Universidade do Estado do Rio de Janeiro

Centro de Tecnologia e Ciências

Faculdade de Engenharia

Matheus Costa Stutzel

Um *framework* para orquestração de recursos com enfoque na escalabilidade para aplicações na Internet das Coisas

Rio de Janeiro

2020

Matheus Costa Stutzel

Um *framework* para orquestração de recursos com enfoque na escalabilidade para aplicações na Internet das Coisas

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre em Ciências, ao Programa de Pós-Gradução em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Redes de Telecomunicações.

Orientador: Prof. D.Sc. Alexandre Sztajnberg

Rio de Janeiro

2020

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

S937 Stutzel, Matheus Costa.
Um framework para orquestração de recursos com enfoque na escalabilidade para aplicações na internet das coisas / Matheus Costa Stutzel. – 2020.
129f.

Orientador: Alexandre Sztajnberg.
Dissertação (Mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

1. Engenharia eletrônica - Teses. 2. Internet das coisas - Teses. 3. Algoritmos - Teses. 4. Otimização - Teses. 5. Aplicações Web - Teses. I. Sztajnberg, Alexandre. II. Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia. III. Título.

CDU 004.77

Bibliotecária: Júlia Vieira – CRB7/6022

Assinatura

Data

Matheus Costa Stutzel

Um *framework* para orquestração de recursos com enfoque na escalabilidade para aplicações na Internet das Coisas

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre em Ciências, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Redes de Telecomunicações.

Aprovado em: 12 de Novembro de 2020

Banca Examinadora:

Prof. D.Sc. Alexandre Sztajnberg (Orientador)
PEL/UERJ

Prof. D.Sc. Francisco Figueiredo Goytacaz Sant'Anna
PEL/UERJ

Prof. D.Sc. José Viterbo Filho
DCC/UFF

Prof. D.Sc. Vinicius Tavares Petrucci
DCC/University of Pittsburgh

Rio de Janeiro
2020

AGRADECIMENTO

Agradeço a todos que fizeram parte dessa caminhada e que me incentivaram. Em especial agradeço ao meu orientador, prof. D.Sc. Alexandre Sztajnberg, por todos esses anos de conselhos, ensinamentos e oportunidades, desde o convite para a iniciação científica, passando por todos os projetos do LCC, até chegar nessa dissertação. Não poderia deixar de agradecer aos meus pais por sempre me apoiarem e por todos os sacrifícios que fizeram para que eu tivesse uma boa educação. Também agradeço a Nathália, minha esposa, por me ajudar em toda essa jornada UERJ.

A tarefa não é tanto ver aquilo que ninguém viu,
mas pensar o que ninguém ainda pensou sobre aquilo que todo mundo vê.

Arthur Schopenhauer

RESUMO

Stutzel, M.C. *Um framework para orquestração de recursos com enfoque na escalabilidade para aplicações na Internet das Coisas.* 129 f. Dissertação (Mestrado em Engenharia Eletrônica) - Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, 2020.

Considerando o número crescente de dispositivos conectados à Internet e os mecanismos e tecnologias desenvolvidas no contexto da Internet das Coisas, a presente dissertação propõe um *framework* e apresenta uma implementação de referência capaz de orquestrar o uso de recursos e promover escalabilidade para aplicações de IoT. Dado um conjunto de recursos de *fog*, *edge* e nuvem disponíveis, potencialmente distribuídos, o *framework* orquestra a comunicação entre clientes, agentes, e estes serviços. Esta orquestração considera o uso de CPU, memória e latência de rede, para selecionar os recursos mais adequados e reorganizar a topologia de interação entre os elementos, com o objetivo de minimizar estas métricas, na média. Com isso, espera-se que aplicações IoT com um número grande de elementos possam realizar suas atividades dentro de um tempo aceitável. O *framework* proposto é transparente para as aplicações de IoT, que continuam utilizando serviços de um *middleware*, e não precisam conhecer o *framework*, nem os efeitos de sua operação. A implementação de referência utiliza como base microsserviços Docker e o *middleware* SiteWhere, amplamente empregados no desenvolvimento de aplicações distribuídas. No processo de implementação, identificou-se uma limitação na versão utilizada do Docker, relacionada às redes virtuais, chamadas de *swarm*. Esta limitação impedia a comunicação transparente entre elementos que estivessem em redes diferentes, o que no caso de IoT é muito comum, dada a natureza distribuída das aplicações e o uso de diferentes nuvens. Foi necessário, assim, propor e implementar uma nova rede *overlay* que permitisse a comunicação transparente entre os elementos de redes diferentes, mesmo aqueles atrás de um servidor NAT. O *framework* realiza ciclos onde as métricas consideradas são monitoradas em todo conjunto de recursos disponível e em seguida aciona um algoritmo de alocação de recursos. Este algoritmo avalia o estado dos elementos da aplicação e as métricas monitoradas e, então, propõe uma nova topologia de interação entre os elementos para melhorar o desempenho da aplicação. Cinco algoritmos de alocação de recursos selecionados na literatura foram integrados ao *framework*. A abordagem de cada um destes algoritmos tende a privilegiar a otimização de métricas diferentes na alocação

de recursos, o que poderia resultar em maior escalabilidade ou melhor uso de recursos, o que pode afetar de forma diferente as aplicações. A implementação de referência foi avaliada em dois grupos de testes. O primeiro teste comparou o desempenho dos algoritmos de alocação de recursos selecionados. Com isso foi possível verificar se o *framework* seria modular para aceitar diferentes algoritmos, e se a implementação de referência se comportaria adequadamente para qualquer algoritmo. Utilizamos a métrica de latência de rede para selecionar o algoritmo que mais favoreceria a escalabilidade, otimizando o tempo para a troca de mensagens entre os elementos da aplicação e os serviços de *middleware*. O algoritmo ERA apresentou os melhores resultados. O segundo teste avaliou a escalabilidade do *framework* com um cenário de até 10.000 clientes simultâneos utilizando o ERA como algoritmo de alocação de recursos. O *framework* ofereceu suporte para os 10.000 clientes, partindo de uma topologia de alocação inicial, até uma topologia que minimizava a latência média inicial em 33%.

Palavras-chave: IoT; Microsserviços; Alocação de microsserviços; *Framework*.

ABSTRACT

Stutzel, M.C. *A framework for orchestrating resources with a focus on scalability for applications on the Internet of Things.* 129 f. Dissertação (Mestrado em Engenharia Eletrônica) - Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, 2020.

Considering the growing number of devices connected to the Internet and the mechanisms and technologies developed in the context of the Internet of Things, this dissertation proposes a framework and presents a reference implementation capable of orchestrating the use of resources and promote scalability for IoT applications. Given a set of available, potentially distributed, services running on fog, edge and cloud resources, the framework orchestrates communication between clients, agents, and these services. This orchestration considers the use of CPU, memory and network latency, to select the most appropriate resources and reorganize the interaction topology between the elements, with the objective of minimizing these metrics, in average. With that, it is expected that IoT applications with a large number of elements can carry out their activities within an acceptable time. The proposed framework is transparent to IoT applications, which continue using middleware services. The applications do not need to know the framework, nor the effect of its operation. The reference implementation uses Docker microservices as the basis and SiteWhere middleware, widely used in distributed application development. In the implementation process, a limitation was identified in the used version of Docker, related to virtual networks, called swarm. This limitation prevented transparent communication between elements on different networks, which in the case of IoT is very common, given the distributed nature of the applications and the use of different clouds. It was therefore necessary to propose and implement a new overlay network that would allow transparent communication between the elements of different networks, even those behind a NAT server. The framework performs cycles where the metrics considered are monitored across the entire set of available resources and then triggers a resource allocation algorithm. This algorithm assesses the state of the elements of the application and the metrics monitored and then proposes a new topology of interaction between the elements to improve the performance of the application. Five data allocation algorithms resources selected in the literature have been integrated into the framework. The approach of each one of these algorithms tends to favor the optimization of different metrics in the

allocation resources, which could result in greater scalability or better use of resources, which can affect applications differently. The reference implementation was evaluated in two groups of tests. The first test compared the performance of the selected resource allocation algorithms. With that it was possible to verify if the framework would be modular to accept different algorithms, and if the reference implementation would behave properly for any algorithm. We use latency metrics to select the algorithm that would most favor scalability, optimizing the time for the exchange of messages between the application elements and the middleware services. The ERA algorithm showed the best results. The second test evaluated the scalability of the framework with a scenario of up to 10,000 simultaneous clients using ERA as a resource allocation algorithm. The framework offered support for 10,000 customers, starting from an initial allocation topology, up to a topology that minimized the initial average latency by 33%.

Keywords: IoT; Microservices; Microservice Allocation; Framework.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

SUMÁRIO

INTRODUÇÃO

Os mecanismos e as tecnologias desenvolvidas no contexto da Internet das Coisas (*Internet of Things*, IoT) têm o potencial para viabilizar aplicações que vão fazer uso de milhares de dispositivos e serviços distribuídos. Em 2010, o número de dispositivos conectados à Internet já estava na ordem de 10^9 e estimava-se que ele alcançasse 10^{11} até o final de 2020 [1] [2], o ano corrente.

Entre os mecanismos e tecnologias mencionados estão aqueles para permitir a integração de dispositivos heterogêneos [3] [4] [5] e viabilizar a interação entre dispositivos, serviços e módulos de software através de troca de mensagem sobre a Internet, usando a pilha de protocolos TCP/IP [6] [7] [8]

A arquitetura de software de aplicações atuais e das aplicações que ainda vão ser desenvolvidas para IoT incorporam recorrentemente (i) sistemas para a comunicação com dispositivos para coleta de dados de sensores e acionamento de atuadores; (ii) rotinas para envio dos dados coletados para algum repositório para serem persistidos e (iii) módulos de inteligência ou regras, para a tomada de decisões e ações que devem ser refletidas nos vários elementos da aplicação, incluindo os dispositivos monitorados e os próprios serviços utilizados.

Motivação

Os sistemas responsáveis pela comunicação com os dispositivos devem estar preparados para operar em cenários altamente distribuídos e lidar com uma quantidade de dispositivos em escala sem precedentes. Mais especificamente, permitir que aplicações façam uso de muitos dispositivos e, ainda, que os tempos envolvidos na interação com estes dispositivos e com os serviços necessários, esteja dentro de limites aceitáveis. Este é o foco principal da presente dissertação.

Os dispositivos que podem fazer parte da IoT são heterogêneos tanto em sua funcionalidade como em sua capacidade de processamento e comunicação. Podem ser empregados em pequena escala, como em ambientes residenciais inteligentes, até escalas maiores como cidades e indústrias inteligentes. Como exemplos, temos lâmpadas inteligentes e conectadas, alto-falantes com assistente digitais integrados e câmeras de segurança, no caso de ambientes residenciais inteligentes; dispositivos mais complexos para

Rede Elétrica Inteligente (*Smart Grid*) e as câmeras e semáforos de trânsito de uma cidade inteligente; ou sensores e atuadores de missão crítica em plantas industriais, na Indústria 4.0 [9] [10] [11].

A diversidade e heterogeneidade de dispositivos apresenta dois desafios dentro do foco deste trabalho:

- alguns dispositivos tem capacidade limitada de processamento e persistência. Assim, é frequentemente necessário (i) delegar o processamento para outros dispositivos ou para nós de processamento, com mais recursos; e (ii) enviar dados aquisitados pelos sensores para nós com maior capacidade de armazenamento, caso seja necessário manter algum histórico desses dados. Esta técnica é chamada de *off-loading* na IoT [12] [13];

- os dispositivos podem (i) oferecer interfaces de comunicação diferentes, (ii) interagir através de Protocolos de Aplicação diferentes e (iii) ter capacidade limitada de comunicação, com pequena largura de banda, velocidades ou taxas de transferências baixas. Os dois primeiros pontos estão associados à integração dos dispositivos e estão fora do escopo deste trabalho (sugerimos consulta aos trabalhos [7] [3]). O terceiro ponto é tratado em nossa proposta.

Objetivos

O objetivo geral, como aprofundado adiante, é alocar instâncias de serviços em nós de processamento selecionados, dentro de um conjunto disponível, e dividir a demanda de serviços da aplicação por estas instâncias, de forma a se diminuir, em média, os gargalos de processamento e atrasos de comunicação (Seção 4.2 do Capítulo 4). O critério de seleção dos nós de processamento pode ter influência na qualidade do resultado e é outro ponto de atenção em nossa proposta, sendo discutido separadamente no Capítulo 6 e, depois, na Seção 7.2.

Seguindo a mesma linha, a seleção de recursos e a alocação de serviços são, em si, outros dois ponto de atenção em nossa proposta:

- a seleção de recursos, na forma tratada aqui, requer que uma lista de elementos, nós de processamento e serviços, seja montada e esteja disponível para consulta, e que os recursos destes elementos sejam constantemente monitorados. Por exemplo, %CPU

e quantidade de memória disponíveis, bem como o atraso de comunicação entre os elementos. Com estas informações seria possível um módulo de orquestração selecionar a melhor instância de determinado recurso ou serviço para atender a demanda da aplicação, e aceitar uma demanda maior ou diminuir a demanda atual de requisições redistribuído as requisições por outras instâncias;

- técnicas de replicação e alocação dinâmica de serviços têm sido utilizadas em soluções para se prover escalabilidade e tolerância à falhas em sistemas distribuídos [14], [15], [16], [17]. Isso está associado à possibilidade de se instanciar e controlar remotamente a *imagem* de um serviço pela rede. Soluções de virtualização como as Máquinas Virtuais oferecem este suporte através de suas APIs [18]. Atualmente, abordagens com microserviços e conteinerização têm sido adotadas com vantagens em relação às máquinas virtuais [19] [20] para oferecer portabilidade, mobilidade, alocação e relocação de serviços. Este ponto será aprofundado na Seção 1.3.

Um último ponto de foco em nossa proposta é, justamente, relacionado ao conjunto de elementos de processamento disponível. Aplicações típicas na Internet das Coisas podem demandar o uso de grande capacidade de processamento e armazenamento, seja para persistir e manter o histórico das informações coletadas para consulta, seja para tratar e realizar inferências sobre estas informações. Ainda, a natureza destas aplicações, a localização de dispositivos, serviços e usuários destas aplicações podem ser geograficamente distribuídas. Além disso, como estratégia de execução e operação, a capacidade de processamento pode também ser distribuída e compartimentada, seja para paralelizar rotinas, para oferecer tolerância a falhas, ou para atender dinamicamente à demandas variáveis, bem como para diminuir latências e atrasos de comunicação. Assim sendo, os recursos utilizados pelas aplicações em IoT são frequentemente distribuídos, e a interação entre os seus elementos suportada por topologias de comunicação dependentes desta distribuição.

É possível estruturar os elementos da aplicação, os recursos utilizados e a topologia de comunicação de forma hierárquica, de forma análoga à hierarquia de memória, onde temos memórias muito rápidas, mas em menor quantidade, e memórias relativamente mais lentas, mas em maior quantidade. Aplicações desenvolvidas para a Internet das Coisas geralmente podem contar com nós de processamento com os seguintes perfis:

- Locais. Nós da rede local, onde se encontram os dispositivos de interesse ou nós de processamento disponíveis, que apresentam baixa latência na comunicação, mas os

recursos podem ser limitados ou podem se esgotar rapidamente. Um dispositivo, sensor ou atuador, pode fazer o papel deste nó, caso haja recursos e capacidade;

- Borda (Edge). Nós que estão na fronteira entre a rede local e uma rede mais ampla, ou a própria Internet. Geralmente possuem recursos mais abundantes que os nós locais e também oferecem baixa latência de comunicação, se considerados como interlocutores os dispositivos ligados na mesma rede local. Outra característica intrínseca é sua função de concentrador do fluxo de dados. Em princípio, os nós de borda repassam os dados coletados ou enviados por todo os dispositivos da rede local. Assim, poderiam também armazenar, mesmo que temporariamente, e (pré)processar estes dados antes de repassá-los;

- Névoa (*Fog*). Consideram-se nós de *fog* aqueles localizados em redes com latência baixa e média, em relação aos seus interlocutores, o que geralmente ocorre em redes de alta velocidade e à poucos *hops* (roteadores no meio do caminho) de distância. Estes nós geralmente possuem recursos em maior quantidade, comparados aos nós locais e aos de *edge*, mas que também são limitados, à depender da demanda da aplicação [21][1];

- Nuvem (*Cloud*). Nós de processamento, virtualizados, geralmente oferecidos como serviços através de chamadas remotas, que apresentam recursos com menos limites, ou "elásticos" — podem ser adicionados sob demanda. Os nós físicos sobre os quais são executados os nós virtualizados podem estar à muitos *hops* de distância do seu interlocutor, o que geralmente implica em latências que podem ser altas ou muito altas.

Uma característica notável das arquiteturas propostas para IoT é a presença de elementos chamados IoT-*gateways* [23]. Esses dispositivos, geralmente localizados na borda da rede e próximos aos dispositivos e agentes de software da aplicação, desempenham tarefas como conversão de protocolos, fusão de dados ou (pré)análise de informações. Esses elementos podem ser máquinas de propósito geral, que possuem componentes de *software* que irão desempenhar essas tarefas [7], ou dispositivos dedicados, com *hardware*/*software* específicos [24]. Assim, fazendo-se uso de um IoT-*gateway* também estamos usando um

[1] Algumas referências, como [22], não fazem distinção dos recursos de computação de *edge* e *fog*. Assim os termos podem ser usados de forma intercambiável.

elemento de borda (*edge*). Com efeito, alguns equipamentos dedicados à serviços de borda têm incorporado as funções tradicionais de roteador, *firewall* e NAT e também funções de IoT-*gateway* ou de *proxy* para vários serviços de rede [25].

Neste contexto, propusemos um *framework* que conecta dinamicamente os módulos das aplicações IoT, sejam eles agentes de software realizando a interface com os dispositivos e clientes à instâncias de serviços de persistência ou de processamento, ou ainda serviços de suporte e de *middleware*. As instâncias de serviços podem estar executando em nós de processamento de *edge*, *fog* ou *cloud*, ou ainda no próprio IoT-*gateway*. A seleção da instância à qual um agente ou cliente vai ser conectado é feita de tal forma que uma aplicação IoT possa obter o melhor desempenho possível, com a comunicação e a execução de rotinas executadas em tempos aceitáveis, para uma escala grande de dispositivos e elementos, independente de sua localização (*edge*, *fog* ou *cloud*). O critério de decisão de conexão de de um módulo à um determinado nó de processamento, pode depender de algumas métricas: a quantidade de processamento que deve ser realizada; a quantidade de dados a serem transferidos pela rede e o tempo máximo permitido para se realizar esta transferência; e a necessidade de armazenamento (ver Capítulo 6).

Implementação

O *framework* proposto, central neste trabalho, foi implementado com o objetivo de integrar os mecanismos necessários para concretizar as estratégias e abordagens discutidas até aqui, de forma que fosse possível experimentar e avaliar técnicas de *off-loading* para a conexão de módulos de software à serviços executando em nós de processamento distribuídos em *edge*, *fog* e *cloud*, para aplicações IoT, e se estas técnicas poderiam oferecer escalabilidade e desempenho aceitáveis para estas aplicações.

Além do *framework* desenvolvido, para realizar as avaliações e como prova de conceito, um sistema de *middleware* para aplicações IoT foi utilizado. O objetivo foi estruturar a avaliação com uma aplicação real composta de módulos cliente, dispositivos e serviços que fariam uso de serviços típicos de IoT como autenticação, conversão de protocolos, leitura de sensores ou envio de comandos para atuadores, fornecidos pelo sistema de *middleware*. O SiteWhere 2.0.2 foi selecionado com esta finalidade, dadas as possibilidades de integração com o *framework* proposto, conforme será discutido em mais detalhes na Seção 3.2 do Capítulo 3.

O *framework* atua de maneira transparente, de forma que os módulos-cliente, dispositivos e serviços da aplicação são desenvolvidos prevendo-se a realização de chamadas apenas à API do SiteWhere. Entretanto, a API do *framework* proposto poderia ser usada diretamente para acelerar algumas operações de reconfiguração. Uma analogia poderia ser feita em relação à sistemas de "coleta de lixo", que geralmente é realizada automaticamente pelo sistema de execução, mas poderia ser acionada pela própria aplicação em determinadas situações críticas. Este ponto é discutido na Seção 4.5.

Como mencionado anteriormente, outra parte importante do *framework* é o mecanismo de orquestração utilizado para determinar onde cada requisição será processada. Neste sentido, o *framework* foi estruturado para permitir que vários algoritmos fossem avaliados, como destacado na Seção 4.7. Foram escolhidos cinco algoritmos disponíveis na literatura — ERA [26], CS, Lat [27], GABVMP [28] e GMEDSWC [29] — e cada um deles foi implementado e integrado ao orquestrador para a avaliação. Esses algoritmos são discutidos separadamente no Capítulo 6.

O *framework* foi implementado sobre o suporte de contêineres baseados em Docker [30], explorando-se as vantagens para configurar e instanciar (micro)serviços dinamicamente em nós de processamento distribuído. Também foram exploradas as características de gerenciamento de rede e de grupos (*clusteres*) de contêineres. Aliado a isso, a escolha do SiteWhare como *middleware* IoT também se deu porque este também foi desenvolvido utilizando microsserviços conteinerizados em Docker. Ou seja, cada parte do sistema é disponibilizada em uma imagem de um contêiner. As dependências do sistema como o banco de dados, *broker* de mensagem e o serviço de descoberta utilizados também são instanciados utilizando contêineres. Assim, a estrutura do *framework* proposto e os mecanismos necessários para efetivar as ações de orquestração foram propostos considerando este aspecto e isso se refletiu na implementação de referência.

Ressalta-se, entretanto, que no processo de implementação do *framework* e de integração dos vários elementos, foram encontradas limitações nas ferramentas de rede do Docker, relacionadas ao endereçamento de redes IP e de uso de NAT. Assim, em certo ponto do trabalho, foi necessário desenvolver uma infraestrutura de rede *overlay* contornando estes problemas. Com isso as aplicações propostas para avaliação do *framework* poderiam utilizar nós de processamento em máquinas publicas na nuvem, máquinas dentro da rede publica da UERJ — simulando *fog*, e máquinas em redes locais e na borda,

ainda que utilizassem faixas de endereçamento específicos e NAT. Detalhes das limitações encontradas e a solução proposta são discutidos separadamente no Capítulo 5.

Avaliação

Com o *framework* implementado e suas funcionalidades testadas, foram, então realizados dois conjuntos de testes para a avaliação. Em um primeiro momento os diversos algoritmos de alocação foram comparados. Os resultados indicaram o ERA como algoritmo de alocação com melhor desempenho para minimizar a latência da aplicação. A partir dos resultados do primeiro conjnto de testes o ERA foi empregado para realizar o segundo conjunto de testes onde a escalabilidade do *framework* foi avaliada com testes envolvendo 10, 20, 200, 500, 1000 e 10000 agentes[2] executassem em tempo aceitável, melhorando em até 33% o desempenho.

Organização da dissertação

Esta dissertação está organizada da seguinte forma: o Capítulo 1 apresenta os conceitos empregados; O Capítulo 2 apresenta trabalhos relacionados; O Capítulo 3 trata das tecnologias selecionadas, destaque para o Docker e SiteWhare, como base para a criação do *framework* apresentado neste trabalho. Além disso, são discutidas as razões para estas escolhas e eventuais adaptações realizadas; No Capítulo 4 é, então, apresentado o *framework* desenvolvido, sua arquitetura, a API proposta, e o orquestrador. Também são discutidos detalhes da implementação de referência. Dois elementos do *framework* são apresentados separadamente: a rede *overlay* criada para resolver uma limitação encontrada nas redes *overlay* do docker é apresentada no Capítulo 5 e os algoritmos selecionados e comparados na avaliação, que deram "musculatura" ao orquestrador são apresentados no Capítulo 6. A visão geral, implementação e resultados da avaliação de desempenho são apresentados no Capítulo 7 juntamente com uma discussão sobre os resultados obtidos.

[2]Ao longo do texto o termo *agente* será utilizado para representar clientes e dispositivos, sejam eles sensores, atuadores ou híbridos.

1 CONCEITOS UTILIZADOS

Esse capítulo apresenta alguns dos conceitos utilizados na proposta, como: *edge*, *fog*, *cloud*, contêineres e microserviços (versus VMs), redes NAT.

1.1 Conceitos básicos

Ao longo dessa Seção iremos utilizar alguns exemplo para apresentar e discutir os conceitos de (i) sensores, (ii) atuadores e (iii) gateway. A partir destes conceitos iremos delimitar o conceito de (iv) clientes e (v) agentes e por fim, será formalizado o conceito de (vi) nó de processamento que será utilizado nos próximos Capítulos.

Vamos tomar como primeiro exemplo um *arduino* [31] com um sensor de temperatura que periodicamente realiza a leitura do sensor e envia essa informação para um sistema externo. Devida a limitação do hardware desse dispositivo, ele não realiza nenhuma outra operação e não oferece nenhum outro serviço. Bem como ele não mantém o histórico dos dados.

Esse é um exemplo clássico de um dispositivo (i) sensor. A principal funcionalidade dele é detectar/captar alterações no ambiente em que está inserido, neste caso alterações na temperatura. Além dos sensores físicos, existem ainda os sensores virtuais, como os componentes de software que agregam os valores obtidos por diversos sensores ou ainda os sensores que monitoram componentes de software.

Este simples sensor de temperatura expõem outras características intrínsecas à IoT, como por exemplo, devido à limitação de hardware, os dados gerados por esse sensor devem ser armazenados em outros dispositivos se for necessário manter o histórico desses valores [32].

Ainda abordando os conceitos mais simples de IoT, vamos utilizar o mesmo exemplo do *Arduino*, entretanto agora vamos trocar o sensor de temperatura por um controlador, por exemplo para controlar uma lâmpada. Ao invés de enviar as leituras do sensor, agora esse dispositivo aguarda comandos para realizar o acionamento desta lâmpada.

Esse exemplo simples demonstra um (ii) atuador. A principal funcionalidade dele é atuar no ambiente em que está inserido, neste caso controlando uma lâmpada. De forma similar aos sensores, também existe o conceito de atuadores virtuais. Novamente podemos utilizar este exemplo para apresentar outros desafios presentes em IoT, como por exemplo

o fato de que esse controlador pode esperar os comandos em diversos formatos diferentes e pode aceitar diferentes parâmetros de acordo com o hardware disponível.

A partir desses dois conceitos simples, sensores e atuadores, podemos classificar os dispositivos responsáveis pelas coletas de dados bem como os dispositivos que realizam ações que alteram o ambiente no qual eles estão inseridos. Esses dispositivos estarão presentes em diversos cenários, seja nos mais simples como uma casa inteligente [33] ou nos mais complexos como nas industrias inteligentes, por exemplo na mineração [34]

Visto que até nos cenários mais simples é possível encontrar os desafios criados pela heterogeneidade presente em IoT, é trivial compreender a importância de se ter algum componente, seja de hardware ou software, responsável por garantir a interoperabilidade dos diversos dispositivos e protocolos.

Os (iii) *gateways* IoT surgem como solução de hardware para este problema e podem ser desenvolvidos especificamente para alguma aplicação como no trabalho [35] ou podem ser um produto de prateleira [24]. Além de realizar a conversão entre os protocolos específicos de cada sensor [36] os *gateways* podem ainda oferecer outros serviços dado que, geralmente, possuem mais recursos que um simples sensor/atuador. Entre esses serviços podemos destacar agregação de dados, criptografia e gerenciamento dos dispositivos.

Outro ponto que podemos destacar aqui é que, devido ao alto nível de customização possível nesse tipo de dispositivo, é comum um mesmo sistema de IoT esteja utilizando diversos protocolos de rede ao mesmo tempo e por isso a importância de *middlewares* que possam tornar essa complexidade transparentes para os clientes. [3]

Com base nesses componentes podemos definir que um (iv) cliente é toda entidade que consome os dados e serviços oferecidos pelos (i) sensores e (ii) atuadores. Seja acessando diretamente esses componentes, seja através de um gateway, ou mesmo utilizando um *middleware*.

A fim de simplificar a nomenclatura dos (i) sensores e dos (ii) atuadores o termo (v) agentes será utilizado para determinar toda entidade que pode ser classificada como (i) sensor e/ou (ii) atuador.

1.2 *Edge*, *Fog* e *Cloud* (Borda, Névoa e Nuvem)

Conforme apresentado na Introdução as aplicações desenvolvidas para a Internet das Coisas geralmente podem contar com nós de processamento nos seguintes perfis:

- Locais. Nós da rede local, onde se encontram os dispositivos de interesse ou nós de processamento disponíveis, que apresentam baixa latência na comunicação, mas os recursos podem ser limitados ou podem se esgotar rapidamente. Um dispositivo, sensor ou atuador, pode fazer o papel deste nó, caso haja recursos e capacidade;

- Borda (*Edge*). Nós que estão na fronteira entre a rede local e uma rede mais ampla, ou a própria Internet. Geralmente possuem recursos mais abundantes que os nós locais e também oferecem baixa latência de comunicação, se considerados como interlocutores os dispositivos ligados na mesma rede local. Outra característica intrínseca é sua função de concentrador do fluxo de dados. Em princípio, os nós de borda repassam os dados coletados ou enviados por todo os dispositivos da rede local. Assim, poderiam também armazenar, mesmo que temporariamente, e (pré)processar estes dados antes de repassá-los;

- Névoa (*Fog*). Consideram-se nós de *fog* aqueles localizados em redes com latência baixa e média, em relação aos seus interlocutores, o que geralmente ocorre em redes de alta velocidade e à poucos *hops* (roteadores no meio do caminho) de distância. Estes nós geralmente possuem recursos em maior quantidade, comparados aos nós locais e aos de *edge*, mas que também são limitados, à depender da demanda da aplicação [21];.

- Nuvem (*Cloud*). Nós de processamento, virtualizados, geralmente oferecidos como serviços através de chamadas remotas, que apresentam recursos com menos limites, ou "elásticos" — podem ser adicionados sob demanda. Os nós físicos sobre os quais são executados os nós virtualizados podem estar à muitos *hops* de distância do seu interlocutor, o que geralmente implica em latência que pode ser alta ou muito alta.

1.3 Máquinas Virtuais, Contêineres e Microserviços

As técnicas tradicionais de virtualização permitem que diversos sistemas sejam executados dentro de uma mesma maquina hospedeira. A principal vantagem da virtualização é o isolamento e a segurança dessas instâncias. Entretanto, em dispositivos com poucos recursos disponíveis, como os *gateways*, isso pode ser considerado como uma das principais desvantagens da virtualização, visto que as instancias são completamente

isoladas e não é possível reaproveitar recursos comuns, como sistemas operacionais, bibliotecas, etc.

A conteinerização surge como alternativa, ela oferece a mesma possibilidade de executar diversos processos isoladamente dentro de uma mesma máquina. Entretanto, aqui esse isolamento é mais fraco do que se comparado ao da virtualização, o que introduz mais desafios para a segurança, mas permite que recursos sejam compartilhados. Como por exemplo o sistema operacional, bibliotecas e o hardware. Assim o resultado é uma forma de executar diversos sistemas e serviços com um custo de recursos menor que o da virtualização.

1.4 NAT

NAT (*Network Address Translation* ou Tradução de Endereço de Rede) é um processo que envolve converter um único endereço IP em outro endereço IP, muitas vezes Público, através da alteração das informações de rede e informações de endereço encontradas no cabeçalho IP dos pacotes de dados. Se tomarmos como exemplo uma rede local doméstica é comum que o equipamento de borda realize NAT. Neste caso todos os dispositivos conectados a esse equipamento de borda recebem um endereço local (10.0.0.0/8, 172.16.0.0/12, ou 192.168.0 0/16). Através do NAT, esses endereços locais são traduzidos em um endereço IP público quando são enviadas solicitações para outra rede que não a local. Para que a resposta possa ser entregue o dispositivo responsável pelo NAT mantém uma tabela com o mapeamento realizado. Ao receber a resposta o NAT altera o IP público para o IP local e redirecionados dados.

Uma limitação em compartilhar um único IP público para diversos dispositivos é que não é possível que dispositivos fora da rede NAT estabeleçam um nova conexão com os dispositivos dentro do NAT, sem que sejam realizadas configurações extras no NAT como mapeamento fixo de portas.

Por exemplo se na Figura 1 o host A possui apenas o IP público do host B não é possível estabelecer uma conexão, considerando que nenhum mapeamento fixo foi realizado no NAT. É importante notar que o host B pode iniciar a conexão e com isso o mapeamento seria realizado no NAT e o host A seria capaz de responder as requisições enviadas pelo host B.

Ainda nesse cenário, o host B e o host C não são capazes de trocar mensagens,

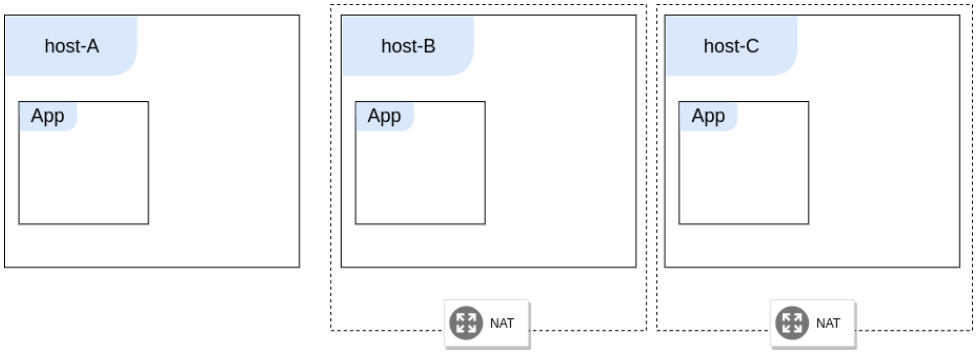


Figura 1 - NAT

independente de quem inicie a conexão. Para que eles sejam capazes de se comunicar é necessário utilizar técnicas de *NAT Traversal* (NAT transversal).

2 TRABALHOS RELACIONADOS

Neste capítulo são apresentados diversos trabalhos relacionados que serviram de base para o desenvolvimento do *framework* e para a sua avaliação de desempenho.

No *survey* [37] são apresentados casos de uso para *edge computing*, bem como desafios existentes. Entre eles podemos destacar o gerenciamento de serviços e as métricas de otimização. Além disso, o *survey* apresenta cinco casos de estudo, sendo eles:

(i) Cloud offloading. São considerados algum cenários envolvendo sistemas de compra online e como o carrinho de compra poderia se beneficiar da baixa latência dos nós de processamento da edge

(ii) Video Analytics. São abordados os benefícios que o processamento na edge pode trazer para esse cenário.

(iii) Casas inteligentes. Um dos cenários mais comuns para o *edge computing* é apresentado junto da de um sistema operacional destinado para este cenário, o *edgeOS*

(iv) Cidades inteligentes. Novamente são destacados os benefícios de realizar o processamento mais próximo da fonte do dado.

(v) Borda colaborativa. Apresenta um cenário onde diversos *stakeholders* compartilham seus dados presentes na edge de maneira *ad-hoc*. Esses dados então poderiam ser utilizados de forma colaborativa.

Já o artigo [38] define as principais características de *fog computing* e apresenta alguns casos de uso em IoT. É apresentada uma arquitetura com múltiplos níveis contendo uma rede de sensores, nós de processamento de *fog/edge* e uma camada de nós de processamento *cloud*. Os casos de uso apresentados são veículos conectados, *smart grid* e rede de sensores e atuadores.

