Análise de Desempenho em um Sistema Distribuído Cliente-Servidor Baseado em Python para Validação de Modelos

Hector J. R. Salgueiros, Ueslei F. R. Ribeiro, Willians S. Santos
Sistemas de Informação - Universidade Federal do Piauí
{hectorsalg,willianssilva}@ufpi.edu.br, ueslei392@gmail.com

***Abstract*—Este artigo apresenta o desenvolvimento e a análise de um sistema distribuído cliente-servidor implementado em Python, projetado para a coleta de métricas de desempenho em diversas etapas de processamento. A ferramenta, inspirada no conceito do PASID-VALIDATOR, foca na captura de tempos intermediários (timestamps) para permitir a validação de modelos de sistemas distribuídos através da métrica de tempo médio de resposta (MRT). O sistema é orquestrado com Docker e Docker Compose, e sua arquitetura modular, composta por Source, Load Balancers e Services — incluindo um serviço de Inteligência Artificial para classificação de sentimentos com BERT —, é submetida a diferentes cargas de trabalho. Analisamos o impacto no MRT ao variar o número de requisições e a complexidade da pipeline de serviços. Os resultados demonstram a importância do paralelismo e do balanceamento de carga para a escalabilidade do sistema, estabelecendo uma base robusta para futuros experimentos de desempenho em ambientes distribuídos.**

***Index Terms*—Sistemas Distribuídos, Python, Docker, Desempenho, Tempo Médio de Resposta, Balanceamento de Carga, BERT.**

I. Introdução

Sistemas distribuídos são onipresentes na infraestrutura tecnológica moderna, desde aplicações em nuvem até microsserviços e sistemas de grande escala. A complexidade inerente a esses sistemas, no entanto, torna sua concepção, implementação e, crucialmente, sua validação, tarefas desafiadoras. A validação de um modelo de sistema distribuído visa garantir que o modelo teórico represente de forma acurada o comportamento do sistema real, permitindo previsões e análises confiáveis. Métricas de desempenho, como o Tempo Médio de Resposta (MRT), são fundamentais nesse processo, fornecendo insights quantitativos sobre a eficiência e a latência das operações.

O projeto PASID-VALIDATOR, originalmente desenvolvido em Java, propõe uma abordagem para a validação de modelos de Stochastic Petri Nets (SPN) através da captura de tempos de processamento em um sistema distribuído cliente-servidor. Embora o escopo deste trabalho não envolva diretamente SPNs, a ferramenta serve como um framework para a coleta sistemática de dados de desempenho em um fluxo de trabalho distribuído.

Este artigo descreve a implementação de uma versão em Python do PASID-VALIDATOR, com foco na captura de timestamps em cada etapa do processamento. A arquitetura modular do sistema, composta por um **Source**, **Load Balancers** e **Services**, é detalhada. Exploraremos os desafios e as soluções encontradas na replicação da funcionalidade de comunicação entre os nós e a coleta de dados temporais precisos, utilizando Docker e Docker Compose para orquestrar o ambiente. O objetivo final é estabelecer uma base robusta para a realização de experimentos de desempenho, onde fatores como o número de requisições, o número de serviços na pipeline e a inclusão de serviços de Inteligência Artificial (IA) são variados para analisar seu impacto no MRT e validar o comportamento do sistema.

As contribuições deste trabalho incluem:

- A portabilidade da arquitetura cliente-servidor do PASID-VALIDATOR para Python, com orquestração via Docker.
- A implementação de um mecanismo de coleta de timestamps detalhado em cada nó do sistema.
- A infraestrutura para simulações de fluxo de trabalho distribuído e análise de desempenho.
- A integração de um serviço de IA com um modelo BERT para simular cargas de trabalho pesadas e realistas, analisando seu impacto no desempenho geral.

O restante do artigo está organizado da seguinte forma: a Seção II discute trabalhos relacionados. A Seção III detalha a arquitetura e implementação da versão Python. A Seção IV descreve a metodologia experimental. A Seção V apresenta e discute os resultados obtidos. Finalmente, a Seção VI conclui o trabalho e propõe direções futuras.

