UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

Bruno Olimpio dos Santos

Análise de Algoritmos de Balanceamento de Carga em Cenários com Hosts Desbalanceados

Recife

Agosto de 2025

Bruno Olimpio dos Santos

Análise de Algoritmos de Balanceamento de Carga em Cenários com Hosts Desbalanceados

Artigo apresentado ao Curso de Bacharelado em Sistemas de Informação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Universidade Federal Rural de Pernambuco – UFRPE
Departamento de Estatística e Informática
Curso de Bacharelado em Sistemas de Informação

Orientador: Lidiano Augusto Nóbrega de Oliveira

Recife
Agosto de 2025

Bruno Olimpio dos Santos

Análise de Algoritmos de Balanceamento de Carga em Cenários com Hosts Desbalanceados

Artigo apresentado ao Curso de Bacharelado em Sistemas de Informação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Aprovada em: 07 de agosto de 2025.

BANCA EXAMINADORA

Lidiano Augusto Nóbrega de Oliveira (Orientador)
Departamento de Estatistica e Informática
Universidade Federal Rural de Pernambuco

Professor Cleviton Vinicius Fonseca Monteiro
Departamento de Estatística e Informática
Universidade Federal Rural de Pernambuco

Dados Internacionais de Catalogação na Publicação
Sistema Integrado de Bibliotecas da UFRPE
Gerada automaticamente, mediante os dados fornecidos
pelo(a) autor(a)

S237a Santos, Bruno Olimpio dos.
Análise de algoritmos de balanceamento de carga em cenários com Hosts desbalanceados / Bruno Olimpio dos Santos. – Recife, 2025.
30 f.; il.

Orientador(a): Lidiano Augusto Nóbrega de Oliveira.

Trabalho de Conclusão de Curso (Graduação) – Universidade Federal Rural de Pernambuco, Bacharelado em Sistemas da Informação, Recife, BR-PE, 2026.

Inclui referências.

1. Balanceamento de carga. 2. Sistemas distribuídos. 3. Teste de carga. 4. Python 5. Nginx. I. Oliveira, Lidiano Augusto Nóbrega de, orient. II. Título

CDD 004

Análise de Algoritmos de Balanceamento de Carga em Cenários com Hosts Desbalanceados

Bruno Olimpio dos Santos [1], **Lidiano Augusto Nóbrega de Oliveira** [1]

[1]Departamento de Estatística e Informática – Universidade Federal Rural de Pernambuco
Rua Dom Manuel de Medeiros, s/n, - CEP: 52171-900 – Recife – PE – Brasil

bruno.olimpio@ufrpe.br, lidiano.oliveira@ufrpe.br

Resumo. *O crescimento exponencial de dispositivos conectados à Internet exige soluções que assegurem desempenho e disponibilidade em sistemas distribuídos. Este trabalho investiga o impacto de diferentes algoritmos de balanceamento de carga, tais como Round Robin, Weighted e Least Connections, em ambientes compostos por dispositivos com recursos computacionais distintos. Para isso, foi projetada uma arquitetura composta por três servidores executando aplicações Flask com Gunicorn, e um balanceador de carga Nginx instalado em um quarto dispositivo. Foram conduzidos experimentos com cenários variados quanto à capacidade dos servidores e à carga computacional das requisições, a fim de medir o tempo de resposta e analisar a distribuição das requisições. Os resultados evidenciam que algoritmos como o Least Connections e o Ponderado apresentam melhor adaptação a cenários assimétricos, promovendo distribuição mais eficiente das requisições e tempos de resposta mais estáveis. O estudo reforça a importância da escolha criteriosa do algoritmo de balanceamento conforme a configuração da infraestrutura e o perfil das requisições.*

Abstract. *The exponential growth of Internet-connected devices demands solutions that ensure performance and availability in distributed systems. This study investigates the impact of different load balancing algorithms, such as Round Robin, Weighted and Least Connections, in environments composed of devices with varying computational resources. To this end, an architecture was designed consisting of three servers running Flask applications with Gunicorn, and a Nginx load balancer installed on a fourth device. Experiments were conducted using scenarios with different server capacities and computational load of requests, in order to measure response time and analyze request distribution. The results show that algorithms such as Least Connections and Weighted adapt better to asymmetric scenarios, promoting more efficient request distribution and more stable response times. The study reinforces the importance of carefully choosing the load balancing algorithm according to the infrastructure setup and request profile.*

1. Introdução

Em 2024, cerca de 5,5 bilhões de pessoas estão conectadas à Internet, o que representa 68% da população mundial. Esse número marca um avanço em relação ao ano anterior, quando a taxa de conectividade era de 65%, evidenciando não apenas a ampliação do

acesso, mas também um crescimento acelerado. A taxa anual de crescimento passou de 2,7% para 3,4%[ITU 2024].

Esse crescimento envolve tanto número de usuários quanto de dispositivos conectados à internet, favorecidos pela popularização de dispositivos móveis, serviços em nuvem e aplicações em tempo real. Isso faz com que grandes provedores de serviços dependam de infraestruturas computacionais cada vez mais robustas, capazes de lidar com volumes elevados de requisições simultâneas. Sendo assim, tornam-se indispensáveis técnicas que garantam desempenho, escalabilidade e disponibilidade dos sistemas, sendo o balanceamento de carga uma das estratégias mais relevantes.

O balanceador de carga é um componente que distribui o tráfego de rede ou requisições entre múltiplos servidores, com o objetivo de otimizar o uso dos recursos disponíveis. Trata-se, portanto, de um elemento essencial da infraestrutura de rede moderna e dos sistemas computacionais, sendo fundamental para a distribuição eficiente de recursos entre um grande número de sistemas e usuários finais. Sua principal função é distribuir a carga de trabalho entre os recursos disponíveis de maneira a otimizar seu uso, minimizar o tempo de resposta e reduzir a sobrecarga dos servidores [Islam 2017].

Com a crescente popularização da computação em nuvem, o balanceamento de carga ganha ainda mais relevância. Provedores como AWS, Azure e Google Cloud oferecem serviços em nuvem, permitindo que usuários finais acessem aplicações sob demanda sem a necessidade de instalação local. Entretanto, essa arquitetura distribuída traz desafios, sendo o balanceamento de carga um dos mais importantes, visto que é responsável por garantir a distribuição equitativa das cargas entre máquinas, clusters, servidores e outros recursos. O objetivo é maximizar a capacidade de trabalho do sistema, utilizar eficientemente os recursos e garantir o desempenho dos dispositivos [Shahid et al. 2020]. Outro aspecto relevante é que as técnicas de balanceamento de carga buscam mitigar problemas relacionados ao desequilíbrio de carga, que podem resultar em dois cenários indesejáveis: a sobrecarga de alguns recursos e a subutilização de outros. Essas situações de desequilíbrio afetam diretamente o desempenho das aplicações e podem comprometer a experiência do usuário final [Afzal and Kavitha 2019].

Diversas soluções e ferramentas têm sido empregadas para implementar estratégias de balanceamento de carga, tanto em ambientes locais quanto em infraestruturas em nuvem. Historicamente, uma das primeiras abordagens adotadas foi a modificação do sistema de nomes de domínio (DNS) para associar múltiplos endereços IP a um único nome de *host*. Essa técnica permitia distribuir requisições entre diferentes servidores físicos de forma simples, porém limitada: não era determinística, apresentava falhas diante de picos de requisições e não respondia bem à indisponibilidade de servidores [Shanmugam and Iyenger 2016]. Com o avanço da demanda por escalabilidade, disponibilidade e desempenho, tornaram-se necessárias abordagens mais sofisticadas e robustas.

Atualmente, as soluções de balanceamento podem ser baseadas em *hardware*, como *appliances* dedicados, ou em *software*, como proxies reversos e serviços gerenciados por provedores de nuvem. Entre as abordagens mais comuns de balanceamento de carga via software estão ferramentas como Nginx, HAProxy e Envoy, bem como soluções nativas das plataformas em nuvem, como o Elastic Load Balancer (ELB) da AWS. A escolha da ferramenta adequada depende de fatores como o tipo de aplicação, a arquitetura

do sistema e os requisitos de escalabilidade e resiliência.

Em cenários com recursos desbalanceados, servidores menos potentes podem ser sobrecarregados, enquanto outros permanecem subutilizados, resultando em tempos de resposta elevados, gargalos de desempenho e, em casos mais críticos, falhas no atendimento das requisições. Quando os recursos computacionais são heterogêneos, como em ambientes com diferentes modelos de máquinas físicas ou virtuais, esse problema se intensifica. Dessa forma, compreender e mitigar o desbalanceamento é fundamental para garantir o uso eficiente dos recursos, a estabilidade da aplicação e a experiência do usuário final.

No caso específico do Nginx, amplamente utilizado como balanceador de carga reverso, diversos algoritmos podem ser configurados para lidar com a distribuição de requisições entre *hosts*. Entretanto, quando os *hosts* apresentam capacidades desiguais — como diferenças no poder de processamento, quantidade de memória ou largura de banda —, algoritmos tradicionais como o *Round-Robin* podem não ser eficazes. Nesses cenários, a adoção de algoritmos adaptativos torna-se crucial para garantir uma distribuição justa e eficiente da carga, otimizando o desempenho do sistema como um todo.

A avaliação do comportamento dos algoritmos de balanceamento, por testes de carga, permite identificar problemas relacionados ao desempenho, e é útil para compreender como cada algoritmo responde a diferentes cenários, especialmente quando há *hosts* desbalanceados [Jiang and Hassan 2015].

Este artigo tem como objetivo analisar o comportamento de diferentes algoritmos de balanceamento de carga oferecidos pelo Nginx em cenários com *hosts* desbalanceados, investigando sua eficácia e impacto no desempenho das aplicações.

1.1. Contexto

O aumento da demanda por aplicações web, aliado à necessidade de alta disponibilidade e escalabilidade, tornou o balanceamento de carga um elemento essencial em arquiteturas distribuídas. Soluções como o Nginx são amplamente utilizadas para distribuir requisições entre diversos servidores, com o intuito de otimizar o desempenho do sistema e garantir uma experiência satisfatória ao usuário final.