Em [39] é apresentado um algorítimo *online* que realiza o cache de serviços no contexto de *mobile edge computing*. É analisado o impacto de fatores como capacidade de armazenamento, restrições de energia, entre outros. Apesar de ser direcionado para *Mobile Edge Computing* (MEC) diversos desafios são compartilhados com a Internet das Coisas, como a heterogeneidade dos serviços/clientes, aplicações sensíveis à latência, etc.

O artigo [40] apresenta um algorítimo de alocação de serviços desenvolvido para um cenário de realidade virtual compartilhada que utiliza conceitos de *edge computing* para alocar os serviços mais próximos ao usuário. Este artigo utiliza um grafo para representar o sistema e converte o problema de otimização da alocação dos nós em um problema de corte de grafo. São utilizados então algoritmos de max-flow para realizar a alocação dos serviços. Esta abordagem poderia ser utilizada em conjunto com o nosso *framework*.

O *survey* [41] apresenta um panorama completo do estado da arte do *fog computing*. Vale destacar deste *survey* a Section II, em especial o item E, onde são discutidos as similaridades e diferenças entre *fog*, MEC e *coudlet* e a Section IV onde são apresentadas diversas arquiteturas para sistemas de *fog*.

Segundo [42], em Fog Computing os principais desafios relacionados à alocação de serviços e o gerenciamento de recursos são: Latência; Consumo de energia; Compartilhamento de recursos; Cache; *Placement* de máquinas virtuais; *Uncertainty* da posição e avaliabilidade dos nós da Fog; Infraestrutura flexível para os operadores de telecomunicação; Além disso, esse *survey* ainda apresenta os arquétipos de aplicações nas áreas de: Smart cities; Smart energy management; Industrial wireless sensor networks; Remote gaming; Healthcare; Intelligent transport system; O nosso *framework* busca minimizara latência do sistema e realizar a alocação dos serviços e isso se relaciona diretamente com os desafios enumerados pelo autor. Além disso, a solução apresentada na Seção 4 pode ser utilizada em diversos do cenários apresentados.

Já o artigo [43] apresenta uma programação não linear inteira mista para o problema de alocação de tarefas, gerenciamento de recursos e balanceamento de carga no contexto de sistemas embarcados definidos por *software*(*Software-Defined Embedded System*). Neste contexto os nós da *edge* não possuem o software que será executado quando é iniciado e devem busca-lo dinamicamente em um outro nó presente na fog. Por isso é comum que durante a execução da tarefa ocorram erros de falta de página (page faults) e seja necessário consultar novamente o nó da fog para requisitar as páginas faltantes. A partir da modelagem em programação não linear inteira mista desse sistema o autor desenvolve um algorítimo e realiza uma avaliação de desempenho.

O artigo [44] estabelece requisitos para um gerente de recurso para *edge computing*. É utilizado como comparação outro gerente de *IaaS*(sigla em inglês para infraestrutura como serviço) o *OpenStack*.

Em [45] é apresentado um *middleware* para a alocação de tarefas na nuvem e em *fog cells*, que são definidas, pelos autores, como componentes de *software* executados pelos dispositivos. Esse *middleware* busca otimizar a utilização dos recursos da fog. A arquitetura apresentada é composta de 3 camadas. (i) o nível mais próximo do dispositivos IoT é composto por diversas *fog cell*, formando uma *fog colony*. (ii) Essas colonias são orquestradas por um nó de controle. (iii) por sua vez esse nó de controle é controlado pelo *middleware* na cloud. Essa solução se assemelha a utilizada pelo nosso *framework*, visto que é utilizada uma arquitetura em camadas e que ambas as soluções utilizam um nó de controle. Entretanto, dentre as principais diferenças vale destacar que a nossa solução é extensível e permite que novos algoritmos sejam adicionados ao nó de controle.

O artigo [46] investiga a relação entre o consumo de energia e o *delay* na alocação de tarefas em um cenário com *fog* e *cloud computing*. O cenário é formulado como um problema de otimização com o objetivo de minimizar o consumo de energia respeitando a restrição do *delay*.

O artigo [47] avalia a utilização do Docker como plataforma de edge. São avaliados os seguintes aspectos:

1. Ciclo de vida das aplicações. São avaliadas as opções oferecidas pelo Docker para inicializar e finalizar a execução de um contêiner,

2. Gerenciamento de recursos e serviços. São avaliados os recursos de um *swarm* Docker e como a descoberta de serviços pode ser realizada utilizando as ferramentas oferecidas.

3. Tolerância a falha. São apresentadas ferramentas auxiliares que podem ser utilizada em conjunto com o Docker para oferecer monitoria dos serviços, gerenciamento dos discos, migração offline de serviços e *deploy* de múltiplas zonas.

4. Cache. Explora os benefícios de utilizar um cache dos dados gerados no nós da edge.

5. Aplicações. São exploradas algumas das vantagens de se utilizar o Docker para fazer o deploy de uma aplicação, no caso do artigo é utilizado como exemplo o Hadoop.

O artigo [48] apresenta uma análise de escalabilidade de um sistema de *edge computing* coltado aplicações sensíveis a latência, como por exemplo realidade aumentada.

Foram executados diverso testes de performance e o autor destaca 3 resultados importates:

1. Um sistema que utiliza apenas o processamento cloud pode obter resultados melhores se houver pouca banda disponível na conexão entre os nós da edge.

2. Um sistema que possua uma boa conexão entre todos os nós (cloud e edge) se aproxima da alocação global ótima

3. Aumentar o poder computacional dos nós da edge, sem alterar a banda disponível pode causar um congestionamento no sistema, degradando a qualidade média do sistema.

O artigo [49] faz uma prova de conceito para avaliar a utilização do Docker como base para um sistema de *fog computing deployment*. Esse artigo usa como base o gateway Kura em conjunto com o protocolo MQTT. Os resultados são positivos e demonstram que até mesmo um host com recursos limitados como um RaspberyPi pode ser utilizado com o Docker para um cenário de IoT e *fog computing*. A abordagem apresentada por este artigo é bem próxima a utilizada pelo nosso *framework*, desde a utilização do Docker até a utilização de um *middleware* IoT. A nossa solução tem como diferencial a escalabilidade e a extensibilidade.

O artigo [50] faz uma avaliação quantitativa e qualitativa de diversos sistemas de *middleware*. Foram selecionados alguns sistemas *open source* e uma solução proprietária apresentada pelo autor. Os sistemas de *middleware* foram submetidos a uma avaliação qualitativa focada nos aspectos de segurança considerados ideais para o autor. Os que se destacaram foram submetidos à avaliação quantitativa composta por diversos testes de desempenho. Entre os avaliados podemos destacar: (i) o *Konker* por ser o único avaliado a possuir determinados componentes de segurança, como autenticação exclusiva para cada dispositivo; (ii) o Fiware(Orion+STH) e (iii) o SiteWhere por apresentarem os melhores resultados quantitativos. Foram realizados testes com 100, 1000, 5000 e 10000 usuários concorrentes. Os testes realizados por este artigo serviram de base para os testes realizados no Capítulo 7.

De maneira similar o artigo [51] realiza uma avaliação de performance de plataformas *open source* de IoT. O artigo compara a performance do SiteWhere e do ThingsBoard. O JMeter é utilizado para simular a carga no sistema. São realizados testes com

10,100,200,500,800 e 1000 clientes. Além disso, são realizados testes variando a frequência de envio das mensagens.

3 TECNOLOGIAS UTILIZADAS

Esse capítulo apresenta as tecnologias adotadas como base para a proposta do *framework* e na construção da implementação de referência: o Docker, software de conteinerização, e o SiteWhere, middleware utilizado como suporte para as aplicações desenvolvidas sobre o *framework*.

A Seção 3.1 apresenta detalhes do Docker, incluindo detalhes da sua rede na subseção 3.1.1. Ainda nessa subseção são discutidas limitações que foram encontradas na criação dos clusters.

O SiteWhere é apresentado na Seção 3.2. São detalhadas a arquitetura (subseção 3.2.1), os serviços disponíveis(subseção 3.2.2) e por último as modificações realizadas para a criação do *framework*(subseção 3.2.3).

3.1 Docker

O Docker [30] é uma ferramenta que provê a virtualização por *contêineres*, discutida na Seção 1.1. Os contêineres compartilham recursos da máquina hospedeira e são uma alternativa para as máquinas virtuais (VM). Na versão para Linux, o Docker utiliza recursos do *kernel* como *cgroups* e *namespaces* para permitir que aplicações sejam executadas de maneira isolada de outros processos. Existem alternativas para o uso de Docker em outros sistemas oepracionais.

O Docker utiliza a abstração de imagens para definir o modelo para um contêiner. As imagens são criadas a partir de um arquivo YAML [52] chamado Dockerfile, onde são definidas a imagem base e uma série de comandos a ser executada. Por exemplo, é possível utilizar como base uma imagem Ubuntu e, através dos comandos, adicionar todas as dependências necessárias para a aplicação a ser executada, e a imagem da própria aplicação. Uma vez definida esta imagem ela pode ser executada e replicada. Cada instância recebe o nome de contêiner. Essas imagens podem ser utilizadas como base para novas imagens e podem ser disponibilizadas em repositórios como o *Docker Hub*.

O Docker utiliza como base para a criação dessas imagens um sistema de arquivos em "cebola" chamado Unionfs. Esse sistema de arquivo permite que cada ação descrita no Dockerfile seja registrada como uma camada. Isso trás como benefício imagens resultantes menores se comparadas às máquinas virtuais. Além do resultado final, de uma imagem

menor, o benefício aumenta quando levamos em consideração que diversas imagens podem utilizar as mesmas camadas. Por exemplo se duas imagens diferentes utilizam a mesma base, o Docker só precisa manter uma cópia dessa base. As 2 imagens irão conter apenas com as suas camadas específicas.

Vale destacar uma das ferramentas auxiliares do Docker, o docker-compose. Essa ferramenta oferece uma API básica para a orquestração de imagens do Docker. Com ela é possível definir um YAML contendo uma lista de imagens, a dependência entre elas e alguns parâmetros que devem ser passados para o Docker. Entretanto, o docker-compose se limita a oferecer essa API e não possui nenhum algoritmo de alocação ou orquestração pré-definido.

Outra ferramenta importante do Docker é a sua rede. O Docker oferece uma abstração de redes virtuais, que permite que novas redes sejam criadas para os contêineres. O comportamento dessa rede virtual depende do *driver* utilizado e será detalhada na Subseção 3.1.1.

O Docker também oferece uma ferramenta para criação de *clusters*, chamados *swarm* (no sentido de grupo, enxame). O *swarm* oferece serviços de balanceamento de carga, redirecionamento de requisições, entre outros. Os nós que compõem um *cluster* podem ser gerentes (*Managers*) ou trabalhadores (*Workers*). Todos os nós podem executar contêineres e fazer parte das redes virtuais criadas entre eles. Entretanto, apenas os gerentes podem criar/iniciar/parar/deletar contêineres.

Para que o *framework* proposto pudesse orquestrar os clietes, agentes e serviços era necessária uma forma de representar e referenciar estes elementos, principalmente os serviços e permitir que os mesmos fossem dinamicamente instanciados. A representação deveria ser flexível o suficiente para permitir que o *framework* determinasse o que deveria executar, onde, e quando. Além disso, não deveriam existir limitações para a linguagem utilizada pelo serviço e, principalmente, a orquestração deveria ser transparente para clientes e agentes da aplicação. Estas características estão presentes no Docker. Assim, a proposta do *framework* e, principalmente, a implementação de referência (apresentada no Capítulo 4) adotam o Docker como base para a infraestrutura.

3.1.1 Rede do Docker

Conforme apresentado na seção anterior, as redes virtuais criadas pelo Docker podem utilizar diversos *drivers*. Esses *drivers* determinam como a rede irá se comportar. Por padrão o Docker inclui os seguintes *drivers*: *host*, *overlay*, *macvlan*, *none*.

O *framework* utiliza os drivers *host* e *overlay*. Os *drivers bridge*, *macvlan* e *none* não foram utilizados neste trabalho.

A rede *host* permite que um contêiner seja conectado diretamente à rede do computador hospedeiro, permitindo que portas sejam reservadas e acessadas diretamente pelas aplicações. Esse tipo de rede se aproxima do comportamento apresentado por uma aplicação que é executada diretamente na máquina hospedeira.

A rede *overlay* cria uma *vxlan* entre os nós e permite que cada contêiner receba o seu próprio IP virtual. Desta forma cada contêiner pode utilizar qualquer porta e acessar qualquer contêiner dentro da rede virtual. Para que o acesso entre contêineres seja possível o Docker realiza todo o roteamento necessário. Além disso o Docker também oferece facilidades de roteamento como o mapeamento das portas dos contêineres para portas do hospedeiro. Esta facilidade, chamada de *routing mesh* (Figura 2), permite, então, que determinado serviço seja acessado pela porta escolhida em qualquer hospedeiro do *cluster*, independente de existir algum contêiner naquele *host* executando o serviço. O Docker se encarrega de realizar o roteamento de forma transparente.

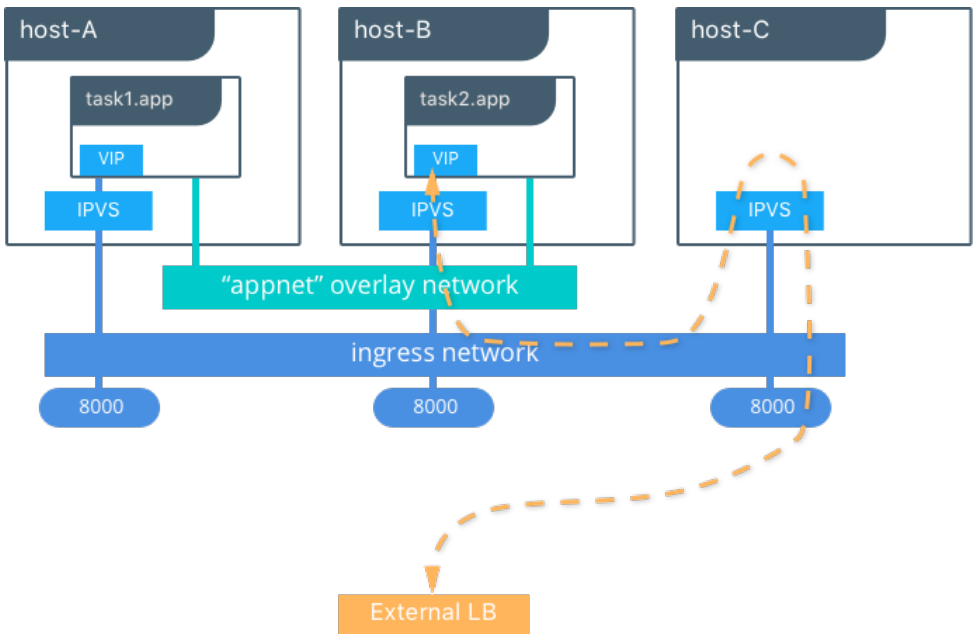


Figura 2 - Docker - rede overlay routing mesh

Durante o desenvolvimento do *framework* foram encontradas, entretanto, limitações na maneira como o Docker lida com o endereçamento dos nós que fazem parte do *cluster* e da rede *overlay*. Foi observado que durante o processo de criação da rede cada nó fica responsável por divulgar o seu próprio endereço de rede. O problema é que nós que estejam em redes com NAT não possuem o endereço correto para divulgar. Isso faz com que o os outros nós tentem utilizar o endereço interno da rede NAT o que causa problemas de conectividade.

Desta maneira mesmo se apenas um nó do *cluster* estiver em uma rede que utiliza NAT isso faz com que haja problemas para a comunicação entre os nós. O primeiro problema identificado estava relacionado com o redirecionamento de portas provido pelo *routing mesh*. Através da utilização de regras do serviço *iptable* do Linux esses problemas puderam ser contornados. Entretanto o problema de conectividade se manteve.

Vamos supor um cenário onde o nó β está atrás de um NAT e possui apenas uma interface de rede com o endereço 192.168.0.100, disponibilizado pelo NAT. Ao ingressar no *cluster* este nó deve escolher uma de suas interfaces para utilizar na comunicação com os outros nós. Neste caso nosso nó β utiliza a única opção disponível e divulga para o *cluster* seu endereço interno. Ao enviar a requisição o NAT faz automaticamente a conversão de endereços.

O nó gerente do *cluster*, ao receber o pedido do nó β, observa o endereço externo e o utiliza para enviar a resposta, entretanto é armazenado o endereço enviado pelo nó, neste caso 192.168.0.100. Neste ponto o nó β recebeu a confirmação de que ele faz parte do *cluster* e de que a conexão foi bem sucedida, além disso o nó gerente do *cluster* reconhece a entrada do nó β. Assim, outros nós do *cluster* podem tentar estabelecer conexão com esse novo nó. Entretanto o endereço armazenado é utilizado para todas as trocas de mensagens que forem iniciadas pelo servidor. Por se tratar de um endereço interno, protegido por NAT, portanto inválido para os outros nós do *cluster*, estas mensagens não chegam ao nó β.

Além disso, como o gerente do *cluster* responde utilizando o endereço externo o nó β não sabe que ocorreu um erro, já que para toda requisição que ele realiza ele recebe uma resposta valida do gerente.

Este problema não impede que sejam instanciados novos contêineres no nó β, entretanto impedem que a rede *overlay* do Docker funcione de maneira adequada. Para

solucionar este problema foi desenvolvida uma nova rede *overlay* como parte do *framework*. Esta rede é apresentada no Capítulo 5.

3.2 SiteWhere

O SiteWhere é uma sistema de *middleware open source* que permite o gerenciamento de dispositivos sensores e atuadores. A partir da sua versão 2.0 o SiteWhere passou a utilizar uma arquitetura modular baseada em microsserviços *stateless* em contêineres. Esses microsserviços oferecem diversas funcionalidades essenciais para aplicações IoT e são preparadas para serem replicadas de maneira independente de acordo com as demandas do sistema.

Para que estes microsserviços possam ser gerenciados o SiteWhere utiliza alguns serviços de base, que são chamados de infraestrutura. Essa infraestrutura é composta pelos seguintes elementos:

(i) Consul, é responsável pelo serviço de descoberta dos microsserviços;

(ii) Zookeeper,permite o gerenciamento centralizado das configurações de cada componente do sistema;

(iii) Kafka, possibilita a troca de mensagens entre os microsserviços.

(iv) MongoDB, armazena todas as informações geradas pelos agentes e todas as requisições dos clientes; e

(v) BrokerMQTT, permite a troca de mensagens entre os agentes e os microserviços responsáveis utilizando o protocolo de aplicação MQTT.

O SiteWhere utiliza diversas abstrações para classificar os agentes e controlar o acesso. Existe a ideia de Cliente (*costumer*) que representa o dono de determinado dispositivo. Vale destacar que o *costumer* é uma abstração utilizada para organizar os dados e não está diretamente relacionado com os Clientes de *software* que utilizam o SiteWhere. Existe ainda o conceito de Área (*area*), que representa a localização de determinado evento. E por últimos temos Ativos (*asset*) que representam entidades do mundo real que estão relacionadas à determinado dispositivo.

Além dessas abstrações o sistema permite a criação de tipos de dispositivos. Esses tipos determinam os atributos, as funções e os metadados de cada dispositivo. Para cada

instancia de um tipo o sistema registra as leituras realizadas, as localizações informadas pelos agentes ou pelos clientes, os alerta gerados pela análise dos dados e estado atual do dispositivo de acordo com uma lista pré determinada no tipo. Além disso, o sistema ainda registra os comandos enviados ao dispositivo e suas respectivas respostas.

3.2.1 Arquitetura

Para a execução do SiteWhere é necessário um *cluster* Docker que já possua os serviços de infraestrutura iniciados e que contenha uma rede *overlay* entre os nós do *cluster* (detalhes da rede *overlay* do docker são apresentados na Seção 3.1.1).

Cada microsserviço que compõe o SiteWhere é disponibilizado em um contêiner Docker. Ao ser iniciado cada contêiner busca na infraestrutura as suas configurações e se registra no serviço de descoberta. A partir deste momento este microserviço pode se conectar aos outros contêineres utilizando o Kafka como canal de comunicação. As mensagens trocadas entre os serviços utilizam gRPC e os dados são representados utilizando o protocolo *Protocol Buffers*.

Os agentes podem trocar mensagens com o SiteWhere utilizando primariamente o MQTT. O conteúdo destas mensagens deve seguir o modelo de dados do protocolo *Protocol Buffers*. Já os clientes deste sistema tem acesso utilizando HTTP através da API RESTful. Para identificar os clientes e garantir os aspectos de segurança, como controle de acesso, o sistema obriga que cada requisição contenha uma assinatura obtida utilizando *JWT*.

Conforme apresentado na Figura 4 além dos protocolos citados, o SiteWhere disponibiliza camadas de compatibilidade para diversos outros protocolos, além de oferecer pontos de extensão/integração com outros serviços.

O SiteWhere trabalha com o conceito de *tenant* onde a mesma instancia do sistema pode atender diferentes atores. Isso permite que o mesmo conjunto de microsserviços seja utilizado como base para diversas aplicações finais ao mesmo tempo. Para que isso seja possível os componentes são instrumentados de maneira a tratar requisições de diversas fontes.

A partir desta funcionalidade surge uma nova classificação dos microsserviços: os globais que gerenciam aspectos independentes do *tenant* utilizado; e os *multitenant* que devem tratar cada requisição de acordo com o *tenant* selecionado.

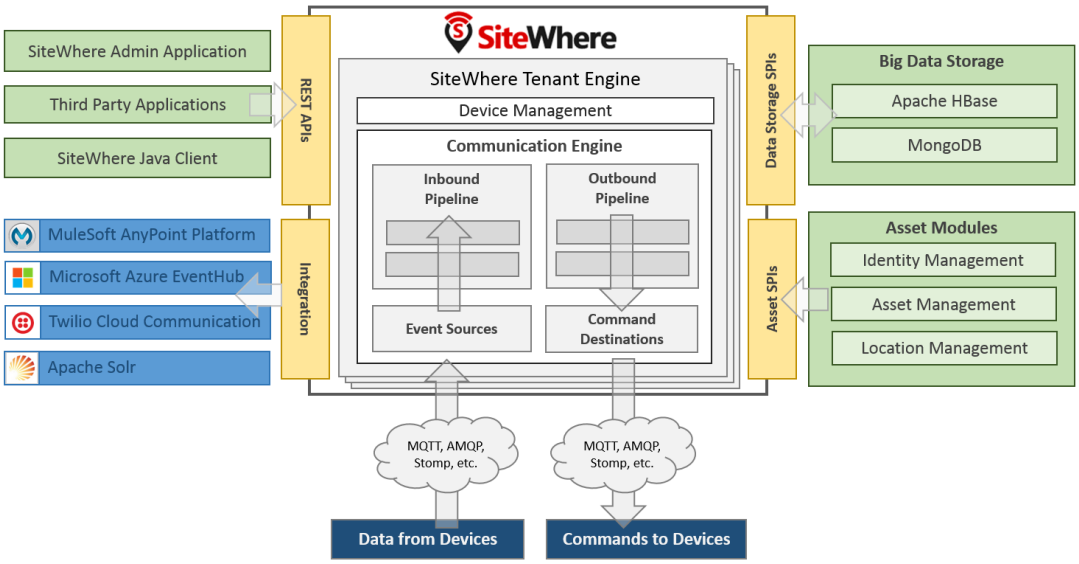


Figura 3 - Arquitetura do SiteWhere
*Figura retirada de [53]

Apesar desta divisão por *tenant* não ser abordada nos testes realizados, o *framework* proposto não interfere nesta funcionalidade.

3.2.2 Serviços

O SiteWhere conta com mais de 15 tipos diferentes de microsserviços (Figura 4). Também permite que o sistema seja executado com todos os serviços ou com um subconjunto, denominado mínimo. Neste trabalho utilizamos uma configuração completa, onde sempre existe pelo menos uma instância de cada serviço sendo executada.

Todos os microsserviços oferecem uma interface gRPC interna utilizada para a comunicação entre os serviços, entretanto, não há a comunicação direta, toda troca de mensagens é realizada de maneira assíncrona e desacoplada utilizando o Kafka como meio de transporte. Além disso, ao ser iniciado o microsserviço não possui uma configuração pré-determinada e não recebe informações de estado. Ou seja, a qualquer momento do ciclo de vida do sistema, se um novo microsserviço for iniciado ele executa o mesmo procedimento.

Dentre os diversos microsserviços iremos destacar aqueles que são responsáveis pela comunicação com os clientes e os agentes. Esta comunicação é feita de maneira assíncrona e desacoplada, entretanto desta vez o meio de comunicação utilizado é o MQTT. Esses microsserviços são:

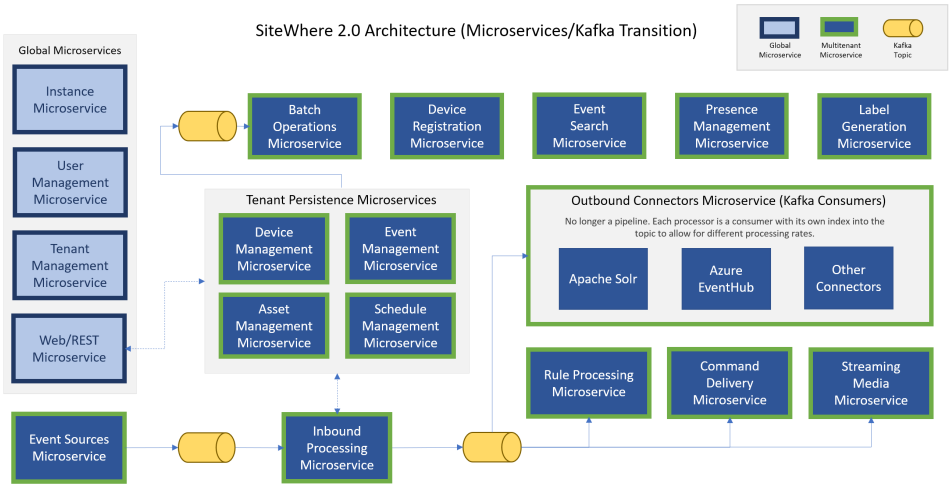


Figura 4 - Serviços do SiteWhere
*Figura retirada de [53]

(i) Web/REST - oferece uma API HTTP REST para que os clientes possam acessar as informações dos agentes. Através desta API também é possível realizar a configuração do sistema.

(ii) Event Management - oferece uma API e gerencia a persistência dos dados relacionados aos eventos/medidas gerados pelos agentes e aos dados gerados pelas requisições dos clientes.

(iii) Command Delivery - converte os comandos do sistema e requisita a execução no dispositivo final.

(iv) Event Sources - é responsável por receber as mensagens dos dispositivos nos mais variados protocolos e converter para o formato padrão interno do SiteWhere.

Os microsserviços *Command Delivery* e *Event Sources* são responsáveis por atender as requisições para os agentes. Para os testes realizados todos os dispositivos trocavam mensagens com o sistema utilizando Protocol Buffers e JSON. Essas mensagens eram transportadas utilizando MQTT e REST, respectivamente, pelos agentes e pelos clientes.

3.2.3 Modificações realizadas no SiteWhere

Para viabilizar todos os serviços oferecidos pelo *framework* apresentado neste trabalho foi necessário realizar modificações no SiteWhere. Serão apresentadas descrições breves, bem como justificativas nos próximos parágrafos.

A primeira modificação altera a forma como a comunicação entre os microserviços trata as mensagens destinadas aos clientes. Por padrão uma nova requisição enviada para o tópico Kafka por um dos microsserviços seria entregue apenas para uma instancia do Command Delivery, que por sua vez a encaminharia para o seu *broker* MQTT. Entretanto, este comportamento adicionava complexidade no encaminhamento de cada mensagem, já que a cada momento o sistema deveria verificar qual *broker* MQTT era utilizado pelo cliente final, só então o Command Delivery seria escolhido e o tópico Kafka determinado. É importante notar que a versão utilizada do Sitewhere originalmente não fazia esta verificação, dado que o sistema considerava que, apesar de poderem existir diversas instancias do Command Delivery ao mesmo tempo, existiria apenas uma instância do *broker* MQTT.

Para resolver este problema sem muito impacto no processamento dos dados o SiteWhere foi modificado para que as mensagens fossem encaminhadas para todos os Commands Deliveries e consequentemente para todos os *brokers* MQTT. Esta modificação foi realizada alterando o comportamento dos tópicos Kafka, fazendo com cada microsserviço pertencesse à um grupo exclusivo.

Além dos microsserviços, os agentes também foram instrumentados para que fosse possível obter todas as informações de operação desejadas. As modificações dos agentes foram realizadas na biblioteca base do SiteWhere, sendo assim não foram necessárias alterações no código específico de cada agente. Essas modificações acrescentam duas novas funções customizadas. Essas funções serão detalhadas na Seção 4.3 do Capítulo 4.

4 O *FRAMEWORK* PROPOSTO

Este capítulo detalha o *framework* proposto. Conforme apresentado na Introdução, o *framework* foi implementado com o objetivo de integrar os mecanismos necessários para que fosse possível experimentar técnicas de *off-loading*, alocação de módulos de software e serviços em recursos de processamento distribuídos em *edge*, *fog* e *cloud*, para aplicações IoT, e avliar se estas técnicas poderiam oferecer escalabilidade e desempenho aceitáveis para estas aplicações.

Na Seção 4.1 é apresentada uma visão geral, onde são discutidos os principais componentes do *framework* e seus papéis. Já na Seção 4.2 é detalhada a arquitetura e como esses componentes são utilizados em conjunto com o SiteWhere. A Seção 4.3 apresenta as APIs oferecidas pelo *framework* e como elas pode ser utilizada para acrescentar novos algoritmos de orquestração e como os clientes podem acelerar as atividades de orquestração. A Seção 4.4 apresenta as etapas de inicialização do *framework*. Na Seção 4.5 é apresentado o ciclo de operação do *framework*. E na Seção 4.6 apresentamos alguns detalhes de implementação, como, por exemplo, a estrutura dos dados armazenados no banco de dados e algumas soluções tilizadas no desenvolvimento. Por fim, é apresentado como o *framework* realiza a orquestração dos agentes e dos clientes na Seção 4.7.

4.1 Visão geral

Inicialmente, cabe definir a nomenclatura utilizada. Cada hospedeiro que faz parte do *framework* é chamado de **nó**. Cada nó pode exercer a função de Gerente (*manager*) ou Trabalhador (*worker*), esta nomenclatura foi baseada naquela utilizada pelo Docker. Além disso, sensores e atuadores conectados ao SiteWhere são denominados *agentes*. Já os usuários, que estão interessados nestes agentes recebem o nome de *clientes*. Agentes e clientes não fazem parte do *framework*.

Outro ponto importante é que a solução foi proposta na forma de um *framework* e, assim, define uma estrutura de classes, interfaces e um modelo de interação entre elas [54]. A especialização dessas classes e interfaces para atender um domínio específico é de responsabilidade do desenvolvedor do sistema final. Entretanto, para realizar uma avaliação ampla do *framework*, uma implementação de referência foi criada utilizando o SiteWhere como sistema *middleware*, integrada ao *framework* proposto através das

interfaces oferecidas, servindo também como prova de conceito do uso das interfaces do *framework*.

Na mesma, a interface do *framework* permite a integração de algoritmos de alocação de recursos, para apoiar a orquestração de serviços. Uma lista de algoritmos disponíveis na literatura foi integrada para a realização de testes. Esses algoritmos são apresentados no Capítulo 6 e os testes realizados são apresentados no Capítulo 7.

O *framework* proposto tem como funcionalidades principais:

(i) coletar as métricas dos nós de processamento

(ii) prover dados necessários para os algoritmos de alocação

(iii) executar os algoritmos de alocação

(iv) realizar a orquestração dos clientes

Dada a lista de funcionalidade do *framework* podemos descrever como cada um dos componentes foi criado para solucionar um desses desafios. Para auxiliar na explicação iremos definir, ao longo dessa seção, um diagrama dos componentes lógicos do *framework*. A arquitetura final utilizada na solução será apresentada e discutida na Seção 4.2.

O primeiro componente dessa arquitetura é o Controlador. Esse componente será a base do *framework* e é responsável por controlar os outros componentes e serve como intermediário para a troca de informações.O ciclo de operação do *framework* é gerenciado por este componente, ou seja, é responsabilidade do Controlador manter o estado atual do sistema e realizar as operações necessárias para que o ciclo seja cumprido. A medida que os componentes lógicos são apresentados, iremos construir uma versão simplificada do ciclo de operação. Um versão completa será apresentada na Seção 4.5.

O próximo componente apresentado é responsável pela coleta das métricas, denominado Coletor de Métricas. A princípio, este componente deve ser capaz de coletar todas as informações necessárias sobre o estado de cada um dos nós de processamento e envia-las ao Controlador. Conforme demonstrado na Figura 5 espera-se que hajam diversas instâncias desse Coletor de Métricas sendo executadas simultaneamente, já que cada nó de processamento irá executar uma cópia desse componente. Com a inclusão desse módulo o Controlador passa a ser capaz de obter o estado atual de todos os nós que estão utilizando o *framework*. As métricas coletadas na implementação de referência incluem

informações sobre a memória do sistema e sobre o uso de CPU nos nós de processamento. Conforme comentado, por se tratar de um *framework*, esse comportamento poderia ser alterado e especializado para o domínio em que a aplicação final será utilizada.

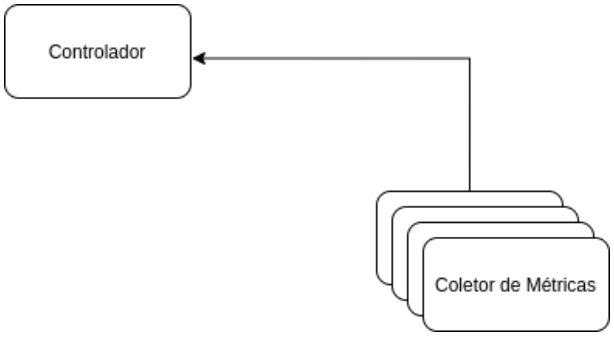


Figura 5 - Componentes do Framework - Coletor de Métricas

Para que o Controlador seja capaz de manter um histórico dessas métricas foi acrescentado um banco de dados na arquitetura, conforme a Figura 6. Mais detalhes deste componente serão discutidos na Seção 4.6. Com a adição desse banco de dados uma versão simplificada do ciclo de operação (apresentado na Seção 4.5) pode ser iniciado. Este ciclo é composto inicialmente de 3 etapas: coletar as métricas disponíveis; gerar o estado atual do sistema; e armazena-las no banco de dados.

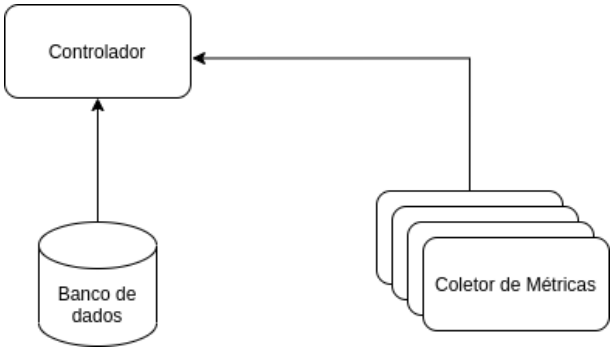


Figura 6 - Componentes do Framework - Banco de dados

A Figura 7 apresenta o próximo componente: API dos algoritmos de alocação. Esse componente é responsável por prover a API necessária para a execução dos algoritmos de alocação. Esses algoritmos são utilizados para otimizar a alocação dos clientes/serviços que estejam conectados ao sistema de acordo com as métricas coletadas. Com a adição desse componente o ciclo de operação ganha mais uma etapa: executar os algoritmos de alocação. Essa etapa é realizada após o armazenamento do estado atual do sistema no banco de dados. O estado do sistema é utilizado como entrada para os algoritmos de alocação de recursos.