II. Trabalhos Relacionados

A análise de desempenho em sistemas distribuídos é uma área de pesquisa consolidada, com uma vasta literatura abordando desde a modelagem teórica até a implementação de ferramentas práticas. A validação de modelos, como proposto pelo PASID-VALIDATOR original, busca conectar a teoria com a prática. Trabalhos como os de Jain [1] estabelecem os fundamentos da medição e avaliação de desempenho de sistemas computacionais, onde métricas como tempo de resposta, vazão e utilização de recursos são cruciais.

Ferramentas de medição de desempenho em sistemas distribuídos, como o Apache JMeter ou o Gatling, são amplamente utilizadas para testes de carga, mas frequentemente focam no sistema como uma ”caixa-preta”, medindo o tempo de resposta de ponta a ponta. Nossa abordagem se diferencia

pela instrumentação interna do sistema para capturar timestamps intermediários, permitindo uma análise mais granular dos gargalos, similar ao que sistemas de rastreamento distribuído como Jaeger e Zipkin oferecem em arquiteturas de microsserviços.

O uso de timestamps para rastreamento de requisições é fundamental em arquiteturas distribuídas modernas. O trabalho do Google sobre o Dapper [2] foi pioneiro ao introduzir um sistema de rastreamento de baixa sobrecarga para entender o comportamento de sistemas complexos em larga escala. Nossa implementação adota um princípio similar, embora em uma escala menor e com foco na validação de modelos específicos.

Finalmente, a comparação de desempenho entre linguagens, como Java e Python, em sistemas distribuídos é um tópico relevante. Embora Java seja frequentemente preferido para sistemas de baixa latência devido à sua Máquina Virtual (JVM) madura e ao seu forte modelo de concorrência, Python, com seu ecossistema robusto de bibliotecas (como ‘transformers‘ para IA) e a facilidade de prototipagem, tornou-se uma escolha viável, especialmente em aplicações de I/O-bound e para orquestração de serviços complexos. O uso de contêineres Docker, como em nosso trabalho, ajuda a mitigar diferenças de ambiente e a criar uma plataforma de implantação consistente, independentemente da linguagem [3]. Além dos trabalhos já citados, destaca-se a literatura sobre rastreamento distribuído, que evoluiu significativamente nos últimos anos. Sistemas como OpenTracing e OpenTelemetry oferecem padrões abertos para instrumentação, permitindo a integração de múltiplas ferramentas de monitoramento. Estudos recentes também exploram o impacto de diferentes estratégias de balanceamento de carga em ambientes de nuvem, evidenciando a importância de políticas adaptativas para maximizar a utilização de recursos e minimizar a latência [3].

Outra linha de pesquisa relevante envolve a comparação de desempenho entre diferentes linguagens e frameworks para sistemas distribuídos. Pesquisas como as de [2] analisam o trade-off entre facilidade de desenvolvimento, desempenho bruto e escalabilidade, especialmente em aplicações que envolvem processamento intensivo de dados ou inferência de modelos de IA.

III. PASID-VALIDATOR em Python: Arquitetura e Implementação

Esta seção descreve a arquitetura do sistema distribuído implementado em Python, replicando o modelo conceitual do PASID-VALIDATOR. O sistema é composto por múltiplos nós interconectados via comunicação por sockets TCP, com um foco central na captura de timestamps para o cálculo de tempos médios de resposta em cada etapa do fluxo de processamento. A orquestração dos componentes é realizada com Docker e Docker Compose, garantindo portabilidade e reprodutibilidade dos experimentos.

A. Visão Geral da Arquitetura

O sistema é organizado em três tipos principais de nós, executados como contêineres Docker distintos:

1) **Source (Nó 01):** O ponto de origem das requisições, responsável por gerar as mensagens, iniciá-las no fluxo de processamento e, posteriormente, coletar e compilar os resultados finais. Atua como orquestrador do experimento.
2) **Load Balancer (Nós 02 e 03):** Componentes intermediários que recebem requisições e as distribuem entre um conjunto de serviços downstream, utilizando uma política de balanceamento de carga Round Robin.
3) **Service (Nós 02 e 03):** Componentes que realizam o processamento real das requisições. No nosso caso, um dos serviços é ”pesado”, realizando inferência de IA com um modelo BERT para classificação de sentimentos. Os serviços podem ser intermediários ou finais.

