INSTITUTO FEDERAL DE SANTA CATARINA

ARTHUR ANASTOPULOS DOS SANTOS

Análise de possibilidades de balanceamento de carga em SIP

São José - SC

agosto/2024

ANÁLISE DE POSSIBILIDADES DE BALANCEAMENTO DE CARGA EM SIP

Monografia submetida à Coordenação de Engenharia de Telecomunicações do Instituto Federal de Santa Catarina para a obtenção do diploma Bacharel em Engenharia de Telecomunicações.

Orientador: Prof. Ederson Torresini, Me.

São José - SC

agosto/2024

Arthur Anastopulos dos Santos

Análise de possibilidades de balanceamento de carga em SIP

Este trabalho foi julgado adequado para obtenção do título de Engenheiro de Telecomunicações, pelo Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina, e aprovado na sua forma final pela comissão avaliadora abaixo indicada.

São José - SC, 28 de Agosto de 2024.

Prof. Ederson Torresini, Me.
Orientador
Instituto Federal de Santa Catarina

Prof. Cleber Jorge Amaral, Dr.
Convidado
Instituto Federal de Santa Catarina

Prof. Eraldo Silveira e Silva, Dr.
Convidado
Instituto Federal de Santa Catarina

RESUMO

Este estudo investigou os efeitos das técnicas de balanceamento de carga em ambientes *SIP*, visando otimizar a qualidade das comunicações em tempo real. Para tal, foram definidos objetivos claros e selecionados ambientes de testes representativos. Foram escolhidas alguns dos diversos algoritmos, como *round-robin* e *least-connection*, para serem implementadas e avaliadas em cenários realistas de implementação. Os dados coletados abrangeram métricas, como latência, perda de pacotes e utilização de recursos. A análise estatística revelou possibilidades de implementação, demonstrando o uso das estratégias de balanceamento de carga, em um cenário com *SIP*. Os resultados demonstram como esses algoritmos garantem a consistência das comunicações simultâneas de mídia em ambientes dinâmicos e congestionados.

Palavras-chave: Balanceamento de carga, *SIP*.

ABSTRACT

This study investigated the effects of load balancing techniques in *SIP* environments, aiming to optimize the quality of communications in real time. To this end, clear objectives were defined and representative test environments were selected. Some of the different algorithms were chosen, such as *round-robin* and *least-connection*, to be implemented and evaluated in realistic implementation scenarios. The data collected covered metrics such as latency, packet loss and resource utilization. The statistical analysis revealed implementation possibilities, demonstrating the use of load balancing strategies, in a scenario with *SIP*. The results demonstrate how these algorithms ensure consistency of media communications in dynamic and congested environments.

Keywords: Load balancing, *SIP*.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

DNS Sistema de Nomes de Domínio.

DTMD *Dual tone multi-frequency*.

HTTP Protocolo de Transferência de Hipertexto.

IP Protocolo de Internet.

IPVS *IP Virtual Server*.

PID Identificador de processo.

QOS Qualidade de Serviço.

RFC Solicitação de comentários.

RTP Protocolo de Transporte em Tempo Real.

SDP Protocolo de descrição de sessão.

SIP *Protocolo de Iniciação de Sessão*.

SPOF Ponto único de falha.

SSL *Secure Sockets Layer*.

TLS *Transport Layer Security*.

UA Agente de usuário.

UAC Agente de usuário Cliente.

UAS Agente de usuário Servidor.

UDP Protocolo de datagrama do usuário.

URI Identificador Uniforme de Recursos.

VoIP Voz sobre IP.

WebRTC Comunicações na Web em tempo real.

SUMÁRIO

1 INTRODUÇÃO

A crescente demanda por comunicações simultâneas e a evolução das tecnologias de rede vêm estimulando a utilização de protocolos, como o *Protocolo de Iniciação de Sessão* (SIP) e Comunicações na Web em tempo real (WebRTC) para estabelecer comunicações de voz, vídeo e dados pela Internet. Segundo dados da CNN Brasil (Denise Ribeiro; Anthony Wells, 2023), durante o primeiro semestre de 2021, o uso de aplicativos móveis para videoconferências registrou um aumento significativo de 150%, evidenciando a crescente demanda por ferramentas de comunicação simultânea em resposta às novas dinâmicas de trabalho e interação à distância. Contudo, assegurar uma experiência de comunicação contínua e confiável por meios desses protocolos torna-se um desafio em cenários de redes extremamente dinâmicas e sobrecarregadas.

O SIP, que é o padrão para estabelecer, modificar e encerrar sessões de comunicação, desde chamadas de voz até videoconferências, oferecendo uma infraestrutura versátil para conectar indivíduos e sistemas.

No cenário atual, onde testemunhamos um crescimento exponencial e verdadeiramente gritante no número de usuários ativos na internet. De acordo com o estudo de Kemp (2022), o número de indivíduos que acessam a rede regularmente se aproximou da marca de 5 bilhões de pessoas em janeiro de 2022. Esse marco representa quase 63% da população global. Esse crescimento meteórico na conectividade tem sido impulsionado por avanços tecnológicos e pela disponibilidade generalizada da internet, permitindo que as pessoas em todo o mundo se comuniquem, colaborem e acessem informações em tempo real.

Apesar dos avanços tecnológicos, o desafio de preservar a qualidade e eficiência dessas comunicações em um cenário de rede dinâmico e frequentemente congestionado através desses protocolos não é uma tarefa trivial. A complexidade existente às comunicações em tempo real se manifesta através de uma série de desafios. Como visto em Loureiro et al. (2023), a diversidade de dispositivos, variando desde *smartphones* a computadores, introduz variações na capacidade de processamento e na qualidade de conexão. Além disso, a oscilação das condições de rede, incluindo atrasos, perda de pacotes e flutuações de largura de banda, pode prejudicar a qualidade das chamadas e comprometer a experiência ao usuário.

Nesse contexto de redes altamente dinâmicas e frequentemente sobrecarregadas, a necessidade de abordagens que garantam a qualidade e a escalabilidade das comunicações se torna ainda mais urgentes. Surge então a exploração de estratégias de balanceamento de carga. Este estudo se concentra, segundo Neghabi et al. (2018), em distribuir eficien-

temente a carga de trabalho entre diversos recursos, otimizando a utilização dos mesmos disponíveis e minimizando os gargalos de desempenho em qualquer um dos recursos.

O escopo deste trabalho se aplica à análise de viabilidade de testes das técnicas de balanceamento de carga em aplicações usando os protocolo SIP. Através da análise crítica dessas técnicas, buscas-se entender como diferentes estratégias podem influenciar diretamente na qualidade das comunicações e na capacidade de expansão destas aplicações em ambientes altamente dinâmicos e potencialmente congestionados.

1.1 Justificativa do tema

A comunicação simultânea por meio de protocolos, como SIP, está se tornando cada vez mais vantajosas para uma variedade de aplicações, incluindo chamadas de voz, videoconferências e até compartilhamento de dados em tempo real. À medida que o mundo se torna cada vez interconectado, a demanda por essas formas de comunicação só tende a crescer, tornando-as fundamentais para a maneira como as pessoas trabalham, interagem e se comunicam.

No entanto, a qualidade dessas comunicações é frequentemente comprometida por desafios inerentes a ambientes de rede dinâmicos e sobrecarregados, como latência, perda de pacotes e congestionamento de rede. Além disso, é crucial considerar os pontos de gargalo e Pontos únicos de falha (SPOFs), que podem surgir em infraestruturas de comunicação complexas.

A justificativa para este estudo sobre o balanceamento de carga em ambientes SIP se baseia na importância crítica de otimizar a experiência do usuário e garantir a confiabilidade contínua desses sistemas vitais. Ao entender profundamente os efeitos das diferentes técnicas de balanceamento de carga, podemos tomar medidas concretas para melhorar a qualidade das comunicações em tempo real, mesmo em condições adversas de rede.

Além da necessidade premente de atualização tecnológica descrita anteriormente, é crucial ressaltar a crescente demanda por sistemas de SIP, particularmente nas empresas estabelecidas na região da Grande Florianópolis. Empresas proeminentes como Khomp[1], Intelbras[2] e Digitro[3] estão enfrentando essa demanda devido à evolução contínua das comunicações digitais e à busca por soluções eficientes e seguras. Essa realidade destaca a importância não apenas da modernização dos sistemas de comunicação, mas também da adaptação às exigências específicas do mercado local, visando manter a competitividade e atender às necessidades dos clientes com eficácia.

[1] https://www.khomp.com/
[2] https://www.intelbras.com/
[3] https://digitro.com/

Portanto, o estudo proposto visa ampliar o conhecimento da área, contribuindo para o avanço das estratégias de balanceamento de carga em ambientes com SIP. Ao fazê-lo, busca-se garantir que essas tecnologias continuem a oferecer uma experiência de comunicação de alta qualidade, promovendo a eficácia das comunicações em tempo real em um mundo cada vez mais conectado e dependente delas.

1.2 Objetivos

Este estudo tem como objetivo geral realizar testes das técnicas de balanceamento de carga no desempenho de aplicações que utiliza o protocolo SIP. Pretende-se compreender como essas técnicas influenciam a qualidade das comunicações em tempo real e a capacidade de expansão destas aplicações em um ambiente de rede dinâmico.

1.2.1 Objetivos específicos

1. Detalhar as técnicas de balanceamento de carga mais utilizadas em aplicações com o uso de SIP, identificando suas características e modos de implementação.

2. Avaliar o desempenho entre técnicas de balanceamento de carga escolhidas em termos de latência, perda de pacotes e capacidade de escalabilidade.

1.3 Organização do texto

O texto está organizado da seguinte forma: no Capítulo 2 é apresentado a fundamentação teórica dando uma visão ampla do protocolo SIP e das múltiplas ferramentas escolhidas para o desenvolvimento do trabalho. Capítulo 3 é apresentado o desenvolvimento do trabalho. Capítulo 4 é apresentado a implementação e testes da proposta. Por fim, Capítulo 5 são apresentadas as conclusões sobre este trabalho.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo, abordaremos a fundamentação teórica que serve como base para a compreensão do tema central deste trabalho. Para uma compreensão mais aprofundada, faz-se necessário explorar conceitos fundamentais relacionados como: compreensão dos protocolos e tecnologias principais utilizadas, bem como a interconexão entre essas tecnologias, considerando o contexto em que são utilizadas e a necessidade de balanceamento de carga.

2.1 Balanceamento de Carga

A prática do balanceamento de carga desempenha um papel crítico no campo das redes de comunicação, incluindo aplicações com SIP. Conforme definido por Belgaum et al. (2020), o balanceamento de carga é uma metodologia que visa alocar eficientemente a carga de rede entre diferentes componentes, resultando na otimização do desempenho da rede e na melhoria da Qualidade de Serviço (QOS) para os usuários finais. Isso é alcançado por meio da aplicação de diversas técnicas, estratégias e algoritmos de balanceamento de carga. Tais métodos auxiliam tanto os usuários finais quanto os provedores de serviços na alocação e redistribuição da carga, com o intuito de otimizar a eficiência da rede.

Uma das vantagens notáveis do balanceamento de carga, conforme destacado por Belgaum et al. (2020), reside na sua capacidade de antecipar e prever gargalos no tráfego de rede antes que ocorram, agindo como "polícia de controle de tráfego de rede". Por meio dessa estratégia, a rede pode ser gerenciada de maneira mais eficaz e assegurar que o tráfego seja distribuído de maneira equitativa e eficiente.

De acordo com Neghabi et al. (2018), o balanceamento de carga é uma técnica que divide a carga de trabalho entre vários recursos da rede, impedindo que qualquer recurso específico seja sobrecarregado. Esse equilíbrio na alocação de carga auxilia na manutenção da estabilidade da rede e na garantia de que os recursos permaneçam operacionais, contribuindo, assim para a confiabilidade e a alta disponibilidade do sistema.

Portanto, é evidente que o balanceamento de carga desempenha um papel essencial na otimização do desempenho das redes de comunicação, garantindo a eficiência a qualidade dos serviços oferecidos.

2.1.1 Importância do Balanceamento de Carga

A importância do balanceamento de carga é evidente ao considerarmos os diversos parâmetros de medição que podem ser utilizados para avaliar a eficácia das técnicas e algoritmos envolvidos, conforme ressaltado por Sajjan e Biradar (2017). Esses parâmetros desempenham um papel importante na determinação de quão eficaz o balanceamento de carga está desempenhando seu papel na rede. Abaixo, destacamos esses parâmetros e como eles influenciam a importância do balanceamento de carga:

- Taxa de Transferência: Definida por Sajjan e Biradar (2017) como, a quantidade de trabalho que pode ser realizada em um determinado período de tempo. O balanceamento de carga é vital para garantir que a rede mantenha uma taxa de transferência ideal, permitindo que o sistema lide eficientemente com as demandas de tráfego variáveis. O aumento da taxa de transferência resulta em um desempenho aprimorado.

- Tolerância a Falhas: A tolerâncias a falhas, como mencionado por Sajjan e Biradar (2017), refere-se à capacidade do algoritmo de balanceamento de carga de permitir que o sistema funcione mesmo em condições falha. Isso garante alta disponibilidade e confiabilidade, pois o balanceamento de carga pode redirecionar o tráfego para servidores funcionais, mesmo quando ocorrem falhas em componentes da rede.

- Escalabilidade: Está é indispensável para sistemas que enfrentam flutuações na demanda de tráfego. Um algoritmo de balanceamento de carga eficaz, como observado por Sajjan e Biradar (2017), é capaz de dimensionar-se conforme necessário, acomodando um aumento ou diminuição na carga de trabalho. Isso permite que a rede mantenha um desempenho consistente independentemente das mudanças nas condições.

- Desempenho: O desempenho geral da rede, considerando precisão, custo e velocidade, é diretamente impactado pelo balanceamento de carga. Um algoritmo eficaz equilibra a carga de maneira que resulte em um desempenho otimizado, garantindo que os recursos da rede sejam utilizados de forma eficiente e econômica.

- Utilização de Recursos: O controle de utilização de recursos é fundamental para otimizar a eficiência da rede. Conforme apontado por Sajjan e Biradar (2017), o balanceamento de carga ajuda a distribuir a carga de trabalho entre os recursos disponíveis de maneira uniforme, impedindo que qualquer recurso específico seja sobrecarregado. Isso garante a utilização eficaz dos recursos e evita desperdícios.

Dessa forma, a importância do balanceamento de carga na rede é ressaltada ao considerar sua capacidade de otimizar a taxa de transferência, reduzir o tempo de resposta, garantir tolerância a falhas escalar conforme necessário, melhorar o desempenho

e controlar a utilização de recursos. Esse parâmetros de medição demonstram que o balanceamento de carga desempenha um papel crítico na manutenção da eficiência e da qualidade da rede, proporcionando uma experiência superior aos usuários.

2.1.2 Algoritmos de Balanceamento de Carga

Os algoritmos de balanceamento de carga constituem uma etapa primordial para otimizar o desempenho das redes, sendo responsáveis pela alocação eficiente de tarefas e recursos, garantindo um uso equitativo e eficaz dos componentes de rede disponíveis.

Segundo Mustafa (2017), os algoritmos de balanceamento de carga em um comutador web tem grande importância para aumentar o desempenho do *cluster*. Quando uma nova solicitação chega, o algoritmo escolhe o servidor mais adequado e atribui a solicitação a ele. Os algoritmos de balanceamento de carga operam com base no princípio de que a carga de trabalho pode ser atribuída durante o tempo de compilação ou em tempo de execução, dependendo da situação.

A seguir, serão descritos alguns dos algoritmos de balanceamento de carga mais utilizados em aplicações de comunicações simultâneas. Esses algoritmos são essenciais para garantir a eficiência e a equidade na distribuição de tarefas e recursos entre os diversos componentes de uma rede. Entre os mais comuns, destacam-se o *Round-Robin*, o *Least-Connection*, cada um com suas características e vantagens específicas que contribuem para a otimização do desempenho de *clusters* e servidores.

2.1.3 Algoritmo Round-Robin

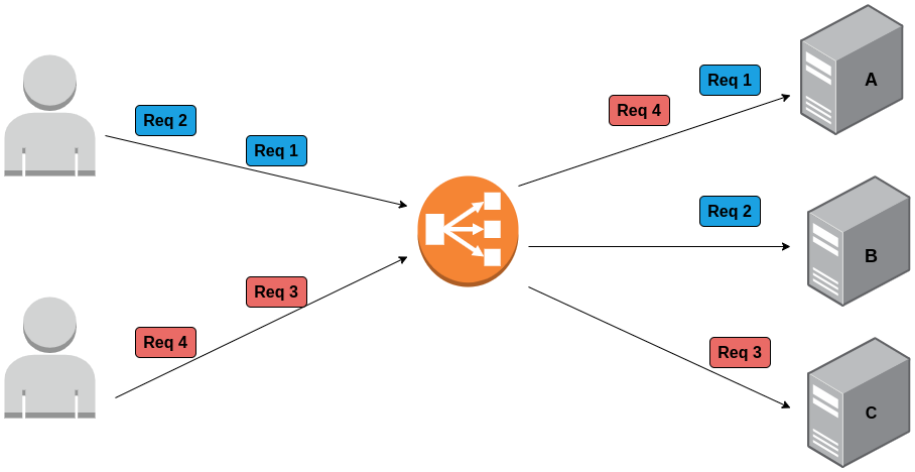
O algoritmo *Round-Robin*, conforme descrito por Sajjan e Biradar (2017), utiliza uma abordagem em que um tempo fixo é alocado para cada tarefa, distribuindo-as de forma circular entre os nós da rede. Essa técnica assegura que todos os processadores recebam tarefas em uma ordem cíclica, prevenindo o problema de inanição, em que nenhum nó fica inativo por longos períodos. Assim, promove uma resposta mais rápida em cenários que exigem uma distribuição equitativa da carga de trabalho. No entanto, como destacado por Mustafa (2017), esse método pode resultar em desequilíbrios na carga, com alguns nós podendo ficar sobrecarregados enquanto outros permanecem subutilizados.