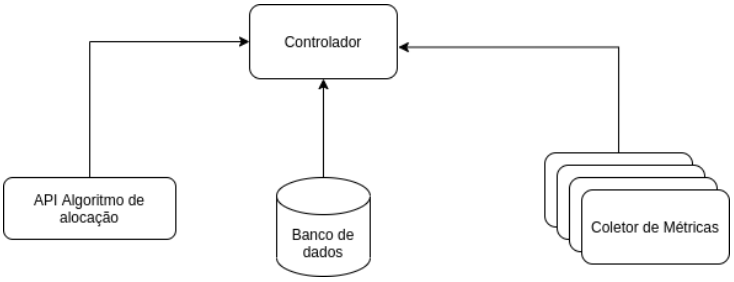


Figura 7 - Componentes do Framework - API Algoritmo de alocação

O próximo componente, apresentado na Figura 8, é a API do *middleware*. Na implementação de referência , este componente é responsável por realizar toda a comunicação com SiteWhere. Através dele é possível solicitar a execução de funções nos Agentes, além de ser possível realizar alterações nas configurações do próprio SiteWhere.

Utilizando as funções dos Agentes, o Coletor de Métricas é capaz de obter a latência média entre os nós de processamento e os agentes. Para tal, o Coletor de Métricas em cada nó de processamento inicia um servidor de *ping* e utilizando as funções dos agentes ele solicita que o agente realize testes de *ping* utilizando o servidor criado. Com isso, ao gerar a visão consolidada do sistema, o Controlador terá não só as métricas dos nós de processamento, mas também todas as informações de latência entre cada agente e nó de processamento. Sendo esse, mais um parâmetro disponível para os algoritmos de alocação.

Ainda utilizando esse componente, o Controlador é capaz de aplicar o resultado dos algoritmos de alocação. Ou seja, de acordo com o resultado obtido pelos algoritmos de alocação o Controlador é capaz de realizar a orquestração dos agentes. Essa orquestração é feita através de alterações nas configurações do *middleware* e da utilização das funções dos agentes.

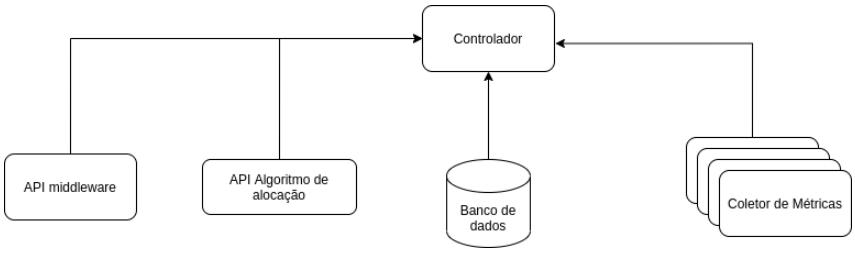


Figura 8 - Componentes do Framework - API Middleware

Com esses componentes apresentados o *framework* é capaz de realizar todas as funcionalidades descritas no começo desta seção e o ciclo de operação seria composto por:

1. coletar as métricas disponíveis

2. gerar uma visão consolidada do sistema

3. armazena-las no banco de dados

4. executar os algoritmos de alocação

5. orquestrar os clientes/agentes

Além disso,mais um componente foi acrescentado ao *framework* (Figura 9). A API de Orquestração pode ser utilizada por agentes/clientes que queiram obter diretamente a melhor alocação disponível. Além disso, esse componente também é utilizado para orquestrar o próprio *framework* e seus microsserviços.

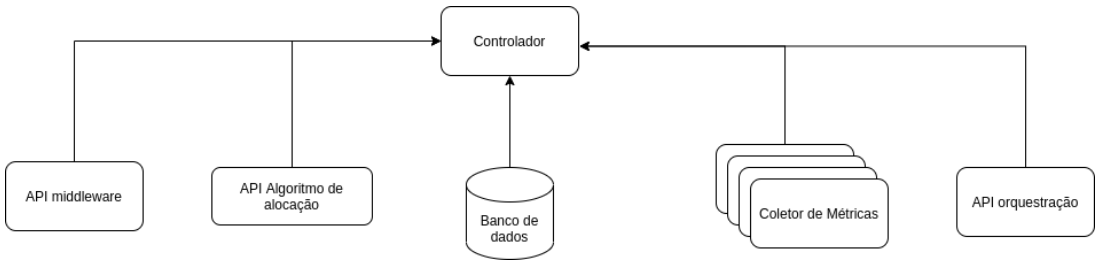


Figura 9 - Componentes do Framework - API de Orquestração

Finalmente, com a lista completa dos componentes o *framework* é capaz de gerenciar os serviços disponíveis providos pelo SiteWhere, bem como obter informações do estado do sistema e dos agentes, a fim de monitorar os clientes/agentes e os nós de processamento, para determinar quais os nós de processamento devem ser utilizados.

Conforme apresentado, o uso deste *framework* pode ser transparente para os clientes. Entretanto, é possível que os clientes utilizem a API de orquestração do *framework* diretamente, sendo então capazes de obter todas a informações geradas pelo sistema. Isto permite que a alocação realizada pelo *framework* seja consultada diretamente pelos usuários, sem a necessidade de transmissão de mensagens pelo *middleware*.

A Figura 10 apresenta o diagrama de componentes do *framework*. De acordo com o apresentado nessa seção temos os 6 componentes do *framework*. Além disso, são apresentados ainda o SiteWhere, o Docker e o banco de dados (Consul).

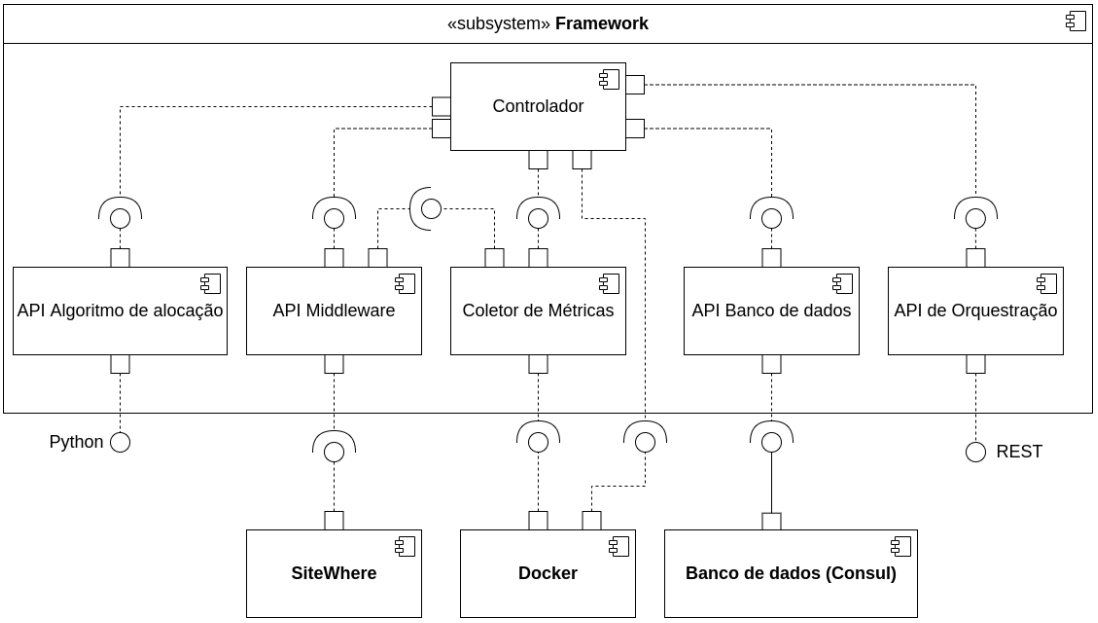


Figura 10 - Diagrama de componentes do Framework

4.2 Arquitetura

A base de todos os componentes é o Docker, desde a infraestrutura do SiteWhere, até a infraestrutura do *framework*. O Docker permite que todos esses elementos sejam desenvolvidos e implantados sem que as características dos hospedeiros sejam conhecidas. Ou seja, através da utilização do Docker foi possível criar uma versão única dos microsserviços, que pode ser utilizada tanto na máquina da nuvem quanto na maquina da borda. Apesar das características do hospedeiro se tornarem transparente para os microsserviços, o Docker oferece APIs para que essas características possam ser consultadas. O Coletor de Métricas utiliza essas APIs para obter as informações de CPU e memória da máquina hospedeira. Além disso, para que seja possível medir a latência esse componente utiliza outra API do Docker para obter uma porta de rede na máquina hospedeira. É importante que essa porta seja realmente na máquina hospedeira, para evitar que a latência da rede interna do Docker não esteja presente nas medidas realizadas pelo Coletor de Métricas.

Uma visão geral da arquiteutra é apresentada na Figura 11. Temos representado o *cluster* Docker, do qual todos os nós de processamento fazem parte. Cada nó de processamento é representado de acordo com o seu papel no *framework*, podendo ser um Gerente ou um Trabalhador. Além disso, a Figura 11 ainda apresenta os usuários e os agentes (triângulo roxo).

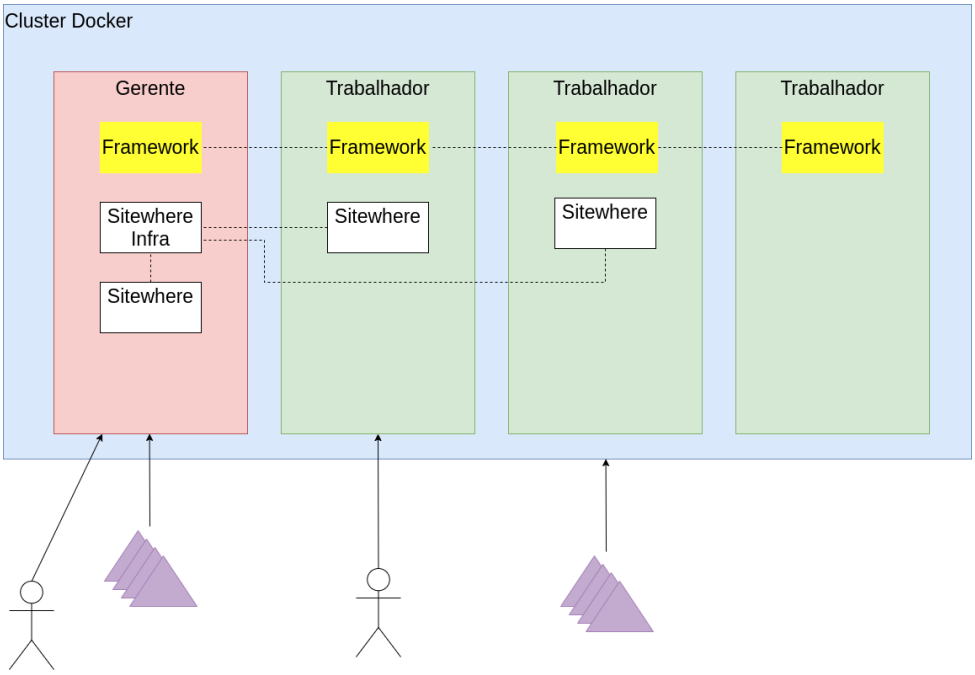


Figura 11 - Arquitetura de alto nível

Na mesma figura, podemos localizar três grupos de microsserviços: (i) o *framework*, (ii) o SiteWhere Infra e (iii) o SiteWhere.

Conforme discutido na Seção 3.2.2 o grupo SiteWhere Infra reúne os serviços essenciais ao SiteWhere, que não estão diretamente ligados ao tratamento de requisições dos usuários. Por exemplo, temos nesse grupo todos os microserviços executando serviços externos (Consul, Zookeeper, MongoDB, etc.) e todos os serviços-base do SiteWhere como o processamento de regras, gerenciamento de usuários, gerenciamento de instâncias, etc. Todos esses serviços devem estar executando a todo tempo e devem estar acessíveis para todos os outros nós do *cluster*. A fim de simplificar o gerenciamento dos recursos do *cluster* (especialmente disco), durante os testes esses serviços foram executados apenas nos nós Gerentes, visto que estas instâncias eram persistidas entre testes. Entretanto, nada impede que esses serviços sejam distribuídos e replicados. É importante considerar que cada um desses serviços deverá ser configurado individualmente para tal.

Os microsserviços do grupo SiteWhere, na Figura 11, fazem referência à todos os serviços diretamente relacionados às requisições dos usuários. Esses serviços são: *Command Delivery*, *Event Source*, *Web Rest*,*Event Management* e os microsserviços necessários para os protocolos suportados (para os testes foi utilizado um *broker* MQTT). Os microsserviços que compõem o SiteWhere já estão preparados para serem distribuídos e replicados, logo não foi necessária nenhuma configuração específica. O mesmo pode ser

dito para o *broker* MQTT. Neste caso, a arquitetura foi configurada para que cada nó possuísse seu *broker* individual e as mensagens não eram replicadas.

Por último temos o grupo de microserviços do *framework*, que pode ser subdividido em três microsserviços diferentes: o Gerente, o Trabalhador e o de Visualização. O serviço de visualização foi desenvolvido como um utilitário para auxiliar a avaliação dos testes realizados, e não será explorado neste texto. A Figura 12 apresenta uma representação detalhada, entre a visão da Figura 11 e os componentes lógicos apresentados na Seção 4.1, especialmente na Figura 9.

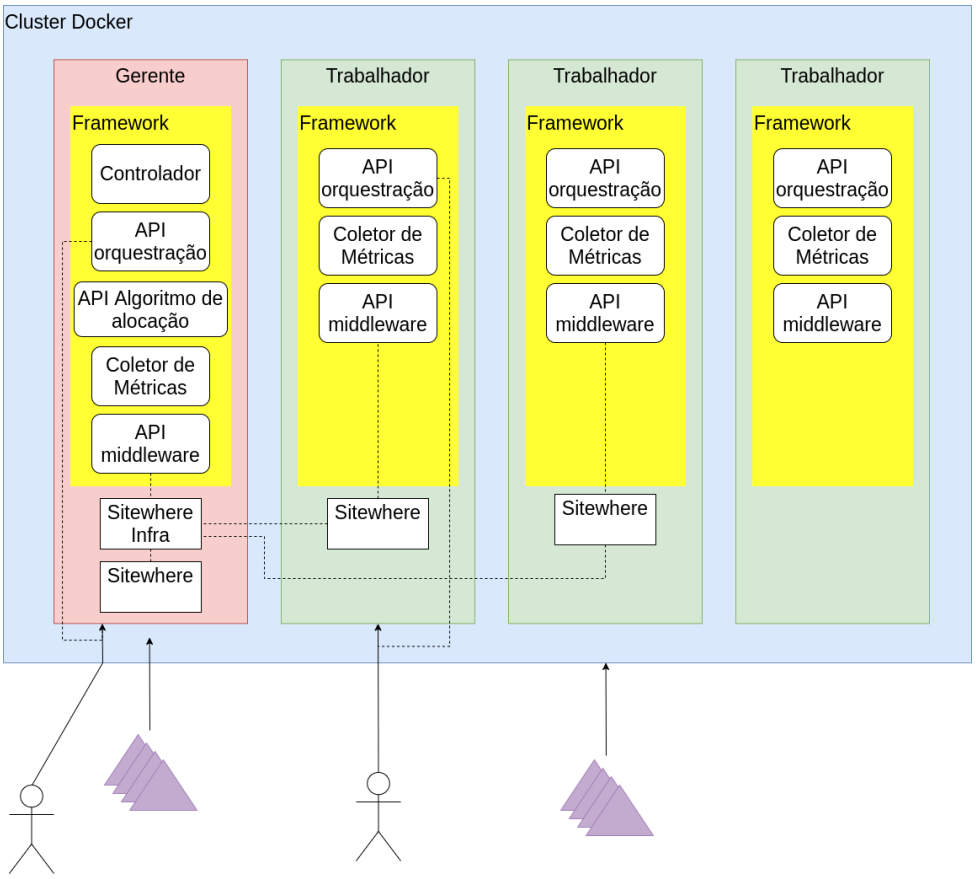


Figura 12 - Arquitetura de alto nível com componentes lógicos

Conforme comentado anteriormente, cada nó do *framework* pode ser um Gerente ou um Trabalhador. Os nós Trabalhadores tem três funcionalidades principais:

(i) Obter os contadores locais para informar suas métricas. Tanto as informações referentes a memória e CPU do próprio nó, quanto informações de latência para os clientes/agentes.

(ii) Oferecer a API de orquestração para os clientes que queiram consumir diretamente os serviços de orquestração, sem a mediação do *middleware*.

(iii) Oferecer a API do *middleware*. Essa API será utilizada para controlar o *middleware* e efetivamente realizar a orquestração. Na implementação de referência esse *middleware* é o SiteWhere.

Já os nós Gerentes possuem todas as funcionalidades dos nós Trabalhadores acrescido de:

(i) Controlar o ciclo de operação.

(ii) Executar os algoritmos de alocação, através da API.

(iii) Controlar quais serviços serão executados em cada nó de processamento.

Apesar de compartilharem diversos componentes e funcionalidades, foi criado um microsserviço diferente para cada um dos papeis que um nó pode assumir. Esses microsserviços foram criados usando imagens Docker.

A Figura 13 apresenta uma visão geral, em camadas, com os principais elementos da estrutura do *framework*.

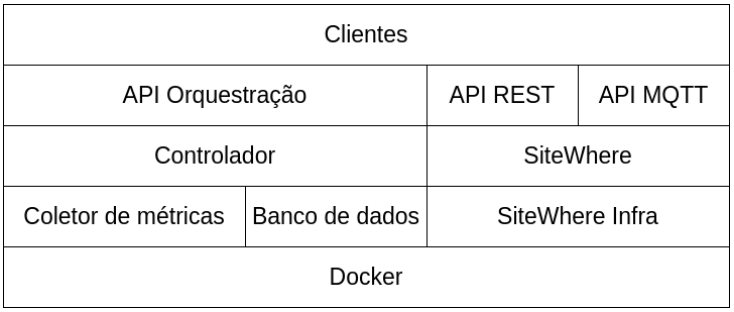


Figura 13 - Principais camadas do framework

São destacadas na Figura 13 as 3 APIs que um cliente pode consumir:

1. API de Orquestração do *framework*
2. API Rest do SiteWhere
3. API MQTT do SiteWhere

A API de Orquestração do *framework* é oferecida por HTTP REST e permite que clientes consultem o *framework* diretamente. As APIs do SiteWhere foram apresentadas na Seção 3.2.2, e forma integradas ao *framework* sem modificações.A API que permite a integração dos algoritmos de alocação, apresentada adiante, não está representada na figura, pois não deve ser consumida por clientes.

4.3 *Application Program Interface*

Nesta seção iremos apresentar as principais APIs desenvolvidas permitir a configuração e acesso aos serviços do *framework*:

(i) **API de Configuração** - quais funções customizadas do SiteWhere devem ser criadas para que a orquestração seja realizada.

(ii) **API de Algoritmos** - qual a interface deve ser implementada pelos algoritmos de alocação para que eles sejam integrados ao *framework*

(iii) **API de Orquestração** - API HTTP REST que permite que clientes interajam diretamente com o *framework*.

API de Configuração. Para que Controlador possa aplicar o resultado do algoritmo de alocação e então realizar a orquestração dos clientes/agentes, é necessário que exista uma API para a realização das configurações necessárias. Essa API foi criada utilizando as funções customizadas do SiteWhere. Para isso foram criadas duas funções conforme apresentadas pelas interfaces descritas em Java no Código 1 e no Código 2.

```
1 void changeHost(
2     String hostA,
3     String hostB,
4     String hostC,
5     IDeviceEventOriginator originator)
6     throws SiteWhereAgentException;
```

Código 1 - Função customizada para alterar host

```
1 void ping(
2     String hostA,
3     String hostB,
4     String hostC,
5     int port,
6     String token,
7     IDeviceEventOriginator originator)
8     throws SiteWhereAgentException;
```

Código 2 - Função customizada para realizar medidas de latência

Ambas as funções recebem três Strings diferentes, *HostA*, *HostB* e *HostC*, que representam os três endereços de IP conhecidos pelo *framework*: local, público e virtual

(da rede Docker). O IP da rede local e o IP da rede virtual do Docker, são obtidos através de uma API oferecida pelo próprio Docker. Para obter o IP público o *framework* utiliza um serviço externo definido através de configuração, como exemplo podemos citar os sites `https://api.myip.com/ehttps://api.ipify.org/?format=json`

Em todas as chamadas o *framework* fornece essas três possibilidades para que o cliente possa escolher a melhor alternativa. Além disso, conforme determinado pela API do SiteWhere, as duas funções recebem também um *IDeviceEventOriginator*, que será utilizado para obter metadados da requisição e para enviar a resposta da chamada de função.

A interface apresentada no Código 2 conta ainda com o número da porta que foi utilizada na criação do servidor de ping e com um token para que o servidor possa confirmar a autenticidade do pedido.

Com apenas essas duas funções customizadas no SiteWhere o *framework* é capaz de realizar a orquestração. O Coletor de Métricas pode iniciar um servidor de *ping* e em seguida utilizar a API de *ping* para solicitar testes de latência para os clientes. Com essa e outras métricas que serão coletadas o Controlador pode executar os algoritmos de alocação e em seguida aplicar o resultado utilizando a api *changeHost*.

API de Algoritmo. Foi definida uma API, apresentada no Código 3, foi criada para que novos algoritmos de alocação pudessem ser adicionados ao *framework* e para que o Controlador não precisasse conhecer detalhes de implementação de cada um deles.

```
1  #Graph G - grafo onde cada nó é um nó da rede, e cada aresta é a latência entre os nós
2  #List A - Todos os agentes/clientes
3  #List F - Todos os nós considerados fog
4  #List C - Todos os nós considerados cloud
5  #Number delay - Restrição de Delay
6  #Dictionary cap - Capacidade/custo de cada nó. Quota/consumo de um servidor/cliente
7  def algoritmo(G,A,F,C,delay,cap):
8      V=[]  #lista de nós utilizados para processar
9      rC={} #mapa onde a chave é o nó de processamento(cloud ou fog) e
10           #o valor são os clientes que serão processados por esse nó

11     #implemetação do algoritmo

12     return V,rC
```

Código 3 - Interface de Programação (API) para os algoritmos

Uma vez que o framework foi implementado em Python 3, para que um novo algoritmo possa ser adicionado ao *framework* é também necessário que ele seja implementado em Python 3, e que possua um método com a mesma assinatura que foi definida na interface apresentada no Código 3.

Essa interface define a lista de parâmetros e o retorno esperado pelo *framework*. Os parâmetros e seus respectivos tipos são:

1. **G** - Graph - Representa o estado atual do sistema e é implementado como um grafo bipartido não direcional onde cada vértice representa um nó de processamento ou um cliente/agente. As arestas desse grafo representam a latência entre os dois vértices.
2. **A** - List - nós do grafo que representam os agentes/clientes.
3. **F** - List - nós do grafo que representam os nós de processamento da Fog/Edge.
4. **C** - List - nós do grafo que representam os nós de processamento da Cloud.
5. **delay** - Number - O limite máximo aceitável para o delay. Esse valor é configurável.
6. **cap** - Dictionary - Contem todos os vértices e seus pesos. O peso é um valor numérico que representa o custo computacional desse vértice no sistema. Esse custo será positivo para os nós de processamento, ou seja, eles "fornecem" poder computacional para o sistema. E de forma análoga o clientes/agentes terão um custo negativo, ou seja, eles "consomem" poder computacional do sistema.

É trivial notar que o primeiro parâmetro(G) por construção é de fato um grafo bipartido, já que sempre será possível dividir os vértices em dois grupos: nós de processamento e clientes/agentes. Como o *framework* não testa as latências entre os nós de processamento não haverá arestas entre os vértices do primeiro grupo. De forma similar o *framework* não realiza testes de latência entre clientes/agentes e portanto não haverá nenhuma aresta entre os vértices do segundo grupo. A Figura 14 exibe um exemplo deste grafo. É importante notar que é possível que um determinado nó de processamento não seja alcançável por nenhum cliente/agente e portanto o vértice que representa este nó, terá grau 0, esse comportamento é esperado e deve ser levando em consideração na implementação dos algoritmos de alocação. Vale destacar também que todo vértice que representa um cliente/agente terá necessariamente $grau \geqslant 1$ já que um $grau = 0$ representaria um cliente/agente que não está conectado ao sistema.

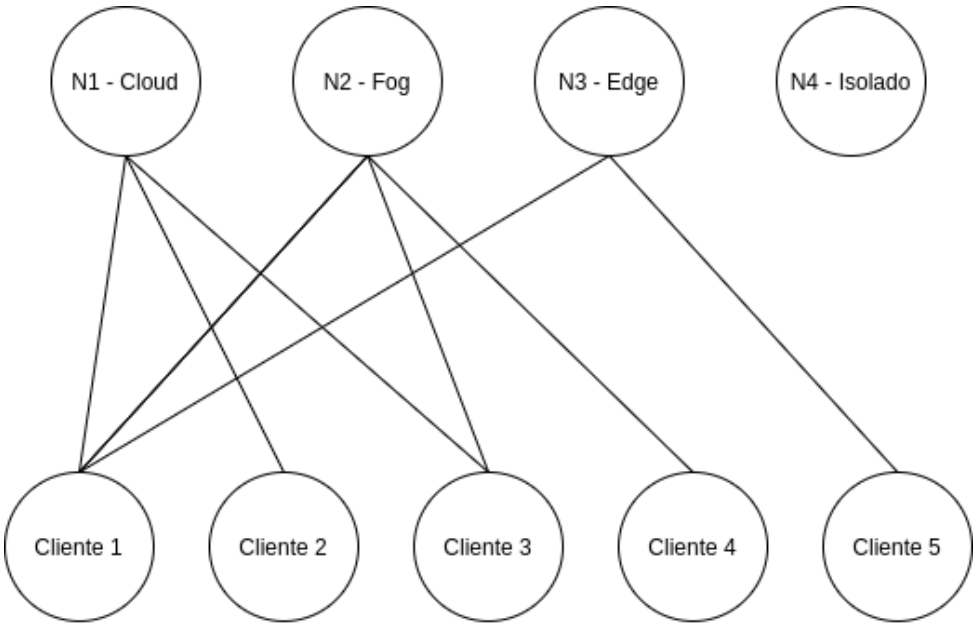


Figura 14 - Grafo que representa o estado do sistema

O primeiro parâmetro(G) é o único que possui como tipo uma classe definida pelo *framework*. Essa definição é apresentada no Código 4 e é construída com base no *Dictionary* do Python.

```
1  class Graph:
2      def __init__(self):
3          self.V={}
4      def link(self,v,u,d):
5          if v not in self.V:
6              self.V[v]={}
7          if u not in self.V:
8              self.V[u]={}
9          self.V[v][u]=d
10         self.V[u][v]=d
```

Código 4 - Classe Graph definida pelo framework

API de Orquestração. O Código 5 apresenta a API de Orquestração, oferecida por HTTP REST. Com ela um cliente pode acessar diretamente o estado atual do sistema e forçar que um novo teste de latência seja realizado para ele. Com isso são definidos 3 *endpoints* diferentes.

O primeiro definido na linha 2 permite que um cliente liste todos os nós de processamento. Com isso um cliente pode se conectar diretamente a um nó que esteja disponível.

O segundo *endpoint* disponibilizado pela linha 5 permite que um cliente acesse a alocação mais recente realizada para ele, mesmo sem realizar uma requisição ao SiteWhere.

```
1 @app.route("/getNodes")
2 def getNodes():
3     return obtemListaDeNosNoBanco()

4 @app.route("/getHost/<id>")
5 def getHost(id):
6     return alocacaoParaCliente(id)


7 @app.route("/forcePing")
8 def forcePing(id):
9     return realizarTesteDeLatenciaParaCliente(id)
```

Código 5 - Api de orquestração

Essa API permite que o cliente já realize a conexão com o nó de processamento.

Por último o cliente tem a opção de utilizar a API apresentada na linha 8 para forçar que um teste de latência seja realizado, ignorando o intervalo da coleta de métricas.

4.4 Inicialização

Ao longo dessa seção é apresentado a etapa de inicialização do *framework*. Para que o *framework* seja inicializado é necessário que o *cluster* Docker esteja previamente criado. Não é necessária nenhuma configuração especial para este *cluster*. O diagrama de atividades da Figura 15 apresenta, do ponto de vista do *framework*, as ações necessárias para que o ciclo de operação possa ser iniciado e executado. É importante notar que esse ciclo do *framework* é realizado em conjunto por todos os nós de processamento. Essa figura pode ser dividida em duas etapas: (i) inicialização, detalhada nessa seção; (ii) ciclo de operação, detalhada na Seção 4.5.

A primeira etapa de cada nó de processamento é identificar o seu papel no *framework*, ou seja, se ele é um nó Gerente ou um nó Trabalhador. Caso seja um nó Gerente, ele deve ainda executar um teste para saber se é o nó Gerente escolhido (NGE) para inicializar o sistema. Essa escolha é feita através de um algoritmo distribuído que deve ser executado por todos os nós Gerentes. Apenas um nó será o NGE, entretanto caso o NGE fique indisponível outro nó será selecionado, se tornando o novo NGE.

O NGE então irá inicializar toda a infraestrutura necessária para o *framework* e para o SiteWhere. Só então todos os nós de processamento salvam seus metadados no banco de dados, para que essas informações estejam disponíveis para os algoritmos de

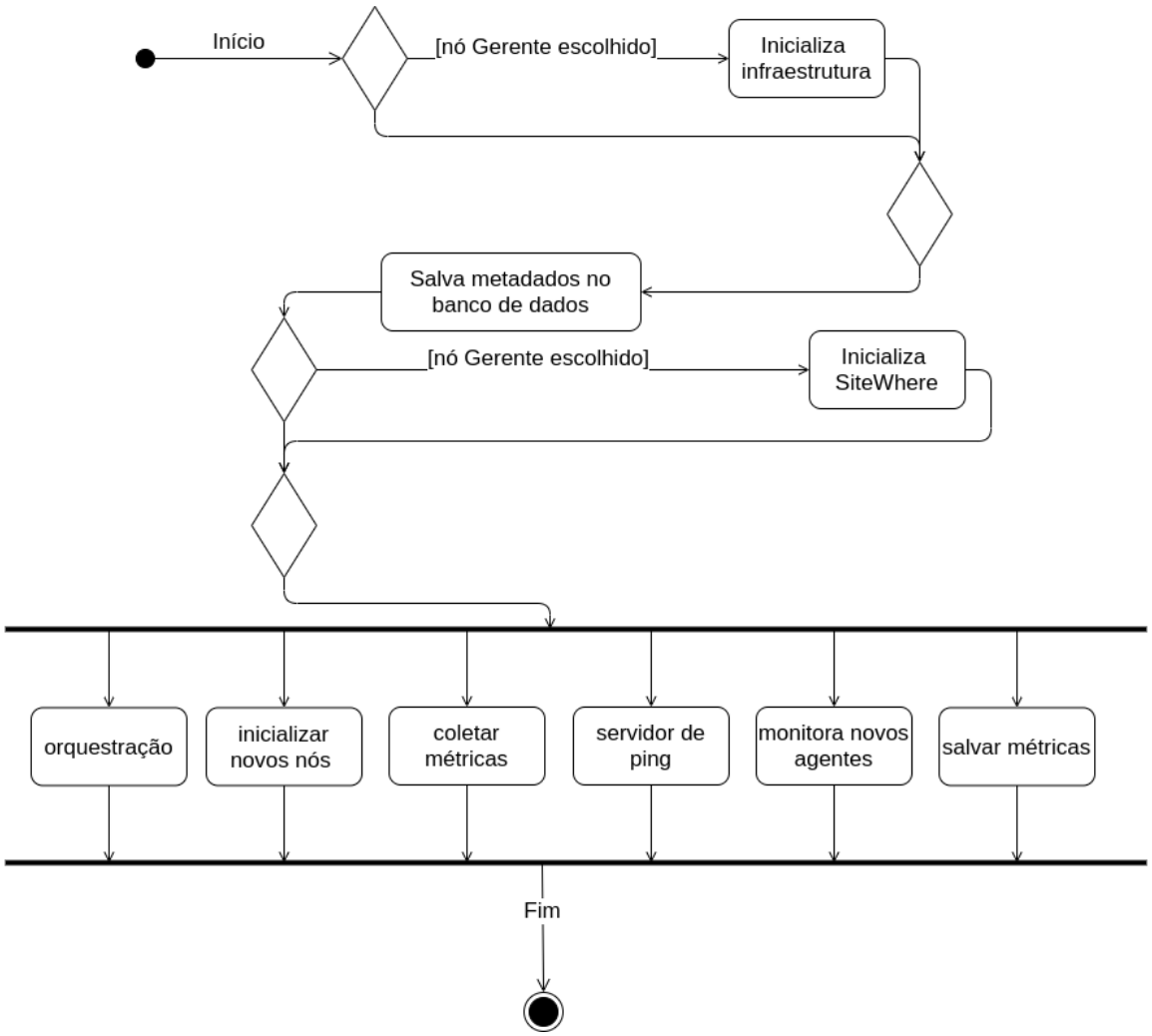


Figura 15 - Diagrama de atividades - Ciclo de operação do framework

alocação. A partir do momento que os nós se salvam seus metadados o NGE inicializa o SiteWhere. Depois da inicialização do SiteWhere o *framework* inicia 6 atividades paralelas e aguarda a conclusão de todas para ser finalizado.

4.5 Ciclo de Operação

Uma vez carregado e configurado, o *framework* inicia um ciclo de operação que é executado continuamente por 6 tarefas paralelas. Essas tarefas são:

1. Coleta de métricas dos agentes (Figura 16)
2. Servidor de ping (Figura 18)
3. Inicialização dos nós (Figura 17)
4. Monitoria dos novos agentes (Figura 19)

5. Monitoria das métricas do nó de processamento (Figura 20)

6. Orquestração (Figura 21)

A Figura 16 apresenta a primeira atividade: coletar métricas. Essa tarefa é executada de forma periódica pelo NGE e consiste em listar todos os agentes/clientes conectados no sistema e então solicitar um teste de latência para cada um eles utilizando a API do SiteWhere.

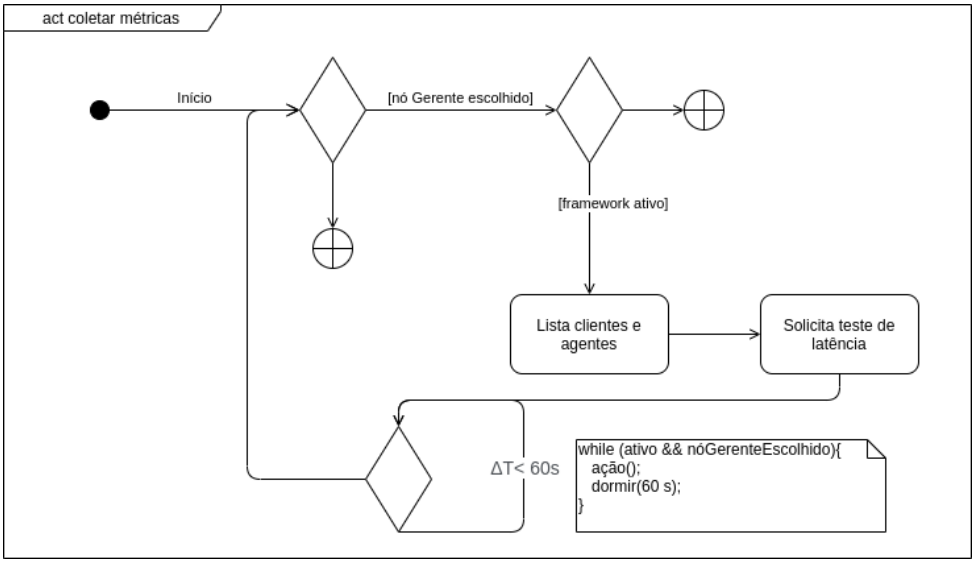


Figura 16 - Diagrama de atividades - coleta de métricas

A Figura 17 apresenta a criação e execução do servidor de ping utilizado para os testes de latência.

