UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

VITOR GREGORIO WUCHRYN MARTINS

ANÁLISE COMPARATIVA DE BALANCEADORES DE CARGA DOS SERVIDORES WEB NGINX E APACHE

GUARAPUAVA

2026

VITOR GREGORIO WUCHRYN MARTINS

ANÁLISE COMPARATIVA DE BALANCEADORES DE CARGA DOS SERVIDORES WEB NGINX E APACHE

Comparative Analysis of Load Balancers of the NGINX and APACHE Web Servers

Projeto de Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Tecnologia em Sistemas para Internet do Curso Superior de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dra. Sediane Carmem Lunardi Hernandes

Coorientador: Prof. Dr. Hermano Pereira

GUARAPUAVA

2026

4.0 Internacional

Esta licença permite compartilhamento, remixe, adaptação e criação a partir do trabalho, mesmo para fins comerciais, desde que sejam atribuídos créditos ao(s) autor(es). Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

RESUMO

Palavras-chave:

ABSTRACT

Keywords:

LISTA DE FIGURAS

Figura 1 – Processo do Apache (a) e processo de trabalho do Nginx (b) **10**
Figura 2 – Núcleo do Apache, *hooks* e funções registradas pelos módulos **10**
Figura 3 – Sete requisições distribuídas a três servidores por *round-robin*, *least-connection* e *ip-hash* **12**
Figura 4 – Atendimento de uma página estática (a) e de uma página dinâmica (b) . **13**
Figura 5 – Serviço como sistema de fila (a) e tempo de resposta relativo, $R/S = 1/(1-U)$**, em função da utilização** U **(b)** **14**
Figura 6 – Geração de carga em malha fechada e em malha aberta **15**
Figura 7 – Organização do ambiente de experimentos **19**
Figura 8 – Afinidade por rota no Apache, do endereço de origem ao servidor de destino **22**
Figura 9 – Caso normal (a), falha do servidor depois de executar a requisição (b) e antes de executá-la (c) **22**
Figura 10 – Protocolo de medição de uma condição experimental **24**

LISTA DE QUADROS

Quadro 1 – Mapeamento dos algoritmos para a configuração de cada servidor . . 12
Quadro 2 – Máquina dos experimentos 17
Quadro 3 – Componentes de software 18
Quadro 4 – Fatores e níveis 20
Quadro 5 – Parâmetros fixados nos balanceadores 21

LISTA DE ABREVIATURAS E SIGLAS

Abreviaturas

art.	Artigo
cap.	Capítulo
sec.	Seção

Siglas

CPU	Unidade Central de Processamento, do inglês *Central Processing Unit*
DNS	Sistema de Nomes de Domínio, do inglês *Domain Name System*
HTTP	Protocolo de Transferência de Hipertexto, do inglês *Hypertext Transfer Protocol*
IP	Protocolo de Internet, do inglês *Internet Protocol*
MPM	Módulo de Multiprocessamento, do inglês *Multi-Processing Module*
NAT	Tradução de Endereços de Rede, do inglês *Network Address Translation*
PHP	*PHP: Hypertext Preprocessor*
TCC	Trabalho de Conclusão de Curso
TCP	Protocolo de Controle de Transmissão, do inglês *Transmission Control Protocol*
TLS	Segurança da Camada de Transporte, do inglês *Transport Layer Security*
URL	Localizador Uniforme de Recursos, do inglês *Uniform Resource Locator*
UTFPR	Universidade Tecnológica Federal do Paraná

SUMÁRIO

1 INTRODUÇÃO 7

1.1 Objetivos 8

1.1.1 Objetivo geral 8

1.1.2 Objetivos específicos 8

1.2 Estrutura do trabalho 8

2 REFERENCIAL TEÓRICO 9

2.1 Balanceamento de carga e *proxy* reverso 9

2.2 Arquiteturas dos servidores web Nginx e Apache 9

2.3 Algoritmos de balanceamento 10

2.4 Páginas estáticas e dinâmicas 11

2.5 Expectativas teóricas e resultados anteriores 13

2.6 Métricas, capacidade e geração de carga 13

2.7 Trabalhos relacionados 15

3 METODOLOGIA 17

3.1 Materiais 17

3.1.1 Hardware 17

3.1.2 Software 18

3.1.3 Ambiente experimental 18

3.2 Métodos 19

3.2.1 Fatores, níveis e condições experimentais 19

3.2.2 Páginas estáticas e dinâmicas 20

3.2.3 Configuração dos balanceadores 20

3.2.4 Gerador de carga 22

3.2.5 Protocolo de medição 23

3.2.6 Métricas e coleta 24

3.2.7 Validação e análise 24

3.2.8 Automação e reprodutibilidade 25

3.2.9 Limitações 25

4 CONSIDERAÇÕES FINAIS 27

REFERÊNCIAS 28

1 INTRODUÇÃO

Um balanceador de carga é o componente que recebe todas as requisições dirigidas a um serviço e as distribui entre vários servidores que executam a mesma aplicação, de modo que o cliente enxerga um único ponto de entrada (CARDELLINI *et al.*, 2002). Na web, esse papel costuma ser exercido por um servidor web configurado como *proxy* reverso: ele recebe a conexão do cliente, interpreta a requisição HTTP e a retransmite a um servidor de destino escolhido por um algoritmo de balanceamento.

O balanceador surgiu como resposta ao limite do servidor único. Ultrapassado o seu limite de carga, um servidor passa a recusar conexões e a responder com erros de serviço indisponível, o que não atende às exigências de disponibilidade e confiabilidade das aplicações web modernas. Há duas formas de lidar com esse limite: a escala vertical, que aumenta a capacidade de uma mesma máquina, e a escala horizontal, que aumenta a quantidade de máquinas semelhantes e, por meio de um único ponto de entrada, distribui entre elas as requisições recebidas (CARDELLINI *et al.*, 2002). O custo de processamento da aplicação é diluído entre as réplicas, mas o custo do balanceamento entra em todas as requisições que passam pelo sistema, e Cardellini, Colajanni e Yu (1999) já apontavam esse componente central como gargalo potencial de toda a arquitetura. Pelo mesmo motivo, aprimoramentos pequenos na sua eficiência, na latência e nos recursos que consome por requisição, se multiplicam por todo o tráfego.

Os servidores web mais comuns nesse papel são o Nginx e o Apache HTTP Server. Os dois oferecem algoritmos de famílias distintas: alternância fixa entre os servidores (*round-robin*), escolha pelo servidor com menos requisições em andamento (*least-connection*) e associação fixa entre um cliente e um servidor (*ip-hash* e seus equivalentes). Esses algoritmos não se comportam da mesma forma em todo tipo de tráfego, e a ordem entre eles muda com as condições do experimento (CARDELLINI; COLAJANNI; YU, 1999; MA; CHI, 2022).

Um fator que esses estudos não isolam é o tipo de página solicitada. Uma página estática é um arquivo pronto, entregue com custo baixo e previsível. Uma página dinâmica é montada a cada requisição por código de aplicação, com tempo de serviço maior e variável. Essa diferença atinge diretamente o que cada algoritmo pressupõe: o *round-robin* supõe requisições de custo parecido, o *least-connection* existe para quando esse custo varia, e o *ip-hash* troca a distribuição uniforme pela afinidade entre cliente e servidor. Além disso, Nginx e Apache têm arquiteturas internas diferentes, e por isso o custo do próprio balanceador pode variar entre os dois para a mesma carga.

Saber qual algoritmo e qual servidor entregam cada tipo de página com menor latência e menor consumo de recursos ajuda a dimensionar ambientes com recursos limitados. Lima, Pereira e Hernandes (2026) mostraram, em *hardware* reaproveitado, que a escolha de um componente de infraestrutura, e não da aplicação, alterou o tempo de resposta em mais de 100%; o balanceador de carga é um componente desse mesmo tipo.

Assim, este trabalho formula a seguinte pergunta de pesquisa: qual algoritmo de balanceamento de carga é mais eficiente para requisições de páginas estáticas e para requisições de páginas dinâmicas, e como esses algoritmos se comportam nos dois servidores web, Nginx ou Apache? Para respondê-la, propõe-se a comparar experimentalmente o Nginx e o Apache HTTP Server como balanceadores de carga, com os algoritmos *round-robin*, *least-connection* e *ip-hash*, servindo páginas estáticas e dinâmicas.

1.1 Objetivos

1.1.1 Objetivo geral

Realizar uma análise comparativa entre os diferentes algoritmos de balanceamento de carga disponíveis nos servidores web NGINX e Apache mostrando a eficiência de cada um em relação às solicitações de páginas web estáticas e dinâmicas.

1.1.2 Objetivos específicos

1. Analisar teoricamente os diferentes algoritmos de balanceamento de carga.
2. Executar os experimentos com os três algoritmos, os dois balanceadores e os dois tipos de página (estáticas e dinâmicas).
3. Sintetizar as métricas de eficiência de cada algoritmo, por balanceador e tipo de página.
4. Avaliar qual algoritmo de balanceamento de carga é mais apropriado para solicitações de páginas estáticas e para solicitações de páginas dinâmicas, e em qual servidor apresenta melhor eficiência.