Segundo Mustafa (2017), o algoritmo de escalonamento *Round-Robin* encaminha cada solicitação para o próximo servidor na lista. Por exemplo, em um *cluster* com três servidores (A, B e C), a primeira solicitação é direcionada ao servidor A, a segunda ao servidor B, a terceira ao servidor C e a quarta novamente ao servidor A, completando o ciclo dos servidores, conforme ilustrado na Figura 1.

Embora o algoritmo trate todos os servidores de maneira igual, independentemente do número de conexões ou do tempo de resposta, sua granularidade de escalonamento ba-

seada no nó pode levar a desequilíbrios dinâmicos de carga entre os servidores, resultando em sobrecarga para alguns e ociosidade para outros.

Figura 1 – Algoritmo *Round-Robin*

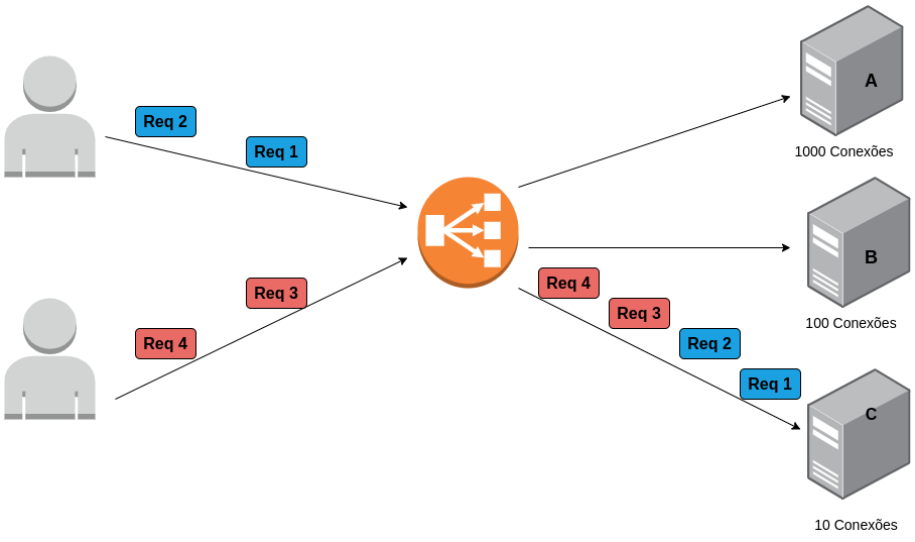


Fonte: Próprio Autor

2.1.4 Algoritmo Least-Connection

O algoritmo de escalonamento *Least-Connection* é uma técnica eficiente para a distribuição de conexões de rede entre servidores. Ao contrário de outros métodos que podem basear sua distribuição em critérios estáticos ou predefinidos, o *Least-Connection* direciona novas conexões para o servidor que possui o menor número de conexões estabelecidas no momento, como podemos ver na figura 2. Esse processo é dinâmico, pois requer a contagem em tempo real das conexões ativas para cada servidor (MUSTAFA, 2017).

Figura 2 – Algoritmo *Least-Connection*



Fonte: Próprio Autor

Este algoritmo é particularmente vantajoso em cenários onde a carga de solicitações pode variar significativamente. A abordagem de *Least-Connection* distribui as novas solicitações de forma mais uniforme entre os servidores, ao identificar e direcionar as conexões para aqueles que estão menos carregados. Isso é possível porque o algoritmo mantém um controle contínuo das conexões ativas através de uma tabela de *IP Virtual Server* (IPVS), o que permite ajustar a distribuição com base nas condições atuais do sistema.

A técnica é ideal para ambientes onde todos os servidores possuem capacidades semelhantes, uma vez que presume que todos têm a mesma capacidade de processamento. Se os servidores em um grupo têm capacidades diferentes, o desempenho do sistema pode não ser ideal. Nesses casos, o escalonamento ponderado por *Least-Connection*, que leva em consideração as diferentes capacidades dos servidores, pode ser uma alternativa mais apropriada (MUSTAFA, 2017).

Embora este seja uma opção versátil e flexível no navegador, é essencial reconhecer que não substitui o HTTP. Cada Transporte possui suas próprias forças e melhores casos de uso. Aplicações que utilizam protocolos personalizados devem gerenciar aspectos como controle de estados compressão e cache, normalmente providos pelos navegadores.

2.2 Proxy Reverso

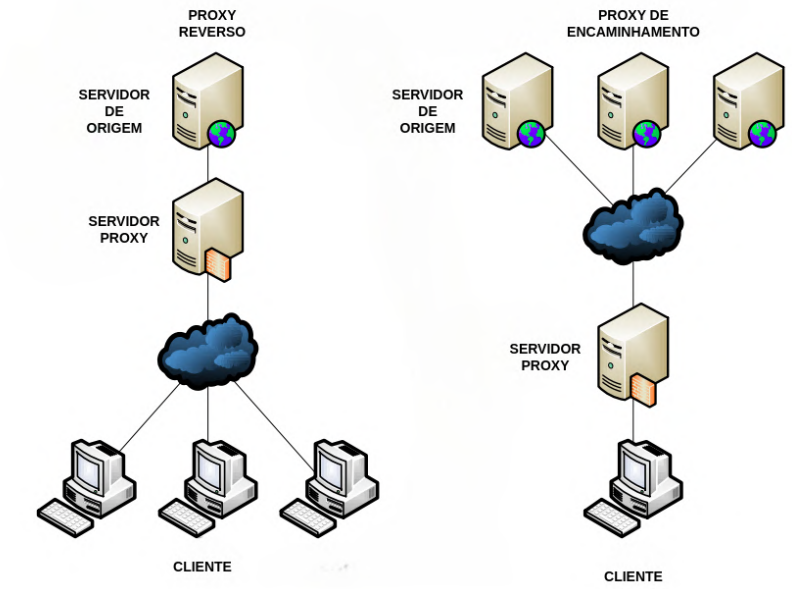
O *Proxy* Reverso desempenha um papel crucial na arquitetura de redes, atuando como intermediário entre clientes e servidores de destino. Conforme explicado por Dély (2014), um servidor *proxy* pode ser tanto um serviço em si quanto um *host* físico que fornece esse serviço. Existem dois principais de servidores *proxy*: o *proxy* de encaminhamento e o *proxy* reverso.

O *proxy* de encaminhamento intervém no tráfego entre o cliente e o destino escolhido pelo cliente. Em contraste, o *proxy* reverso adota uma abordagem diferente, apresentando ser o servidor de origem para o s clientes conectados. Ele encaminha as solicitações dos clientes para o servidor de origem e, em seguida, repassa o conteúdo ou informações do servidor de volta aos clientes.

No contexto do balanceamento de carga, o *proxy* Reverso desempenha um papel significativo. Ao contrário do *proxy* de encaminhamento, que age como um intermediário direto entre o cliente e o servidor de destino, o *proxy* reverso tem a capacidade de distribuir a carga de maneira eficiente entre vários servidores de origem. Ele gerencia as solicitações dos clientes encaminhando-as para diferentes servidores, otimizando assim o desempenho e garantindo a disponibilidade do serviço.

Além disso, o *proxy* reverso pode ser implementado em diferentes camadas da

Figura 3 – Diferenças entre um *proxy* reverso (à esquerda) e um *proxy* de encaminhamento (à direita)



Fonte: Adaptado pelo autor a partir de Dély (2014).

aplicação, desde a camada de transporte até a camada de aplicação, permitindo uma flexibilidade maior na gestão do tráfego e na distribuição de carga.

As vantagens do *proxy* reverso incluem a melhoria da segurança, pois oculta a estrutura interna dos servidores de origem, e a capacidade de realizar o cache de conteúdo reduzindo assim a carga nos servidores e melhorando a velocidade de resposta. No entanto, é importante considerar que o uso inadequado do *proxy* reverso pode resultar em desafios de configuração e complexidade adicional na gestão da infraestrutura.

2.3 Cache Web

O uso de cache web é uma estratégia eficaz para aprimorar o desempenho na web, conforme destacado por Nottingham (2013). O cache web atua como intermediário entre um servidor de origem e um ou mais clientes, armazenando temporariamente as respostas enviadas pelo servidor. Esse armazenamento temporário permite que as respostas sejam servidas diretamente quando solicitadas novamente, sem a necessidade de envolver o servidor de origem.

Nottingham (2013) menciona que os navegadores frequentemente realizam o cache de conteúdo baixado durante a navegação de usuários na web. Essa prática evita o download repetido do mesmo material, uma vez que o navegador salva os arquivos em cache. Além disso, o cache web pode ser implementado por meio de *proxies* reversos, os quais são capazes de armazenar em cache as respostas dos servidores de origem. Essa

abordagem melhora o tempo de resposta do site e a quantidade de solicitações que podem ser tratadas simultaneamente (NOTTINGHAM, 2013).

A utilização eficiente do cache web desempenha um papel crucial no balanceamento de carga. Ao armazenar temporariamente as respostas frequentemente solicitadas, o cache reduz a carga nos servidores de origem, otimizando assim o tempo de resposta e a capacidade de lidar com um maior volume de solicitações simultâneas. Essa prática contribui para a eficiência do sistema, melhorando a experiencia do usuários e garantido a disponibilidade de conteúdo.

Apesar das vantagens proporcionadas pelo cache web, é importante considerar que ele pode resultar em desafios, como a necessidade de gerenciar o tempo de vida útil do cache e lidar com questões relacionadas à consistência dos dados armazenados. No entanto, quando implementado corretamente, o cache web é uma ferramenta valiosa para otimizar o desempenho e a eficiência operacional em ambientes online.

2.4 Nginx

O *Nginx* é um servidor *web* e *proxy* reverso de código aberto que também atua como um balanceador de carga, *gateway* de *e-mail* e cache *HTTP*, amplamente utilizado em aplicações de alta performance. O mesmo foi projetado para lidar com o problema de desempenho e escalabilidade associado ao processamento de múltiplas conexões simultâneas, superando as limitações dos servidores web tradicionais. (Nginx, Inc., 2024)

O *Nginx* utiliza uma arquitetura assíncrona e baseada em eventos, permitindo que ele processe milhares de requisições simultâneas sem a necessidade de alocar um novo processo para cada conexão, como fazem servidores baseados em processos como o *Apache*. Isso o torna extremamente eficiente no uso de recursos de *CPU* e memória, ideal para ambientes que exigem alta disponibilidade e escalabilidade.

Além de seu desempenho como servidor *web*, o *Nginx* é amplamente utilizado como um *proxy* reverso. Nesse papel, ele atua como intermediário entre os clientes e os servidores *backend*, direcionando requisições de entrada para diferentes servidores, de acordo com a carga de trabalho ou a necessidade de balanceamento de tráfego. Essa funcionalidade é utilizada em sistemas distribuídos que exigem distribuição eficiente de tráfego e tolerância a falhas.

O *Nginx* também oferece suporte a balanceamento de carga, permitindo a distribuição de requisições de rede entre múltiplos servidores *backend*. Ele suporta diversos algoritmos de balanceamento, como o *round-robin* e o *least-connections*, que são usados para otimizar o uso de recursos em *clusters* de servidores. Outra funcionalidade importante é o suporte a *SSL/TLS*, o que permite a implementação de comunicação segura,

sendo amplamente utilizado para gerenciar o tráfego *HTTPS* em grandes escalas.

O *Nginx* é altamente configurável e modular, permitindo a adição de módulos para ampliar suas funcionalidades, sem comprometer a eficiência ou a simplicidade de sua arquitetura. Graças à sua flexibilidade, é utilizado em uma ampla gama de aplicações, desde servidores web básicos até infraestrutura complexa de microsserviços e plataformas em nuvem.

2.5 SIP (Session Initiation Protocol)

O *Protocolo de Iniciação de Sessão* (SIP), conforme definido pelo RFC 3261 (SCHOOLER et al., 2002), desempenha um papel crucial na facilitação de aplicações na internet que demandam a criação e gestão de sessões entre participantes. Uma sessão, nesse contexto, refere-se a uma troca de dados entre uma associação de participantes, podendo envolver diversas formas de dados em tempo real, como voz, vídeo ou mensagens de texto.

Este atua em conjunto com vários protocolos que transportam dados de sessão multimídia em tempo real, permitindo que pontos finais da internet, conhecidos como agentes de usuários, descubram uns aos outros e concordem com a caracterização de uma sessão que desejam compartilhar. Isso é particularmente útil em cenários onde os usuários podem se movimentar entre diferentes pontos finais, ter múltiplos endereços e se comunicar em diversos meios simultaneamente.

O protocolo suporta quatro principais recursos para estabelecer e encerrar sessões multimídia:

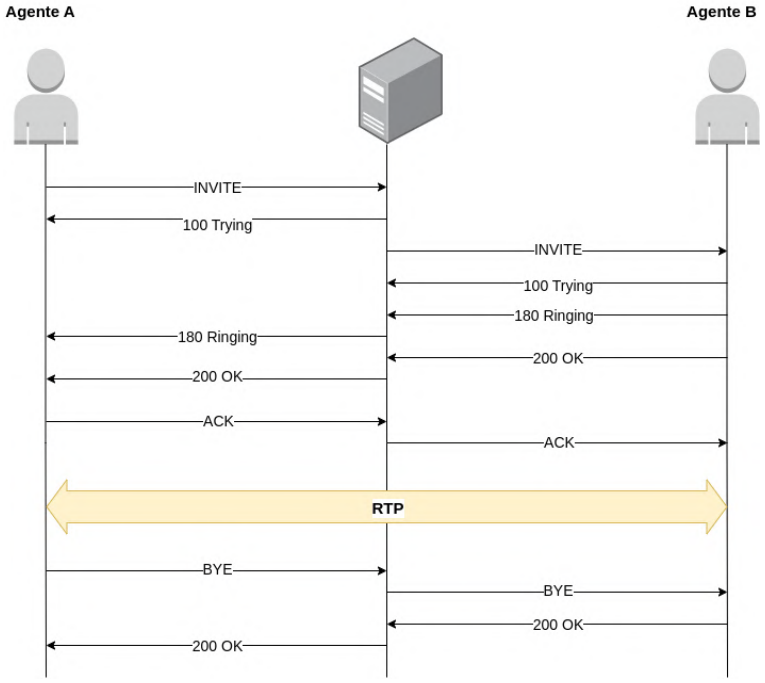
- Localização do usuário, que define o endereço de destino final usado para comunicação;
- Disponibilidade do usuário, que verifica se o usuário está disponível para estabelecer uma sessão;
- Estabelecimento de chamada, que determina os parâmetros para o chamador e o chamado e informa as duas partes sobre o andamento da chamada (toque, retorno de chamada, congestionamento);
- Gerenciamento de chamadas, que inclui a transferência e fechamento de sessão.

Desta maneira fica evidente seu papel no contexto do balanceamento de carga, onde o SIP permite a distribuição eficiente de solicitações entre servidores, garantindo uma utilização equitativa dos recursos disponíveis. Além disso, o SIP se integra de maneira relevante ao WebRTC, uma vez que fornece os meios descobrir participantes de sessões e acordar sobre os parâmetros da sessão compartilhada.

2.5.1 Fluxo Básico: Operação no modo SIP Proxy

O fluxo de comunicação básico, o modo SIP *proxy*, é aquela onde toda a sinalização SIP passa pelo servidor, facilitando processos como o de bilhetagem, pois pode reter informações das sessões. No entanto, as comunicações SIP durante o estabelecimento da sessão podem causar uma sobrecarga no servidor, representando uma desvantagem para este modelo. Mesmo operando em modo SIP *proxy*, os pacotes Protocolo de Transporte em Tempo Real (RTP) sempre irão diretamente de um destino final para o outro.

Figura 4 – Fluxo Básico de Operação no modo SIP *Proxy*



Fonte: Elaborado pelo autor.

No fluxo da figura 4, um agente de usuário A, como por exemplo um telefone SIP, inicia a comunicação utilizando uma mensagem conhecida como `INVITE` para enviar a chamada para um agente B. Ao detectar que a chamada é direcionada para um domínio externo, o servidor *proxy*, envia de imediato para o agente A uma mensagem `100 Trying`, enquanto procura o endereço IP do destino para encaminha a mensagem de `INVITE` para o agente B.

Após receber a mensagem SIP, o agente B confirma o `INVITE` para chamada enviando de volta uma mensagem de confirmação provisoria de `180 Ringing`. Em seguida uma mensagem de `200 OK` é encaminhada quando o agente B atende a ligação.

Agora com a confirmação, o agente B recebe um `ACK`, e agora possui todas as informações necessárias para estabelecer uma sessão RTP com o agente A (SCHULZRINNE et al., 2003). A comunicação continua até que uma das partes envie uma mensagem `BYE`, terminando assim a sessão.