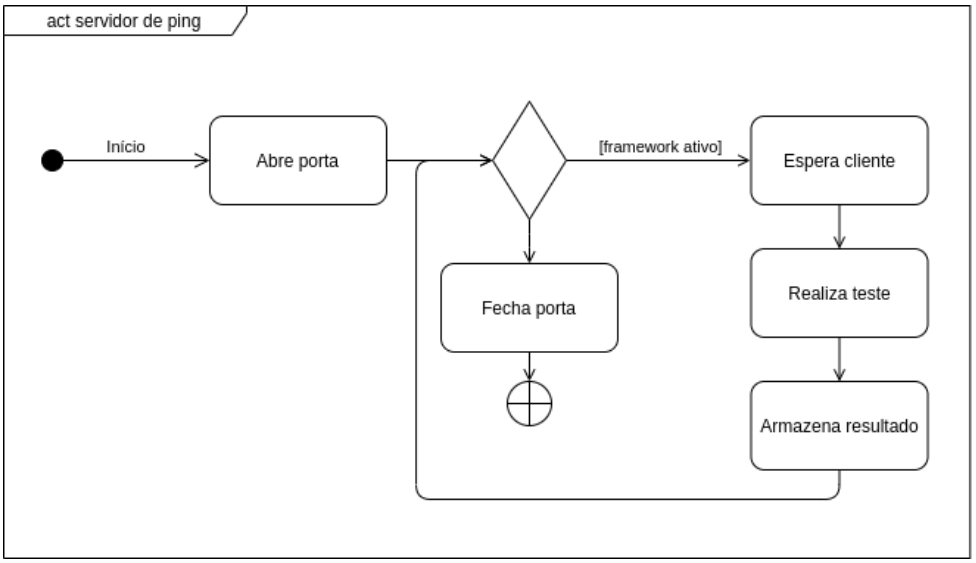


Figura 17 - Diagrama de atividades - servidor de ping

A Figura 18 apresenta a próxima atividade: inicializar os novos nós. De forma similar a coleta de métricas, essa tarefa é executada periodicamente pelo NGE e tem como objetivo inicializar os serviços necessários nos nós de processamento. Para tal, o NGE lista todos os nós de processamento registrados no banco de dados e então inicializa todos os serviços necessários para os nós novos utilizando a API do Docker.

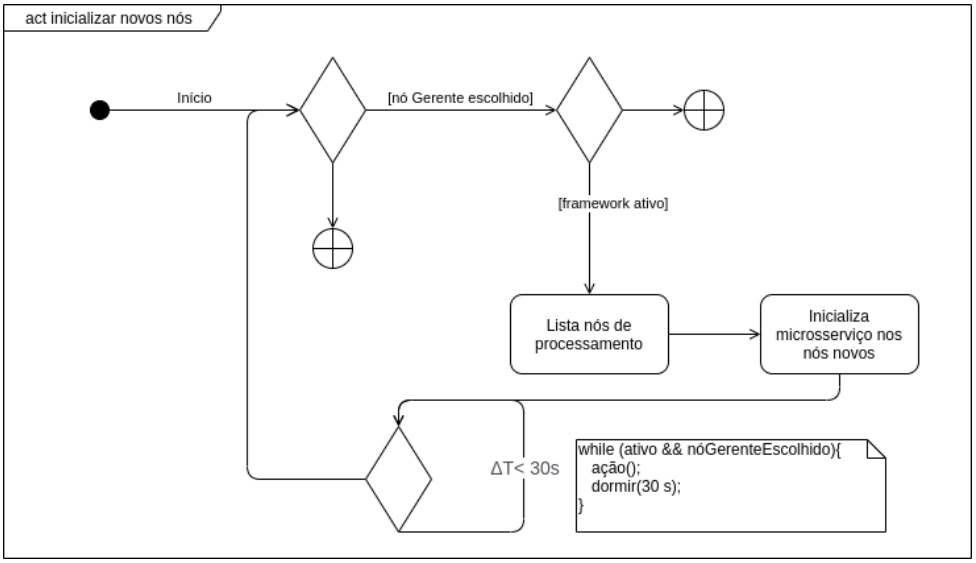


Figura 18 - Diagrama de atividades - inicialização dos nós

Já a Figura 19 apresenta como o *framework* é capaz de monitorar os agentes do SiteWhere, utilizando as mensagens geradas pelo middleware em um canal MQTT. Para isso o *framework* se inscreve em um canal MQTT especificado pelo SiteWhere e então passa a receber todas as mensagens. É necessário que o *framework* de decodifique essas mensagens e de acordo com o conteúdo delas novos agentes são cadastrados no banco de dados do *framework*.

Além dessas atividades cada nó do *framework* ainda registra periódicamente suas próprias métricas, como uso de CPU e memória, conforme apresentado na Figura 20.

Por último a atividade mais importante: a orquestração. A Figura 21 apresenta todos as etapas associadas à essa tarefa. Como podemos observar a orquestração começa com a obtenção da informações de operação que foram salvas no banco de dados e então essas informações são preparadas de acordo com a estrutura apresentada na Seção 4.3. Uma vez que os parâmetros estão prontos o *framework* então carrega o algoritmo de alocação escolhido e o executa. O resultado desse algoritmo é persistido e então aplicado pelo *framework*. Primeiro os clientes que utilizam diretamente a API do *framework* tem a reconexão dos serviços para as novas instâncias e então nos agentes que estão conectados

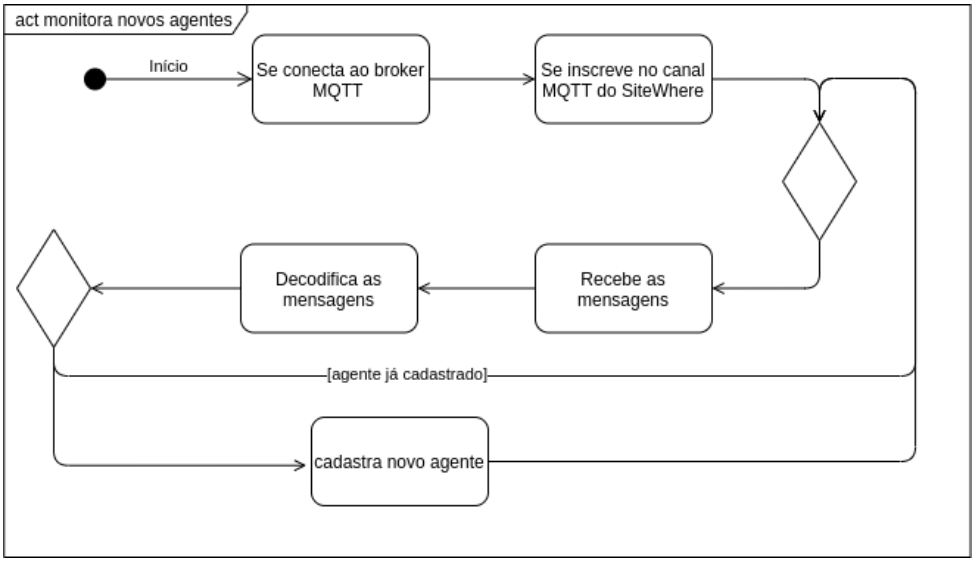


Figura 19 - Diagrama de atividades - monitorando novos agentes

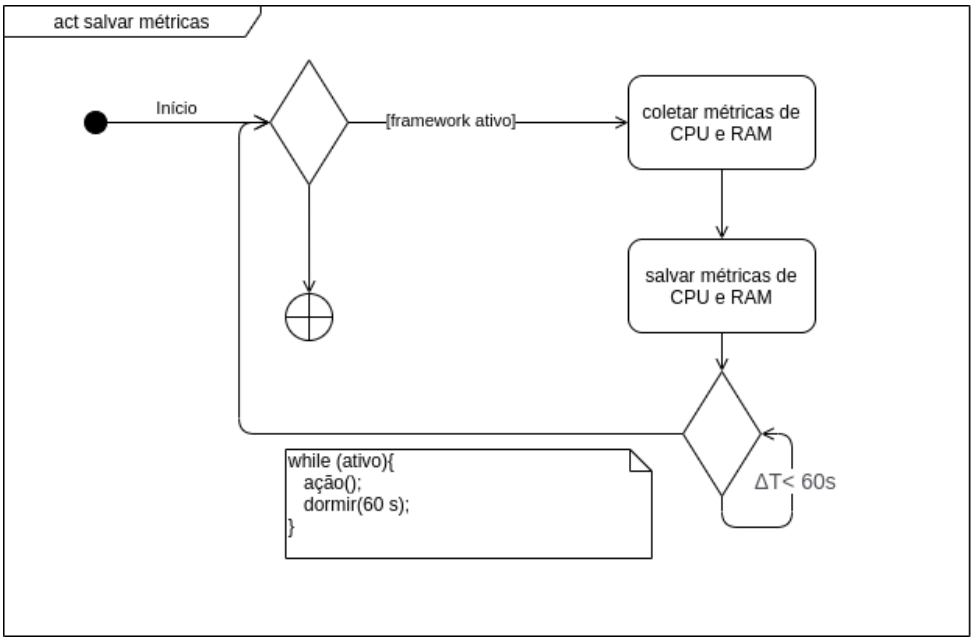


Figura 20 - Diagrama de atividades - salvar métricas do nó de processamento

ao SiteWhere são também reconectados a outras instâncias. Por último são salvas as informações de operação, formado o contexto, sobre essa rodada de execução do algoritmo de alocação.

Além do ciclo de operação do *framework*, iremos apresentar ainda como ele afeta um cliente genérico do SiteWhere. O primeiro diagrama de sequência apresentado pela Figura 22 demostra que o processo de conexão e utilização do SiteWhere não é afetado, ou seja, as atividades do *framework*, representadas aqui pelo registro de um novo cliente, são transparentes para o cliente final.

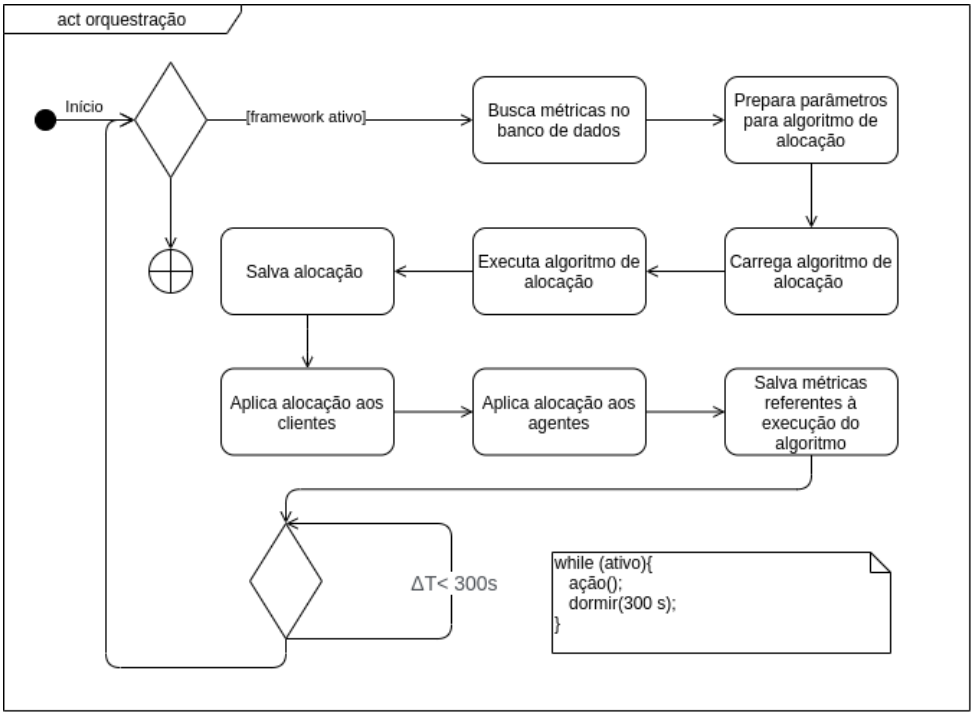


Figura 21 - Diagrama de atividades - orquestração

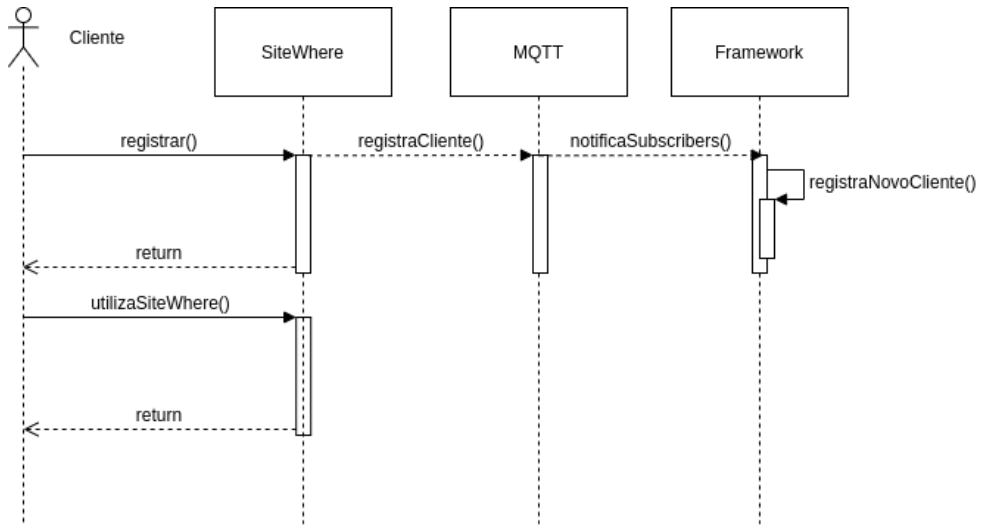


Figura 22 - Diagrama de sequência - cliente se registrando e utilizando o sistema

De maneira similar, um usuário genérico do SiteWhere não precisa conhecer o *framework* para que sejam coletadas as métricas. Para tal é necessário apenas que o cliente seja capaz de atender as funções customizadas do SiteWhere. A Figura 23 apresenta um diagrama de sequência com um teste de latência sendo solicitado e realizado, sem que o

cliente conheça nenhuma API do *framework*.

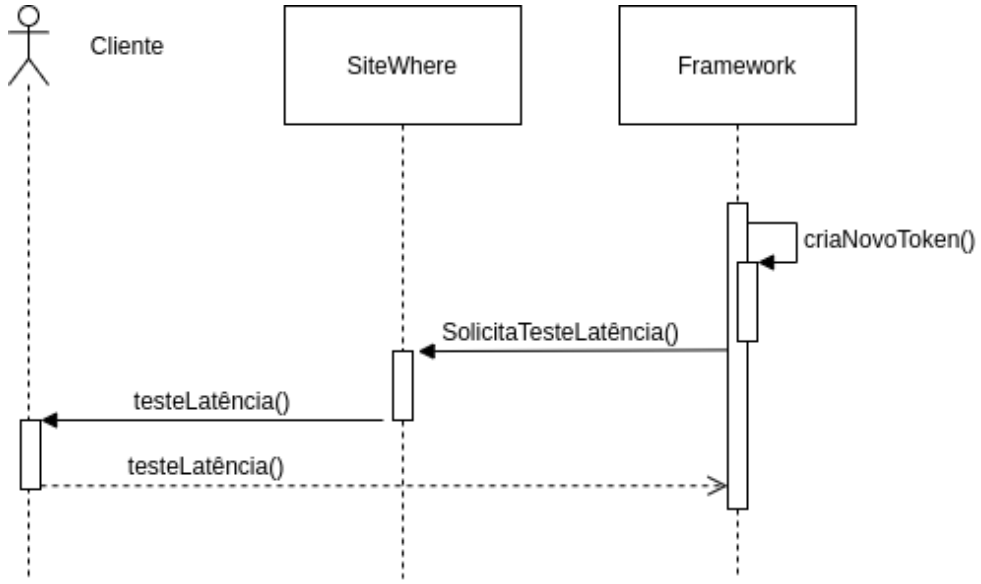


Figura 23 - Diagrama de sequência - framework solicitando teste de latência

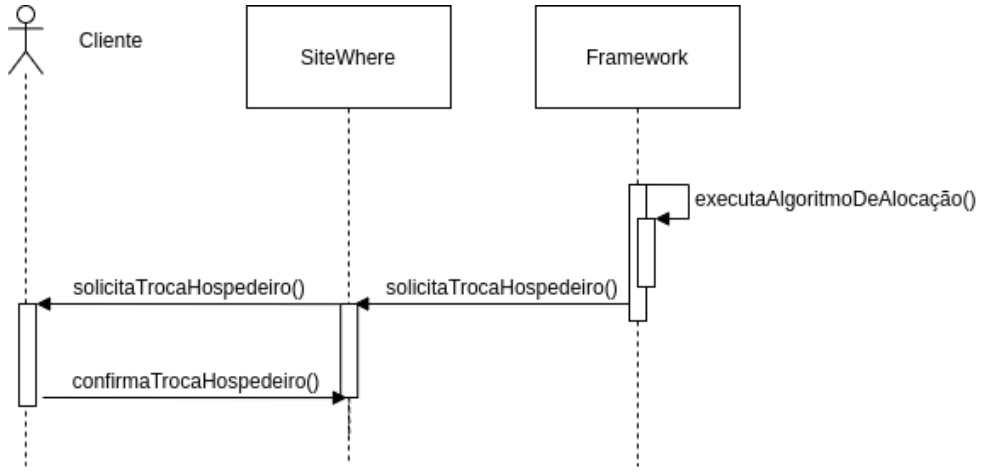


Figura 24 - Diagrama de sequência - framework solicitando troca de hospedeiro

Por último temos apresentado na Figura 24 um diagrama de sequência que demonstra como a solicitação de troca de hospedeiro, ou seja, a orquestração de um cliente é realizada. É importante notar que diferentemente do teste de latência a resposta do cliente tem como destino o SiteWhere, já que é necessário que o middleware receba a confirmação que o cliente foi capaz de realizar a troca solicitada.

4.6 Implementação

Um protótipo foi implementado como referência. Este protótipo foi usado nos testes e foi desenvolvido em Python 3. Foi utilizado *Flask* para oferecer a API de Orques-

tração. Toda a implementação dos algoritmos é feita em um módulo separado para que seja facilmente estendida. Ao todo foram criadas 3 imagens[3] e 5 *tags*:

1. Visualização - matheusstutzel/controlador-interface:latest
2. Nó Gerente - matheusstutzel/controlador:latest
3. Nó Gerente ARM - matheusstutzel/controlador:arm
4. Nó Trabalhador - matheusstutzel/worker:latest
5. Nó Trabalhador ARM - matheusstutzel/worker:arm

Como comentado anteriormente o serviço de visualização foi criado apenas para auxiliar nos testes e não será detalhado neste texto. Como pode ser observado nesta lista os nós Gerentes e os nós Trabalhadores tem suas próprias imagens, além disso, vale destacar que foram criadas versões x86 e ARM dessas imagens.

```
1 ARG baseARG
2 ARG final
3 FROM ${baseARG} as builder
4 COPY ./requirements.txt /
5 RUN pip3 install -r requirements.txt

6 FROM ${final}
7 WORKDIR /app
8 EXPOSE 8001
9 ENTRYPOINT [ "python" ]

10 # Configura Variáveis de ambiente
11 # CONSUL_HOST,SiteWhere_url, SiteWhere_port
12 # SiteWhere_password, SiteWhere_protocol
13 # SiteWhere_tenant-id, SiteWhere_tenant-auth

14 COPY --from=builder /install /usr/local

15 COPY *.py ./
16 CMD ["app.py"]

17 COPY ./algorithm/*.py ./algorithm/
```

Código 6 - Dockerfile para criação das imagens

O Código 6 apresenta o Dockerfile utilizado para a criação das imagens. Com exceção da última linha o conteúdo é o mesmo para os nós Gerentes e Trabalhadores.

[3]Todas as imagens são públicas e estão disponíveis no DockerHub, junto das imagens do SiteWhere utilizadas nos testes. [55]

Conforme discutido anteriormente apenas os nós Gerentes executam os algoritmos e por isso não faz sentido inclui-los na imagem dos nós Trabalhadores. Um ponto muito importante a ser destacado é que a instalação dos requisitos(linha 5) é realizado em uma imagem baseARG, enquanto todas as outras configurações são realizadas na imagem final. Só então na linha 14 os requisitos são copiados para a imagem final. Isso é feito por dois motivos principais: reduzir o tamanho da imagem final e prover uma camada de compatibilidade a mais.

Essa camada de compatibilidade foi criada para que fosse possível utilizando uma máquina x86 gerar as imagens destinadas às máquinas ARM. O único comando que realmente será executado durante a construção da imagem é o da linha 5, portanto é o único que deve utilizar a camada de compatibilidade. Todos os outros comandos são realizados pelo Docker. Conforme demonstrado no Código 7 baseARG foi utilizado para definir uma imagem base personalizada. Enquanto final utilizou uma imagem padrão. A mesma abordagem é utilizada para o nó Gerente e para o nó Trabalhador.

```
1 version: '3.5'

2 services:
3   gerente-arm:
4     build:
5       context: ./gerente/
6       args:
7         baseARG: matheusstutzel/python3.6-armqemu:debian
8         final: arm32v7/python:3.6-stretch
9     image: matheusstutzel/controlador:arm

10  trabalhador-arm:
11    build:
12      context: ./trabalhador/
13      args:
14        baseARG: matheusstutzel/python3.6-armqemu:debian
15        final: arm32v7/python:3.6-stretch
16    image: matheusstutzel/worker:arm
```

Código 7 - Criação das imagens para arquitetura arm

Essa imagem personalizada foi construída utilizando o pacote qemu-user-static disponível para o ubuntu. Esse pacote permite que comandos destinados a arquitetura ARM sejam executados em uma plataforma x86 utilizando uma biblioteca estática do QEMU. O Código 8 apresenta o Dockerfile completo da imagem `matheusstutzel/python3.6-armqemu:debian`.

```
1 FROM arm32v7/python:3.6-alpine
2 COPY qemu-arm-static /usr/bin
```

Código 8 - Imagem Docker de virtualização ARM

A criação da imagem destinada à arquitetura x86 utiliza o mesmo Dockerfile apresentado no Código 8, entretanto todos os parâmetros baseARG e final são trocados para a imagem python:3.6.

```
1 docker = Docker()
2 if iAmTheChosenOne():
3         docker.startInfra(SW_LOCATION)

4 consul = getConsulOrDie()
5 setNodeInfo()
6 sitewhere = SiteWhere.getSitewhere(url=getLocalIp())
7 if iAmTheChosenOne():
8     sitewhere.startBase(docker,SW_LOCATION,getIpMosquitto(SW_LOCATION))

9 Thread(target=orquestra,daemon=True).start()

10 Thread(target=createSensorNodes,daemon=True).start()

11 Thread(target=coletaStats,daemon=True).start()

12 Thread(target=pingServer,args=(8010,"delay",),daemon=True).start()

13 Thread(target=pingServer,args=(8011,"clients",),daemon=True).start()

14 Thread(target=mqttClient,daemon=True).start()

15 Thread(target=updateConsul,daemon=True).start()

16 app.run(host='0.0.0.0', port=8001)
```

Código 9 - Main framework

O Código 9 apresenta o código completa da função main presente em um nó Gerente. Este código implementa as atividades apresentadas no diagrama da Figura 15. A linha 1 obtém uma instância da classe Docker, criada para encapsular todas as operações necessárias que o *framework* tem que realizar com o Docker. Essa biblioteca utiliza como base a biblioteca padrão do Docker para Python.

Conforme descrito na Seção 4.5 o nó gerente deve executar um algoritmo para determinar se ele é o nó Gerente escolhido(NGE), isso foi encapsulado na função iAmTheChosenOne que simplesmente retorna verdadeiro se for o NGE ou falso caso contrário(linha 2

apresenta isso pela primeira vez).

Caso o nó seja realmente o NGE ele irá na linha 3 inicializar a infraestrutura necessária para o *framework*. Logo em seguida o nó irá tentar se conectar com o banco de dados, conforme descrito pela linha 4. É importante notar que o *framework* depende do banco de dados e portanto não irá inicializar corretamente caso ele esteja indisponível.

Na linha 6 o nó obtem uma instância da classe SiteWhere. Essa biblioteca foi construida para encapsular todas as operações necessárias que o *framework* tem que realizar com o SiteWhere. Essa biblioteca foi construída do zero, visto que a versão utilizada do SiteWhere não oferecia uma biblioteca Python. Caso o nó seja um NGE ele irá, na linha 8, inicializar a infraestrutura do SiteWhere.

Cada uma das atividades descritas na Seção 4.5 é implementada como uma função e então passada para uma thread nova, de acordo com a seguinte lista:

1. orquestração, linha 9
2. iniciar novos nós, linha 10
3. coletar métricas, linha 11
4. servidor de ping, linhas 12 e 13
5. monitorar agentes, linha 14
6. salvar métricas, linha 15

Além disso, temos ainda a API de orquestração inicializada na linha 16. A próxima Seção apresenta detalhes dessa API e da atividade de orquestração definida na linha 9.

4.7 Orquestração

Conforme apresentado anteriormente na Figura 21 da Seção 4.5 o *framework* conta com um ciclo de operação para realizar a orquestração dos clientes/agentes. A implementação deste ciclo é apresentada no Código 10.

A linha 3 realiza o teste para determinar se este é um NGE. Caso não seja a thread dorme por 30 segundos e recomeça o loop. Isso é importante pois apenas o NGE deve executar a orquestração e conforme discutido anteriormente o NGE pode mudar. Caso

```
1 def orquestra():
2     while ativo:
3         if iAmTheChosenOne():
4             try:
5                 algorithm()
6                 time.sleep(5*60)
7             except Exception as e:
8                 log(e)
9                 time.sleep(30)
10        else:
11            time.sleep(30)

12 def algorithm():
13     estadoAtual = obtemEstadoAtualDoSistema()
14     G,A,F,C,cap,translate= extrairParametros(estadoAtual)
15     start=time.time()
16     V,rC=Algo(G,A,F,C,delayLimit,cap)
17     novoEstado = convertToFramework(V,rC,cap)
18     aplicarConfig(novoEstado,translate,A,"Algo",time.time()-start)

19 def aplicarConfig(novoEstado,translate,A,algo,t):
20     persisteAlocacao(novoEstado,translate)
21     for i in A:
22         if i in novoEstado:
23             log("Trocando "+str(translate[i])+" para o nó "+str(novoEstado[i]))
24             if translate[i][0]=='client':
25                 aplicaTrocaDeClienteNoBanco(translate[i][1], translate[novoEstado[i]][1])
26             elif translate[i][0]=='device':
27                 host = obtemHostId(translate[novoEstado[i]][1])
28                 sitewhere.changeHost(translate[i][1],host[0])
29         else:
30             log(str(i)+" não recebeu servidor")
31     registraMetricaExecucaoAlgoritmo(id, algo, t)
```

Código 10 - Orquestração

este nó seja o NGE na linha 5 a orquestração é realizada e então na linha 6 a thread dorme por 5 minutos antes de recomeçar o loop.

A função executada na linha 5 contem todos os passos para realizar a orquestração. Primeiro, na linha 13, o estado atual do sistema é carregado do banco de dados e então na linha 14 esse estado atual é convertido nos parâmetros que a interface apresentada no Código 3 da Seção 4.3 espera. É importante notar que o *framework* cria mais um parâmetro intitulado *translate*. O *translate* é utilizado como dicionário entre a representação dos nós no grafo G e os metadados presentes no estado atual.

Após a criação dos parâmetros o algoritmo de alocação é executado na linha 16 e o retorno é então convertido para um novo estado do sistema na linha 17. Finalmente na linha 18 esse novo estado é aplicado.

Existe uma função dedicada a aplicar esse estado do sistema, isso é, para realizar a orquestração. A primeira etapa dessa função é salvar o estado novo do sistema no banco de dados(linha 20). Após persistir o *framework* irá iterar sobre todos os agentes e clientes. Apesar de ser o mesmo loop para os dois a lista A é ordenada de tal forma que os clientes precedem os agentes. Para todo cliente o *framework* irá aplicar as alterações diretamente no banco(linha 25) e elas estarão imediatamente disponíveis na API Rest. Já para os agentes a orquestração acontece através do SiteWhere(linha 28) e requer que uma função customizada seja executada pelo *middleware* para que o agente efetivamente troque de nó de processamento.

5 REDE OVERLAY

Durante os testes iniciais do *framework* foram encontradas limitações nas redes virtuais que o Docker permite formar. Conforme discutido na Seção 3.1.1, por padrão, o Docker não consegue estabelecer as conexões necessárias para o *swarm* se os nós não estiverem na mesma subrede. Apesar de ser possível criar um *swarm* com nós de subredes diferentes através de configurações específicas, é necessário, neste caso, que todos os nós tenham endereços IP reais. Esse não é um cenário comum na Internet das Coisas. Para que o *framework* pudesse se manter compatível com as infraestruturas de rede fazendo uso de NAT e com topologias com múltiplas subredes, foi necessário introduzir mais um componente à arquitetura proposta, discutido neste Capítulo.

Na Seção 5.1 utilizamos um cenário de exemplo para mostrar a limitação encontrada nas redes virtuais oferecidas pelo Docker. Em seguida, discutimos como esta limitação pode ser resolvida utilizando uma rede *overlay*. A partir dessa rede *overlay*, introduzimos alguns dos conceitos base utilizados na solução para a rede *overlay* proposta.

Antes do desenvolvimento de uma nova rede *overlay* foram consideradas algumas técnicas de redes virtuais, discutidas na Seção 5.2. É importante pontuar que a solução buscada deveria permitir que o *framework* pudesse ser utilizado em qualquer topologia e sem nenhuma configuração adicional nos equipamentos de rede.

Na Seção 5.3 é, então, apresentada a solução proposta para a rede *overlay*. Detalhes de como os pacotes são enviados entre os elementos que se comunicam sobre a rede *overlay* são apresentados na Seção 5.4 e, a seguir, na Seção 5.5, é apresentada a técnica utilizada para resolver o problema do acesso aos nós que estejam dentro de redes servidas por NAT. Por último, na Seção 5.6, são apresentados os resultados dos testes de desempenho realizados.

5.1 Visão geral

Consideramos um cenário recorrente em aplicações IoT, onde o problema detectado fica evidente, e como este problema influenciou algumas das decisões tomadas durante o desenvolvimento da solução final.

Na Figura 25, temos 3 *hosts*: A, B e C. Cada um desses *hosts* está executando um aplicativo, *App*. O *host* B está dentro de uma rede com *NAT* e o *host* C está em outra

rede com *NAT*. Para esse cenário, consideramos que o *host* A possui um IP real.

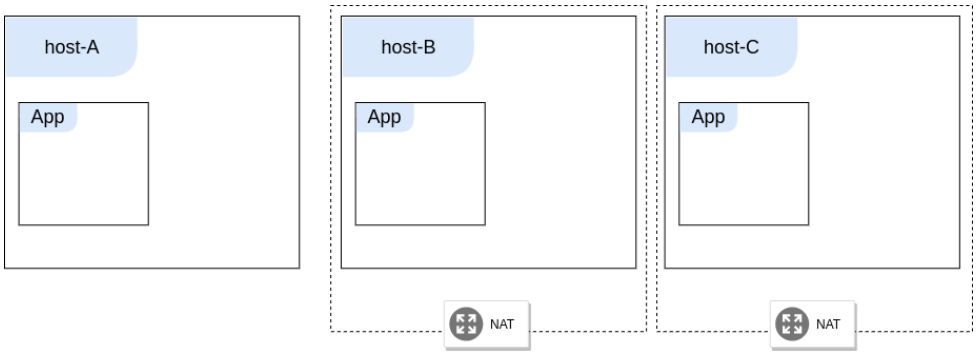


Figura 25 - Cenário inicial

Neste cenário, o *host* B e o *host* C não conseguem trocar mensagens diretamente, conforme explicado na Seção 1.4. Mais do que isso, eles só conseguem trocar mensagens com o *host* A se eles iniciarem a conexão. Como discutido na Seção 1.4, é possível resolver esse problema configurando um mapeamento adequado de portas no servidor *NAT* de cada uma das redes. Essa solução depende, entretanto, do acesso ao equipamento responsável pelo serviço de *NAT* e isso não é sempre possível. Além disso, não é trivial estabelecer este mapeamento de forma dinâmica, sob demanda.

Este cenário é diretamente comparável ao encontrado durante os testes com o Docker, onde cada um dos *hosts* executa um contêiner Docker (*App* na figura) e é necessário estabelecer a conexão entre eles para a criação do *swarm*. Visto que não é possível pré-definir a ordem das conexões feitas pelo Docker, não seria possível garantir que o *host* B e o *host* C iriam sempre iniciar a conexão.

Para que seja possível criar o *swarm* é necessário que o *host* A possa, a qualquer momento, estabelecer uma conexão com os outros *hosts*. Mais do que isso, é necessário também que a qualquer momento seja possível estabelecer uma conexão entre quaisquer pares de*hosts*. Assim, temos a primeira limitação ilustrada, dado que existe a dificuldade de se "encontrar" os *hosts* atrás de um serviço NAT sem que haja um mapeamento explícito previamente disponível.

Na rede proposta, um dos nós da rede assume o papel de servidor de endereços virtuais e todos os outros nós devem se conectar a ele. Esse servidor também fica responsável pelo controle dos endereços de IP virtuais dos integrantes dessa rede. Ou seja, cada nó novo que entra na rede recebe um novo endereço de IP que será fornecido pelo nó servidor. Todos os endereços virtuais pertencem à mesma sub-rede, resolvendo assim

o problema das sub-redes do Docker.

A solução proposta organiza a rede *overlay* como uma camada de software adicional no suporte à comunicação, que deve ser executada em cada *host*. As seguintes características são destacadas:

- Para tornar sua atuação transparente, uma interface de rede virtual é criada. Cada *host* conectado a esta rede *overlay* se comunica diretamente com os outros *hosts* desta rede, independente da topologia física, simplesmente usando o número IP (virtual) associado a esta interface;
- Um *host* é selecionado para executar o serviço de endereço, que gerencia a atribuição dos endereços IP virtuais e permite a interação direta entre os outros *hosts*. O número IP real deste *host* deve ser conhecido pelos outros *hosts* para que possam solicitar participação na rede *overlay*;
- O transporte de mensagens entre os *hosts* da rede *overlay* é feito por UDP, utilizando uma técnica que encapsula outros protocolos de transporte, fazendo o papel comparável aos protocolos de enlace.

A implantação da rede *overlay* deve preceder a ativação dos outros componentes do *framework*. Primeiro, o módulo é iniciado como servidor no *host* selecionado e, em seguida, os o módulo cliente é iniciado nos outros *hosts*. No cenário de exemplo, ao se iniciar o módulo como servidor no *host* A, Figura 26, uma porta de serviço é reservada (55555, no caso, mas a escolha é arbitraria).