A comunicação entre esses nós é baseada em sockets TCP, com mensagens serializadas em formato JSON.

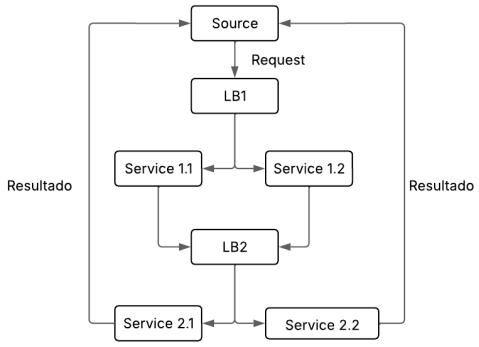

Fig. 1. Fluxo de dados e arquitetura do sistema distribuído PASID-VALIDATOR em Python.

B. Detalhamento dos Componentes

A modularidade do código é um pilar do projeto, com cada componente e utilitário em seu próprio módulo.

1) Orquestração com Docker e ‘run_experiments.sh‘: A execução dos experimentos é totalmente automatizada por um script shell (‘run_experiments.sh‘). Este script itera sobre diferentes configurações de cenário (número de serviços e número de requisições), gera um arquivo ‘.env‘ com as variáveis de ambiente necessárias para cada execução (portas, hosts, alvos dos LBs) e utiliza o ‘docker-compose‘ para construir e iniciar os contêineres. Esta abordagem garante que cada experimento seja executado em um ambiente limpo e com a configuração exata desejada.

2) Módulo ‘service.py‘ e ‘SentimentClassifier.py‘: A classe ‘ServiceNode‘ representa um serviço de processamento. Uma característica central do nosso experimento é a integração de um serviço de IA. O módulo ‘SentimentClassifier.py‘ encapsula um modelo BERT pré-treinado da biblioteca ‘transformers‘. Quando uma requisição chega a um serviço de IA, ele invoca o classificador para analisar o sentimento de um texto no payload da mensagem. Este processo é computacionalmente intensivo e simula uma carga de trabalho realista e ”pesada”, sendo um fator crucial na análise de desempenho.

3) Módulos 'source.py', 'loadbalancer.py', e 'socket_utils.py': O 'SourceNode' orquestra o início e o fim da simulação, gera a carga de trabalho e coleta os resultados. O 'LoadBalancerNode' implementa a lógica de distribuição Round Robin. O 'socket_utils.py' abstrai a comunicação de baixo nível, tratando da serialização JSON e do envio/recebimento de mensagens pela rede.

C. Nomenclatura e Coleta de Timestamps

A metodologia de coleta de tempos segue rigorosamente os marcos temporais definidos:

- **M1_source_prep_start:** Início da preparação da requisição no Source.
- **M2_source_sent_to_lb1:** Requisição enviada do Source para o LoadBalancer1.
- **M3_s1_exit_processed:** Requisição processada e saindo de um Service1.X.
- **M4_lb2_entry_after_transit_S1_LB2:** Requisição chegando no LoadBalancer2.
- **M5_s2_exit_processed:** Requisição processada e saindo de um Service2.X.
- **M6_source_entry_received_result:** Resultado final recebido pelo Source.

Esses marcos são utilizados para calcular os tempos intermediários (T1 a T5) e o tempo de resposta total de ponta a ponta para cada requisição.

IV. Metodologia Experimental

Esta seção detalha como os experimentos foram conduzidos para avaliar o desempenho do sistema.

