UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Uma arquitetura de micro-Paas em *fog computing*: Orquestração baseada em container para aplicações IoT utilizando single board computer

Walter do Espírito Santo

SÃO CRISTÓVÃO/SE
2019

UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Walter do Espírito Santo

Uma arquitetura de micro-PaaS em *fog computing*: Orquestração baseada em container para aplicações IoT utilizando single board computer

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal do Sergipe (UFS) como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Admilson de Ribamar Lima Ribeiro
Coorientador: Prof. Dr. Rubens de Souza Matos Júnior

SÃO CRISTÓVÃO/SE

2019

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL
UNIVERSIDADE FEDERAL DE SERGIPE

Santo, Walter do Espírito
S237a Uma arquitetura de micro-PaaS em *fog computing*: orquestração baseada em container para aplicações IoT utilizando single board computer / Walter do Espirito Santo ; orientador Admilson de Ribamar Lima Ribeiro . - São Cristóvão, 2019.
139 f. : il.

Dissertação (mestrado em Ciência da Computação) – Universidade Federal de Sergipe, 2019.

1. Computação. 2. Computação em nuvem. 3. Cluster (Sistema de computador). 4. Internet das coisas. I. Ribeiro, Admilson de Ribamar Lima orient. II. Título.

CDU 004

Walter do Espírito Santo

Uma arquitetura de micro-PaaS em *fog computing*: Orquestração baseada em container para aplicações IoT utilizando single board computer

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal do Sergipe (UFS) como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

Prof. Dr. Admilson de Ribamar Lima Ribeiro, Presidente
Universidade Federal de Sergipe (UFS)

Prof. Dr. Rubens de Souza Matos Júnior, Membro
Instituto Federal de Sergipe (IFS)

Prof. Dr. Edward David Moreno Ordenez, Membro
Universidade Federal de Sergipe (UFS)

Prof. Dr. Adenauer Corrêa Yamin, Membro
Universidade Federal de Pelotas (UFPel)

Uma arquitetura de micro-PaaS em *fog computing*: Orquestração baseada em container para aplicações IoT utilizando single board computer

Este exemplar corresponde à redação na íntegra, da Dissertação de Mestrado, submetida à banca examinadora designada pelo Colegiado do Programa de Pós-Graduação em Ciência da Computação, para o Exame de Defesa do mestrando **WALTER DO ESPÍRITO SANTO,** como parte de requisitos para obtenção do título de Mestre em Ciência da Computação.

Aprovada em 09 de dezembro de 2019

Prof. Dr. Admilson de Ribamar Lima Ribeiro

Orientador

Prof. Dr. Rubens de Souza Matos Júnior

Coorientador

Prof. Dr. Edward David Moreno Ordenez

Membro

Prof. Dr. Adenauer Corrêa Yamin

Membro

UNIVERSIDADE FEDERAL DE SERGIPE
PRÓ-REITORIA DE PÓS-GRADUAÇÃO E PESQUISA
COORDENAÇÃO DE PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Ata da Sessão Solene de Defesa da Dissertação do Curso de Mestrado em Ciência da Computação-UFS.
Candidato: WALTER DO ESPÍRITO SANTO

Em 09 dias do mês de Dezembro do ano de dois mil e dezenove, com início às 14h00min, realizou-se na Sala de Seminários do DCOMP da Universidade Federal de Sergipe, na Cidade Universitária Prof. José Aloísio de Campos, a Sessão Pública de Defesa de Dissertação de Mestrado do candidato **Walter do espírito Santo**, que desenvolveu o trabalho intitulado: **" *Uma arquitetura de micro-PaaS para fog computing: Orquestração baseada em container para aplicações IoT utilizando single board computer*",** sob a orientação do Prof. Dr. **Admilson de Ribamar Lima Ribeiro**. A Sessão foi presidida pelo Prof. Dr. **Admilson de Ribamar Lima Ribeiro** (PROCC/UFS), que após a apresentação da dissertação passou a palavra aos outros membros da Banca Examinadora, Prof. Dr. **Edward David Moreno Ordonez** (PROCC/UFS), Prof. Dr. **Rubens de Souza Matos Júnior** (PROCC/UFS) e, em seguida, ao Prof. Dr. **Adenauer Corrêa Yamin** (UFPel). Após as discussões, a Banca Examinadora reuniu-se e considerou o mestrando (a) aprovado *"(aprovado/reprovado)"*. Atendidas as exigências da Instrução Normativa 01/2017/PROCC, do Regimento Interno do PROCC (Resolução 67/2014/CONEPE), e da Resolução nº 25/2014/CONEPE que regulamentam a Apresentação e Defesa de Dissertação, e nada mais havendo a tratar, a Banca Examinadora elaborou esta Ata que será assinada pelos seus membros e pelo mestrando.

Cidade Universitária *"Prof. José Aloísio de Campos"*, 09 de dezembro de 2019.

Prof. Dr. Admilson de Ribamar Lima Ribeiro
(PROCC/UFS)
Presidente

Prof. Dr. Edward David Moreno Ordonez
(PROCC/UFS)
Examinador Interno

Prof. Dr. Rubens de Souza Matos Júnior
(instituição)
Examinador Interno

Prof. Dr .Adenauer Corrêa Yamin
(UFPel)
Examinador Externo

Walter do Espírito Santo
Candidato

"Dedico este trabalho de dissertação de mestrado à Deus, por Sua infinita misericórdia e bondade a qual me fez transpor todas as adversidades com sabedoria, paz de espírito e força. Dedico também a minha esposa Elisângela, às minhas filhas Júlia e Ana Teresa, aos meus pais Silvio e Josefa e ao meu irmão Willian".

Agradecimentos

A obtenção dessa conquista não seria possível e também não faria sentido se não fosse pelo principal combustível, fonte de inspiração e razão de todos os meus dias: minha esposa Elisângela e minhas amadas filhas Júlia e Teresa.

Aos meus pais Silvio e Josefa pelo exemplo de educação em que fui criado, sempre pautada, na honestidade, no amor, no trabalho, e sobretudo, pela abdicação para dar sempre o melhor para mim e meu irmão Willian, o qual também extendo meus agradecimentos.

Agradeço ao meu orientador o professor Admilson pela orientação, paciência e por acreditar em nossa proposta de pesquisa dando o apoio necessário. Ao meu co-orientador o professor Rubens, colega de trabalho e exemplo de profissional, pelas contribuições enriquecedoras, pelo compromentimento e por sua disponibilidade às noites, às tardes e até mesmo nos finais de semana, sem palavras para externar tamanho companheirismo. Da mesma forma, agradeço ao professor Dr. Edward pelo encorajamento e apoio sempre de forma muito generosa durante esta gratificante caminhada.

Aos colegas de pesquisa Danilo, Reneilson e Machado pela troca de experiência, apoio técnico e uma parceria ímpar vivenciadas durante o mestrado.

Ao meu amigo irmão Bruno Gustavo, que mesmo apesar de toda a distância, ocupações e preocupações sempre encontrava um tempo para me incentivar, acreditando em mim mesmo em momentos em que eu próprio não acreditara.

E por fim, ao Instituto Federal de Sergipe campus Lagarto pelo suporte e apoio nessa formação tão importante para minha vida pessoal e profissional.

“Onde você está é resultado de quem você era, mas para onde você vai depende inteiramente de quem você escolhe ser”. (Hal Elrod)

Resumo

A Internet das Coisas (IoT) é um paradigma de tecnologia emergente em que sensores onipresentes monitoram infraestruturas físicas, ambientes e pessoas em tempo real para que sejam tomadas decisões que melhorem a eficiência e a confiabilidade dos sistemas adicionando conforto e qualidade de vida à sociedade. O armazenamento e processamento de dados de IoT são comumente realizados em infraestruturas de computação em nuvem (*cloud computing*), porém questões como a limitação de recursos computacionais, latência elevada e diferentes requisitos de qualidade de serviços (QoS), relacionados aos dispositivos IoT e que acabam movendo tecnologias *cloud* em direção a *fog computing*, e à adoção de soluções de virtualização leve, como as tecnologias baseadas em *containers* para atender às diversas necessidades de diferentes domínios. O presente estudo, tem como objetivo propor e implementar uma arquitetura de micro plataforma como serviço (micro-PaaS) em *fog computing*, em *cluster* de plataforma *single board computer* (SBC). A arquitetura proposta engloba a orquestração de aplicações utilizando *containers*, aplicada à IoT e que atendam a critérios de QoS, como, por exemplo, alta disponibilidade, escalabilidade, balanceamento de carga e latência. A partir do modelo proposto, a micro-PaaS Fog foi implementada com tecnologia de virtualização em *containers* utilizando serviços de orquestração em um *cluster* formado por dispositivos *Raspberry Pi* para monitoramento inteligente do consumo de água e energia em pontos focais do Instituto Federal de Sergipe Campus Lagarto. Os resultados mostraram que é possível implementar uma micro-PaaS Fog performática a um custo total de propriedade (TCO) equivalente a 23% de uma plataforma como serviço (PaaS) pública, latência média entre *mote* e a *fog* de 26ms e tempo médio de recuperação de falhas de 1,33 segundos levando-se em consideração um intervalo de confiança (IC) de 95%. Esta pesquisa também contribuiu com um mapeamento sistemático para identificação das principais características e requisitos para orquestração eficiente em ambientes *fog computing*. Por meio de um estudo comparativo, foram identificadas as seguintes características como sendo as mais comumente abordadas nessa área: heterogeneidade, gerenciamento de QoS, escalabilidade, mobilidade, federação e interoperabilidade.

Palavras-chave: *Fog Computing, Orchestration, Cluster, Container, Single Board Computing*

Abstract

The Internet of Things (IoT) is an emerging technology paradigm in which ubiquitous sensors monitor, physical infrastructure, environments and people in real time to make decisions that improve systems efficiency and reliability while adding comfort and quality of life to society. IoT data storage and processing are commonly performed on cloud computing infrastructures, but issues such as computational resource constraints, high latency, and different quality of service (QoS) requirements related to IoT devices that eventually move cloud technologies around. towards fog computing, and the adoption of light virtualization solutions such as container-based technologies to address the diverse needs of different domains. The present study aims to propose and implement a micro Platform as a Service (micro-PaaS) architecture in fog computing, in a single board computer (SBC) cluster. The proposed architecture encompasses application orchestration using containers, applied to the Internet of Things and meeting QoS criteria such as high availability, scalability, load balancing and latency. Based on the proposed model, the micro-PaaS Fog was implemented with container virtualization technology using orchestration services in a cluster of Raspberry Pi devices for intelligent monitoring of water and energy consumption at Sergipe Campus Lagarto Federal Institute focal points. . The results showed that it is possible to implement a performance micro-PaaS Fog at a total cost of ownership (TCO) equivalent to 23% of a public platform as a service (PaaS), 26m average mote-fog latency and average recovery time. 1.33 second failures taking into account a confidence interval (CI) of 95%. This research also contributed to a systematic mapping to identify key features and requirements for efficient orchestration in fog computing environments. Through a comparative study, the following characteristics were identified as being the most commonly addressed in this area: heterogeneity, QoS management, scalability, mobility, federation, and interoperability.

Keywords: *Fog Computing, Orchestration, Cluster, Container, Single Board Computing*

Lista de ilustrações

Lista de quadros

Lista de tabelas

Lista de siglas

API	*Application programming interface*
API	*Application Programming Interface*
CLI	*Command-line interface*
CRIU	*Checkpoint/restore in userspace*
DMCCP	*Dynamic mobile cloudlet cluster policy*
DVFS	*Dynamic Voltage and Frequency Scaling*
FaaS	*Fog as a service*
FSPs	*Fog services providers*
GPIO	*General Purpose Input/Output*
IaaS	*Infraestructure as a service*
IPSEC	*IP Security Protocol*
IoT	*Internet of Things*
LCD	*Lyquid Crystal Display*
LXC	*Linux containers*
MDC	*Micro datacenter*
MEC	*Mobile edge Computing*
MINLP	*Mixed-integer nonlinear programming*
NFV	*Network function virtualization*
OODA	*Observe Orient Decide Act*
PaaS	*Platform as a service*
PICOC	*Population, intervention, control, outcome, context*
QoE	*Quality of Experience*
QoS	*Quality of service*
RFID	*Radio-Frequency IDentification*
SaaS	*Software as a service*
SBC	*Single board computer*

SDN	*Software defined network*
SLOs	*Service level objectives*
TCO	*Total Cost of Ownership*
USB	*Universal serial bus*
VIoLET	*Large-scale Virtual Environment for Internet of Things*
VM	*Virtual machine*
VxLAN	*Virtual Extensible LAN*
WiFi	*Wireless Fidelity*
WSN	*Wireless sensor network*
4G	*Fourth Generation*

Sumário

1

Introdução

A Internet das Coisas do inglês *Internet of Things* (IoT) é um paradigma tecnológico emergente, onde sensores e atuadores ajudam a monitorar diversos tipos de aplicações em tempo real. Este paradigma está se expandindo rapidamente à medida que diversos domínios implantam sensores, comunicação e infraestrutura de *gateway* para suportar aplicações como, cidades inteligentes, saúde personalizada, veículos autônomos, entre outros. A IoT também está acelerando a necessidade e o uso de recursos de *edge computing*, *fog computing* e *cloud computing*, em um ambiente coordenado (BADIGER, BAHETI, SIMMHAN, 2018).

A integração entre a IoT e a *cloud computing* permite que simples objetos físicos executem tarefas, compartilhem informações e coordenem decisões entre si, transformando objetos tradicionais em objetos inteligentes, explorando suas tecnologias subjacentes, como computação onipresente e abrangente, dispositivos incorporados, tecnologias de comunicação, redes de sensores, protocolos e aplicações da Internet. Os objetos inteligentes, juntamente com suas supostas tarefas, constituem aplicações de domínios específicos, conhecidos como mercados verticais, enquanto as aplicações onipresentes de computação e análise formam serviços independentes de domínio, e são conhecidas como mercados horizontais (AL-FUQAHA *et al*., 2015).

Embora o paradigma da *cloud computing* seja capaz de lidar com grandes quantidades de dados de *clusters* de IoT, a transferência de enormes quantidades de dados para computadores na *cloud* apresenta um desafio devido a uma série de restrições apontados na literatura, como por exemplo, largura de banda, latência, economia de energia, taxa de transferência entre outros. Consequentemente, surge a necessidade de processar dados próximos à fonte de dados, e a *fog*

computing, também compreendida como computação de nevoeiro, névoa ou neblina, fornece uma solução promissora para esse problema (MUNIR *et al*., 2017).