2.5.2 Componentes do SIP

Toda a sinalização SIP passa pelo servidor *proxy* SIP. Entretanto, a mídia, que é transportada pelo protocolo RTP, é transmitida diretamente de um ponto final a outro. Alguns dos componentes do SIP incluem:

Tabela 1 – Componentes SIP

Componente	**Descrição**
Agente de usuário (UA)	O terminal SIP (telefone IP, *softphone* e assim por diante).
Agente de usuário Cliente (UAC)	O cliente ou terminal que inicia a sinalização SIP.
Agente de usuário Servidor (UAS)	O servidor que responde à sinalização SIP proveniente de um UAC.
Servidor *Proxy*	Recebe solicitações de um UA e transfere para outro *proxy* SIP se este terminal específico não estiver sob seu domínio.
Servidor redirecionador	Recebe solicitações e responde ao chamador com uma mensagem contendo dados sobre o destino.
Servidor de localização	fornece os endereços de contato do destinatário aos servidores de *proxy* e redirecionamento.
Servidor de registro	Aceita as solicitações de registro e salva as informações recebidas dentro desses pacotes no banco de dados de localização para seus domínios gerenciados.

2.5.3 Mensagens Básicas

As mensagens básicas utilizadas em um ambiente SIP são:

O uso dessas mensagens é comum em diversos cenários SIP, proporcionando uma comunicação eficiente e padronizada entre dispositivos e servidores. Cada mensagem desempenha um papel específico na sinalização e gerenciamento de sessões, desde o estabelecimento inicial com a mensagem `IINVITE` até o término da sessão com a mensagem `BYE`. Além disso, mensagens como `REGISTER` e `SUBSCRIBE` são essenciais para a atualização de registros e a subscrição de eventos futuros. Esse conjunto de mensagens, conforme definido pelas RFC, garante a interoperabilidade e a robustez do protocolo SIP em diferentes aplicações e ambientes de rede, sendo amplamente adotado na indústria de telecomunicações.

2.5.4 Codecs em Comunicação SIP

O SIP é responsável por iniciar, modificar e encerrar sessões de comunicação, mas não transporta diretamente os dados de mídia. Essa função é desempenhada pelo RTP, que entrega os dados de áudio e vídeo entre os agentes de usuário. Quando uma sessão SIP é estabelecida, os agentes de usuário trocam informações sobre como a mídia será

Tabela 2 – Mensagens SIP e suas descrições

Mensagem	**Descrição**	**RFC**
`ACK`	Reconhecer um `INVITE`	3261
`BYE`	Terminar uma sessão existente	3261
`CANCEL`	Cancelar um registo pendente	3261
`INFO`	Informação de sinalização de chamada intermediária	2976
`INVITE`	Estabelecimento da sessão	3261
`MESSAGE`	Transporte de mensagem instantânea	3428
`NOTIFY`	Enviar informações depois de subscrever	3265
`PRACK`	Reconhecer uma resposta provisória	3262
`PUBLISH`	Informações de estado de carregamento para o servidor	3903
`REFER`	Pedir outro UA para agir sobre URI	3515
`REGISTER`	Registrar o usuário e atualizar a tabela de localização	3261
`SUBSCRIBE`	Estabelecer uma sessão para receber futuras atualizações	3265
`UPDATE`	Atualizar informações de estado de uma sessão	3311

transmitida usando o protocolo Protocolo de descrição de sessão (SDP). O SDP negocia os parâmetros da sessão, incluindo os endereços IP e portas para o fluxo RTP, bem como os *codecs* a serem utilizados.

A escolha dos *codecs*, negociada durante o estabelecimento da sessão, é indispensável para a qualidade e eficiência da transmissão dos dados de mídia. Os *codecs* são algoritmos utilizados para codificar e decodificar esses dados, e, em uma comunicação SIP, os mais comuns incluem:

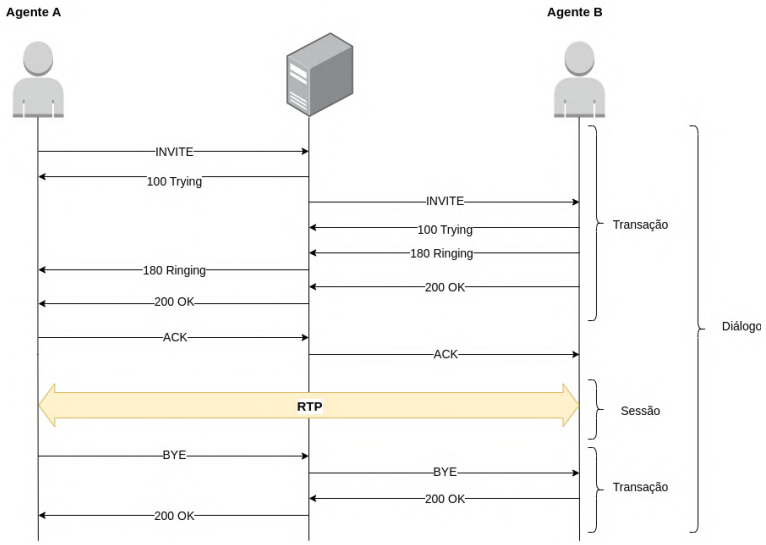
Tabela 3 – Exemplos de *codecs* utilizados em comunicação SIP

Codec	Tipo	Descrição
`G.711`	Áudio	Um *codec* de áudio que fornece alta qualidade, mas requer maior largura de banda (64 kbps).
`G.729`	Áudio	Um *codec* de áudio que oferece boa qualidade com menor largura de banda (8 kbps), ideal para redes com capacidade limitada.
`H.264`	Vídeo	Um *codec* de vídeo que proporciona alta qualidade de vídeo com taxas de compressão eficientes, amplamente utilizado em videoconferências.

2.5.5 Diferença entre Diálogo, Transação e Sessão SIP

O SIP é amplamente utilizado para estabelecer, modificar e encerrar sessões multimídia em redes IP. Dentro desse contexto, é essencial compreender as distinções entre os conceitos de Diálogo, Sessão e Transação, que são fundamentais para o funcionamento eficiente do protocolo.

Figura 5 – Diálogo, Transação e Sessão SIP



Fonte: Elaborado pelo autor.

Diálogo: Um diálogo em SIP refere-se a uma relação persistente entre dois agentes de usuário, que é identificada por um `Call-ID` específico. Esse identificador exclusivo permite que as mensagens trocadas entre os agentes de usuário sejam corretamente sequenciadas e roteadas, garantindo a continuidade da comunicação. O diálogo não representa a chamada em si, mas sim o contexto em que uma série de transações ocorrem entre os agentes envolvidos.

Transação: A transação é definida como uma sequência de mensagens SIP, começando por uma requisição enviada por um agente de usuário e terminando com a recepção de uma resposta final. Dentro de um diálogo, várias transações podem ocorrer, cada uma responsável por uma interação específica, como o início de uma chamada, atualização de parâmetros ou término de uma comunicação. As transações são, portanto, os blocos construtivos que sustentam o diálogo.

Sessão: A sessão representa a chamada em si, isto é, a fase em que ocorre a troca efetiva de pacotes de áudio ou vídeo, geralmente utilizando o RTP. A sessão SIP é estabelecida e gerenciada por meio de transações e diálogos, mas sua existência é caracterizada pelo fluxo contínuo de dados multimídia entre os agentes de usuário.

2.5.6 Diálogo de mensagens SIP

As mensagens SIP são sequenciadas de maneira ordenada. Por exemplo, conforme ilustrado na figura, o agente A utiliza um telefone IP para chamar, através da rede, outro telefone IP. Para completar a chamada, são utilizados dois *proxies* SIP, no caso deste exemplo é feito com o auxilio da ferramente *SIPp*. O agente A emprega sua identidade SIP (`sip:sipp@127.0.0.1:5061`), conhecida como SIP URI, para chamar o agente B (`sip:service@127.0.0.1:5060`).

Também é possível utilizar um SIP URI seguro com *Transport Layer Security* (TLS) no transporte entre chamador e destinatário. Neste agente A envia uma solicitação `INVITE` ao agente B contendo diversos campos de cabeçalho que fornecem informações adicionais sobre a mensagem. Estes campos incluem um identificador exclusivo, o destino e informações sobre a sessão.