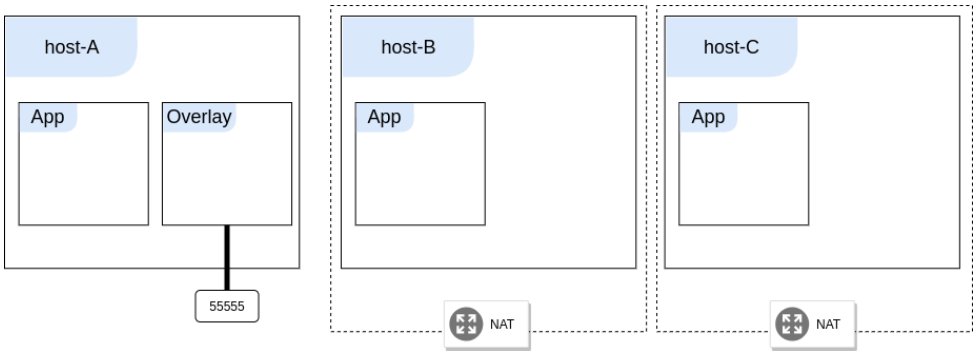


Figura 26 - Host A como servidor

Conforme descrito inicialmente, o *host* A possui um endereço IP real válido. Quando um outro *host* necessita participar da rede *overlay*, deve enviar uma mensagem

de requisição para para este endereço IP real, porta 55555, e será recebido pela aplicação que implementa o servidor da rede *overlay*.

Na inicialização do módulo servidor no *host* A, uma nova interface de rede virtual é também criada. Por ser o nó servidor essa interface recebe o IP 10.0.0.1, o primeiro endereço IP da faixa empregada pela rede *overlay*, informação também disponível na configuração do servidor e na configuração de todos os clientes da rede *overlay*. Todo o fluxo de mensagens enviado e recebido por essa interface virtual é tratado pelo módulo servidor. Isso é transparente para os outros elementos do *framework* e para as aplicações que estejam utilizando esta interface de rede.

Os outros *hosts* que precisam fazer parte da mesma rede *overlay* inicializam o módulo da rede como cliente e, assim, uma interface virtual é criada. No papel de cliente não é necessário reservar uma porta específica (Figura 27).

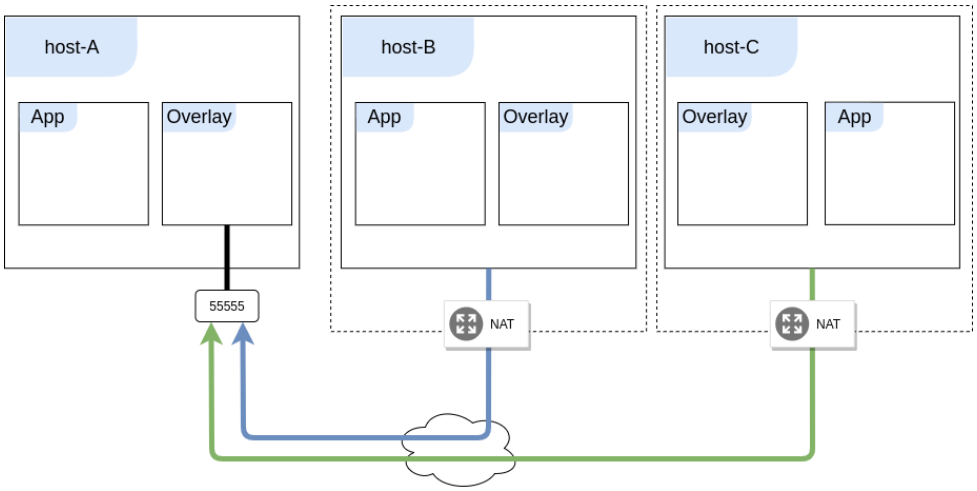


Figura 27 - Host B e C como clientes

Para concluir a inserção de um *host* cliente na rede *overlay*, uma conexão TCP é estabelecida com o servidor e, em seguida, uma requisição é enviada ao servidor, que responde com o número IP virtual, dentro da faixa registrada na configuração do servidor, a ser usado pelo cliente. Em seguida o cliente associa este número IP à interface de rede virtual também criada. No exemplo, Como indicado pelas setas em azul e verde, os *hosts* B e C enviam mensagens para o IP real do *host* A na porta padronizada (55555, no caso) solicitando inclusão na rede *overlay*. Com a resposta do servidor, as interfaces virtuais destes *hosts* passam para o estado "conectado" e recebem, respectivamente, os endereços 10.0.0.2 e 10.0.0.3, e os *hosts* agora podem usar a rede *overlay*. Para que as aplicações executando no *hosts* (App, na figura) utilizem a rede *overlay* basta apenas se fazer uso

do número IP da rede *overlay* atribuído pelo servidor, que é associado à interface virtual criada. Por exemplo, para trocar mensagens com o App no *host* C é necessário utilizar o IP 10.0.0.3.

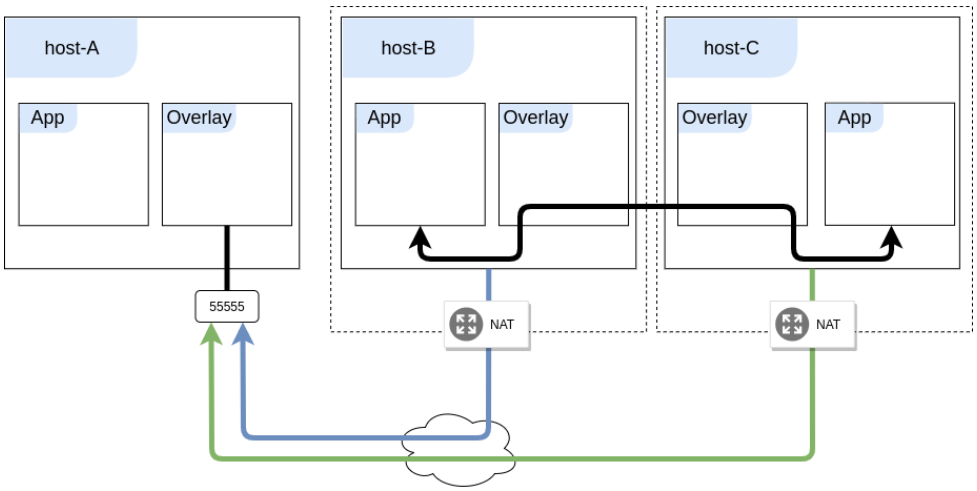


Figura 28 - App B e C conectados

A Figura 28 apresenta a troca de mensagens entre os App dos *hosts* B e C. Propositalmente, esta figura não representa o caminho real do dado. Em um primeiro momento é simples imaginar que o dado enviado por B será enviado primeiro para A e só então será enviado para C. Principalmente se for considerado que as conexões estabelecidas na Figura 27 continuam presentes. Entretanto, este tipo de solução não teria o desempenho necessário para IoT. A solução proposta para a rede *overlay*, como veremos na sequência, garante que a mensagem enviada pelo *host* B é diretamente entregue ao *host* C.

5.2 Soluções Relacionadas

Existem outras soluções propostas para esse problema, disponíveis na literatura. Algumas delas serão descritas em seguida, com o objetivo de justificar por que, ainda assim, foi necessário o desenvolvimento de uma solução personalizada.

A primeira técnica considerada foi a rede *overlay Virtual EXtensible Local Area Network* (VxLAN) [56] [57]. Esse tipo de rede *overlay* permite interconectar múltiplas redes de camada 3. Para tal, são utilizados utilizados dispositivos na borda da rede chamados *VXLAN tunnel endpoint*(VTEP). Esses VTEPs recebem os datagramas IP e os encapsulam em um novo datagrama IP com o endereço de outro VTEP, que faz o processo reverso utilizando o endereço de destino.

A principal diferença é que para a rede proposta não existe um VTEP e quem fica responsável por realizar essa encapsulamento é a própria aplicação que define a interface de rede virtual. Apesar de ser possível estabelecer um conjunto de túneis VTEP, com o intuito de combinar diversas redes VxLAN, e assim estabelecer uma topologia *mesh*, esta solução não seria suficiente para o cenário proposto, já que a topologia da rede é dinâmica e portanto, não poderia ser pré-configurada em cada VTEP.

Em nossos experimentos de prospecção não foi possível estabelecer a conexão do *cluster* Docker utilizando essa técnica sem que fosse necessário fazer alterações adicionais no equipamento de rede que responsável pelo NAT. Sendo assim essa alternativa não foi utilizada. Além disso, seria necessário gerenciar diversos VTEP e todas as interconexões.

Em seguida foi considerado o uso do *Wireguard* [58], uma implementação de VPN com grande foco em sua criptografia, que, naquele momento, estava sendo desenvolvida para se tornar parte do kernel do Linux.

O *Wireguard* utiliza a mesma abordagem da nossa rede *overlay* e cria uma interface de rede virtual. Todos os dados trafegados por essa rede são encapsulado em datagramas UDP criptografados. Além disso, existem diversas outras camadas de segurança que são adicionadas pelo *Wireguard*.

Apesar de promissora, essa solução não oferecia opção de construir uma rede *mesh*, sendo possível apenas a topologia estrela. Esse tipo de topologia tornava inviável a utilização dessa solução nos cenários e aplicações IoT alvo de nosso *framework*.[4]

5.3 Rede Overlay Proposta

A rede *overlay* proposta tem como base o conceito de *Stateless Transport Tunneling (STT)* [59]. Especificamente, são utilizados túneis UDP através de uma interface TUN, desta forma todos os integrantes podem pertencer a mesma rede virtual, independente da topologia física da rede.

No contexto do nosso *framework*, isso permite que todos os nós de processamento, que executam o orquestrador e os serviços do SiteWhere estejam na mesma rede virtual. É importante notar que os clientes, que fazem uso da API HTTP REST do SiteWhare, e os agentes que fazem uso do protocolo MQTT com um *broker* também provido pelo

[4]Durante a elaboração dessa dissertação, o *Wireguard* passou a dar suporte a topologia *mesh*. Apesar disso, o *Wireguard* ainda não oferece ferramentas para criação e manutenção de tais topologias.

SiteWhare, não precisam fazer parte do *cluster* para consumir os serviços e portanto não são afetados por esta solução.

A interface TUN é uma interface virtual do *kernel* do Linux [60]. Todos os dados enviados através deste dispositivo virtual são capturados e redirecionados para um processo definido pelo usuário. A interface TUN captura pacotes IP enquanto a interface TAP captura *frames* Ethernet. Esse dispositivo virtual também permite que este software possa injetar novos pacotes na rede, simulando pacotes recebidos de outros nós da rede.

Na Figura 29 temos um exemplo de utilização de uma interface virtual TUN (tun0). Considerando que a rede overlay já está operacional e as aplicações, App, nos *hosts* A e B estão configuradas para utilizar a interface tun0. O App sendo executado no *host* A precisa enviar uma mensagem para o App sendo executado no *host* B. O pacote contendo a mensagem é direcionado para a interface tun0, que intercepta a mesma e a redireciona para o módulo de software da rede *overlay*. Esse pacote então é encapsulado em um datagrama UDP e reenviado através da interface eth0 para o *host* B em uma porta predefinida. Essa mensagem é recebida pela instância do módulo de software da *overlay* na porta predefinida do *host* B, que por sua vez desencapsula a mensagem original e a injeta na interface virtual tun0, para que seja finalmente recebida pelo App B. Observa-se que esta estratégia de interceptação-redirecionamento é bastante empregada em sistemas distribuídos como o RPC.

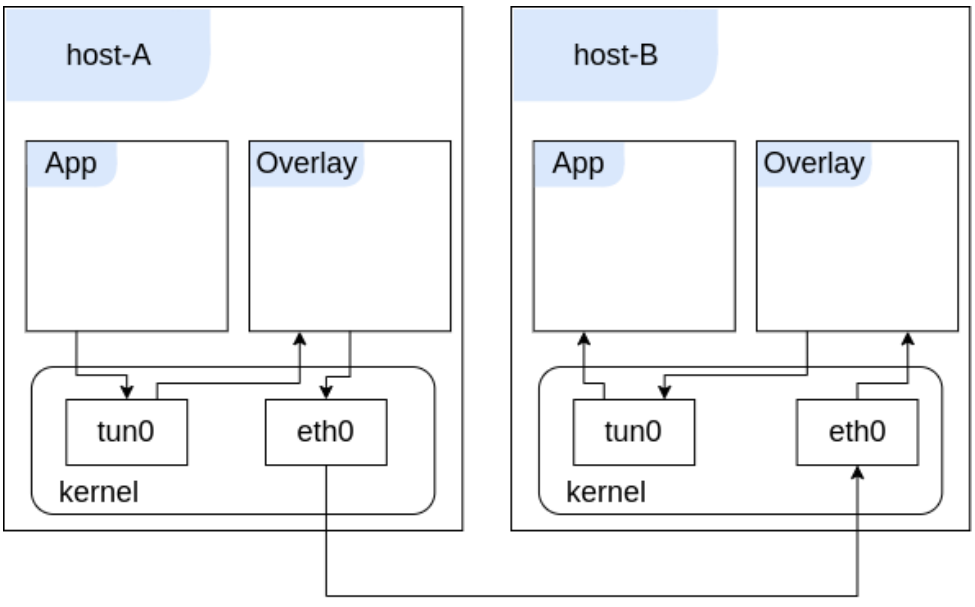


Figura 29 - Interface TUN/TAP

Para se estabelecer a rede *overlay*, permitindo a inclusão de novos *hosts*, é necessário

que um nó servidor seja acessível para todos os outros na rede. Este requisito vem da utilização da técnica de *UDP Hole Punching* e será explorada na Seção 5.5. Este nó se tornará o *Master* desta nova rede virtual e receberá o IP virtual 10.0.0.1, passando a responder requisições de inclusão de novos nós e a consultas de "rotas", para nós desejando se comunicar com outros. Por padrão o Master oferece endereços na faixa 10.0.0.0/8 aos novos nós. Utilizando como base o exemplo discutido na Seção 5.1 o *host* A seria o Master dessa rede.

Um nó cliente que deseja fazer parte da rede *overlay* deve, primeiro, se conectar ao nó *Master* e solicitar um endereço IP virtual. Na implementação desenvolvida esta solicitação é realizada utilizando TCP e o IP real (e a interface física) do nó *Master*. Entretanto, nada impede que esta requisição seja feita utilizando a própria rede virtual. Ao receber o endereço IP solicitado, o novo nó atribui este endereço à interface de rede virtual e, então, este nó esta pronto para trocar mensagens diretamente com os outros nós que fazem parte da rede *overlay*.

Para que seja possível manter o uso do mecanismo *UDP Hole Punching*, a partir do momento em que um cliente recebe um endereço IP ele passa a enviar mensagens de *keep alive* para o nó Master utilizando a rede *overlay*.

Conforme demonstrado na Figura 29, Uma mensagem enviada por uma aplicação (App do *host* A), com destino à outro processo (App do host B), é encaminhada para a interface virtual (tun0). A mensagem é interceptada pelo *kernel* do Linux que encaminha a mesma para o processo de controle da rede *overlay*. Esse processo encapsula a mensagem em um datagrama UDP e o envia pela interface real para o processo de controle do *host* de destino (*host* B), usando o endereço IP válido. O datagrama UDP recebido pela interface física no host de destino é desencapsulados pelo processo de controle e reinjetados na interface virtual (tun0). Finalmente a mensagem original é entregue ao App do *host* B.

Para que as mensagens interceptados na interface virtual no *host* de origem possam ser enviados para o *host* destinatário na rede *overlay* o processo de controla do cliente deve manter uma tabela de conversão entre o endereço virtual e o endereço real de cada interlocutor, como ocorre com a tabela ARP — fazendo um paralelo. Essa tabela é atualizada toda vez que um nó recebe uma mensagem. Caso o nó cliente deseje enviar uma mensagem para um nó que não esteja na tabela local ele deve consultar o nó *Master*. O nó *Master* é o único nó da rede que, a qualquer momento, conhece todos os outros nós

e é o único nó que, garantidamente, é conhecido por todos os outros.

O processo de controle utiliza a própria rede *overlay* para trocar mensagens de controle com os outros integrantes da rede. Essas mensagens de controle são detalhadas na Seção 5.4.

5.4 Protocolo e Mensagens

Resumindo as principais trocas de mensagens na rede *overlay* proposta temos:

- Troca de mensagens entre processos cliente e servidor, com um pacote de dados estruturado;
- Mensagens de *keep alive* enviadas periodicamente para o nó *Master* mantendo a consistência sobre a informação de que um nó permanece, ou não está mais, participando da rede *overlay*;
- Troca de mensagens *who is* enviada para consultar o mapeamento de um endereço de rede real em seu correspondente na rede virtual, na mesma linha de uma consulta ARP, com pacotes de requisição e respostas estruturados de forma específica.

Pacote de dados. Conforme mencionado anteriormente, a rede *overlay* exerce um papel similar à camada de enlace. Para tal, todos os pacotes IP recebidos na interface virtual são encapsulados em datagramas UDP e enviados para o destinatário utilizando a rede física. Ao receber este datagrama o destinatário então desencapsula a mensagem e injeta o pacote original na rede virtual.

A Tabela 1 apresenta o pacote final enviado na rede física. Ele é composto dos campos:

- *Payload*: dados do pacote IP original;
- *Inner IPv4 header*: cabeçalho original do pacote IP original;
- *Size*: tamanho do pacote IP original, incluindo o cabeçalho;
- *Outer UDP header*: cabeçalho UDP do novo datagrama;
- *Outer IP header*: cabeçalho IP do novo pacote.

Tabela 1 - Pacote de dados

Outer header	
Outer IP header 20 bytes	Outer UDP header 8 bytes
Inner header	
Size 2 bytes	Inner IPv4 header 20 bytes
Data	
Payload 0-1450 bytes	

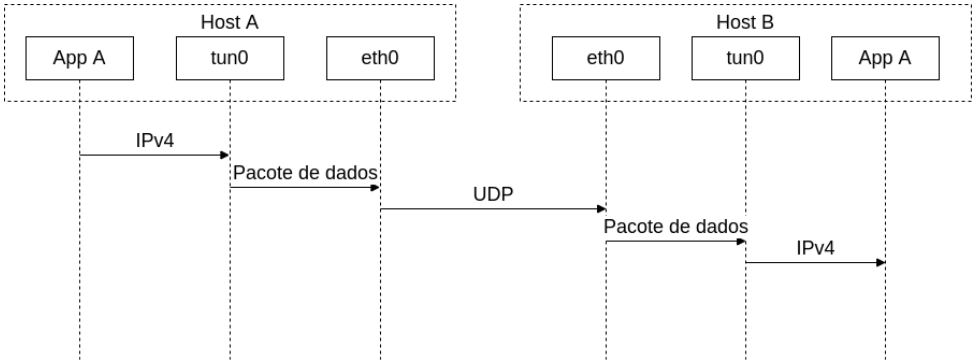


Figura 30 - Diagrama de sequência - Pacotes de dados

A Figura 30 apresenta um diagrama de sequência detalhando como os pacotes de dados são enviados.

Keep alive. Conforme comentado na seção anterior, todos os clientes da rede *overlay* enviam periodicamente mensagens controle denominadas *keep alive*. A Tabela 2 apresenta o pacote final enviado na rede física. Ele é composto dos campos:

- *My virtual IP*: IP virtual do nó que está enviando o *keep alive*;
- *Reserved*: campo reservado;
- *Identification*: campo utilizado para identificar o pacote *keep alive*;
- *Outer UDP header*: cabeçalho UDP do novo datagrama;
- *Outer IP header*: cabeçalho IP do novo pacote.

A Figura 31 apresenta um diagrama de sequência de como o uso de pacotes *keep alive* são enviados. As mensagens de *keep alive* são eivadas pelos módulos de software da rede *overlay*, diretamente para o servidor *Master* pela rede física, não passando pelo processo de interceptação e encapsulamento dos pacotes vindos as aplicações.

Tabela 2 - Pacote de keep alive

Outer header	
Outer IP header 20 bytes	Outer UDP header 8 bytes
Inner header	
Identification 2 bytes	Reserved 2 bytes
Data	
My virtual IP 4 bytes	

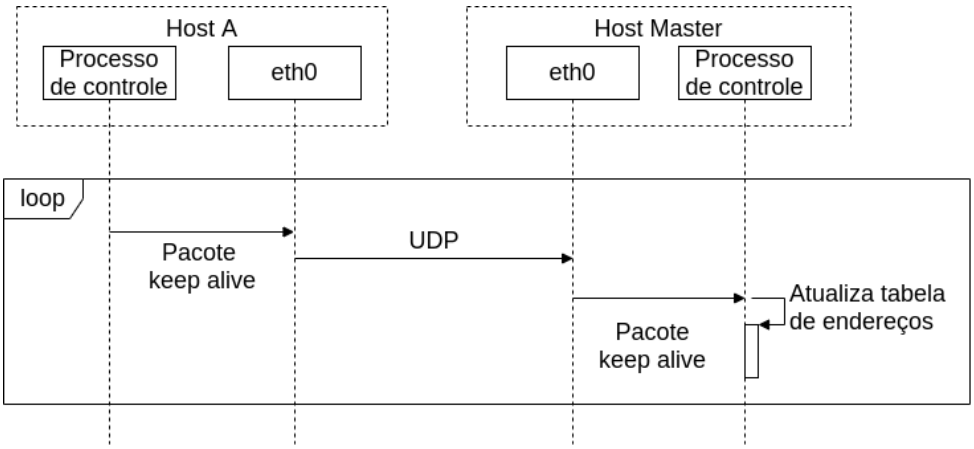


Figura 31 - Diagrama de sequência - Pacotes keep alive

Who is. Quando um nó α deseja enviar uma mensagem para um outro nó β ele deve primeiro consultar a sua tabela de conversão. Caso a entrada referente ao nó β não esteja nesta tabela, o nó α deve enviar uma mensagem *who is*.

A Tabela 3 apresenta o pacote final enviado na rede física. Ele é composto dos campos:

- *Dest. IP*: IP virtual do nó β;
- *Source IP*: IP virtual do nó α;
- *Reserved*: campo reservado;
- *Identification*: campo utilizado para identificar o pacote *who is*;
- *Outer UDP header*: cabeçalho UDP do novo datagrama;
- *Outer IP header*: cabeçalho IP do novo pacote.

Ao receber uma mensagem de *who is* o nó *Master* envia duas mensagens com o formato apresentado na Tabela 4:

Tabela 3 - Pacote de who is

Outer header	
Outer IP header 20 bytes	Outer UDP header 8 bytes
Inner header	
Identification 2 bytes	Reserved 2 bytes
Data	
Source IP 4 bytes	Dest. IP 4 bytes

(i) Uma para o nó de origem(α, no exemplo) com as informações do nó de destino (β, no exemplo) e com o código de identificação da resposta de *who is*.

- *Dest. port*: Porta utilizada pelo módulo de software da rede *overlay* do nó β;
- *Dest. real IP*: IP real do nó β;
- *Dest. virtual IP*: IP virtual do nó β;
- *Reserved*: campo reservado;
- *Identification*: campo utilizado para identificar o pacote *who is* de resposta;
- *Outer UDP header*: cabeçalho UDP do novo datagrama;
- *Outer IP header*: cabeçalho IP do novo pacote.

(ii) E outra mensagem para o nó de destino com os dados do nó de origem e o código de identificação referente à mensagem de *talk back*.

- *Dest. port*: Porta utilizada pelo módulo de software da rede *overlay* do nó α;
- *Dest. real IP*: IP real do nó α;
- *Dest. virtual IP*: IP virtual do nó α;
- *Reserved*: campo reservado;
- *Identification*: campo utilizado para identificar o pacote *talk back*;
- *Outer UDP header*: cabeçalho UDP do novo datagrama;
- *Outer IP header*: cabeçalho IP do novo pacote.

Desta forma o nó de origem agora tem todos os dados necessários para identificar o nó de destino. E o servidor solicitou, através do pacote de *talk back*, que o nó de destino envie uma mensagem para o nó de origem. Esse processo é apresentado na Figura 32.

Tabela 4 - Pacote de resposta who is/talk back

Outer header		
Outer IP header 20 bytes	Outer UDP header 8 bytes	
Inner header		
Identification 2 bytes	Reserved 2 bytes	
Data		
Dest. virtual IP 4 bytes	Dest. real IP 4 bytes	Dest. port 4 bytes

5.4.0.1 Visão integrada dos protocolos

A Figura 32 apresenta um diagrama de sequência completo com todas as trocas de mensagem necessárias para que o *Host* A possa enviar uma mensagem pela primeira vez para o *Host* B, até então desconhecido por A. É importante notar que para simplificar a imagem tanto e o processo de controle quanto a interface virtual foram representados pelo mesmo elemento (tun0). As mensagens trocadas são:

1. O App A envia uma mensagem para o App B utilizando a rede *overlay*;
2. O Processo de controle de A não conhece o *Host* B e, por isso, envia uma mensagem *who is* para o Master.
3. Essa mensagem é transportada pela rede física dentro de um datagrama UDP.
4. O Processo de controle do Master recebe a mensagem *who is*.
5. O Processo de controle do Master responde à mensagem *who is*.
6. Essa mensagem é também transportada pela rede física dentro de um datagrama UDP.
7. O Processo de controle de A recebe a resposta da mensagem *who is*.
8. O Processo de controle do Master envia uma mensagem de *talk back* para o Processo de controle de B.

9. Essa mensagem é trafegada pela rede física dentro de um datagrama UDP.
10. O Processo de controle de B recebe uma mensagem de *talk back*.
11. O Processo de controle de B envia uma mensagem de *talk back* para o Processo de controle de A.
12. Essa mensagem é trafegada pela rede física dentro de um datagrama UDP.
13. O Processo de controle de A recebe a mensagem de *talk back* de B.
14. O Processo de controle de A encapsula a mensagem original do App A em um pacote de dados.
15. Essa mensagem é trafegada pela rede física dentro de um datagrama UDP.
16. O Processo de controle desencapsula a mensagem do pacote de dados.
17. O App B recebe a mensagem enviada pelo App A.

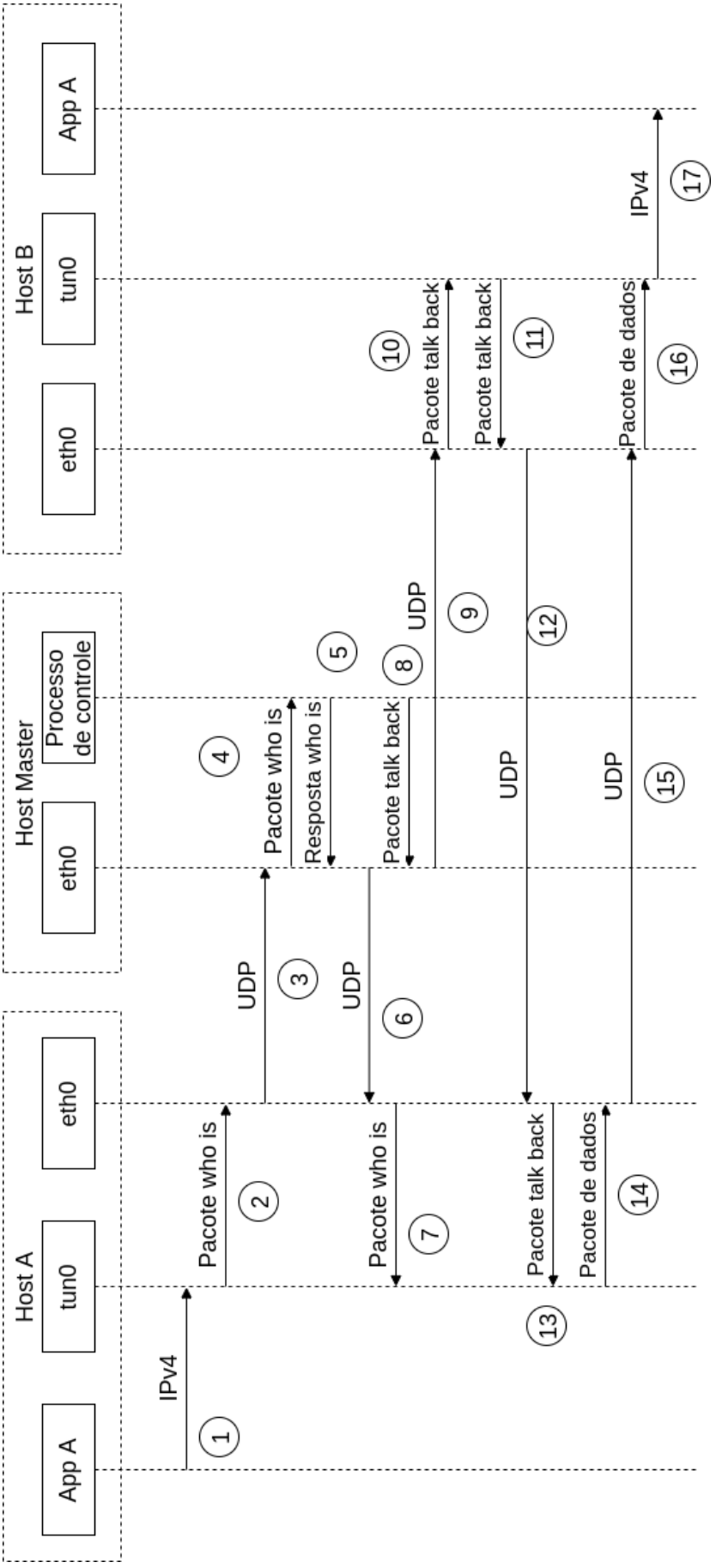


Figura 32 - Diagrama de sequência - Pacotes who is

5.5 UDP Hole Punching

Para que os nós possam se comunicar sobre a rede *overlay* mesmo estando dentro de um rede protegida por NAT, foram empregadas técnicas de UDP Hole Punching [61], também conhecido como *NAT traversal*.

Um elemento integrante de um servidor NAT é uma tabela contendo o mapeamento de *hosts* e portas (tabela NAT) presente no equipamento responsável pelo serviço. Assim, quando um *host* A, dentro da rede protegida por NAT, envia uma mensagem para um outro *host* fora desta rede, a tabela de NAT deve ser preenchida com o IP interno do *host* A e a porta externa utilizada. Quando a resposta é recebida a porta de destino é comparada com aquela armazenada na tabela de NAT e a mensagem é entregue ao *host* A utilizando o IP interno.

A técnica de UDP Hole Punching utiliza a tabela de NAT não apenas para o destinatário da troca de mensagens original, mas também para qualquer outro *host*. Utilizando como exemplo a rede *overlay*, a partir do momento que o *host* A envia uma mensagem para o *host* executando o Master, é feito um registro na tabela de NAT na rede do *host* A. Quando o Master recebe a mensagem do A, ele guarda os dados do endereço de origem dessa mensagem e a porta utilizada. Quando o *host* B quer enviar uma mensagem para o *host* A ele pode consultar os registros do Master e utilizar o mesmo endereço e porta. Neste momento é realizado o UDP Hole Punching, visto que o *host* B está enviando mensagens para o *host* A utilizando o mapeamento da tabela de NAT que foi realizado para o Master.

5.6 Testes e desempenho

Antes de integrar a rede *overlay* desenvolvida ao *framework* realizamos alguns testes de comportamento e desempenho.

Para a realização do teste de desempenho da rede *overlay* foram utilizados 3 *hosts* diferentes, conforme apresentado na Figura 33. O *host* A era uma máquina virtual alocada no serviço de nuvem da Google (GCP), na região *asia-northeast1-a*, que fica localizada em Tóquio, Japão. O *host* B era uma máquina virtual alocada na infraestrutura da UERJ. Já o *host* C era uma máquina física também localizada no Rio de Janeiro. Durante o teste o *host* A foi o servidor *Master* da rede *overlay*, enquanto os *hosts* B e C eram

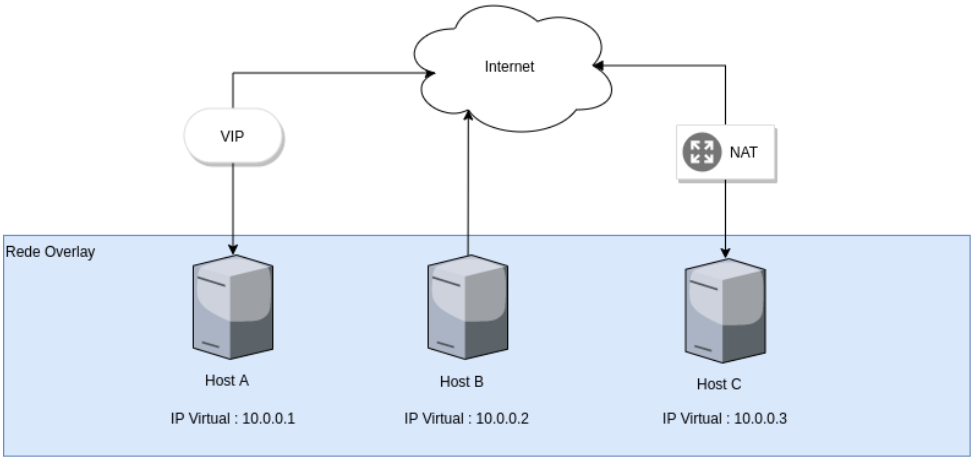


Figura 33 - Overlay

clientes. Estes *hosts* foram escolhidos para simular o pior caso, onde o *Master* da rede está geograficamente distante dos clientes que querem se comunicar entre si.

O primeiro teste, de *ping* (icmp echo), foi realizado entre os *hosts* B e C, com e sem a utilização da rede *overlay*. O resultado é apresentado na Tabela 5 com intervalo de confiança de 95%. Observa-se que sem a rede *overlay* Conforme esperado o teste de *ping* de B para C não funciona, já que C está atrás de um serviço NAT. O teste entre B e A com a rede *overlay* tem um acréscimo de menos de 10 ms, menos de 4% em média.

Tabela 5 - Resultados do teste ping na rede overlay

	Ping entre B e C	Ping entre A e B
Sem rede overlay	NA	205.267 ± 0.54868 ms
Com rede overlay	50.003 ± 0.27613 ms	212.596 ± 0.40561 ms

Além do teste de *ping* foi realizado ainda um teste de vazão utilizando o utilitário *iperf3*, com dois clientes na mesma rede local acessando simultaneamente o servidor, novamente localizado em nuvem no Japão. Novamente o resultado indica uma sobrecarga pequena da rede *overlay* verificada na taxa de vazão e a perda de pacotes estatisticamente igual entre as versões com e sem o uso da rede *overlay*.

Tabela 6 - Resultados do teste com iperf na rede overlay

	Taxa de transferência	Perda de pacotes
Sem rede overlay	95.6 MBits/sec	0.003811219 %
Com rede overlay	93.8 MBit/sec	0.003865231 %

Os resultados indicaram que a proposta e implementação da rede *overlay* é transparente para a aplicação e com sobrecarga aceitável. Por exemplo, com a rede *overlay* foi

possível realizar o teste de *ping* (icmp echo, que não utiliza um protocolo de transporte) mesmo com um dos nós em uma rede protegida por NAT. Para a interface virtual *tun0* a rede *overlay* é também transparente. A sobrecarga verificada em termos absolutos e percentuais é aceitável. Mesmo sem a realização de testes em escala, os resultados obtidos foram suficientes para um teste em escala integrado ao *framework*, conforme apresnetado no Capítulo 7.

6 ALGORITMOS DE ALOCAÇÃO

A qualidade das decisões de reconfiguração utilizadas pelo orquestrador está diretamente ligada ao desempenho e à escalabilidade das aplicações quando executam no *framework* proposto, como destacado na Seção 4.7. As decisões poderiam privilegiar, por exemplo, o desempenho da comunicação dos módulos-cliente e serviços na nuvem ou o processamento mais rápido. Outra possibilidade seria utilizar procedimentos de otimização que levariam ao melhor balanço de comunicação, processamento e até a economia de energia dos dispositivos. Neste sentido, como mencionado, o *framework* foi estruturado para permitir que vários algoritmos de alocação e escalonamento fossem avaliados. Ao invés de propor um algoritmo de alocação e escalonamento, o que fugiria um pouco dos objetivos, a abordagem adotada foi a de selecionar algoritmos disponíveis. Com isso também demonstramos a flexibilidade do *framework* desenvolvido.