Entretanto, em ambientes reais, os servidores envolvidos no balanceamento nem sempre possuem capacidades homogêneas. Fatores como diferenças de *hardware*, limitações de rede ou sobrecargas momentâneas podem resultar em cenários desbalanceados, nos quais certos servidores ficam sobrecarregados enquanto outros permanecem ociosos. Esse desequilíbrio compromete diretamente a eficiência do sistema e evidencia a necessidade de algoritmos de balanceamento mais inteligentes.

Assim, torna-se relevante analisar como diferentes algoritmos de balanceamento de carga se comportam diante da heterogeneidade dos *hosts*. Essa investigação é particularmente importante em aplicações onde a latência e o uso eficiente de recursos têm impacto direto na qualidade do serviço. Ao compreender as limitações dos algoritmos tradicionais e comparar seu desempenho em condições realistas, é possível direcionar melhores escolhas de configuração e contribuir para o aprimoramento das soluções de infraestrutura computacional.

1.2. Motivação

O presente estudo é motivado pela crescente demanda por sistemas web resilientes e eficientes, especialmente em ambientes onde a capacidade de resposta dos servidores pode variar ao longo do tempo. Em muitos casos, o desbalanceamento entre os *hosts* é causado por fatores dinâmicos durante a execução da aplicação, como sobrecarga temporária de CPU, execução de tarefas concorrentes, ou limitações momentâneas de rede. A escolha do algoritmo de balanceamento de carga torna-se ainda mais crítica, pois pode influenciar diretamente a distribuição eficiente das requisições e a estabilidade do sistema como um todo.

O uso do Nginx como balanceador de carga é amplamente difundido devido à sua leveza e flexibilidade. No entanto, os diferentes algoritmos de balanceamento implementados pelo Nginx — como *Round-Robin, Weighted e Least Connections* — podem apresentar comportamentos distintos quando aplicados a ambientes desbalanceados. Este trabalho parte do pressuposto de que a escolha adequada da estratégia de balanceamento pode mitigar gargalos, melhorar a distribuição de carga e otimizar o uso dos recursos computacionais disponíveis, especialmente em situações com variação significativa no uso de CPU.

1.3. Objetivos

Este trabalho tem como objetivo principal analisar o desempenho de diferentes algoritmos de balanceamento de carga do Nginx em cenários compostos por *hosts* com capacidades desbalanceadas. A abordagem proposta visa identificar quais estratégias são mais adequadas de acordo com a natureza das requisições processadas pela aplicação, considerando tanto cenários com respostas imediatas quanto aqueles que exigem maior processamento.

Para atender o objetivo principal, os seguintes objetivos específicos serão elencados:

- Implementar um *cluster* de aplicação web para simular o ambiente com *hosts* desbalanceados.
- Avaliar o desempenho de diferentes algoritmos de balanceamento de carga — *Round Robin*, *Weighted* e *Least Connections* — em cenários com distribuição desigual de recursos computacionais.
- Comparar os algoritmos nos cenários de requisições leves (resposta imediata) e pesadas (uso intenso de CPU), utilizando métricas tais como tempo de resposta e distribuição de carga.
- Fornecer recomendações baseadas em evidências experimentais para a escolha do algoritmo mais adequado em ambientes heterogêneos.

O restante do artigo está organizado da seguinte forma. A Seção 2 apresenta o referencial teórico, abordando os conceitos fundamentais sobre balanceamento de carga e os principais algoritmos aplicáveis. A Seção 3 discute os trabalhos relacionados, destacando contribuições relevantes da literatura. A Seção 4 descreve a abordagem proposta, detalhando os objetivos dos experimentos e os cenários considerados. A Seção 5 apresenta os materiais e métodos utilizados, incluindo a configuração do ambiente, ferramentas adotadas e a metodologia de execução dos testes. Por fim, a Seção 6 discute os resultados obtidos e analisa o desempenho dos algoritmos nos diferentes contextos avaliados.

2. Referencial teórico

Nesta seção serão abordados os principais conceitos técnicos com o propósito de apoiar a compreensão do presente artigo.

2.1. Balanceamento de Carga

Balanceamento de carga é um termo genérico usado para descrever a distribuição de uma carga de processamento maior entre nós de processamento menores, com o objetivo de melhorar o desempenho geral do sistema. Em um ambiente de sistema distribuído, trata-se do processo de distribuir a carga entre diversos nós do sistema distribuído para melhorar tanto a utilização dos recursos quanto o tempo de resposta das tarefas. Um algoritmo de balanceamento de carga ideal deve evitar a sobrecarga ou subutilização de qualquer nó específico [Mondal et al. 2016].

De forma complementar, pode-se entender o balanceamento de carga como a técnica de distribuir a carga de trabalho entre os recursos disponíveis, de modo a obter uma utilização ideal dos recursos, minimizar o tempo de resposta ou reduzir a sobrecarga no sistema [Islam 2017]. O balanceador de carga é o componente responsável por decidir como os trabalhos serão atribuídos a cada recurso, utilizando para isso algum tipo de métrica ou algoritmo de escalonamento.

Em aplicações web, "carga" refere-se ao volume de requisições recebidas pelo sistema, que podem envolver tarefas como processamento de dados, execução de lógica de negócio ou consultas a banco de dados. Essa carga pode variar ao longo do tempo e, se mal distribuída, pode causar lentidão ou até falhas na aplicação. O balanceamento de carga atua justamente para mitigar esses efeitos, promovendo um fluxo de trabalho mais eficiente e estável.

2.2. Algoritmos de balanceamento de carga

Os algoritmos de balanceamento de carga, em geral, podem ser divididos em duas categorias: algoritmos de balanceamento de carga estáticos e dinâmicos. Um algoritmo de balanceamento de carga é denominado estático quando ignoram o estado dos nós, enquanto os dinâmicos consideram esse estado na distribuição da carga. A vantagem de se usar um algoritmo dinâmico é que, ao definir o destino final das requisições em tempo de execução, se algum nó falhar, isso não interromperá o sistema; afetará apenas o desempenho do sistema, uma vez que o *host* de menor capacidade continuará recebendo proporcionalmente menos requisições[Fatima et al. 2019].

Os três algoritmos a seguir representam abordagens distintas de balanceamento de carga e foram utilizados neste trabalho para avaliar seus desempenhos em diferentes cenários práticos:

2.2.1. Round Robin

É um algoritmo estático que distribui as requisições de maneira sequencial entre os servidores disponíveis, como se fosse uma fila circular. Cada nova requisição é encaminhada para o próximo servidor na lista, independentemente do volume de requisições ou capacidade de cada *host*. Embora simples, esse método pode gerar sobrecarga em servidores menos potentes quando os nós do sistema são heterogêneos [NGINX, Inc. 2025].

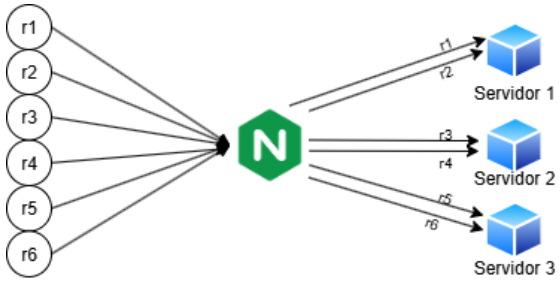

Figura 1. Distribuição de requisições em modo Round Robin.

2.2.2. Weighted

É uma variação do *Round Robin* que atribui pesos distintos aos servidores. Servidores com maior capacidade computacional recebem um número proporcionalmente maior de requisições. Essa estratégia distribui a carga de maneira mais equilibrada, respeitando os limites de cada servidor, e é especialmente recomendada em ambientes com *hardware* heterogêneo [NGINX, Inc. 2025].

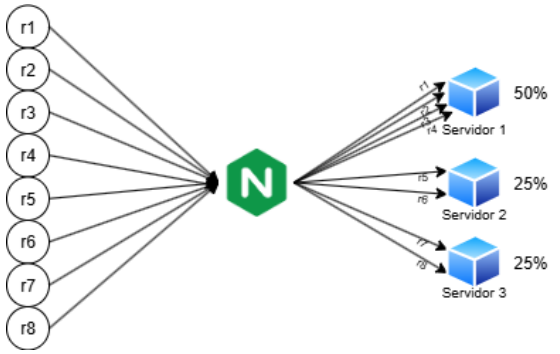

Figura 2. Distribuição de requisições em modo Weighted.

2.2.3. Least Connections

É um algoritmo dinâmico que encaminha a próxima requisição para o servidor com o menor número de conexões ativas no momento. Essa abordagem se adapta ao comportamento do sistema em tempo real e é eficaz em cenários onde o tempo de resposta das requisições varia, como aplicações que realizam operações mais intensas em CPU ou I/O [NGINX, Inc. 2025]. Nesse algorítmo, o Nginx mantém o registro de quantas conexões estão abertas entre o balanceador de carga e os *hosts*. Através da consulta periódica desse dados é feita a escolha do host que receberá a requisição, sendo escolhido aquele com menor número de requisições abertas ao balanceador no dado momento.

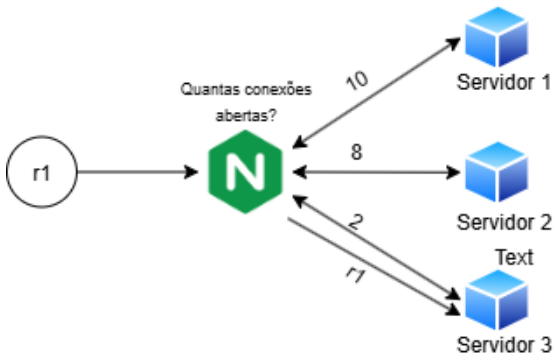

Figura 3. Distribuição de requisições em modo Least Connections.

2.3. Balanceadores de Carga e suas Implementações

Um balanceador de carga opera na camada de aplicação (L7) ou na camada de transporte (L4) do modelo OSI, interceptando as requisições dos clientes e distribuindo-as entre os servidores de *backend* conforme critérios definidos. Ao atuar como ponto único de entrada (*single point of access*), o balanceador de carga abstrai a complexidade da infraestrutura subjacente, promovendo escalabilidade horizontal e maior tolerância a falhas.