Para a *fog computing*, a orquestração de aplicações utilizando *containers* pode auxiliar no gerenciamento do ambiente *fog* como um mecanismo de empacotamento de aplicações. A *fog computing* diminui significativamente o volume de dados que deve ser movido entre os dispositivos finais e a *cloud*, e permite que a análise de dados e a geração de conhecimento ocorram na origem dos dados. Além disso, a distribuição geográfica densa do ambiente *fog* ajuda a obter melhor precisão da localização para muitas aplicações em comparação com a *cloud* (MUNIR, KANSAKAR, KHAN, 2017).

Apesar da *fog computing* mitigar alguns dos problemas enfrentados na execução de aplicações IoT, os nós da *fog*, por exemplo, servidores de borda, roteadores inteligentes e estações base podem não atender às restrições de desempenho, alta disponibilidade, balanceamento de carga, escalabilidade e latência de aplicações para IoT. Nota-se, portanto, a necessidade de uma arquitetura em *fog computing* que seja adaptada para atender a esses requisitos.

1.1 Problemática e Hipótese

Pahl e Lee (2015) afirmam que a evolução da virtualização resultou em soluções mais leves, como, por exemplo na virtualização baseada em *containers*. Isso é especificamente relevante para o empacotamento de aplicações em uma plataforma de software e nível de aplicação. O *Docker* é uma solução de *container*, baseado nas técnicas do *Linux containers* (LXC), muito citado na literatura. Uma imagem do *Docker* é composta de sistemas de arquivos sobrepostos, semelhante à pilha de virtualização do *Linux*, usando os mecanismos LXC. Um *daemon* com reconhecimento de *container*, chamado *systemd*, inicia os *containers* como processos de aplicação, desempenhando um papel fundamental como a raiz da árvore de processos do usuário.

Para nortear o desenvolvimento desta pesquisa formulou-se a seguinte problemática: a capacidade de orquestração de aplicações na *fog computing* é capaz de fornecer processamento distribuído, selecionando e coordenando de forma inteligente os nós da *fog*, para que sejam atendidos critérios de QoS como baixa latência, escalabilidade, alta disponibilidade, balanceamento e custo total de propriedade reduzido?

A seguinte questão de propósito geral foi definida:

Quais tecnologias de conteinerização e técnicas de orquestração de aplicação utilizando *container* podem servir como micro-PaaS em ambientes *fog computing* utilizando *clusters* de plataformas *single board computing*, e que atendam a requisitos de QoS no contexto da IoT?

Como hipótese temos que por meio da tecnologia de conteinerização *Docker Container* com *Docker Swarm* é possível prover uma micro-PaaS orquestrada para aplicações IoT em ambientes *fog computing* em *clusters* de plataformas *Raspberry Pi*, com custo reduzido e respeitando critérios de QoS, como, por exemplo, alta disponibilidade, balanceamento de carga e latência.

1.2 Motivação

Nos últimos anos, a *cloud computing* sofreu uma transformação profunda impulsionada pela evolução tecnológica, como, por exemplo, novas técnicas de conteinerização e novas exigências impostas pelos domínios emergentes da Internet das Coisas. Do ponto de vista tecnológico, a virtualização do sistema operacional, ou seja, a conteinerização complementa a máquina virtual tradicional, porque é mais leve, flexível e portátil. Assim, desenvolvedores e provedores da *cloud* podem desenvolver padrões arquiteturais inovadores, como microsserviços e novos paradigmas. Ao mesmo tempo, as tecnologias de IoT abrangentes exigem suporte distribuído e descentralizado pelos serviços na *cloud*, dessa forma a *fog computing* surge para migrar dos *data centers* centralizados os serviços da *cloud* fornecendo-os mais próximos aos usuários ou às fontes de dados (SANTORO *et al.*, 2017).

O número de objetos físicos que está sendo conectado à *Internet* cresce a uma velocidade sem precedentes. Dentre estes podem ser citadas, termostatos, sistemas de monitoramento e controle de aquecimento, ventilação e ar condicionado em residências inteligentes. Há também outros domínios e ambientes em que a IoT pode desempenhar um papel notável e melhorar a qualidade de vida das pessoas. Aplicações para IoT incluem transporte, assistência médica, automação industrial e resposta de emergência a desastres naturais e provocados pelo homem, onde a tomada de decisão humana é difícil (AL-FUQAHA *et al*., 2015).

Os dispositivos IoT tornaram-se uma parte necessária da vida humana em todo o mundo. No entanto, o poder de computação destes dispositivos ainda não é suficiente para satisfazer a alta exigência de aplicativos. A *cloud computing* é uma das soluções, pois, desempenha uma função de servidor remoto para compartilhar a grande quantidade de serviços

sob demanda e fornece recursos flexíveis de tecnologia da informação (TI) para diferentes demandas de aplicações. (LI *et al*., 2018).

Bellavista e Zanni (2017), reconhecem que a literatura de integração *cloud* e *fog computing* ainda é incipiente. Apesar disso a *fog computing* está demonstrando ser um paradigma poderoso para superar vários desafios técnicos de aplicações IoT, no entanto, alguns problemas de design, implementação, implantação, tanto para atender especificidades de aplicações quanto na utilização de uso geral em diferentes domínios de aplicação, ainda necessitam ser endereçadas para transformar soluções eficazes da *fog* em realidade.

Existem também questões como a limitação de recursos computacionais, latência elevada e diferentes requisitos de QoS, relacionados aos dispositivos IoT, a tecnologia *cloud* está se movendo em direção a *fog computing*, e as soluções de virtualização leve, como as tecnologias baseadas em *containers*, mostram-se vantajosas para hospedar serviços de aplicações e plataformas como serviço, do inglês *Platform as a Sevice* (Paas) para arquiteturas com dispositivos menores. A utilização de *containers*, como, por exemplo o *Docker*, é discutido atualmente como uma solução de virtualização leve e que promove benefícios em relação às máquinas virtuais tradicionais na nuvem em termos de tamanho e flexibilidade (PAHL, LEE, 2015), (HOQUE *et al.*, 2017).

Outra questão que deve ser levada em consideração sob o aspecto motivacional pode ser encontrada em Brito, de *et al.* (2017), onde os autores afirmam que, atualmente, existem propostas de soluções de orquestração em diversas áreas, apresentadas em diferentes cenários para indicar o gerenciamento do ciclo de vida de um ou mais componentes distribuídos que juntos fornecem um serviço ou funcionalidade, em termos de máquinas virtuais e orquestração de *containers*. No entanto, os autores destacam a necessidade de realização de estudos envolvendo orquestração direcionados à *fog computing*.

1.3 Justiicativa

Os dispositivos IoT visam melhorar a qualidade da vida das pessoas usando a tecnologia para melhorar a eficiência dos serviços para atender às necessidades de uma localidade ou domínios específicos. Atingir este objetivo requer a integração de múltiplas tecnologias de informação e comunicação de forma segura, eficiente e confiável para gerenciar as instalações de domínios específicos de forma eficaz. De acordo com Wen *et al.* (2017), tais sistemas consistem em dois componentes principais: sensores integrados com sistemas de

monitoramento em tempo real, e aplicações integradas com os dados recolhidos do sensor (ou dispositivo). Ainda de acordo com os autores, atualmente, os serviços de IoT são rudimentares e só se integram a tipos de sensores específicos. Isso resulta da falta de padrões e protocolos universalmente aceitos para comunicação de dispositivos de IoT e representa um desafio para alcançar um ecossistema global de coisas interconectadas.

Para resolver problemas como os apresentados no parágrafo anterior, uma alternativa seria utilizar sistemas de orquestração de aplicações utilizando *containers* para ambientes da *fog*. O uso da orquestração permite uma formação mais flexível de funcionalidades o que contribui diretamente para reduzir a probabilidade de correlação de falhas entre os componentes do aplicativo (WEN *et al.*, 2017), (GOGOUVITIS *et al.*, 2018).

Gogouvitis *et al.* (2018), afirmam ainda que a orquestração permite um gerenciamento automatizado e a coordenação de aplicações com base em um fluxo de trabalho predefinido. A orquestração pode ter alguns aspectos autonômicos e, assim, alcançar um sistema auto gerenciado.

Os nós da *fog* são distribuídos pela borda e nem todos são altamente ocupados por recursos. Nesse caso, a implantação de aplicações em larga escala em um único nó da *fog* nem sempre é viável. O desenvolvimento modular de aplicações de grande escala e sua distribuição sobre nós da *fog* sob restrição de recursos pode ser uma solução eficaz. Na literatura, várias plataformas de programação para distribuição de desenvolvimento e implementação de aplicações foram propostas. No entanto, os problemas relacionados à distribuição de aplicações distribuídas, como gerenciamento de latência, gerenciamento de fluxo de dados, garantia de QoS, afinidade centralizada nas bordas de aplicações em tempo real, etc., não foram abordados adequadamente (MAHMUD, KOTAGIRI, BUYYA, 2018).

Embora a orquestração em *datacenters* e *clusters* estejam consolidadas, na *fog computing* ainda é um desafio devido à natureza dinâmica e heterogênea dos dispositivos envolvidos no processo. O número de restrições em uma orquestração e seus blocos de construção, como, por exemplo, os micros serviços podem ser muitos. Além disso, a orquestração pode ser feita centralmente ou distribuída, o que significa que um nó da *fog* pode começar a orquestrar as aplicações principais tentando mantê-las funcionais garantindo assim, aspectos de QoS (BRITO *et al.*, 2017).

1.4 Objetivo geral

Propor uma arquitetura de micro-PaaS em *fog computing*, em *cluster* de plataforma *single board computer* (SBC), para orquestração de aplicações utilizando *containers*, aplicada à Internet das Coisas e que atendam a critérios de QoS, como, por exemplo, alta disponibilidade, escalabilidade, balanceamento de carga e latência.

1.4.1 Objetivos Específicos

Para alcançar o objetivo principal, alguns objetivos específicos foram delineados, são eles:

- Identificar quais tecnologias de conteinerização são mais adequadas para plataformas como serviço em ambientes *fog computing;*
- Identificar quais técnicas de orquestração podem ser utilizadas em aplicações IoT e que sejam compatíveis com plataformas *single board computer*;
- Propor um modelo conceitual e implementar uma arquitetura de micro-PaaS em *fog computing* que atenda aos requisitos de orquestração de *containers* em *cluster* de plataformas SBC para aplicações IoT;
- Avaliar as métricas de QoS no protótipo da arquitetura proposta, considerando os seguintes aspectos: latência, escalabilidade, alta disponibilidade e balanceamento de carga;
- Desenvolver um protótipo para validar a arquitetura proposta.

1.5 Organização da dissertação

Esta dissertação está organizada em 7 (sete) capítulos, onde são discutidos assuntos os que fornecem a base conceitual acerca de tecnologias de conteinerização e orquestração de aplicações contextualizados com a Internet das Coisas em ambientes *fog computing*. No capítulo 1 é apresentado o problema de pesquisa, ou seja, questão não resolvida e que é objeto de discussão desta pesquisa de mestrado; a hipótese que consiste em oferecer uma solução possível, mediante uma proposição; motivação, justificativas, objetivo geral e objetivos específicos. No capítulo 2 apresenta-se a fundamentação teórica da dissertação, nela são

discutidos os principais conceitos utilizados pela literatura no paradigma *fog computing* relacionando tecnologias de conteinerização e orquestração de *containers* em aplicações para a Internet das Coisas. No capítulo 3 é realizada a síntese e discussão dos trabalhos relacionados, destacando as principais contribuições, suas limitações e desafios futuros de pesquisa, fazendo uma correlação entre estes e comparando-os com esta pesquisa de mestrado. Já no capítulo 4 são discutidos os materiais e métodos de pesquisa, baseados em uma pesquisa experimental, cujo objetivo principal foi implementar e analisar uma arquitetura em *fog computing* capaz de orquestrar aplicações destinadas aos dispositivos inteligentes. O capítulo 5 discute a arquitetura de micro-PaaS Fog e as premissas de concepção do modelo arquitetural que são utilizados para solução da problemática apresentada e a descrição de todas as etapas necessárias até a conclusão da pesquisa. No capítulo 6 apresentam-se os resultados e discussões das informações coletadas do ambiente de modo a identificar na arquitetura implementada, as vantagens em termos de melhoria da qualidade de serviço para aplicações IoT relacionados à baixa latência, alta disponibilidade, balanceamento de carga e o custo total de propriedade. Por fim, o capítulo 7 traz as conclusões, limitações e dificuldades encontradas e possíveis direcionamentos de pesquisa para trabalhos futuros.

2

Fundamentação Teórica

2.1 Internet das Coisas

A Internet das Coisas (IoT), é um paradigma tecnológico emergente, onde milhões de sensores e atuadores ajudam a monitorar e gerenciar sistemas em tempo real. Estes dispositivos são interconectados e capazes de trocar dados entre si (HONG *et al.*, 2013), (CIRANI *et al.*, 2015), (GIANG *et al.*, 2015). A ideia básica deste conceito é a presença generalizada em torno de *nodes* de uma variedade de coisas ou objetos – tais como identificadores de *Radio-Frequency IDentification* (RFID), sensores, atuadores, telefones celulares, etc. – que, através de esquemas de endereçamento únicos, são capazes de interagir uns com os outros e cooperar entre seus vizinhos para alcançar objetivos comuns (GIUSTO, 2010).

As palavras “Internet” e “Coisas”, quando juntas, assumem um significado que introduz um nível disruptivo de inovação no mundo atual das Tecnologias de Informação e Comunicação (TIC). De fato, “Internet das Coisas” semanticamente significa uma rede mundial de objetos interconectados exclusivamente endereçáveis, baseada em protocolos de comunicação padrão.

A Internet das Coisas pode ser caracterizada sob três visões diferentes, conforme ilustradas na Figura 1– orientada a coisas (sensores), orientada à Internet (middleware) e orientada a semântica (conhecimento). Embora esse tipo de delimitação seja necessário devido à natureza interdisciplinar do assunto, a utilidade da IoT pode ser desencadeada apenas em um domínio de aplicação entre a intersecção das três visões (GUBBI *et al.*, 2013), (ATZORI, IERA, MORABITO, 2010).

Figura 1 – Adaptado de Visões da Internet das Coisas

Fonte: (ATZORI, IERA, MORABITO, 2010)

Essas visões destacam os principais conceitos, tecnologias e padrões que compõem a IoT, classificando-as em três grupos distintos:

- **Visão orientada às coisas:** compreendem propostas que assegurem o melhor aproveitamento dos recursos dos dispositivos e sua comunicação;
- **Visão orientada à internet:** tem o intuito de conceber modelos e técnicas destinadas a interoperabilidade dos dispositivos em rede.
- **Visão orientada à semântica:** foca na representação, armazenamento, pesquisa e organização da informação gerada, procurando soluções para a modelagem das descrições que permitam um tratamento adequado para os dados produzidos pelos objetos;

Inquestionavelmente, a principal força da ideia da IoT é o alto impacto que ela terá em vários aspectos da vida cotidiana e no comportamento dos usuários em potencial. Por conseguinte, a domótica, a vida assistida, a *e-health*, o aprendizado aprimorado são apenas alguns exemplos de possíveis cenários de aplicação nos quais o novo paradigma terá um papel

de liderança no futuro próximo. Da mesma forma, sob o ponto de vista empresarial, as consequências mais aparentes serão igualmente visíveis em campos como automação e manufatura industrial, logística, gerenciamento de negócios, processos, transporte inteligente de pessoas e mercadorias (ATZORI, IERA, MORABITO, 2010).