1.2 Estrutura do trabalho

O trabalho é dividido como segue. O Capítulo 1 apresenta o problema, a pergunta de pesquisa e os objetivos. O Capítulo 2 apresenta o referencial teórico: o balanceador de carga como *proxy* reverso, as arquiteturas dos servidores web Nginx e Apache, os três algoritmos, a distinção entre páginas estáticas e dinâmicas, as expectativas teóricas e os resultados anteriores, as métricas, a capacidade e a geração de carga, e os trabalhos relacionados. O Capítulo 3 apresenta a metodologia: os materiais e o ambiente experimental, os fatores e as condições, as páginas servidas, a configuração dos balanceadores, o gerador de carga, o protocolo de medição, as métricas e a coleta, a validação e a análise, a automação e as limitações. O Capítulo 4 traz as considerações finais.

2 REFERENCIAL TEÓRICO

2.1 Balanceamento de carga e *proxy* reverso

Cardellini, Colajanni e Yu (1999) classificam os sistemas de servidores web distribuídos pela entidade que distribui as requisições: o cliente, o DNS, um despachante dedicado ou os próprios servidores. O balanceador deste trabalho é um despachante: um componente central que recebe todas as requisições, escolhe o servidor de destino e controla todo o roteamento, a forma com maior controle sobre a distribuição e, por isso mesmo, um ponto único que pode virar gargalo. Despachantes que leem a requisição HTTP antes de decidir atuam na camada de aplicação, a camada 7 do modelo OSI; os que decidem só pelos cabeçalhos de rede e transporte atuam na camada 4, a camada de transporte, e podem ser mais rápidos, porque não interpretam a requisição da camada de aplicação.

Um servidor web atuando como *proxy* reverso é um despachante da camada de aplicação. Hosseini, Jahangir e Daraby (2021) enumeram o que ele faz a cada requisição: termina a conexão do cliente, lê e interpreta a requisição, abre ou reutiliza uma conexão com o servidor de destino, repassa a requisição, recebe a resposta e a devolve ao cliente, que nunca vê o endereço real dos servidores de destino. O Nginx, usado por cerca de um terço dos sites web (KITHULWATTA *et al.*, 2022), e o Apache HTTP Server, por meio dos módulos *mod_proxy* e *mod_proxy_balancer*, são os dois servidores mais comuns para esse papel. Heinzl e Metz (2013) escolheram o Apache pela familiaridade da equipe e pela facilidade de suporte, não por evidência de desempenho; essa ausência é parte da lacuna em que esse trabalho trabalha.

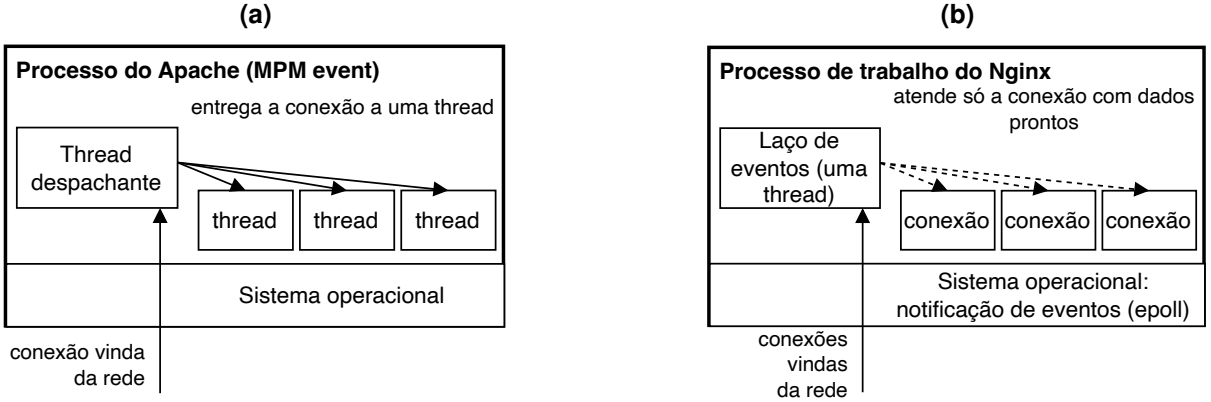
2.2 Arquiteturas dos servidores web Nginx e Apache

Os dois servidores atendem conexões de formas diferentes. O Nginx é orientado a eventos, em que um número fixo de processos, em geral um por núcleo de CPU, atende cada um milhares de conexões ao mesmo tempo, tratando cada uma só quando há dados prontos. O Apache delega esse modelo a módulos de multiprocessamento (*Multi-Processing Modules*, MPM); o MPM *event*, padrão nas instalações atuais, combina processos e *threads*, com *threads* dedicadas às conexões ociosas (APACHE SOFTWARE FOUNDATION, 2026c). O Apache utiliza o modelo de *multithreading*, Figura 1(a), e o Nginx, o de máquina de estados finitos, Figura 1(b).

Há três formas de se implementar uma *thread* (VAN STEEN; TANENBAUM, 2025): (i) *multithread*, com paralelismo e chamadas bloqueantes, o modelo de cada processo do MPM *event* do Apache; (ii) processo único, sem paralelismo; e (iii) máquina de estados finitos, com paralelismo e chamadas não bloqueantes, que é o modelo orientado a eventos, o de cada processo de trabalho do Nginx. O Apache é um núcleo que faz poucas suposições sobre o tratamento da requisição e executa, em pontos de *hook* ordenados, as funções que os módulos

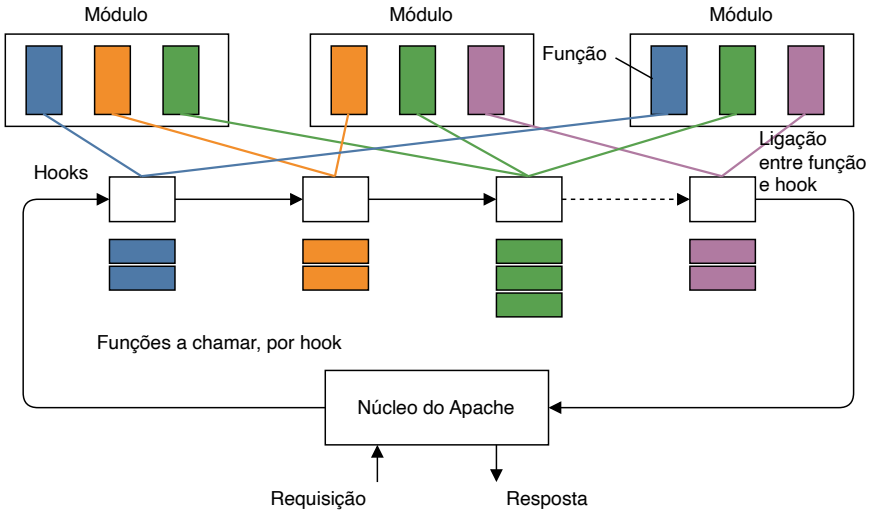
registram, Figura 2. Servidores baseados em uma *thread* por conexão perdem vazão quando o número de conexões cresce, enquanto os orientados a eventos se mantêm estáveis sob muitas conexões simultâneas (WELSH; CULLER; BREWER, 2001; PARIAG *et al.*, 2007). A causa está no escalonamento e nas trocas de contexto, cujo custo maior é indireto, pela poluição de *cache* (VAN STEEN; TANENBAUM, 2025). Como balanceador, cada servidor repete, por requisição, as etapas do *proxy* reverso, e a diferença de arquitetura é a razão para esperar que o custo dessas etapas em CPU e memória não seja igual nos dois.

Figura 1 – Processo do Apache (a) e processo de trabalho do Nginx (b)



Fonte: (a) Adaptada de Van Steen e Tanenbaum (2025); (b) autoria própria (2026).

Figura 2 – Núcleo do Apache, *hooks* e funções registradas pelos módulos



Fonte: Adaptada de Van Steen e Tanenbaum (2025).

2.3 Algoritmos de balanceamento

Os algoritmos se dividem pelo estado que consultam. Os estáticos seguem regras fixas, sem olhar o estado corrente dos servidores, e funcionam bem quando a carga varia pouco entre

eles; os dinâmicos monitoram o estado dos servidores e decidem a partir dele (CARDELLINI *et al.*, 2002). Nessa classificação, o *round-robin* e o *ip-hash* são estáticos, e o *least-connection* é dinâmico.

O *round-robin* alterna entre os servidores em ordem fixa, um passo por requisição. É o algoritmo mais simples e o de menor custo de decisão (CARDELLINI; COLAJANNI; YU, 1999). Sua premissa é que cada requisição consome aproximadamente o mesmo tempo no servidor de destino; se algumas demoram muito, a alternância fixa acumula carga em um servidor. Ele também trata todos os servidores como iguais, por isso só é adequado quando têm a mesma capacidade.

O *least-connection* encaminha cada requisição ao servidor com menos conexões ativas. É o algoritmo indicado quando algumas requisições são longas, e o que Tamanaha, Domingues e Bittencourt (2021) encontraram sustentando a menor latência máxima sob carga elevada com servidores heterogêneos.

