PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

EDGEEMUPY: UMA SOLUÇÃO PARA EMULAR AMBIENTES DE COMPUTAÇÃO NA BORDA

JOÃO VICTOR ZUCCO MARMENTINI

Trabalho de Conclusão II apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Tiago Ferreto

Porto Alegre
2024

EDGEEMUPY: UMA SOLUÇÃO PARA EMULAR AMBIENTES DE COMPUTAÇÃO NA BORDA

RESUMO

A computação na borda, impulsionada pelo aumento de dispositivos conectados e aplicações que precisam de baixa latência, trouxeram a necessidades de ferramentas eficientes para o desenvolvimento, teste e validação de novas estratégias. A fim de criar uma solução modular e escalável, esse trabalho propõe o estudo e desenvolvimento do EdgeEmuPy, uma ferramenta para emulação de computação na borda baseado em Python e Docker. A contribuição aborda desafios de mobilidade, qualidade de serviço e alocação de recursos e permite o controle em tempo real por uma API e a integração de protocolos como OSPF para comunicação. O EdgeEmuPy demonstrou desempenho em cenários de migração de dispositivos e testes de carga, validando consistentemente os problemas propostos.

Palavras-Chave: Computação na borda, Emulador, Simulador, Python, Docker.

EDGEEMUPY: AN EMULATION SOLUTION FOR EDGE COMPUTING ENVIROMENTS

ABSTRACT

Driven by the rise of connected devices and applications requiring low latency, edge computing has created the need for efficient tools for the development, testing, and validation of new strategies. To provide a modular and scalable solution, this work proposes the study and development of EdgeEmuPy, a tool for edge computing emulation based on Python and Docker. The contribution addresses challenges related to mobility, quality of service, and resource allocation, enabling real-time control through an API and the integration of protocols such as OSPF for communication. EdgeEmuPy demonstrated consistent performance in scenarios involving device migration and load testing, effectively validating the proposed challenges.

Keywords: Edge computing, Emulator, Simulator, Pyhton, Docker.

SUMÁRIO

1. INTRODUÇÃO

A computação na nuvem transformou significativamente a maneira como os dados são processados e armazenados, trazendo o processamento para um data centers centralizados, permitindo a computação em larga escala. A nuvem permitiu que os dispositivos IoT (Internet das Coisas) se desenvolvesse, oferecendo uma plataforma para consolidar a informação, e eliminando a necessidade de que os dispositivos IoT realizassem tarefas computacionalmente intensas de maneira local.

No entanto, algumas aplicações IoT demandam de baixas latências, inviabilizando a utilização da nuvem. A computação na borda surgiu como um paradigma transformador em resposta ao rápido aumento de aplicações de aparelhos conectados na internet e da necessidade deles de terem aplicações em tempo real [5].

Diferente da computação em nuvem tradicional, a computação na borda processa os dados próximo aos dispositivos, a partir de uma infraestrutura local, que possibilita a otimização do uso de banda e a tomada de decisões em tempo de execução. Ela também pode exercer um papel em que pré-processa os dados não críticos, e envia para a nuvem, permitindo grande flexibilidade.

No entanto, essa mudança na decentralização da nuvem introduz novos desafios. A natureza dinâmica das arquiteturas distribuídas, característica da computação na borda, apresenta dificuldades tanto no desenvolvimento quanto na validação de algoritmos e sistemas. A variabilidade nas condições de rede, a diversidade dos dispositivos e a integração com diferentes camadas de infraestrutura tornam os testes e validações altamente desafiadores. Além disso, a ausência de infraestrutura física adequada para testes práticos pode dificultar a análise de casos de uso e a validação de estratégias.

Para isso, a utilização de simuladores, emuladores e test-beds são essenciais pois permitem validar sistemas de computação em diferentes aspectos, trazendo níveis de realismo diferentes. Simuladores, como o EdgeSimPy [8], permitem modelar cenários complexos de borda com alto nível de abstração, enquanto emuladores como o EmuEdge [9] oferecem maior realismo ao permitir a execução de softwares reais em ambientes controlados.

Esse trabalho tem como objetivo principal o estudo e desenvolvimento de um emulador de computação em borda para a validação de estratégias para o gerenciamento de recursos, gerenciamento de rede, teste de aplicações e infraestrutura necessária para a computação na borda.

Além disso, o EdgeEmuPy tem como foco criar uma infraestrutura baseada em containers que emula o comportamento dos nós, dispositivos e aplicações em um ambiente de computação na borda. A ferramenta permite a execução e avaliação de cenários dinâ-

micos, como o caso de uso de migração de dispositivos e alterações da rede em tempo de execução.

A validação foi realizada por meio do cenário de migração de dispositivos e quebra de rota e ela demonstra como a infraestrutura se comporta diante das mudanças da rede. Também foi realizado testes de carga que avaliam recursos computacionais e desempenho da ferramenta diante de cenários de alta demanda.

O EdgeEmuPy utiliza de Python, para gerenciar dinamicamente os recursos necessários para a criação, atualização e remoção do ambiente Docker, assim como permite que a ferramente seja modular e capaz de ser utilizada como uma API. E para a emulação, containers Docker são aplicados pela suas caraterísticas de alta escalabilidade e baixo custo computacional.

O documento está organizado da seguinte forma. O Capítulo 2 apresenta a fundamentação teórica do trabalho abordando o conceito de computação na borda e ferramentas de suporte para validação. O Capítulo 3 descreve exemplos de simuladores e emuladores atuais para validação de estratégias em computação na borda. O Capítulo 4 apresenta a descrição da contribuição, detalhando as camadas da arquitetura. O Capítulo 5 mostra a avaliação dos casos de uso da ferramenta e testes de carga e o Capítulo 6 as conclusões e trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos de computação na borda, comparando-os com a nuvem, e mostra o desenvolvimento dessa tecnologia, mostrando as características fundamentais e suas vantagens e desvantagens.

2.1 Computação na Borda

A computação na borda surge com o rápido desenvolvimento do *IoT*, devido à grande quantidade de dados gerados por esses dispositivos para serem processados na nuvem. Contudo, mesmo com os avanços na capacidade de computação da nuvem, existe um grande gargalo na velocidade de transmissão da rede, resultando em altas latências.

Shi et al. [6] define computação na borda como “qualquer computação ou utilização de recursos de rede entre o caminho da fonte do dado e a nuvem”. Portanto, ao trazer a computação mais perto da fonte, é possível prover serviços que antes seriam inviáveis para a nuvem devido a latência, como, por exemplo, carros autônomos.

Como mostrado na figura 2.1, a camada intermediária de borda também diminui a carga de processamento e armazenamento da nuvem, atuando como uma cache, porém com uma utilização de energia reduzida e mais econômica [1].

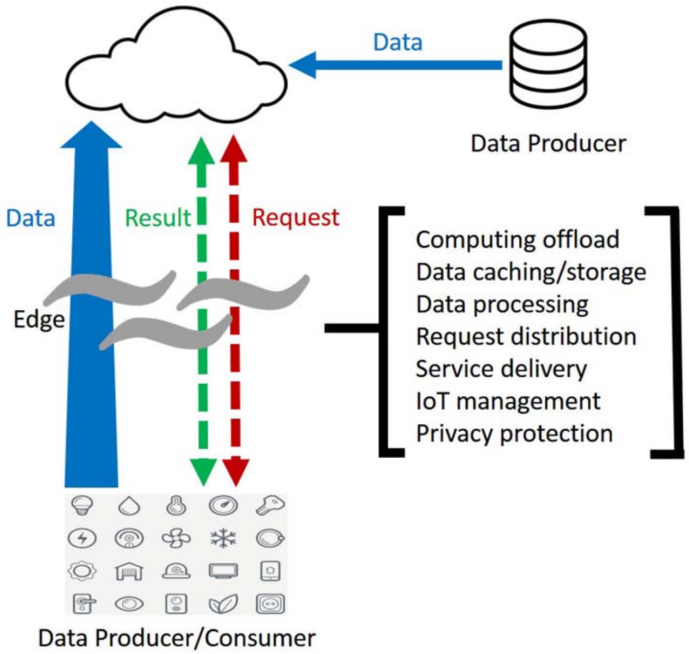

Figura 2.1 – Arquitetura da computação na borda [6]

Por estar geograficamente perto do usuário final, a computação em borda permite serviços que dependem da localização dos usuários [2]. Alguns exemplos de cenários incluem: realidade aumentada, carros autônomos, entre outros.