A conectividade inteligente com redes existentes e computação ciente de contexto usando recursos de rede é uma parte indispensável da IoT. Com a crescente presença do *Wireless Fidelity* (WiFi) e do acesso sem fio à Internet por meio das tecnologias de quarta geração da telefonia celular (4G), a evolução para redes onipresentes de informação e comunicação tornou-se uma realidade. No entanto, para que a visão da Internet das Coisas possa ir além dos cenários tradicionais de computação móvel que usam smartphones e portáteis e evoluir para conectar objetos do cotidiano e incorporar inteligência em nosso ambiente é necessário: (i) uma compreensão compartilhada da situação de seus usuários e seus dispositivos, (ii) arquiteturas de software e redes de comunicação difundidas para processar e transmitir a informação contextual para onde é relevante e (iii) ferramentas de análise na Internet das Coisas que visam o comportamento autônomo e inteligente (GUBBI *et al.*, 2013) .

Neste contexto, as potencialidades oferecidas pela IoT possibilitam o desenvolvimento de um grande número de aplicações em domínios como transporte e logística, saúde, ambientes inteligentes, pessoal e social. No entanto, apenas uma parte muito pequena está atualmente disponível para a sociedade. Dar a esses objetos a possibilidade de se comunicarem uns com os outros e tratar as informações percebidas a partir do entorno implica ter diferentes ambientes onde uma ampla gama de aplicações possam ser implementadas (GIANG *et al.*, 2015).

2.2 Computação em Nuvem

Ao longo dos anos os paradigmas da computação evoluíram da computação distribuída, paralela e de grade para a computação em nuvem. A computação em nuvem é um paradigma de computação baseado na Internet que fornece acesso onipresente e sob demanda a um conjunto compartilhado de recursos configuráveis, como, por exemplo, processadores, armazenamento, serviços e aplicativos para outros computadores ou dispositivos (MUNIR *et al*., 2017).

A computação em nuvem traz com vários recursos inerentes, como escalabilidade, alocação de recursos sob demanda, esforços reduzidos de gerenciamento, modelo de preço

flexível (*pay-as-you-go*) e fácil provisionamento de aplicações e serviços (MOURADIAN *et al.*, 2017). De acordo com Antoni, Vivian e Preuss, (2015) a computação em nuvem é considerada como o próximo passo na evolução natural dos sistemas computacionais, permitindo que o usuário acesse, através da internet, uma grande quantidade de informação e serviços que não estão armazenados localmente no seu dispositivo.

A computação em nuvem é uma forma de computação baseada na Internet que pode fornecer recursos de hardware e software, de acordo com as necessidades. De acordo com Zhang, Cheng e Boutaba, (2010), a arquitetura de um ambiente de computação em nuvem pode ser dividida em 4 camadas: A camada de *hardware* / *datacenter*; a camada da infraestrutura; a camada de plataforma e a camada de aplicação, como mostrado na Figura 2.

A computação em nuvem virtualiza a infraestrutura de computação, redes, armazenamento e outros, para formar uma grande gama de recursos compartilhados, mensuráveis e dinâmicos, e fornece um modelo de computação para todos os tipos de usuários na forma de serviços controlados através de uma plataforma de gestão (BOTTA *et al.*, 2016),(ANTONI, VIVIAN, PREUSS, 2015),(BABU, LAKSHMI, RAO, 2015).

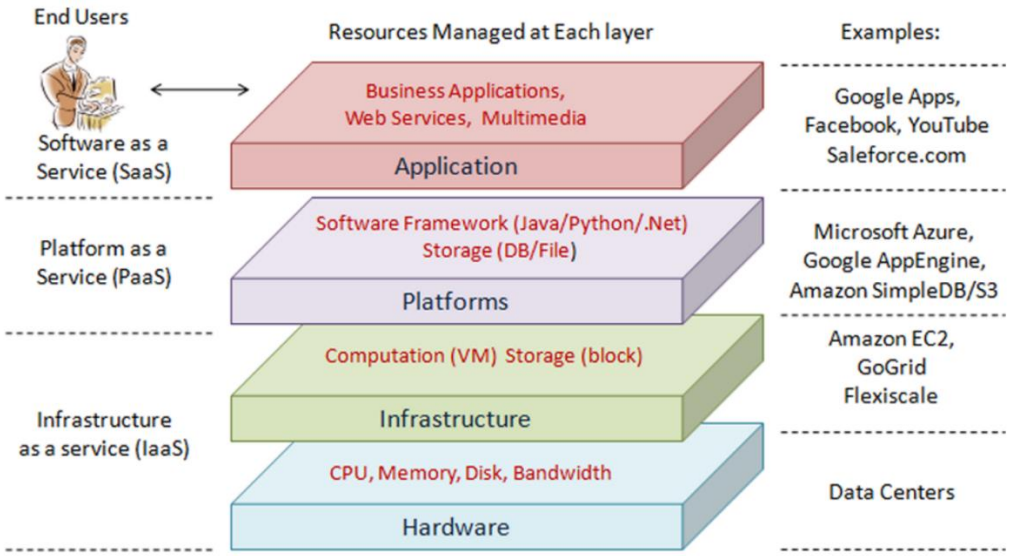
A computação em nuvem compreende três modelos de serviços principais: Infraestrutura como serviço (IaaS); Plataforma como serviço (PaaS) e Software como serviço (SaaS) (MOURADIAN *et al.*, 2017).

- IaaS fornece os recursos virtualizados, como computação, armazenamento e rede. O proprietário da nuvem que oferece IaaS é chamado de provedor de IaaS, como, por exemplo: *Amazon EC2*, *GoGrid* e *Flexiscale*;

- A PaaS fornece ambientes de software para o desenvolvimento, implementação e gerenciamento de aplicações, como, por exemplo: *Google App Engine*, *Microsoft Windows Azure* e *Force.com*;

- O SaaS fornece aplicações de software e serviços sob demanda através da internet para usuários finais, como, por exemplo: *Salesforce.com*, *Rackspace* e *SAP Business ByDesign*.

Figura 2 - Arquitetura *cloud computing*



Fonte: (ZHANG, CHENG, BOUTABA, 2010)

2.2.1 Plataforma como Serviço - PaaS

Na configuração da *cloud*, a PaaS opera os recursos físicos virtualizados fornecidos pela IaaS para hospedar e executar aplicações finais do usuário. Essas aplicações são oferecidas como SaaS e podem estar localizados longe dos usuários finais e das fontes de dados (VAQUERO E RODERO-MERINO, 2014).

A PaaS trouxe, também, a produtividade do desenvolvedor para a vanguarda, tornando possível criar uma camada de abstração no laaS para a implantação de serviços com facilidade e rapidez, levando a desafios e oportunidades únicos. Além disso, a PaaS é capaz de fornecer mecanismos para implantar e projetar aplicações para a nuvem, expandir aplicações para seu ambiente de implementação, migrar bancos de dados, mapear domínios personalizados; implementar *plugins* de IDE ou ferramentas de integração de compilação (DUA, RAJA, KAKADIA, 2014).

A PaaS concentra-se na produtividade do desenvolvedor, em vez de gerenciar a rede, o armazenamento e o processamento. Isso possibilita abstrair a implementação do ambiente de tempo de execução do aplicativo. De acordo com Dua, Raja e Kakadia, (2014), alguns dos principais requisitos para a PaaS hospedar uma aplicação são:

- Isolamento da aplicação no nível de rede, computação e armazenamento;

- Suportar várias linguagens de programação e ambientes de execução de aplicação;
- Divisão justa do tempo da CPU;
- Gerenciamento fácil de eventos do ciclo de vida da aplicação;
- Persistência e migração do estado da aplicação.

2.2.2 Micro Plataforma como Serviço – micro-PaaS

De acordo com Pahl *et al.* (2016), há uma tendência evolutiva de PaaS da computação na nuvem avançando para a computação na borda da rede (*edge computing*), dessa forma os autores classificaram o modelo de serviço, PaaS, em quatro gerações:

- *Proprietary*: primeira geração de PaaS incluiu plataformas proprietárias fixas, por exemplo, *Azure* ou *Heroku*;
- *Open-source*: segunda geração de PaaS incluiu soluções de código aberto, por exemplo, *Cloud Foundry* ou *Open-Shift*, permitindo que os usuários executem seus próprios PaaS (no local ou na nuvem), muitos com suporte embutido a *containers*;
- Micro-PaaS: terceira geração de PaaS são construídos no *Docker* a partir do zero e podem ser implementados em servidores próprios ou em nuvens públicas (IaaS). As micro-PaaS baseiam-se em serviços leves e, portanto, possibilitam a multilocação distribuída em plataformas de recursos limitados, como, por exemplo, *Raspberry Pi*;
- PaaS *edge cloud*: quarta geração de PaaS fornecem recursos de PaaS para ambientes de computação na borda da rede, ou seja, desenvolvem micro-PaaS em ambientes de borda, concentrando-se mais em *clustering* e orquestração.

2.3 *Fog Computing*

A *fog computing* foi definida de várias maneiras na literatura pela academia e indústria. O termo *fog computing* é frequentemente associado à Cisco, como em "Cisco *fog computing*" (CISCO, 2016); no entanto, está aberto à comunidade em geral. Uma coalizão de indústria e academia (ARM, Cisco, Dell, Intel, Microsoft e Princeton University) fundaram o "*OpenFog* Consortium" em novembro de 2015 para promover e acelerar a adoção da *fog computing* aberta

(OPENFOG CONSORTIUM, 2017). O *OpenFog Consortium* define a *fog computing* da seguinte forma: "A *fog computing* é uma arquitetura horizontal de nível de sistema que distribui recursos e serviços de computação, armazenamento, controle e rede em qualquer lugar ao longo da *cloud*. "

A *fog computing* é uma arquitetura de computação distribuída geograficamente com um *pool* de recursos que consiste em um ou mais dispositivos heterogêneos conectados de forma ubíqua na borda da rede e não exclusivamente garantidos por serviços em nuvem (YI *et al.* 2015). Aazam e Huh, (2016), definem a *fog computing* da seguinte forma: "A *fog computing* aproxima os recursos de rede das redes subjacentes. É uma rede entre a rede subjacente e a nuvem. A *fog computing* estende o paradigma tradicional de computação em *cloud* até a borda da rede, permitindo a criação de aplicação ou serviços refinados e melhores. A *fog* é um paradigma de computação de borda e *micro datacenter* (MDC) para IoT e redes de sensores sem fio, do Inglês, *wireless sensor network* (WSN). "

Além disso, a *fog computing* facilita a consciência de localização, suporte de mobilidade, interações em tempo real, escalabilidade e interoperabilidade (BONOMI *et al.*, 2012). Com o acelerado crescimento da Internet das Coisas exigi-se recursos de computação mais próximos dos dispositivos, mostrando que a nuvem não pode atender sozinha às necessidades das aplicações IoT, um problema que tecnologias como *fog computing* tentam resolver. As principais vantagens desse novo paradigma são a redução da latência na comunicação com serviços implantados na borda, maior segurança e uso de largura de banda minimizado em relação à *cloud*, pois o nó *fog* armazena e processa dados próximos aos dispositivos em que são gerados (HOQUE *et al.*, 2017), (COMPUTING, 2015).

Assim, a computação *fog* pode executar eficientemente em termos de latência de serviço, consumo de energia, tráfego de rede, despesas operacionais e de capital, distribuição de conteúdo, etc. Nesse sentido, a *fog computing* atende melhor aos requisitos relacionados a aplicações IoT em comparação com o uso exclusivo da computação em nuvem (SARKAR, CHATTERJEE, MISRA, 2018).

As distinções entre *fog computing* e outras abordagens computacionais relacionadas como, computação na nuvem, computação na borda da rede e *cloudlet* não foram elucidadas em muitos trabalhos acadêmicos relevantes (MUNIR, KANSAKAR, KHAN, 2017), (MAHMUD, KOTAGIRI, BUYYA, 2018) para fornecer uma compreensão clara do ambiente *fog*, discutimos a distinção entre estas nas subseções a seguir.

2.3.1 Distinção entre *fog computing* e computação na nuvem

A *fog computing* empurra aplicações, serviços, dados, poder de computação e tomada de decisão dos nós centralizados para os extremos lógicos de uma rede. A palavra *fog* na *fog computing* transmite a ideia de aproximar as vantagens da *cloud* à fonte de dados. A *cloud computing* geralmente é um modelo para permitir o uso conveniente e *on-demand* de um *pool* compartilhado de recursos de computação configuráveis. A *fog computing* estende a *cloud computing* e os serviços até a borda da rede (MUNIR, KANSAKAR, KHAN, 2017).

A *fog computing* pode ser distinguida da *cloud computing* baseada em várias métricas (Abdelshkour, Maher., 2015). A proximidade da *fog* aos usuários finais é uma das principais características que diferencia a *fog* da *cloud*; ou seja, a *fog* reside na borda da rede, enquanto a *cloud* está localizada na Internet. A *cloud* tem uma distribuição geográfica centralizada, enquanto a *fog* pode ter uma distribuição geográfica localizada ou distribuída. Os sistemas da *cloud computing* normalmente consistem em apenas alguns nós de servidor com recursos, enquanto a *fog* compreende um grande número de nós da *fog* relativamente menos engenhosos. Além disso, o processamento nos nós da *fog* libera a largura de banda da rede principal, o que ajuda a melhorar a eficiência geral da rede.

A distância entre os nós do cliente e do servidor na *cloud* é tipicamente de vários saltos, enquanto os clientes podem se conectar aos nós *fog* normalmente por meio de um único salto. Consequentemente, a *fog computing* reduz a latência da transmissão de dados de dispositivos IoT para o servidor descarregado devido à proximidade da *fog* aos dispositivos finais em comparação à *cloud*. As plataformas da *cloud computing* normalmente geram um *jitter* de atraso mais alto para aplicativos em comparação com os aplicativos em execução nos nós da *fog*. Portanto, a *fog computing* é mais adequada para aplicações IoT em tempo real do que a *cloud computing* (MUNIR, KANSAKAR, KHAN, 2017).