A. Ambiente de Execução

Todos os componentes do sistema (Source, Load Balancers, Services) foram executados como contêineres Docker isolados, orquestrados por um arquivo 'docker-compose.yml'. Os experimentos foram executados em uma única máquina host, onde o Docker gerencia a alocação de recursos de CPU e memória para cada contêiner. A rede entre os contêineres foi gerenciada pelo Docker, simulando um ambiente de rede local. A automação foi garantida pelo script 'run_experiments.sh', que controla todo o ciclo de vida dos experimentos.

B. Configuração dos Cenários

Os experimentos foram projetados para analisar o impacto de duas variáveis principais: a complexidade da pipeline (número de serviços) e a carga de trabalho (número de requisições).

- **Serviço de IA:** Os serviços da família 'Sx.x' foram configurados para realizar uma tarefa de inferência de IA. Especificamente, eles utilizam um modelo BERT pré-treinado ('bert-base-multilingual-uncased-sentiment') para classificação de sentimentos. Este modelo é considerado "pesado" devido ao seu tamanho e à complexidade computacional da inferência, tornando-o ideal para simular uma carga de trabalho realista que consome CPU.
- **Variação de Fatores (linhas do gráfico):** A complexidade da pipeline foi variada em quatro cenários, definidos pelo número de "serviços ativos" que uma requisição atravessa:
  1) **1 Serviço Ativo:** Source → LB1 → S1.1 → Source
  2) **2 Serviços Ativos:** Source → LB1 → S1.1 → S1.2 → Source
  3) **3 Serviços Ativos:** Source → LB1 → S1.x → LB2 → S2.1 → Source
  4) **4 Serviços Ativos:** Source → LB1 → S1.x → LB2 → S2.x → Source

  Nos cenários 3 e 4, os Load Balancers distribuem a carga entre as instâncias de serviço disponíveis (e.g., S1.1 e S1.2), introduzindo paralelismo.
- **Variável X (eixo horizontal do gráfico):** O eixo X representa o **número total de requisições** geradas pelo Source em cada experimento. Foram testados os seguintes valores: 20, 50, 100, 150, 200, 250 e 300 requisições. O Source as envia em um loop, e a taxa efetiva é limitada pela capacidade de processamento do sistema.

C. Coleta de Dados e Geração de Gráficos

Para cada combinação de cenário e número de requisições, o 'SourceNode' executa a simulação, enviando todas as requisições e aguardando seus respectivos retornos. Ele registra os timestamps de 'M1' a 'M6' para cada uma. Ao final, calcula o tempo de resposta total ('M6 - M1') para cada requisição e, então, a média para aquele experimento (MRT). Os dados de MRT foram salvos em arquivos de log, e a biblioteca 'matplotlib' em Python foi utilizada para gerar o gráfico de desempenho consolidado.

V. Resultados e Discussão

Esta seção apresenta e analisa os resultados dos experimentos de desempenho, com foco na relação entre o número de requisições, a complexidade da pipeline de serviços e o Tempo Médio de Resposta (MRT).

A. Apresentação dos Resultados

A Figura 2 consolida os resultados dos experimentos. O gráfico ilustra o MRT (em milissegundos) no eixo Y em função do número total de requisições no eixo X. Cada linha colorida representa uma das quatro configurações de pipeline de serviços testadas.

B. Interpretação e Discussão

A análise do gráfico revela vários insights importantes sobre o comportamento do sistema sob diferentes condições.

- **Impacto da Complexidade da Pipeline:** Conforme o esperado, pipelines com mais estágios de processamento (mais serviços) apresentam um MRT base mais elevado. Isso é visível ao comparar os pontos de partida (com 20 requisições) das curvas: a linha azul (4 serviços) e a verde (3 serviços) começam com um MRT significativamente maior que a laranja (2 serviços) e a vermelha (1 serviço).