O *ip-hash* associa cada cliente a um mesmo servidor a partir do *hash* do seu endereço IP. Garante que todas as requisições de um cliente cheguem ao mesmo servidor, o que aplicações com estado em memória precisam, mas impede qualquer balanceamento baseado na carga. No Nginx, o `ip_hash` usa os três primeiros octetos do endereço IPv4 (NGINX, 2026b), de modo que clientes de uma mesma rede, ou atrás do mesmo *proxy* ou NAT, caem no mesmo servidor. E uma função de *hash* só distribui bem os clientes quando há muitos endereços distintos (KARGER *et al.*, 1997). Ma e Chi (2022), mesmo com mil endereços de origem, observaram um servidor com visitas em excesso e outro em falta. Por isso, o desequilíbrio do *ip-hash* é esperado e precisa ser medido com um número de endereços conhecido.

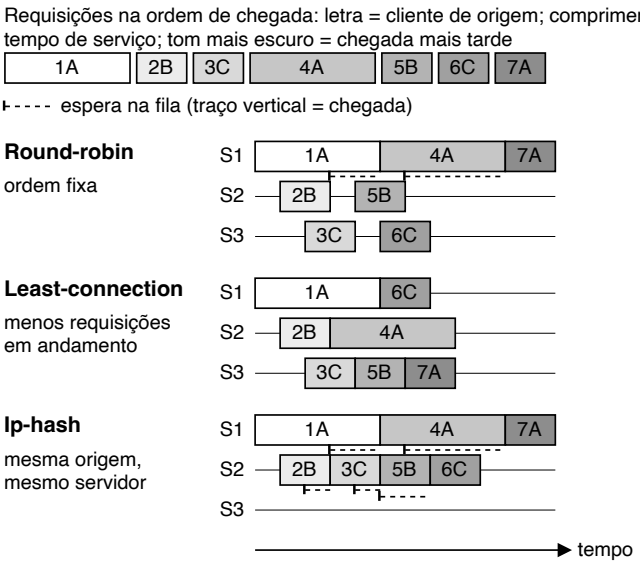
A Figura 3 mostra sete requisições iguais, duas delas longas, distribuídas pelos três algoritmos: a alternância fixa do *round-robin* faz duas requisições esperarem no mesmo servidor; o *least-connection* evita a espera; e o *ip-hash*, com dois clientes no mesmo servidor, deixa o terceiro ocioso.

No Apache, o *mod_proxy_balancer* oferece escalonadores por contagem de requisições (`byrequests`), por tráfego (`bytraffic`), por requisições pendentes (`bybusyness`) e por *heartbeat* (APACHE SOFTWARE FOUNDATION, 2026a). O `byrequests` com pesos iguais corresponde ao *round-robin*, e o `bybusyness` ao *least-connection*. Não há escalonador por *hash* de endereço: a documentação justifica a ausência pelos clientes atrás de *proxies* e pelos endereços dinâmicos, e o módulo implementa a afinidade por *cookie* ou por parâmetro de URL. O Quadro 1 resume o mapeamento.

2.4 Páginas estáticas e dinâmicas

Uma página estática é um arquivo já existente no servidor de destino, entregue sem processamento de aplicação: o servidor localiza o arquivo e envia o conteúdo (VAN STEEN; TANENBAUM, 2025). O tempo de serviço é curto e varia pouco entre requisições.

Figura 3 – Sete requisições distribuídas a três servidores por *round-robin*, *least-connection* e *ip-hash*



Fonte: Autoria própria (2026).

Quadro 1 – Mapeamento dos algoritmos para a configuração de cada servidor

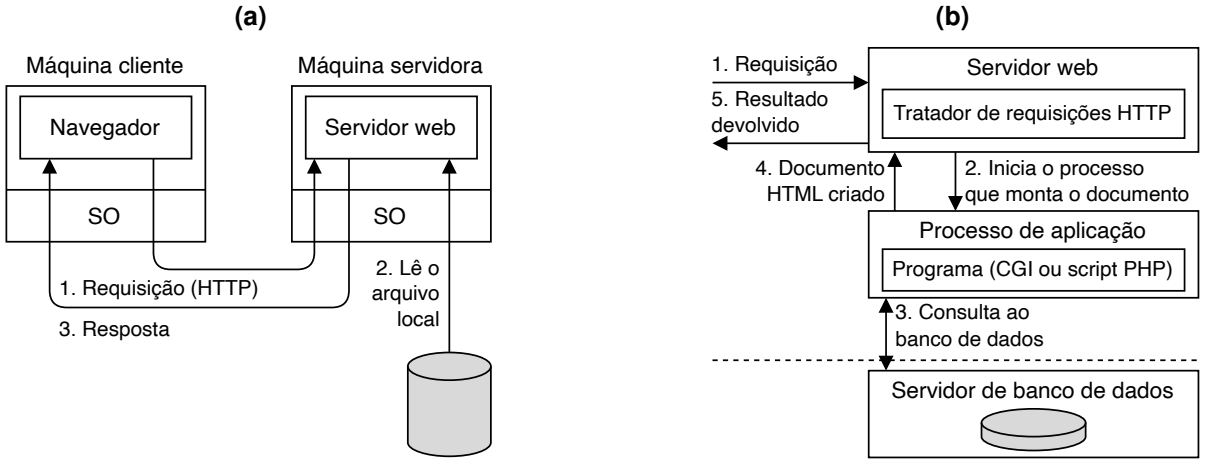
Algoritmo	**Família**	**Nginx**	**Apache (*mod_proxy_balancer*)**
round-robin	estático	padrão do grupo `upstream`	`lbmethod=byrequests` com pesos iguais
least-connection	dinâmico	`least_conn`	`lbmethod=bybusyness`
ip-hash	estático, com afinidade	`ip_hash`	sem equivalente nativo; afinidade de rota derivada do endereço de origem

Fonte: Autoria própria (2026).

Uma página dinâmica é gerada a cada requisição por código de aplicação, como um *script* PHP, que consome CPU para produzir a resposta e pode consultar outros serviços (VAN STEEN; TANENBAUM, 2025); o tempo de serviço é maior e variável. A Figura 4 contrasta os dois casos.

Essa diferença é o que liga o tipo de página à escolha do algoritmo. Páginas estáticas aproximam a condição ideal do *round-robin*: requisições de custo semelhante, para as quais a alternância fixa já é uniforme. Páginas dinâmicas aproximam a condição para a qual o *least-connection* foi concebido: tempos de serviço desiguais, em que contar requisições em andamento evita acumular trabalho em um servidor. A afinidade do *ip-hash* é indiferente para páginas estáticas, que não têm estado, e relevante para aplicações dinâmicas que mantêm sessão em memória. Além disso, com páginas estáticas o balanceador representa uma fração maior do tempo total da requisição, porque o servidor de destino responde rápido; com páginas

Figura 4 – Atendimento de uma página estática (a) e de uma página dinâmica (b)



Fonte: Adaptada de Van Steen e Tanenbaum (2025).

dinâmicas, o tempo do servidor de destino domina e o custo do balanceador se dilui. Por isso a eficiência de cada combinação precisa ser medida separadamente para os dois tipos.

2.5 Expectativas teóricas e resultados anteriores

A teoria aponta uma expectativa clara: *round-robin* para requisições de custo parecido, *least-connection* quando esse custo varia, e o *ip-hash* renunciando à distribuição uniforme em favor da afinidade (KUMAR; KUMAR, 2019).

Os resultados empíricos complicam essa expectativa: Cardellini, Colajanni e Yu (1999) observaram, por simulação, o *round-robin* superando a política de menor carga; Ma e Chi (2022), no Nginx com concorrência crescente, encontraram o *ip-hash* com o menor tempo médio de resposta e a menor taxa de falhas sob alta concorrência, e o *least-connection* com a maior vazão; e Jader *et al.* (2021 apud PINASTAWA; PRADANA; MAULANA, 2024) encontraram o *round-robin* ligeiramente melhor em agrupamentos Nginx e o *ip-hash* melhor em agrupamentos Apache. Nenhum deles controlou o tipo de página, fator que este trabalho acrescenta, com a hipótese de que páginas estáticas favorecem o *round-robin* e páginas dinâmicas, o *least-connection*. Para o *ip-hash*, a expectativa é aproximar-se do *round-robin* nas páginas estáticas e, nas dinâmicas, não superar o *least-connection* na cauda da latência; o seu desequilíbrio é reportado, não corrigido. A ordem dos algoritmos deve ser a mesma nos dois balanceadores, exceto pelo *ip-hash* do Apache, obtido por emulação, cujo custo entra na comparação.