Contudo, essa nova arquitetura traz algumas desvantagens que devem ser estudadas e analisadas. Entre elas, a que mais se destaca é a dificuldade em prover a segurança dos dados e a defesa contra ataques maliciosos, tanto físicos quanto virtuais [3]. Além da segurança, também existe um aumento considerável na complexidade da rede e da arquitetura, que gera novos desafios.

2.2 Estratégias para validação

A computação na borda traz diversas vantagens e desafios novos para a computação. Diante dessa nova arquitetura, algoritmos que buscam eficiência são desenvolvidos especificamente para as condições apresentadas na borda. Contudo, ainda é difícil testar na prática essa tecnologia, tendo em vista que ainda não existem infraestruturas de borda amplamente disponíveis. Desta forma, é necessário utilizar simuladores, emuladores e *Testbeds* para comprovar os ganhos da computação na borda.

Simulação é uma estratégia que permite desenvolver e testar conceitos de maneira facilmente controlada, precisa e flexível, pois permite que partes específicas sejam isoladas [4]. Por essas características a simulação é amplamente utilizada para experimentação.

A principal diferença entre simulação e emulação é o tempo. Como mostrado anteriormente, a simulação tem o foco em experimentação, onde o ambiente pode ser facilmente controlado e replicado. Para ter essas características, normalmente, as simulações são desenvolvidas para executar no menor tempo possível, que não é necessariamente a realidade do experimento.

Por sua vez, a emulação necessita trabalhar em tempo real, pois essa condição possibilita uma aproximação da realidade [4]. Outra característica importante é que na emulação não temos como replicar exatamente os resultados, devido à característica de tempo e a fidelidade com o real. Isso não necessariamente é um problema, pois nos permite desenvolver gatilhos de segurança que garantem o funcionamento do sistema em condições adversas.

Portanto, a emulação pode ser considerada a metade do caminho entre a arquitetura de um sistema, arquitetura ou aplicação e a realidade, pois é possível ter tanto elementos simulados e controlados, quanto elementos reais.

Enquanto a simulação permite provar os conceitos e a emulação entrega uma visão mais real, ambas estratégias dependem do software e podem se mostrar demoradas para reproduzir ambientes complexos [7]. Para isso, a utilização de *Testbeds* é uma alter-

nativa adotada para alcançar a maior aproximação da realidade. Para gerar essa fidelidade é comum desenvolvê-las em um ambiente totalmente real, visando ter uma visão geral do sistema.

A tabela 2.1 mostra uma comparação entre simulação, emulação e *testbeds*. Por mais que existam diferenças nessas estratégias, elas não devem ser comparadas como competidores, pois cada uma traz potenciais vantagens e desvantagens para a validação de um conceito quando utilizadas em conjunto.

Tabela 2.1 – Comparação entre Simulação, Emulação e *Testbeds*

	Simulação	Emulação	Testbeds
Definição	Modela o comportamento de um sistema ou um processo	Replica o comportamento de um sistema	Utiliza componentes reais para analisar o comportamento de um sistema
Realismo	Baixo realismo	Alcança níveis medianos de realismo	Alto nível de realismo
Custo computacional	Utiliza poucos recursos e por pouco tempo	Requer mais recursos pois emula em tempo real	Alta demanda de recursos para criar um sistema complexo
Complexidade	Permite flexibilidade na modelagem, porém baixa complexidade do sistema simulado	Boa complexidade ao custo de menor flexibilidade	Geralmente possuem baixa flexibilidade, porém alta complexidade
Casos de uso	Usada para modelar sistemas e replicar resultados	Usada para validar interações do sistema	Usada para prototipar sistemas

3. TRABALHOS RELACIONADOS

Este capítulo apresenta uma breve descrição dos principais simuladores e emuladores disponíveis para validar novas estratégias em ambiente de computação na borda.

3.1 EdgeSimPy

EdgeSimPy [8] é um simulador de computação na borda baseado em Python e desenvolvido para análise de recursos como consumo de energia, consumo e tráfico de rede, consumo de computação das aplicações e mobilidade do usuário. Essas características, somadas a capacidade de análise da vida útil das aplicações, diferencia o EdgeSimPy dos outros simuladores e permite a utilização para casos de uso mais relevantes para a atualidade.

A arquitetura do EdgeSimPy permite diferentes maneiras de controle, como manutenção, migração e roteamento, mantendo as característica de abstração do usuários e recursos de rede, mostrado na figura 3.1. As camadas funcionam da seguinte maneira:

1. Camada Core: Implementa o modelo de simulação.
2. Camada Física: Contém as abstrações dos usuários e recursos.
3. Camada Lógica: Representa as aplicações que estão sendo executadas.
4. Camada de Gerenciamento: Controla os recursos de rede e facilita a prototipagem

Para validação foram utilizados alguns casos de uso como a migração de uma aplicação de um nodo da borda para outro e a manutenção de um servidor. Devido ao caráter experimental das simulações, a validação da ferramenta por meio desses cenários permite a refinação dos parâmetros necessários para aumentar a precisão dos modelos.

3.2 EmuEdge

EmuEdge [9] é um emulador para computação da borda que traz suporte para simulações e testbeds. Ele traz vários níveis de realismo, como sistema operacional, tráfico de rede, topologia e gerenciamento de recursos, enquanto permite uma boa escalabilidade por um baixo custo.

Por ser uma emulador hibrido, funcionando como um *framework* para simular ou emular a computação e a rede, o EmuEdge permite modelar sistemas de computação na

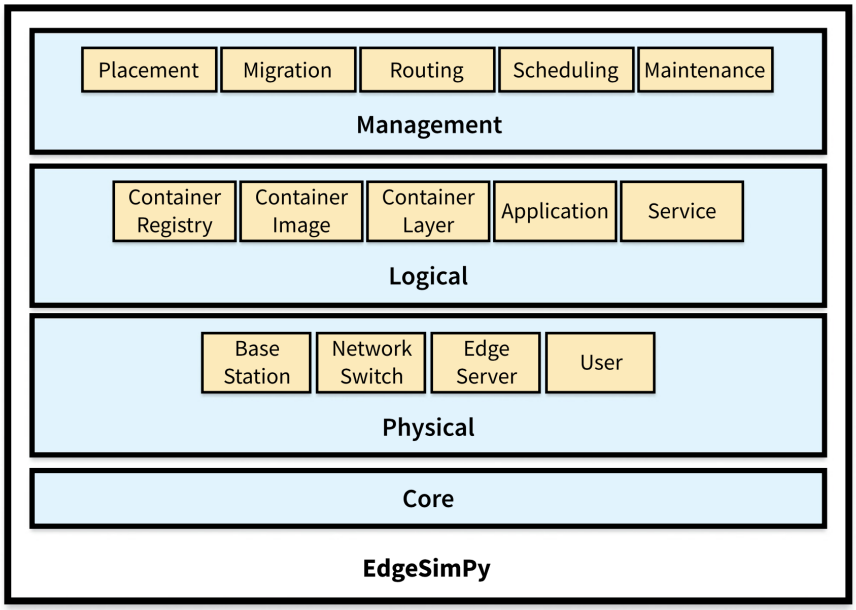

Figura 3.1 – Arquitetura do EdgeSimPy [8]

borda mais próximos da realidade dos *testbeds*. A figura 3.2 mostra a arquitetura que permite as interfaces híbridas.

O EmuEdge permite que a emulação das aplicações, ou seja a computação, utiliza máquinas virtuais, que permitem maior isolamento e heterogeneidade do sistema. Também é possível utilizar *containers* que trazem menos custo devido ao sistema operacional leve, que gera alta escalabilidade.