Entre os softwares mais utilizados para implementação de balanceamento de carga destaca-se o Nginx, que atua como um balanceador de carga reverso e permite a configuração de diversos algoritmos como *Round Robin, Weighted e Least Connections*. Outra ferramenta bastante utilizada é o HAProxy, conhecido por sua alta performance e capacidade de lidar com milhares de conexões simultâneas, sendo amplamente adotado em ambientes corporativos. Além disso, o Traefik tem ganhado popularidade em arquiteturas baseadas em microsserviços e containers, por oferecer integração nativa com orquestradores como Docker e Kubernetes. A escolha da ferramenta depende do contexto da aplicação, requisitos técnicos e facilidade de integração com os demais componentes da infraestrutura.

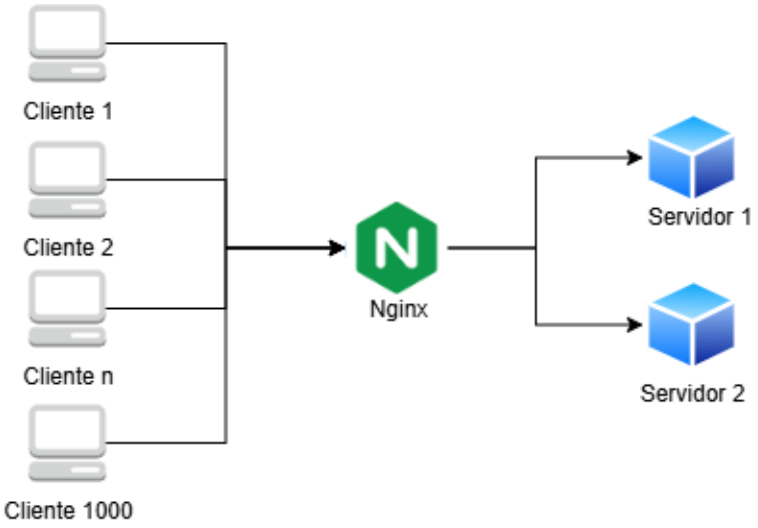

Figura 4. Posicionamento do Load Balancer.

2.4. Teste de carga

Ferramentas de balanceamento de carga são frequentemente avaliadas por meio de experimentos que medem características como tempo de resposta, uso de CPU, consumo de memória e distribuição de requisições entre os nós, considerando diferentes cenários de carga e configurações [Beyer et al. 2019]. O teste de carga permite comparar diferentes soluções de balanceamento de carga, bem como avaliar variações de algoritmos (por exemplo, *Round Robin, Weighted e Least Connections*) dentro de uma mesma ferramenta. Essa metodologia também é empregada para verificar o impacto de mudanças na infraestrutura ou nas políticas de roteamento de requisições. O teste consiste no envio de múltiplas requisições simultâneas aos servidores, simulando acessos concorrentes, com o objetivo de observar o comportamento do sistema sob diferentes níveis de estresse.

3. Trabalhos Relacionados

Diversas abordagens de balanceamento de carga vêm sendo propostas na literatura com o objetivo de otimizar a distribuição de requisições em ambientes distribuídos. O estudo experimental conduzido por [Shahid et al. 2020] compara os algoritmos *Round Robin, Random, Shortest Queue* e uma variação do *Shortest Queue* com informações desatualizadas, demonstrando que algoritmos que consideram o estado dos servidores podem oferecer ganhos expressivos em tempo de resposta, especialmente quando a informação sobre a fila é atualizada com frequência. No entanto, destacam que em cenários com latência ou *overhead* de monitoramento, técnicas híbridas que mantêm histórico durante períodos de estagnação da fila apresentam melhor desempenho.

As técnicas de balanceamento de carga em ambientes de nuvem têm sido amplamente estudadas e categorizadas em abordagens estáticas, dinâmicas, híbridas e inspiradas na natureza. Uma revisão abrangente realizada por [Lohumi et al. 2023] destaca ainda os principais desafios enfrentados por essas técnicas, como escalabilidade, consumo de energia, tempo de resposta e tolerância a falhas. Os autores enfatizam a importância de métricas adequadas e sugerem que abordagens dinâmicas apresentam melhor adaptação a cenários com variabilidade no volume de requisições e heterogeneidade entre os nós.

No trabalho de [Mondal et al. 2016] o foco é o papel dos algoritmos dinâmicos em ambientes de computação em nuvem, sugerindo que estratégias cooperativas entre os nós podem reduzir o tempo de resposta e aumentar a resiliência do sistema.

De forma semelhante, [Kumar and Venkatesan 2019] destacam que algoritmos dinâmicos, embora mais complexos, são mais eficazes em ambientes com variação frequente de carga e heterogeneidade de *hardware*, como é o caso de clusters compostos por dispositivos embarcados com diferentes capacidades computacionais. O uso de algoritmos híbridos também é sugerido como alternativa promissora, combinando as vantagens de abordagens determinísticas e heurísticas

Além das abordagens algorítmicas, diversos estudos têm explorado o desempenho de servidores web sob diferentes métricas e cenários de carga. Um desses estudos é a revisão sistemática realizada por Kunda et al. [Kunda et al. 2017], que analisou os servidores Apache e Nginx considerando parâmetros como tempo de resposta, uso de memória e utilização da CPU nos servidores de aplicação. Os testes mostraram que o Nginx apresenta desempenho superior em cargas com alto grau de concorrência, mantendo consumo de memória estável mesmo com o aumento do número de requisições.

O trabalho de [Zhang and Fan 2008] propõe um modelo de filas com dois servidores e analisam políticas de roteamento em ambientes centralizados e distribuídos. O estudo busca identificar a estratégia que minimize a latência.

Um algoritmo de balanceamento dinâmico que utiliza a equação de condução de calor como modelo foi desenvolvido por He et al. [He et al. 2024], combinando redes neurais e algoritmos genéticos para ajustar pesos de servidores em tempo real. A abordagem simula a transferência de carga como fluxo térmico, otimizando o algoritmo Weighted Least Connections. Os resultados mostraram menor tempo de resposta e maior estabilidade sob alta concorrência, além de boa eficiência em ambientes com computação paralela

Em uma avaliação prática dos algoritmos nativos de balanceamento de carga do Nginx, [Ma and Chi 2022] analisaram estratégias como *Round Robin, Weighted Round Robin (WRR)*, `IP_HASH` e *Least Connections* — em um ambiente de alta concorrência com suporte ao Keepalived. O estudo propôs o algoritmo `NEW_HASH`, uma versão aprimorada do `IP_HASH`, que introduz duas inovações principais: uma nova função de hash baseada no algoritmo Times 33 e uma substituição do processo de busca sequencial por uma busca binária com ordenação prévia dos pesos.

O presente trabalho contribui com uma avaliação prática e comparativa de três estratégias implementáveis no Nginx — *Round Robin, Weighted e Least Connections* — em um ambiente com dispositivos heterogêneos, simulando diferentes padrões de carga (leve e CPU-intensiva).

4. Abordagem proposta

A metodologia consistiu na construção de um ambiente experimental composto por três servidores *backend* com capacidades computacionais distintas. Essa heterogeneidade foi intencionalmente introduzida a fim de simular situações realistas em que recursos variam em desempenho, como ocorre comumente em ambientes híbridos ou com máquinas legadas. Foram realizadas duas categorias de testes: requisições com baixa carga de processamento (operações de leitura em memória) e requisições com carga intensiva (operações matemáticas pesadas), de forma a avaliar o impacto dos algoritmos sob diferentes perfis de uso.

Cada teste foi executado com os três algoritmos nativos do Nginx em sessões distintas, e o número de requisições direcionadas a cada servidor foi registrado. As métricas observadas incluíram a porcentagem de requisições por nó, tempo de resposta médio e distribuição de carga em função do tipo de requisição. Essa análise permitiu observar como cada algoritmo se comporta diante do desbalanceamento entre os *hosts* e quais estratégias se mostram mais eficazes na mitigação de sobrecargas.

O Nginx foi escolhido como balanceador de carga para este estudo por ser amplamente adotado na indústria, devido à sua leveza, alto desempenho e facilidade de configuração [Garnett and Ellingwood 2022]. Sua popularidade se deve também ao suporte nativo a diferentes algoritmos de balanceamento, como *Round Robin, Weighted e Least Connections*, o que o torna ideal para comparar estratégias distintas em cenários reais. Além disso, o Nginx possui ampla documentação, comunidade ativa e é compatível com sistemas embarcados e ambientes de baixo consumo, como os utilizados nos experimentos deste trabalho.

Com essa abordagem, busca-se não apenas comparar o desempenho entre os algoritmos de balanceamento, mas também fornecer subsídios para decisões de configuração em ambientes de produção onde a igualdade entre os servidores nem sempre pode ser garantida. O estudo oferece, portanto, uma visão prática e orientada por dados sobre a eficácia do Nginx frente ao desafio do balanceamento desigual.

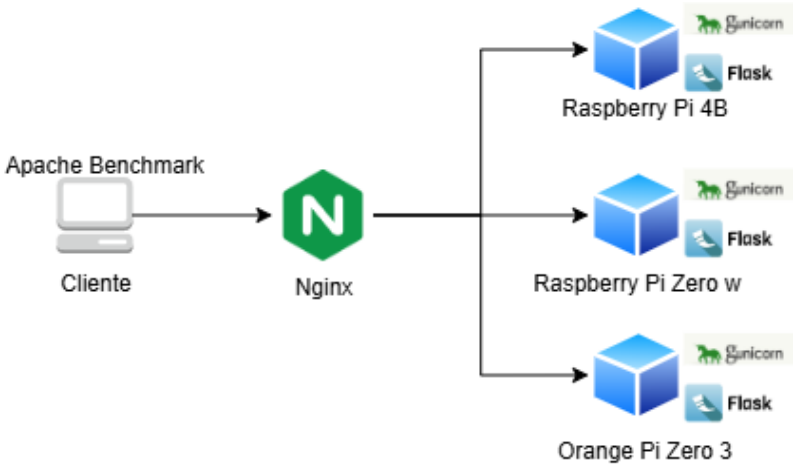

Figura 5. Topologia utilizada no experimento.

5. Materiais e métodos

Esta seção descreve a infraestrutura empregada nos experimentos realizados, abrangendo os dispositivos utilizados, os sistemas operacionais adotados, além das ferramentas e tecnologias responsáveis pela construção e execução dos serviços. A seguir, detalham-se as características das placas computacionais, os sistemas Linux utilizados, bem como os principais componentes de software como o servidor web Nginx, o servidor de aplicações Gunicorn, a linguagem de programação Python, o microframework Flask e a ferramenta de análise de desempenho Apache Benchmark.