A capacidade da *fog* fornecer personalização baseada em localização de conteúdo, serviços e aplicações para dispositivos IoT IoT é outra característica que a diferencia dos demais paradigmas baseados em *cloud*. A *cloud*, por outro lado, na maioria dos casos, não é capaz de fornecer conteúdo especializado, serviços e aplicativos para dispositivos. A customização baseada em localização de serviços e informações é imperativa porque a informação pode ser relevante em um contexto local (proximidade de coordenadas geográficas específicas) e pode ser irrelevante além da proximidade física com aquela localização. Por fim, a *cloud* fornece

suporte limitado à mobilidade para os dispositivos finais, a mobilidade destes é melhor suportada na *fog* (MUNIR, KANSAKAR, KHAN, 2017).

Embora os paradigmas da *cloud computing* e *fog* tenham distinções bem definidas, um não substitui o outro. De fato, a *fog* e a *cloud*, são interdependentes e mutuamente benéficas, uma vez que certas funções são naturalmente mais vantajosas para serem executadas na *fog*, enquanto outras são mais adequadas à *cloud*. A segmentação das tarefas de *back-end* que vão para a *fog* e quais tarefas vão para a *cloud* é específica da aplicação e pode mudar dinamicamente com base no estado da rede, incluindo cargas de processador, largura de banda, capacidade de armazenamento, eventos de falha e ameaças de segurança (OpenFog Consortium, 2017). A *cloud* fornece vários serviços, como, *infraestructure as a service* (IaaS), *platform as a service* (PaaS) e *software as a service* (SaaS), para organizações que exigem escala elástica. A *fog computing* pode fornecer a *fog as a service* (FaaS) para lidar com vários desafios de negócios. A FaaS pode fornecer serviços, tais como: aceleração da rede; *network function virtualization* (NFV) por meio das *software defined network* (SDN); distribuição de conteúdo; gestão de dispositivos de processamento de eventos complexos; codificação de vídeo; descarregamento de tráfego, criptografia, e plataformas de análises (OpenFog Consortium, 2017).

2.3.2 Distinção entre *fog computing* e computação na borda da rede

A distinção entre *fog computing* e *edge computing* é sutil. A maior parte da literatura trata ambas como sinônimo e usa os termos *fog computing* e *edge computing* de forma intercambiável. Neste contexto o termo *mobile-edge computing* (MEC), também é frequentemente usado como jargão. O MEC é uma instância *edge computing* em que o objetivo é fornecer recursos da *cloud computing* na borda da rede celular. O servidor de borda no MEC está localizado na estação base do celular. Tanto a *fog computing* quanto a *edge computing* empurram aplicações, dados, serviços e poder de computação dos nós centralizados para os extremos lógicos da rede. No entanto, o paradigma da *fog computing* tem um controle mais descentralizado e distribuído em comparação com o paradigma da *edge computing*, que possui um controle relativamente mais centralizado.

Outra distinção entre *fog computing* e *edge computing* é que aquela faz parte de uma arquitetura aberta e interoperável, fundamental para o sucesso de um ecossistema onipresente para plataformas e aplicações IoT. As soluções proprietárias ou de fornecedor único,

normalmente encontradas na *edge computing*, podem gerar limitações, o que pode ter um impacto negativo no custo, na qualidade, na adoção do mercado e na inovação do sistema. Como um paradigma de computação localizada, a *edge computing* fornece respostas mais rápidas às solicitações de serviços computacionais, no entanto, em geral, a *edge computing*, diferentemente da *fog computing*, não associa espontaneamente IaaS, PaaS, SaaS e outros serviços baseados na *cloud* e se concentra mais no lado dos dispositivos finais (SHI *et al.*, 2016).

2.4 Princípios de Virtualização

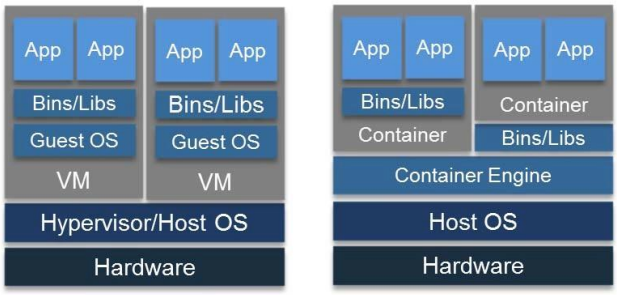
A virtualização de recursos normalmente inclui a utilização de uma camada de *software* adicional acima do sistema operacional do *host*, com o objetivo de lidar com vários recursos. As *virtual machines* (VM) podem ser consideradas como um ambiente de execução separado. Várias abordagens são usadas para fins de virtualização. Uma técnica popular é a virtualização baseada em *hipervisor*, como, por exemplo: *kernel-basead virtual machine* (KVM) e *VMware*. Para usar esse tipo de tecnologia, deve haver um monitor de VM acima do sistema físico subjacente. Cada VM também tem suporte para sistemas operacionais convidados (isolados). Um sistema operacional hospedeiro pode suportar muitos sistemas operacionais convidados dentro dessa abordagem de virtualização (XAVIER *et al.*, 2013).

De acordo com Xavier *et al.* (2013), a tecnologia de virtualização baseada em *container* aloca os recursos de *hardware*, com propriedades de isolamento (seguras), de acordo com a quantidade de instâncias implementadas. Para Soltesz *et al.* (2007), Pahl e Lee, (2015), um *container* é essencialmente um conjunto de partes de aplicativos autocontido e pronto para ser implantado, pode incluir *middleware* e lógica de negócios na forma de binários e bibliotecas para executar aplicações. A diferença entre as duas tecnologias pode ser vista na Figura 3. Os processos convidados obtêm abstrações imediatamente com tecnologias baseadas em *containers* enquanto operam através da camada de virtualização diretamente no nível do sistema operacional. No entanto, nas abordagens baseadas em *hipervisor*, normalmente há uma máquina virtual por sistema operacional convidado (XAVIER *et al.*, 2013).

Soluções baseadas em *container* compartilham o *kernel* do sistema operacional entre instâncias virtuais. Portanto, há uma suposição de que a segurança desse tipo de abordagem é mais fraca do que com os *hipervisores*. Do ponto de vista dos usuários, os *containers* operam

como sistemas operacionais autônomos, que parecem capazes de executar independentemente da combinação de *hardware* e sistema operacional subjacentes (SOLTESZ *et al.*, 2007).

Figura 3 - Arquitetura de virtualização VM (esquerda) e *container* (direita)



Fonte: (PAHL, LEE, 2015)

Kozhirbayev e Sinnott (2017), afirmam que o uso de soluções baseadas em *containers* permite a implantação dinâmica e o uso de microsserviços em ambientes de hospedagem compartilhados. Os padrões de microsserviços não são uma ideia nova na arquitetura de *software*, e, atualmente, são amplamente reconhecidos como uma solução eficiente para desenvolver aplicativos. Antes da arquitetura de microsserviços, a abordagem geral para o desenvolvimento de serviços era criar aplicações amplamente monolíticas. De um ponto de vista funcional, isso exige um ambiente único que lida com todas as coisas. Em ambientes na nuvem, muitos desses problemas podem ser superados por meio de abordagens de *script* que oferecem suporte a IaaS, PaaS e SaaS (KOZHIRBAYEV, SINNOTT, 2017).

2.5 Orquestração em Sistemas Distribuídos

Um sistema de computação distribuído é uma coleção de componentes de software independentes projetados para cooperar e fornecer uma intenção comum. As características de sistemas distribuídos, motivações para usá-los, seus benefícios e vários desafios na concepção e implementação de sistemas de computação modernos são amplamente discutidos na literatura (MIKKILINENI, MORANA, ZITO, 2015).

Já a orquestração é normalmente vinculada ao provisionamento automatizado de serviços na *cloud comptuing* ou em redes habilitadas para NFV (*Network Function Virtualization*) (ETSI, 2013). A implantação de novos serviços depende de configurações manuais de diferentes dispositivos situados em uma rede física, que normalmente possuem sistemas de

gerenciamento independentes não integrados. Nesse sentido, se diferentes entidades de negócios do operador forem responsáveis por diferentes subsistemas de gerenciamento ou domínios, um impacto negativo no tempo de provisionamento de serviço é geralmente observado.

A orquestração é entendida como automação de ponta a ponta da implementação de serviços em um ambiente da *cloud*. Mais especificamente, é a automatização da organização, da coordenação e do gerenciamento de complexos sistemas de computação, *middleware* e serviços. E tudo isso ajuda a acelerar a entrega de serviços de TI e ainda reduz custos. Ela é usada para gerenciar a infraestrutura *cloud*, que fornece e designa ao cliente os recursos de nuvem necessários, como, criação de VMs, alocação de capacidade de armazenamento, gerenciamento de recursos de rede e concessão de acesso ao software em nuvem. Usando os mecanismos de orquestração apropriados, os usuários podem implementar e começar a usar serviços em servidores ou em qualquer plataforma de nuvem (SUCHITRA, 2016). Ainda de acordo com o autor há três aspectos da orquestração na *cloud* que devem ser considerados:

- Orquestração de recurso – em que os recursos são alocados;
- Orquestração de carga de trabalho – em que as cargas de trabalho são compartilhadas entre os recursos;
- Orquestração de serviço.

Normalmente, a orquestração quebra as fronteiras de domínios técnicos e administrativos, agilizando a implantação desses serviços. A implementação da orquestração pode ser feita de várias maneiras. Ela pode ser orientada por um script de baixo nível usado apenas para um ambiente de rede específico ou por uma descrição de serviço de alto nível, que deve ser convertida em comandos de baixo nível pelo orquestrador, dando assim mais liberdade e aumentando a eficiência da implementação (KUKLIŃSKI *et al.*, 2016).

De acordo com Kukliński *et al.*, (2016), a orquestração diz respeito a vários domínios administrativos ou tecnológicos ou camadas do sistema, que já possuem gerenciamento e / ou controle intrínseco e seu objetivo principal é a implantação e manutenção de serviços dinâmicos. Ele deve exibir as seguintes propriedades: (i) o serviço dinamicamente implantado deve ser definido em alto nível (não é um fluxo de trabalho automatizado) e o orquestrador deve ter uma inteligência incorporada para converter a descrição de serviço de alto nível em comandos de baixo nível; (ii) o orquestrador deve ser consciente do ambiente de uma maneira similar ao gerenciamento de uma rede autonômica e cognitiva (deve ser baseado em ciclos de

retroalimentação); (iii) o orquestrador deve estar ciente das restrições e preferências do operador (políticas).

2.6 Tecnologias de Virtualização baseadas em *Containers*

Pahl e Lee (2015), destacam como os *containers*, diferentemente das VMs, são mais flexíveis para empacotamento, entrega e orquestração de serviços de infraestrutura de software e componentes de nível de aplicação, ou seja, tarefas tipicamente executadas por uma PaaS. Além disso, os *containers* fornecem uma abstração que torna cada *container* uma unidade autônoma de computação (LXC, 2018), permitindo assim uma portabilidade e interoperabilidade mais fáceis, com componentes leves e boa adequação para aplicações distribuídas. Várias ferramentas fornecem a abstração e implementam os modelos orientados a *containers*, por exemplo, LXC, *Docker*, *checkpoint/restore in userspace* (CRIU), *systemd-nspawn*, *runC*, *OpenVZ* entre outros.

As soluções baseadas em *containers* oferecem vantagens, como, por exemplo, abstrair detalhes de implementação, maior facilidade para alcançar interoperabilidade e portabilidade entre nós, possivelmente, heterogêneos, etc. Estas soluções comportam-se similarmente à adoção de VM, mas com migrações leves e de melhor desempenho (FELTER *et al.*, 2015), (JOY, 2015).

Os *containers* podem encapsular diversos componentes de uma aplicação por meio do processo de camadas e extensão de imagens. Uma solução de *container* consiste em dois componentes principais: (i) um mecanismo de *container* de aplicativo para executar imagens e (ii) um repositório ou registro que é operado por meio de operações *push* e *pull* para transferir imagens para mecanismos de *container* baseados em *host*.

Os repositórios de *containers* desempenham um papel central no fornecimento de imagens de *container* privadas e públicas possivelmente reutilizáveis. A *application programming interface* (API) do *container engine* suporta operações de ciclo de vida, como criar, compor, distribuir *containers*, iniciar e executar comandos em imagens.

Os *containers* são criados montando-os a partir de imagens individuais, possivelmente extraídas dos repositórios. O armazenamento e o gerenciamento de rede são dois serviços específicos de plataforma/*middleware* que são necessários para suportar *containers* como pacotes de aplicações para nuvens de borda distribuídas.

As operações de armazenamento de dados podem adicionar volumes de dados a qualquer *container*. Um volume de dados é um diretório designado em um ou mais *containers* que ultrapassam a união de sistemas de arquivos. Isso permite fornecer dados persistentes ou compartilhados. Os volumes podem então ser compartilhados e reutilizados entre os *containers* pertencentes a domínios distintos. Um *container* de volume de dados permite compartilhar dados persistentes entre *containers* de aplicações por meio de um *container* de armazenamento de dados separado e dedicado.

O gerenciamento de rede é baseado em dois métodos: o primeiro para designar portas em um mapeamento de *host* de rede; e o segundo para vinculação de *container*. As aplicações podem se conectar dentro de um *container* por meio de uma porta de rede. A vinculação de *container* permite vincular vários *containers* e enviar informações entre estes (PAHL *et al.*, 2016).

Dentre as tecnologias de *container*, o *Docker* destaca-se por suas funcionalidades de gerenciamento de ciclo de vida de aplicações, *pipelines* de integração contínua ou implantação contínua (CI / CD), além disso permite utilizar ferramentas de orquestração, como o *kubernetes* e o *Docker Swarm*.

Ismail *et al.* (2015), afirma que o *Docker* estabelece uma plataforma de computação de borda com suporte descentralizado, reduzindo o tempo de resposta da aplicação e melhorando a experiência geral do usuário (QoE). Bellavista e Zanni (2017) argumentam que a conteinerização com o *Docker* é uma abordagem sólida para implantar aplicações em ambientes *fog*.

Da mesma forma, Brogi, Forti e Ibrahim (2017), utilizando *Docker*, implementaram um protótipo chamado *FogTorch* cujo objetivo era atender a requisitos específicos de *hardware*, *software* e QoS. Já os autores, Jiang, Huang e Tsang (2018), fazendo o uso desta mesma tecnologia, descreveram desafios e soluções para orquestrar aplicações em *fog computing*, considerando os requisitos que as soluções de orquestração devem atender, os autores também fazem uso de soluções de *cloud computing* existentes adaptando-as para aplicações da *fog*.