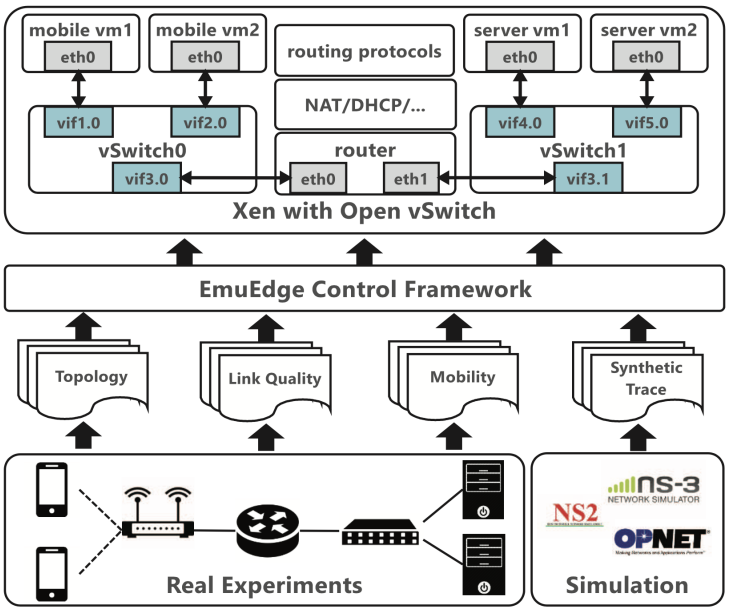

Figura 3.2 – Arquitetura do EmuEdge [9]

4. DESCRIÇÃO DA CONTRIBUIÇÃO

Esse capítulo apresenta a descrição da contribuição, definindo o problema motivador, apresentando os objetivos gerais e específicos e a metodologia utilizada para atingir o cronograma proposto. Após isso, a arquitetura geral da aplicação e as responsabilidades de cada camada e por fim as chamadas de API.

4.1 Motivação

O paradigma de computação na borda traz desafios e problemas novos para a computação. Um deles é a mobilidade dos dispositivos em uma rede distribuída, que demanda diversas estratégias possíveis para solucionar o problema. A mobilidade cria cenários complexos nos quais dispositivos podem entrar e sair do alcance dos nós, gerando problemas para dispositivos que necessitam de baixa latência, alta confiabilidade e sessões contínuas. Nesses cenários, os dispositivos em movimento devem se manter conectados aos nós que possuem a aplicação. Essas situações são especialmente críticas em aplicações como veículos autônomos ou dispositivos médicos móveis, onde interrupções e atrasos podem causar falhas significativas.

A mobilidade dos dispositivos em um ambiente distribuído também impacta diretamente a alocação eficiente de recursos, já que eles trocam frequentemente de nó. Isso não apenas dificulta a manutenção da qualidade do serviço, mas também afeta a utilização geral das aplicações, considerando que os dispositivos podem se mover para um nó onde o limite de latência não é mais aceitável.

Para abordar esses desafios, ferramentas como simuladores, emuladores e testbeds são fundamentais para a validação de estratégias em ambientes de computação na borda, principalmente devido à falta dessas ferramentas em ambientes reais. Essas ferramentas permitem modelar cenários dinâmicos, onde é possível testar diferentes soluções de alocação de recursos, comunicação entre os nós e tolerância a falhas.

A partir do estudo realizado sobre as ferramentas existentes que simulam e emulam a computação na borda, este trabalho tem como motivação criar uma nova ferramenta, o EdgeEmuPy. Essa ferramenta permitirá a emulação dos componentes de computação na borda utilizando Python e Docker, para gerenciar os containers e a rede, sendo acessada por meio de uma API de controle.

4.2 Objetivos

O objetivo principal deste trabalho é desenvolver uma ferramenta de emulação capaz de emular os três componentes de uma computação na borda: a topologia dos nós, os dispositivos conectados e as aplicações. Essa ferramenta será acessada por meio de uma API, que facilitará a utilização, garantirá a integridade dos componentes e permitirá que trabalhos futuros possam contribuir com essa solução.

Para alcançar esse objetivo, a ferramenta utiliza containers Docker, permitindo leveza e escalabilidade para emular ambientes de computação na borda. Todos os componentes devem permitir alterações em suas condições de mobilidade e gerência em tempo real, além de gerar dados e métricas para validar estratégias de emulação.

Além desse escopo, este trabalho possui os seguintes objetivos secundários:

- Permitir a emulação de arquiteturas complexas, com controles de ambientes rápidos e precisos e alta capacidade de emulação de usuários.
- Gerar métricas em tempo de execução, possibilitando a análise dos resultados.
- Gerar a infraestrutura dos nós e dispositivos, resultando em uma imagem atualizada conforme as mudanças realizadas em tempo de execução.

4.3 Metodologia

A metodologia utilizada para a realização deste trabalho consiste no estudo teórico da problemática da computação na borda e das ferramentas existentes. A partir desse estudo, foi possível desenvolver cada componente da ferramenta de maneira isolada utilizando Jupyter Notebooks, facilitando os testes e o desenvolvimento.

Dessa maneira, foi possível garantir que cada componente estivesse funcionando corretamente e implementar, de fato, a ferramenta que os unifica e cria a camada de comunicação entre eles, utilizando Python e a biblioteca Docker. A partir do desenvolvimento de cada camada, foi possível criar a API, que unifica todas as chamadas em um menu.

4.4 Arquitetura

A arquitetura do EdgeEmuPy é definida por três camadas: Topologia, Dispositivos e Aplicações, onde cada uma delas é responsável por emular diferentes componentes da

computação na borda, e pela API que faz as requisições ao núcleo. A Figura 4.1 mostra uma visão geral das camadas e a comunicação entre elas.

A camada de Topologia contém os nós da computação na borda. Cada nó pode ter diversas aplicações diferentes, emulando um servidor. Os nós têm acesso direto às aplicações e as executam quando necessário. Por fim, a camada de Dispositivos emula o comportamento dos aparelhos que utilizam a computação na borda e se comunicam com a camada de Topologia por meio de requisições HTTP.

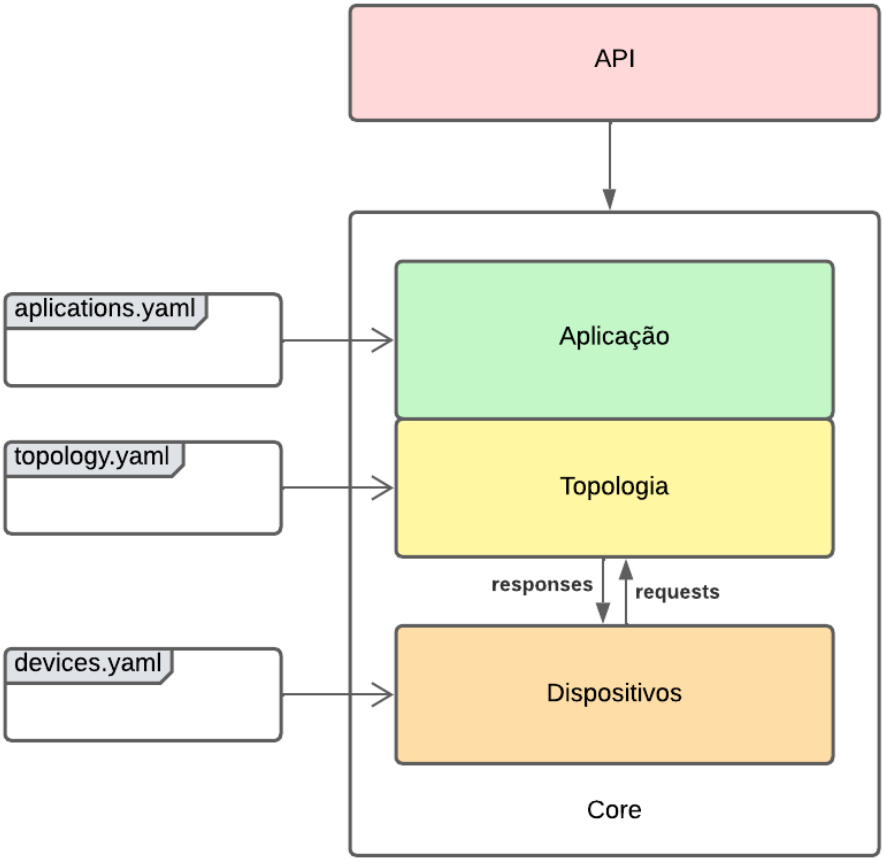

Figura 4.1 – Arquitetura do EdgeEmuPy

As funcionalidades do EdgeEmuPy são acessadas por meio de chamadas de API, garantindo o isolamento dos componentes da arquitetura e o funcionamento esperado da biblioteca. Alguns componentes específicos da emulação, como as estratégias de mobilidade, serão modelados em conjunto com a API de inicialização, permitindo que outros trabalhos possam contribuir facilmente.