Quatro algoritmos presentes na literatura foram selecionados, tendo como base o survey [41]:

1. ERA [26] apresentado na Seção 6.1, tem como principal parâmetro a latência entre nós. Este algoritmo minimiza a latência média do sistema, mas não leva em consideração, em um primeiro momento, a capacidade de processamento de cada nó. Caso aconteça alguma sobrecarga em um nó as requisições excedentes são redirecionadas para o nó cloud com a menor latência disponível.

2. Small Cell [27] apresentado na Seção 6.2, é um algoritmo guloso que busca alocar os nós de acordo com um único parâmetro. O artigo original define uma versão que prioriza a capacidade de processamento (CS) e outra que prioriza a latência(LAT).

3. GABVMP [28] apresentado na Seção 6.3, é um algoritmo genético que utiliza como base as métricas de latência, capacidade de processamento e custo de processamento de cada nó.

4. GMEDSWC [29] apresentado na Seção 6.4, é um algoritmo de grafos que tenta minimizar o número de nós de processamento utilizado, respeitando a capacidade de processamento de cada nó.

Cada algoritmo foi, primeiramente, estudado e depois adaptado para o *framework*.

Em seguida, foram implementados de acordo com a API apresentada no Código 3 na Seção 4.3 e integrado ao orquestrador, para a avaliação.

As próximas seções apresentam os algoritmos, suas características, vantagens e desvantagens, bem como as adaptações realizadas para sua incorporação ao *framework*.

6.1 ERA

O primeiro algoritmo implementado foi o ERA [26].

Objetivo. Este algoritmo é proposto para realizar alocação de tarefas em uma arquitetura de infraestrutura como serviço (*Infrastructure-as-a-Service*, *IaaS*) com 3 camadas: (i) cloud computing, (ii) camada de rede e (iii) fog computing, conforme apresentado na Figura 34.

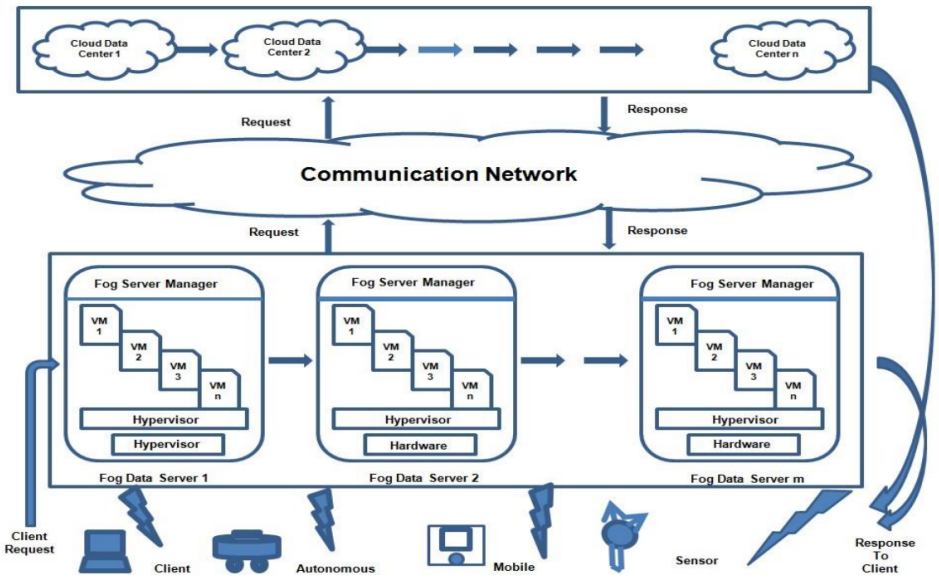


Figura 34 - Arquitetura de 3 camadas utilizada pelo algoritmo ERA
*Figura retirada de [26]

Estrutura. O algoritmo ERA possui, originalmente, sete etapas:

1. Todos os nós da fog são enumerados(FS), assim como os nós do cloud(CS)
2. Cada nó FS contém um gerente (FSM) para coletar as métricas periodicamente.
3. Inicialmente todos as requisições são realizadas à um FS. Cada FS então encaminha para o FSM. O FS é escolhido de acordo com a distância geográfica para o cliente.
4. O FSM tentará atender a requisição. Se for possível, a execução acaba aqui.

5. O usuário pode ser notificado que terá que esperar um certo tempo para a resposta caso o nó da fog esteja sendo inicializado

6. Caso necessário o FSM encaminha o pedido de processamento para o CS.

7. O CS envia um ACK para o FS e atende o pedido do usuário.

Adaptações. É possível notar que a arquitetura base do ERA é similar à utilizada pelo *framework* proposto, o que simplificou a integração deste algoritmo. Entretanto, existem algumas diferenças que fizeram com que o algoritmo tivesse que ser adaptado. Entre elas está o conceito de gerente de fog (*Fog Server Manager*, FSM) que não está presente na nossa arquitetura.

Para contornar as diferenças na arquitetura foi considerado que cada nó de *fog* era também o seu próprio FSM. Algumas das etapas do algoritmo já são realizadas pelo *framework* e então puderam ser removidas. Como por exemplo as etapas 1, 2 e 5. Com isso as etapas 3, 4, 6 e 7 foram adaptadas e o algoritmo final implementado em Python é apresentado no Código 11.

De acordo com o modelo de orquestração do *framework* os agentes são redirecionados para os nós de processamento, não necessariamente cada uma das requisições. Com isso, o algoritmo deixa de realizar a escolha para cada requisição e passa a olhar para a alocação do desses componentes.

Implementação. Entre as linhas 1 e 6 são preparadas as estruturas de dados para armazenar os resultados.

Entre as linhas 7 e 18 é realizada a etapa 3, onde é determinado o melhor FS e o melhor CS para cada agente. Essa foi uma otimização realizada, no algoritmo original apenas o FS seria escolhido na etapa atual (3), entretanto, muitas vezes também é necessário encontrar o CS na etapa 6.

Com os nós FS e CS escolhidos o algoritmo então passa para a próxima etapa (4), onde o FS tenta atender a requisição. Aqui as etapas 4,6 e 7 são realizadas entre as linhas 19 e 28. Primeiro é considerado considerado a capacidade do FS e se possível o agente é alocando para esse nó (etapa 3), caso o nó de processamento não possua capacidade disponível o agente então é alocado para o CS (etapas 6 e 7). Depois que essa escolha é feita para todos os agentes o algoritmo retorna a lista dos nós de processamento utilizados e a alocação dos agentes

```
def ERA(G,A,F,C,delay,cap):
    V=[]#lista de nós utilizados para processar
    rC=defaultdict(lambda: [])#<key,[value]> nó que processa => [agentes]
    setF=set(F)
    setC=set(C)
    best = defaultdict(lambda: {"f":None,"fT":-1,"c":None,"cT":-1})
    for a in A:
        fs = list(set(G.V[a]).intersection(setF))
        cs = list(set(G.V[a]).intersection(setC))
        for f in fs:
            if(best[a]["fT"]==-1 or G.V[a][f]<best[a]["fT"]):
                best[a]["fT"]=G.V[a][f]
                best[a]["f"]=f
        for c in cs:
            if(best[a]["cT"]==-1 or G.V[a][c]<best[a]["cT"]):
                best[a]["cT"]=G.V[a][c]
                best[a]["c"]=c


    lcap = defaultdict(lambda: 0)
    for a in A:
        if(cap[best[a]["f"]]+lcap[best[a]["f"]]>=cap[a]):
            rC[best[a]["f"]].append(a)
            lcap[best[a]["f"]]-=cap[a]
            V.append(best[a]["f"])
        else:
            rC[best[a]["c"]].append(a)
            V.append(best[a]["c"])
    V=list(set(V))
    return V,rC
```

Código 11 - Era

Resultado. Este algoritmo prioriza a escolha do nó de processamento da fog que tenha a menor latência. Se esse nó não tiver capacidade suficiente disponível, o algoritmo escolhe o nó da cloud que tenha a menor latência. Conforme definido pela API o algoritmo retorna a lista dos nós de processamento que foram utilizados e o mapa das atribuições, ou seja, que nó deve processar as requisições de cada agente.

6.2 Small Cell

O artigo [27] apresenta o algoritmo *Small Cell*.

Objetivo. Este algoritmo é proposto para determinar as configurações otimizadas de um *cluster* de *fog computing* no cenário das redes 5G. Esse artigo descreve os componentes das redes 5G denominados de small cell e utiliza os recursos disponíveis para atender as requisições dos usuários. Quando necessário são criados *clusters* de small cell para atender

as requisições remanescentes.

Estrutura. O algoritmo original é descrito com 5 passos divididos em 2 etapas:

1. Alocação dos recursos nas SmallCells (SC)
   (a) Cada SC lista as requisições e ordenas elas de acordo com o parâmetro adotado.
   (b) De acordo com a classificação feita no item anterior, a SC atende o número máximo de requisições dado os recursos disponíveis.
2. Clusters de SmallCells (SCC) são formados.
   (a) Uma lista dos recursos ociosos são enviadas para o gerente das SmallCells
   (b) O mesmo processo de classificação realizado em 1a é realizado novamente com as requisições que não foram atendidas
   (c) SCC são criados de acordo com parâmetros adotados para que as requisições sejam atendidas.

Adaptações. Novamente foram realizados ajustes para que este algoritmo pudesse ser utilizado no *framework* proposto. Como não existem *Small Cells* nesses contexto os nós da fog foram considerados como as SC. Outro ajuste realizado foi referente à segunda etapa do algoritmo, como não seria simples criar outros *clusters* dentro da arquitetura atual, foi considerado que os nós da *cloud* eram o SCC. Desta maneira o passo 2a não existe e o 2c é simplesmente encaminhar a requisição para a nuvem escolhida no passo 2b

Durante as simulações foram utilizados dois parâmetros diferentes para a etapa 1a: Latência entre o cliente e o nó da fog e o Custo computacional total de determinada tarefa. Assim, nos resultados esses diferentes parâmetros serão comparados independentemente.

Implementação. O Código 12 apresenta a implementação em Python. Entre as linhas 1 e 6 as estruturas de dados auxiliares são preparadas. A etapa 1a é realizada entre as linhas 7 e 10, e nesse ponto os agentes são ordenados de acordo com o parâmetro utilizado, latência ou custo computacional. Entre as linhas 11 e 19. Finalmente, os agentes que não foram alocados são encaminhados para os melhores nós do cloud a partir da linha 20

Parâmetros. Conforme apresentado, o desempenho desse algoritmo depende do parâmetro utilizado, no Capítulo 7 esse algoritmo é avaliado com parâmetros diferentes e com isso foram utilizados nomes diferentes para representar cada cenário. Quando o SmallCell foi

```
def smallCellBase(G,A,F,C,delay,cap,param):
    lcap = defaultdict(lambda: 0)
    lista = defaultdict(lambda: [])
    V=set()
    rC=defaultdict(lambda: [])
    aS = set(A)
    for i in F:
        for v in aS.intersection(set(G.V[i])):
            lista[i].append((v,G.V[i][v],cap[v]))
        lista[i]=ordena(lista[i],param)
    for i in F:
        for v in lista[i]:
            if(cap[i]+lcap[i]>=v[2]):
                V.add(i)
                rC[i].append(v[0])
                aS.discard(v[0])
                lcap[i]-=v[2]
                if(cap[i]+lcap[i]<=0):
                    break
    for i in C:
        V.add(i)
    for v in aS:
        best = C[0]
        for c in C:
            if(G.V[c][v]<G.V[best][v]):
                best = c
        rC[best].append(v)
    return list(V),rC
```

Código 12 - SmallCell

avaliado utilizando a latência ele recebeu o nome de LAT, seus resultados serão discutidos na Seção 7.2.2. E quanto ele foi avaliado com o parâmetro de capacidade de processamento de cada nó ele recebeu o nome de CS, seus resultados serão discutidos na Seção 7.2.3.

Resultado. Este algoritmo prioriza a escolha do nó de processamento da fog que tenha o melhor valor do parâmetro utilizado. Se esse nó não tiver capacidade suficiente disponível o algoritmo tenta escolher o melhor nó da fog de acordo com o parâmetro utilizado que tenha capacidade disponível. Se nenhum nó da fog for selecionado o melhor nó da cloud será selecionado. Conforme definido pela API o algoritmo retorna a lista dos nós de processamento que foram utilizados e o mapa das atribuições, ou seja, que nó deve processar as requisições de cada agente. É importante notar que diferentemente do algoritmo ERA, o SmallCell adiciona todos os nós da cloud na lista de nós utilizados.

6.3 GABVMP

O artigo [28] apresenta o algoritmo GABVMP (*Genetic Algorithm Based Virtual Machine Placement*).

Objetivo. Um algoritmo genético para a alocação de recursos. Os algoritmos genéticos foram propostos em [62] e consistem de uma série de técnicas não determinística que simula a evolução para encontrar uma solução para o problema proposto. Para isso, deve ser possível representar cada solução possível através de um simples array ou mesmo de uma string, chamados de cromossomos. Cada cromossomo pode ser avaliado utilizando uma função de *fitness* que irá variar de acordo com o problema a ser solucionado.

Estrutura. A primeira etapa do algoritmo genético consiste em criar a primeira geração dos cromossomos de forma randômica. As novas gerações são criadas com base na geração anterior e o algoritmo é finalizado de acordo com uma das duas condições: (i) um número máximo de gerações é alcançado ou (ii) se um certo valor de *fitness* for alcançado.

Para a criação das novas gerações são aplicadas aos cromossomos disponíveis uma cominação das três operações:

(i) *Reprodução.* Os melhores cromossomos são escolhidos e mantidos para a próxima geração. Essa técnica é chamada de troca por elitismo e apesar de não ser a única opção de Reprodução [63], foi a utilizada originalmente pelo GABVMP;

(ii) *Crossover.* Novos cromossomos são gerados a partir da combinação dos valores de um par de cromossomos. Por exemplo, a partir do cromossomo A e B, são formados os cromossomos C e D. Nesse cenário é comum que C seja formado pela primeira metade de A e a segunda metade de B. Enquanto que, D seria formado pela primeira metade de B e a segunda metade de A;

(iii) *Mutação.* Os cromossomos selecionados podem, com uma probabilidade baixa, sofrer alterações aleatórias, como por exemplo a alteração de um bit no *array* que representa o cromossomo.

O GABVMP utiliza essas técnicas para encontrar a melhor solução para a alocação. Cada cromossomo é representado por um *array* de N posições, sendo N o número de clientes. A i-ésima posição do array contem o valor X e representa que a requisição do cliente i serão processada pelo nó X, podendo X ser um nó da fog ou um nó da nuvem.

Os cromossomos iniciais são escolhidos de forma pseudo-aleatória, mas com a garantia de que o resultado obtido é válido, ou seja, as restrições de capacidades dos nós de processamento são respeitadas. A partir destes cromossomos inciais o algoritmo é executado até 10000 vezes, aplicando as operações supracitadas. Cada cromossomo é avaliado de acordo com o *fitness* obtido através do cálculo apresentado na Equação 1.

$$fitness = \sum_{i=1}^{n} \beta \times \frac{custo_i}{cap_x} + \alpha \times lat_{i_x} \tag{1}$$

Para evitar que soluções inválidas sejam obtidas o valor de *fitness* é substituído por ∞ caso a configuração representada por aquele cromossomo viole as restrições de capacidade de algum nó.

```
1  def GABVMP(G,A,F,C,delay,cap,pop=5,stop=10000):
2      V=[]#lista de nós utilizados para processar
3      rC={}#<key,[value]> nó que processa => [agentes]
4      vms = A
5      pms = F+C
6      chromosomes = initialization(G,vms,pms,cap,C,pop)
7      chromosomes = mysorted(chromosomes,vms,G,cap)
8      count = 0
9      while(((count<stop and chromosomes[0]["fit"]>10))):
10         count+=1
11         aux1 = crossover(chromosomes[0],chromosomes[1])
12         aux2 = mutation (chromosomes[0])
13         for i in range(len(chromosomes)-5):
14             aux2+=mutation(chromosomes[i%2])
15         chromosomes=replacement(chromosomes,aux1,aux2)
16         chromosomes = mysorted(chromosomes,vms,G,cap)
17     lcap = defaultdict(lambda: 0)
18     best = chromosomes[0]
19     V = list(set(best["place"]+C))
20     for v in V:
21         rC[v]=[]
22     for idx,vm in enumerate(best["place"]):
23         if(cap[vm]+lcap[vm]>=cap[idx+1]):
24             rC[vm].append(idx+1)
25             lcap[vm]-=cap[idx+1]
26         else:
27             rC[C[0]].append(idx+1)
28     return V,rC
```

Código 13 - GABVMP

Implementação. O Código 13 apresenta a implementação em python. Na linha 6 é

criada a primeira geração de cromossomos e logo em seguida eles são ordenados de acordo com o *fitness*, na linha 7. Entre as linhas 9 e 16 são criadas todas as novas gerações e podemos ver a aplicação de cada uma das operações: reprodução, crossover e mutação respectivamente nas linhas 15, 11 e 12. Por último entre as linhas 22 e 28 é aplicada a escolha do cromossomo. Como o algoritmo pode acabar pelo número de tentativas, aqui fazemos também a verificação dos recursos, para garantir que caso acabem os recursos da *fog* os agentes serão encaminhados para um nó do *cloud*.

Resultado. Este algoritmo prioriza as soluções que tenham o maior valor de *fitness* de acordo com a Equação 1, que atribui um peso para a relação custo/capacidade (β) e para a latência(α). Caso alguma restrição de capacidade esteja sendo violada, ao final do algoritmo, os agentes que estejam acima do limite estabelecido são redirecionados para um nós da cloud arbitrário. Conforme definido pela API o algoritmo retorna a lista dos nós de processamento que foram utilizados e o mapa das atribuições, ou seja, que nó deve processar as requisições de cada agente. Em relação à lista de nós este algoritmo tem um comportamento similar ao do Small Cells e adiciona todos os nós da cloud.

6.4 GMEDSWC

O artigo [29] apresenta e discute o problema da escolha do posicionamento dos nós de edge em uma rede de área metropolitana sem fio ($MWAN$). O problema intitulado de *Edge Server Placement Problem*(ESPP) é dividido em duas categorias: (i) Sem restrições, onde cada nó do edge pode ser customizados de acordo com a demanda, ou seja, o poder computacional de cada nó de processamento é determinado pelas tarefas que lhe são atribuídas; (ii) Com restrições - neste contexto todos os nós do edge possuem limites pré determinados para o seu poder computacional;

Objetivo O artigo apresenta ainda uma formulação de um problema de programação linear inteira que representa o problema proposto e que pode ser mapeado para cada uma das duas categorias de acordo com as restrições de capacidade. Além disso, são apresentados alguns algoritmos que podem resolver esse problema, entre eles iremos destacar o algoritmo guloso denominado *Greed-based minimum extended dominating set algorithm with capacity constraint* (GMEDSWC)

Estrutura Este algoritmo utiliza uma representação em grafo de toda a rede para criar clusters de pontos de acesso (AP) e então determinar que um desses APs será o servidor

edge que irá atender as requisições, conforme demonstrado pela Figura 35.

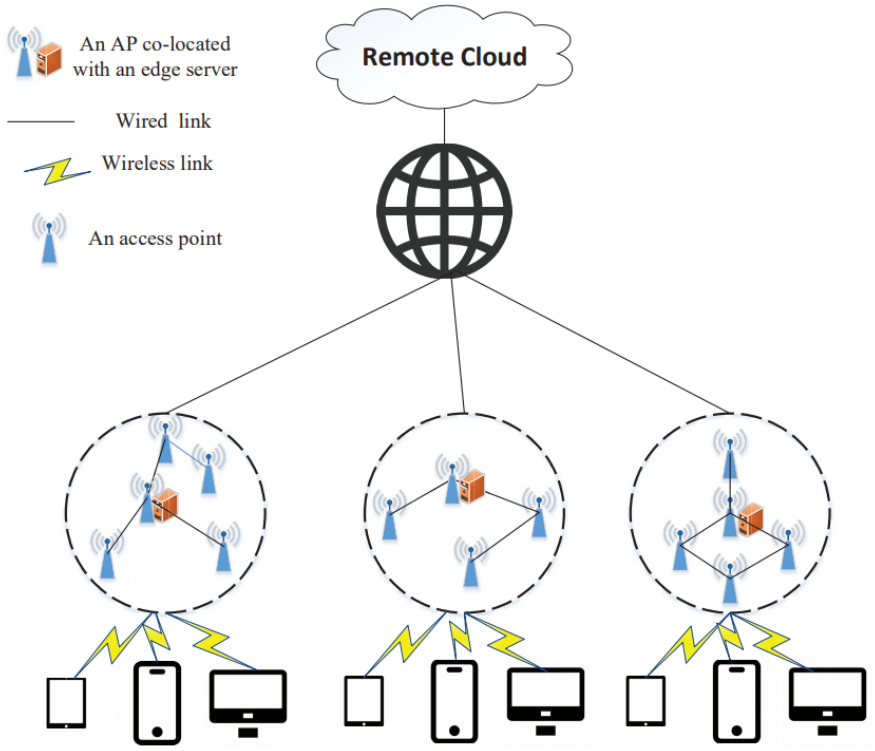


Figura 35 - Solução proposta pelo GMEDSWC
*Figura retirada de [29]

A partir desse grafo, é escolhido o AP que possui o maior número de conexões, vamos chama-lo de S. Neste momento tenta-se criar um cluster contendo S e todos os nós que possuem conexão para S, entretanto devido a limitações no poder de processamento de S é necessário remover alguns dos nós desse cluster. O algoritmo propõe 3 formas de fazer a escolha dos nós que serão retirados: (i) maiores primeiro; (ii) aleatório; (iii) pequenos primeiro. Vamos considerar a primeira abordagem. Realizada essa retirada, teremos um conjunto contendo o nó S e alguns nós que tem conexão com ele, a partir desse conjunto é criado um cluster, ou seja, S passa a ser o servidor edge para esses outros nós. Com isso esses nós que foram escolhidos são retirados do grafo e o processo recomeça até que não tenha mais nenhum nó.

Adaptações Apesar do algoritmo GMEDSWC ter sido originalmente proposto para um contexto diferente ele é facilmente convertido para o nosso caso de uso. Iremos considerar que o grafo é composto apenas dos nós de fog e de cloud, entretanto, iremos considerar os agentes no cálculo dp número de conexões entre os nós. Assim, apenas os nós com poder computacional podem ser escolhidos como candidatos à servidor edge, enquanto

que no momento da escolha dos nós, aqueles que possuem mais conexões continuam sendo beneficiados. Nesse contexto os nós que possuem mais conexões são aqueles que estão visíveis para mais agentes, seja por questões da topologia da rede, ou limitações na latência máxima. Vale destacar que algumas restrições propostas no algoritmo original foram removidas nos testes realizados, já que não se aplicam no nosso contexto.

Implementação. O Código 14 apresenta a implementação em Python da adaptação feita com base no algoritmo GMEDSWC. Entre as linhas 2 e 5 são inicializadas algumas estruturas de dados auxiliares, além de ser realizada a criação do grafo contendo todos os nós de processamento(*Servers*). O loop principal do algoritmo está contido entre as linhas 6 e 31. Entre as linhas 10 e 15 é realizada a primeira etapa, onde o nó com mais conexões é selecionado (S), essa operação é equivalente à escolher o nó com o maior grau. Ainda na linha 15 é realizada a tentativa de criar um cluster com todos os nós que possuem conexão para S, essa operação é equivalente a selecionar todos os vizinhos de S no grafo atual. Então entre as linhas 16 e 18 são removidos os nós que excedem a capacidade do nó S. Conforme comentado entre as linhas 19 e 26, não foram incluidas as limitações relacionadas ao número máximo de conexões que um nó pode ter dentro de um cluster e a restrição do tamanho máximo de um cluster. Finalmente entre as linhas 27 e 31 são removidos do grafo todos os nós selecionados no cluster atual.

Apesar do algoritmo original terminar com essa etapa, foi preciso acrescentar ainda mais uma etapa na versão adaptada. A partir da linha 32 é possível observar que os agentes que não foram atribuídos a nenhum cluster farão parte do cluster do primeiro nó cloud disponível. Essa é apenas mais uma garantia que foi adicionada para que nenhum agente fique sem nó de processamento atribuído.

Resultado. Este algoritmo tenta minimizar o número de nós de processamento utilizados para atender todas as requisições. Novamente, caso alguma restrição de capacidade esteja sendo violada, ao final do algoritmo, os agentes que estejam acima do limite estabelecido são redirecionados para um nós da cloud arbitrário. Conforme definido pela API o algoritmo retorna a lista dos nós de processamento que foram utilizados e o mapa das atribuições, ou seja, que nó deve processar as requisições de cada agente.

```
def GMEDSWC(G,A,F,Cl,delay,cap,Dq=0,Sq=0):
    U = set(A)
    Servers = F+Cl
    C = defaultdict(lambda: [])
    D= []
    while len(U)>0 and len(Servers)>0:
        us = set(U)
        s=Servers[0]
        degS= extDeg(s,delay,G.V[s],us)
        for v in Servers:
            degV=extDeg(v,delay,G.V[s],us,degS)

            if degV > degS:
                s=v
                degS=degV
        S={'head':s,'nodes':extNei(s,delay,G.V[s],us)}

        if len(S['nodes'])>cap[s]:
            ordena(S,G)
            S['nodes']=S['nodes'][0:cap[s]]

         '''
        Deveria remover todos os nós que possuem deg()>Dq,
        Mas essa limitação não existe na nossa rede
        '''

        '''
        Deveria limitar o número de nós <Sq,
        Mas essa limitação não existe na nossa rede
        '''
        C[s]=S['nodes']
        D.append(s)
        Servers.remove(s)
        for v in S['nodes']:
            U.remove(v)
    if len(U)>0:
        if Cl[0] not in D:
            D.append(Cl[0])
        C[Cl[0]]+=U
    return D,C
```

Código 14 - GMEDSWC

6.5 Discussão

Nas seções anteriores foram apresentados os quatro algoritmos selecionados. Para cada um dos algoritmos foram discutidos o objetivo, o cenário original para o qual foram propostos, e, em contraponto, quais adaptações tiveram que ser realizadas e por fim a implementação proposta.

As adaptações e implementações propostas obrigatoriamente aderem à interface apresentada no Código 3, Seção 4.3. Assim, cada algoritmo recebe como parâmetro uma representação em grafo do estado atual da rede, onde cada nó é um agente ou um nó de processamento, e as arestas são as latências entre os nós. Além disso, são fornecidas também as listas dos agentes, dos nós da *fog* e dos nós de *cloud*, bem como o *delay* máximo permitido, e a lista com os custos/capacidades de cada agente/nó de processamento.

No próximo capítulo serão apresentados os testes realizados com cada um desses algoritmos.

7 AVALIAÇÃO DE DESEMPENHO

Neste capítulo são apresentadas as avaliações de desempenho realizadas sobre a implementação de referência do *framework*.

Inicialmente, a Seção 7.1 apresenta a estrutura da rede utilizada nas avaliações: os nós de processamento que compunham o *cluster*, composto por nós de processamento onde os elementos do SiteWhere e do *framework* seriam executados, e as características dos clientes e agentes utilizados nos testes. Também discutimos a seleção dos nós para os quais são exibidos os resultados da avaliação, e as métricas utilizadas.

Para avaliar o *framework* desenvolvido foram realizados dois conjuntos de testes. No primeiro conjunto, cada um dos algoritmos apresentados no Capítulo 6 foi testado com um conjunto de clientes e agentes. Estes são implantados estrategicamente para explorar cenários iniciais de uso de recursos (CPU e memória), e de latência para as instâncias de serviços, e como as sugestões de reconfiguração de cada algoritmo para uso de outras instâncias de serviço afetariam o desempenho da aplicação.

O resultado de cada algoritmo é avaliado pela métrica da média da latência de cada elemento da aplicação à instância do serviço do SiteWhere alocado, comparado com o melhor resultado possível no cenário e com a instância de serviço disponível no nó de processamento de *Cloud* mais conveniente (Seção 7.2).

Com base nestes resultados, o algoritmo ERA, apresentado na 6.1, foi escolhido para a realização do segundo conjunto de testes: escalabilidade. Neste conjunto de testes a mesma infraestrutura configurada para executar o *framework* no primeiro conjunto foi utilizada para executar testes com 10, 20, 200, 500, 1000 e 10000 clientes conectados simultaneamente. A métrica da latência média também foi utilizada para a avaliação, mas também são apresentados e discutidos resultados sobre o uso de CPU e memória nos nós de processamento do *cluster* de suporte ao *framework*. Ainda, são observadas como a orquestração "movimentou" as conexões entre clientes/agentes e as instâncias de serviço disponíveis, Seção 7.3.

Por último, na Seção 7.4 são discutidos os resultados encontrados. Além disso, discutimos as limitações encontradas no SiteWhere bem como os pontos de melhoria que podem ser explorados em uma próxima versão do *framework*.

7.1 Visão geral

Para a realização dos testes o *cluster* Docker que dá suporte ao *framework* foi configurado com três grupos distintos de nós de processamento, a fim de representar a *Cloud*, o *Fog* e a *Edge*.

Os nós de processamento da *Cloud* foram criados utilizando a infraestrutura do *Google Cloud Platform* (GCP) em 3 regiões geográficas distintas, para que pudéssemos melhor avaliar o impacto da distância e número de hops (e correspondente latência) nos resultados e nas escolhas dos algoritmos. As regiões escolhidas foram: *southamerica-east1* (São Paulo, BR), *us-west2* (Los Angeles, USA) e *asia-northeast1* (Tóquio, JP). Todas as máquinas utilizadas foram configuradas com IPs reais e foram realizadas as configurações necessárias para a abertura das portas, para que fosse possível acessar os módulos do SiteWhere diretamente. Para materializar estes nós, foram utilizadas instâncias genéricas do *Compute Engine* da GCP, onde apenas o Docker e a rede *overlay* foram instalados.

Para representar os nós de *Fog* foi utilizada a infraestrutura da UERJ, provida pelo Laboratório de Ciência da Computação (LCC). Foram utilizadas ao todo três máquinas distintas, sendo duas máquinas virtuais e uma máquina física. Estas também foram configuradas com endereços IP reais e preparadas para que o SiteWhere fosse acessado diretamente. As máquinas virtuais foram configuradas especificamente para os testes com apenas o Docker e a rede *overlay* instalados. Apesar da máquina física possuir outros sistemas de software instalados, durante os testes apenas o Docker e a rede *overlay* seriam executados.

Por último, para representar um elemento de *Edge* foi utilizado um dispositivo Raspberry Pi 3B [64]. Novamente, este nó de processamento foi configurado especialmente para os testes e possuía apenas o Docker e a rede *overlay* instalados. Diferente dos outros nós de processamento, este nó não possuía um IP real, já que estava atrás de um servidor NAT. Nenhum tipo de configuração especial foi realizada no servidor NAT.

As especificações de cada um dos nós de processamentos são apresentadas na Tabela 7. O *host* GCP BR seria responsável por manter a infraestrutura do SiteWhere e o banco de dados principal, logo precisava de maior capacidade de memória e armazenamento, comparado com os outros *hosts* de *Cloud*. Os outros *hosts* GCP utilizavam uma das menores configurações de máquina disponíveis. Em relação aos *hosts* instalados na UERJ, vale destacar que o *host* UERJ 1 foi implantado com 51 GB de memória RAM, maior que

as demais, para identificar se algum algoritmo iria privilegiar a métrica de memória.

A Figura 36 e a Figura 37 apresentam respectivamente a topologia dos nós de processamento e seus endereços na rede *overlay*. Observa-se que a segmentação da rede em camadas deixa de existir dentro da rede *overlay*, ou seja, os nós pertencem virtualmente à mesma subrede, e estão no mesmo domínio de *broadcast*, embora as características particulares de latência permaneçam. Em outras palavras, para as aplicações sendo executadas sobre a infraestrutura do Docker ficam transparentes os detalhes da rede física.

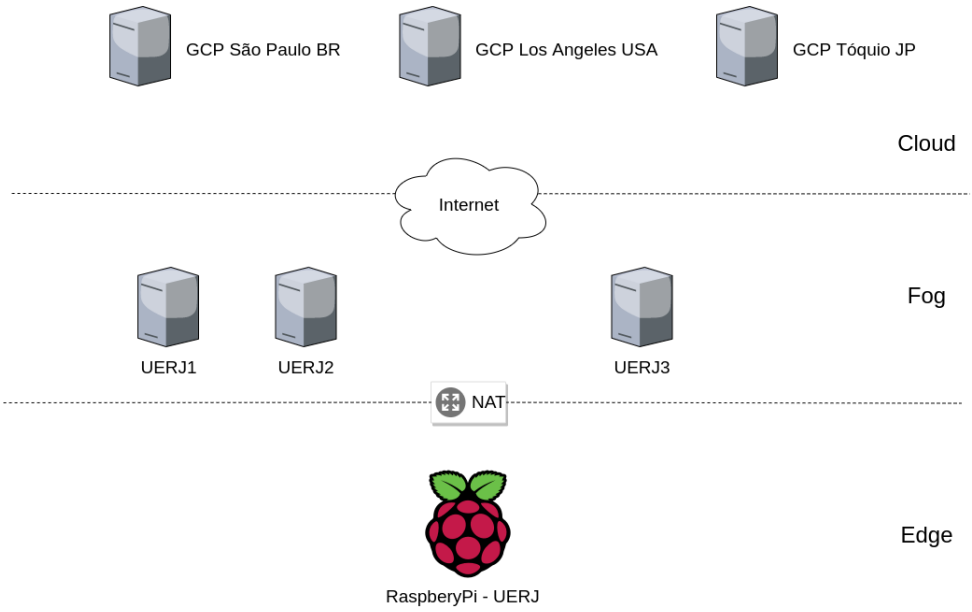


Figura 36 - Topologia dos nós de processamento

Tabela 7 - Especificações dos nós de processamento

Host	Processador	Memória	Armazenamento	Sistema operacional
GCP BR	4 vCPU	20 GB	100 GB	Ubuntu 16.04
GCP USA	1 vCPU	1,7 GB	10 GB	Ubuntu 16.04
GCP JP	1 vCPU	1,7 GB	10 GB	Ubuntu 16.04
UERJ1	4 vCPU	51 GB	100 GB	Ubuntu 16.04
UERJ2	2 vCPU	16 GB	100 GB	Ubuntu 16.04
UERJ3	Intel i7-4790	16 GB	455 GB	Ubuntu 16.04
RaspberyPi	Broadcom BCM2837	1 GB	16 GB	Raspbian 8

Para a realização dos testes foi proposta uma aplicação simples com dois elementos, que chamamos *Clientes* e *Agentes*, com as seguintes funções:

(i) **Cliente**. Simula um usuário comum que se conecta ao SiteWhere utilizando a API RestFul para fazer requisições aos dados históricos coletados pelos sensores e realizar operações com os atuadores.