De acordo com Bellavista e Zanni (2017), o *Docker* como tecnologia de conteinerização oferece uma extensão de alto nível dos recursos do LXC, garantindo ao mesmo tempo um uso razoavelmente e leve de recursos, com resultados de desempenho comparáveis ao próprio LXC. Os autores também afirmam que o *Docker* é baseado em uma grande comunidade de desenvolvedores, sendo uma vantagem significativa se comparado aos concorrentes. Ainda de

acordo com estes o CRIU não foi considerado porque, no momento, ele suporta apenas a migração de processos e não uma solução de *containers* de suporte completo.

Pelas razões supracitadas entre outras evidências encontradas na literatura adotaremos neste estudo o *Docker* como tecnologia de conteinerização.

2.7 Orquestração de *Containers*

Os *containers* tornaram-se reconhecidos como uma opção de virtualização mais leve, uma técnica muito eficiente de empacotamento de aplicações e de execução em um ambiente isolado. As aplicações baseadas em microsserviços geralmente são compostas por *clusters* de centenas de instâncias de serviços em *container*. Para beneficiar-se idealmente dos benefícios dos *containers*, o *cluster* deve ser tolerante a falhas, disponível, confiável, dimensionável e potencialmente disperso geograficamente. O processo de orquestração resulta em plano de provisionamento de recursos que define onde os serviços específicos são implementados e em quais serviços as solicitações de tarefas recebidas são processadas (SILVA, D. S., 2018). As plataformas de orquestração de *containers* podem ser amplamente definidas como um sistema que fornece uma estrutura de nível corporativo para integrar e gerenciar *containers* em escala. Essas plataformas simplificam o gerenciamento de recursos e fornecem uma estrutura não apenas para definir a implantação inicial de um *container*, mas também para gerenciar vários *containers* como uma entidade para fins de disponibilidade, dimensionamento e rede (KHAN, 2017).

Alguns dos principais recursos de uma plataforma de orquestração de *containers* são:

- Gerenciamento e agendamento de estado do *cluster*;
- Fornecer alta disponibilidade e tolerância a falhas;
- Garantir segurança;
- Simplificação da rede;
- Permitir a descoberta de serviço;
- Tornar possível a implantação contínua;
- Fornecer monitoramento e governança.

Para implantar e gerenciar aplicativos e escalonar com *containers*, muitas ferramentas de orquestração, de código aberto, foram desenvolvidas por empresas de TI nos últimos dois anos, como, por exemplo, o *Google Kubernetes*, o *Apache Mesos Marathon*, o *Docker Swarm.* Entre as tecnologias de *container*, o *Docker* é o padrão de fato utilizado por plataformas que utilizam tecnologia na nuvem. Tecnologias de orquestração de *containers* são suportada pelos principais provedores de computação na nuvem (*AWS*, *Azure* e *Google cloud*) são indispensáveis para cenários produtivos (GOGOUVITIS *et al.*, 2018), (HOQUE *et al.*, 2017).

2.7.1 *Docker Swarm*

O *Docker Swarm* é uma ferramenta nativa para gerenciar o armazenamento em *cluster* do *Docker*, fácil de usar e flexível. Além disso, expõe o *Docker* API para ser usado por outras ferramentas do *Docker*, por exemplo, o *Docker Compose*[1]. O *Docker Swarm* é uma ferramenta relativamente nova, que está se desenvolvendo rapidamente e ganhando impulso, mas, como toda aplicação ainda precisa de melhorias, como o suporte a agendamento complexo, para ser usado em ambiente de produção ou em sistemas de larga escala. No entanto, estendê-la para atender às necessidades específicas de orquestração de *containers* em ambientes *fog* é possível (BELLAVISTA, ZANNI, 2017), (BRITO, de *et al.*, 2017), (HOQUE *et al.*, 2017).

Por meio de comandos da CLI (*command-line interface*) do *Docker*, é possível criar um *cluster* de nós de *fog*, adicionar e removê-los. Além disso, o *Docker Swarm* pode ser executado em várias plataformas, incluindo o *Raspberry Pi*.

Dentre as principais características ao se utilizar o *Swarm* destacam-se:

- Integração nativa com *Docker Engine*. Nenhum software adicional é necessário. O único requisito é a versão do *Docker* 1.12.0 ou superior.
- *Design* descentralizado em relação aos *nodes* do *cluster*. *Swarm* não exige que os *hosts* adicionados ao *cluster* sejam do mesmo tipo ou dimensão. Além disso, *Swarm* abstrai as camadas inferiores (*host*) do tempo de *deploy*. Todo ajuste necessário é realizado automaticamente em tempo de execução e dessa maneira, é possível criar um *cluster* a partir de um único disco, por exemplo, e ir expandindo a capacidade conforme a necessidade em tempo de execução.

[1] Ferramenta para definição e execução de múltiplos *containers* Docker a partir de um único arquivo de configuração contendo os parâmetros necessários para implementação de cada *container*.

- Modelo de serviço declarativo. Você pode compor uma aplicação completa (*frontend*, *backend*, banco de dados, por exemplo) de maneira declarativa, utilizando *docker-compose* para isso.
- Escalabilidade. O *Docker Swarm* permite escalar (para mais ou para menos) o número de *containers* executando cada instância da aplicação de uma maneira simples, como será visto a diante.
- Gerenciamento de estado. O *host* identificado como *manager* no *cluster* terá como uma de suas responsabilidades monitorar o estado dos *containers* em execução visando garantir a confiabilidade do *cluster* de acordo com a política de escalabilidade determinada.
- *Multi-networking*. Ao realizar o *deployment* de um novo serviço é possível criar camadas de rede sobre a rede interna do *cluster*. De maneira automática o node *manager* atribuirá IPs internos a estes novos *containers* entrantes no instante em que forem iniciados.
- Resolução automática de DNS. Quando um novo serviço é enviado para o *cluster Swarm*, o *node manager* automaticamente atribui um nome de domínio único (registrado em seu servidor DNS embutido) para este. Ao fazer isso, uma camada de balanceamento é gerada para que qualquer requisição para o serviço seja automaticamente resolvida e a carga automaticamente distribuída entre os *nodes* do *cluster*.
- Balanceamento de cargas. É possível expor um serviço para a internet para que requisições externas sejam distribuídas diretamente entre os *containers* em execução no *cluster*, por meio de um algoritmo de distribuição de cargas conveniente.
- Atualização de serviços em execução (*rolling updates*). É relativamente simples fazer qualquer tipo de atualização em um dado serviço em execução.

2.7.2 Kubernetes

O *Kubernetes* é uma ferramenta de orquestração, de código aberto, geralmente usada como um mecanismo de *clustering* para definir a organização de *containers* em uma aplicação. Segundo Bellavista e Zanni (2017), o *Kubernetes* demonstrou alcançar um bom desempenho e

de forma escalável, sem adicionar uma sobrecarga significativa aos *containers* existentes. Além disso, graças à sua arquitetura de *plug-in*, integra-se facilmente às ferramentas e tecnologias de fornecedores diferentes. Da mesma forma, os mecanismos de autocorreção, reinicialização automática, replicação e reprogramação do *Kubernetes* o tornam robusto e adequado para aplicativos baseados em *containers*. As desvantagens do *Kubernetes* estão relacionadas à complexidade de configuração e as variações nos procedimentos de instalação de plataforma para plataforma (ISMAIL *et al.*, 2015), (BELLAVISTA, ZANNI, 2017).

É possível executar o *Kubernetes* em infraestruturas locais ou em nuvem pública. O *Kubernetes* configura um cluster que consiste em *Kubernetes-master*, *Kubernetes-minions* (*worker nodes*) e *Pod* (único ou grupo de *containers* para executar uma única aplicação). O *Kubernetes-master* supervisiona um ou mais *minions* e gerencia as aplicações. O *Kubernetes* usa sua própria ferramenta chamada *kubelet* para agrupar os nós, onde todos os nós são *hosts* do *Docker*. Usando o modo *multinode* do *Docker* e a ferramenta *kubeadm* é possível criar um *cluster* personalizado, adicionar e remover nós do *cluster* (HOQUE *et al.*, 2017).

2.7.3 Apache Mesos Marathon

O *Apache Mesos Marathon* é uma plataforma de orquestração de *container*, de baixo nível, portátil e muito confiável. Ela foi projetada para funcionar no *Apache Mesos* e em sistemas operacionais para *datacenter*. O *Apache Mesos* atua como um sistema operacional de *datacenter* possibilitando a visualização de um *datacenter* como um único *pool* de recursos. *Marathon* executa em cima do *Mesos* e orquestra aplicações baseadas em *container*. O *Apache Mesos Marathon* possui alto desempenho e é capaz de escalonar para *clusters* muito grandes, mesmo que tenha uma sobrecarga alta de recursos. Embora seja muito interessante para outros cenários de implantação, devido à sobrecarga para os nós físicos do cluster, não é recomendável o uso em alguns domínios de aplicações IoT e na *fog computing*. (BRITO, de *et al.*, 2017), (HOQUE *et al.*, 2017).

2.7.4 Considerações sobre as ferramentas de orquestração

O *Docker Swarm* representa a melhor relação entre boa solidez em nível de mercado, alto desempenho e sobrecarga limitada. De acordo com os estudos realizados em (BELLAVISTA, ZANNI, 2017), (HOQUE *et al.*, 2017), (BRITO, de *et al.*, 2017), o *Docker Swarm* demonstrou

fornecer um bom desempenho geral, conta com uma comunidade relativamente grande de usuários por trás dele que proporcionam melhorias constantes, e do ponto de vista do consumo de recursos, mostrou-se mais leve do que *Marathon* e *Kubernetes*.

Segundo Hoque *et al.* (2017), o *Kubernetes* possui limitações, pois, utiliza *containers* em grupos. A unidade de implementação denominada *Pod*, pode conter um ou mais *containers*. Então, se a aplicação precisar ser acessada de fora, a "implantação" tem que ser exposta usando o serviço. O mapeamento de serviço para *Pod* a *Container* ajuda a aumentar a escala. Já para aplicações IoT em execução no *container*, precisam de acesso aos recursos dos nós da *fog*, o que não é possível devido à camada de abstração de *Pod* e serviço pertencer à parte superior do *container*.

Em relação ao *Apache Mesos Marathon,* este não possui suporte para todas as plataformas, especialmente, as de SBC, como, por exemplo, o *Raspberry Pi*, que utilizaremos nessa dissertação. Além disso, o *Apache Mesos Marathon* foi projetado para *cluster* maiores e sistemas operacionais de *datacenter* que não pertencem ao escopo deste trabalho.

Assim concluí-se que o *Marathon* (MARATHON, 2019) e *Kubernetes* (KUBERNETES, 2019) podem fornecer orquestradores mais completos e robustos, não apenas limitados a *containers Docker*, no entanto, com base no objetivo de nossa proposta é necessária a adequação de uma solução mais leve, e por vezes, mais fácil de ser integrada a mecanismos e políticas adicionais, projetados especificamente para o objetivo de orquestrar recursos na *fog computing.*

3
Estado da Arte

A *fog computing* é uma nova tendência na computação que visa processar dados perto da fonte. Ela processa aplicações, serviços, dados, poder de computação e tomada de decisão dos nós centralizados nos extremos lógicos da rede. A *fog computing* diminui significativamente o volume de dados que deve ser movido entre os dispositivos finais e a nuvem, e permite que a análise de dados e a geração de conhecimento ocorram na origem dos dados. Além disso, a distribuição geográfica densa da *fog* ajuda a obter melhor precisão da localização para muitas aplicações em comparação com a computação na nuvem (MUNIR, KANSAKAR, KHAN, 2017).

Os nós da *fog* que pertencem à borda podem manter um ambiente de execução colaborativa, formando um *cluster* entre eles. Os *clusters* podem ser formados com base na homogeneidade dos nós da *fog* (CARDELLINI *et al.*, 2015), ou conforme sua localização (Hou *et al.*, 2016). O balanceamento de carga computacional (Oueis, Strinati e Barbarossa, 2015), e o desenvolvimento do subsistema funcional (AL FARUQUE, VATANPARVAR, 2016), também podem receber maior prioridade ao formar um *cluster* entre os nós.

De acordo com Mahmud, Kotagiri e Buyya (2018), a colaboração baseada em *cluster* é eficaz na exploração de capacidades de vários nós *fog* simultaneamente. No entanto, é difícil fazer com que os *clusters* estáticos sejam escaláveis em tempo de execução e a formação dinâmica de *clusters* depende, em grande parte, da carga existente e da disponibilidade de nós da *fog*. Em ambos os casos, a sobrecarga de rede desempenha um papel vital.

Neste capítulo são identificados esforços de pesquisas e alguns projetos que abordam questões semelhantes à deste trabalho onde procurou-se identificar e discutir estudos no domínio da *fog computing* em termos de orquestração de aplicações cuja finalidade é atingir níveis de serviço, especializado em *Service Level Objectives* (SLOs) aceitáveis, como os relacionados ao gerenciamento de problemas de latência, escalabilidade, balanceamento de carga e alta disponibilidade.

3.1 Mapeamento sistemático da Literatura

Os trabalhos de pesquisa relacionados foram selecionados seguindo critérios predefinidos do mapeamento sistemático, que consiste em basear as respostas das questões de pesquisa em evidências. Por este fato, Kitchenham (2004) apresentou o paradigma denominado engenharia de software baseada em evidências. O princípio deste paradigma é responder as questões de pesquisa dos trabalhos desenvolvidos, com base em evidências encontradas na literatura.

Com o objetivo de estudar e mapear o estado da arte acerca das arquiteturas em *fog computing* capazes de fornecer um ambiente de orquestração para aplicações destinadas a Internet das Coisas utilizando *cluster* de plataformas *single board computer* respeitando critérios de QoS, foi adotado neste trabalho a metodologia de mapeamento sistemático da literatura. Segundo Petersen *et al.* (2008) o mapeamento sistemático consiste em definir questões de pesquisa, realizar a busca e seleção dos estudos relevantes, extrair dados e mapear os resultados. Kitchenham (2004) adicionalmente, define o método de mapeamento sistemático como uma ampla revisão dos estudos primários relevantes para um problema de pesquisa específico e tem como objetivo identificar os estudos disponíveis e abordagens utilizadas em alguma área específica.

Para executar o mapeamento foram utilizadas três fases: planejamento, condução e relatório do estudo (KITCHENHAM, 2004). Como apoio ao processo de mapeamento foi utilizada a ferramenta *Parsifal*[2] (*Perform Systematic Literature Reviews*).

A fase do planejamento visa definir os objetivos a serem atingidos pela pesquisa e o protocolo do mapeamento, precisando o método de pesquisa e permitindo a reprodutibilidade do estudo, além de reduzir riscos de ocorrerem vieses nos resultados (STEINMACHER, CHAVES, GEROSA, 2013).

O protocolo desenvolvido foi constituído de objetivo, questões de pesquisa, estratégia PICOC (*Population, Intervention, Control, Outcome, Context*), *string* de busca, critérios de inclusão e exclusão dos trabalhos encontrados na literatura e critérios de seleção das bases de pesquisa das publicações.