2.6 Métricas, capacidade e geração de carga

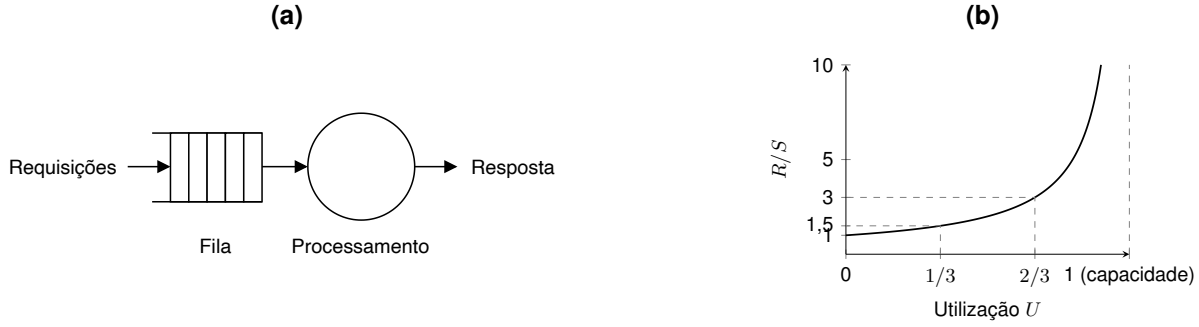
Kumar e Kumar (2019) listam as métricas usuais de balanceamento: tempo de resposta, vazão, uso de recursos e grau de desequilíbrio entre os nós; e registram que a literatura se

concentra no tempo de resposta e raramente mede custo. Kithulwatta *et al.* (2022) definem requisição falha como a que termina em erro de conexão, *timeout* ou código de *status* de erro, com o limiar prático de menos de 1% de falhas para considerar o experimento válido.

A latência é reportada em mediana e percentis 95 e 99, e não só em média, porque são as requisições mais lentas que distinguem um algoritmo que acumula requisições longas de um que as dilui; trabalhos anteriores reportam só médias (MA; CHI, 2022; PARIAG *et al.*, 2007). O consumo de CPU é medido no sistema inteiro da máquina do balanceador, somando espaço de usuário, *kernel* e interrupções, porque parte do trabalho do *proxy* reverso acontece no *kernel* e não apareceria na contabilidade do processo. Como o número de requisições de cada experimento é fixo, o consumo acumulado se converte em custo por requisição; a eficiência de uma combinação é esse custo lido em conjunto com a latência e a taxa de falhas. Cada metade da pergunta de pesquisa tem as suas métricas: com três servidores de destino, o custo de decisão dos três algoritmos é desprezível, e o custo por requisição do balanceador separa os dois servidores, não os algoritmos; estes se separam pela cauda da latência, pela vazão perto da capacidade, pela taxa de falhas acima dela e pelo desequilíbrio entre os servidores de destino.

A capacidade de uma combinação é a taxa oferecida a partir da qual a vazão deixa de crescer. Além desse ponto, a vazão cai, porque o servidor gasta CPU com conexões que vai descartar, e o modo de localizá-lo é repetir o experimento com taxas crescentes até a vazão atingir um patamar (BANGA; DRUSCHEL, 1997; MOSBERGER; JIN, 1998). Comparar sem conhecer a capacidade leva a resultados como os de Pinastawa, Pradana e Maulana (2024), que compararam dois servidores com falhas acima de 85%. Em um modelo de fila com chegadas independentes, Figura 5(a), o tempo de resposta R, relativo ao tempo de serviço S, segue $R/S = 1/(1-U)$, em que U é a utilização, e dispara quando U se aproxima de 1, Figura 5(b).

Figura 5 – Serviço como sistema de fila (a) e tempo de resposta relativo, $R/S = 1/(1-U)$, em função da utilização U (b)

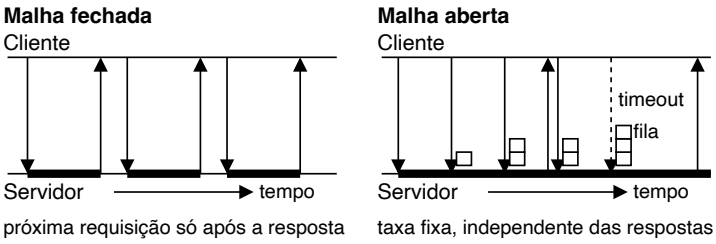


Fonte: Adaptada de Van Steen e Tanenbaum (2025).

A forma de gerar carga decide o que se consegue observar. No esquema usual, em malha fechada, cada cliente simulado só envia a próxima requisição depois de receber a resposta anterior, fica em passo com o servidor e não revela o comportamento em sobrecarga. Na malha aberta, o gerador envia requisições a uma taxa fixa, independentemente das respostas, com

número fixo de requisições e *timeout* por requisição (BANGA; DRUSCHEL, 1997; MOSBERGER; JIN, 1998). Esses autores alertam ainda que o gargalo pode migrar do servidor para o próprio gerador e prescrevem a verificação que o Capítulo 3 adota: o resultado não pode depender do número de geradores. A Figura 6 contrasta as duas formas. Na prática, só a malha aberta reproduz o que acontece quando os usuários continuam chegando sem esperar pelos anteriores, que é a situação que os experimentos querem observar.

Figura 6 – Geração de carga em malha fechada e em malha aberta



Fonte: Autoria própria (2026).

2.7 Trabalhos relacionados

Os estudos que avaliam algoritmos de balanceamento em servidores web o fazem, em geral, dentro de uma única ferramenta e sem separar o tipo de página. Tamanaha, Domingues e Bittencourt (2021) comparam os algoritmos do Nginx em nuvem e mostram que a heterogeneidade dos servidores altera a ordem entre eles. Ma e Chi (2022) avaliam os três algoritmos no Nginx com três servidores Tomcat e concorrência crescente, medindo tempo médio de resposta, taxa de falha e vazão; não medem CPU ou memória do balanceador, não reportam percentis nem repetições e não descrevem o conteúdo servido. Pramono, Buwono e Waskito (2018) são o comparativo mais próximo deste: HAProxy e Nginx, com *round-robin* e *least-connection*, à frente de três servidores web; mas não incluem o Apache, não descrevem o conteúdo nem as versões e não medem recursos do balanceador.

Os comparativos entre Nginx e Apache raramente os avaliam como balanceadores. Pinastawa, Pradana e Maulana (2024) comparam os dois servindo WordPress com o k6 e revisam nove trabalhos anteriores; entre eles, Putro e Supono (2022) comparam os dois como balanceadores sem variar o algoritmo, e Jader *et al.* (2021 apud PINASTAWA; PRADANA; MAULANA, 2024) variam o algoritmo sem variar o conteúdo.

A literatura sobre balanceamento em nuvem tem outro objeto, o escalonamento de tarefas para máquinas virtuais em simuladores (KUMAR; KUMAR, 2019; MORAIS *et al.*, 2017); a de avaliação de servidores web estabelece o método de medição, mas mede um único servidor com páginas estáticas (BANGA; DRUSCHEL, 1997; MOSBERGER; JIN, 1998; PARIAG *et al.*, 2007), caso também de Carraro (2025), que compara versões do HTTP em um único Nginx;

e Lima, Pereira e Hernandes (2026) fundamentam medir o custo do balanceador ao lado da latência.

Não há, nesses trabalhos, comparação entre Nginx e Apache como balanceadores de carga sob os mesmos algoritmos, a mesma carga, a mesma ferramenta e as mesmas métricas, nem avaliação do efeito do tipo de página sobre a escolha do algoritmo, nem medição do custo do próprio balanceador por requisição. É a lacuna que este trabalho pretende preencher.

3 METODOLOGIA

Este capítulo apresenta o plano dos experimentos em duas partes. A seção de materiais descreve o *hardware*, o software e o ambiente experimental; a seção de métodos define os fatores e as condições, as páginas servidas, a configuração dos balanceadores e o gerador de carga, e depois o protocolo de medição, as métricas e a coleta, a validação e a análise, a automação e as limitações. O desenho é um experimento fatorial: dois balanceadores, três algoritmos e dois tipos de página formam doze condições experimentais, cada uma medida em níveis de carga definidos como frações da sua própria capacidade. Os valores ainda não fixados, como durações e proporções, serão definidos em experimentos-piloto e registrados posteriormente.

3.1 Materiais

3.1.1 Hardware

Todos os experimentos serão executados em uma única máquina virtual dedicada, descrita no Quadro 2, hospedada no servidor de laboratório 78pve. Um segundo servidor, 30pve, fica como reserva, pois tem menos núcleos e serviços permanentes que não podem ser interrompidos. Usar uma máquina só é uma limitação declarada, tratada na última seção deste capítulo. A vantagem de uma máquina só é que as doze condições enxergam exatamente o mesmo *hardware* e a mesma rede, o que tira a diferença entre máquinas da lista de causas possíveis para uma diferença medida.

Quadro 2 – Máquina dos experimentos

Item	**Valor**
Hospedeiro	78pve, Proxmox VE, hipervisor QEMU/KVM
Processador do hospedeiro	Intel Xeon E5-2678 v3, 12 núcleos físicos, 24 *threads*, 2,5 GHz base
Memória do hospedeiro	125 GB ECC DDR4-2133
Armazenamento	NVMe
Máquina virtual	12 vCPU com `cpu=host`, afinidade aos núcleos físicos 0 a 11, irmãos SMT ociosos, memória a definir
Sistema operacional convidado	Ubuntu Server, versão e *kernel* a registrar
Swap	desativado
Governador de frequência	`performance` no hospedeiro, turbo desativado durante os experimentos

Fonte: Autoria própria (2026).