A modularização dos componentes permite a separação da arquitetura, o que facilita experimentos e análises, assim como o desenvolvimento a partir da modelagem e testes unitários. Os arquivos de configuração para cada camada também são separados, de maneira que cada camada possa operar de forma independente.

A Figura 4.2 mostra um exemplo de uma arquitetura emulada no EdgeEmuPy. As três camadas estão interconectadas por meio de diferentes redes, que atuam de maneira a isolar seus componentes. Os containers dos nós recebem as requisições dos usuários, buscam aplicações no Registry e se comunicam entre si, para emular o comportamento distribuído.

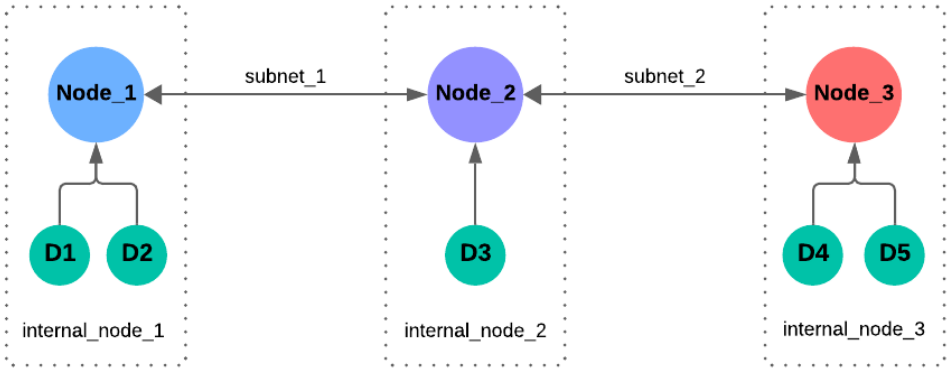

Figura 4.2 – Exemplo de arquitetura emulada no EdgeEmuPy

4.5 Topologia

A camada de Topologia é a principal camada no núcleo da arquitetura do EdgeEmuPy. Esta camada é responsável por gerenciar os containers que emulam os nós da computação na borda, bem como as ligações entre eles.

A topologia é definida em um arquivo de configuração que especifica os parâmetros necessários para a criação dos containers e das redes, como mostra a Figura 4.3. Os detalhes das redes e dos nós são abordados nas próximas subseções.

A partir do arquivo de configuração da topologia, o EdgeEmuPy é capaz de inicializar e verificar se há algum erro de configuração. A biblioteca Docker do Python não realiza nenhum tratamento de erro nem limpa containers finalizados, portanto, todos os casos são tratados em código.

Dessa maneira, independentemente de onde o erro ocorra, o ambiente Docker é sempre limpo com segurança. A ordem de limpeza é parar e deletar todos os containers, depois as redes e, por fim, os volumes, garantindo que nenhum problema de dependência aconteça.

Os nós são emulados a partir de containers Docker, permitindo virtualização e escalabilidade com um pequeno custo computacional em comparação a máquinas virtuais. Os nós estão interconectados por meio de redes Docker, que representam as conexões de rede entre os nós e permitem a comunicação.

topology.yaml > ...

```yaml
networks:
  - name: link_1
    subnet: 10.0.0.0/8
    latency: 10
    bandwidth: 1.5
  # ...
nodes:
  - name: node_1
    lat: 2
    lon: 5
    connect_to:
      - link_1
      - link_2
  # ...
```

Figura 4.3 – Arquivo de configuração da topologia

A Figura 4.4 exemplifica uma topologia simples formada por três nós, identificados como Node_1, Node_2 e Node_3, e duas ligações, denominadas Subnet_1 e Subnet_2. Essa configuração demonstra o funcionamento do canal de comunicação. Cada nó possui um IP específico determinado pela sub-rede à qual pertence, mapeando interfaces de rede diferentes para cada endereço.

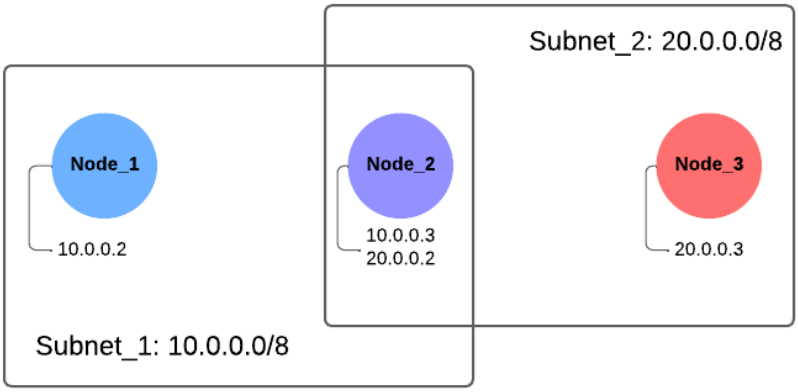

Figura 4.4 – Exemplo de uma configuração de topologia

4.5.1 Nós

Os containers da camada de topologia emulam nós na computação de borda. No arquivo de definição da topologia, cada nó recebe um nome identificador. Esse nome é único, e um prefixo é adicionado a ele para garantir que não existam conflitos com possíveis containers já criados. Cada nó possui os atributos de latitude e longitude, além das redes às quais está conectado, conforme descrito na Figura 4.3.

Após ler e verificar as configurações, o EdgeEmuPy realiza o build da imagem Docker utilizada pelos containers. Essa imagem foi construída a partir do Ubuntu 16.04 LTS e possui ferramentas Linux de rede, teste e monitoramento, além de configuração de logs e daemons necessários.

A imagem Docker dos nós, assim como a dos dispositivos, é otimizada visando leveza e simplicidade, permitindo que a emulação suporte redes complexas e testes de carga.

Uma característica da camada de topologia é o isolamento dos nós utilizando redes Docker, exemplificado na Figura 4.4. Dessa maneira, os containers que não estão conectados entre si não conseguem se comunicar. No entanto, em casos como o exemplo mostrado, onde o Node_2 está conectado às sub-redes dos outros containers, o Node_1 não consegue se comunicar com o Node_3 (e vice-versa), pois não conseguem resolver os endereços IP das máquinas em outras sub-redes, uma vez que não as conhecem.

Portanto, para que os containers resolvam IPs de outras sub-redes, cada container é responsável por inicializar o protocolo OSPF (Open Shortest Path First), que habilita a comunicação entre nós em redes diferentes de maneira dinâmica. Isso permite que os próprios containers se comuniquem e criem as tabelas de roteamento, eliminando a necessidade de definir cada regra manualmente no código.

A configuração do OSPF é feita utilizando o software Quagga, uma ferramenta Linux que simplifica a configuração. Todos os nós compartilham arquivos de configuração do Quagga, chamados Zebra (core daemon que abstrai a camada de kernel do Unix) e Vtysh (uma API que facilita a configuração da rede). Além disso, cada nó cria um arquivo de configuração do OSPF único, que possui arquivos de log e é atribuído um router-id e a área onde o container está localizado.

Cada container também cria um volume que conecta o arquivo de configuração do OSPF gerado na máquina host. Assim, qualquer modificação pode ser realizada fora dos containers. Os volumes possuem permissão de leitura e escrita.

Os nós também possuem atributos de latitude e longitude, armazenados como variáveis de ambiente. Esses atributos servem para localizar o nó e, dessa forma, quando

um novo dispositivo é inicializado, ele pode ser atribuído ao nó mais próximo, otimizando a latência.