[2] http://parsif.al

Os estudos primários selecionados devem contribuir diretamente para a resposta das questões de pesquisa do mapeamento para que esses trabalhos não sejam desconsiderados pela estratégia de busca (KITCHENHAM, 2004).

A definição das palavras-chaves de busca, utilizadas na pesquisa, foi feita com base na estratégia PICOC (PETTICREW, ROBERTS, 2008), conforme Quadro 1.

Quadro 1 - Estratégia PICOC

PICOC	**KEY WORDS**
População (*Population*)	*Fog Computing, cloudlets, edge computing*
Intervenção (*Intervention*)	*Orchestration, cluster, clustering, security, embedded system, single board computer*
Controle (*Control*)	Não definido
Resultado (*Outcome*)	*Methodology, methodologies, technique, method, prototype, protocol, architecture*
Contexto (*Context*)	*Internet of Thing*

Fonte: Próprio autor, 2019

Mediante as palavras-chave identificadas foi possível derivar uma *string* de busca genérica, que foi utilizada como estrutura para gerar as *strings* específicas, contemplando as particularidades sintáticas de cada uma das bases de pesquisa (*ACM, IEEEXplorer, ScienceDirect, Scopus, Web of Science*) escolhida para este mapeamento. A *string* de busca genérica consiste em: ((fog computing OR cloudlets) AND (edge AND computing)) AND (Orchestration OR cluster OR clustering OR security) OR (embedded AND system) OR (single AND board AND computer)) AND (methodology OR methodologies OR technique OR method OR prototype OR protocol OR architecture).

Na condução a primeira atividade executada do mapeamento foi a obtenção dos estudos. Para isso, foi executada, a *string* de busca por base, os resultados obtidos foram exportados para um arquivo do tipo *BibTex*, e importados para a ferramenta de apoio *Parsifal*. As buscas foram executadas em outubro de 2018 e foi obtida uma quantidade total de 511 estudos. O Quadro 2 mostra a distribuição destes resultados pelas bases utilizadas.

Quadro 2 - Resultados obtidos por base de pesquisa

#	Biblioteca Digital	Endereço Web	Resultados
1	ACM Digital Library	http://dl.acm.org/	84
2	IEEE Xplorer	http://ieeexplore.ieee.org/search/advsearch.jsp?expression-builder	213
3	Science Direct	http://www.sciencedirect.com/science/search	89
4	Scopus	http://www.scopus.com/home.url	87
5	Web of Science	http://apps-webofknowledge.ez141.periodicos.capes.gov.br	38

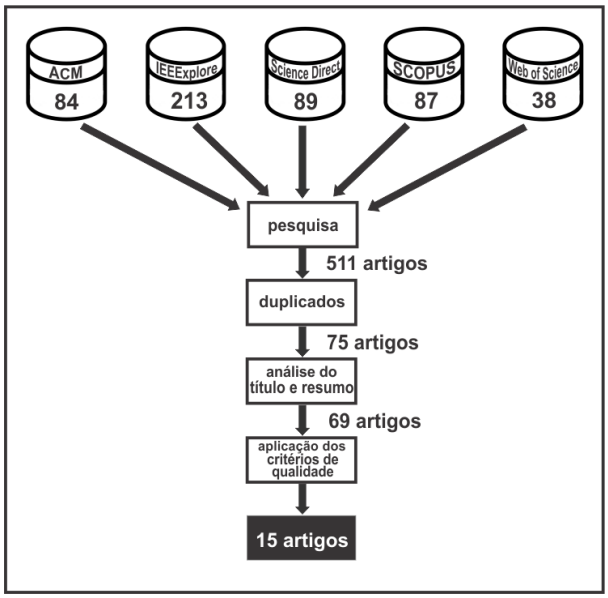
Fonte: Próprio autor, 2019

Dentre estes, foram localizados 75 estudos duplicados, identificados e excluídos por meio da ferramenta *Parsifal*, resultando em 436 estudos potencialmente relevantes, ou seja, encontrados na pesquisa, para aplicação dos critérios de inclusão e exclusão.

Em seguida, foi efetuada a leitura dos títulos e resumos dos artigos resultantes e aplicados os critérios de inclusão e exclusão. Neste processo foram excluídos 367 estudos, restando 69 estudos relevantes para aplicação dos critérios de qualidade segundo (Petersen *et al.*, 2008).

A verificação da qualidade dos estudos relevantes foi ratificada por meio de formulação de perguntas de caráter qualitativo à que os estudos relevantes foram submetidos, resultando em 15 artigos conforme fluxo de condução do mapeamento sintetizado na Figura 4.

Figura 4 - Fluxo de condução do mapeamento sistemático



Fonte: Próprio autor, 2019

3.2 Trabalhos de pesquisa relacionados

O desenvolvimento de aplicações que demandam recursos e o crescimento exponencial de aplicativos dentro do contexto dos paradigmas Cidades Inteligentes e Internet das Coisas levam à busca de recursos que, até certo ponto, foram cobertos pelo uso do paradigma da computação em nuvem. A nuvem representa um conjunto logicamente centralizado de recursos que são explorados por aplicações que consomem muitos recursos, no entanto, esta solução não satisfaz as necessidades de todas as aplicações. A *fog* é uma extensão da nuvem que aproxima os recursos dos usuários, até a borda da rede. Esse paradigma foi concebido para abordar aplicações e serviços que não se encaixam bem no paradigma da nuvem trazendo benefícios como baixa latência, geodistribuição, mobilidade, alta resiliência e sistemas distribuídos em larga escala (VELASQUEZ *et al.*, 2017).

Gogouvitis *et al.*, (2018) propuseram uma arquitetura de referência que suporte funcionalidade da computação integrada baseada em *containers* e no *framework* de orquestração *Kubernetes*. A maioria dos requisitos de alto nível da computação integrada é atendida na implementação do protótipo proposto, com exceção do suporte à mobilidade de carga de trabalho dinâmica e cargas de trabalho em tempo real. Os autores afirmam que mesmo com essas restrições, a plataforma implementada pode ser usada para aumentar a eficiência, a flexibilidade do desenvolvimento e a implantação de um conjunto de aplicações industriais, como, por exemplo, sistemas *Supervisory Control and Data Acquisition* (SCADA).

Li *et al.*, (2018) apresentaram uma política de *cluster cloudlet* móvel dinâmica (DMCCP) para o descarregamento de servidor *fog*. Esta política faz com que o servidor *fog* agrupe dinamicamente o *cloudlet* correspondente a diferentes demandas de recursos da tarefa do dispositivo móvel local. Todos os dispositivos móveis disponíveis, que são recursos potenciais de TI para o *cloudlet*, devem ser pré-conectados a um servidor *fog* por meio da rede sem fio. Para usar com eficiência o *pool* de recursos de TI em potencial, este trabalho faz uso do monitor de recursos móveis (MRM) no servidor *fog* para observar a quantidade de recursos disponíveis de cada dispositivo móvel. As informações do dispositivo móvel, como a unidade de processamento central (CPU), a memória e o nível da bateria, são enviados ao servidor *fog* para monitoramento por meio do MRM. Se um único dispositivo móvel solicitar uma tarefa ao servidor *fog*, este agrupará um *cloudlet* por DMCCP com base nas demandas de recursos dessa tarefa específica.

Abreu *et al.*, (2018) propuseram um algoritmo de escalonamento simples para escalonamento de tarefas em um ambiente inter *fog*, ou ambiente federativo *fog*, onde os *Fog Services Providers* (FSPs) podem capitalizar na cooperação entre instâncias da *fog*, ou *cloudlets*, enquanto também alcança balanceamento de carga entre nós da *fog*.

Bellavista e Zanni, (2017) projetaram, implementaram e testaram uma solução de *middleware* inovadora em *fog computing* baseada em duas direções primárias de melhoria de nós de *gateway*: i) escalabilidade extensões do *gateway* IoT fornecido pela estrutura *Kura*, de código aberto, com o objetivo de descentralizar funcionalidades do *broker* MQTT da estrutura *Kura* da nuvem para as bordas envolvidas; ii) *Container* com base no *Docker* sobre dispositivos *RaspberryPi*, objetivando explorar a conteinerização para facilitar a interoperabilidade e a portabilidade por meio da padronização da configuração de nós em termos de definição de macro serviços.

Wen *et al.*, (2017) apresentam um algoritmo genético paralelo (GA-Par) em *fog computing* capaz de orquestrar serviços de IoT. A principal responsabilidade do orquestrador é selecionar recursos e implantar o fluxo de trabalho de serviço geral de acordo com os requisitos de segurança de dados, confiabilidade e eficiência do sistema.

Velasquez *et al.*, (2017) definiram uma arquitetura para o gerenciamento de ambientes *fog*, levando em consideração uma abordagem híbrida incluindo orquestração e coreografia. A orquestração é usada na região do encadeamento norte entre a *fog* e a nuvem, para ter uma visão global do sistema. Por outro lado, a coreografia será usada na região de encadeamento sul da *fog*, permitindo respostas rápidas automatizadas nos níveis mais baixos. As contribuições apontadas pelos autores foram: a proposta de uma abordagem híbrida para orquestração e coreografia de serviços na *fog*; e uma arquitetura para o orquestrador, seguindo a abordagem híbrida proposta.

Brito, de *et al.*, (2017) apresentaram uma arquitetura para gerenciamento de infraestrutura e orquestração de serviços com base nos principais requisitos da *fog computing*, baseado em um ambiente virtualizado, onde os nós *fog* são capazes de executar aplicações, serviços virtualizados e conteinerizados, oferecendo acesso a dispositivos conectados, sobre diferentes tecnologias de comunicação, para realizar suas tarefas.

Uma avaliação de como os *containers* podem afetar o desempenho geral das aplicações em nós da *fog*, foi proposta em (HOQUE *et al.*, 2017). Além da avaliação os autores analisaram diferentes ferramentas de orquestração de *container* e como elas atendem aos requisitos da *fog*

para executar aplicações, com isso, propuseram um *framework* de orquestração de *container* para infraestruturas da *fog computing*.

Santoro *et al.*, (2017) descreveram a *Foggy*, uma estrutura arquitetônica e plataforma de *software* para orquestração de carga de trabalho e negociação de recursos em um sistema de computação na nuvem multicamada, altamente distribuído, heterogêneo e descentralizado. Através da apresentação de três casos de uso, a *Foggy* provou ser capaz de orquestrar a carga de trabalho em um ambiente *fog computing*, atuando como um mediador entre o proprietário da infraestrutura e os locatários, melhorando: i) o uso eficiente, efetivo e eventualmente otimizado da infraestrutura e ii) o desempenho da aplicação para satisfazer os requisitos impostos.

O projeto europeu H2020 BEACON serviu de base para a pesquisa realizada em (Villari *et al.*, 2017), cuja proposta foi a orquestração para implantação de serviços de distribuição de *edge computing*. A abordagem descrita orienta a implantação do serviço de *edge computing* em um ambiente de rede da *cloud* federada por meio de um manifesto do serviço *Heat Orchestration Template* (HOT), que é analisado pelo *Orchestration Broker* que extrai automaticamente todos os elementos que descrevem como microsserviços relacionados, os quais devem ser implantados em *clouds* federadas por meio de módulos. Um recurso importante dessa abordagem é a capacidade de selecionar a *cloud* federada de destino como função da posição geográfica de *clouds* federadas que permite implantar microsserviços nas bordas da rede.

Zeng *et al.*, (2016) propuseram um modelo capaz de investigar o problema de minimização de tempo de conclusão de tarefa na *Fog Computing Supported Software-Defined Embedded System* (FC-SDES), por meio da consideração conjunta do posicionamento da imagem da tarefa e do agendamento de tarefas. Para lidar com a alta complexidade computacional, os autores apresentaram um algoritmo heurístico em três estágios. Os dois primeiros estágios lidam individualmente com o tempo de E/S e o tempo de computação, transformando-os em problemas do tipo min-max de programação näo-linear inteira mista, do inglês *mixed-integer nonlinear programming* (MINLP). O terceiro estágio reúne o processamento de I/O e o cálculo da tarefa, juntamente com a incorporação do tempo de transmissão.

Motivados por um estudo de caso em *resorts* de esqui modernos, os quais operam com infraestruturas de alta latência, coletando informações da região local, como temperatura e umidade do ar, intensidade do sol, umidade e temperatura da neve, localização e número de

pessoas, os autores Pahl *et al.*, (2016) propuseram um *middleware* de PaaS de *edge computing* como gerenciamento de *container* e *cluster* em *Raspberry Pi*. A arquitetura está organizada em três camadas: o nível mais baixo é composto pelos dispositivos IoT; o nível intermediário é responsável pela rede de área de campo e infraestrutura básica de IP, e no nível superior computação virtual e armazenamento em nuvem.

Em (OUEIS *et al.*, 2015), os autores propuseram um método de otimização de agrupamento de pequenas células multiusuário em *fog computing* distribuída. Esta abordagem parte de dois pressupostos. Primeiro, permite o dimensionamento adaptativo e o gerenciamento de recursos de *clusters* de computação. Em segundo, estabelece simultaneamente *clusters* de computação para todas as solicitações ativas para uma melhor exploração dos recursos disponíveis, visando uma maior QoE. O método proposto é uma otimização conjunta da distribuição de carga computacional, alocação de taxa computacional e controle de potência de transmissão.

Cardellini *et al.*, (2015) apresentaram uma avaliação experimental completa do escalonador distribuído QoS-aware para sistemas *data stream processing* (DSP), baseados em *Storm*, que é capaz de operar em um ambiente distribuído de computação em *cloud*. Os autores utilizaram dois conjuntos de aplicações: o primeiro é baseado em uma topologia de referência com requisitos diferentes; o segundo em algumas aplicações bem conhecidas. Os resultados mostram que o escalonador proposto supera o padrão da *Storm*, melhorando o desempenho da aplicação e aprimorando o sistema com recursos de adaptação.

Oueis, Strinati e Barbarossa, (2015) definiram um algoritmo customizável para estabelecimento e gerenciamento de recursos de *clusters* de pequenas células de baixa complexidade para *clustering* da *fog* para melhorar a QoE dos usuários, abordando a questão do balanceamento de carga.

3.3 Análise dos Resultados.

Diante dos estudos selecionados, conforme especificações definidas no mapeamento sistemático, pode-se fazer uma análise comparativa de acordo com um conjunto de critérios de avaliação concisos, tidos como essenciais, para ambientes *fog computing* (MOURADIAN *et al.* 2017). Estes critérios são: heterogeneidade, gerenciamento de QoS, escalabilidade, mobilidade, federação e interoperabilidade; os quais são resumidos no Quadro 3. Dessa forma, entendemos

que estes critérios, devem, intrinsicamente, estar presentes em domínios que utilizem orquestração como micro-PaaS na *fog computing*.