Figura 6 – Requisição de SIP `INVITE`

```
UDP message sent (509 bytes):

INVITE sip:service@127.0.0.1:5060 SIP/2.0
Via: SIP/2.0/UDP 127.0.0.1:5061;branch=z9hG4bK-41540-1-0
From: sipp <sip:sipp@127.0.0.1:5061>;tag=41540SIPpTag001
To: service <sip:service@127.0.0.1:5060>
Call-ID: 1-41540@127.0.0.1
CSeq: 1 INVITE
Contact: sip:sipp@127.0.0.1:5061
Max-Forwards: 70
Subject: Performance Test
Content-Type: application/sdp
Content-Length:   129

v=0
o=user1 53655765 2353687637 IN IP4 127.0.0.1
s=-
c=IN IP4 127.0.0.1
t=0 0
m=audio 6000 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Fonte: Elaborado pelo autor.

O cabeçalho da mensagem SIP abrange desdo o `INVITE`

até o campo `Content-Length`, conforme visto na figura 6. A primeira linha da mensagem de solicitação `INVITE` exibe a identidade SIP do destino (`sip:service@127.0.0.1:5060`) e a versão do protocolo `(SIP/2.0)`. Em seguida, temos o campo `Via`, exibe o "circuito virtual" em nível de aplicação, neste caso: o endereço e a porta onde o usuário agente A espera receber a resposta de sua solicitação. O endereço utilizado é "127.0.0.1:5061" e a porta 5061, que é uma das portas padrão do SIP. Ainda neste mesmo campo, há o parâmetro `branch`, que serve como um identificador de transação. Desta forma, as respostas referentes a esta transação podem ser identificadas, pois deverão possuir o mesmo valor no parâmetro `branch`.

O campo `TO` carrega um nome de exibição e a URI SIP que identifica o destinatário. Já o campo `FROM` carrega um nome de exibição e a URI SIP que identifica o chamador. Observa-se também a presença do parâmetro `tag`, que é uma sequência de caracteres gerada aleatoriamente pela origem.

O próximo campo é o `Call-ID`, outra sequência de caracteres aleatória gerada pela origem com o objetivo de identificar uma sessão SIP e, portanto, deve ser única no dispositivo SIP de origem. Os User Agents de origem e destino contribuem com a identificação da chamada cada um com mais uma sequência de caracteres aleatória, que são os parâmetros `tag` nos campos `TO` e `FROM`. Esses três valores (`Call-ID`, `tag` do campo `TO` e `tag` do campo `FROM`) identificarão completamente um diálogo aberto entre origem e destino (SCHOOLER et al., 2002). Nesta mensagem, não há parâmetro `tag` no campo `TO`, porque ele será adicionado quando o destinatário responder a esta solicitação `INVITE`.

O campo seguinte é o `CSeq` (Command Sequence), que traz um valor inteiro e o nome do método. A cada nova solicitação enviada dentro de um diálogo, o valor inteiro deve ser incrementado. O próximo campo na mensagem `INVITE` é o `Contact`, que contém o SIP URI do originador da chamada, para o qual o UA de destino poderá encaminhar as transações seguintes diretamente

O campo `Max-Forwards` exibe a quantidade máxima de dispositivos SIP que a mensagem pode atravessar. Onde cada salto, o campo é decrementado e, quando chega a zero, é descartado pelo dispositivo SIP que receber a mensagem. Os dois campos seguintes, `Content-Type` e `Content-Length`, informam o tipo de mensagem que está sendo carregada no corpo do protocolo SIP e qual o tamanho desta mensagem, respectivamente.

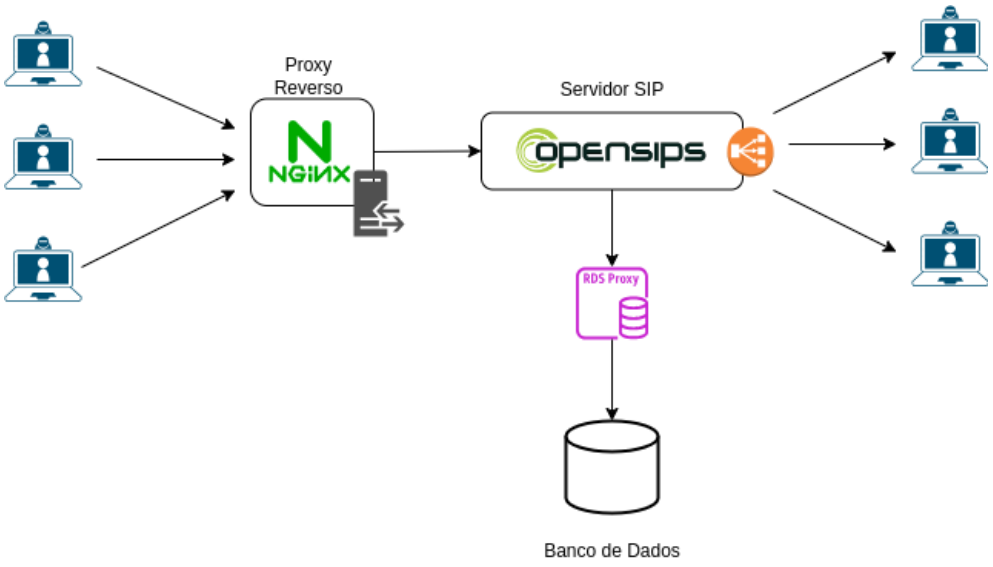
3 DESENVOLVIMENTO

O cenário proposto neste trabalho consiste em uma análise de possibilidades na utilização de balanceadores de carga com sinalização SIP. O objetivo principal é analisar a eficiência e a performance dos balanceadores de carga ao lidar com um grande volume de requisições em um ambiente de comunicação em tempo real.

Com base no trabalho de conclusão de curso de Kolakoski (2018), foi definido de montar uma arquitetura com base no software livre *OpenSIPS*[1], pois atendia as necessidades de processamento de chamadas, escalabilidade e adaptabilidade.

3.1 Arquitetura da Aplicação

Figura 7 – Arquitetura da Aplicação proposta



Fonte: Elaborado pelo autor.

A arquitetura do cenário envolveu a integração dos seguintes componentes por conteinerização *Docker*[2] dos mesmos, afim de criar um ambiente de teste robusto e escalável. A figura 7 ilustra a ideia proposta, utilizando os seguintes componentes:

- *OpenSIPS*: Configurado como o servidor SIP principal, responsável pela sinalização SIP e pelo balanceamento de carga. Sua escolha se deve à capacidade de manipular um grande número de chamadas SIP simultâneas e às funcionalidades avançadas de roteamento e balanceamento de carga. Essa configuração permitiria a investigação

[1] https://opensips.org/
[2] https://www.docker.com/

de diferentes estratégias de balanceamento, explorando como o servidor pode distribuir de forma eficiente as chamadas entre múltiplos destinos para evitar sobrecarga e melhorar a qualidade do serviço.

- *Nginx*[3]: Serviu como *proxy* reverso no cenário, facilitando a gestão de tráfego entre os clientes SIP e o servidor *OpenSIPS*. O *Nginx* foi configurado para lidar com o tráfego HTTP e HTTPS, oferecendo suporte adicional à segurança e desempenho do sistema. O uso do *Nginx* também permite a investigação de possíveis cenários onde o balanceamento de carga pode ser realizado diretamente no *proxy* reverso, comparando sua eficácia em relação ao balanceamento no *OpenSIPS*.

Na arquitetura apresentada na Figura 7, a arquitetura proposta envolve a utilização de um Proxy Reverso (Nginx) e um servidor SIP (OpenSIPS), integrados a um banco de dados. O fluxo de troca de mensagens SIP acontece da seguinte forma:

1. Início da comunicação: O cliente (User Agent) inicia uma sessão enviando uma solicitação SIP (como um `INVITE`) ao Proxy Reverso (Nginx), que atua como um ponto de entrada, gerenciando o tráfego e direcionando as requisições ao servidor SIP apropriado.

2. Encaminhamento de mensagens: O Nginx, configurado como Proxy Reverso, recebe as mensagens SIP e as repassa para o OpenSIPS, que é o servidor responsável por processar as requisições SIP e por realizar balanceamento de carga. Contudo vale notar que o Nginx também pode realizar balanceamento de carga, distribuindo o tráfego entre diferentes instâncias do OpenSIPS, caso configurado.

3. Processamento no servidor SIP : Ao receber a mensagem do Nginx, o OpenSIPS, que atua como Servidor de Sinalização SIP , realiza o processamento da requisição. Esse servidor pode consultar o banco de dados para verificar informações de roteamento, autenticação ou de estado do usuário, através do componente RDS Proxy, garantindo que a chamada seja direcionada corretamente.

4. Roteamento para os destinatários: Após o processamento no OpenSIPS, a mensagem SIP é roteada para o cliente de destino (User Agent). Se for uma mensagem `INVITE`, por exemplo, o destino poderá aceitar ou rejeitar a chamada, respondendo com mensagens de status adequadas, como `200 OK` (chamada aceita) ou `4XX` (erros ou recusas).

5. Troca de mídia: Quando o convite (`INVITE`) é aceito, é estabelecido um canal de mídia diretamente entre os dois agentes de usuário (User Agents). A mídia, como áudio

[3] https://nginx.org/

ou vídeo, é transmitida diretamente entre os clientes, sem passar pelo OpenSIPS ou Nginx, que ficam encarregados apenas da sinalização.

6. Encerramento da chamada: Para encerrar a chamada, uma mensagem `BYE` é enviada de um dos clientes, seguindo o mesmo caminho através do Nginx até o OpenSIPS, que processa e repassa a mensagem ao outro agente de usuário.

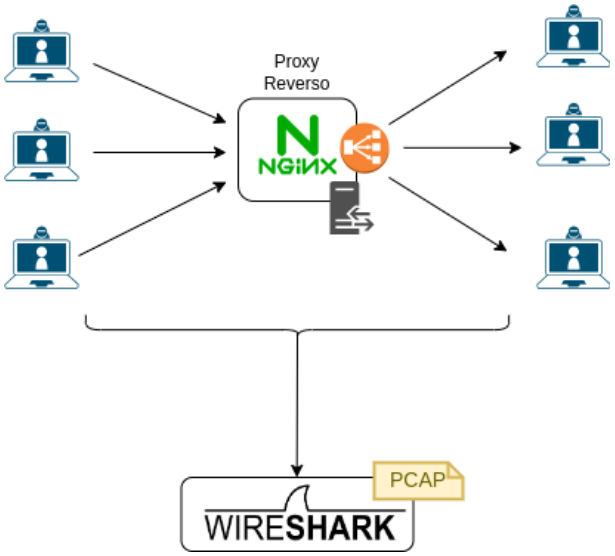
3.2 Infraestrutura da Aplicação

A infraestrutura utilizada para a implementação do cenário descrito foi duas máquinas virtuais, ambas interagindo entre por meio da rede, fornecida pela campus, contando com os seguintes recursos:

- 8 vCPUs: Permitindo a execução simultânea de múltiplas *threads* e processos, essencial para lidar com o tráfego SIP e WebRTC de alto volume.
- 16 GB RAM: Proporcionando memória suficiente para a operação eficiente dos servidores e ferramentas de teste, garantindo estabilidade durante os testes de carga.
- 100 GB Disco: Armazenamento suficiente para o sistema operacional, servidores, *logs* e dados de teste.
- 1 IPv4 e 1 IPv6: Garantindo conectividade e compatibilidade com diferentes redes e dispositivos.
- DNS (A e AAAA): Configuração de DNS para a máquina virtual, permitindo fácil acesso e identificação da máquina na rede.

3.3 Apresentação do novo Cenário

Figura 8 – Arquitetura da Aplicação Final



Fonte: Elaborado pelo autor.

A implementação do cenário proposto encontrou diversos desafios, que resultaram em ajustes e reconfigurações ao longo do desenvolvimento. Para implementar o novo cenário proposto, foram utilizadas as seguintes tecnologias e componentes, todos integrados por meio de conteinerização *Docker*. Como ilustrado na figura 8.

- *Nginx*: Atuando como *proxy* reverso, o *Nginx* facilitou a gestão do tráfego entre os clientes, sendo configurado como balanceador de carga para distribuir eficientemente as requisições recebidas. Esta configuração permitiu investigar diferentes possibilidades de uso do balanceamento de carga em um cenário SIP, utilizando os diferentes algoritmos disponíveis pelo *proxy*.
- *SIPp*[4]: Utilizado como ferramenta de teste de carga para simular um grande número de chamadas SIP, tanto no papel de UAC quanto no de UAS. *SIPp* permitiu a criação de cenários de teste realistas e a coleta de métricas de desempenho detalhadas.

Adicionalmente, para capturar o tráfego da rede durante os testes, utilizou-se a ferramenta TShark[5]. Com todos os contêineres *Docker* representando o cenário de testes configurados na mesma rede, o *TShark* possibilitou a captura eficiente dos pacotes de dados trafegados entre os componentes. Essa configuração facilitou uma análise detalhada do tráfego de rede, permitindo validar os resultados obtidos e assegurar a precisão dos dados coletados durante os testes de carga. A utilização do *TShark* foi essencial para

[4] https://github.com/SIPp/sipp
[5] https://www.wireshark.org/docs/man-pages/tshark.html

garantir que as métricas de desempenho refletissem com exatidão o comportamento do sistema em condições de teste.

No arquitetura apresentada na Figura 8, a arquitetura final da aplicação mantém o uso do Proxy Reverso (Nginx) e o servidor SIP para gerenciar a troca de mensagens SIP , porém com a adição da ferramenta Wireshark para captura e análise de tráfego de rede em formato PCAP. A seguir, está a explicação do fluxo de troca de mensagens SIP:

1. Início da comunicação: O cliente (User Agent) envia uma solicitação SIP (como um `INVITE`) ao Proxy Reverso (Nginx), que agora desempenha a função de balanceamento e encaminhamento de tráfego para o servidor SIP , garantindo a distribuição eficiente das requisições.

2. Encaminhamento de mensagens: Assim como no cenário anterior, o Nginx recebe as mensagens SIP e as encaminha ao servidor SIP para processamento, sendo este o próprio SIPp tendo servidores SIP diretamente no UAC e UAS .

3. Encaminhamento de mensagens: Assim como no cenário anterior, o Nginx recebe as mensagens SIP e as encaminha ao SIPp para processamento, sendo responsável por gerenciar as sessões SIP .

4. Troca de mídia: Após o estabelecimento da sessão de sinalização (por exemplo, após a troca de `INVITE` e `200 OK`), a mídia, como áudio ou vídeo, é transmitida diretamente entre os clientes de origem e destino, de forma independente da camada de sinalização.

5. Encerramento da sessão: A chamada pode ser encerrada com o envio de uma mensagem `BYE`.

4 TESTES DO CENÁRIO

Foi realizado testes utilizando os hardware e software mencionados no capitulo, seguindo uma metodologia experimental com base na de Jiang et al. (2012), e métricas como:

4.1 Métricas

A análise dos resultados dos testes foi realizada com base em várias métricas para avaliar o desempenho do balanceador de carga em diferentes condições de rede e cenários de carga. As principais métricas consideradas foram as seguintes:

- Latência: A latência foi medida com o objetivo de avaliar o tempo necessário para que os pacotes RTP fossem entregues entre os nós da rede. Esta métrica é determina o atraso experimentado nas chamadas de VoIP e está diretamente relacionada à qualidade do serviço oferecido. Durante os testes, foi registrada a latência média para cada cenário de carga e condição de rede.
- Jitter: Representa a variação na latência dos pacotes RTP, que pode impactar negativamente a qualidade da comunicação. Um alto valor de jitter resulta em uma experiência de áudio inconsistente para os usuários. A análise focou na medição do jitter para identificar possíveis flutuações de atraso nos pacotes de áudio, principalmente em cenários com redes degradadas.
- Quantidade de INVITE em um Instante no Tempo: Foi monitorada a quantidade de mensagens SIP INVITE enviadas em determinados momentos ao longo do teste, com o objetivo de avaliar o comportamento do balanceador de carga em relação à distribuição de chamadas em tempo real. Essa métrica ajuda a entender a capacidade do balanceador em lidar com picos de requisições.
- Distribuição Total das Chamadas SIP entre os Nós: Para verificar a eficiência dos algoritmos de balanceamento de carga (Round-Robin e Least-Connection), foi analisada a distribuição total das chamadas SIP entre os nós do sistema. Essa métrica permitiu verificar se os algoritmos distribuíram uniformemente as chamadas ou se houve concentração em determinados nós, o que poderia indicar ineficiência no balanceamento.

4.2 Metodologia

Como metologia será feito teste de carga com duração de 3 minutos cada, considerando dois cenários de chamadas simultâneas: 10 chamadas por segundo e 51 chamadas por segundo. Os algoritmos de balanceamento de carga utilizados foram o *Round-Robin* e o *Least-Connection*. Uma mídia de 30 segundos com codec de *payload* g711A, foi utilizada no testes para geração de pacotes RTP. Além disso, os testes foram realizados em dois tipos de rede: uma rede sem degradação (normal) e uma rede com degradação simulada (degradada).

Para a execução dos testes, foi utilizado um ambiente controlado onde a degradação da rede foi aplicada artificialmente, seguindo os padrões da norma escrita por Bradner (1999). A degradação incluiu aumento de latência e perda de pacotes. Onde:

- 100ms de latência: Emula uma rede com latência mais alta, como pode ocorrer em conexões internacionais.
- 5% de perda de pacotes: Simula um ambiente de rede ainda mais degradado, útil para testar a resiliência extrema do sistema.

O principal objetivo dos testes foi comparar o desempenho dos algoritmos de balanceamento de carga sob diferentes condições de rede e níveis de carga de chamadas SIP.

Tabela 4 – Fatores e Níveis para os testes

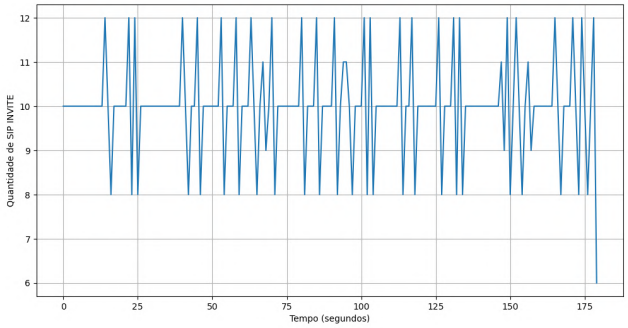
Fator	**Níveis**
Cenário de Carga	10 chamadas por segundo, 51 chamadas por segundo
Algoritmo de Balanceamento	Round-Robin, Least-Connection
Condição da Rede	Normal, Degradada

4.3 Resultados

A seguir será descrito os resultados das métricas obtidas dos testo seguindo as metologia mencionada:

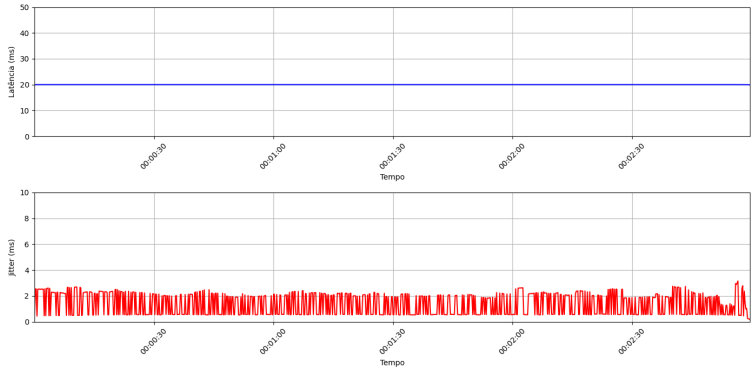
4.3.1 Round-Robin – 10 Chamadas Simultâneas – Sem Degradação da Rede

Figura 9 – Quantidade SIP INVITES sobre o tempo



Fonte: Elaborado pelo autor.

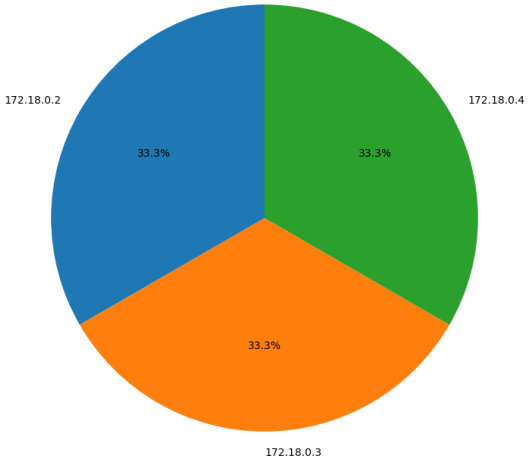
Figura 10 – latência e Jitter sobre o tempo



Fonte: Elaborado pelo autor.