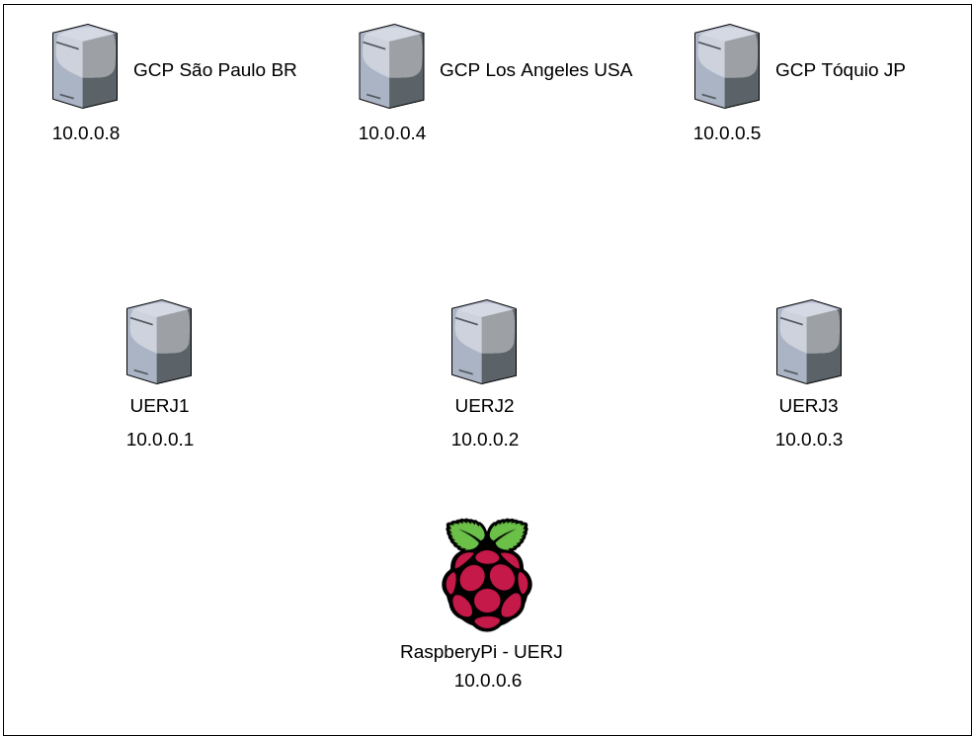


Figura 37 - Topologia dentro da rede overlay

(ii) **Agentes**. Agentes de software que simulam sensores e atuadores reais. Conectam-se ao SiteWhere utilizando o protocolo MQTT para gerar dados ou para receber comandos de atuação.

Clientes e Agentes foram implementados em Java e podem ser executados diretamente na linha de comando ou através de uma imagem contida num instância de contêiner Docker.

Os Agentes são agentes SiteWhere simples e utilizam apenas as APIs disponibilizadas pelo *middleware*. Sendo assim, eles não conhecem e não utilizam nenhuma API do *framework* proposto. Já os Clientes, estes podem conhecer e utilizar as APIs do *framework*, como discutido na Seção 4.3, além das APIs do SiteWhere.

Periodicamente o *framework* coleta informações de operação, utilizando as APIs apresentadas na Seção 4.3, e armazena estas informações no banco de dados representando, assim, o contexto atual do sistema, como apresentado na Seção 4.5. O contexto atual do sistema é então utilizado pelos algoritmos para determinar uma reorganização de uso dos serviços. O *framework* então utiliza este resultado para aplicar a nova alocação Cliente/Agente-serviço, formando uma nova topologia, fechando assim o ciclo onde: (i) as métricas são coletadas; (ii) as métricas são consolidadas; (iii) o algoritmo de alocação

é executado com as métricas consolidadas; (iv) a nova alocação é aplicada.

A periodicidade da coleta do informações de operação para se obter uma "fotografia" atual do contexto do sistema, como se dá em sistemas de monitoramento, pode ser personalizada. Para a realização dos testes, o *framework* foi configurado para realizar esta coleta a cada segundo.

Durante os testes, as seguintes informações de operação foram coletadas:

- uso de CPU dos nós de processamento;
- uso de memória RAM dos nós de processamento;
- latência (RTT) entre os Clientes/Agentes e os nós de processamento.

Os resultados apresentados nas próximas seções foram compilados a partir dos dados de contexto armazenados no banco de dados, a cada segundo, durante rodadas de execução dos testes. Este conjunto de dados, na sua forma "bruta", foi tratado inicialmente removendo-se os pontos referentes à inicialização do sistema. Em seguida, para cada um dos cenários foram selecionados 15 minutos de dados, a partir do ponto onde todos os elementos já estavam ativados. Cada valor exibido nos gráficos deste capítulo representa a média destes 15 minutos, com intervalo de confiança de 95%.

Todas as modificações apresentadas na Seção 3.2.3 foram mantidas durante os testes. Além disso, o *cluster* Docker criado, interconectado com a rede *overlay* apresentada no Capítulo 5, foi o mesmo utilizado nos dois grupos de testes.

7.2 Comparativo dos algoritmos

Para este conjunto de testes foram utilizados quatro pares de Clientes/Agentes distribuídos segundo a Tabela 8, oito elementos no total. Cada par representa um cenário de operação de interesse para aplicações IoT. O primeiro par Cliente/Agente (C1,A1), implantado nos EUA, tem como objetivo representar clientes e agentes que têm como melhor nó de processamento um nó da *Cloud*. O próximo par Cliente/Agente (C2,A2), foi implantado no Brasil em uma rede sem NAT, com objetivo de testar se os algoritmos iriam conectá-los ao nó de *Fog*. O terceiro par Cliente/Agente (C3,A3), foi implantado na Austrália a fim de testar se os algoritmos iriam escolher o nó de processamento com base na latência ou na capacidade de processamento. Por último, o par Cliente/Agente

C4 e A4, foi implantado no Brasil em uma rede com NAT, com o objetivo de testar se os algoritmos iriam aloca-los no nó de *Edge* localizado na mesma rede.

A Tabela 8 apresenta ainda, para cada par Cliente/Agente, o "Nó Cloud padrão". Este seria o nó de processamento de *Cloud* com menor latência possível e com recursos suficientes para executar os serviços do *framework* e SiteWhere necessários para o Cliente/Agente. O objetivo de usar este nó para comparação é que ele seria o "último recurso", ou seja, entre os nós de processamento de *Cloud*, que na maioria das aplicações estaria localizado "longe", ou seja, com latência alta este seria a melhor escolha. Este nó foi determinado após a analise dos dados obtidos e foi utilizado como "pior caso, ainda adequado". Alguns dos algoritmos avaliados utilizam o conceito de um nó como este justamente como alternativa de "último recurso" (Capítulo 6).

Tabela 8 - Agentes e Clientes utilizados no teste 1

Cliente/Agente	Localização	Nó Cloud padrão
Cliente 1 (C1)	US	US
Agente 1 (A1)	US	US
Cliente 2 (C2)	BR	BR
Agente 2 (A2)	BR	BR
Cliente 3 (C3)	AU	JP
Agente 3 (A3)	AU	JP
Cliente 4 (C4)	BR	BR
Agente 4 (A4)	BR	BR

A Figura Figura 38 ilustra a configuração inicial de conexão dos Cliente e Agentes aos serviços do SiteWhere. Inicialmente, todos os nós acessam o nó GCP BR. Nas seções que apresentam os testes de cada algoritmo a configuração final é apresentada para comparação.

Antes da realização de cada teste, todos os nós de processamento eram (re)criados e instalados apenas o Docker e a rede *overlay*. O *cluster* apresentado na Seção 7.1 era recriado e o *framework* inicializado. O *framework*, por sua vez era reconfigurado para utilizar cada algoritmo sendo testado. Ao mesmo tempo os quatro pares Cliente/Agente eram inicializados.

Cada rodada de teste foi executado por uma hora, com Clientes e Agentes executando suas rotinas. Ao fim, os nós, os Agentes e os Clientes eram desativados.

Cada algoritmo avaliado utiliza as informações de operação do contexto atual segundo sua politica específica. Como resultado, espera-se que, para cada Cliente ou Agente,

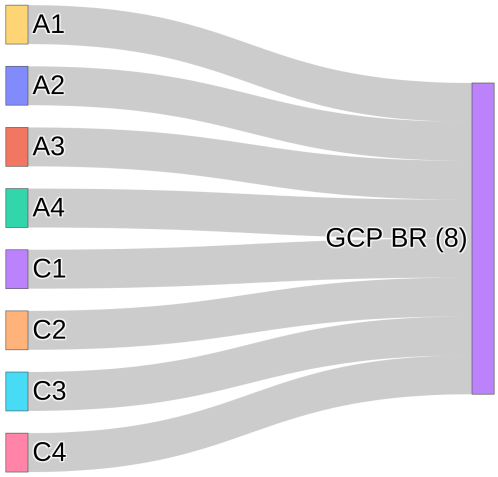


Figura 38 - Configuração inicial de conexão aos serviços

seja feita uma recomendação de qual nó usar, dentre os nós de processamento disponíveis, sejam estes de *Cloud*, *Fog* ou *Edge*. Dependendo do algoritmo, esta recomendação pode priorizar nós com mais recursos de CPU e memória disponíveis, parâmetro chamado de "capacidade de processamento" por vários dos algoritmos (Capítulo 6), ou a latência. Entretanto, a métrica de comparação de interesse em nosso trabalho é a latência. O objetivo era verificar qual dos algoritmos levariam a uma configuração da topologia de acesso ao serviços que obtivesse a menor latência média. Com isso, era esperado que com um volume maior de Clientes e Agentes a aplicação seria mais escalável, uma vez que o tempo de acesso aos serviços seria menor. Por outro lado, consideramos que, dentro de suas politicas, cada algoritmo só recomendaria o uso de nós de processamento com recursos suficientes para atender todas as requisições em tempo aceitável. Por isso, não usamos as informações de CPU e memória como métricas de avaliação. Isso foi também reforçado com o cenário proposto para este conjunto de teste, onde o número de Clientes e Agentes não representaria um gargalo, ainda que, por recomendação do algoritmo de alocação, todos acessassem os serviços executando no mesmo nós de processamento.

Para a comparação dos algoritmos são apresentados, para cada um dos Clientes e Agentes os valores das seguintes medidas de latência:

- a média da latência do nó escolhido pelo algoritmo;

- a latência para o *Nó Cloud padrão*, como sendo o pior caso;
- a menor latência possível, calculada posteriormente com base nos dados coletados durante os testes, representada como "Ótimo" nos gráficos.

7.2.1 ERA

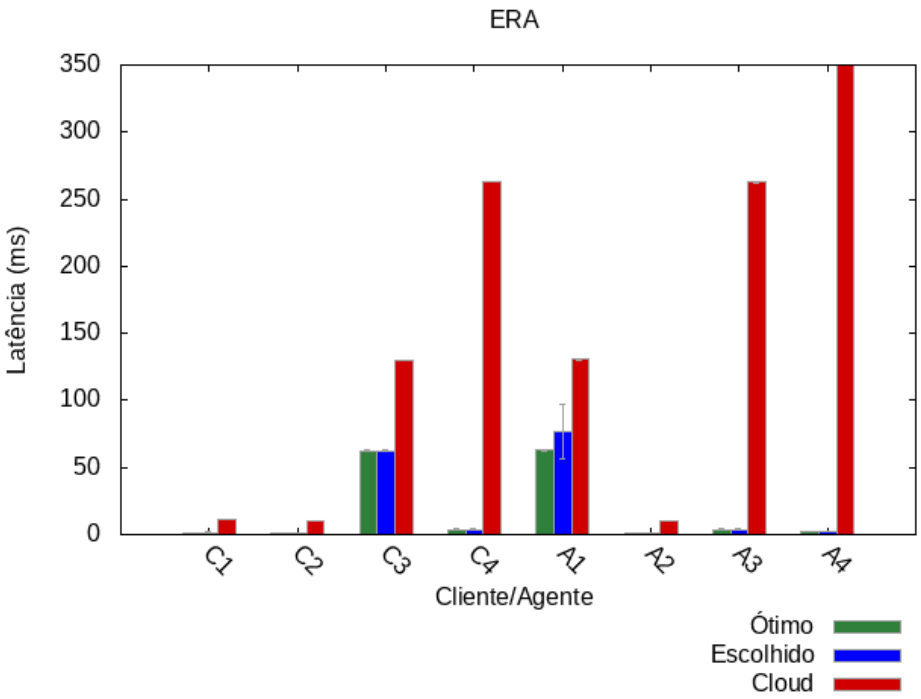


Figura 39 - Resultados ERA

O algoritmo ERA apresentou soluções para seleção dos serviços próximas do ideal para todos os Clientes e para a maioria dos Agentes, exceção ao A1 (Figura 39). O Agente A1, ficou oscilando entre o uso do melhor servidor e um outro nó de processamento. Como a alocação para o servidor com menor latência não era estável a cada execução do algoritmo o A1 trocava de nó de processamento. Isso explica a grande variação no valor da latência, o que afetou o intervalo de confiança.

É importante destacar que em nenhum momento as escolhas apresentadas por esse algoritmo foram piores do que os respectivos *Nós Cloud padrão*.

7.2.2 LAT

O algoritmo Small Cell com o parâmetro de latência, LAT, apresentou, para alguns Agentes/Clientes o melhor resultado possível (Figura 41).

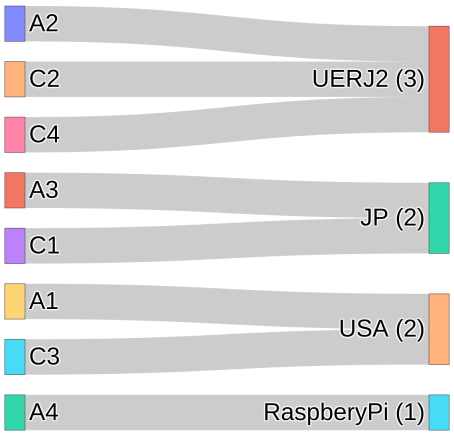


Figura 40 - Configuração final ERA

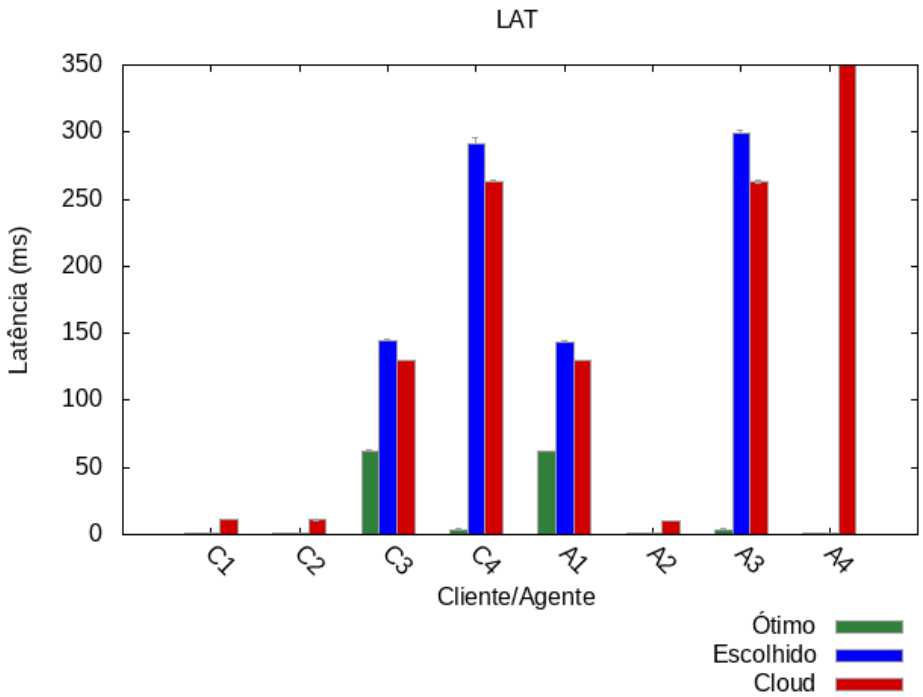


Figura 41 - Resultados LAT

Entretanto, vários Agentes/Clientes foram recomendados nós de processamento de *Cloud* piores do que o escolhido como parâmetro de comparação (*Nós Cloud padrão*). Isso pode ser observado no C3, C4, A1 e A3.

Considerando que os nós de processamento da *Cloud* selecionados possuíam recursos suficiente para atender as requisições desses Agentes/Clientes as escolhas desse algoritmo não foram muito boas para a aplicação.

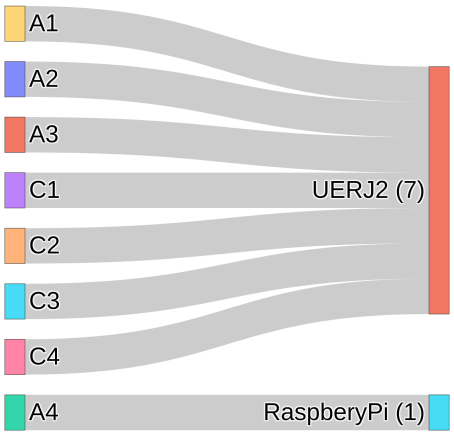


Figura 42 - Configuração final LAT

7.2.3 CS

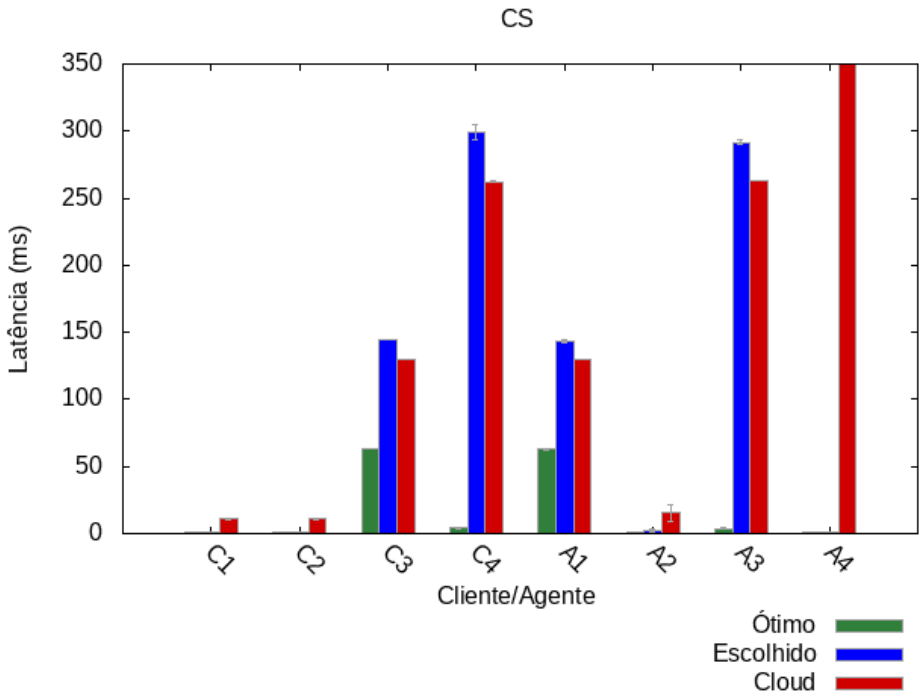


Figura 43 - Resultados CS

O algoritmo Small Cell com o parâmetro de capacidade de processamento, CS, também apresentou, para alguns pares Agente/Cliente o melhor resultado possível (Figura 41).

Entretanto, de forma similar ao apresentado na Seção 7.2.2 o CS também apresen-

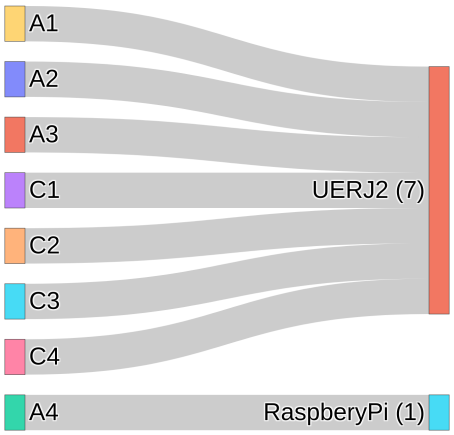


Figura 44 - Configuração final CS

tou resultados piores do que o *Nó Cloud padrão*. Isso pode ser observado novamente para os elementos C3, C4, A1 e A3.

7.2.4 GABVMP

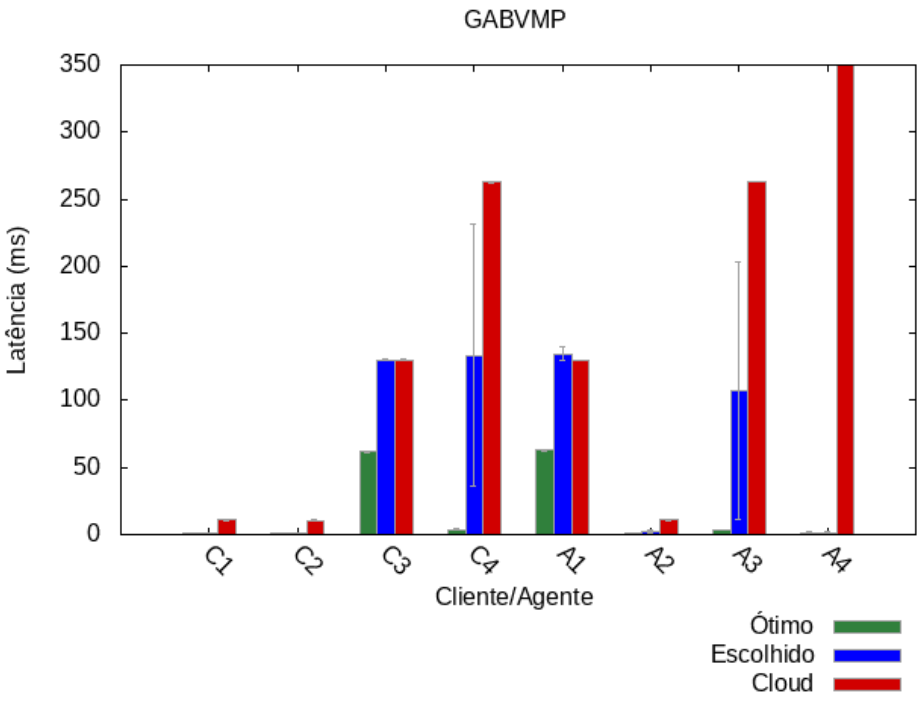


Figura 45 - Resultados GABVMP

O algoritmo GABVMP mostrou um comportamento muito próximo do ideal, próximo

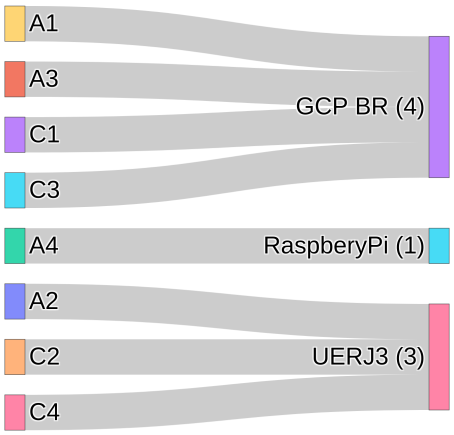


Figura 46 - Configuração final GABVMP

da melhor seleção possível, para diversos pares Agente/Cliente (Figura 45).

Vale destacar que a latência do nó sugerido pelo algoritmo ficou abaixo do Nó CLoud padrão para todos os elementos, com exceção do A1. Além disso, para C4 e A3 esse algoritmo apresentou resultados com grande desvio padrão apontando escolha de nó com constantes mudanças.

7.2.5 GMEDSWC

Conforme apresentado na Seção 6.4, o GMEDSWC emprega um algoritmo genético. Seu desempenho depende do número de gerações que foram simuladas. Como estamos em um cenário onde o tempo de execução do algoritmo é um fator limitante não é possível obter o número de gerações necessários para se ter um resultado consistentemente. Devido a está restrição o GMEDSWC não apresentou um bom desempenho.

Pelos resultados apresentados na Figura 47, a alocação escolhida por este algoritmo leva a médias de latência bem próximas ao melhor valor possível para alguns Agentes/Clientes (C3,A2 e A4), entretanto, para os outros elementos apresenta um resultado pior do que o *Nó Cloud padrão*.

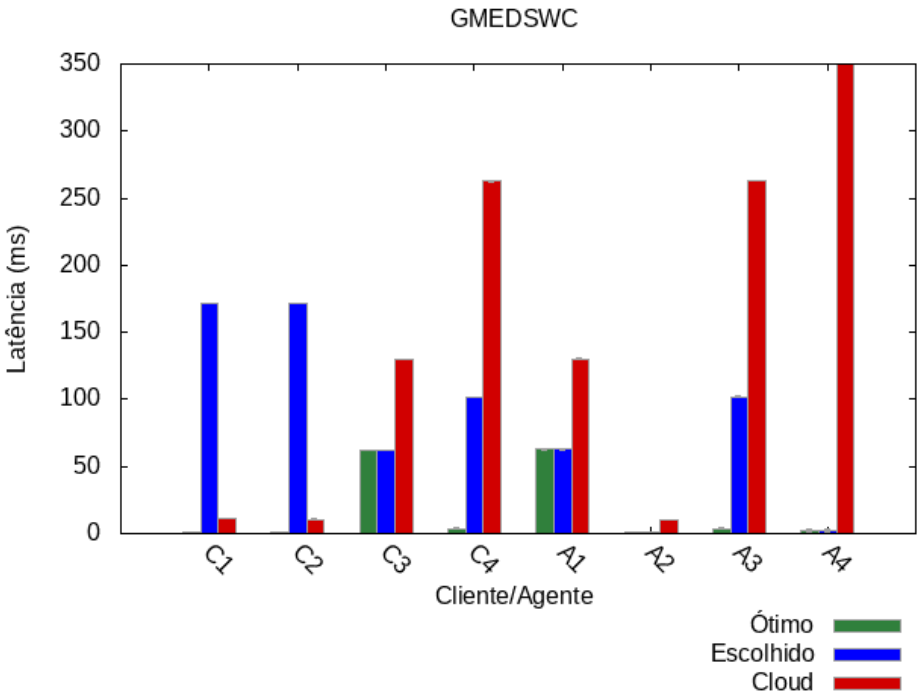


Figura 47 - Resultados GMEDSWC

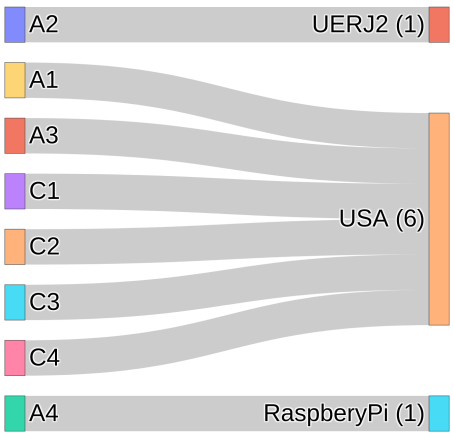


Figura 48 - Configuração final GMEDSWC

7.2.6 Discussão

Com base nos resultados obtidos, verifica-se que o algoritmo ERA apresentou resultados que combinam menores latências e mais consistentes (menores desvios padrão) na escolha dos nós de processamento para os quais Clientes e Agentes deveriam enviar suas requisições de serviço.

Assim, esse algoritmo foi utilizado na realização do segundo conjunto de testes, que tem seus resultados apresentados na Seção 7.3.

7.3 Escalabilidade

O objetivo do segundo conjunto de testes era avaliar a escalabilidade da solução proposta. Para tal o mesmo *cluster* apresentado na Seção 7.1 foi submetido à execução da mesma rotina descrita na Seção 7.2, agora variando-se o número de Clientes e Agentes a partir de 10, 20, 200, 500, 1.000 até 10.000.

Para o teste de escalabilidade os resultados são apresentados como a média da latência para todos os Agentes e Clientes, sem destacar elementos da aplicação específicos de interesse. Também foi necessário avaliar como a orquestração do *framework* afetou o uso de CPU e memória nos nós de processamento do *cluster*.

7.3.1 Preparo para instanciar 10 mil

Antes de apresentar os resultados do teste de escalabilidade devem ser feitas algumas considerações sobre o preparo e a execução do teste.

Em um primeiro momento para este teste seriam utilizadas as imagens Docker dos Clientes/Agentes Java para instanciar todos as 10 mil instâncias necessárias, como se deu no primeiro conjunto. Entretanto, foi constatado que essa abordagem não funcionaria. Foram encontrados diversos problemas, desde limitações no *kernel* do Linux para número máximo de interfaces de rede virtuais que podem ser criadas paralelamente (recurso utilizado pelo Docker para isolar cada contêiner) até limites de núcleos de CPU e memória RAM impostos pela GCP para a conta utilizada (em dado momento foram utilizados mais de 30 vCPUs e 400+GB de memória ao total).

Para contornar esses problemas empregamos uma outra abordagem: utilizamos diretamente Agentes e Clientes Java, sem a sobrecarga do Docker. Com isso, foi necessário gerenciar todos as 10 mil instâncias manualmente. Esta troca não invalida os resultados apresentados na seção anterior, dado que o *framework* e todos os serviços oferecidos pelo SiteWhere são os únicos e continuam sendo executados sobre a infraestrutura do Docker e a rede *overlay* proposta.

Essa troca abordagem foi suficiente para instanciar todos os 10 mil elementos

clientes/agentes necessários utilizando menos recursos, ainda que com o custo extra do controle manual. Ao todo foram utilizadas 4 máquinas auxiliares com um total de 16 vCPUs e 64 GB de memória para executar os clientes e agentes.

Num segundo momento, ao realizar um primeiro teste de conexão, foi constatado que não seria possível conectarem-se mais de 2500 Clientes/Agentes ao mesmo tempo utilizando a API MQTT do SiteWhere, que se tornava instável e impedia a realização dos testes. Por isso os testes de escalabilidade foram realizados utilizando apenas os Clientes. Lembrando que os Agentes utilizam a API MQTT do SiteWhere e os Clientes utilizam a API Restful. Essa mudança também não invalida os resultados obtidos anteriormente já que os mesmos elementos do *framework* foram utilizados nos dois cenários. Como a API MQTT faz parte do SiteWhere, não foi investigado se seria possível alterar a estrutura interna do *middelware* para resolver este problema.

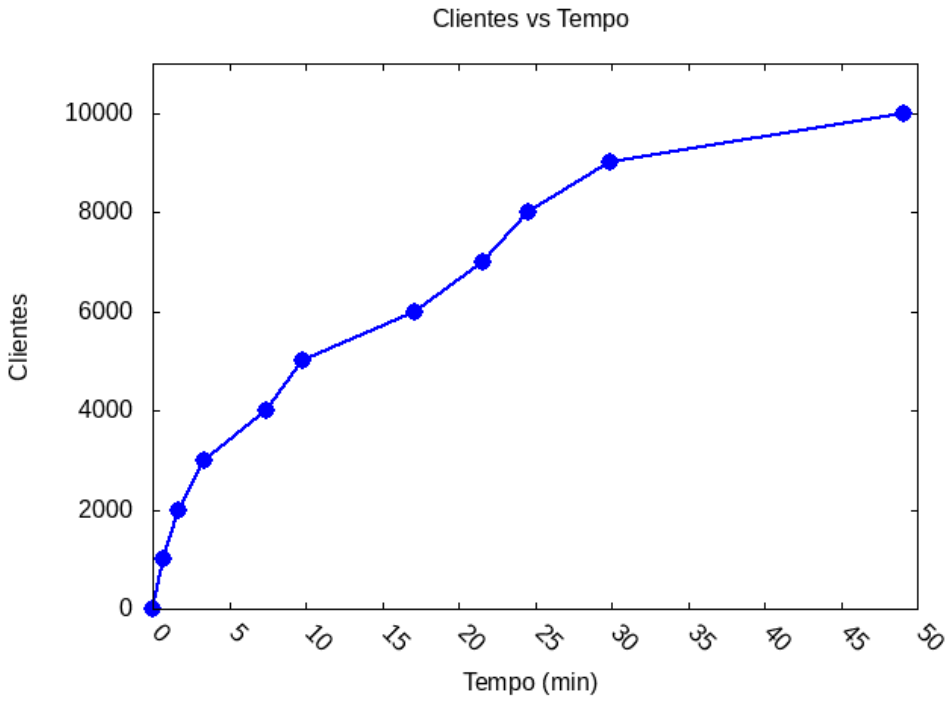


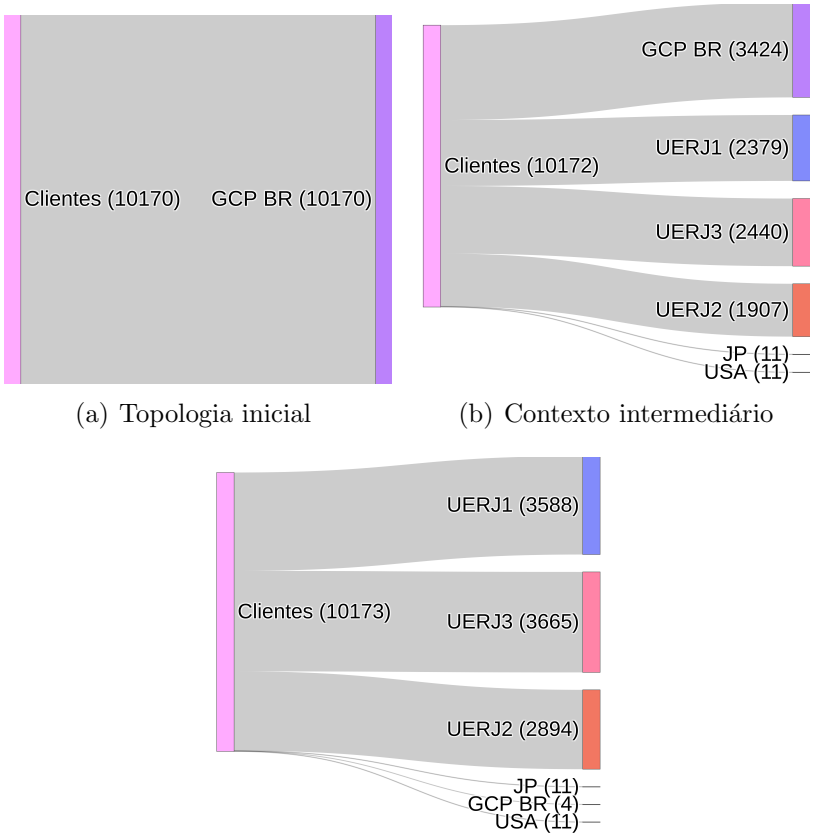
Figura 49 - Tempo de inicialização dos 10 mil clientes

Por último, ainda foi encontrado um desafio em relação à inicialização e conexão de todos os 10 mil clientes. Conforme comentado na Seção 7.1 em todos os testes foram descartadas as medidas realizadas durante a inicialização dos testes. Isso representa, no primeiro conjunto de testes, na média, alguns segundos de dados até que todos os sistemas sejam iniciados e todos os Clientes e Agentes estejam conectados e sejam reconhecidos pelo SiteWhere e pelo *framework*. Entretanto, no segundo cenário de testes, na instância

com 10 mil Clientes esse tempo de inicialização foi de aproximadamente 50 minutos. A Figura 49 apresenta a evolução do tempo de inicialização para os testes realizados com o número crescente de elementos da aplicação. Por isso, nos testes de escalabilidade foram utilizados 60 minutos de inicialização, mais 60 minutos de execução dos testes. A mesma metodologia foi utilizada para selecionar os 15 minutos de dados que são apresentados como médias nos gráficos da próxima seção.

7.3.2 Resultados

A Figura 50 destaca a atuação do algoritmo ERA na orquestração dos 10 mil Clientes. Na configuração inicial todos os clientes utilizavam o serviços do nó *GCP BR* (Figura 50(a)). Com a evolução dos ciclos de orquestração, o uso dos serviços pelos Clientes foi se distribuído por outros nós de serviço, privilegiando a diminuição da latência (Figura 50(b)). No final dos testes esta distribuição estava consolidada (Figura 50(c)).