Durante os experimentos, a máquina virtual será a única ativa no hospedeiro. O *swap* ficará desativado, porque um servidor web não deve paginar, e a frequência do processador ficará fixa, pelo governador `performance` e pela desativação do turbo, para que não varie com a carga entre experimentos. Dentro da máquina virtual, cada papel receberá núcleos exclusivos,

e os núcleos lógicos irmãos (SMT) ficarão ociosos, para que dois papéis não compartilhem um núcleo físico.

3.1.2 Software

O Quadro 3 lista os componentes e o papel de cada um. A versão de cada componente será registrada na execução dos testes. Os servidores de destino, para os quais os dois balanceadores encaminharão as requisições, usarão Nginx na mesma versão do Nginx balanceador, para que a única diferença entre as condições esteja no balanceador e no algoritmo.

Quadro 3 – Componentes de software

Componente	**Papel**
Docker Engine	isolamento de gerador, balanceador e servidores de destino
Nginx	balanceador sob teste; servidor de destino para a página estática e *front* do PHP-FPM
Apache HTTP Server	balanceador sob teste, MPM *event*
PHP-FPM	página dinâmica
k6	gerador de carga
sysstat	CPU por núcleo da máquina
OpenTofu e Ansible	criação e configuração da máquina virtual e dos contêineres

Fonte: Autoria própria (2026).

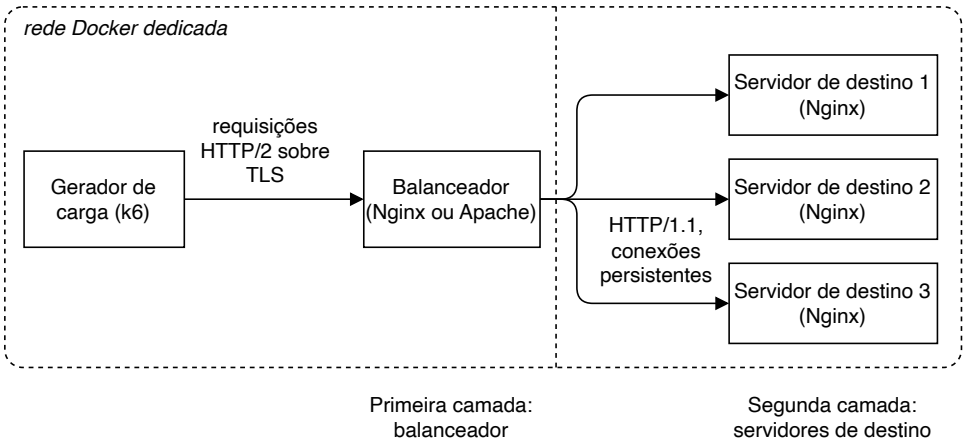
3.1.3 Ambiente experimental

O ambiente terá cinco contêineres em uma rede Docker dedicada: um gerador de carga, um balanceador e três servidores de destino, como na topologia de referência da documentação do Nginx (NGINX, 2026b). Três destinos são suficientes para que o desequilíbrio entre servidores apareça e, ao mesmo tempo, poucos o bastante para que sobrem núcleos para o gerador e o balanceador na máquina virtual de doze vCPU. A Figura 7 mostra a organização: as duas primeiras camadas de um *cluster* de servidores, sem a camada de dados. Todo o tráfego percorrerá a rede de contêineres dentro da máquina virtual, sem placa de rede física no caminho, de modo que a rede não limite o balanceador; o custo da rede de contêineres é o mesmo para os dois balanceadores e fica fora da comparação.

O gerador falará HTTP/2 sobre TLS com o balanceador, porque o HTTP/2 é a versão predominante no tráfego web atual, presente em cerca de 70% das páginas iniciais (MARX, 2024), envia várias requisições por uma única conexão TCP (THOMSON; BENFIELD, 2022) e só é negociado sobre TLS pelos navegadores e pelo k6; o balanceador decide o destino por requisição, e não por conexão. O balanceador falará HTTP/1.1 com conexões persistentes com os servidores de destino, porque o encaminhamento em HTTP/2 ao servidor de destino é recente no Nginx e experimental no Apache (NGINX, 2026a; APACHE SOFTWARE FOUNDA-

TION, 2026b). As conexões persistentes precisam ser iguais nos dois balanceadores: Pramono, Buwono e Waskito (2018) mediram ganho de cerca de 46% em requisições por segundo no Nginx só por ativar o *keep-alive*. A terminação TLS é paga igualmente pelos dois, com o mesmo certificado e a mesma suíte criptográfica.

Figura 7 – Organização do ambiente de experimentos



Fonte: Adaptada de Van Steen e Tanenbaum (2025).

Cada papel receberá núcleos de CPU exclusivos e memória limitada, sem sobreposição. Assim, o consumo de CPU medido nos núcleos do balanceador será do balanceador e do *kernel* a seu serviço, não do gerador nem dos servidores de destino. O gerador receberá mais núcleos que o balanceador, porque o k6 custa mais por requisição que o balanceador que ele testa, e os limites do sistema que Mosberger e Jin (1998) apontam como fontes de falsos gargalos no cliente ficarão elevados e registrados.

3.2 Métodos

3.2.1 Fatores, níveis e condições experimentais

Os fatores são o balanceador, o algoritmo e o tipo de página, com os níveis do Quadro 4. O cruzamento balanceador $\times$ algoritmo dá seis combinações; cada combinação será medida com os dois tipos de página, em doze condições experimentais. O nível de carga será definido por condição, como fração da capacidade daquela condição.

O nível 2/3 segue Cardellini, Colajanni e Yu (1999), que o adotam como carga média; o nível 1 é a capacidade; o nível 1.2 observa o comportamento além da capacidade, onde o modelo de fila deixa de valer e a saturação e as falhas são o observável esperado; o nível 1/3 dá o custo de base, sem fila. Os quatro níveis cobrem, em ordem, o balanceador folgado, em uso moderado, no limite e além dele, que são as situações em que se espera que os algoritmos se separem de formas diferentes.

Quadro 4 – Fatores e níveis

Fator	Níveis	Nginx	Apache
Balanceador	2	`ngx_http_upstream_-module`	`mod_proxy_balancer`
Algoritmo	3	*round-robin* (padrão), `least_conn`, `ip_hash`	`lbmethod=byrequests`, `lbmethod=bybusyness`, afinidade por rota derivada do endereço de origem
Tipo de página	2	estática, dinâmica	estática, dinâmica
Nível de carga	4 por condição	1/3, 2/3, 1 e 1.2 da capacidade	1/3, 2/3, 1 e 1.2 da capacidade

Fonte: Autoria própria (2026).

3.2.2 Páginas estáticas e dinâmicas

A página estática será um conjunto de três arquivos, de 1 KB, 10 KB e 100 KB, requisitados em proporção fixa. A faixa segue a caracterização de Arlitt e Williamson (1997), em que a maioria das transferências é pequena. Um único tamanho diria pouco; com três, o experimento inclui tanto respostas pequenas quanto respostas que levam muito mais tempo para transferir, como em um site real.

A página dinâmica será um *script* PHP que executa uma quantidade de trabalho de CPU definida por um parâmetro da URL, sem *cache* e sem banco de dados, no modelo do servidor de Freitas e Souza (2022). Três valores do parâmetro definirão três custos de serviço, requisitados em proporção fixa. O trabalho é determinístico por valor do parâmetro, o que mantém a reprodutibilidade; a variabilidade do tempo de serviço vem da mistura, que aproxima a carga web real e é a condição para a qual o *least-connection* foi concebido. Uma aplicação completa, como o WordPress, saturaria a máquina antes de o balanceador ser o gargalo. O objetivo é medir o balanceador, não a aplicação, e uma página sintética com custo controlado deixa claro quanto do tempo é do servidor de destino e quanto é do caminho até ele. Antes dos experimentos, será verificado que cada servidor de destino, isolado, responde ao mesmo valor do parâmetro no mesmo tempo.

3.2.3 Configuração dos balanceadores

Os dois balanceadores operarão com configuração padrão, exceto pelos itens do Quadro 5, que a literatura e a documentação apontam como fatores de confusão e que, por isso, serão fixados e igualados. Pariag *et al.* (2007) mostram que o ajuste fino altera os resultados de forma significativa; uma segunda série de experimentos com configurações ajustadas fica como trabalho futuro.