Neste cenário, o algoritmo *Round-Robin* foi utilizado com 10 chamadas simultâneas em uma rede sem degradação. O gráfico de latência do fluxo RTP mostrou valores consistentes e baixos, indicando que a rede estava estável e o algoritmo conseguiu balancear as chamadas de maneira eficaz. O jitter também permaneceu baixo, o que sugere uma boa qualidade de serviço.

Figura 11 – Distribuição das Chamadas SIP

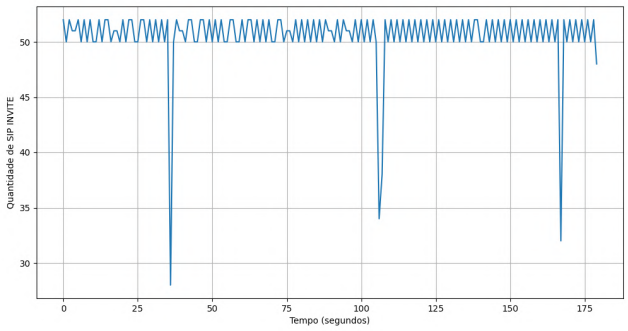


Fonte: Elaborado pelo autor.

A distribuição das chamadas entre os nós foi equilibrada, sem sobrecarregar nenhum dos servidores. A quantidade de mensagens `SIP INVITE` por segundo apresentou variações mínimas, sendo um indicativo de que o sistema foi capaz de manter um comportamento previsível em condições normais de rede.

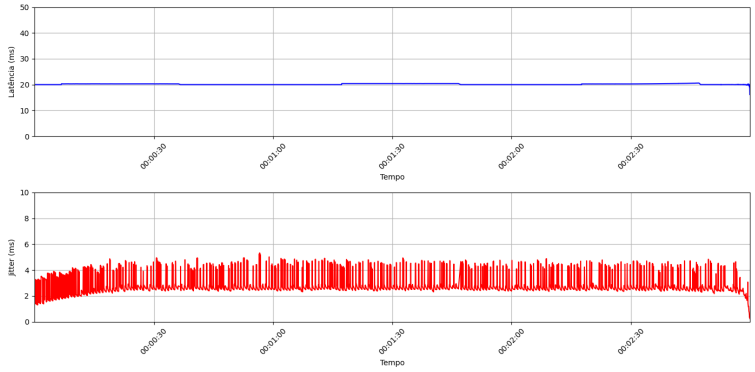
4.3.2 Round-Robin – 51 Chamadas Simultâneas – Sem Degradação da Rede

Figura 12 – Quantidade SIP INVITES sobre o tempo



Fonte: Elaborado pelo autor.

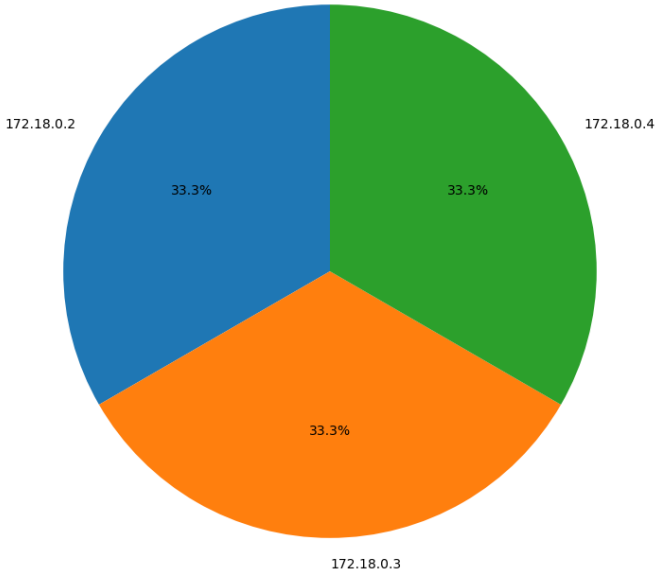
Figura 13 – latência e Jitter sobre o tempo



Fonte: Elaborado pelo autor.

Com o aumento da carga para 51 chamadas simultâneas, o algoritmo *Round-Robin* continuou a apresentar latência estável, sem grandes oscilações. O jitter, embora um pouco mais elevado que no cenário anterior, ainda se manteve dentro de uma faixa aceitável, não comprometendo a qualidade das chamadas.

Figura 14 – Distribuição das Chamadas SIP

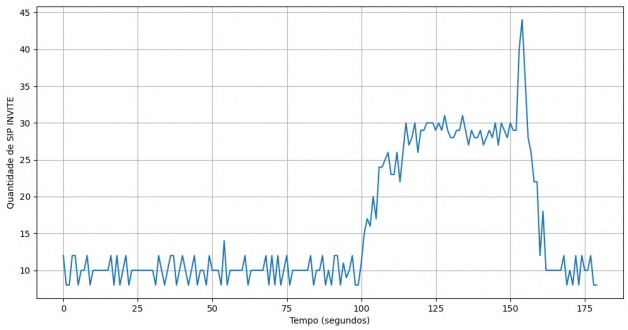


Fonte: Elaborado pelo autor.

O gráfico de distribuição das chamadas indica que o algoritmo distribuiu de maneira uniforme a carga entre os nós. A quantidade de mensagens `SIP INVITE` por segundo demonstrou alguns picos, mas sem comprometer o desempenho geral do sistema. Esse comportamento é esperado quando o número de chamadas simultâneas aumenta, mas a rede sem degradação lidou bem com a carga extra.

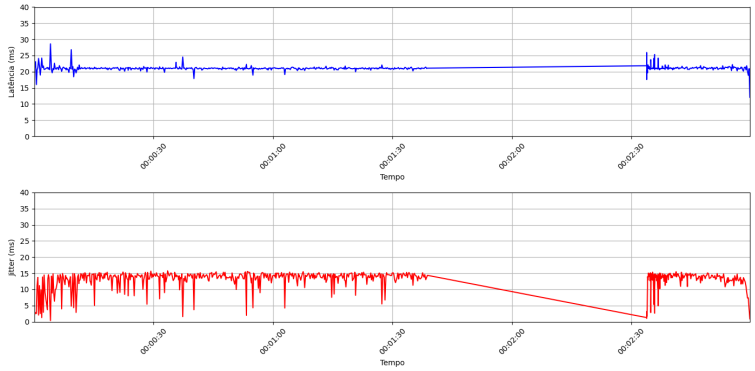
4.3.3 Round-Robin – 10 Chamadas Simultâneas – Com Degradação da Rede

Figura 15 – Quantidade SIP INVITES sobre o tempo



Fonte: Elaborado pelo autor.

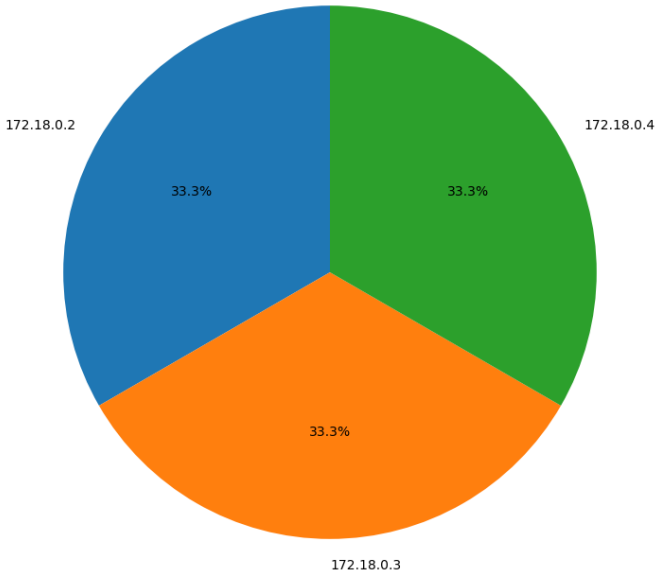
Figura 16 – latência e Jitter sobre o tempo



Fonte: Elaborado pelo autor.

Neste cenário com degradação de rede, o comportamento da latência sofreu leve aumento, mas ainda dentro de limites aceitáveis para chamadas de voz. No entanto, o jitter apresentou variações significativas em comparação ao cenário sem degradação, indicando que a qualidade do áudio pode ter sido prejudicada por variações na entrega dos pacotes RTP.

Figura 17 – Distribuição das Chamadas SIP

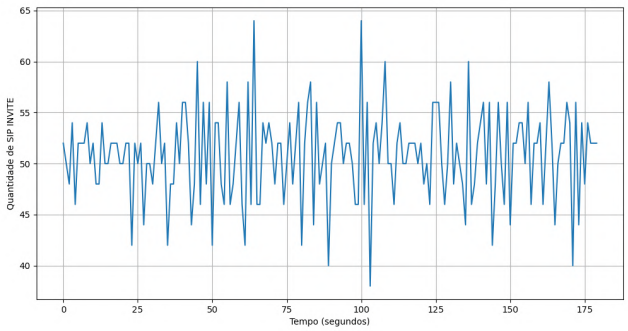


Fonte: Elaborado pelo autor.

A distribuição das chamadas entre os nós continuou a ser equilibrada, mas houve maior flutuação na quantidade de mensagens `SIP INVITE` por segundo, o que reflete as dificuldades do algoritmo em manter a consistência sob condições de rede degradada. Mesmo assim, a rede conseguiu lidar com as 10 chamadas simultâneas de forma razoável.

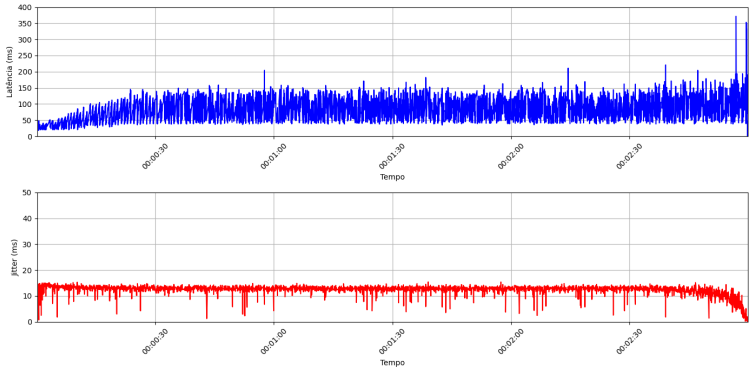
4.3.4 Round-Robin – 51 Chamadas Simultâneas – Com Degradação da Rede

Figura 18 – Quantidade SIP INVITES sobre o tempo



Fonte: Elaborado pelo autor.

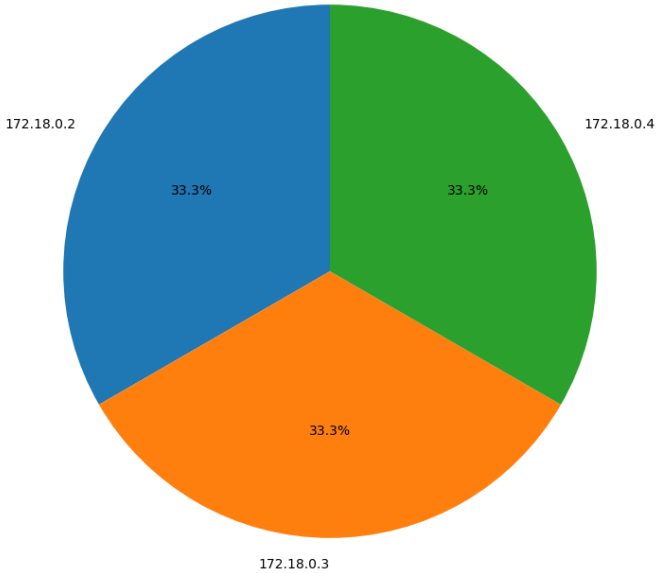
Figura 19 – Latência e Jitter sobre o tempo



Fonte: Elaborado pelo autor.

Quando a carga foi aumentada para 51 chamadas simultâneas em uma rede degradada, o desempenho do *Round-Robin* mostrou maior impacto. O aumento da latência foi mais pronunciado, e o jitter sofreu grandes flutuações, o que indica uma degradação significativa na qualidade das chamadas.

Figura 20 – Distribuição das Chamadas SIP

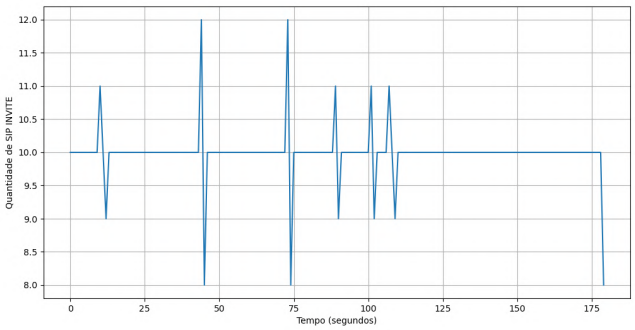


Fonte: Elaborado pelo autor.

A distribuição das chamadas entre os nós permaneceu relativamente uniforme, mas a quantidade de mensagens `SIP INVITE` por segundo apresentou picos mais altos e inconsistentes, refletindo a dificuldade de lidar com um número elevado de chamadas sob uma rede degradada. Esses resultados sugerem que o algoritmo *Round-Robin* não se adapta bem quando há uma combinação de alta carga e degradação de rede.

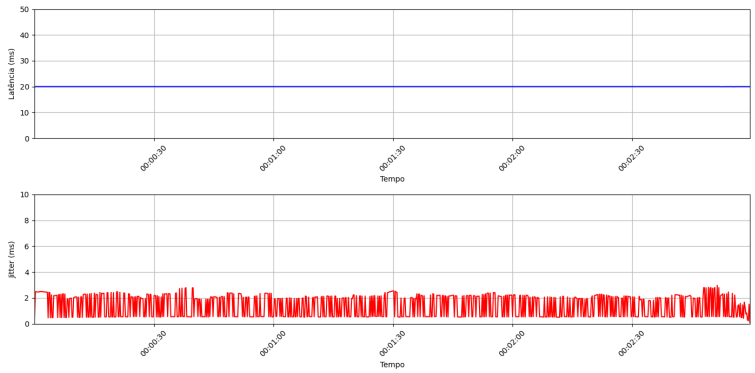
4.3.5 Least-Connection – 10 Chamadas Simultâneas – Sem Degradação da Rede

Figura 21 – Quantidade SIP INVITES sobre o tempo



Fonte: Elaborado pelo autor.

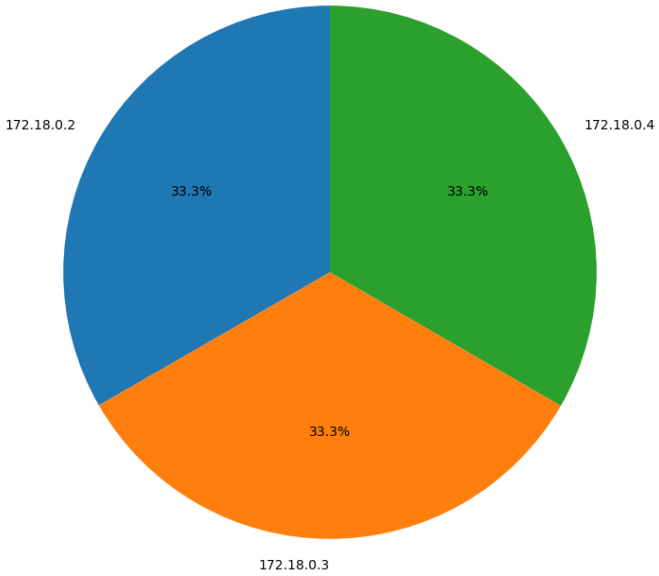
Figura 22 – latência e Jitter sobre o tempo



Fonte: Elaborado pelo autor.

No cenário sem degradação e com 10 chamadas simultâneas, o algoritmo *Least-Connection* mostrou desempenho semelhante ao *Round-Robin* em termos de latência e jitter. Os valores se mantiveram estáveis, e o jitter foi baixo, o que garantiu uma boa qualidade nas chamadas.

Figura 23 – Distribuição das Chamadas SIP

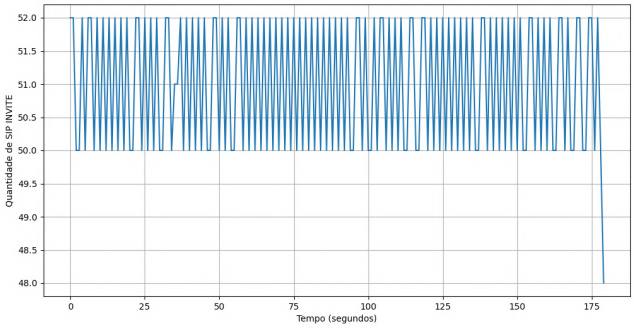


Fonte: Elaborado pelo autor.

A distribuição das chamadas SIP foi ligeiramente menos uniforme do que no Round-Robin, mas ainda assim bem balanceada. A quantidade de mensagens `SIP INVITE` por segundo foi consistente, com poucas variações, o que indica que o algoritmo lidou bem com esse cenário de baixa carga e sem problemas de rede.

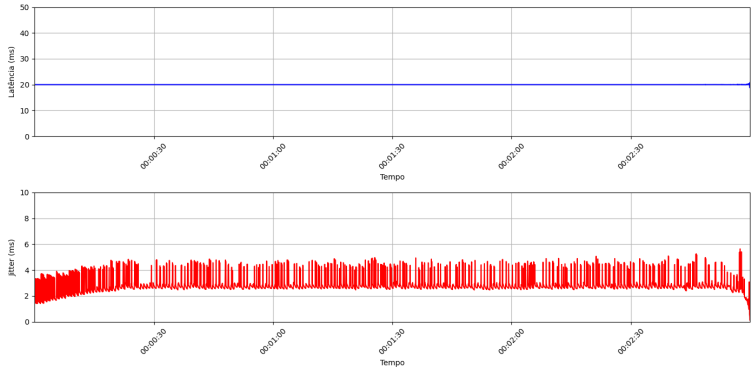
4.3.6 Least-Connection – 51 Chamadas Simultâneas – Sem Degradação da Rede

Figura 24 – Quantidade SIP INVITES sobre o tempo



Fonte: Elaborado pelo autor.

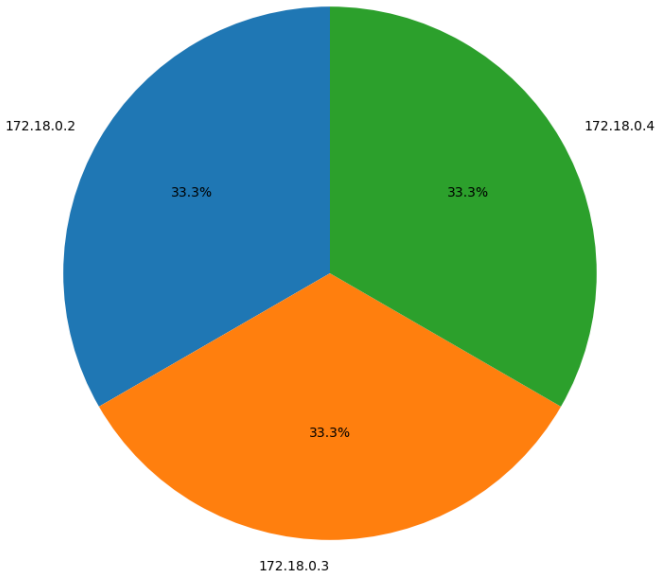
Figura 25 – Latência e Jitter sobre o tempo



Fonte: Elaborado pelo autor.

Com 51 chamadas simultâneas, o *Least-Connection* continuou apresentando boa estabilidade nos gráficos de latência e jitter, com desempenho semelhante ao Round-Robin em termos de latência. O jitter permaneceu em níveis aceitáveis, sem variações significativas que pudessem comprometer a qualidade das chamadas.

Figura 26 – Distribuição das Chamadas SIP



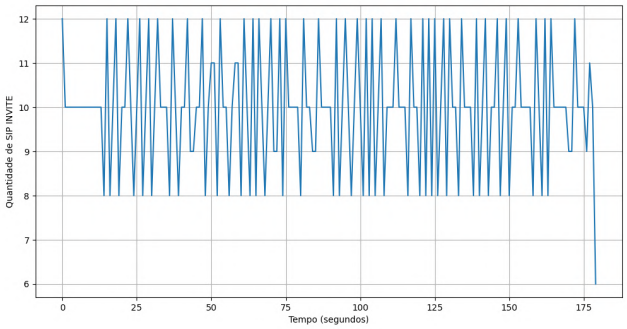
Fonte: Elaborado pelo autor.

A distribuição das chamadas foi um pouco menos uniforme, com mais chamadas sendo atribuídas a um nó específico, como esperado pelo comportamento do algoritmo *Least-Connection*, que prioriza nós com menos conexões ativas. Apesar disso, a quantidade de mensagens `SIP INVITE` por segundo foi consistente, com picos menores do que

os observados no Round-Robin, o que sugere uma melhor adaptação a um maior número de chamadas.

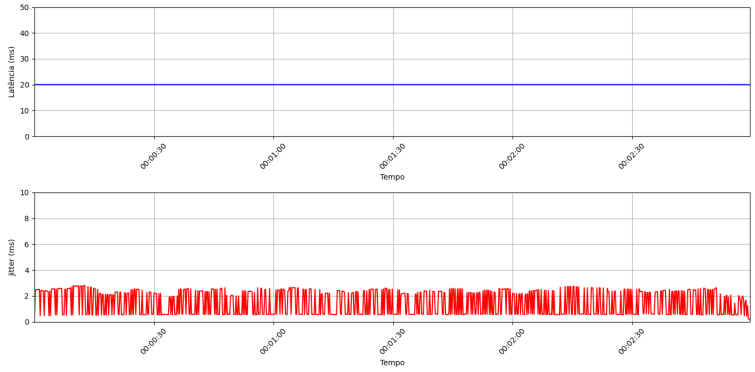
4.3.7 Least-Connection – 10 Chamadas Simultâneas – Com Degradação da Rede

Figura 27 – Quantidade SIP INVITES sobre o tempo



Fonte: Elaborado pelo autor.

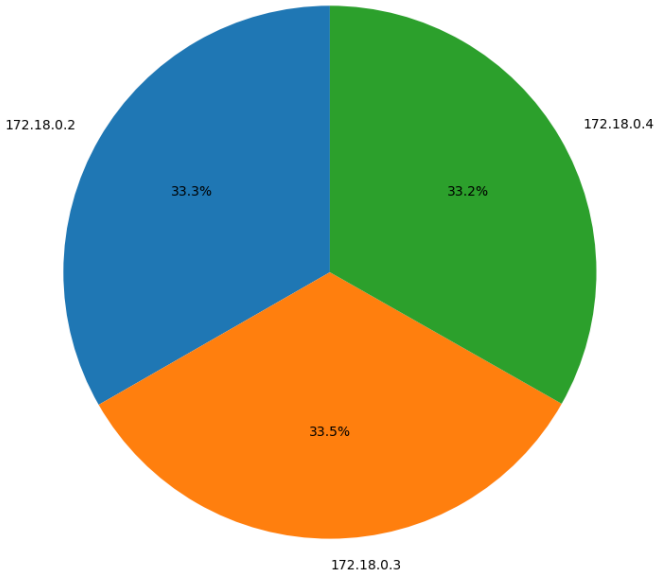
Figura 28 – Latência e Jitter sobre o tempo



Fonte: Elaborado pelo autor.

Quando a rede foi degradada com 10 chamadas simultâneas, o *Least-Connection* apresentou uma leve elevação na latência, mas ainda conseguiu manter os valores dentro de uma faixa aceitável. O jitter, no entanto, teve um aumento mais significativo em comparação ao cenário sem degradação, mas ainda menor do que o Round-Robin nas mesmas condições.

Figura 29 – Distribuição das Chamadas SIP

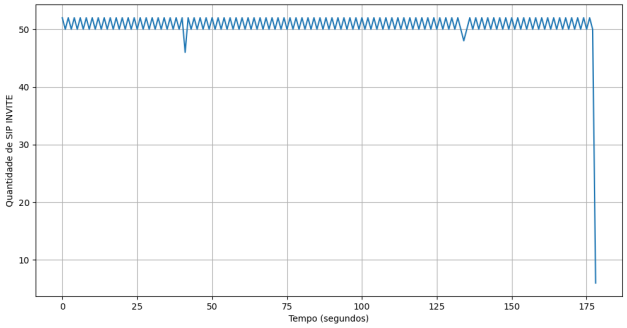


Fonte: Elaborado pelo autor.

A distribuição das chamadas entre os nós foi semelhante ao cenário sem degradação, com uma leve concentração em um nó. A quantidade de mensagens `SIP INVITE` por segundo mostrou algumas variações, mas manteve um comportamento mais estável do que o *Round-Robin*, sugerindo que o *Least-Connection* é mais eficiente em cenários de rede degradada com baixa carga.

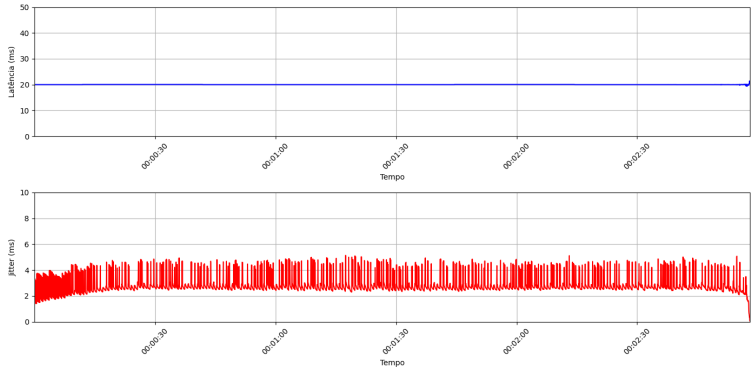
4.3.8 Least-Connection – 51 Chamadas Simultâneas – Com Degradação da Rede

Figura 30 – Quantidade SIP INVITES sobre o tempo



Fonte: Elaborado pelo autor.

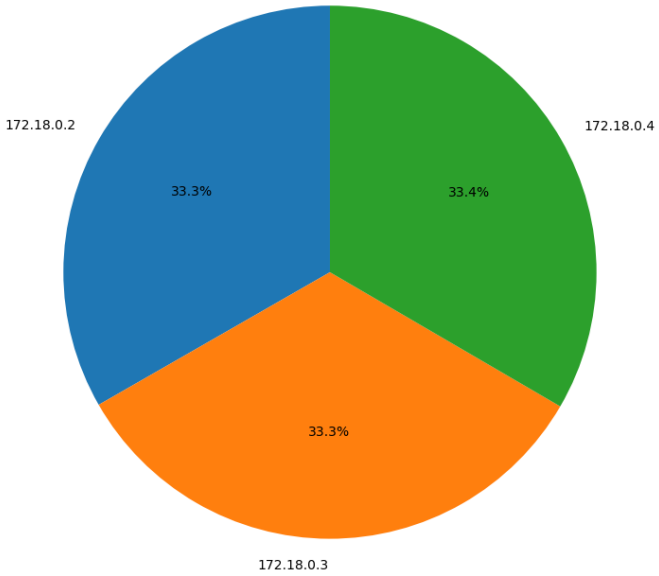
Figura 31 – Latência e Jitter sobre o tempo



Fonte: Elaborado pelo autor.

Este foi o cenário mais desafiador, com alta carga (51 chamadas simultâneas) em uma rede degradada. O *Least-Connection* demonstrou uma elevação acentuada na latência e jitter, mas ainda assim apresentou um desempenho melhor do que o *Round-Robin*. O jitter, embora presente, foi menos volátil do que nos testes de Round-Robin, indicando que a qualidade das chamadas, embora impactada, foi mais consistente.

Figura 32 – Distribuição das Chamadas SIP



Fonte: Elaborado pelo autor.

A distribuição das chamadas continuou com uma leve concentração em um nó, como esperado para o algoritmo. A quantidade de mensagens `SIP INVITE` por segundo apresentou picos mais moderados em comparação ao *Round-Robin*, indicando que o *Least-Connection* é mais adaptável a condições adversas com alta carga.

5 CONCLUSÕES

Este trabalho apresentou uma abordagem para balanceamento de carga de servidores SIP.

Com base nos resultados obtidos, em redes sem degradação, tanto o *Round-Robin* quanto o *Least-Connection* apresentaram desempenho satisfatório em termos de latência e jitter, com o Round-Robin oferecendo uma distribuição mais uniforme de chamadas e o Least-Connection priorizando nós menos ocupados.

Em redes degradadas, o *Least-Connection* mostrou-se mais eficiente do que o *Round-Robin*, especialmente sob alta carga (51 chamadas simultâneas). O *Least-Connection* conseguiu manter um controle melhor sobre o jitter e distribuiu as mensagens `SIP INVITE` de forma mais estável, enquanto o Round-Robin sofreu maiores variações, impactando negativamente a qualidade das chamadas.

Assim, para ambientes com redes sujeitas a degradação e alta carga, o algoritmo *Least-Connection* parece ser a escolha mais indicada, pois oferece uma maior consistência nas métricas de desempenho.

5.1 Trabalhos Futuros

Para trabalhos futuros, propõe-se o desenvolvimento de uma ferramenta mais robusta para testes de carga, com foco específico em suporte para SIP e, se possível, WebRTC. Esta ferramenta pode facilitar a realização de testes em cenários de alta demanda, permitindo que desenvolvedores e administradores de sistemas simulem uma variedade de condições reais de tráfego.

A inclusão de recursos avançados, como a simulação de diferentes tipos de carga e variações de latência, que poderá proporcionar a otimização de sistemas de comunicação. Além disso, a ferramenta poderá integrar técnicas de análise de desempenho, permitindo a coleta e a visualização de métricas relevantes, como latência, *jitter* e taxa de perda de pacotes.

Outro aspecto a ser considerado é a contribuição ao *SIPp*, uma ferramenta de código aberto amplamente utilizada para simulação de tráfego SIP. Ao integrar novos recursos e funcionalidades, como suporte a cenários de teste para SIP sobre WebSocket, será possível ampliar suas capacidades e torná-la ainda mais útil para a comunidade. Essa contribuição não só reforçará o papel do *SIPp* como uma ferramenta essencial para desenvolvedores e engenheiros de redes, mas também incentivará a colaboração entre

usuários e desenvolvedores, promovendo inovações e melhores práticas na realização de testes de carga. Além disso, a criação de uma documentação abrangente e tutoriais para novos recursos poderá facilitar a adoção da ferramenta, aumentando sua relevância no cenário atual de comunicações.

REFERÊNCIAS

BELGAUM, M. R. et al. A systematic review of load balancing techniques in software-defined networking. *IEEE Access*, v. 8, p. 98612–98636, 2020. Disponível em: https://ieeexplore.ieee.org/document/9097181. 14

BRADNER, S. *Benchmarking Methodology for Network Interconnect Devices*. 1999. RFC 2544. Obsoletes RFC 1242. Disponível em: https://tools.ietf.org/html/rfc2544. 34

Denise Ribeiro; Anthony Wells. *Com pandemia, demanda por videoconferências dispara em empresas brasileiras*. 2023. Acessado em 12 de setembro de 2023. Disponível em: https://www.cnnbrasil.com.br/economia/com-pandemia-demanda-por-videoconferencias-dispara-em-empresas-brasileiras/. 11

DéLY, T. L. Caching HTTP: A comparative study of caching reverse proxies Varnish and Nginx. *Journal of Web Performance and Optimization*, June 2014. Disponível em: https://www.diva-portal.org/smash/get/diva2:734117/FULLTEXT01.pdf. 18, 19

JIANG, H. et al. Design, implementation, and performance of a load balancer for sip server clusters. *IEEE/ACM Transactions on Networking*, ago. 2012. Disponível em: https://doi.org/10.1109/TNET.2012.2183612. 33

KEMP, S. *DIGITAL 2022: GLOBAL OVERVIEW REPORT*. 2022. Acessado em 12 de setembro de 2023. Disponível em: https://datareportal.com/reports/digital-2022-global-overview-report. 11

KOLAKOSKI, B. A. *Serviço Distribuído de Áudio e Videoconferência baseado em SIP e WebRTC para Aplicações Web*. Dissertação (Dissertação (Graduação)) — Instituto Federal de Santa Catarina, São José, março 2018. 28

LOUREIRO, A. A. et al. Comunicação sem fio e computação móvel: Tecnologias, desafios e oportunidades. 2023. Acessado em 12 de setembro de 2023. Disponível em: https://homepages.dcc.ufmg.br/~loureiro/cm/docs/jai03.pdf. 11

MUSTAFA, D. M. E. Load-balancing algorithms round-robin (rr), least-connection, and least loaded efficiency. *GESJ: Computer Science and Telecommunications*, n. 1(51), 2017. ISSN 1512-123. Disponível em: https://search.ebscohost.com/login.aspx?direct=true&db=iih&AN=125216431&lang=pt-br&site=ehost-live. 16, 17, 18

NEGHABI, A. A. et al. Load balancing mechanisms in the software defined networks: A systematic and comprehensive review of the literature. *IEEE Access*, v. 6, p. 14159–14178, 2018. 11, 14

Nginx, Inc. *Nginx*. 2024. https://nginx.org/. Accessed: 2024-09-20. 20

NOTTINGHAM, M. *Caching Tutorial*. 2013. Disponível em: http://www.mnot.net/cache_docs/. 19, 20

SAJJAN, R. S.; BIRADAR, R. Y. Load balancing and its algorithms in cloud computing: A survey. *Survey Paper*, E-ISSN: 2347-2693, v. 5, n. 1, 2017. Available online at: http://www.ijcseonline.org. Disponível em: https://www.researchgate.net/profile/Rajani-Sajjan-2/publication/313766818_Load_Balancing_and_its_Algorithms_in_Cloud_Computing_A_Survey/links/58a554d44585150402cc4376/Load-Balancing-and-its-Algorithms-in-Cloud-Computing-A-Survey.pdf. 15, 16

SCHOOLER, E. et al. *SIP: Session Initiation Protocol*. RFC Editor, 2002. RFC 3261. (Request for Comments, 3261). Disponível em: https://www.rfc-editor.org/info/rfc3261. 21, 27

SCHULZRINNE, H. et al. *RTP: A Transport Protocol for Real-Time Applications*. [S.l.], 2003. Disponível em: http://www.rfc-editor.org/rfc/rfc3550.txt. Acesso em: 13 ago. 2024. 22

Apêndices

APÊNDICE A – IMPLEMENTAÇÃO DA APLICAÇÃO

A seguir é demonstrado como foi implementado o cenário utilizado nos testes para analise dos dados obtidos:

A.1 Conteinerização do Cenário

A criação dos contêineres ocorreu a partir da criação de um *docker-compose* que utiliza os *Dockerfiles* que carrega o *nginx* e suas configurações, como *proxy* reverso e balanceador de carga, e três contêineres *SIPp* como UAS para o recebimento das chamadas SIP. O sistema operacional nos contêineres é o *Debian*.

A.1.1 Criando o Dockerfile Nginx

O *Dockerfile* apresentado é um arquivo de configuração utilizado para criar uma imagem de *Docker*, que por sua vez é empregada para construir contêineres, ambientes isolados que replicam o sistema operacional e as dependências necessárias para a execução de aplicações.

O primeiro passo na criação do *Dockerfile* é a escolha do sistema operacional, aproveitou-se uma imagem já existente no *Docker Hub*, repositório de imagens *Docker*, que é compatível com *Nginx*.

Após descarregar a imagem do *Debian* é realizada a atualização dos repositórios de *software* e a instalação de pacotes essenciais.

Código A.1 – Execução de Comandos