Quadro 3 – Resumo critérios de avaliação

Critério	Definição
C1 - Heterogeneidade	Nós da camada da *fog* e da camada da *cloud* são muito heterogêneos em termos computacionais e capacidade de armazenamento. A *fog* deve ser capaz de lidar com a heterogeneidade.
C2 - Gerenciamento de QoS	A *fog* é uma solução promissora para aplicações em tempo real devido à proximidade com os dispositivos IoT e usuários finais. No entanto, a latência varia muito dependendo de onde os componentes do aplicativo estão localizados, o que exige o gerenciamento de QoS
C3 - Escalabilidade	É esperado que a *fog* cubra milhões de dispositivos de usuários finais. Além disso, eles podem abranger um grande número de aplicações, domínios e nós da *fog*. Algumas aplicações também podem ter um grande número de componentes. A *fog computing* precisa estar operacional em grande escala e deve aumentar e diminuir de forma elástica.
C4 - Mobilidade	Dispositivos IoT/usuários finais e nós da *fog* podem ser móveis. Sistemas *fog* devem ser capaz de lidar com essa mobilidade
C5 - Federação	A *fog* distribui geograficamente a implantação em larga escala, em que cada domínio da *fog* pode pertencer a um provedor diferente, bem como os diferentes componentes da *cloud*. As aplicações de provisionamento exigem a federação desses diferentes provedores, os quais podem hospedar diferentes componentes.
C6 - Interoperabilidade	Como parte de um sistema federado, uma aplicação pode ser executada com seus componentes espalhados por diferentes provedores. A *fog computing* deve ser interoperável no nível de provedores e módulos arquitetônicos.

Fonte: Adaptado de Mouradian *et al.* (2017)

A luz da literatura pode-se observar, no Quadro 4, as evidências destes critérios nos estudos selecionados, para obtenção de uma visão geral dos *gaps* que os estudos relevantes não abordaram fazendo uma correlação com a nossa proposta de dissertação de mestrado.

Quadro 4 - Principais características e requisitos para orquestração eficiente em ambientes *fog computing*.

Artigo	**Contribuição Principal**	**Validação**[3]	**Métricas avaliadas**	**Critérios**[4]					
				C1	**C2**	**C3**	**C4**	**C5**	**C6**
(Gogouvitis *et al.*, 2018)	Propomos uma arquitetura de referência que suporte a funcionalidade da computação integrada baseada em *containers*	P	▪ latência ▪ carga de trabalho ▪ tempo de migração	✓	x	x	x	x	x

[3] Na coluna validação: P = protótipo; S = simulação;

[4] Na coluna critérios: ✓ = critério atendido; x = critério não atendido

	e no framework de orquestração Kubernetes.								
(Li *et al.*, 2018)	Propor uma política de cluster cloudlet móvel dinâmica (DMCCP) para o descarregamento do servidor de névoa. Esta política fará com que o servidor de névoa agrupe dinamicamente o cloudlet correspondente a diferentes demandas de recursos da tarefa do dispositivo móvel local.	S	▪ recursos computacionais ▪ taxa de transferência	x	✓	x	✓	x	x
(Abreu *et al.*, 2018)	Propor um algoritmo de escalonamento simples para escalonamento de tarefas em um ambiente federativo *fog*, onde os Fog Services Providers (FPSs) podem capitalizar na cooperação entre instâncias, enquanto, também, alcança balanceamento de carga entre nós da *fog*.	S	▪ recursos computacionais ▪ latência	x	✓	x	x	✓	x
(Bellavista e Zanni, 2017)	Projetar, implementar e testar uma solução de Middleware inovadora da *fog computing* baseada em duas direções primárias de melhoria de nós de gateway: escalabilidade e conteinerização	P	▪ latência ▪ tempo de execução	x	x	✓	x	x	✓
(Wen *et al.*, 2017)	Propuseram um algoritmo genético paralelo (GA-Par) em ambiente *fog* capaz de orquestrar serviços de IoT para determinar e selecionar os melhores dispositivos para composição dinâmica de fluxos de trabalho holístico para funções mais complexas.	S	▪ segurança ▪ desempenho	x	✓	✓	x	x	x
(Velasquez *et al.*, 2017)	Apresentar uma arquitetura para o gerenciamento de ambientes *fog*, levando em consideração uma abordagem híbrida incluindo orquestração e coreografia.	x	▪ latência ▪ resiliência ▪ escalabilidade	x	✓	✓	x	x	✓
(Brito, de *et al.*, 2017)	Apresentar uma arquitetura para gerenciamento de infraestrutura e orquestração de serviços com base nos principais requisitos da *fog computing*.	P S	▪ recursos computacionais ▪ tempo de orquestração ▪ oportunidades de perdas ▪ tempo de escalação ▪ tempo de migração	x	✓	x	x	x	✓
(Hoque *et al.*, 2017)	Propor um Framework de Orquestração de *Containers* para infraestruturas de *fog computing*	x	▪ recursos computacionais ▪ tempo de escalação	x	✓	✓	x	x	x
(Santoro *et al.*, 2017)	Apresentar uma estrutura arquitetônica e plataforma de software para orquestração de	P S	▪ carga de trabalho	✓	x	x	x	✓	✓

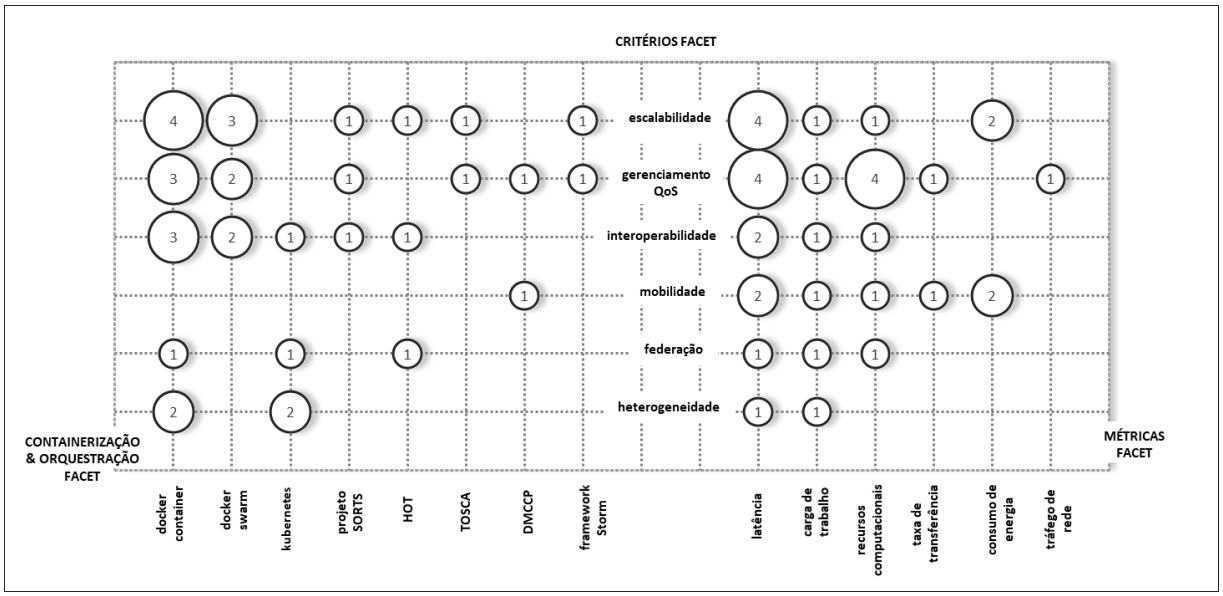
	carga de trabalho e negociação de recursos em um sistema de *fog computing*.								
(Villari *et al.*, 2017)	Propor uma orquestração para a implantação de serviços de distribuição de *edge computing*, em um ambiente *cloud* federada por meio de um manifesto do serviço Heat Orchestration Template (HOT)	P	▪ tempo de implementação ▪ tempo de análise	x	x	✓	x	✓	✓
(Zeng *et al.*, 2016)	Propor um modelo capaz de investigar o problema de minimização máxima de tempo de conclusão de tarefa na fog computing supported software-defined embedded system (FC-SDES) por meio da consideração conjunta do posicionamento da imagem da tarefa e do agendamento de tarefas.	S	▪ latência ▪ carga computacional	x	✓	x	x	x	x
(Pahl *et al.*, 2016)	Propor uma arquitetura de PaaS de *edge computing* como gerenciamento de *container* e cluster em Raspberry Pi.	P	n/a	x	✓	✓	x	x	x
(Oueis *et al.*, 2015)	Propor um método de otimização de agrupamento de pequenas células multiusuário em *fog* distribuída, partindo de dois pressupostos: primeiro, permitir o dimensionamento adaptativo e o gerenciamento de recursos de *clusters*; segundo, estabelecer simultaneamente *clusters* de computação para todas as solicitações ativas	S	▪ latência ▪ consumo de energia ▪ carga computacional ▪ alocação de taxa computacional ▪ controle de potência de transmissão	x	x	✓	✓	x	x
(Cardellini *et al.*, 2015)	Apresentar uma avaliação experimental completa do escalonador distribuído QoS-aware para sistemas DSP baseados em Storm, que é capaz de operar em um ambiente distribuído da *cloud computing*.	x	▪ tráfego de rede ▪ latência	x	✓	✓	x	x	x
(Oueis, Strinati e Barbarossa, 2015)	Propor um algoritmo customizável para estabelecimento e gerenciamento de recursos de clusters de pequenas células de baixa complexidade para clustering de neblina para melhorar a QoE dos usuários, abordando a questão do balanceamento de carga.	S	▪ latência ▪ consumo de energia	x	x	x	✓	x	x
Este trabalho	Propor uma arquitetura de micro-PaaS em *fog computing* utilizando orquestração baseada em container para aplicações IoT utilizando *single board computer*.	P	▪ latência ▪ balanceamento de carga ▪ alta disponibilidade	✓	✓	✓	✓	✓	✓

Fonte: Próprio autor, 2019

Embora muitos aspectos importantes da *fog* tenham sido identificados em nossa pesquisa, de acordo com o mapeamento, existem alguns outros direcionamentos que precisam ser abordados e melhorados. Para realizar uma análise mais profunda, fora elaborado um gráfico de bolhas de três dimensões para analisar, juntas, as facetas: “critérios”, “conteinerização & orquestração” e “métricas”. Os números dentro das bolhas representam a quantidade de trabalhos relacionados que abordaram em suas pesquisas os assuntos atinentes às respectivas facetas. A faceta “critérios” traz os critérios de avaliação necessários para o ambiente *fog computing de acordo com* Mouradian *et al.* (2017); a faceta “conteinerização & orquestração” abordas as tecnologias de *containers* e as técnicas de orquestração de maior incidência na literatura, já a faceta “métricas” reflete as métricas de maior relevância encontrada nos estudos relacionados.

As bolhas no topo esquerdo da Figura 5 mostram que a técnica de conteinerização “*Docker Container*” em conjunto com o critério de “escalabilidade” constitui-se em um dos focos mais comum dos estudos analisados. Ainda desse mesmo lado é observado que a técnica de orquestração “*Docker Swarm*” e “escalabilidade” também tem sido um foco frequente de pesquisa. Ainda, da Figura 5 é possível concluir que “mobilidade” e “D*ocker Container*” não tiveram nenhuma bolha associada, assim como, “mobilidade”, “federação” e “heterogeneidade” relacionada com “*Docker Swarm*” reportando gaps que serão atendidos em nossa proposta de dissertação.

Figura 5 - Gráfico de Bolhas



Fonte: Próprio autor, 2019

Já na parte direita do Gráfico de Bolhas, a "taxa de transferência" está relacionada apenas aos critérios "gerenciamento de QoS" e "mobilidade" e a métrica "consumo de energia" está relacionada apenas aos critérios "escalabilidade" e "mobilidade". Isto indica que existem *gaps* de grande relevância que não foram tratados em pesquisas anteriores, e que são abordados nesta dissertação. Vale ressaltar, que a "latência" tem correspondência com todas as facetas de "critérios" o que demonstra um grande interesse da comunidade científica por esta métrica.

Os trabalhos analisados descrevem diferentes tecnologias de conteinerização em ambientes de *fog computing* (GOGOUVITIS *et al.*, 2018), (BELLAVISTA, ZANNI, 2017), (BRITO, de *et al.*, 2017), (HOQUE *et al.*, 2017), (SANTORO *et al.*, 2017), (PAHL *et al.*, 2016), e técnicas de orquestração, como as discutidas nos estudos (GOGOUVITIS *et al.*, 2018), (LI *et al.*, 2018), (BELLAVISTA, ZANNI, 2017), (VELASQUEZ *et al.*, 2017), (BRITO, de *et al.*, 2017), (HOQUE *et al.*, 2017), (SANTORO *et al.*, 2017), (VILLARI *et al.*, 2017), (PAHL *et al.*, 2016), (CARDELLINI *et al.*, 2015).

A literatura apresentada mostra que, embora algumas pesquisas considerem níveis de QoS, questões relacionadas à taxa de transferência foram observadas apenas em Cardellini *et al.*, (2015). O congestionamento da rede ocorre principalmente devido ao aumento da sobrecarga, devido as interações de um grande número de dispositivos IoT, sensores finais, os quais estão distribuídos pela borda, com *datacenters* da *Cloud*, degradando o desempenho do sistema. Da mesma forma, apenas 13 % dos estudos consideram métricas de consumo de energia. Devido à natureza descentralizada e heterogênea da *fog*, obter desempenho desejável não será uma tarefa fácil, em ambientes sem o gerenciamento apropriado dos recursos computacionais. Diferentemente, a métrica latência foi objeto de pesquisa em 50 % dos trabalhos relacionados demonstrando que esta métrica é bastante relevante e vem sendo pesquisada com muita intensidade pela comunidade científica.

Por outro lado, 40 % dos artigos selecionados abordaram o *Docker Container* como tecnologia de conteinerização. Segundo (GOGOUVITIS *et al.*, 2018), (BELLAVISTA, ZANNI, 2017), (SANTORO *et al.*, 2017), (PAHL *et al.*, 2016), os *containers* aparecem como uma tecnologia altamente adequada para empacotamento e gerenciamento de aplicativos em *edge computing* muito mais flexíveis e leves que as VMs. Nesta direção os trabalhos apresentados por (BELLAVISTA, ZANNI, 2017), (BRITO, de *et al.*, 2017), (HOQUE *et al.*, 2017) mostram que a orquestração de *containers* por meio da tecnologia *Docker Swarm*, além de ser compatível com dispositivos de placa única é bem mais leve se comparado com outras tecnologias de orquestração.