O *ip-hash* não existe no Apache, cuja afinidade é por *cookie* ou por rota. Para comparar os dois servidores sob o mesmo *ip-hash*, o trabalho adota como definição operacional a propri-

Quadro 5 – Parâmetros fixados nos balanceadores

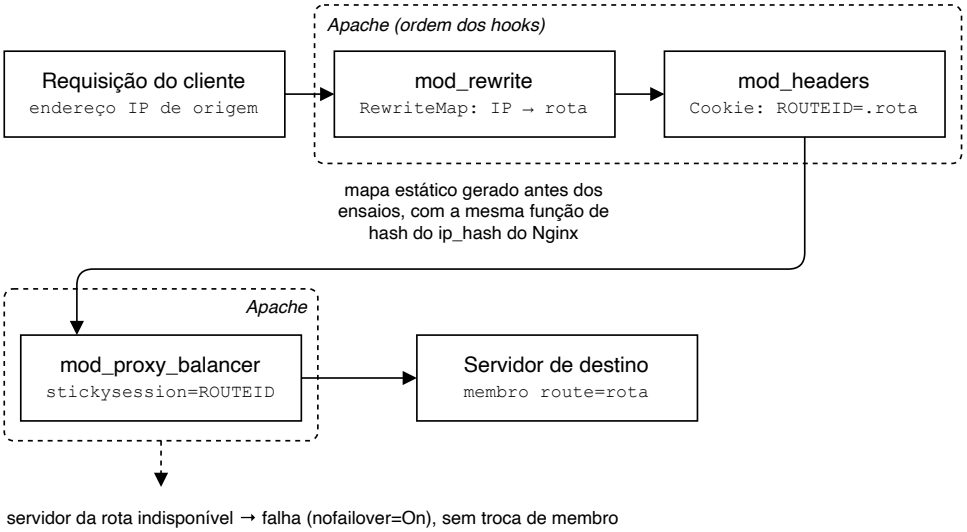
Item	Nginx	Apache	Motivo
Processos e *threads*	um processo de trabalho por núcleo	MPM *event* dimensionado para o número de núcleos	mesma proporção entre processos e núcleos
HTTP/2 e TLS	`http2 on`; mesmo certificado e suíte	`Protocols h2 http/1.1`; mesmo certificado e suíte	versão predominante na web (MARX, 2024); TLS igual nos dois
Estado do *least-connection*	`zone` compartilhada entre processos	*scoreboard* compartilhado, nativo	sem a zona, cada processo do Nginx conta sozinho (NGINX, 2026b)
Conexões persistentes com destino	`keepalive` e `proxy_http_version 1.1`	`keepalive=On`	Pramono, Buwono e Waskito (2018)
Repetição implícita	`proxy_next_upstream off`	`maxattempts=0` e `nofailover=On`	repetições mascaram falhas
Registro de acesso	formato com o servidor de destino escolhido	formato com o membro escolhido	validação funcional

Fonte: Autoria própria (2026).

edade essencial do algoritmo, mesma origem leva ao mesmo servidor, e a obtém no Apache pela afinidade por rota do *mod_proxy_balancer*: cada servidor de destino recebe um identificador de rota (`route`), e o balanceador encaminha a requisição ao membro cuja rota consta no *cookie* nomeado em `stickysession` (APACHE SOFTWARE FOUNDATION, 2026a). Como o cliente não envia esse *cookie*, o próprio Apache o injeta na requisição: um mapa estático do *mod_rewrite* (`RewriteMap`), gerado antes dos experimentos com a mesma função de *hash* do `ip_hash` do Nginx, converte o endereço de origem em rota, e o *mod_headers* grava o *cookie* de rota antes de o balanceador escolher o membro, na ordem de *hooks* da Figura 2. O mapa fica em memória após a primeira consulta, e `nofailover=On` faz a requisição falhar, em vez de ir a outro membro, se o servidor da rota estiver indisponível. Um piloto com três servidores de destino verificará o encaminhamento pela rota derivada do endereço, a troca de rota ao alterar o mapa e a falha, em vez da troca de membro, com `nofailover=On`; a Figura 8 resume o mecanismo. Assim, os dois balanceadores produzirão a mesma partição de clientes entre servidores de destino, e a comparação medirá o custo de obter a mesma afinidade em cada um. O desequilíbrio dessa partição será conhecido antes do experimento e reportado.

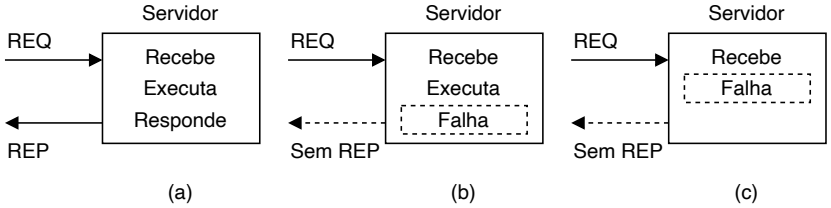
Além disso, repetições ficarão desativadas nos dois balanceadores (Quadro 5) porque o balanceador, como o cliente de uma chamada remota, não distingue um servidor que falhou antes de executar a requisição de um que falhou depois, Figura 9. Repetir pode duplicar o trabalho e esconde a falha que o experimento quer contar, de modo que cada requisição é tentada uma única vez.

Figura 8 – Afinidade por rota no Apache, do endereço de origem ao servidor de destino



Fonte: Autoria própria (2026).

Figura 9 – Caso normal (a), falha do servidor depois de executar a requisição (b) e antes de executá-la (c)



Fonte: Adaptada de Van Steen e Tanenbaum (2025).

3.2.4 Gerador de carga

O gerador precisa falar HTTP/2, gerar carga em malha aberta a taxa fixa, reportar a distribuição da latência e executar um roteiro por cliente para a página dinâmica. O k6, ferramenta de código aberto com roteiros em JavaScript, é a única, entre as avaliadas, que atende aos quatro requisitos ao mesmo tempo: o executor `constant-arrival-rate` inicia requisições a uma taxa fixa, independentemente da resposta do sistema sob teste, a ferramenta negocia HTTP/2 automaticamente sobre TLS e o resumo de fim de experimento reporta mediana, percentis e taxa de falhas (GRAFANA LABS, 2026a; GRAFANA LABS, 2026b; GRAFANA LABS, 2026c). As alternativas foram descartadas por não falarem HTTP/2, caso do httperf, do Apache Benchmark e do wrk, ou por não executarem roteiros por cliente, caso do h2load usado por Carraro (2025), mais barato por requisição que o k6, o que obriga a dimensionar o gerador acima do balanceador.

Pelo executor `constant-arrival-rate`, a taxa oferecida e a duração fixam o número de requisições, e o número de usuários virtuais ficará folgado para que a taxa seja mantida; a métrica `dropped_iterations` deverá ser zero em todo experimento aceito, o que

prova que a malha aberta foi mantida. A carga será constante durante cada experimento, com uma única taxa e uma única mistura de páginas do início ao fim.

O gerador usará vários endereços de origem, em redes distintas, pela opção `localIPs` (GRAFANA LABS, 2026d). É isso que permite medir o *ip-hash*: Tamanaha, Domingues e Bittencourt (2021) o mediram com um único endereço e observaram todo o tráfego em um servidor. O número de endereços será fixado nos experimentos-piloto, grande o bastante para que o desequilíbrio esperado do *hash* seja pequeno frente às diferenças que se quer medir. O *timeout* por requisição será de 5 s, contado como falha: o valor fica acima do tempo de serviço da página dinâmica mais cara e bem abaixo do padrão de 60 s do k6, que esconderia falhas, e será confirmado nos experimentos-piloto. As amostras brutas de cada requisição serão gravadas para o cálculo dos percentis.

3.2.5 Protocolo de medição

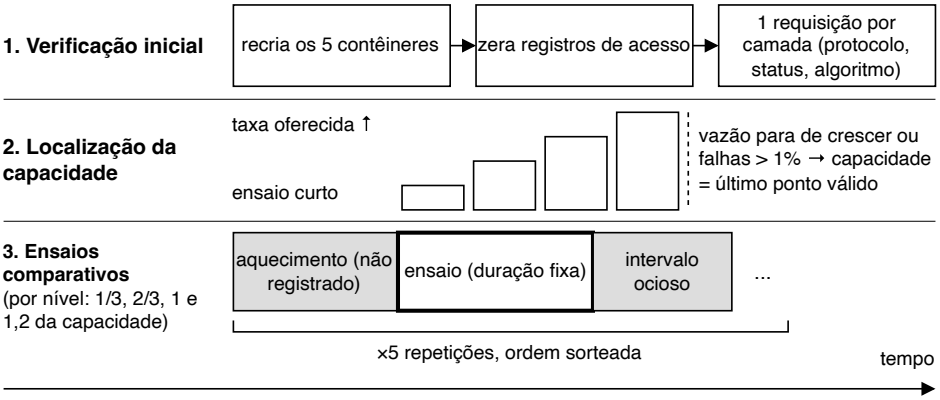
Cada condição experimental passará por 3 etapas (Figura 10): verificação inicial, localização da capacidade e experimentos comparativos. O procedimento será o mesmo para as doze condições e será executado por um orquestrador versionado no repositório. Na verificação inicial, os cinco contêineres serão recriados do zero e os registros de acesso zerados, e uma requisição a cada camada confirmará protocolo, código de *status* e algoritmo ativo antes de o experimento começar.