Por fim, os containers são inicializados com as permissões NET_ADMIN e NET_RAW, além da flag de privilégio do Docker, necessárias para que as configurações de rede funcionem corretamente. Eles se conectam à rede default do Docker, para permitir a conexão com o container Registry. Para manter o isolamento, são configuradas regras de cadeia utilizando iptables, a fim de impedir o encaminhamento de IPs.

Após a inicialização, cada nó é conectado às redes necessárias. Primeiro, é feita uma varredura pelo parâmetro connect_to (ver Figura 4.3), e ele é conectado às redes da topologia. Depois, é criada e conectada uma rede interna para cada nó, possibilitando a conexão dos dispositivos.

Um dos desafios encontrados durante o desenvolvimento do trabalho foi o rastreamento do IP dos containers. Tendo em vista que os containers dispositivos podem se locomover, trocando de rede e, portanto, trocando de IP, foi necessário encontrar uma solução que garantisse que todos conseguissem se identificar. Para isso, foi utilizado o servidor DNS do Docker, que resolve os IPs dos containers com base em um arquivo hosts.

src > shared_hosts > hosts

```
1   127.0.0.1     localhost
2   ::1           localhost ip6-localhost ip6-loopback
3   fe00::0       ip6-localnet
4   ff00::0       ip6-mcastprefix
5   ff02::1       ip6-allnodes
6   ff02::2       ip6-allrouters
7   c1  10.0.0.2
8   c2  10.0.0.3
9   u1  192.168.160.2
```

Figura 4.5 – Arquivo hosts para configuração do DNS

Uma vez que os nós são conectados às redes e atribuídos os endereços IP, o EdgeEmuPy varre todas as redes e seleciona o primeiro IP referente a um container. Assim, ele cria um dicionário que mapeia o IP para o container e gera um arquivo na máquina hospedeira chamado hosts, com as informações do dicionário, como mostra a Figura 4.5.

O arquivo hosts da máquina hospedeira é vinculado ao arquivo /etc/hosts de todos os containers, sejam eles nós ou dispositivos. Dessa maneira, qualquer atualização feita no arquivo base será propagada para todos os containers, permitindo que eles resolvam qualquer IP.

As configurações de rede, como latência e banda larga, são definidas utilizando a ferramenta Linux traffic control, capaz de modificar esses atributos individualmente para cada container.

4.5.2 Rede

O EdgeEmuPy emula as conexões entre os nós utilizando redes Docker. Cada rede definida no arquivo de topologia, Figura 4.3, possui um nome identificador único que, semelhante aos nós, é concatenado a um prefixo. As redes também possuem informações de sub-rede e máscara, definidas no arquivo de topologia, além de parâmetros como latência e banda larga.

As redes são criadas utilizando o driver user-defined bridge. Esse driver permite que containers na mesma rede possam se comunicar e utilizem DNS para resolver os nomes de outros containers, enquanto os isola de todas as outras redes. Ele também permite que containers possam conectar e desconectar de uma rede, facilitando a emulação de quedas ou a adição de novos nós.

4.5.3 Rede interna

Cada nó também implementa uma rede utilizando o user-defined bridge driver, que habilita a comunicação direta com os containers dispositivos. Essa rede interna é exclusiva para um único nó; no entanto, permite a conexão de múltiplos dispositivos, oferecendo flexibilidade, escalabilidade e isolamento.

Sempre que um dispositivo deseja fazer uma requisição a uma aplicação, ele se comunica com o container do nó, que sempre está associado ao segundo IP disponível daquela sub-rede, garantindo consistência. Por utilizarmos redes Docker para a comunicação, é possível utilizar DNS para resolver os nomes dos containers, facilitando a comunicação.

4.6 Aplicação

A camada de Aplicação representa os programas que são executados nos nós e acessados pelos dispositivos. A Figura 4.6 mostra um exemplo do arquivo de configuração das aplicações, que inicializa cada aplicação em um container nó.

Durante a inicialização da camada de dispositivos, todos os containers são inicializados e conectados a um arquivo da máquina host chamado applications. Esse arquivo funciona como um dicionário, mantendo o registro das aplicações da seguinte maneira: nome da aplicação, porta e nó servidor.

src > applications.yaml > ...

```yaml
applications:
  - node: c1
    apps:
      a1: 3306
      a2: 5432

  - node: c2
    apps:
      a4: 8001
```

Figura 4.6 – Arquivo de configuração das Aplicações

Dessa maneira, os dispositivos conseguem rapidamente descobrir para qual container enviar a mensagem. Caso a aplicação troque de container, esse arquivo é editado e atualizado em todos os dispositivos.

Para simular as aplicações dentro dos nós, o EdgeEmuPy utiliza o comando nc (netcat) do Linux em um laço contínuo, que captura qualquer pacote enviado para aquele nó, naquela porta, e responde com uma mensagem OK.

4.7 Dispositivos

A camada de Dispositivos emula os aparelhos da computação na borda. Esses dispositivos representam aparelhos que interagem com os nós em uma arquitetura cliente-servidor, emulando as requisições de aplicações.

Na Figura 4.7 está um exemplo de arquivo de configuração padrão para um dispositivo. Eles possuem um nome identificador único com prefixo, latitude e longitude representadas nos containers como variáveis de ambiente, uma aplicação de destino, um behaviour e o threshold.

Tanto a aplicação de destino quanto o behaviour são configurações opcionais, que simulam o comportamento de um usuário ativo ou passivo na rede. Os behaviours podem ser configurados manualmente, seguindo a ordem de pacotes enviados e tempo de espera, ou configurados a partir de rotinas pré-definidas, categorizadas de A até D. As rotinas são definidas da seguinte maneira:

- Behaviour A: Envia 1 pacote por ping para a aplicação a cada 30 segundos.
- Behaviour B: Envia 1 pacote por ping para a aplicação a cada 10 segundos.
- Behaviour C: Envia 10 pacotes por ping para a aplicação a cada 30 segundos.

src > devices.yaml > ...

```yaml
devices:
  - name: u1
    lat: 1
    lon: 1
    application: a1
    behavior: A
    threshold: 50

  - name: u2
    lat: 2
    lon: 2
    application: a2
    threshold: 100

  - name: u3
    lat: 3
    lon: 3
    application: a1
    behavior: [1, 1]
  # ...
```

Figura 4.7 – Arquivo de configuração dos Dispositivos

- Behaviour D: Envia 1 pacote por ping para a aplicação a cada X segundos, definido aleatoriamente.

O último parâmetro dos dispositivos é o threshold, métrica utilizada para definir um limite das trocas de mensagens. Ele serve para emular o comportamento de dispositivos que necessitam de baixa latência, como carros autônomos.

Os dispositivos são representados por containers construídos a partir de uma imagem Alpine, simples e leve, para permitir que vários dispositivos sejam emulados em uma mesma máquina, otimizando o uso de recursos. Cada container utiliza ferramentas Linux básicas de rede, como iputils, telnet, tcpdump, entre outras, para permitir a comunicação com os nós.

4.7.1 Mobilidade

Uma característica dos dispositivos é a capacidade de se "movimentar", alterando suas coordenadas de latitude e longitude. Esse comportamento imita dispositivos IoT móveis, como carros, drones, smartphones e outros.

A mobilidade dos dispositivos gera cenários dinâmicos, nos quais é possível emular diferentes estratégias de alocação de recursos, avaliadas com base em métricas como latência e banda larga. Dessa forma, o algoritmo pode determinar qual solução deve ser aplicada em cada caso, como mover o dispositivo de um nó para outro, trazer a aplicação para um nó mais próximo ou utilizar outra rota.

Portanto, após serem criados, é executada uma rotina que busca o nó mais próximo daquele container e conecta o dispositivo a ele, utilizando uma função de distância euclidiana. Esse comportamento foi implementado para que os dispositivos possam ser desenvolvidos e testados com maior facilidade, por ser um algoritmo simples.

Contudo, emular diferentes estratégias de mobilidade está fora do escopo deste trabalho. Por isso, a API possui chamadas específicas para a camada de dispositivos, abordadas na última seção deste capítulo.