(a) Topologia inicial (b) Contexto intermediário

(c) Topologia final

Figura 50 - Evolução da topologia de acesso dos Clientes

Observa-se que a alocação intermediária não era estável e portanto foi substituída

pela alocação final, isto está relacionado ao consumo de CPU do nó de processamento GCP BR, que será discutido adiante na Figura 53.

O primeiro resultado apresentado na Figura 51 e na Figura 52 mostra que houve um impacto mínimo na média de utilização de CPU e memória nos nós de processamento do *cluster* durante os testes. O *framework* com o SiteWhere, Docker e com a rede *overlay* consumiu em média entre 40% e 70% de CPU. É possível observar, inclusive, que a utilização de memória parece ter diminuído quando o sistema estava com 10 mil clientes conectados.

Os nós considerados nesse gráfico são os nós de processamento do *cluster*, apresentados na Tabela 7 da Seção 7.1.

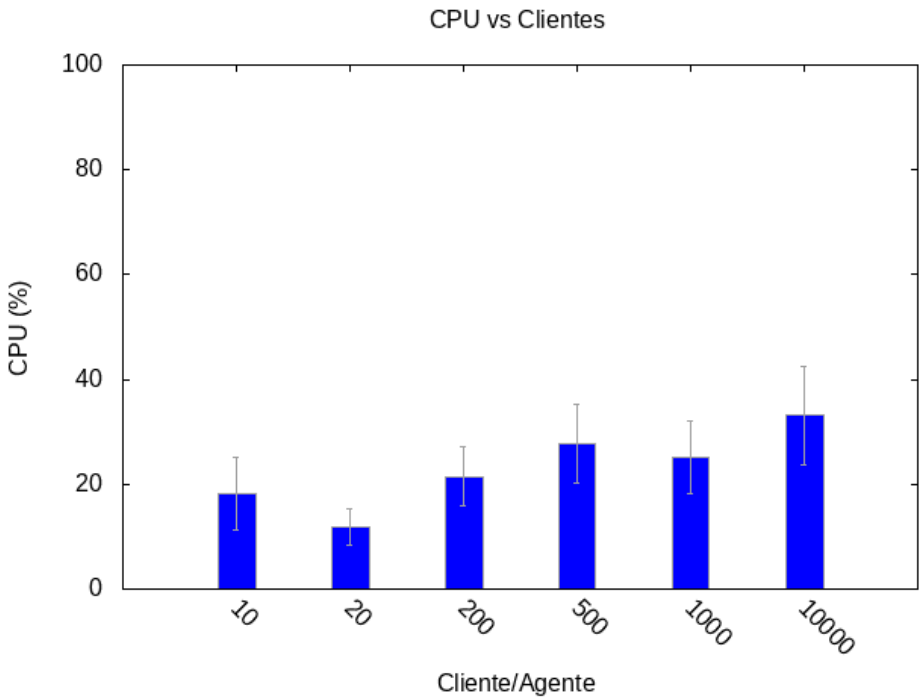


Figura 51 - CPU vs Clientes

Este comportamento, a princípio estranho, pode ser explicado se olharmos para a Figura 53 e Figura 54. Nessas figuras foram destacadas as médias do uso de CPU e memória, respectivamente, do servidor selecionado para executar a infraestrutura do SiteWhere. Com isso observa-se o nó de processamento principal estava estava sobrecarregado, já que este servidor estava com 99% de utilização de CPU.

É importante notar que apesar de todas as requisições realizadas pelos Clientes terem sido atendidas/concluídas com sucesso, ficou constatado que em alguns momentos o algoritmo de orquestração (ERA) foi executado com dados desatualizados, visto que os

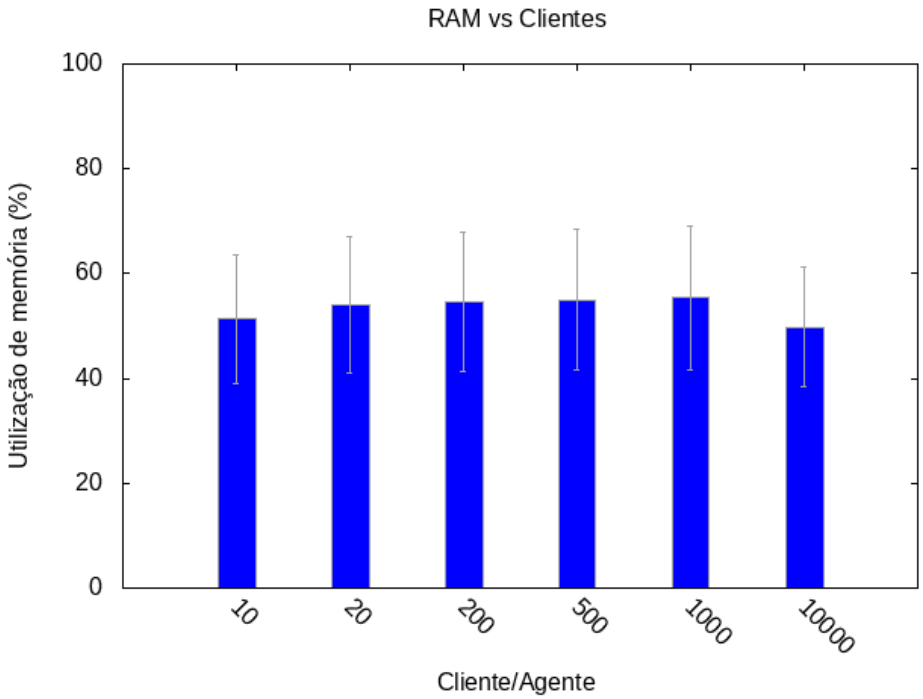


Figura 52 - RAM vs Clientes

dados mais recentes ainda não tinham sido processados e disponibilizados.

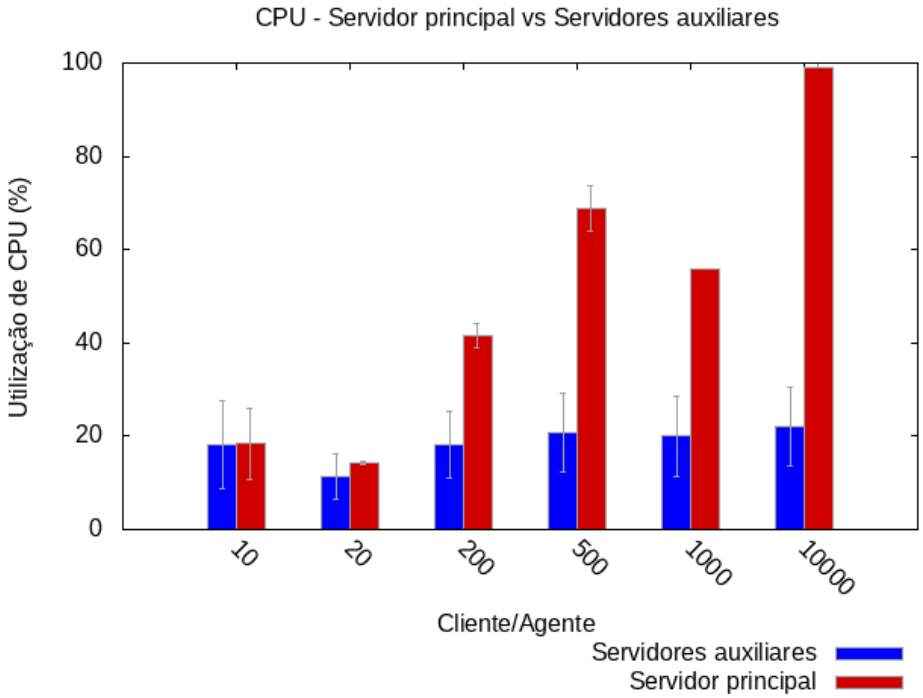


Figura 53 - CPU - Servidor principal vs Servidores auxiliares

Por último a Figura 55 apresenta a latência média das requisições realizadas pelos Clientes de acordo com o número total de clientes conectados no sistema. Foi constatado

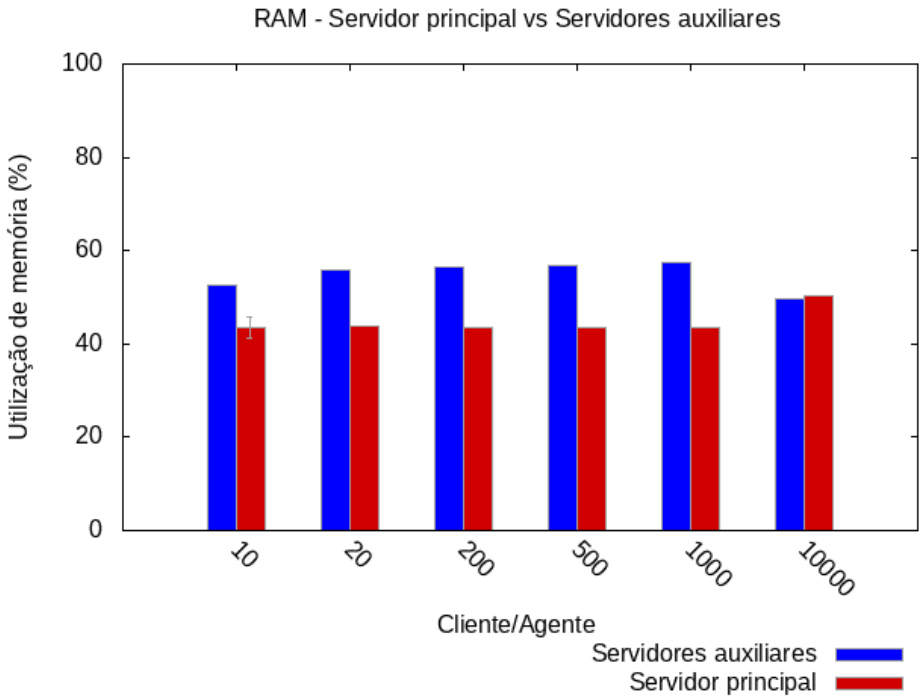


Figura 54 - RAM - Servidor principal vs Servidores auxiliares

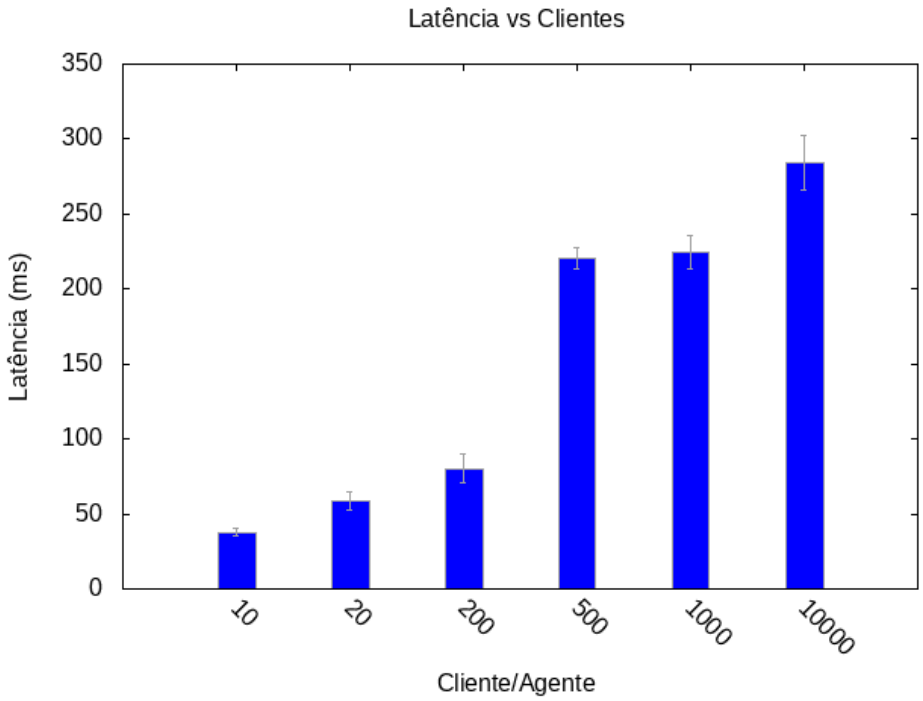


Figura 55 - Latência vs Clientes

que o *framework* foi capaz de lidar com 10000 Clientes simultâneos.

Os resultados para 10, 20 e 200 Clientes tem uma média de latência menor que 100 ms, um resultado considerado muito bom. Para 500 e 1000 Clientes o sistema apresenta uma latência média de aproximadamente 215 ms. Esse resultado mostra que houve uma

degradação do sistema se comparado com os testes com menos Clientes. Essa degradação pode ser explicada pela alta carga no banco de dados utilizado.

Por último, para 10.000 Clientes, apesar de termos 10 vezes o número de Clientes conectados, se comparado com o resultado com 1.000 Clientes, houve um crescimento aproximado de apenas 25% na latência média ($265.97 \pm 36,08$ ms). Esse resultado é muito bom se considerada a quantidade de informação sendo processada pelo *framework* e por todos os componentes da infraestrutura.

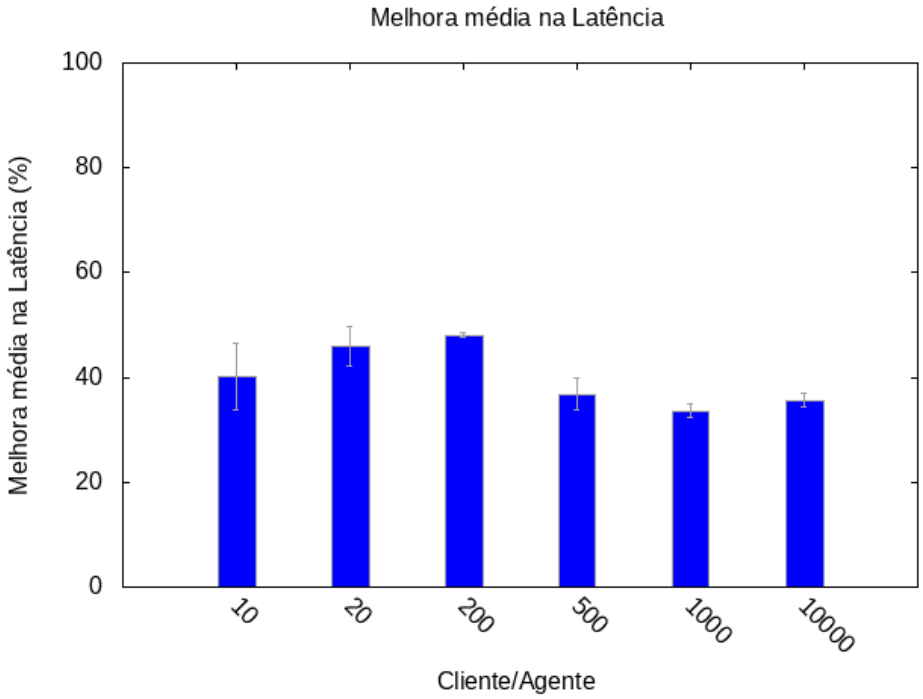


Figura 56 - Comparação da latência sem e com o framework

O gráfico da Figura 56 apresenta uma comparação percentual entre a latência média na configuração inicial dos Clientes e na configuração final, mostrando a melhora percentual (diminuição da latência).

7.4 Discussão Geral

Como resultado direto do primeiro conjunto de testes, foi constatado que o *framework* é extensível e que pode ser utilizado com diferentes algoritmos, dado que eles sejam implementados utilizando a API apresentada em 4.3. Além disso, ficou demonstrado que o algoritmo ERA apresenta os melhores resultados considerando a métrica de latência, quando comparado com os outros algoritmos analisados.

Com a escolha do ERA, foi realizado um teste de escalabilidade. Esse teste mostra que o *framework* é capaz de gerenciar 10 mil clientes simultâneos, com baixo impacto na latência, principalmente se comparado com os resultados encontrados no artigo de referência [50].

A análise dos resultados aponta também pontos de melhoria, bem como as limitações no *middleware* IoT escolhido. Por exemplo, não foi possível estabelecer a conexão de mais de 2500 Clientes/Agentes simultâneos utilizando a API MQTT do SiteWhere. O que fez com que o teste de escalabilidade fosse feito exclusivamente com os clientes que utilizavam a API RESTful.

Além dessa limitação, foi possível constatar através dos resultados obtidos, que houve uma sobrecarga no servidor principal, ou seja, aquele que oferecia os serviços de infraestrutura necessários para a execução do SiteWhere. Isso poderia ser minimizado através da distribuição desses serviços entre os nós de processamento da *Cloud*. Para tal seria necessário realizar as configurações necessárias para a criação do *swarm* do Docker e os serviços do SiteWhere teriam que ser modificados para se tornarem compatíveis.

O último ponto de melhoria evidente diz respeito à maneira como o *framework* armazena os dados das métricas coletadas. Conforme comentado na Seção 7.3.2 em determinados ciclos o ERA chegou a ser executado com dados desatualizados, visto que os dados mais recentes ainda não haviam sido processados e disponibilizados. Isso indica que o armazenamento destes dados precisa ser otimizado e seria necessário realizar outros testes para determinar se a limitação encontra-se no *framework* proposto ou no banco de dados utilizado (Consul).

Contudo, todas as requisições realizadas pelos 10 mil clientes foram atendidas e a orquestração desses 10 mil clientes foi realizada sem problemas. Isso demonstra que o *framework* é capaz de lidar com 10 mil clientes, mesmo que um cenário extremo como esse não seja recomendado.

CONCLUSÃO

O *framework* proposto utiliza algoritmos de alocação para orquestrar a conexão entre clientes, agentes e serviços de uma aplicação IoT. Para isso o *framework* monitora os recursos de CPU e memória de uma lista de nós de processamento, onde os serviços estão disponíveis e a latência de cada cliente e agente ao nó de processamento ao qual está correntemente conectado para usar o serviço. O conjunto destas informações forma o contexto da aplicação, que é submetido ao algoritmo de alocação. O resultado do algoritmo pode sugerir mudanças de configuração, selecionando outro nó de processamento, mais apto, ou seja com menor latência e capacidade adequada, para cada cliente. Em seguida o *framework* atua redesenhando a malha de conexão dos clientes aos serviços sugeridos.

Esta solução permite tratar o uso de nós de processamento de *edge*, *fog* e *cloud* sem restrições para fazer o *offloading*. Vários dos algoritmos de alocação estudados recebem como entrada apenas as informações de contexto para decidir a seleção dos nós a serem utilizados por cada cliente. Por outro lado, alguns dos algoritmos aceitam a informação do tipo de cada nó de processamento e podem dar prioridade a determinado tipo, por exemplo privilegiando nós de *fog*. Em qualquer caso, o *framework* orquestra a malha de conexões entre clientes, agentes e serviços de forma transparente, abstraindo a aplicação e o *middleware* utilizado da preocupação de vencer os procedimentos de localização, autenticação, credenciamento e estabelecimento de novas conexões.

Uma implementação de referência foi desenvolvida, utilizando microsserviços implementados em Python e encapsulados em imagens Docker. A implementação do *framework* emprega ainda o SiteWhere, um sistema de *middleware* IoT, para prover os serviços necessários para uma aplicação IoT. O *framework* é capaz de realizar todas as operações necessárias para a orquestração sem que a implementação dos clientes e agentes do SiteWhere seja alterada. Assim, a aplicação pode usufruir das vantagens do *framework* sem alterações de código, mantedo-se a compatibilidade de aplicações já desenvolvidas.

O *framework* foi projetado para ser extensível, conforme discutido na Seção 6, e com isso diversos algoritmos de alocação disponíveis na literatura foram integrados à implementação de referência. Um conjunto de testes de desempenho, empregando Clientes e Agentes com características distintas foi realizado com cada algoritmo. De acordo com os resultados apresentados na Seção 7.2.6 o algoritmo de alocação ERA [26] foi escolhido

como o melhor dentre os testados por apresentar a métrica de latência com resultados melhores e mais estáveis.

Também foi realizado um conjunto de testes para avaliar a escalabilidade do *framework*. Nestes testes a implementação de referência do *framework* foi utilizada em um cenário com 10, 20, 200, 500, 1000 e 10000 Clientes simultâneos, utilizando o algoritmo de alocação ERA para orquestrar todos estes Clientes cuja rotina era realizaar operações periódicas requisitando serviços do SiteWhere. Conforme apresentado na Seção 7.3.2, o *framework* apresentou um resultado muito bom. Quando observado o consumo de CPU e memória nos nós de processamento foi constatado que o impacto causado pelo crescimento no número de usuários foi baixo. Além disso, quando observado a latência média percebida pelos Clientes, foi possível constatar que o sistema se manteve com um tempo de resposta muito bom, tendo respostas abaixo de 100 ms para até 200 Clientes e respostas abaixo de 350 ms para até 10000 Clientes.

Além disso, durante os testes preliminares do *framework* foram encontradas limitações na rede formada pelo Docker e na maneira como ela lida com elementos implantados em redes redes protegidas por NAT. Para resolver estas limitações foi desenvolvida uma rede *overlay* que utiliza UDP Hole Punching e permite que os nós que fazem parte dessa rede possam se comunicar como se fizessem parte da mesma rede local.

Com a utilização do *framework* em conjunto com a rede *overlay* todos os detalhes da infraestrutura, tanto de rede quanto dos nós de processamento, ficam transparente para o *middleware* e seus Clientes. Já para os algoritmos de alocação, ficam disponível tanto as informações de contexto coletadas, quanto os detalhes necessários da infraestrutura.

Como prosseguimento deste trabalho seria interessante estudar questões não abordadas e superar as limitações da proposta. Por exemplo, aspectos de segurança associadas ao *framework*, bem como à rede *overlay* desenvolvida deveriam ser abordados. Outro aspecto interessante seria incorporar informações de consumo de energia dos recursos do sistema ao contexto a ser passado como entrada aos algoritmos e alocação e que estas novas informações se refletissem nos resultados sugerindo o uso de serviços em outros nós de processamento. Na versão atual essas métricas não são coletadas, entretanto poderiam facilmente serem acrescentadas.

Existem pontos de aprimoramento relacionados à implementação. Durante os testes foi constatado que em alguns casos o SiteWhere impunha limites à escalabilidade

das aplicações, por exemplo o número de requisições concorrentes para um mesmo *host*. Estas limitações podem ser tratadas com pequenas mudanças nas soluções práticas adotadas e ajustes no uso do SiteWhere. Por exemplo, seria possível ter os clientes utilizando diretamente a API do *framework* contornando as limitações do SiteWhere. Também seria possível realizar pequenas modificações na arquitetura do sistema, como por exemplo com a adição de balanceadores de carga ou através de mudanças na distribuição dos serviços pelos nós Gerentes. Da mesma forma, os benefícios de outras tecnologias poderiam ser avaliados para o *framework*, como por exemplo o uso de Kubernets.

Além disso, a implementação atual apenas orquestra o uso de serviços em nós de processamento previamente configurados e disponíveis. Uma possibilidade para trabalhos futuros seria realizar a integração de APIs das plataformas de computação em nuvem, para que o *framework* fosse capaz de ativar e desativar nós de processamento, bem como instanciar e migrar serviços de acordo com a demanda atual da aplicação e a disponibilidade de recursos de CPU, memória, ainda minimizando a latência média do sistema e o consumo de energia.

REFERÊNCIAS

[1] SUNDMAEKER, H. et al. *Vision and Challenges for Realising the Internet of things*. The address of the publisher, 3 2010.

[2] EVANS, D. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, v. 1, n. 2011, p. 1–11, 2011.

[3] PöTTER, H. B.; SZTAJNBERG, A. Adapting heterogeneous devices into an iot context-aware infrastructure. In: *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. New York, NY, USA: Association for Computing Machinery, 2016. (SEAMS '16), p. 64–74. ISBN 9781450341875. Disponível em: <https://doi.org/10.1145/2897053.2897072>.

[4] SADEK, R. A. Hybrid energy aware clustered protocol for iot heterogeneous network. *Future Computing and Informatics Journal*, v. 3, n. 2, p. 166 – 177, 2018. ISSN 2314-7288. Disponível em: <http://www.sciencedirect.com/science/article/pii/S2314728818300163>.

[5] Kulik, V.; Kirichek, R. The heterogeneous gateways in the industrial internet of things. In: *2018 10th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. [S.l.: s.n.], 2018. p. 1–5.

[6] SZTAJNBERG, A.; MACEDO, R.; STUTZEL, M. Protocolos de aplicação para a internet das coisas: conceitos e aspectos práticos. In: _____. [S.l.: s.n.], 2018. p. 99–148. ISBN 9786587003085.

[7] MACEDO, R. da S. *Um estudo e avaliação do desempenho de Protocolos de Aplicação para a Internet das Coisas*. Dissertação (Mestrado) — Programa de Pós-Graduação em Ciências Computacionais, UERJ, 2019.

[8] Fysarakis, K. et al. Which iot protocol? comparing standardized approaches over a common m2m application. In: *2016 IEEE Global Communications Conference (GLOBECOM)*. [S.l.: s.n.], 2016. p. 1–7.

[9] Samuel, S. S. I. A review of connectivity challenges in iot-smart home. In: *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*. [S.l.: s.n.], 2016. p. 1–4.

[10] YUN, M.; YUXIN, B. Research on the architecture and key technology of internet of things (iot) applied on smart grid. In: IEEE. *2010 International Conference on Advances in Energy Engineering*. [S.l.], 2010. p. 69–72.

[11] SOARES, C. A.; AMARAL, J. L. Posimnet-r: Uma ferramenta de apoio a projeto de redes sem fio resilientes para automação industrial. *Anais da Sociedade Brasileira de Automática*, v. 1, n. 1, 2019.

[12] MACH, P.; BECVAR, Z. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, IEEE, v. 19, n. 3, p. 1628–1656, 2017.

[13] CHEN, X. et al. Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM Transactions on Networking*, IEEE, v. 24, n. 5, p. 2795–2808, 2015.

[14] MONTEIRO, A. F.; AZEVEDO, M. V.; SZTAJNBERG, A. Virtualized web server cluster self-configuration to optimize resource and power use. *Journal of Systems and Software*, Elsevier, v. 86, n. 11, p. 2779–2796, 2013.

[15] VALENCIA, J.; BOERES, C.; REBELLO, V. E. Combining vm preemption schemes to improve vertical memory elasticity scheduling in clouds. In: IEEE. *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. [S.l.], 2018. p. 53–62.

[16] MONTEIRO, A.; LOQUES, O. Quantum virtual machine: power and performance management in virtualized web servers clusters. *Cluster Computing*, Springer, v. 22, n. 1, p. 205–221, 2019.

[17] KUSIC, D. et al. Power and performance management of virtualized computing environments via lookahead control. *Cluster computing*, Springer, v. 12, n. 1, p. 1–15, 2009.

[18] BARHAM, P. et al. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, ACM New York, NY, USA, v. 37, n. 5, p. 164–177, 2003.

[19] SOLTESZ, S. et al. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys european conference on computer systems 2007*. [S.l.: s.n.], 2007. p. 275–287.

[20] CELESTI, A. et al. Exploring container virtualization in iot clouds. In: IEEE. *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*. [S.l.], 2016. p. 1–6.

[21] YOUSEFPOUR, A. et al. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, Elsevier, 2019.

[22] YI, S.; LI, C.; LI, Q. A survey of fog computing: Concepts, applications and issues. In: *Proceedings of the 2015 Workshop on Mobile Big Data*. New York, NY, USA: ACM, 2015. (Mobidata '15), p. 37–42. ISBN 978-1-4503-3524-9. Disponível em: <http://doi.acm.org/10.1145/2757384.2757397>.

[23] ZACHARIAH, T. et al. The internet of things has a gateway problem. In: ACM. *Proceedings of the 16th international workshop on mobile computing systems and applications*. [S.l.], 2015. p. 27–32.

[24] POSTSCAPES. 2018. Web page. `https://www.postscapes.com/iot-gateways/`, Acessado em 8 de agosto de 2018.

[25] Ojo, M.; Adami, D.; Giordano, S. A sdn-iot architecture with nfv implementation. In: *2016 IEEE Globecom Workshops (GC Wkshps)*. [S.l.: s.n.], 2016. p. 1–6.

[26] AGARWAL, S.; YADAV, S.; YADAV, A. K. An efficient architecture and algorithm for resource provisioning in fog computing. *International Journal of Information Engineering and Electronic Business*, Modern Education and Computer Science Press, v. 8, n. 1, p. 48, 2016.

[27] OUEIS, J.; STRINATI, E. C.; BARBAROSSA, S. The fog balancing: Load distribution for small cell cloud computing. In: IEEE. *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*. [S.l.], 2015. p. 1–6.

[28] AKINTOYE, S. B.; BAGULA, A. Improving quality-of-service in cloud/fog computing through efficient resource allocation. *Sensors*, Multidisciplinary Digital Publishing Institute, v. 19, n. 6, p. 1267, 2019.

[29] ZENG, F. et al. Cost-effective edge server placement in wireless metropolitan area networks. *Sensors*, Multidisciplinary Digital Publishing Institute, v. 19, n. 1, p. 32, 2019.

[30] DOCKER, i. *Docker Documentation*. 2013. `https://docs.docker.com`, Acessado em 13 de janeiro de 2019.

[31] ARDUINO. *Arduino Official Site*. 2005. `https://www.arduino.cc/`, Acessado em 13 de janeiro de 2019.

[32] Jiang, L. et al. An iot-oriented data storage framework in cloud computing platform. *IEEE Transactions on Industrial Informatics*, v. 10, n. 2, p. 1443–1451, 2014.

[33] Baraka, K. et al. Low cost arduino/android-based energy-efficient home automation system with smart task scheduling. In: *2013 Fifth International Conference on Computational Intelligence, Communication Systems and Networks*. [S.l.: s.n.], 2013. p. 296–301.

[34] KIM, S.-M.; CHOI, Y.; SUH, J. Applications of the open-source hardware arduino platform in the mining industry: A review. *Applied Sciences*, Multidisciplinary Digital Publishing Institute, v. 10, n. 14, p. 5018, 2020.

[35] FERDOUSH, S.; LI, X. Wireless sensor network system design using raspberry pi and arduino for environmental monitoring applications. *Procedia Computer Science*, Elsevier, v. 34, p. 103–110, 2014.

[36] CHEN, H.; JIA, X.; LI, H. A brief introduction to iot gateway. In: IET. *IET international conference on communication technology and application (ICCTA 2011)*. [S.l.], 2011. p. 610–613.

[37] SHI, W. et al. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, IEEE, v. 3, n. 5, p. 637–646, 2016.

[38] BONOMI, F. et al. Fog computing and its role in the internet of things. In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. New York, NY, USA: ACM, 2012. (MCC '12), p. 13–16. ISBN 978-1-4503-1519-7. Disponível em: <http://doi.acm.org/10.1145/2342509.2342513>.

[39] XU, J.; CHEN, L.; ZHOU, P. Joint service caching and task offloading for mobile edge computing in dense networks. *arXiv preprint arXiv:1801.05868*, 2018.

[40] WANG, L. et al. Service entity placement for social virtual reality applications in edge computing. In: *Proc. INFOCOM*. [S.l.: s.n.], 2018.

[41] Mouradian, C. et al. A comprehensive survey on fog computing: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, v. 20, n. 1, p. 416–464, Firstquarter 2018. ISSN 1553-877X.

[42] Mukherjee, M.; Shu, L.; Wang, D. Survey of fog computing: Fundamental, network applications, and research challenges. *IEEE Communications Surveys Tutorials*, v. 20, n. 3, p. 1826–1857, thirdquarter 2018. ISSN 1553-877X.

[43] ZENG, D. et al. Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system. *IEEE Transactions on Computers*, IEEE, v. 65, n. 12, p. 3702–3712, 2016.

[44] CHERRUEAU, R.-A. et al. Edge computing resource management system: a critical building block! initiating the debate via openstack. In: *The USENIX Workshop on Hot Topics in Edge Computing (HotEdge'18)*. [S.l.: s.n.], 2018.

[45] SKARLAT, O. et al. Resource provisioning for iot services in the fog. In: IEEE. *2016 IEEE 9th Conference on Service-Oriented Computing and Applications (SOCA)*. [S.l.], 2016. p. 32–39.

[46] DENG, R. et al. Optimal workload allocation in fog-cloud computing toward balanced delay and power consumption. *IEEE Internet of Things Journal*, IEEE, v. 3, n. 6, p. 1171–1181, 2016.

[47] Ismail, B. I. et al. Evaluation of docker as edge computing platform. In: *2015 IEEE Conference on Open Systems (ICOS)*. [S.l.: s.n.], 2015. p. 130–135.

[48] MAHESHWARI, S. et al. Scalability and performance evaluation of edge cloud systems for latency constrained applications. In: IEEE. *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. [S.l.], 2018. p. 286–299.

[49] BELLAVISTA, P.; ZANNI, A. Feasibility of fog computing deployment based on docker containerization over raspberrypi. In: *Proceedings of the 18th international conference on distributed computing and networking*. [S.l.: s.n.], 2017. p. 1–10.

[50] CRUZ, M. A. da et al. Performance evaluation of iot middleware. *Journal of Network and Computer Applications*, v. 109, p. 53 – 65, 2018. ISSN 1084-8045. doi:10.1016/j.jnca.2018.02.013. Disponível em: <http://www.sciencedirect.com/science/article/pii/S108480451830064X>.

[51] ISMAIL, A. A.; HAMZA, H. S.; KOTB, A. M. Performance evaluation of open source iot platforms. In: IEEE. *2018 IEEE Global Conference on Internet of Things (GCIoT)*. [S.l.], 2018. p. 1–5.

[52] YALM - YAML Ain't Markup Language. 2001. Acessado em: 25/03/2019. Disponível em: <https://yaml.org/>.

[53] SITEWHERE. *Sitewhere - The Open Platform for the Internet of Things*. 2019. Acessado em: 17/03/2019. Disponível em: <https://sitewhere.io/en/>.

[54] RIEHLE, D. *Framework design: A role modeling approach*. Tese (Doutorado) — ETH Zurich, 2000.

[55] STUTZEL, M. *Imagens Docker criadas*. 2019. `https://hub.docker.com/u/matheusstutzel`, Acessado em 24 de junho de 2019.

[56] MAHALINGAM, e. a. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. *The Internet Engineering Task Force*, 2014. Disponível em: <https://tools.ietf.org/html/rfc7348>.

[57] VXLAN. Acessado em: 25/03/2019. Disponível em: <https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2012_2/nvgre/vxlan.html>.

[58] DONENFELD, J. A. *Wireguard*. 2015. Acessado em: 25/03/2019. Disponível em: <URL: https://www.wireguard.com/>.

[59] DAVIE ED., J. G. B. A Stateless Transport Tunneling Protocol for Network Virtualization (STT). *The Internet Engineering Task Force*, 2012. Disponível em: <https://tools.ietf.org/html/draft-davie-stt-01>.

[60] KRASNYANSKY, M.; YEVMENKIN, M. *Universal TUN/TAP device driver*. 2007. Acessado em: 25/03/2019. Disponível em: <URL: http://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/Documentation/networking/tuntap.txt>.

[61] FORD, B.; SRISURESH, P.; KEGEL, D. Peer-to-peer communication across network address translators. In: *USENIX Annual Technical Conference, General Track*. [S.l.: s.n.], 2005. p. 179–192.

[62] HOLLAND, J. H. et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. [S.l.]: MIT press, 1992.

[63] THIERENS, D.; GOLDBERG, D. Convergence models of genetic algorithm selection schemes. In: SPRINGER. *International Conference on Parallel Problem Solving from Nature*. [S.l.], 1994. p. 119–129.

[64] RASPBERRYPI. 2017. Web page. Disponível em: < https://www.raspberrypi.org/>.