```
1 RUN apt-get update && \
2     apt-get install -y nginx certbot python3-certbot-nginx libnginx-mod-stream
    libnginx-mod-http-geoip libnginx-mod-rtmp && \
3     apt-get clean
```

Nesta etapa, utilizando comando `RUN` realiza a execução de um comando. O comando `apt-get update` é utilizado para atualizar a lista de pacotes disponíveis nos repositórios. Em seguida, são instalados os pacotes `nginx`, `certbot` e outros módulos do *Nginx*, como `libnginx-mod-stream`, `libnginx-mod-http-geoip` e `libnginx-mod-rtmp`. Estes pacotes são essenciais para configurar o servidor web *Nginx* com suporte para SSL (*Certbot*) e módulos adicionais que expandem suas funcionalidades.

Em seguida é feito a exposição das portas de rede que serão utilizadas pelo contêiner.

Código A.2 – Exposição das Portas

```
1 EXPOSE 80/tcp
2 EXPOSE 443/tcp
```

Aqui, as portas 80 e 443 são expostas, utilizando o comando `EXPOSE`, para permitir o tráfego HTTP e HTTPS, respectivamente. O protocolo tcp é especificado, garantindo que o *Nginx* seja acessível através dessas portas no contêiner.

Por fim, é copiado um *script* de renovação automática de certificados SSL para dentro do contêiner. É utilizado o comando `COPY` para copiar o arquivo para o contêiner.

Código A.3 – Script de renovação de Certificados

```
1 #!/bin/bash
2
3 # Renovar os certificados SSL usando o Certbot
4 certbot renew
5 # Recarregar o Nginx para aplicar as alterações
6 nginx -s reload
```

Código A.4 – Copia de *script* para Contêiner

```
1 COPY certbot_renew.sh /usr/local/bin/
2 RUN chmod +x /usr/local/bin/certbot_renew.sh
```

O script `certbot_renew.sh` é copiado para o diretório `/usr/local/bin/` dentro do contêiner, e em seguida, seu atributo de execução é ajustado com o comando `chmod +x`, permitindo que o *script* seja executado corretamente. Este *script* simples mente executa o `certbot renew` para renovar certificações e executa o *nginx* com `nginx -s reload`.

A.1.2 Criando o Dockerfile SIPp

Um processo similar ao da secção anterior será feito para criação do *Dockerfile* do *SIPp*. Será utilizado o mesmo sistema operacional, o *Debian Buster Slim*, para este container.

Após descarregar a imagem do *Debian* é realizado é definido um argumento que especifica a versão do *SIPp* a ser instalada:

Código A.5 – Argumento para Escolha de Versão do *SIPp*

```
1 ARG SIPP_VERSION="3.7.2"
```

O uso do comando `ARG` traz a criação de um argumento `SIPP_VERSION` permite que a versão do *SIPp* seja facilmente alterada sem modificar outras partes do *Dockerfile*. Neste caso, a versão definida é a 3.7.2.

O próximo bloco de comandos realiza a atualização dos repositórios de pacotes e instala as dependências necessárias, o mesmo também baixa e compila a versão especificada do *SIPp*.

Código A.6 – Execução de Comandos para instalação do *SIPp*

```
1 RUN apt-get update && \
2     apt-get install -y --no-install-recommends build-essential cmake wget libssl-
    dev libpcap-dev libsctp-dev libncurses5-dev && \
3     wget --no-check-certificate "https://github.com/SIPp/sipp/releases/download/
    v$SIPP_VERSION/sipp-$SIPP_VERSION.tar.gz" && \
4     tar xzf sipp-$SIPP_VERSION.tar.gz -C . && \
5     cd sipp-$SIPP_VERSION && \
6     ./build.sh --full
```

Aqui, `apt-get update` atualiza a lista de pacotes disponíveis, seguido da instalação dos pacotes essenciais para a construção do *SIPp*, como `build-essential`, `cmake`, `wget`, `libssl-dev`, `libpcap-dev`, `libsctp-dev` e `libncurses5-dev`. A opção `-no-install-recommends` é utilizada para evitar a instalação de pacotes recomendados, resultando em uma imagem mais leve.

O `wget` é utilizado para baixar o arquivo `tarball` da versão específica do *SIPp* a partir do *GitHub*. O arquivo é então descompactado usando `tar`, e a partir do diretório descompactado, o *SIPp* é compilado com o *script* `build.sh` usando a opção `-full`, que garante que todos os módulos necessários sejam construídos.

Após a compilação, a porta padrão para o protocolo SIP, 5060, é exposta:

Código A.7 – Exposição da Porta Padrão SIP

```
1 EXPOSE 5060
```

Esta linha garante que a porta 5060 estará disponível para conexões externas ao contêiner, permitindo que o *SIPp* funcione como um agente usuário de servidor (UAS) em testes de SIP.

Por fim, o comando padrão que será executado quando o contêiner for iniciado é definido:

Código A.8 – Execução do *SIPp* como UAS

```
1 CMD ["/sipp/sipp-3.7.2/sipp", "-sn", "uas"]
```

Este comando executa o *SIPp* no modo UAS, que é comumente utilizado em cenários de teste de SIP para simular um servidor de chamadas. A versão específica do *SIPp* é chamada diretamente, utilizando o caminho do binário compilado.

A.1.3 Configurando Docker Compose

O *Docker Compose* apresentado é utilizado para definir e orquestrar múltiplos serviços em contêineres *Docker*, permitindo a criação de um ambiente completo para testes e desenvolvimento de aplicações

Foram definidos quatro serviços: três instâncias do *SIPp* (UAS), que serão utilizadas como nodos do balanceamento de carga, e um serviço *Nginx* configurado como *proxy* reverso, utilizado como balanceador de carga.

O primeiro serviço definido é o `sip_uas_1`:

Código A.9 – Serviço *SIPp* UAS