5.1. Servidores

Foram utilizadas três placas do tipo *Single Board Computer* (SBC), com diferentes capacidades computacionais, a fim de simular cenários de balanceamento com *hosts* desbalanceados.

5.1.1. Raspberry Pi 4B

A Raspberry Pi 4 Modelo B é uma das placas mais potentes da linha Raspberry Pi, equipada com processador Broadcom BCM2711 quad-core Cortex-A72 de 1.5 GHz e 4 GB de memória RAM. Ela foi utilizada como um dos principais servidores de aplicação, representando um nó de alta capacidade computacional. Sua maior performance foi essencial para observar como o balanceador de carga redirecionava requisições em cenários com forte disparidade de recursos entre os nós.

Como sistema operacional foi usado o Raspberry Pi OS, a distribuição oficial mantida pela Raspberry Pi Foundation. Sua familiaridade com o ecossistema Raspberry e a boa compatibilidade com os pacotes necessários tornaram essa escolha ideal.

Figura 6. Exemplo de placa Raspberry Pi 4 Model B. Fonte: Site Raspberry Pi.

5.1.2. Raspberry Pi Zero W

A Raspberry Pi Zero W é uma placa de baixo custo e consumo, com processador single-core ARM1176JZF-S de 1 GHz e 512 MB de RAM. Essa placa foi escolhida propositalmente para simular um nó com recursos muito limitados, permitindo verificar como os algoritmos de balanceamento reagem à presença de um servidor com baixa capacidade de resposta. Sua atuação foi especialmente relevante nos testes com cargas mais pesadas, evidenciando gargalos e possíveis falhas na distribuição de requisições.

Assim como o Raspberry Pi 4B, o Raspberry Pi Zero W também usou o Raspberry Pi OS como sistema operacional, a versão utilizada foi a “Lite”, sem interface gráfica, otimizando o uso de memória e processamento.

Figura 7. Exemplo de placa Raspberry Pi Zero W. Fonte: Site Raspberry Pi.

5.1.3. Orange Pi Zero 3

A Orange Pi Zero 3 é uma placa compacta equipada com processador Allwinner H618 quad-core Cortex-A53 e 1 GB de RAM DDR3. Esta SBC serviu como um intermediário entre as duas placas anteriores em termos de desempenho. Sua inclusão permitiu observar como os algoritmos ponderavam os diferentes níveis de recursos disponíveis, funcionando como um nó de desempenho moderado no ambiente balanceado.

O sistema operacional usado foi a distribuição Linux Armbian IoT, devido ao seu baixo consumo de recursos e suporte nativo à placa. Essa distribuição permitiu executar os serviços de forma eficiente, mesmo em condições de uso contínuo e carga elevada.

Figura 8. Exemplo de placa Orange Pi Zero 3. Fonte: Site Orange Pi.

5.1.4. Dell Inspiron 3437

O balanceador de carga Nginx foi executado em um computador do tipo notebook modelo Dell Inspiron 3437, equipado com processador Intel Core i5 ULV de quarta geração, 8GB de memória RAM DDR3L e interface de rede Ethernet 10/100 Mbps integrada à placa-mãe. O sistema operacional usado aqui foi o Ubuntu Linux 22.04.

5.2. Nginx

O Nginx foi utilizado como balanceador de carga reverso, instalado em um nó externo responsável por distribuir as requisições HTTP entre as placas. Foram exploradas diferentes

diretivas de balanceamento como *Round Robin, Weighted, Least Connections*, avaliando-se o desempenho em cenários variados. A configuração do Nginx foi ajustada manualmente em cada experimento, com *logs* ativados para análise posterior das distribuições de carga.

```
# Define which servers to include in the load balancing scheme.
# It's best to use the servers' private IPs for better performance and security.
# You can find the private IPs at your UpCloud control panel Network section.

http {
upstream backend {
      server 192.168.0.101:8000 weight=6; # Raspberry Pi 4B
      server 192.168.0.102:8000 weight=1; # Raspberry Pi Zero W
      server 192.168.0.103:8000 weight=3; # Orange Pi Zero 3
   }

   # This server accepts all traffic to port 80 and passes it to the upstream.
   # Notice that the upstream name and the proxy_pass need to match.

   server {
      listen 80;

      location / {
          proxy_pass http://backend;
          proxy_set_header Host $host;
          proxy_set_header X-Real-IP $remote_addr;
      }
   }
}
```

Figura 9. Arquivo de configuração do NGINX.

5.3. Aplicações nos servidores

Cada servidor da arquitetura foi configurado para executar uma aplicação web desenvolvida com Flask, um microframework Python, e servida pelo Gunicorn, um servidor WSGI(*Web Server Gateway Interface)* de alto desempenho. O Gunicorn permite que múltiplas requisições sejam atendidas simultaneamente por meio da criação de múltiplos *workers* (processos independentes) e, opcionalmente, múltiplas *threads* por *worker*. Essa combinação permite que as aplicações sejam executadas de forma eficiente e responsiva, mesmo em dispositivos com recursos computacionais limitados. A seguir, temos a descrição detalhada da configuração das duas aplicações.

5.3.1. Flask

No experimento, a aplicação Flask foi desenvolvida com dois *endpoints* distintos, implementados na mesma base de código, com o objetivo de simular cargas de trabalho com características diferentes. O primeiro endpoint (/api/ram/¡param¿) retorna uma resposta imediata, consistindo em uma simples *string* fixa, representando requisições com baixa demanda computacional. Já o segundo endpoint (/api/fatorial/¡param¿) realiza o cálculo do fatorial de 1000, uma operação propositalmente intensiva em CPU, utilizada para simular cenários de alta carga computacional.

Essa diferenciação entre os *endpoints* permitiu avaliar como os algoritmos de balanceamento de carga do Nginx se comportam frente a requisições com perfis distintos — uma mais leve e outra bastante exigente em termos de processamento —, fornecendo uma

visão mais completa sobre a distribuição e o desempenho dos servidores em diferentes condições de estresse.

```python
import logging
from flask import Flask, jsonify, request
from math import factorial

logging.basicConfig(level=logging.INFO)
app = Flask(__name__)


@app.route('/api/ram/<identificador>')
def get_data(identificador):
    app.logger.info(f"Recebi Request /api/data - Raspberry Pi 4B - Teste: {identificador}")
    return jsonify({"name": "cluster_pi", "status": "success"})


@app.route('/api/factorial/<identificador>', methods=['GET'])
def compute_factorial(identificador):
    app.logger.info(f"Recebi Request /api/factorial - Raspberry Pi 4B - Teste: {identificador}")
    n = int(request.args.get('number', 1000))
    result = factorial(n)
    return jsonify({"factorial_length": len(str(result))})  # Retorna o tamanho do resultado

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, threaded=True)
```

Figura 10. Aplicação Flask.

5.3.2. Gunicorn

No experimento, cada instância da aplicação Flask foi servida por um processo Gunicorn responsável por aceitar conexões e distribuir o processamento entre múltiplos *workers*. O comando utilizado para iniciar o serviço foi o seguinte:

```
gunicorn -w 9 -b 0.0.0.0:8000 app:app
```

Esse comando configura o Gunicorn com 9 *workers* (-w 9), escutando na interface de rede local (0.0.0.0) e na porta 8000, executando a aplicação definida no módulo app.

De acordo com a documentação oficial do Gunicorn [Gunicorn Developers 2024], o número de *workers* não deve ser diretamente proporcional ao número de clientes esperados. Em geral, recomenda-se uma quantidade entre 4 e 12 *workers*, o que já é suficiente para atender centenas ou milhares de requisições por segundo. Uma fórmula comum para estimar um ponto de partida razoável é:

```
(2 × número_de_núcleos) + 1
```

Essa estimativa considera que, para cada núcleo, um *worker* pode estar ocupado com operações de I/O (como leitura ou escrita em *sockets*), enquanto outro está processando uma requisição. Embora a fórmula não seja rigorosa, ela serve como ponto de partida para ajustes conforme o comportamento da aplicação em testes de carga.

No caso do experimento, o Raspberry Pi 4B e o Orange Pi Zero 3 possuem processadores *quad-core*, e por isso utilizaram 9 *workers*, seguindo diretamente a fórmula sugerida. Já o Raspberry Pi Zero W, com processador *single-core*, foi configurado com 3 *workers*, respeitando o mesmo critério proporcional. Essa configuração buscou equilibrar o uso eficiente dos recursos de *hardware*, evitando a criação excessiva de processos em dispositivos mais limitados, o que poderia prejudicar o desempenho ao invés de melhorá-lo.

Vale destacar que o número de *workers* deve ser ajustado conforme o tipo de aplicação, o perfil das requisições e os limites do sistema operacional. Excesso de *workers* pode levar à competição por recursos, causando degradação de desempenho. Por isso, a escolha dos valores neste experimento levou em consideração tanto a orientação da documentação quanto testes prévios realizados em cada dispositivo.

5.4. Apache Benchmark

O Apache Benchmark (ab) é uma ferramenta de linha de comando utilizada para realizar testes de carga em servidores HTTP. Por meio dela, é possível simular múltiplas requisições simultâneas a um determinado *endpoint*, permitindo avaliar o desempenho do serviço sob diferentes níveis de estresse.

Durante os experimentos, o Apache Benchmark foi utilizado para gerar carga controlada e coletar métricas como tempo médio de resposta, *throughput* e número de falhas. Um dos comandos utilizados foi o seguinte:

```
ab -n 10000 -c 100 192.168.0.105:8000/api/ram/20252025
```

Esse comando realiza 10.000 requisições HTTP (-n 10000) ao *endpoint* /api/-ram/20252025, distribuídas com uma concorrência de 100 requisições simultâneas (-c 100). O endereço IP 192.168.0.105 corresponde ao balanceador de carga Nginx, que distribui essas requisições entre os servidores de aplicação.

O objetivo desse teste foi simular um cenário de uso intenso, permitindo observar como os diferentes algoritmos de balanceamento (*Round Robin, Weighted, Least Connections*) se comportam diante de um grande volume de acessos concorrentes. Os dados gerados pelo Apache Benchmark foram posteriormente analisados para quantificar o desempenho de cada estratégia e identificar gargalos ou comportamentos anômalos.