Os dispositivos são conectados à rede interna do nó mais próximo e recebem o primeiro IP disponível daquela rede. Para resolver os problemas de DNS discutidos na seção de Nós, é criado um dicionário que armazena os IPs de todos os dispositivos e, por fim, os acrescenta ao arquivo hosts, como mostrado na Figura 4.5.

Diferente dos nós, que sempre terão o mesmo IP, os dispositivos podem trocar de uma rede interna para outra, resultando em diferentes IPs. Por isso, sempre que um dispositivo troca de rede, ele ativa uma função que atualiza o dicionário e o arquivo hosts. Como todos os arquivos hosts dos containers estão vinculados ao arquivo da máquina hospedeira, todos são atualizados com o novo IP.

4.7.2 Comunicação com as aplicações

Uma vez que o usuário estiver conectado à rede e com todas as configurações de mobilidade resolvidas, ele inicializa uma rotina de troca de mensagens com a aplicação.

Para a troca de mensagens, os dispositivos utilizam o comando nc do Linux e enviam uma mensagem qualquer até o container de destino, com o nome resolvido pelo DNS. Também é utilizado o comando time, do Linux, para medir o tempo que a mensagem levou para ir até a aplicação e retornar.

Conforme citado anteriormente, os dispositivos possuem um parâmetro chamado threshold, que define um limite para o tempo de comunicação com a aplicação de destino. Quando esse limite é excedido, o dispositivo dispara uma função que busca outra rota até o nó de destino. Essa função testa todas as combinações possíveis de rotas e define a nova rota como a mais rápida.

As estratégias avançadas de busca da melhor rota e de realocação de aplicações ficaram fora do escopo deste projeto. Contudo, para validar o EdgeEmuPy, foi implementada essa função simples.

4.8 API

As funcionalidades do *EdgeEmuPy* foram desenvolvidas buscando a integração com uma API que facilita sua utilização e permite que outras integrações possam contribuir com esse trabalho. Portanto, a camada de API disponibiliza essa interface a partir de métodos de inicialização e métodos específicos para camada camada.

A camada de Topologia possui chamadas de API que controlam os nós e as redes:

- initialize_topology(): Inicializa a topologia a partir do arquivo de configuração e interrompe a emulação caso encontre algum erro.
- create_node(): Cria um novo nodo. Ele não possui latitude nem longitude e não está conectado a nenhuma rede
- get_node_location(): Retorna a latitude e longitude de um nodo
- set_node_location(): Define a latitude e longitude de um nodo
- create_network(): Cria uma rede a partir de um nome e uma máscara
- connect_nodo_to_network(): Adiciona um nodo a uma rede.
- disconnect_nodo_from_network(): Remove um nodo de uma rede
- topology_status(): Retorna o status dos nodos e das redes

A camada de Dispositivos expõe as seguintes funções para os dispositivos:

- initialize_devices(): Inicializa os dispositivos a partir do arquivo de configuração e interrompe a emulação caso encontre algum erro.
- create_device(): Cria um novo dispositivo sem latitude nem longitude
- get_user_location(): Retorna a latitude e longitude de um dispositivo
- set_user_location(): Define a latitude e longitude de um dispositivo
- remove_device(): Remove um dispositivo
- devices_status(): Retorna o status dos dispositivos

A camada de Aplicações possui as seguintes chamadas:

- initialize_applications(): Inicializa as aplicações a partir do arquivo de configuração e interrompe a emulação caso encontre algum erro.
- create_application(): Cria uma aplicação e verifica se aquela porta já está em uso
- remove_application(): Remove uma aplicação
- applications_status(): Retorna o status das aplicações

Além dessas funcionalidades, a API disponibiliza métodos para controle geral, como:

- shutdown_emulation(): Termina a emulação e limpa todos os containers, redes e volumes criados
- restart_emulation(): Termina a emulação e inicializa as três camadas a partir dos arquivos de configuração.
- set_verbose(): Retorna todos os logs do nível de DEBUG.

5. AVALIAÇÃO

Este capítulo aborda o estudo de cenários práticos envolvendo topologias de rede, de migração dos dispositivos e testes de carga.

5.1 Caso base

Para a validação da migração dos dispositivos, foi utilizada a arquitetura da Figura 5.1, onde 4 nós estão interconectados por redes e o dispositivo e a aplicação se encontra no nó N1. A rede é composta por latências específicas, emulando um cenário mais real.

O Dispositivo D1 foi inicializado nas coordenadas [1,1], e conectado ao nodo N1 por meio da rede interna do mesmo. A aplicação também foi inicializada em N1 e é exposta pela porta 8001.

A banda larga de todas as redes foi definida conforme o padrão, em 1mbit e a perca de pacotes em 0%.

O dispositivo está configurado para ter um comportamento manual, onde envia 1 pacote a cada 10 segundos para a aplicação e o *threshold* do dispositivo foi configurado para 90ms.

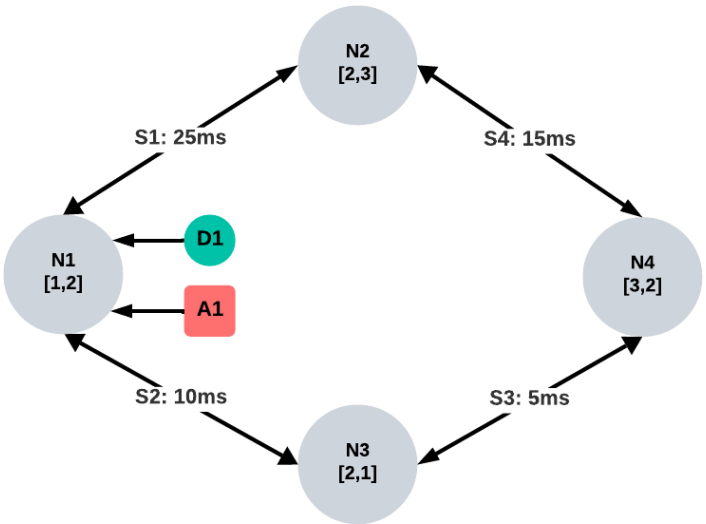

Figura 5.1 – Arquitetura do caso base

5.2 Migração de dispositivo

A partir do caso base, foi realizado uma sequencia de passos, demonstrados na Figura 5.2, que busca emular o comportamento de um dispositivo se movendo e alterações na rede.

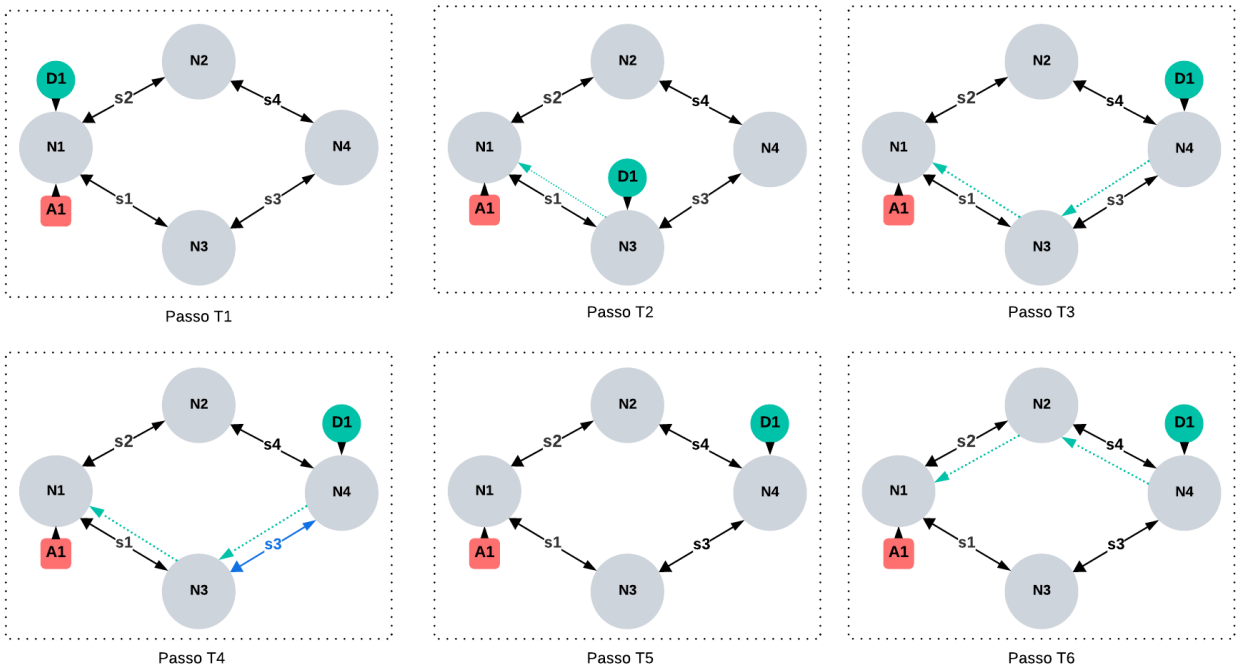

Figura 5.2 – Passos da validação da migração de um dispositivo

O primeiro passo, T1, é a configuração inicial da rede, onde o dispositivo e a aplicação se encontram no nó N1 e a comunicação entre eles leva menos que 1ms/pacote.