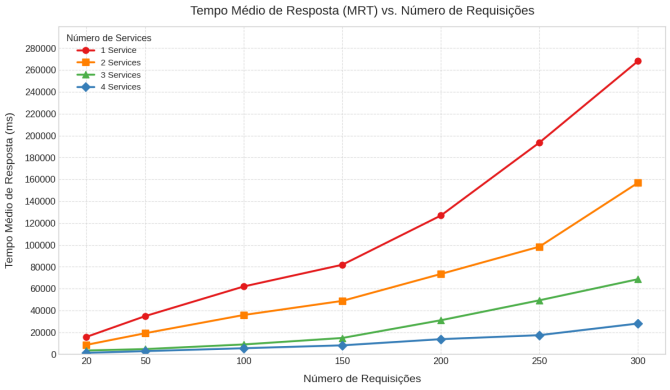

Fig. 2. Tempo Médio de Resposta (MRT) vs. Número de Requisições para diferentes configurações de pipeline de serviços.

Este aumento inicial é atribuído ao tempo de processamento acumulado em cada serviço, incluindo a custosa inferência de IA, e à latência de rede (trânsito) entre os múltiplos nós (contêineres).

- **Impacto da Carga (Número de Requisições):** Para todas as configurações, o MRT aumenta à medida que o número de requisições cresce. Este comportamento indica que o sistema começa a atingir pontos de saturação, onde as requisições passam mais tempo em filas de espera nos Load Balancers ou nos próprios Services antes de serem processadas.
- **Escalabilidade, Paralelismo e Gargalos:** A observação mais interessante é a *taxa de crescimento* do MRT para cada cenário.
  - As curvas vermelha (1 serviço) e laranja (2 serviços) mostram um crescimento exponencial acentuado. Nesses cenários, o processamento é largamente serial. Com apenas uma instância de serviço (ou duas em sequência), não há paralelismo para lidar com a carga crescente. O serviço único (ou o primeiro da sequência) torna-se um gargalo claro, e sua fila de espera aumenta drasticamente com mais requisições, resultando em uma explosão do MRT.
  - Em contraste, a curva azul (4 serviços) e, em menor grau, a verde (3 serviços) exibem um crescimento muito mais contido, quase linear. Embora a pipeline de 4 serviços seja a mais longa e tenha o maior MRT base, sua escalabilidade sob carga é superior. Isso se deve diretamente à eficácia dos Load Balancers. Nos estágios de serviço S1 e S2, a carga é distribuída entre duas instâncias de serviço cada (‘S1.1‘/‘S1.2‘ e ‘S2.1‘/‘S2.2‘). Esse paralelismo permite que o sistema processe múltiplas requisições simultaneamente, mitigando a formação de longas filas e mantendo o MRT sob controle, mesmo com um volume elevado de requisições.

Em suma, os resultados demonstram um trade-off clássico em sistemas distribuídos: adicionar mais etapas a uma pipeline aumenta a latência base de uma requisição individual, mas a introdução de paralelismo através de balanceamento de carga nesses estágios pode aumentar drasticamente a vazão e a escalabilidade do sistema como um todo. O experimento valida empiricamente que, para sistemas que enfrentam cargas variáveis e pesadas, uma arquitetura paralela, mesmo que mais complexa, é fundamental para manter um desempenho aceitável. A análise dos tempos intermediários (T1-T5), embora não detalhada em gráficos, confirmou que os maiores tempos de espera (filas) ocorriam nos nós de serviço nos cenários com menos paralelismo (1 e 2 serviços).

C. Análise dos Tempos Intermediários

Além do MRT, a análise dos tempos intermediários (T1 a T5) revelou padrões interessantes. Observou-se que a maior parte da latência adicional em cenários com múltiplos serviços está concentrada nos nós de IA, especialmente quando o volume de requisições é elevado. Isso reforça a necessidade de otimização e paralelização desses componentes em sistemas reais.

D. Ameaças à Validade

Entre as principais ameaças à validade dos resultados, destacam-se possíveis variações de desempenho do host durante os experimentos e a ausência de falhas reais de rede. Para mitigar esses fatores, os experimentos foram realizados em horários de baixa utilização da máquina e com monitoramento constante dos recursos.