6. Resultados

Para avaliar o desempenho dos algoritmos de balanceamento de carga, foram definidos dois tipos distintos de requisição às aplicações Flask hospedadas nos servidores:

- **Requisição simples**: corresponde ao *endpoint* `/api/ram`, que apenas retorna uma resposta JSON estática sem operações computacionais além do mínimo necessário. Essa requisição simula um cenário de carga leve, sendo útil para analisar o comportamento dos algoritmos em situações de grande volume de acessos com baixa complexidade de processamento.
- **Requisição com uso intensivo de CPU**: corresponde ao *endpoint* `/api/factorial`, no qual a aplicação realiza o cálculo do fatorial de um número elevado (como 1000) e retorna o número de dígitos do resultado. Este tipo de requisição representa um cenário de carga pesada, exigindo processamento mais intenso por parte dos servidores, o que permite avaliar como os algoritmos distribuem requisições com alto custo computacional.

Durante os testes, foram registrados os seguintes indicadores temporais para cada requisição, medidos em milissegundos:

- **ctime (connection time)**: tempo necessário para estabelecer a conexão com o servidor antes de enviar a requisição.

- **dtime (duration time)**: tempo que a conexão permaneceu aberta após ser estabelecida, ou seja, o tempo total da requisição subtraído do tempo de conexão.
- **ttime (total time)**: tempo total gasto desde o início da conexão até o recebimento completo da resposta.
- **wait**: intervalo entre o envio da requisição e o início da leitura da resposta, refletindo o tempo que o servidor levou para processar e responder.
- **Time taken for tests**: tempo total decorrido para a execução de todas as requisições do teste de carga.
- **Requests per second**: número médio de requisições atendidas por segundo durante o teste, representando a vazão (*throughput*) do sistema.
- **Time per request**: tempo médio gasto para atender cada requisição individualmente, considerando o total de requisições enviadas.

Essas métricas permitem uma análise detalhada do comportamento dos algoritmos testados em diferentes cenários de carga e infraestrutura.

6.1. Resultados com Estratégia Round Robin

Na primeira etapa do experimento, foi utilizada a estratégia de balanceamento de carga *Round Robin*, na qual as requisições são distribuídas uniformemente entre os servidores disponíveis. A aplicação foi testada com dois tipos distintos de carga: uma requisição simples (identificada como RAM) que retorna uma *string* pré-definida, e uma requisição mais intensiva (Fatorial) que realiza o cálculo de um fatorial, com o objetivo de simular processamento computacional intensivo.

Tabela 1. Distribuição da quantidade de requisições por tipo e dispositivo

Tipo de Requisição	RP Zero	RP 4B	OP Zero	Total
Simples	3333	3334	3333	10000
Uso intenso de CPU	3334	3333	3333	10000

6.1.1. Requisição do Tipo Simples

A análise dos dados coletados, representando 10.000 requisições, revelou padrões distintos nos tempos de resposta do sistema. O tempo de conexão com o servidor (*ctime*) apresentou valores medianos extremamente baixos (1 ms), indicando uma conexão geralmente rápida, embora com picos isolados que ultrapassaram 3 segundos, o que elevou a média para 8,76 ms. O tempo de duração da conexão (*dtime*) e o tempo de espera pela resposta (*wait*) apresentaram comportamento praticamente idêntico, com mediana de 5 ms, mas com alta dispersão (desvio padrão de aproximadamente 731 ms), sugerindo que, em diversas situações, a resposta do servidor sofreu atrasos significativos. O tempo total da requisição (*ttime*), que combina os demais tempos, variou de 4 ms a mais de 5 segundos, com mediana de 6 ms, evidenciando que a maioria das requisições foi concluída rapidamente, mas uma parcela considerável sofreu lentidão acentuada. A Figura 11 com o boxplot gerado a partir dos dados confirma essa assimetria, destacando a presença de *outliers* que impactam negativamente a performance geral. Esses resultados sugerem que, embora o sistema seja eficiente na maior parte do tempo, sua estabilidade pode ser comprometida em cenários de carga elevada ou processamento intensivo.

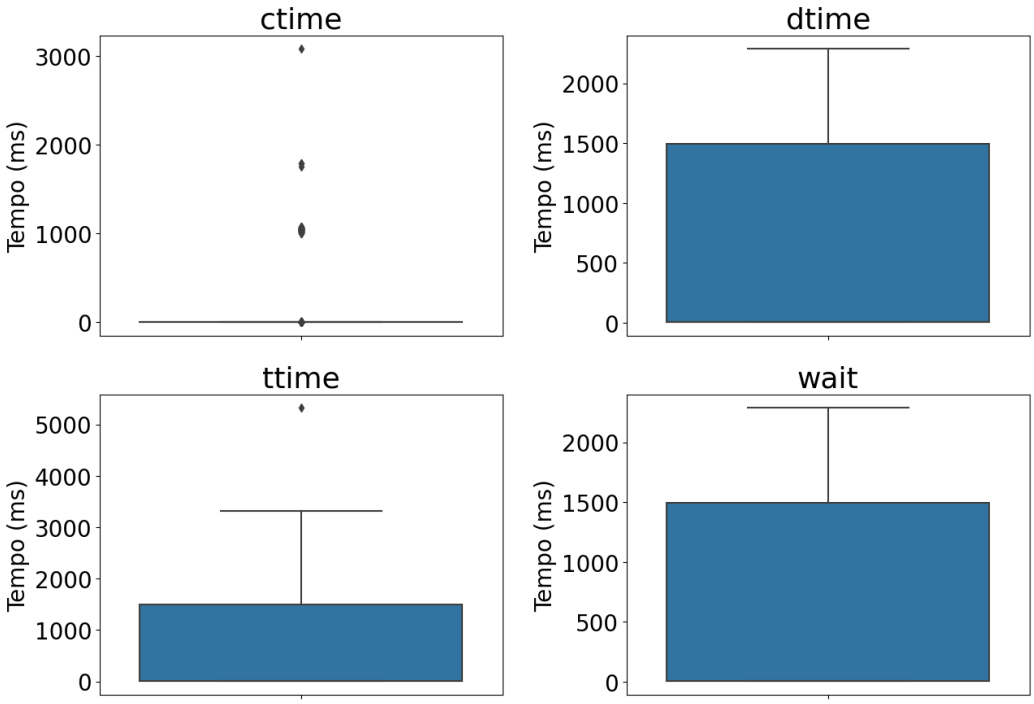

Figura 11. Métricas para método Round Robin em Requisições do Tipo Simples (eixo Y em milissegundos).

6.1.2. Requisição com uso intenso de CPU

Os dados para uso intenso de CPU na estratégia *Round Robin* revelam uma distribuição assimétrica nos tempos de resposta das requisições, com forte presença de valores extremos. O tempo de conexão com o servidor (ctime) apresentou mediana de 1 ms, com média de 9,05 ms e valores máximos que chegaram a 1814 ms, indicando ocorrências isoladas de alta latência na fase inicial da requisição. O tempo de duração da conexão (dtime) teve mediana de 16 ms e desvio padrão elevado (1598 ms), com registros que ultrapassaram 4 segundos. O tempo de espera pela resposta (wait) exibiu comportamento semelhante ao de dtime, com valores concentrados entre 5 ms e 15 ms, mas com grande dispersão e máximos superiores a 4 segundos. Já o tempo total da requisição (ttime) variou de 6 ms a mais de 5 segundos, com mediana de 16 ms, evidenciando que, apesar da maioria das requisições ser processada rapidamente, há uma fração considerável sujeita a atrasos significativos. A Figura 12 destaca a assimetria e a presença de outliers, sugerindo variações pontuais de desempenho que impactam a consistência do sistema.

6.2. Resultados com Estratégia Weighted

Na segunda etapa do experimento, adotou-se a estratégia de balanceamento de carga *Weighted*, na qual as requisições foram distribuídas de forma proporcional à capacidade estimada de cada dispositivo. A distribuição foi configurada como segue: 60% das requisições para o Raspberry Pi 4B, 30% para o Orange Pi Zero 3 e 10% para o Raspberry Pi Zero W. Essa configuração visa atenuar a sobrecarga nos dispositivos menos potentes, direcionando a maior parte das requisições para o nó com maior capacidade de processamento.

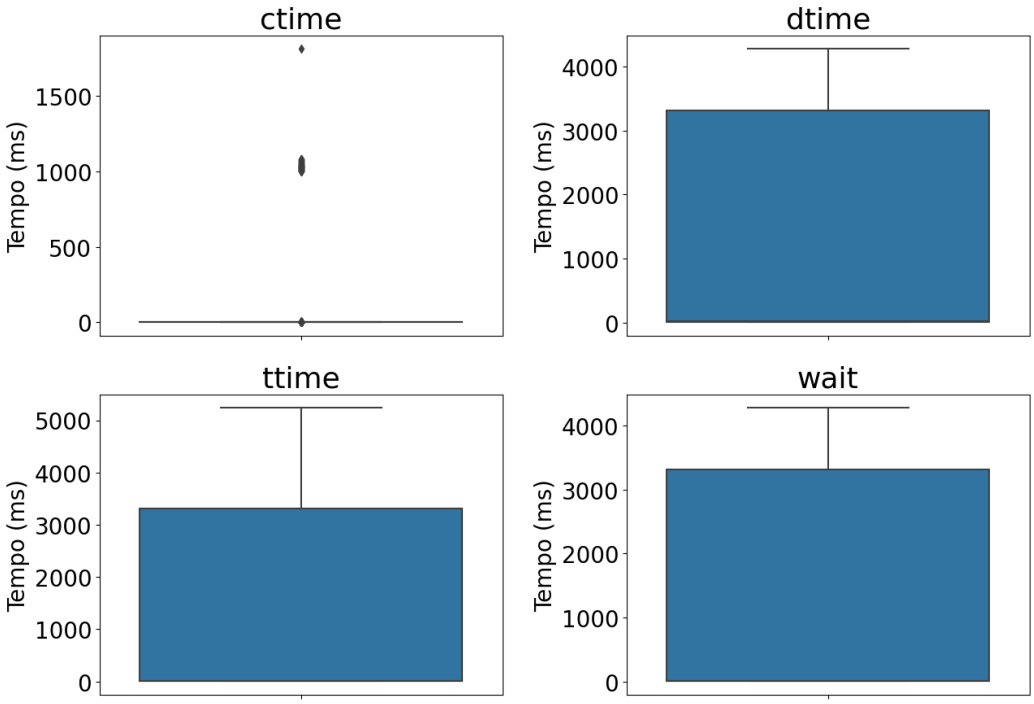

Figura 12. Métricas Para método Round Robin em Requisições de Uso Intenso de CPU (eixo Y em milissegundos).