Na localização da capacidade, a taxa oferecida será elevada em passos, com um experimento curto por ponto, até que a vazão deixe de crescer ou as falhas ultrapassem 1%; o último ponto que atende aos dois critérios é a capacidade da condição e fixará os seus quatro níveis de carga. Localizar a capacidade antes de comparar, e comparar em frações dela, evita que uma combinação seja medida folgada e outra já saturada, caso em que a comparação diria mais sobre a carga escolhida do que sobre o algoritmo.

Nos experimentos comparativos, cada nível de carga terá cinco repetições de um experimento com a sequência: aquecimento à taxa do nível, não registrado, porque servidores web levam tempo para atingir regime estável; experimento com duração fixa; e intervalo ocioso, para esvaziar as conexões encerradas. Cinco repetições seguem trabalhos anteriores, e o número, justificado e reportado como pede a avaliação reproduzível, será elevado se os intervalos de confiança não se separarem (CARRARO, 2025; PAPADOPOULOS *et al.*, 2021). A ordem das condições e das repetições será sorteada, para que efeitos de aquecimento da máquina não se alinhem com um balanceador.

O teste de gargalo do gerador de Mosberger e Jin (1998) será feito uma vez por tipo de página, na condição de maior capacidade: a curva de vazão por taxa oferecida obtida com uma instância do k6 deve coincidir com a obtida com duas instâncias, cada uma com metade da taxa. Em todo experimento, a CPU dos núcleos do gerador deverá ficar abaixo de um limite fixado nos experimentos-piloto, e a taxa oferecida atingida será confrontada com a configurada.

Figura 10 – Protocolo de medição de uma condição experimental



Fonte: Autoria própria (2026).

3.2.6 Métricas e coleta

As métricas virão de três fontes: (i) do gerador, por experimento, vazão e taxa oferecida efetivamente atingida, latência de cada requisição, de que se calculam mediana e percentis 95 e 99 apenas sobre as respostas com sucesso, taxa de falhas e `dropped_iterations`; (ii) da máquina do balanceador, a cada segundo, pelo `mpstat` restrito aos seus núcleos, tempo de CPU em espaço de usuário, *kernel* e interrupções, somados como CPU do sistema inteiro, e, pelo *cgroup* do contêiner, memória residente máxima; e (iii) dos servidores de destino, ao fim do experimento, o número de requisições recebidas por servidor, contado no registro de acesso, a partir do qual se calcula o desequilíbrio entre os três. Essas fontes traduzem as métricas usuais de balanceamento em latência, vazão, consumo de CPU e memória da máquina do balanceador e distribuição efetiva das requisições entre os servidores de destino; o consumo do próprio balanceador é a dimensão que a literatura raramente mede (KUMAR; KUMAR, 2019). Requisição falha é a que termina em erro de conexão, *timeout* ou código de *status* de erro (KITHULWATTA *et al.*, 2022). O custo por requisição será o tempo de CPU acumulado no experimento dividido pelo número de respostas com sucesso. Todos os dados brutos, com os metadados de condição, nível, repetição, data, versões e revisão do repositório, serão gravados em arquivos nomeados pela condição, e versionados.

3.2.7 Validação e análise

Um experimento entrará na comparação só se a taxa oferecida atingida ficou dentro da tolerância da configurada e `dropped_iterations` foi zero; as falhas ficaram em até 1% nos níveis 1/3, 2/3 e 1, limiar que não se aplica ao nível 1.2, cujo objeto é a sobrecarga; a CPU do gerador ficou abaixo do limite; e a distribuição efetiva correspondeu ao algoritmo: contagens iguais no *round-robin*, contagens iguais à partição calculada no *ip-hash* e, no *least-connection*,

sem expectativa fixa, o desequilíbrio apenas reportado. Experimento rejeitado será repetido, e a rejeição ficará registrada.

Para cada condição, nível e métrica, serão reportados média, desvio-padrão e intervalo de confiança de 95% das repetições; intervalos que não se sobrepõem indicam diferença e, sobrepostos, a diferença não fica demonstrada. O intervalo é o que separa uma diferença real entre combinações da variação normal de uma repetição para outra: sem ele, uma diferença pequena poderia ser só ruído. Os resultados serão apresentados em três formas: curvas de capacidade, com vazão e percentil 99 da latência por taxa oferecida, um gráfico por tipo de página; quadros de eficiência, com custo por requisição, latência e taxa de falhas por combinação e tipo de página; e a matriz de resultado, que indica, para cada tipo de página, o algoritmo e o balanceador de melhor eficiência e o nível de carga em que a diferença se sustenta. Os resultados serão então confrontados com as expectativas do referencial e com os resultados dos trabalhos relacionados.

3.2.8 Automação e reprodutibilidade

A máquina virtual será criada por código: um modelo de nuvem Ubuntu será clonado pelo OpenTofu, que fixa vCPU, afinidade e memória, e o Ansible instalará o Docker, o sysstat e o k6, aplicará os limites do sistema e gerará as definições dos contêineres. A máquina virtual será destruída e recriada entre as campanhas de cada balanceador, o que reproduz, a custo baixo, a reinstalação completa que Lima, Pereira e Hernandes (2026) fazem entre configurações. Configurações dos balanceadores, *scripts* do k6, mapa de rotas do Apache, definição dos contêineres, código de provisionamento e *scripts* de análise serão versionados no repositório do trabalho, de modo que a reexecução completa produza os mesmos arquivos de dados.

3.2.9 Limitações

Gerador, balanceador e servidores de destino compartilharão a máquina, e a máquina é virtual, e não *hardware* nativo; Felter *et al.* (2015) medem o custo do KVM como pequeno para CPU e memória, mas registram sobrecarga em cada operação de entrada e saída, que torna o KVM menos adequado a cargas sensíveis à latência ou com alta taxa de E/S. O tráfego HTTP destes experimentos é uma carga desse tipo, ainda que não saia da máquina virtual, e esse custo, igual nas doze condições, entra nos valores absolutos e não na comparação. Limites de recursos na máquina cliente distorcem a medida do servidor, e Rampanelli e Broza (2019) registram que balanceador e servidores reais não devem compartilhar a máquina. A mitigação é a separação de núcleos, memória e rede entre os papéis e o teste de gargalo do gerador; ainda assim, *cache* de último nível, barramento de memória e *kernel* são compartilhados, e os valores absolutos valem para esta máquina. O que o trabalho reivindica é a comparação entre

condições, feitas sob o mesmo *hardware*, a mesma rede e a mesma carga, e não os valores absolutos. O *ip-hash* do Apache é uma emulação: o mapa de reescrita e a injeção do *cookie* de rota acrescentam trabalho por requisição que o `ip_hash` do Nginx não faz, e esse custo fica dentro das condições *ip-hash* do Apache, como parte do custo de obter a mesma afinidade. A configuração é a padrão, e o ajuste fino fica como trabalho futuro.

4 CONSIDERAÇÕES FINAIS

REFERÊNCIAS

APACHE SOFTWARE FOUNDATION. **Apache module mod_proxy_balancer**: Apache HTTP Server version 2.4 documentation. 2026. Disponível em: https://httpd.apache.org/docs/2.4/mod/mod_proxy_balancer.html. Acesso em: 17 set. 2026.

APACHE SOFTWARE FOUNDATION. **Apache module mod_proxy_http2**: Apache HTTP Server version 2.4 documentation. 2026. Disponível em: https://httpd.apache.org/docs/2.4/mod/mod_proxy_http2.html. Acesso em: 10 out. 2026.

APACHE SOFTWARE FOUNDATION. **Apache MPM event**: Apache HTTP Server version 2.4 documentation. 2026. Disponível em: https://httpd.apache.org/docs/2.4/mod/event.html. Acesso em: 10 out. 2026.

ARLITT, M. F.; WILLIAMSON, C. L. Internet web servers: workload characterization and performance implications. **IEEE/ACM Transactions on Networking**, v. 5, n. 5, p. 631–645, out. 1997.

BANGA, G.; DRUSCHEL, P. Measuring the capacity of a web server. *In*: USENIX SYMPOSIUM ON INTERNET TECHNOLOGIES AND SYSTEMS (USITS), 1., 1997, Monterey. **Proceedings [...]**. Berkeley: USENIX Association, 1997.

CARDELLINI, V. *et al.* The state of the art in locally distributed web-server systems. **ACM Computing Surveys**, v. 34, n. 2, p. 263–311, jun. 2002.

CARDELLINI, V.; COLAJANNI, M.; YU, P. S. Dynamic load balancing on Web-server systems. **IEEE Internet Computing**, v. 3, n. 3, p. 28–39, 1999. DOI: 10.1109/4236.769420.

CARRARO, P. A. **Um estudo comparativo de desempenho entre as diferentes versões do protocolo HTTP**. 2025. Trabalho de Conclusão de Curso (Tecnologia em Sistemas para Internet) — Universidade Tecnológica Federal do Paraná, Guarapuava, 2025. Orientadora: Sediane Carmem Lunardi Hernandes. Coorientador: Hermano Pereira.