No passo T2, o dispositivo se move até a posição [2,1], e automaticamente é conectado ao nó N2. Contudo, a aplicação continua em N1, portanto o dispositivo precisa utilizar a rede s1 para se comunicar, o que aumenta o tempo da troca de mensagens para aproximadamente 21ms/pacote.

O passo T3 vemos o mesmo comportamento, onde o dispositivo se move até [3,1], é conectado a N3 e a troca de mensagens passa a levar aproximadamente 31ms/pacote.

Durante o passo T4, a rede s3 sofre uma alteração que aumenta a latência para 50ms. Essa mudança simula o caso de alta demanda daquela rede, gerando impacto nos dispositivos. Dessa maneira, o próximo pacote de D1 leva aproximadamente 121ms e ultrapassa o *threshold* definido naquele dispositivo.

O passo T5 demonstra o período que D1 está sem conexão até a aplicação e buscando uma nova rota e então encontra um caminho pro N2, cujo tempo está dentro do

limite estabelecido. Durante o processo de queda e busca de nova rota, o dispositivo deixou de enviar 1 pacote, respeitando o comportamento definido anteriormente.

Por fim, o passo T6 mostra a nova rota que D1 utiliza para troca de mensagens com A1, que leva aproximadamente 81ms/pacote.

A validação foi executada 10 vezes, e o tempo das trocas de mensagens são a média de todas as execuções. Todas as alterações na rede, sendo elas a troca de coordenadas do dispositivos e a mudança da latência da rede, foram realizadas manualmente, a partir das funções da API.

5.3 Teste de carga

O teste de carga foi realizado para avaliar o desempenho dos dispositivos e nodos sob alta demanda. O objetivo foi observar como os recursos do sistema se comportam diante a diferentes quantidades de containers e outras métricas específicas.

Os testes foram executados em um MacBook Pro 2019, em um processador Intel i-9 de 8 cores e 16 GB de memória. Contudo, a Docker Engine utiliza apenas 2 cores de processamento e 8 GB de memória.

5.3.1 Teste de dispositivos

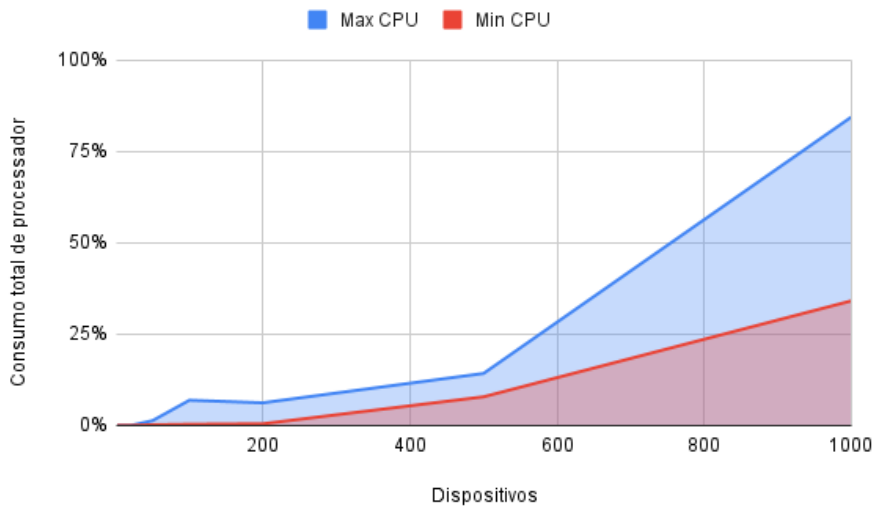

Figura 5.3 – Utilização de CPU

Para os testes de dispositivos, foi utilizada uma arquitetura simples, com dois nodos e uma rede ligando eles. Então foi inicializados, a partir do arquivo de configuração, 10, 20, 50, 100, 200, 500 e 900 dispositivos.

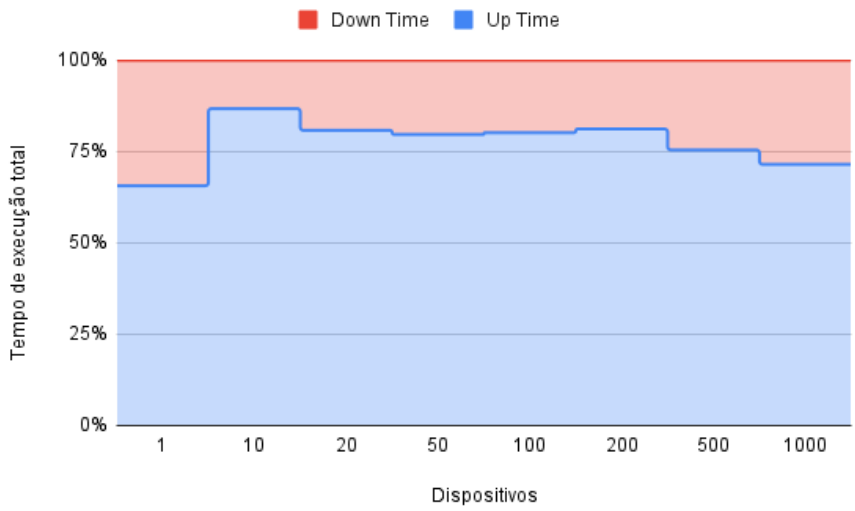

Figura 5.4 – Tempo de inicialização e remoção dos Dispositivos

A Figura 5.3 exibe a variação no uso de CPU durante o teste. A linha vermelha mostra a menor utilização de CPU, que acontece durante o processo de inicialização dos dispositivos. A linha azul mostra o maior consumo de CPU dos testes, que tende a acontecer depois que todos estão executando seus comportamentos por aproximadamente 30 segundos. Durante o teste, a utilização de memória dos dispositivos se manteve baixa e constante.

Já a Figura 5.4 mostra o tempo de inicialização dos dispositivos relativo ao tempo de remoção. Essa métrica é importante para entender o quanto de *overhead* temos na inicialização dos dispositivos, tendo em vista que a remoção deles é linear.

5.3.2 Teste de nodos

Além dos testes de carga, que focam apenas nos dispositivos, também foram realizados testes de desempenhos específicos para os nós. A arquitetura utilizada para os testes de topologia se limitou a quantidade máxima de nós suportada, que é de 29 nós. Para levar a topologia ao máximo estresse, cada nó é ligado por redes com todos os outros nós, gerando uma arquitetura densa.

Ambos os recursos de CPU e memória tem comportamento linear, que é explicado pela baixa quantidade de nós suportados. Contudo, as imagens Docker utilizadas para os nós consomem muito mais recursos que os dispositivos, chegando em 50% da capacidade máxima da engine, mostrado nas Figuras 5.5 e 5.6.

Apesar do comportamento linear dos recursos da máquina, o tempo de execução do protocolo OSPF é um dos indicadores mais importantes para a medida de desempenho da arquitetura, que aumenta exponencialmente conforme o número de nodos cresce.