A Tabela 2 resume a quantidade de requisições recebidas por cada máquina.

Tabela 2. Distribuição da quantidade de requisições por tipo e dispositivo

Tipo de Requisição	RP Zero	RP 4B	OP Zero	Total
Simples	1001	5999	3000	10000
Uso intenso de CPU	1000	6000	3000	10000

6.2.1. Requisição do Tipo Simples

A análise dos tempos de requisição registrados para requisições simples em estratégia *Weighted* revela uma distribuição fortemente concentrada em valores baixos, com exceções pontuais de alta latência. O tempo de conexão (*ctime*) apresentou mediana de 1 ms e média de 8,85 ms, com valores extremos que chegaram a 3157 ms. O tempo de duração da conexão (*dtime*) teve mediana de 4 ms, com a maioria das amostras situadas entre 3 e 5 ms, mas também com registros que ultrapassaram 1,5 segundo. O tempo de espera pela resposta (*wait*) teve distribuição praticamente idêntica à de *dtime*, com média de 140,86 ms e desvio padrão elevado (410,61 ms), o que indica a ocorrência de latências pontuais no processamento das respostas. O tempo total da requisição (*ttime*) variou de 3 ms a mais de 3 segundos, com mediana de 5 ms. A Figura 13 evidencia que, apesar da maior parte das requisições ser tratada de forma eficiente, há uma pequena quantidade de *outliers* com tempos significativamente maiores, o que pode impactar a experiência do usuário em situações específicas. O padrão observado sugere um sistema geralmente estável, mas sujeito a variações esporádicas de desempenho.

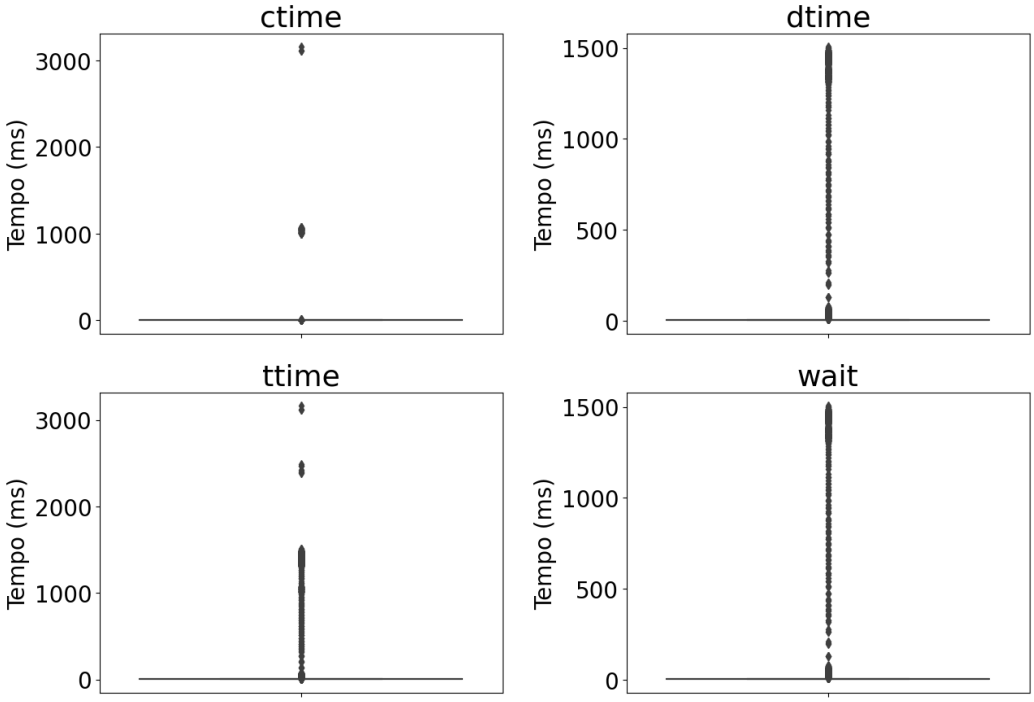

Figura 13. Métricas para Método Weighted em Requisições do Tipo Simples (eixo Y em milissegundos).

6.2.2. Requisição com uso intenso de CPU

A análise dos tempos de requisição cno cenário com uso intenso de CPU na estratégia Weighted indica que, embora a maioria das requisições tenha sido processada com baixa latência, há uma presença relevante de valores extremos que afetam as médias gerais. O tempo de conexão (*ctime*) apresentou mediana de 1 ms e média de 9,21 ms, com valores máximos de até 7200 ms, refletindo casos isolados de latência elevada na fase de conexão. O tempo de duração da conexão (*dtime*) concentrou-se majoritariamente em torno de 5 a 18 ms, com mediana de 15 ms, mas também registrou valores que superaram 3,9 segundos. O tempo de espera pela resposta (*wait*) teve comportamento semelhante, com média de 327,49 ms e desvio padrão elevado (923,56 ms), evidenciando instabilidades pontuais. O tempo total da requisição (*ttime*) variou entre 6 ms e 7216 ms, com mediana de 16 ms. A Figura 14 revela a forte concentração dos dados em torno de valores baixos e a presença de *outliers* que indicam lentidão esporádica no sistema. Esses resultados sugerem que, apesar da eficiência média observada, o desempenho do sistema pode ser comprometido em momentos de maior carga ou instabilidade.

6.3. Resultados com Estratégia Least Connections

Na terceira etapa do experimento, foi testada a estratégia de balanceamento de carga *Least Connections*, que direciona novas requisições para o servidor com menor número de conexões ativas no momento. Esta abordagem visa melhorar a eficiência dinâmica da distribuição, ajustando-se de forma reativa à carga atual de cada nó.

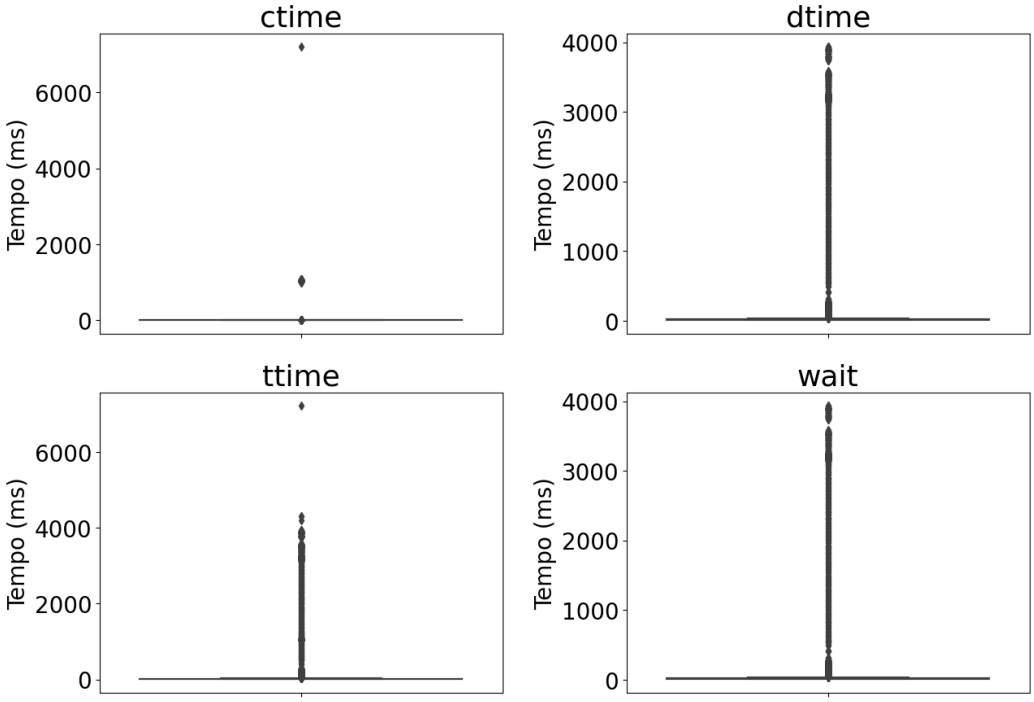

Figura 14. Métricas Para método Wheighted em Requisições de Uso Intenso de CPU (eixo Y em milissegundos).

Tabela 3. Distribuição da Quantidade de Requisições com Least Conn

Tipo de Requisição	RP Zero	RP 4B	OP Zero	Total
Simples	265	5932	3803	10000
Uso intenso de CPU	279	2525	7196	10000

6.3.1. Requisição do Tipo Simples

A análise do cenário de requisições do tipo simples na estratégia *Least Connections* evidencia um padrão de desempenho com baixa latência na maioria das requisições, mas com variações pontuais significativas. O tempo de conexão (*ctime*) apresentou mediana de 0 ms, indicando que a maioria das conexões foi estabelecida de forma quase imediata, embora o valor máximo tenha alcançado 1076 ms. O tempo de duração da conexão (*dtime*) variou entre 6 ms e 867 ms, com mediana de 17 ms e média de aproximadamente 31 ms. O tempo de espera pela resposta (*wait*) apresentou comportamento quase idêntico ao de *dtime*, o que sugere que a maior parte do tempo da requisição concentrou-se na espera por resposta após a conexão. Já o tempo total da requisição (*ttime*) teve mediana de 18 ms e valor máximo de 1102 ms, com desvio padrão relativamente elevado (111,76 ms), indicando a presença de *outliers*. A Figura 15 confirma uma distribuição concentrada em valores baixos. Esses resultados apontam para um sistema com desempenho geralmente eficiente e estável, mas que ocasionalmente ainda apresenta latência acima do esperado.