E. Comparação com Trabalhos Relacionados

Comparando os resultados obtidos com estudos prévios, nota-se que a introdução de paralelismo e balanceamento de carga é uma estratégia recorrente para melhorar a escalabilidade de sistemas distribuídos. No entanto, a integração de serviços de IA impõe desafios adicionais, exigindo abordagens específicas para otimização de desempenho.

VI. Conclusão e Trabalhos Futuros

A. Conclusão

Este trabalho apresentou com sucesso a implementação em Python e a análise de desempenho de um sistema distribuído inspirado no PASID-VALIDATOR. Utilizando Docker e Docker Compose, construímos uma plataforma robusta e reproduzível para simular fluxos de trabalho cliente-servidor, incluindo um serviço de IA computacionalmente intensivo com um modelo BERT.

Os experimentos realizados demonstraram claramente a relação entre a complexidade da arquitetura, a carga de trabalho e o tempo médio de resposta. A principal descoberta foi a confirmação empírica da importância do paralelismo e do balanceamento de carga para a escalabilidade do sistema. Enquanto pipelines mais simples e seriais sofrem uma degradação exponencial do desempenho sob carga, arquiteturas com múltiplos serviços paralelos, gerenciados por balanceadores de carga, exibem uma escalabilidade muito superior, mantendo o MRT mais estável mesmo com o aumento do número de requisições. A ferramenta desenvolvida provou ser eficaz para coletar métricas de desempenho detalhadas e obter insights valiosos sobre o comportamento de sistemas distribuídos.

B. Trabalhos Futuros

As bases estabelecidas neste trabalho abrem diversas direções para pesquisas futuras. As propostas incluem:

- Integração de outros modelos de balanceamento de carga (e.g., least connections, weighted round robin) para comparar sua eficácia.
- Implementação de persistência de dados para os resultados dos experimentos em um banco de dados.
- Adição de mais nós ou exploração de topologias de rede mais complexas.
- Geração de tráfego mais realista, com distribuições de chegada de requisições (e.g., Poisson) e tamanhos de payload variáveis.
- Uso de bibliotecas de rede assíncronas (e.g., ‘asyncio‘) em Python para investigar melhorias de desempenho e escalabilidade nos componentes, especialmente no Source e nos Load Balancers.
- Automação da análise de resultados e geração de relatórios.
- Desenvolvimento de uma interface de usuário para facilitar a configuração dos experimentos e a visualização dos resultados em tempo real.
- Validação formal dos resultados experimentais com modelos teóricos, como Stochastic Petri Nets ou teoria de filas, fechando o ciclo proposto pelo conceito original do PASID-VALIDATOR.

C. Implicações Práticas

Os resultados deste trabalho têm implicações diretas para o projeto de sistemas distribuídos modernos, especialmente em ambientes que demandam alta escalabilidade e integração de serviços de IA. A abordagem proposta pode ser adaptada para diferentes domínios, como processamento de dados em tempo real, sistemas financeiros e aplicações de saúde.

D. Sugestões para Pesquisas Futuras

Além das propostas já mencionadas, sugerimos a investigação do impacto de diferentes estratégias de escalonamento de contêineres, a integração com plataformas de monitoramento em tempo real e a avaliação do sistema em ambientes de nuvem pública. Outra linha promissora é a aplicação de técnicas de aprendizado de máquina para otimizar dinamicamente o balanceamento de carga e a alocação de recursos.

E. Fundamentação Teórica

A avaliação de desempenho em sistemas distribuídos é um campo consolidado, envolvendo conceitos como latência, throughput, escalabilidade, disponibilidade e tolerância a falhas. Segundo Tanenbaum e Van Steen [**?**], a escalabilidade é um dos maiores desafios, pois sistemas distribuídos devem crescer em número de nós e volume de requisições sem perda significativa de desempenho. Técnicas de balanceamento de carga, replicação e particionamento de dados são amplamente estudadas para mitigar gargalos e aumentar a resiliência do sistema.