```
services:
  sip_uas_1: &sip_uas
    build:
      context: ./SIPp
      dockerfile: Dockerfile
    restart: always
```

- `build`: Especifica que o serviço `sip_uas_1` deve ser construído a partir do Dockerfile localizado no diretório `./SIPp`. O contexto (`context`) define o diretório de trabalho e o arquivo Dockerfile a ser utilizado para a construção da imagem.
- `restart`: O parâmetro `restart: always` garante que o contêiner será reiniciado automaticamente caso falhe ou seja interrompido, assegurando a continuidade dos testes.

Os serviços `sip_uas_2` e `sip_uas_3` são definidos como cópias do serviço `sip_uas_1`, estas de extrema importância para o trabalho, pois representam copias idênticas dos nodos que serão distribuídos a carga do balanceador. As cópias foram criados utilizando a âncora `*sip_uas`:

Código A.10 – Âncora do *SIPp* UAS

```
  sip_uas_2:
    <<: *sip_uas
  sip_uas_3:
    <<: *sip_uas
```

- `«: *sip_uas`: Esta sintaxe `YAML` permite a reutilização das configurações do serviço `sip_uas_1` para os serviços `sip_uas_2` e `sip_uas_3`, evitando duplicação de código e facilitando a manutenção.

O serviço `my_nginx_proxy` é responsável por configurar um servidor *Nginx* para atuar como *proxy* para as instâncias do *SIPp*:

Código A.11 – Serviço *Nginx*

```
my_nginx_proxy:
  build:
    context: ./nginx
    dockerfile: Dockerfile
  volumes:
    - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
  ports:
    - "5060:5060/udp"
  depends_on:
    - sip_uas_1
    - sip_uas_2
    - sip_uas_3
  command: nginx -g 'daemon off;'
```

- `build`: Especifica que o serviço *Nginx* deve ser construído a partir do Dockerfile localizado no diretório `./nginx`.
- `volumes`: O arquivo `nginx.conf` localizado no diretório `./nginx` do *host* é montado como um volume no contêiner, garantindo que o *Nginx* utilize essa configuração específica, que será explicada na secção seguinte. O `:ro` indica que o volume é montado em modo somente leitura.
- `ports`: A porta `5060/udp` do *host* é mapeada para a porta `5060/udp` do contêiner, permitindo o tráfego SIP entre o *host* e o contêiner.
- `depends_on`: Esta opção garante que o *Nginx* só será iniciado após a inicialização bem-sucedida dos três serviços *SIPp* (`sip_uas_1`, `sip_uas_2`, `sip_uas_3`).
- `command`: O comando `nginx -g 'daemon off;'` é utilizado para iniciar o *Nginx* em primeiro plano, mantendo o processo ativo dentro do contêiner.

A.2 Configurando Proxy Reverso

O arquivo de configuração *Nginx* apresentado define um servidor *Nginx* como um *proxy* reverso e balanceador de carga para gerenciar chamadas SIP.

O arquivo começa especificando o local onde o processo principal do *Nginx* armazenará seu Identificador de processo (PID), o que é essencial para o gerenciamento do serviço, seguido do caminho para o registro de *logs* de erros, facilitando a depuração e o monitoramento do servidor.

Código A.12 – Informações Inicias

```
pid /run/nginx.pid;
error_log /var/log/nginx/error.log;
include /etc/nginx/modules-enabled/*.conf;
```

A linha include `/etc/nginx/modules-enabled/*.conf`; permite que módulos do *Nginx* sejam carregando as funcionalidades disponíveis.

Em seguida é feito a configuração define o número de processos de trabalho (`worker_processes auto;`), o que ajusta automaticamente a quantidade de processos de trabalho de acordo com o número de núcleos disponíveis na *CPU*, otimizando o desempenho.

Código A.13 – Configuração da quantidade de Conexões

```
worker_processes auto;
events {
  worker_connections 10240;
}
```

No bloco `events`, o parâmetro `worker_connections 10240;` determina o número máximo de conexões simultâneas que cada processo de trabalho pode gerenciar, garantindo que o servidor possa lidar com um alto volume de conexões simultâneas.

Código A.14 – Definição do `Stream`

```
stream {
  upstream backend_udp {
    server sip_uas_1:5060;
    server sip_uas_2:5060;
    server sip_uas_3:5060;
  }
```

O bloco `stream` configura o *Nginx* para funcionar como um *proxy* de nível de transporte para o protocolo SIP, com o bloco `upstream` definindo um grupo de servidores backend denominado `backend_udp`, que consiste em três servidores SIP, identificados como `sip_uas_1`, `sip_uas_2` e `sip_uas_3`, todos escutando na porta 5060. Esta configuração permite o balanceamento de carga, distribuindo as chamadas SIP de forma uniforme entre as instâncias do *SIPp*, aumentando a eficiência e a resiliência do sistema.

Caso a linha `least_conn;` seja adicionada ao bloco `upstream`, o algoritmo de balanceamento de carga padrão, que é o *round-robin*, será substituído pelo algoritmo *least-connection*.

Código A.15 – Especificação do Servidor

```
  server {
    listen 5060 udp reuseport;
    proxy_pass backend_udp;
  }
}
```

Por fim, o bloco `server` dentro de stream especifica que o servidor *Nginx* deve escutar na porta 5060 para conexões UDP (`listen 5060 udp reuseport;`). O parâmetro `reuseport` permite que múltiplos processos escutem na mesma porta, melhorando o balanceamento de carga e a escalabilidade. A diretiva `proxy_pass backend_udp;` encaminha o tráfego recebido para o grupo de servidores definidos no bloco `upstream`, completando a configuração do *proxy* reverso e balanceador de carga para as chamadas SIP.

A.3 Configurando SIPp

O arquivo `XML` fornecido configura um cenário de teste para o *SIPp*, em que um UAC realiza chamadas SIP com transmissão de mídia RTP. Esse cenário é utilizado para simular o envio e recepção de mensagens SIP, incluindo a reprodução de um fluxo RTP a partir de um arquivo de áudio `WAV`.

Código A.16 – Configuração da mensagem `SIP INVITE`

```
  <send retrans="500">
    <![CDATA[

      INVITE sip:[service]@[remote_ip]:[remote_port] SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
      From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[pid]SIPpTag09[call_number]
      To: [service] <sip:[service]@[remote_ip]:[remote_port]>
      Call-ID: [call_id]
      CSeq: 1 INVITE
      Contact: sip:sipp@[local_ip]:[local_port]
      Max-Forwards: 70
      Subject: Performance Test
      Content-Type: application/sdp
      Content-Length: [len]

      v=0
      o=user1 53655765 2353687637 IN IP[local_ip_type] [local_ip]
      s=-
      c=IN IP[local_ip_type] [local_ip]
      t=0 0
      m=audio [media_port] RTP/AVP 8 101
      a=rtpmap:8 PCMA/8000
```

```
23      a=rtpmap:101 telephone-event/8000
24      a=fmtp:101 0-11,16
25
26    ]]>
27  </send>
```

O cenário começa com a configuração de uma mensagem SIP `INVITE`, que é enviada para o servidor remoto com o objetivo de iniciar uma chamada. O campo `retrans="500"` especifica que a retransmissão da mensagem será realizada a cada 500 milissegundos até que uma resposta seja recebida. Dentro do bloco `<send>`, a mensagem `INVITE` é definida com diversos parâmetros SIP, como o endereço IP local e remoto, as portas utilizadas, e o conteúdo Protocolo de descrição de sessão (SDP), que define as características da mídia a ser usada na chamada, como o tipo de codificação (`PCMA/8000`) e os eventos DTMD (`telephone-event/8000`).

Código A.17 – Configuração para previsão de possíveis repostas a ser recebidas

```
1   <recv response="100" optional="true">
2   </recv>
3
4   <recv response="180" optional="true">
5   </recv>
6
7   <!-- By adding rrs="true" (Record Route Sets), the route sets         -->
8   <!-- are saved and used for following messages sent. Useful to test   -->
9   <!-- against stateful SIP proxies/B2BUAs.                             -->
10  <recv response="200" rtd="true" crlf="true">
11  </recv>
```

A seguir, o cenário prevê a recepção de diferentes respostas SIP utilizando blocos `<recv>`. A primeira resposta, 100, indica que o servidor está processando a solicitação `INVITE`. A segunda, 180, indica que o telefone do destinatário está tocando. A terceira, 200, confirma que a chamada foi estabelecida com sucesso, ativando o rastreamento da rota (`rtd="true"`) para mensagens subsequentes.

Código A.18 – Configuração da mensagem SIP `ACK` e Execução de arquivo de mídia

```
1   <send>
2     <![CDATA[
3
4       ACK sip:[service]@[remote_ip]:[remote_port] SIP/2.0
5       Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
6       From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[pid]SIPpTag09[call_number]
7       To: [service] <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
8       Call-ID: [call_id]
9       CSeq: 1 ACK
```

```
10      Contact: sip:sipp@[local_ip]:[local_port]
11      Max-Forwards: 70
12      Subject: Performance Test
13      Content-Length: 0
14
15    ]]>
16  </send>
17
18  <!-- Play a wav file (RTP stream)                    -->
19  <nop>
20    <action>
21      <exec rtp_stream="audio/betinho.wav,1,8,PCMA/8000"/>
22    </action>
23  </nop>
```

Após a confirmação da chamada, o cenário envia um `ACK` para confirmar a recepção da resposta `200 OK`. Este `ACK` completa a fase de estabelecimento da chamada, e em seguida, o fluxo de mídia começa com a execução de um arquivo `WAV` definido no bloco `<nop>`. Neste ponto, o comando `<exec rtp_stream="audio/betinho.wav,1,8,PCMA/8000"/>` instrui o *SIPp* a reproduzir o arquivo de áudio `betinho.wav` como um fluxo RTP usando a codificação *PCMA* a 8000 Hz.

Código A.19 – Configuração da mensagem `SIP BYE`

```
1   <pause milliseconds="30000"/>
2
3   <!-- <pause milliseconds="1000"/> -->
4
5   <!-- The 'crlf' option inserts a blank line in the statistics report. -->
6   <send retrans="500">
7     <![CDATA[
8
9       BYE sip:[service]@[remote_ip]:[remote_port] SIP/2.0
10      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
11      From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[pid]SIPpTag09[call_number]
12      To: [service] <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
13      Call-ID: [call_id]
14      CSeq: 2 BYE
15      Contact: sip:sipp@[local_ip]:[local_port]
16      Max-Forwards: 70
17      Subject: Performance Test
18      Content-Length: 0
19
20    ]]>
21  </send>
22
23  <recv response="200" crlf="true">
```

```
24   </recv>
```

O cenário então inclui uma pausa de 30 segundos (`<pause milliseconds="30000"/>`), representando a duração aproximada do fluxo RTP, permitindo que a mídia seja reproduzida durante esse período. Esta pausa visa simular com precisão o comportamento de uma chamada real. Durante uma chamada de voz ou vídeo, o tempo de execução do arquivo de mídia corresponde ao período de transmissão entre as partes envolvidas.

Dessa forma, ao reproduzir um arquivo de mídia, o *SIPp* aguarda a duração total do arquivo para garantir que o fluxo de mídia seja transmitido de maneira contínua e realista, proporcionando testes mais próximos de um cenário de comunicação real. Essa espera também tem como objetivo sincronizar as mensagens SIP com o tempo da chamada, assegurando que as respostas e a finalização do teste ocorram no momento adequado, após a conclusão da mídia.

Após a pausa, o cenário finaliza a chamada com uma mensagem `BYE`, encerrando a sessão SIP. A mensagem `BYE` também é retransmitida a cada 500 milissegundos até que uma resposta `200 OK` seja recebida, confirmando o término da chamada.

Código A.20 – Definição das tabelas de repartição de tempo de resposta e duração

```
1  <!-- definition of the response time repartition table (unit is ms)   -->
2  <ResponseTimeRepartition value="10, 20, 30, 40, 50, 100, 150, 200"/>
3
4  <!-- definition of the call length repartition table (unit is ms)     -->
5  <CallLengthRepartition value="10, 50, 100, 500, 1000, 5000, 10000"/>
```

Além das mensagens SIP, o cenário define tabelas de repartição de tempo de resposta e de duração das chamadas, requiridas pelo *SIPp*, com os elementos `<ResponseTimeRepartition>` e `<CallLengthRepartition>`, respectivamente. Esses elementos permitem que o *SIPp* colete estatísticas detalhadas sobre a distribuição dos tempos de resposta e a duração das chamadas durante a execução do cenário, proporcionando métricas valiosas para a análise de desempenho.

A.3.1 Execução do cenário de testes

Para facilitar o processo de execução do cenário, foi feito uma automação do cenário gerenciado por meio de um `Makefile`, cuja função principal é facilitar a execução de tarefas repetitivas, como a inicialização, construção e limpeza dos contêineres. A seção inicial do `Makefile` define diferentes alvos para gerenciar o ciclo de vida dos contêineres *Docker*.

Código A.21 – `Makefile` para automação da execução do *Docker*

```
all: stop start

clean:
  docker ps -a -q | docker rm
  docker images -q | xargs docker rmi

build:
  make -C SIPp
  make -C nginx

start:
  ulimit -n 10240
  docker-compose up -d

stop:
  docker-compose down

full:
  make stop
  make -i clean
  make build
  make start
```

O alvo `all`, que é o alvo padrão, chama as tarefas `stop` e `start`, garantindo que quaisquer contêineres existentes sejam parados antes de iniciar uma nova instância dos serviços definidos no *Docker Compose*. A tarefa `clean` remove todos os contêineres e imagens *Docker* criados anteriormente, garantindo que a próxima execução ocorra em um ambiente limpo. A tarefa `build` executa os contêineres do *SIPp* e do *Nginx* a partir de seus respectivos diretórios.

A tarefa `start` é responsável por iniciar os contêineres, configurando um limite de arquivos abertos (`ulimit -n 10240`) para suportar um grande número de conexões simultâneas. A tarefa `stop` finaliza os contêineres utilizando o comando `docker-compose down`. Finalmente, a tarefa `full` realiza uma execução completa, parando os contêineres existentes, limpando o ambiente, reconstruindo os contêineres e iniciando os serviços novamente.

Após o início dos contêineres, a captura de tráfego de rede é iniciada utilizando o *Tshark*, junto com o `tc` para simular condições adversas de rede, como latência e perda de pacotes. O comando `tc qdisc add dev <nome da interface> root netem delay 20ms 5ms distribution normal loss 15%` introduz um atraso de 20ms com uma variação de 5ms e uma perda de pacotes de 15%, emulando uma rede com condições de qualidade de serviço degradada.

O comando `tshark -i <nome da interface> -w scenario_capture.pcap -a`

`duration:7500` captura o tráfego de rede durante 7500 segundos, salvando-o em um arquivo pcap. Simultaneamente, o *SIPp* é executado como UAC, utilizando o cenário XML descrito anteriormente, com o comando `sipp -sf uac_with_media_long.xml -r 10 -rp 5000 -l 120 <ip do nginx>:5060`. Esse comando simula 10 chamadas por segundo, com um intervalo de 5000 milissegundos entre cada chamada, e limita o número total de chamadas simultâneas a 120, enviando as solicitações SIP para o endereço IP do *proxy Nginx*, que, por sua vez, distribui as chamadas entre as instâncias *SIPp* configuradas.

Após a captura do tráfego, os arquivos pcap gerados são analisados utilizando o Tshark. O comando `tshark -r scenario_capture.pcap -Y "sip-T fields -e sip.To -e sip.Call-ID -e ip.dst > sip_calls.txt` extrai as mensagens SIP, filtrando por campos específicos, como o destinatário, o ID da chamada e o endereço IP de destino, e salva os resultados em um arquivo de texto. O comando `tshark -r scenario_capture.pcap -q -z rtp,streams > scenario_rtpStream.txt` gera um relatório detalhado dos fluxos RTP capturados, permitindo uma análise profunda da qualidade da mídia transmitida.

A.3.2 Processamento dos Dados

Para processar os dados obtidos pelos relatórios gerados pelo *TShark*, foram desenvolvidos *scripts Python* que realizam a leitura, processamento e visualização dos dados de fluxos RTP. Esses *scripts* têm como objetivo extrair métricas de latência, *jitter*, perda de pacotes e outras informações relevantes, gerando gráficos para análise visual.

APÊNDICE B – REPOSITÓRIO DA APLICAÇÃO

O cenário desenvolvido e suas configurações estão disponíveis no repositório do Github: https://github.com/ArthurAnastopulos/tcc-load-balacing

APÊNDICE C – ARQUIVOS DE CONFIGURAÇÃO

Código C.1 – docker-compose.yml

```yaml
version: '3'
services:
  sip_uas_1: &sip_uas
    build:
      context: ./SIPp
      dockerfile: Dockerfile
    restart: always
  sip_uas_2:
    <<: *sip_uas
  sip_uas_3:
    <<: *sip_uas
  my_nginx_proxy:
    build:
      context: ./nginx
      dockerfile: Dockerfile
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    ports:
      - "5060:5060/udp"
    depends_on:
      - sip_uas_1
      - sip_uas_2
      - sip_uas_3
    command: nginx -g 'daemon off;'
```

Código C.2 – Nginx Dockefile

```
# Use a Debian base image
FROM debian:buster-slim

# Update repositories and install Nginx and Certbot
RUN apt-get update && \
    apt-get install -y nginx certbot python3-certbot-nginx libnginx-mod-stream
    libnginx-mod-http-geoip libnginx-mod-rtmp && \
    apt-get clean