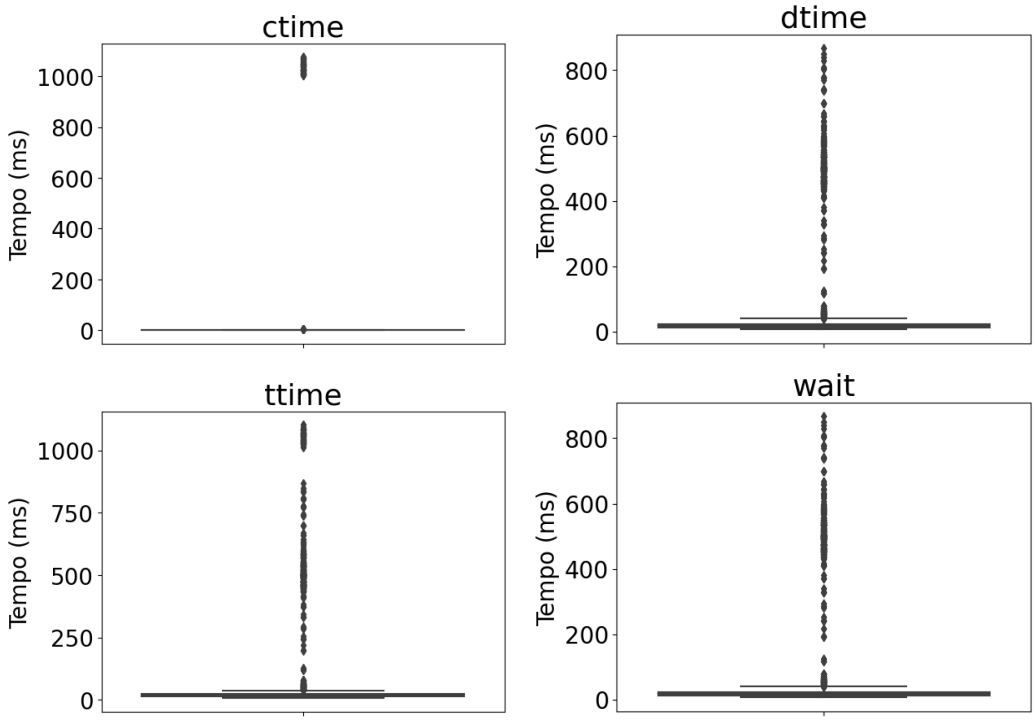

Figura 15. Métricas para método Least Connections em Requisições do Tipo Simples (eixo Y em milissegundos).

6.3.2. Requisição com uso intenso de CPU

Os resultados obtidos demonstram um desempenho altamente eficiente e estável. O sistema foi capaz de processar 1041,25 requisições por segundo, com um tempo médio por requisição de apenas 96,04 ms. Internamente, o tempo de espera pela resposta (*wait*) e o tempo de duração da conexão (*dtime*) apresentaram médias próximas a 82,2 ms, com desvios padrão de aproximadamente 199 ms, indicando baixa variabilidade na distribuição de carga entre os nós. O tempo de conexão (*ctime*) também foi reduzido, com média de 4,25 ms, o que sugere uma boa capacidade de aceitação e encaminhamento de requisições mesmo em momentos de alta concorrência. O tempo total da requisição (*ttime*) ficou limitado a 2,3 segundos no pior caso, com uma mediana de 35 ms e desvio padrão de 208,6 ms, revelando que a maioria das respostas foi entregue rapidamente, com apenas poucos *outliers*.

6.4. Comparação entre Estratégias de Balanceamento

A seguir, comparam-se as três estratégias de balanceamento de carga analisadas — *Round Robin*, *Weighted* e *Least Connections* — quanto à eficiência na distribuição da carga e ao desempenho nas métricas de tempo. A avaliação considera tanto as requisições simples, com baixa demanda computacional, quanto as requisições de fatorial, que impõem alta carga de CPU.

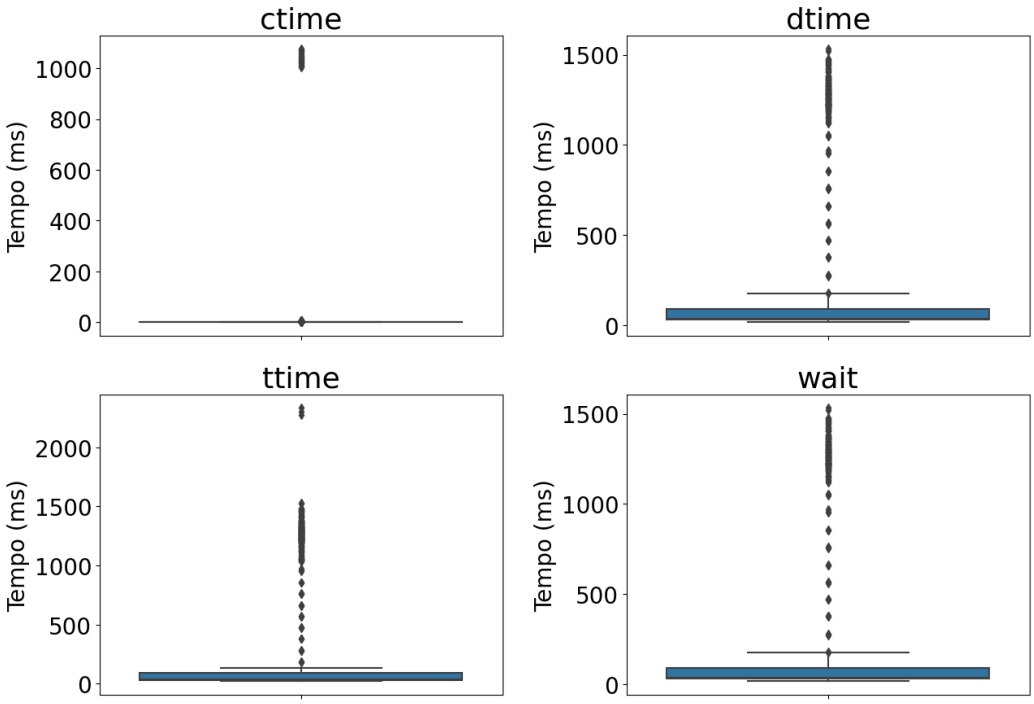

Figura 16. Métricas Para método Least Connections em Requisições de Uso Intenso de CPU (eixo Y em milissegundos).

6.4.1. Distribuição de Requisições

A Tabela 4 apresenta um resumo da distribuição de requisições em cada estratégia. Observa-se que:

- *Round Robin* realiza uma divisão equitativa, ignorando a capacidade dos dispositivos.
- *Weighted* aplica uma política fixa de distribuição proporcional à capacidade estimada.
- *Least Connections* ajusta dinamicamente a distribuição com base na carga atual de cada nó.

Tabela 4. Percentual de Requisições por Estratégia

Estratégia	Tipo	RP Zero W	RP 4B	OP Zero
Round Robin	Simples	33%	33%	33%
Round Robin	CPU	33%	33%	33%
Weight	Simples	10%	60%	30%
Weight	CPU	10%	60%	30%
Least Conn	Simples	2.6%	59%	38%
Least Conn	CPU	2.8%	25%	72%

A estratégia *Least Connections* destaca-se por evitar o uso excessivo do RP Zero W, redirecionando o tráfego dinamicamente de acordo com a sobrecarga detectada, o que se reflete diretamente nas métricas de latência.

6.4.2. Comparação de métodos em requisições do tipo simples

A comparação entre as estratégias de balanceamento de carga abordadas no estudo revela uma diferença significativa no desempenho da aplicação, tanto em termos de capacidade de atendimento quanto na regularidade das respostas. No primeiro cenário, *Round Robin*, o sistema alcançou 188,1 requisições por segundo, com um tempo médio por requisição de 531,5 ms. Embora a mediana do tempo total (*ttime*) tenha sido bastante baixa (6 ms), os valores extremos atingiram 5322 ms e o desvio padrão foi elevado (736,9 ms), evidenciando alta instabilidade. O tempo de conexão (*ctime*) apresentou média de 8,76 ms, mas com grande dispersão (desvio padrão de 94,8 ms e máximo de 3084 ms), e o tempo de espera (*wait*) — que reflete a latência efetiva da resposta — teve média de 515,7 ms, indicando que o principal gargalo estava no processamento da resposta pelo servidor.

No segundo cenário, com a estratégia *Weighted*, observa-se uma melhoria substancial. O *throughput* médio subiu para 638,9 requisições por segundo, e o tempo médio por requisição caiu para 156,5 ms. O tempo de conexão manteve média semelhante (8,85 ms), mas com maior dispersão (desvio padrão de 100,3 ms e máximos superiores a 3 segundos). A grande diferença, no entanto, está nos tempos de duração (*dtime*) e espera (*wait*), que tiveram médias de 140,9 ms e desvios padrão de aproximadamente 410,6 ms, muito menores que no cenário anterior. O tempo total (*ttime*) foi mais controlado, com menor variabilidade (desvio padrão de 422,6 ms), o que indica maior estabilidade operacional.

O estratégia de *Least Connections*, demonstra um avanço ainda mais expressivo. O sistema atingiu 2381,8 requisições por segundo, com tempo médio por requisição de apenas 42 ms, o melhor resultado entre os três. Apesar de a mediana de *ttime* ter subido para 18 ms, o desvio padrão caiu drasticamente para 111,8 ms, e o valor máximo ficou limitado a 1102 ms. O tempo de conexão (*ctime*) manteve-se em média de 6,36 ms, com menor variabilidade (desvio padrão de 78,1 ms). Tanto o tempo de duração da conexão (*dtime*, média de 30,9 ms) quanto o tempo de espera (*wait*, média de 30,9 ms) tiveram comportamento estável, com desvios padrão reduzidos (80,9 ms), refletindo um sistema bem dimensionado e com resposta consistente mesmo sob alta carga.

Em síntese, os dados mostram uma transição clara de um sistema com baixa taxa de atendimento e alta variabilidade (*Round Robin*), passando por uma configuração intermediária mais equilibrada (*Weighted*), até chegar a uma arquitetura otimizada e robusta (*Least Connections*). As melhorias progressivas nos tempos de *dtime* e *wait*, acompanhadas do aumento no *throughput* e da redução nas dispersões, indicam avanços tanto na infraestrutura quanto na eficiência do processamento das requisições.

A Figura 17 apresenta uma visão comparativa da quantidade de requisições atendidas em cada estratégia de balanceamento.