Além disso, a instrumentação para coleta de métricas, como timestamps intermediários, é uma prática recomendada em ambientes de produção para identificar gargalos e otimizar fluxos de trabalho. Ferramentas como Prometheus, Grafana e Jaeger são frequentemente utilizadas para monitoramento contínuo, enquanto abordagens acadêmicas, como o uso de Petri Nets, permitem modelar e prever o comportamento do sistema sob diferentes condições. O uso de métricas como o Tempo Médio de Resposta (MRT) é fundamental para validar modelos teóricos e garantir que o sistema atenda aos requisitos de desempenho estabelecidos.

F. Desafios de Implementação

Durante o desenvolvimento do PASID-VALIDATOR em Python, enfrentamos desafios relacionados à sincronização de relógios entre contêineres, serialização eficiente de mensagens e gerenciamento de conexões TCP. A escolha do formato JSON para serialização foi motivada pela sua simplicidade e ampla adoção, mas alternativas como Protocol Buffers poderiam oferecer maior desempenho em cenários de alta demanda.

Outro desafio foi a integração do modelo BERT, que exige considerável memória e processamento. Para mitigar o impacto, optamos por carregar o modelo uma única vez por contêiner e reutilizá-lo em todas as requisições, reduzindo o overhead de inicialização.

G. Protocolos de Comunicação e Tolerância a Falhas

A comunicação entre os nós utiliza sockets TCP, garantindo entrega confiável das mensagens. Implementamos mecanismos de timeout e retransmissão para lidar com possíveis falhas de rede. Além disso, cada nó registra logs detalhados de eventos, facilitando a identificação de falhas e a análise pós-experimento.

H. Fluxo de Dados e Comunicação

O fluxo de dados no sistema segue uma trajetória bem definida, desde a geração da requisição no Source até o retorno do resultado processado. Cada mensagem trafega por múltiplos nós, sendo enriquecida com informações de timestamp em cada etapa. A comunicação é realizada de forma síncrona, garantindo que cada nó aguarde a resposta do próximo antes de prosseguir, o que facilita a rastreabilidade e a análise posterior dos tempos de processamento.

I. Gerenciamento de Estado e Logs

Cada componente mantém um registro detalhado das requisições processadas, armazenando localmente os logs de eventos e os tempos de cada etapa. Essa abordagem facilita a auditoria e a depuração, além de permitir a análise posterior dos dados coletados. Os logs são estruturados em formato JSON, facilitando sua integração com ferramentas de análise e visualização.

J. Escalabilidade e Modularidade

A arquitetura modular do sistema permite a fácil adição ou remoção de nós, bem como a alteração das políticas de balanceamento de carga. Essa flexibilidade é essencial para a realização de experimentos comparativos e para a adaptação do sistema a diferentes cenários de uso.

K. Parâmetros de Configuração

Além das variáveis principais, outros parâmetros foram cuidadosamente ajustados para garantir a reprodutibilidade dos experimentos. Entre eles, destacam-se o tamanho do payload das mensagens, o intervalo entre o envio de requisições e a configuração dos recursos alocados para cada contêiner (CPU, memória). Todos os experimentos foram repetidos múltiplas vezes para garantir a robustez dos resultados, e os valores apresentados correspondem à média das execuções.

L. Limitações e Considerações Éticas

Apesar dos resultados promissores, este trabalho apresenta algumas limitações. Os experimentos foram realizados em ambiente controlado, com todos os contêineres executando em uma única máquina física, o que pode não refletir integralmente os desafios de ambientes distribuídos reais, como latências de rede geograficamente dispersas e falhas intermitentes. Além disso, o uso de modelos de IA demanda atenção especial à privacidade dos dados processados, sendo fundamental garantir a conformidade com legislações como a LGPD e o GDPR em aplicações práticas.

REFERENCES

[1] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley Professional Computing, 1991.

[2] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep. dapper-2010-1, April 2010.

[3] D. L. L. R. Martínez, T. M. Brizuela, F. Agostini, J. T. F. Martínez, and J. Acosta, "Balanceo inteligente de carga de trabajo en sistemas distribuidos heterogéneos," *Brazilian Journal of Development*, vol. 11, no. 1, pp. e76 681–e76 681, 2025.