

4.6.1.4 Máxima

A Figura 10 mostra os valores máximos da latência. Podemos notar que a latência de cada algoritmo sobe proporcionalmente ao número de requisições simultâneas feitas. Além disso, vemos também que o Least Conn foi o algoritmo que manteve a menor latência, mesmo com as 500 requisições. Isso mostra a excelência do mesmo para fazer a distribuição com base no processamento dos servidores.

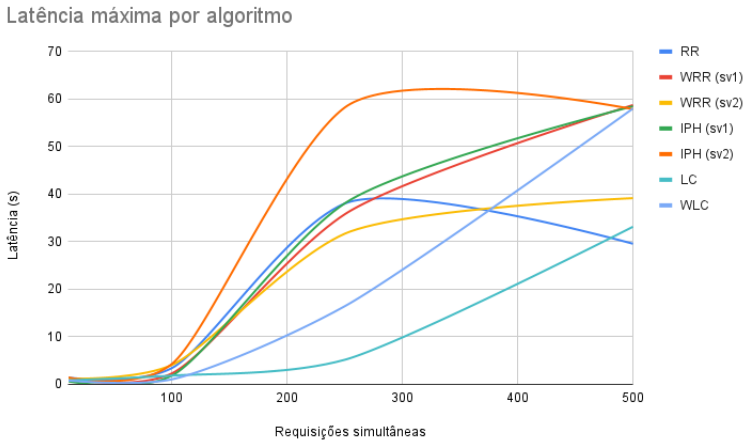

Figura 10 - Valor da latência máxima para cada algoritmo.

4.6.2 Falhas por segundo

A Figura 11 compara os valores das falhas por segundo. Nas análises individuais mostrou-se que em 500 usuários concorrentes todos os algoritmos apresentaram falhas por segundo. Podemos ver que as falhas acompanham a performance dos servidores. O IP Hash (sv2) está com as maiores falhas, quase 500 falhas/segundo. Os demais apresentam falhas menores que 300, e os algoritmos que distribuem as requisições entre os servidores, como o RR, WLC e LC apresentam os menores valores.

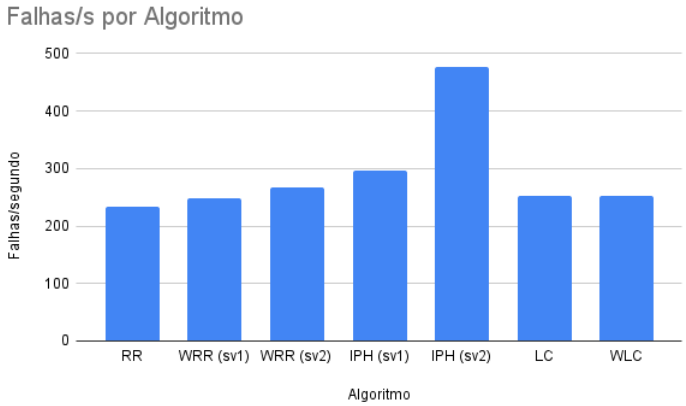

Figura 11 - Quantidade de falhas/s por tipo de algoritmo.

5 Conclusão

Neste estudo foi possível diferenciar a performance dos algoritmos dinâmicos e estáticos, assim como identificar suas características. Notou-se também que, em uma configuração

heterogênea de máquinas, quanto o poder computacional dos servidores impactam os resultados.

Dessa forma, pôde-se ver que alguns algoritmos conseguem deixar essa diferença menos evidente, fazendo um balanceamento mais inteligente entre os servidores. É o caso do Least Connection, que mostrou o menor valor entre as latências máximas dos algoritmos analisados. Por outro lado, pode-se notar, nos casos de balanceamento com maior direcionamento ao servidor de menor desempenho, o quanto essa decisão prejudica o sistema.

Referências

[1] **NGINX**. Acesso em 12 dezembro de 2021. Disponível em: <https://www.nginx.com/>

[2] **Locust**. Acesso em 12 dezembro de 2021. Disponível em: <https://locust.io/>

[3] N. J. Yeager, R. E. McGrath, **Web Server Technology**, Morgan Kaufmann Publishers, p. 14-55, 1996

[4] C. Kopparapu, **Load Balancing Servers, Firewalls and Caches**, John Wiley & Sons Inc, p. 10-12, 2002

[5] J. Vashistha, A. K. Jayswal, **Comparative Study of Load Balancing Algorithms**, IOSR Journal of Engineering (IOSRJEN), Vol. 3, p. 45-50, 2013

[6] **Success Stories**. Acesso em 12 dezembro de 2021. Disponível em: <https://www.nginx.com/success-stories/>

[7] **Inside NGINX: How we designed for Performance Scale**. Acesso em: 12 dezembro de 2021. Disponível em: <https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale>

[8] **Key concepts and definitions for burstable performance instances**. Acesso em: 12 dezembro de 2021. Disponível em: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html>