# Expose HTTP and HTTPS ports
EXPOSE 80/tcp
```

```
EXPOSE 443/tcp

# Add a script to automatically renew SSL certificates
COPY certbot_renew.sh /usr/local/bin/
RUN chmod +x /usr/local/bin/certbot_renew.sh

# Start a shell for interactive use
CMD ["/bin/bash"]
```

Código C.3 – SIPp Dockefile

```
# Use a imagem Debian Slim como base
FROM debian:buster-slim
ARG SIPP_VERSION="3.7.2"

WORKDIR /sipp
RUN apt-get update && \
    apt-get install -y --no-install-recommends build-essential cmake wget libssl-dev libpcap-dev libsctp-dev libncurses5-dev && \
    wget --no-check-certificate "https://github.com/SIPp/sipp/releases/download/v$SIPP_VERSION/sipp-$SIPP_VERSION.tar.gz" && \
    tar xzf sipp-$SIPP_VERSION.tar.gz -C . && \
    cd sipp-$SIPP_VERSION && \
    ./build.sh --full

EXPOSE 5060
CMD ["/sipp/sipp-3.7.2/sipp", "-sn", "uas"]
```

Código C.4 – uac_with_media_long.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE scenario SYSTEM "sipp.dtd">

<!-- This program is free software; you can redistribute it and/or      -->
<!-- modify it under the terms of the GNU General Public License as     -->
<!-- published by the Free Software Foundation; either version 2 of the -->
<!-- License, or (at your option) any later version.                    -->
<!--                                                                    -->
<!-- This program is distributed in the hope that it will be useful,    -->
<!-- but WITHOUT ANY WARRANTY; without even the implied warranty of     -->
<!-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the      -->
<!-- GNU General Public License for more details.                       -->
<!--                                                                    -->
<!-- You should have received a copy of the GNU General Public License  -->
<!-- along with this program; if not, write to the                      -->
<!-- Free Software Foundation, Inc.,                                    -->
<!-- 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA             -->
<!--                                                                    -->
```

```
<!--                 Sipp 'uac' scenario with pcap (rtp) play           -->
<!--                                                                    -->

<scenario name="UAC with media">
  <!-- In client mode (sipp placing calls), the Call-ID MUST be         -->
  <!-- generated by sipp. To do so, use [call_id] keyword.                -->
  <send retrans="500">
    <![CDATA[

      INVITE sip:[service]@[remote_ip]:[remote_port] SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
      From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[pid]SIPpTag09[call_number]
      To: [service] <sip:[service]@[remote_ip]:[remote_port]>
      Call-ID: [call_id]
      CSeq: 1 INVITE
      Contact: sip:sipp@[local_ip]:[local_port]
      Max-Forwards: 70
      Subject: Performance Test
      Content-Type: application/sdp
      Content-Length: [len]

      v=0
      o=user1 53655765 2353687637 IN IP[local_ip_type] [local_ip]
      s=-
      c=IN IP[local_ip_type] [local_ip]
      t=0 0
      m=audio [media_port] RTP/AVP 8 101
      a=rtpmap:8 PCMA/8000
      a=rtpmap:101 telephone-event/8000
      a=fmtp:101 0-11,16

    ]]>
  </send>

  <recv response="100" optional="true">
  </recv>

  <recv response="180" optional="true">
  </recv>

  <!-- By adding rrs="true" (Record Route Sets), the route sets         -->
  <!-- are saved and used for following messages sent. Useful to test   -->
  <!-- against stateful SIP proxies/B2BUAs.                             -->
  <recv response="200" rtd="true" crlf="true">
  </recv>

  <!-- Packet lost can be simulated in any send/recv message by         -->
```

```
66  <!-- by adding the 'lost = "10"'. Value can be [1-100] percent.       -->
67  <send>
68    <![CDATA[
69
70      ACK sip:[service]@[remote_ip]:[remote_port] SIP/2.0
71      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
72      From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[pid]SIPpTag09[call_number]
73      To: [service] <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
74      Call-ID: [call_id]
75      CSeq: 1 ACK
76      Contact: sip:sipp@[local_ip]:[local_port]
77      Max-Forwards: 70
78      Subject: Performance Test
79      Content-Length: 0
80
81    ]]>
82  </send>
83
84  <!-- Play a wav file (RTP stream)                        -->
85  <nop>
86    <action>
87      <exec rtp_stream="audio/betinho.wav,1,8,PCMA/8000"/>
88    </action>
89  </nop>
90
91  <!-- Pause 2 minutes, which is approximately the duration of the      -->
92  <!-- PCAP file                                                        -->
93  <pause milliseconds="30000"/>
94
95  <!-- <pause milliseconds="1000"/> -->
96
97  <!-- The 'crlf' option inserts a blank line in the statistics report. -->
98  <send retrans="500">
99    <![CDATA[
100
101     BYE sip:[service]@[remote_ip]:[remote_port] SIP/2.0
102     Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
103     From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[pid]SIPpTag09[call_number]
104     To: [service] <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
105     Call-ID: [call_id]
106     CSeq: 2 BYE
107     Contact: sip:sipp@[local_ip]:[local_port]
108     Max-Forwards: 70
109     Subject: Performance Test
110     Content-Length: 0
111
112   ]]>
```

```
113   </send>
114
115   <recv response="200" crlf="true">
116   </recv>
117
118   <!-- definition of the response time repartition table (unit is ms)   -->
119   <ResponseTimeRepartition value="10, 20, 30, 40, 50, 100, 150, 200"/>
120
121   <!-- definition of the call length repartition table (unit is ms)     -->
122   <CallLengthRepartition value="10, 50, 100, 500, 1000, 5000, 10000"/>
123
124 </scenario>
```

Código C.5 – certbot_renew.sh

```
1 #!/bin/bash
2
3 # Renovar os certificados SSL usando o Certbot
4 certbot renew
5 # Recarregar o Nginx para aplicar as alterações
6 nginx -s reload
```

Código C.6 – nginx.conf

```
1  pid /run/nginx.pid;
2  error_log /var/log/nginx/error.log;
3  include /etc/nginx/modules-enabled/*.conf;
4
5  worker_processes auto;
6  events {
7    worker_connections 10240;
8  }
9
10 stream {
11   upstream backend_udp {
12     server sip_uas_1:5060;
13     server sip_uas_2:5060;
14     server sip_uas_3:5060;
15   }
16
17   server {
18     listen 5060 udp reuseport;
19     proxy_pass backend_udp;
20   }
21 }
```

Código C.7 – Makefile

```
all: stop start

clean:
  docker ps -a -q | docker rm
  docker images -q | xargs docker rmi

build:
  make -C SIPp
  make -C nginx

start:
  ulimit -n 10240
  docker-compose up -d

stop:
  docker-compose down

full:
  make stop
  make -i clean
  make build
  make start
```

Código C.8 – pcap_extract.py

```
import argparse
from datetime import datetime, timedelta
import matplotlib.dates as mdates
import matplotlib.pyplot as plt
import numpy as np

def extract_numeric(value):
    """ Extracts numeric value from a string that may contain additional characters
     """
    try:
        return float(value.split()[0])
    except ValueError:
        return None

def convert_to_absolute_percentage(value):
    """ Convert a percentage string to an absolute float """
    try:
        # Remove parentheses and percent sign, then convert to float
        value = value.strip(' ()%')
        number = abs(float(value))
        return number
    except ValueError:
```

```
22          return None
23
24  def process_txt(txt_file):
25      timestamps = []
26      latencies = []
27      jitters = []
28      packets = []
29      losses = []
30      stream_ids = []
31
32      with open(txt_file, 'r') as file:
33          lines = file.readlines()
34
35      # Identify the start of the data section
36      data_started = False
37      for line in lines:
38          line = line.strip()
39
40          # Skip header and separator lines
41          if line.startswith('=') or not line:
42              continue
43
44          if not data_started:
45              # Find the line where data starts
46              if line.startswith('Start time'):
47                  data_started = True
48              continue
49
50          # Split line into fields
51          fields = line.split()
52          if len(fields) < 16:
53              print(f"Skipping malformed line: {line}")
54              continue
55
56          try:
57              start_time = fields[0]
58              end_time = fields[1]
59              src_ip = fields[2]
60              src_port = int(fields[3])
61              dest_ip = fields[4]
62              dest_port = int(fields[5])
63              ssrc_hex = fields[6]
64              payload = fields[7]
65              pkts = int(fields[8])
66              lost_str = fields[9]
67              lost_ptc = convert_to_absolute_percentage(fields[10])
68              min_delta = extract_numeric(fields[11])
```

```
69              mean_delta = extract_numeric(fields[12])
70              max_delta = extract_numeric(fields[13])
71              min_jitter = extract_numeric(fields[14])
72              mean_jitter = extract_numeric(fields[15])
73              max_jitter = extract_numeric(fields[16])
74              # Convert SSRC from hex to decimal
75              ssrc = int(ssrc_hex, 16)
76
77              # Convert time to ISO format
78              absolute_start_time = "00:00:00"
79              start_time_ = datetime.strptime(absolute_start_time, "%H:%M:%S")
80              start_time_absolute = start_time_ + timedelta(seconds=float(start_time)
    )
81              end_time_absolute = start_time_ + timedelta(seconds=float(end_time))
82
83              timestamp_ns = int(start_time_absolute.timestamp())
84              # Extract numeric part of 'lost'
85              lost = extract_numeric(lost_str)
86
87              # Collect data for plotting
88              timestamps.append(start_time_absolute)
89              latencies.append(mean_delta)  # or min_delta, max_delta as needed
90              jitters.append(mean_jitter)   # or min_jitter, max_jitter as needed
91              packets.append(pkts)
92              losses.append(lost_ptc)
93              stream_ids.append(ssrc)
94
95          except (ValueError, IndexError) as e:
96              print(f"Error processing line: {line}")
97              print(f"Exception: {e}")
98
99      # Plot metrics
100     plot_metrics(timestamps, latencies, jitters)
101     plot_loss_jitter_relation(losses, jitters)
102     plot_loss_latencies_relation(latencies, jitters)
103
104 def plot_metrics(timestamps, latencies, jitters):
105     plt.figure(figsize=(14, 7))
106
107     time_format = mdates.DateFormatter('%H:%M:%S')  # Formatter for the time part
    only
108
109     plt.subplot(2, 1, 1)
110     plt.plot(timestamps, latencies, linestyle='-', color='b')
111     plt.title('Latency Over Time')
112     plt.xlabel('Time')
113     plt.ylabel('Latency (ms)')
```

```
114     plt.grid(True)
115     plt.gca().xaxis.set_major_formatter(time_format)  # Set the formatter for the x
        -axis
116     plt.xticks(rotation=45)
117     plt.xlim([min(timestamps), max(timestamps)])
118 
119     plt.subplot(2, 1, 2)
120     plt.plot(timestamps, jitters, linestyle='-', color='r')
121     plt.title('Jitter Over Time')
122     plt.xlabel('Time')
123     plt.ylabel('Jitter (ms)')
124     plt.grid(True)
125     plt.gca().xaxis.set_major_formatter(time_format)  # Set the formatter for the x
        -axis
126     plt.xticks(rotation=45)
127     plt.xlim([min(timestamps), max(timestamps)])
128 
129     plt.tight_layout()
130     plt.show()
131 
132 def plot_loss_jitter_relation(losses, jitters):
133     plt.figure(figsize=(14, 7))
134 
135     hb = plt.hexbin(losses, jitters, gridsize=50, cmap='viridis', mincnt=1)
136     plt.colorbar(hb, label='Count')
137 
138     plt.title('Hexbin Plot of Jitter vs. Packet Loss')
139     plt.xlabel('Packet Loss Percentage')
140     plt.ylabel('Mean Jitter')
141     plt.grid(True)
142     plt.tight_layout()
143     plt.show()
144 
145 def plot_loss_latencies_relation(latencies, jitters):
146     plt.figure(figsize=(14, 7))
147 
148     hb = plt.hexbin(latencies, jitters, gridsize=50, cmap='viridis', mincnt=1)
149     plt.colorbar(hb, label='Count')
150 
151     plt.title('Hexbin Plot of latencies vs. Packet Loss')
152     plt.xlabel('Packet Loss Percentage')
153     plt.ylabel('Mean latencies')
154     plt.grid(True)
155     plt.tight_layout()
156     plt.show()
157 
158 if __name__ == "__main__":
```

```
159     parser = argparse.ArgumentParser(description='Extract RTP Stream and plot
        metrics')
160     parser.add_argument('txt_file', type=str, help='Path to the TXT file containing
         RTP stream data')
161 
162     args = parser.parse_args()
163     process_txt(args.txt_file)
```

Código C.9 – show_sip_data.py

```
1  import sys
2  import csv
3  import matplotlib.pyplot as plt
4  
5  # Verify command-line arguments
6  if len(sys.argv) != 5:
7      print("Usage: python script.py <input_txt_file> <ip_node1> <ip_node2> <ip_node3
       >")
8      sys.exit(1)
9  
10 # Capture command-line arguments
11 input_txt = sys.argv[1]
12 ip_node1 = sys.argv[2]
13 ip_node2 = sys.argv[3]
14 ip_node3 = sys.argv[4]
15 
16 # Initialize counters for each node
17 node_calls = {ip_node1: 0, ip_node2: 0, ip_node3: 0}
18 
19 # Process the TXT file line by line
20 with open(input_txt, 'r') as file:
21     for line in file:
22         destination_ip = line.strip().split()[-1]  # Get the last element as the
       destination IP
23         if destination_ip in node_calls:
24             node_calls[destination_ip] += 1
25 
26 # Calculate the total number of successful calls
27 total_calls = sum(node_calls.values())
28 
29 # Calculate the difference for each node compared to the total
30 percentages = {node: (count / total_calls) * 100 for node, count in node_calls.
       items()}
31 
32 # Generate the histogram
33 nodes = list(percentages.keys())
34 values = list(percentages.values())
35 
```

```
bars = plt.bar(nodes, values)
plt.xlabel('Nodos')
plt.ylabel('Porcentagem de chamadas bem-sucedidas (%)')
# plt.title('Percentage of Successful Calls per Node')

# Add percentage labels inside the bars
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval - 5, f'{yval:.2f}%', ha='center'
    , va='bottom', color='white', fontweight='bold')

plt.show()
```

Código C.10 – tempoRespostaSIP.py

```
import matplotlib.pyplot as plt

# Inicializar dicionário para contar SIP INVITE por segundo
invite_count = {}

# Abrir o arquivo e processar linha por linha
with open('sip_lc_51_invite_degradado.csv', 'r') as file:
    for line in file:
        # Remover espaços em branco e dividir a linha
        parts = line.strip().split('\t')

        # Verificar se a linha tem pelo menos três partes e se o método é 'INVITE'
        if len(parts) >= 3 and parts[2] == 'INVITE':
            timestamp = float(parts[0])  # Pegar o timestamp
            second = int(timestamp)        # Converter para inteiro (segundo)

            # Contar os SIP INVITE por segundo
            if second not in invite_count:
                invite_count[second] = 0
            invite_count[second] += 1

# Preparar os dados para plotagem
seconds = sorted(invite_count.keys())
counts = [invite_count[sec] for sec in seconds]

# Plotar o gráfico
plt.figure(figsize=(12, 6))
plt.plot(seconds, counts)
# plt.title('Quantidade de SIP INVITE por segundo')
plt.xlabel('Tempo (segundos)')
plt.ylabel('Quantidade de SIP INVITE')
plt.grid()
plt.show()
```

Código C.11 – graficoTotalCalls.py

```
import matplotlib.pyplot as plt
import sys

# Função principal
def process_csv(csv_file, nodes):
    # Inicializar dicionário para contar chamadas SIP por nó
    node_count = {node: 0 for node in nodes}
    total_calls = 0

    # Abrir o arquivo CSV e processar linha por linha
    with open(csv_file, 'r') as file:
        for line in file:
            # Remover espaços em branco e dividir a linha
            parts = line.strip().split('\t')

            # Verificar se a linha tem pelo menos três partes
            if len(parts) >= 3:
                node = parts[2]  # O nó está na terceira coluna

                # Verificar se o nó está nos nós fornecidos
                if node in node_count:
                    node_count[node] += 1
                    total_calls += 1

    # Exibir o total de chamadas SIP no cenário e o total por nó
    print(f"Total de chamadas SIP no cenário: {total_calls}")
    for node, count in node_count.items():
        print(f"Total de chamadas para {node}: {count}")

    # Preparar os dados para o gráfico de pizza
    nodes_in_use = [node for node in nodes if node_count[node] > 0]
    calls_in_use = [node_count[node] for node in nodes_in_use]

    # Criar gráfico de pizza
    plt.figure(figsize=(8, 8))
    plt.pie(calls_in_use, labels=nodes_in_use, autopct='%1.1f%%', startangle=90)
    # plt.title('Distribuição de Chamadas SIP entre os Nós do Balanceador de Carga
    ')
    plt.axis('equal')  # Garantir que o gráfico de pizza seja um círculo
    plt.show()


if __name__ == "__main__":
    # Argumentos: nome do arquivo CSV seguido pelos endereços dos nós
```

```
44      if len(sys.argv) < 3:
45          print("Uso: python script.py <arquivo_csv> <endereco_no_1> <endereco_no_2>
        ...")
46          sys.exit(1)
47
48      csv_file = sys.argv[1]
49      nodes = sys.argv[2:]
50
51      # Processar o CSV e gerar o gráfico
52      process_csv(csv_file, nodes)
```