6.4.3. Comparação de métodos em requisições com uso intenso de CPU

Neste cenário, as requisições submetidas ao sistema exigem processamento computacional intenso, o que evidencia ainda mais as diferenças de desempenho entre as estratégias de balanceamento. Quando utilizado o algoritmo *Round Robin*, o sistema apresentou

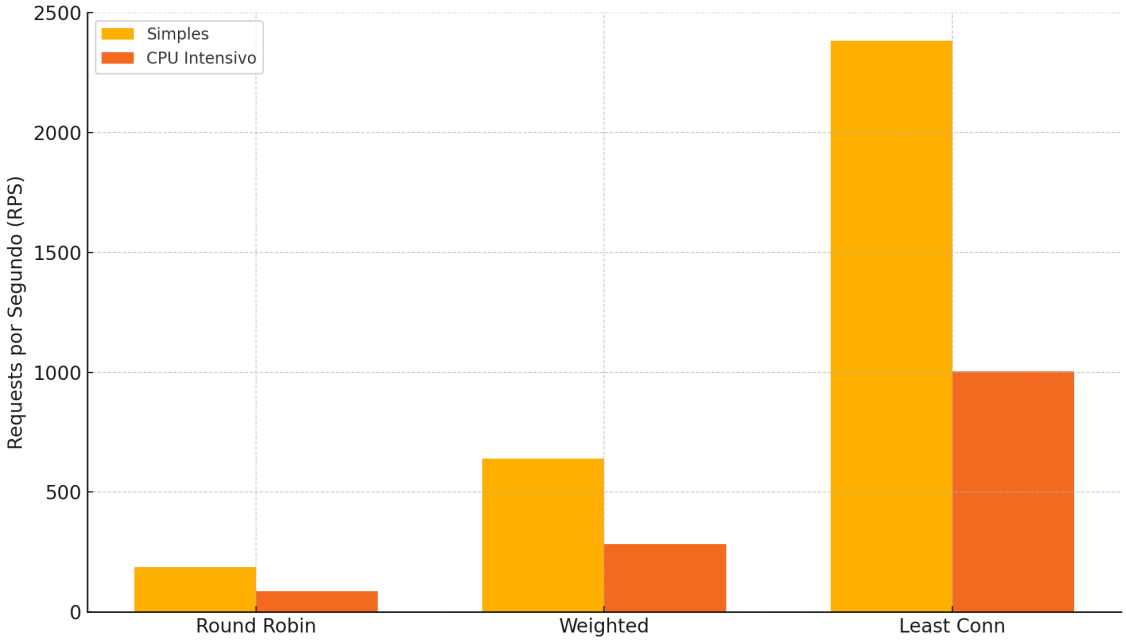

Figura 17. Comparação de *throughput* por estratégia de balanceamento.

desempenho insuficiente: apenas 86,4 requisições por segundo com tempo médio por requisição de 1156,9 ms. Internamente, os tempos de espera (*wait*) e duração da conexão (*dtime*) ultrapassaram 1100 ms em média, com desvios padrão próximos de 1600 ms. Essa variabilidade extrema mostra que o algoritmo, por não considerar o estado dos nós, gerou sobrecarga nos dispositivos menos potentes, degradando o desempenho geral.

Com o uso da estratégia *Weighted*, houve um salto considerável no desempenho. O *throughput* alcançou 283,1 requisições por segundo, e o tempo médio por requisição foi reduzido para 353,2 ms. As métricas internas de *wait* e *dtime* caíram para cerca de 327,5 ms, e o desvio padrão de *ttime* foi reduzido para 929 ms. Embora ainda houvesse variação significativa e alguns *outliers*, a distribuição das requisições tornou-se mais eficiente e menos propensa à sobrecarga dos nós mais frágeis.

O melhor desempenho foi obtido com o algoritmo *Least Connections*, que processou 1041,25 requisições por segundo com tempo médio de apenas 96,04 ms. Esse resultado representa uma melhora expressiva tanto em capacidade de atendimento quanto em latência média. Os tempos médios de *dtime* e *wait* ficaram em torno de 82,2 ms, com desvios padrão muito menores (199 ms), e o tempo de conexão (*ctime*) foi também o mais baixo entre os três (média de 4,25 ms). Além disso, o tempo total (*ttime*) foi limitado a no máximo 2,3 segundos, com mediana de 35 ms — indicando um sistema não apenas eficiente, mas também altamente estável mesmo sob carga pesada.

Em resumo, enquanto o *Round Robin* falhou em distribuir adequadamente as requisições, o *Weighted* trouxe melhorias significativas, e o *Least Connections* combinou alto *throughput* com estabilidade, demonstrando ser a estratégia mais eficaz para esse tipo de carga.

7. Conclusões

Neste trabalho, implementamos uma configuração com três *Single Board Computers* (SBCs) atuando como servidores de aplicação web, cada um com diferentes capacidades computacionais, de forma a simular um ambiente com *hosts* desbalanceados. Sobre esse conjunto, foi colocado um distribuidor de carga utilizando o Nginx, com o objetivo de avaliar o desempenho dos algoritmos de balanceamento *Round Robin*, *Weighted* e *Least Connections*.

Para isso, foi realizada a implementação das aplicações web com Flask, servidas pelo servidor WSGI Gunicorn, com configurações específicas de quantidade de *workers* adaptadas ao número de núcleos de cada dispositivo. Esse ambiente permitiu a condução de experimentos controlados, nos quais foram enviados grandes volumes de requisições, tanto leves (resposta pré-definida) quanto pesadas (cálculo de fatorial com uso intensivo de CPU), e coletadas métricas como tempo de resposta e distribuição da carga entre os nós.

Foi avaliado o desempenho do algoritmo *Round Robin*, que mostrou-se ineficiente em cenários heterogêneos. Sua incapacidade de considerar as diferenças de capacidade entre os servidores levou à sobrecarga do nó mais fraco, resultando em degradação da performance, especialmente sob cargas elevadas.

A estratégia *Weighted*, por outro lado, demonstrou maior equilíbrio ao atenuar o impacto do nó com menor capacidade, graças à atribuição manual de pesos. No entanto, sua limitação em se adaptar dinamicamente às mudanças do sistema comprometeu a eficiência máxima, principalmente em situações de variação de carga.

O algoritmo *Least Connections* apresentou o melhor desempenho geral. Sua capacidade de distribuir as requisições com base no número atual de conexões ativas permitiu uma melhor utilização dos *hosts* mais potentes e evitou a sobrecarga do mais fraco. Isso resultou em maior estabilidade e menor tempo de resposta médio, especialmente nos testes com carga pesada.

8. Trabalhos futuros

Inicialmente, recomenda-se a realização de testes em ambientes reais e com maior variedade de condições operacionais, a fim de validar os resultados obtidos e identificar possíveis limitações práticas que não foram observadas em ambiente controlado.

Outro ponto relevante seria a integração com ferramentas de monitoramento e visualização em tempo real, com a inclusão de *dashboards* interativos ou relatórios automáticos que auxiliem na análise contínua do desempenho do sistema.

Também se destaca a possibilidade de ampliar o escopo do projeto com a utilização de arquiteturas alternativas, algoritmos otimizados ou diferentes abordagens de balanceamento e gerenciamento de recursos, de acordo com os avanços tecnológicos e as necessidades específicas de cada cenário.

Referências

Afzal, S. and Kavitha, G. (2019). Load Balancing in Cloud Computing – A Hierarchical Taxonomical Classification. *Journal of Cloud Computing*, 8(1):22.

Beyer, D., Löwe, S., and Wendler, P. (2019). Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29.

Fatima, S. G., Fatima, S. K., Sattar, S. A., Khan, N. A., and Adil, S. (2019). Cloud Computing and Load Balancing. *International Journal of Advanced Research in Engineering and Technology (IJARET)*, 10(2):189–209. Article ID: IJARET_10_02_019.

Garnett, A. and Ellingwood, J. (2022). Apache vs nginx: Practical considerations. DigitalOcean Community Tutorial. Disponível em: `https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations` Acesso em: 20 jul. 2025.

Gunicorn Developers (2024). Gunicorn Design — How Many Workers? Disponível em: `https://docs.gunicorn.org/en/latest/design.html#how-many-workers`. Acesso em: 20 jul. 2025.

He, H., Wang, L., Liu, J., and Qin, L. H. (2024). Optimizing Cloud Service Load Balancing Through Heat Conduction Equation Applications. *International Journal of Heat and Technology*, 42(1):320–328.

Islam, S. (2017). Network Load Balancing Methods: Experimental Comparisons and Improvement.

ITU (2024). Measuring digital development: Facts and Figures 2024. Disponível em: `https://www.itu.int/itu-d/reports/statistics/2024/11/10/ff24-internet-use/` Acesso em: 20 jul. 2025.

Jiang, Z. and Hassan, A. E. (2015). A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering*, 41:1–1.

Kumar, A. M. S. and Venkatesan, M. (2019). Task scheduling in a cloud computing environment using HGPSO algorithm. *Cluster Computing*, 22(S1):2179–2185.

Kunda, D., Chihana, S., and Muwanei, S. (2017). Web Server Performance of Apache and Nginx: A Systematic Literature Review. 8:43–52.

Lohumi, Y., Gangodkar, D., Srivastava, P., Khan, M., Alahmadia, A., and Alahmadia, A. (2023). Load Balancing in Cloud Environment: A State-of-the-Art Review. *IEEE Access*, PP:1–1.

Ma, C. and Chi, Y. (2022). Evaluation Test and Improvement of Load Balancing Algorithms of Nginx. *IEEE Access*, 10:14311–14324.

Mondal, R. K., Ray, P., and Sarddar, D. (2016). Load balancing. *International Journal of Research in Computer Applications & Information Technology*, 4(1):01–21. DOA: 03012016.

NGINX, Inc. (2025). *HTTP Load Balancing*. Disponível em: `https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/`. Acesso em: 20 jul. 2025.

Shahid, M. A., Islam, N., Alam, M. M., Su'ud, M. M., and Musa, S. (2020). A Comprehensive Study of Load Balancing Approaches in the Cloud Computing Environment and a Novel Fault Tolerance Approach. *IEEE Access*, 8:130500–130526.

Shanmugam, S. and Iyenger, N. C. S. N. (2016). Effort of Load Balancer to Achieve Green Cloud Computing: A Review. *International Journal of Multimedia and Ubiquitous Engineering*, 11:317–332.

Zhang, Z. and Fan, W. (2008). Web Server Load Balancing: A Queueing Analysis. *European Journal of Operational Research*, 186:681–693.