FELTER, W. *et al.* An updated performance comparison of virtual machines and Linux containers. *In*: IEEE INTERNATIONAL SYMPOSIUM ON PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE (ISPASS), 2015, Philadelphia. **Proceedings [...]**. Piscataway: IEEE, 2015. p. 171–172.

FREITAS, M. H.; SOUZA, V. B. Statera: um balanceador de carga rápido e flexível para aplicações HTTP na nuvem. *In*: SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO (SSCAD), 23., 2022, Florianópolis. **Anais [...]**. Porto Alegre: Sociedade Brasileira de Computação, 2022. p. 241–252.

GRAFANA LABS. **Constant arrival rate: k6 documentation**. 2026. Disponível em: https://grafana.com/docs/k6/latest/using-k6/scenarios/executors/constant-arrival-rate/. Acesso em: 10 out. 2026.

GRAFANA LABS. **End of test summary: k6 documentation**. 2026. Disponível em: https://grafana.com/docs/k6/latest/results-output/end-of-test/. Acesso em: 10 out. 2026.

GRAFANA LABS. **HTTP/2: k6 documentation**. 2026. Disponível em: https://grafana.com/docs/k6/latest/using-k6/protocols/http-2/. Acesso em: 10 out. 2026.

GRAFANA LABS. **Options reference: k6 documentation**. 2026. Disponível em: https://grafana.com/docs/k6/latest/using-k6/k6-options/reference/. Acesso em: 10 out. 2026.

HEINZL, S.; METZ, C. Toward a cloud-ready dynamic load balancer based on the Apache web server. *In*: WORKSHOPS ON ENABLING TECHNOLOGIES: INFRASTRUCTURE FOR COLLABORATIVE ENTERPRISES (WETICE), 22., 2013, Hammamet. **Proceedings [...]**. Piscataway: IEEE, 2013. p. 342–345. DOI: 10.1109/WETICE.2013.63.

HOSSEINI, S. M.; JAHANGIR, A. H.; DARABY, S. Session-persistent load balancing for clustered web servers without acting as a reverse-proxy. *In*: INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT (CNSM), 17., 2021. **Proceedings [...]**. [S. l.]: IFIP, 2021. p. 360–364. DOI: 10.23919/CNSM52442.2021.9615592.

JADER, O. H. *et al.* Ultra-dense request impact on cluster-based web server performance. *In*: INTERNATIONAL IRAQI CONFERENCE ON ENGINEERING TECHNOLOGY AND THEIR APPLICATIONS (IICETA), 4., 2021, Najaf. **Proceedings [...]**. Piscataway: IEEE, 2021. p. 252–257. DOI: 10.1109/IICETA51758.2021.9717748. Citado por Pinastawa, Pradana e Maulana (2024); texto completo a obter.

KARGER, D. *et al.* Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. *In*: ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING (STOC), 29., 1997, El Paso. **Proceedings [...]**. New York: ACM, 1997. p. 654–663. DOI: 10.1145/258533.258660.

KITHULWATTA, W. M. C. J. T. *et al.* Performance evaluation of Docker-based Apache and Nginx web server. *In*: INTERNATIONAL CONFERENCE FOR EMERGING TECHNOLOGY (INCET), 3., 2022, Belgaum. **Proceedings [...]**. Piscataway: IEEE, 2022. p. 1–6. DOI: 10.1109/INCET54531.2022.9824303.

KUMAR, P.; KUMAR, R. Issues and challenges of load balancing techniques in cloud computing: a survey. **ACM Computing Surveys**, v. 51, n. 6, p. 1–35, fev. 2019. Art. 120. DOI: 10.1145/3281010.

LIMA, F. B.; PEREIRA, H.; HERNANDES, S. C. L. Análise de desempenho de distribuições Kubernetes lightweight em hardware de baixo custo para processamento distribuído. *In*: WORKSHOP EM DESEMPENHO DE SISTEMAS COMPUTACIONAIS E DE COMUNICAÇÃO (WPerformance), 25., 2026, Gramado. **Anais [...]**. Porto Alegre: Sociedade Brasileira de Computação, 2026. p. 24–33.

MA, C.; CHI, Y. Evaluation test and improvement of load balancing algorithms of Nginx. **IEEE Access**, v. 10, p. 14311–14324, 2022. DOI: 10.1109/ACCESS.2022.3146422.

MARX, R. **HTTP**. 2024. In: HTTP Archive. Web Almanac 2024, cap. 18. Disponível em: https://almanac.httparchive.org/en/2024/http. Acesso em: 18 out. 2026.

MORAIS, N. B. *et al.* Avaliação de algoritmos de balanceamento de carga para ambientes em nuvem. *In*: SIMPÓSIO BRASILEIRO DE SISTEMAS DE INFORMAÇÃO (SBSI), 13., 2017, Lavras. **Anais [...]**. Porto Alegre: Sociedade Brasileira de Computação, 2017. p. 356–363.

MOSBERGER, D.; JIN, T. httperf: a tool for measuring web server performance. **ACM SIGMETRICS Performance Evaluation Review**, v. 26, n. 3, p. 31–37, dez. 1998. DOI: 10.1145/306225.306235.

NGINX. **Module ngx_http_proxy_module**. 2026. Disponível em: https://nginx.org/en/docs/http/ngx_http_proxy_module.html. Acesso em: 10 out. 2026.

NGINX. **Using nginx as HTTP load balancer**. 2026. Disponível em: https://nginx.org/en/docs/http/load_balancing.html. Acesso em: 17 set. 2026.

PAPADOPOULOS, A. V. *et al.* Methodological principles for reproducible performance evaluation in cloud computing. **IEEE Transactions on Software Engineering**, v. 47, n. 8, p. 1528–1543, ago. 2021.

PARIAG, D. *et al.* Comparing the performance of web server architectures. *In*: ACM SIGOPS/EUROSYS EUROPEAN CONFERENCE ON COMPUTER SYSTEMS (EuroSys), 2., 2007, Lisboa. **Proceedings [...]**. New York: ACM, 2007. p. 231–243. DOI: 10.1145/1272996.1273021.

PINASTAWA, I. W. R.; PRADANA, M. G.; MAULANA, N. Web server performance evaluation: comparing Nginx and Apache using K6 testing methods for load, spike, soak, and performance. *In*: INTERNATIONAL CONFERENCE ON INFORMATICS, MULTIMEDIA, CYBER AND INFORMATION SYSTEM (ICIMCIS), 2024. **Proceedings [...]**. Piscataway: IEEE, 2024. p. 1002–1007. DOI: 10.1109/ICIMCIS63449.2024.10956402.

PRAMONO, L. H.; BUWONO, R. C.; WASKITO, Y. G. Round-robin algorithm in HAProxy and Nginx load balancing performance evaluation: a review. *In*: INTERNATIONAL SEMINAR ON RESEARCH OF INFORMATION TECHNOLOGY AND INTELLIGENT SYSTEMS (ISRITI), 2018, Yogyakarta. **Proceedings [...]**. Piscataway: IEEE, 2018. p. 367–372. DOI: 10.1109/ISRITI.2018.8864455.

PUTRO, Z. P.; SUPONO, R. A. Comparison analysis of Apache and Nginx webserver load balancing on Proxmox VE in supporting server performance. **International Research Journal of Advanced Engineering and Science**, v. 7, n. 3, p. 144–151, 2022. Disponível em: https://irjaes.com/wp-content/uploads/2022/08/IRJAES-V7N3P161Y22.pdf. Acesso em: 18 out. 2026.

RAMPANELLI, R. L.; BROZA, S. **Cluster de desempenho para mecanismos de segurança em redes**. 2019. 62 p. Trabalho de Conclusão de Curso (Tecnologia em Sistemas de Telecomunicações) — Universidade Tecnológica Federal do Paraná, Curitiba, 2019. Disponível em: https://repositorio.utfpr.edu.br/jspui/. Acesso em: 10 out. 2026.

TAMANAHA, L. K. G.; DOMINGUES, G. L.; BITTENCOURT, L. F. **Algoritmos para balanceamento de carga no NGINX**. Campinas: Instituto de Computação, Universidade Estadual de Campinas, 2021. 17 p. (Relatório Técnico IC-PFG-21-49). Projeto Final de Graduação. Disponível em: https://www.ic.unicamp.br/~reltech/PFG/2021/PFG-21-49.pdf. Acesso em: 17 set. 2026.

THOMSON, M.; BENFIELD, C. **HTTP/2**. Internet Engineering Task Force, 2022. RFC 9113. DOI: 10.17487/RFC9113. Disponível em: https://www.rfc-editor.org/rfc/rfc9113. Acesso em: 10 out. 2026.

VAN STEEN, M.; TANENBAUM, A. S. **Distributed systems**. 4. ed. [S. l.]: Maarten van Steen, 2025. Versão 4.03. ISBN 978-90-815406-4-3. Disponível em: https://www.distributed-systems.net/. Acesso em: 18 out. 2026.

WELSH, M.; CULLER, D.; BREWER, E. SEDA: an architecture for well-conditioned, scalable internet services. *In*: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP), 18., 2001, Banff. **Proceedings [...]**. New York: ACM, 2001. p. 230–243.