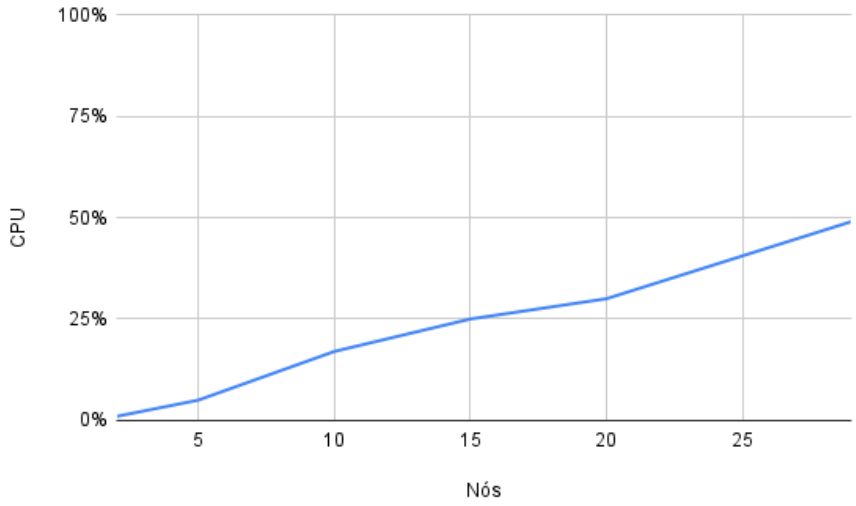

Figura 5.5 – Utilização de CPU

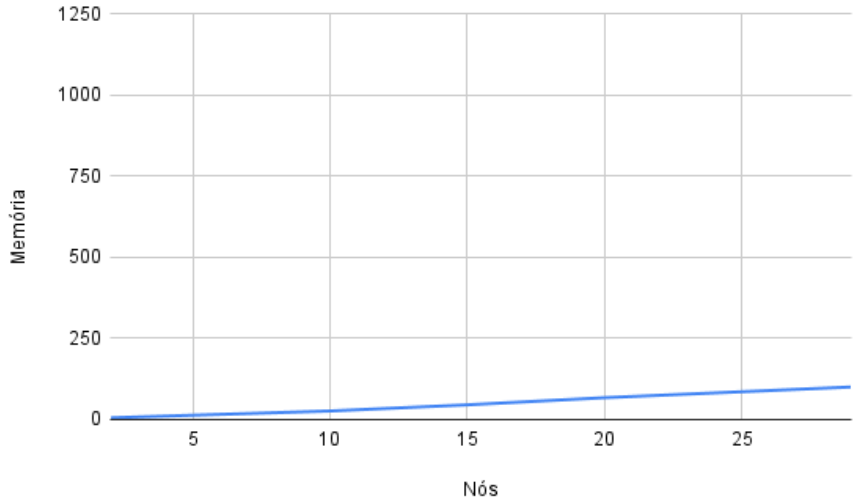

Figura 5.6 – Utilização de Memória

Os resultados dos testes de desempenho mostram que a arquitetura é balanceada, e demonstra o comportamento esperado. Contudo, diversas otimizações podem ser feitas para diminuir o consumo de CPU das imagens.

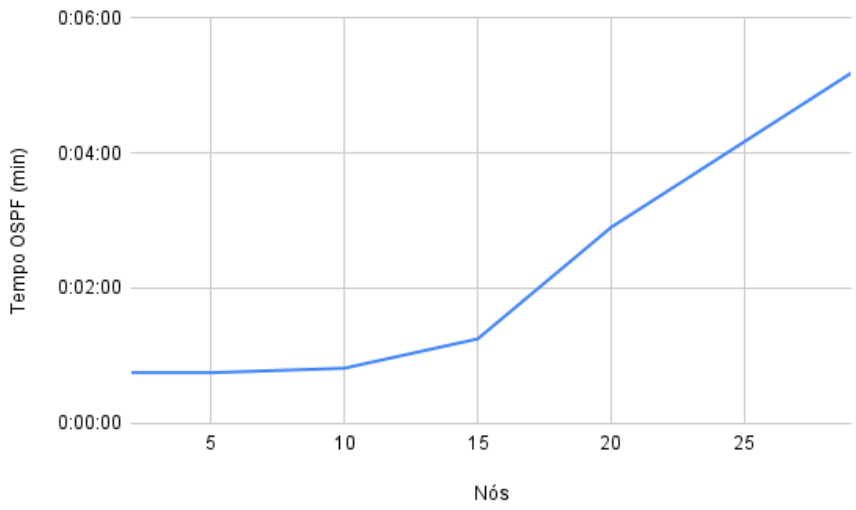

Figura 5.7 – Tempo de Configuração do OSPF em Relação ao Número de Nós

6. CONCLUSÃO E TRABALHOS FUTUROS

Com o avanço da Internet das Coisas (IoT) e o ganho de mercado de plataformas na nuvem, a computação na borda surge como um novo paradigma para solucionar os problemas dessas frentes. Porém a escassez de infraestrutura capaz de suportar essa nova tecnologia dificulta o desenvolvimento de novas estratégias para viabilizar a mesma.

Dessa maneira, a criação de ferramentas de emulação é fundamental para o desenvolvimento de novos paradigmas, com o intuito de validar novas soluções, com uma visão mais próxima da realidade da arquitetura, permitindo avaliar medidas em tempo de execução.

Motivado pela necessidade de soluções robustas, para desenvolver e testar os desafios da computação na borda, esse trabalho busca compreender as ferramentas existentes e o material teórico para desenvolver uma nova ferramenta para emular os componentes utilizando containers Docker. Essa característica permite a flexibilidade, dinamismo e escalabilidade necessários para emulação de ambientes complexos.

O desenvolvimento do EdgeEmuPy permitiu uma maior compreensão dos desafios impostos pela computação na borda. A solução proposta para as camadas de Topologia e Dispositivos foram implementadas assim como a conexão entre elas, e a criação da camada de API oferece uma interface simples e iterativa a ferramenta. Por fim, a camada de aplicação, apesar de ser a mais simples, cumpre o papel projetado incialmente.

Apesar disso, diversas contribuições podem ser realizadas no futuro, como a otimização dos protocolos utilizados pelas camadas, a emulação de aplicações, a expansão da capacidade da arquitetura para emulação e a emulação de diferentes estratégias para redirecionamento de dispositivos e aplicações. O EdgeEmuPy é uma ferramenta que foi desenvolvida pensando em ser o bloco inicial para o estudo de emulações em um ambiente de computação em borda.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Cao, K.; Liu, Y.; Meng, G.; Sun, Q. “An overview on edge computing research”, *IEEE Access*, vol. 8, 2020, pp. 85714–85728.

[2] Khan, W. Z.; Ahmed, E.; Hakak, S.; Yaqoob, I.; Ahmed, A. “Edge computing: A survey”, *Future Generation Computer Systems*, vol. 97, 2019, pp. 219–235.

[3] Mao, Y.; You, C.; Zhang, J.; Huang, K.; Letaief, K. B. “A survey on mobile edge computing: The communication perspective”, *IEEE Communications Surveys Tutorials*, vol. 19–4, 2017, pp. 2322–2358.

[4] McGregor, I. “The relationship between simulation and emulation”. In: Proceedings of the Winter Simulation Conference, 2002, pp. 1683–1688 vol.2.

[5] Satyanarayanan, M. “The emergence of edge computing”, *Computer*, vol. 50–1, 2017, pp. 30–39.

[6] Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. “Edge computing: Vision and challenges”, *IEEE Internet of Things Journal*, vol. 3–5, 2016, pp. 637–646.

[7] Siaterlis, C.; Garcia, A. P.; Genge, B. “On the use of emulab testbeds for scientifically rigorous experiments”, *IEEE Communications Surveys Tutorials*, vol. 15–2, 2013, pp. 929–942.

[8] Souza, P. S.; Ferreto, T.; Calheiros, R. N. “Edgesimpy: Python-based modeling and simulation of edge computing resource management policies”, *Future Generation Computer Systems*, vol. 148, 2023, pp. 446–459.

[9] Zeng, Y.; Chao, M.; Stoleru, R. “Emuedge: A hybrid emulator for reproducible and realistic edge computing experiments”. In: 2019 IEEE International Conference on Fog Computing (ICFC), 2019, pp. 153–164.