Diferente dos trabalhos apresentados, nossa arquitetura é baseada em uma micro-PaaS (Micro Plataforma como Serviço), em *fog computing* utilizando tecnologia de conteinerização *Docker* e técnica de orquestração *Docker Swarm* em um *clustering* de *fog* formados por dispositivos *single board computer*, *Raspberry Pi*. Nossa arquitetura é capaz de fornecer processamento local das requisições de serviço de forma orquestrada com TCO reduzido, selecionando de forma inteligente o melhor nó *da fog* obedecendo a critérios de QoS como baixa latência, escalabilidade, balanceamento de carga e alta disponibilidade (conforme mostrado nos capítulos seguintes).

4

Materiais e Métodos de Pesquisa

Os procedimentos adotados no desenvolvimento deste trabalho são baseados em uma pesquisa experimental, devido à execução de experimentos controlados em laboratório. O objetivo principal deste experimento é implementar e analisar uma arquitetura em *fog computing* que seja capaz de orquestrar aplicações destinadas à Internet das Coisas em um ambiente de rede real composto por dispositivos inteligentes, *clusters* constituídos por SBC, *containers*, aplicações e sistemas de monitoramento executados nos nós da *fog*. Além disso, para apoiar no desenvolvimento desta pesquisa realizou-se o mapeamento sistemático da literatura conforme visto no capítulo anterior.

4.1 Método de pesquisa

A pesquisa experimental consiste em determinar um objeto de estudo, selecionar as variáveis que seriam capazes de influenciá-lo, definir as formas de controle e de observação dos efeitos que a variável produz no objeto (GIL, 2002). As variáveis manipuladas são chamadas de variáveis independentes, e o seu efeito afeta as variáveis chamadas de dependentes.

Para alcançar o objetivo desta pesquisa experimental, foi necessária a realização das atividades a saber:

1. Mapeamento sistemático da literatura;
2. Seleção das plataformas de *hardware*;
3. Escolha dos sistemas operacionais, tecnologias de *containers* e orquestração;
4. Planejamento da arquitetura;
5. Implementação da camada *fog orchestration overlay*;
6. Implantação dos sistemas de gerenciamento e monitoramento do ambiente;
7. Planejamento e desenvolvimento do experimento da camada IoT;

8. Desenvolvimento do sistema de monitoramento para IoT e implantação na camada *fog*;
9. Implementação da camada a *Cloud*;
10. Análise e interpretação dos dados coletados.

Mapeamento Sistemático da Literatura

O Mapeamento Sistemático da Literatura realizado neste trabalho e detalhado no capítulo anterior, tem como objetivo apresentar esforços de pesquisas atuais a fim de compreender, identificar métricas e lacunas na literatura atual sobre orquestração de aplicações para Internet das Coisas utilizando *containers*, baseados em *clusters* de plataformas *single board computer*, como *Raspberry Pi*, em ambientes da *fog computing*.

Seleção das plataformas de *hardware*

Essa atividade engloba a pesquisa, análise e seleção de plataformas de *hardwares* de baixo custo capazes de permitir a implementação de uma arquitetura em *fog computing*, bem como o monitoramento de variáveis de ambiente. As plataformas selecionadas devem suportar tecnologias de *containers* e orquestração de aplicações e serviços para IoT sem comprometer o desempenho do ambiente proposto e a qualidade dos serviços.

Escolha dos sistemas operacionais, tecnologias de contaneirização e orquestração

Seleção dos sistemas operacionais, tecnologias de *containers* e orquestração de acordo com os estudos analisados no mapeamento sistemático e com padrões de mercado, que possam ser aplicados em ambientes para Internet das Coisas e às plataformas de *hardware* definidas.

Planejamento da arquitetura

A atividade contempla o planejamento da arquitetura de micro-PaaS em *fog computing* e suas respectivas camadas pela orquestração das aplicações dos domínios horizontais.

Implementação da camada *fog orchestration overlay*

Nessa etapa foi realizada a implementação e configuração de um *cluster* com alta disponibilidade utilizando a tecnologia *Docker Engine* para compor a camada *fog*. O *cluster* implementado nesta camada é responsável pela orquestração e escalabilidade das aplicações

destinadas à Internet das Coisas. A camada é constituída por seis *Raspberry Pi*, sendo três destes configurados como *managers* e os demais configurados como *workers*.

Implantação dos sistemas de gerenciamento e monitoramento do ambiente

A atividade contempla a implantação de sistemas de gerenciamento e monitoramento, em tempo real, das aplicações, dos *containers*, da infraestrutura e demais serviços que integram a arquitetura. Além disso, o sistema de monitoramento implantado é responsável pela coleta das métricas do ambiente.

Planejamento e desenvolvimento do experimento da camada IoT

Aqui são implementados os dispositivos da Internet das Coisas para avaliação da arquitetura implementada. Os *motes* foram desenvolvidos por meio do projeto de pesquisa MOREA (Monitoramento em Tempo Real de Consumo de Eletricidade e Água) do Instituto Federal de Sergipe campus Lagarto.

Desenvolvimento do sistema de monitoramento para IoT e implantação na camada *fog*

O SysMorea, sistema desenvolvido por meio do projeto de pesquisa MOREA, é responsável por receber dos dispositivos IoT as variáveis do ambiente (e.g., consumo atual de eletricidade de um prédio, em Wh). O sistema foi adaptado e consequentemente implementado à arquitetura de micro-PaaS em *fog computing* utilizando tecnologias de microsserviços.

Implementação da camada *Cloud*

Esta estapa do trabalho compreende a implementação do envio de dados da micro-PaaS fog para o ambiente de PaaS em nuvem pública, que ficará responsável por tarefas de processamento mais intensivas e exibição de dados para o público externo à rede local da micro-PaaS fog.

Coleta e análise de dados de desempenho e custo da arquitetura completa

Essa atividade contempla a análise, avaliação e interpretação dos dados que foram coletados na arquitetura implementada de modo a identificar os benefícios e limitações do ambiente de *fog computing*. Essa análise é feita em termos de otimização da qualidade de serviço para aplicações IoT relacionados à latência, alta disponibilidade, balanceamento de

carga, economia no valor gasto por dados trafegados na rede, mas também, considerando o custo de implementação da arquitetura utilizando plataformas de *hardware* de baixo valor se comparado às arquiteturas de grandes servidores.

4.2 Instrumentação

Para a implementação do trabalho proposto, foi necessária a utilização dos materiais e recursos abordados nesta seção.

4.2.1 *Single Board Computer* - SBC

Um computador de placa única, do inglês *single board computer* (SBC) é um computador completo construído em uma única placa de circuito, com microprocessador (es), memória, entrada / saída (E/S) e outros recursos necessários de um computador funcional. Os SBCs foram idealizados como sistemas de demonstração ou desenvolvimento, para sistemas educacionais, ou para uso como controladores de computador embarcados. Muitos tipos de computadores domésticos ou computadores portáteis integram todas as suas funções em uma única placa de circuito impresso.

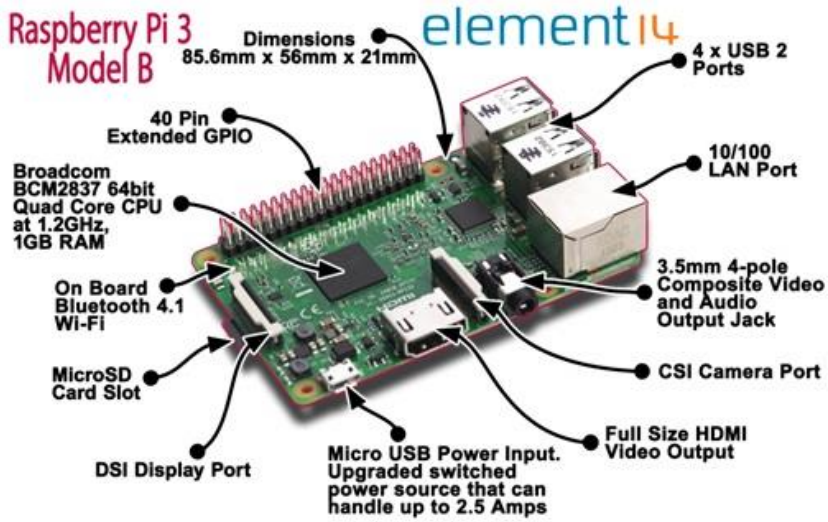
Ao contrário de um computador *desktop*, os SBCs geralmente não dependem de slots de expansão para funções periféricas ou expansão. Computadores de placa única foram construídos usando uma ampla gama de microprocessadores. Projetos mais simples, geralmente, usam RAM estática e processadores de 8 ou 16 bits de baixo custo. Mas também, existem tipos mais robustos que se assemelham a um computador servidor, só que em um formato mais compacto (DIAMOND SYSTEM CORPORATION, 2016).

Os SBCs foram possíveis devido ao aumento da densidade dos circuitos integrados. Uma configuração de placa única reduz o custo total de um sistema, reduzindo o número de placas de circuito necessárias e eliminando conectores e circuitos de driver de barramento. O *Raspberry Pi* é o tipo de SBC mais comum e mais citado nos artigos científicos, mas existem outros modelos, como por exemplo, *Banana Pi* (BANANA PI, 2019), *Orange Pi* (ORANGE PI, 2019), entre outros.

O *Raspberry Pi*, visualizado na Figura 6, é um pequeno computador potente baseado em microcontrolador ARM, ele funciona com o sistema operacional *Raspbian*, que consiste em uma adaptação da distribuição GNU/Linux Debian. Uma placa *Raspberry Pi* suporta cartão de

memória SD, teclado e mouse USB, monitor HDMI e fonte de alimentação. É possível fazer o *Raspberry Pi* funcionar como um computador normal de propósito geral (PRINCY, NIGEL, 2015).

Figura 6 - *Raspberry Pi* 3 modelo B



Fonte: https://www.element14.com

Apesar do tamanho reduzido, medindo aproximadamente 9 cm x 6 cm x 2 cm ele vem equipado com 1GB de memória RAM, processador ARMv8 quadcore de 1.2 Ghz, armazenamento de 16GB (cartão micro SD) podendo ser expandido, além disso, quatro portas USB, uma porta Ethernet e uma saída HDMI que podem ser utilizadas como expansão (ANTONI, VIVIAN, PREUSS, 2015).

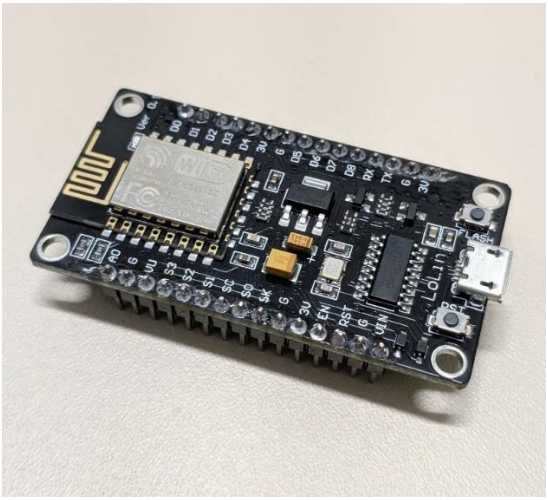
4.2.2 Placa NodeMCU

O NodeMCU é um kit de desenvolvimento aberto projetos de IoT, que inclui tanto um *firmware* baseado na linguagem de programação Lua, quanto uma placa que contém um microcontrolador ESP8266 ESP-12, com conector micro USB para ligação direta da placa com um PC, permitindo sua programação nos moldes da placa Arduino.

O ESP8266 trabalha com alimentação de 3,3V, mas o NodeMCU pode ser alimentado com tensões de até 9V. A placa NodeMCU possui um regulador de tensão que reduz a tensão da fonte para o nível requerido pelo ESP8266. Dessa forma, a placa pode ser alimentada por porta USB, bateria ou conjunto pilhas.

A Figura 7 mostra que a placa NodeMCU já possui duas fileiras de pinos em distância padronizada, permitindo seu encaixe em placas de prototipação tipo protoboard ou mesmo em placas de circuito impresso prontas.

Figura 7 - Placa NodeMCU



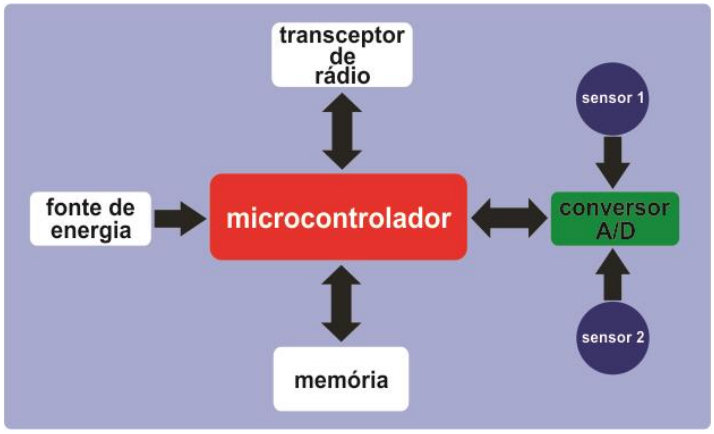
Fonte: Próprio autor, 2019

4.2.3 *Motes*

Os nós sensores de uma WSN, também são conhecidos como *motes* (abreviação dos termos em inglês "*mobile*" e "*remote*"), (DAGALE *et al.*, 2015; LEVIS *et al.*, 2005; WANG, ZHU E CHENG, 2006). O termo surgiu por volta do ano 2002 entre os pesquisadores da Universidade de Berkeley, nos Estados Unidos, após a apresentação da Plataforma *Mica* (Hill e Culler, 2002).

Para ser considerado um *mote*, o dispositivo deve possuir alguns componentes como microcontrolador, memória, transceptor de rádio, fonte de energia (geralmente bateria), conversor analógico digital e alguns sensores. Sua arquitetura pode ser observada na Figura 8:

Figura 8 - Arquitetura *mote*



Fonte: Próprio autor, 2019

Para compor a camada IoT da arquitetura proposta foram implementados dois tipos diferentes de *motes*, um para coletar o consumo de eletricidade e outro para coletar o consumo de água no Instituto Federal de Sergipe Campus Lagarto de forma inteligente (MENEZES, VIEIRA, MATOS JUNIOR, 2019) . Ambos foram prototipados com base no circuito integrado ESP8266 que faz parte do NodeMCU.

Uma das maiores vantagens encontradas nesse *chip* é o fato de ele suportar conexões IEEE 802.11 b/g/n, que o capacita a trabalhar diretamente com a pilha TCP/IP. O suporte a conexões IEEE 802.11 b/g/n permite implementar toda a arquitetura da IoT e seu gerenciamento dispensando o uso de outros elementos de rede. O dispositivo pode ser conectado diretamente à rede local e à Internet sem uso de nó de *gateway* para tradução de protocolos.

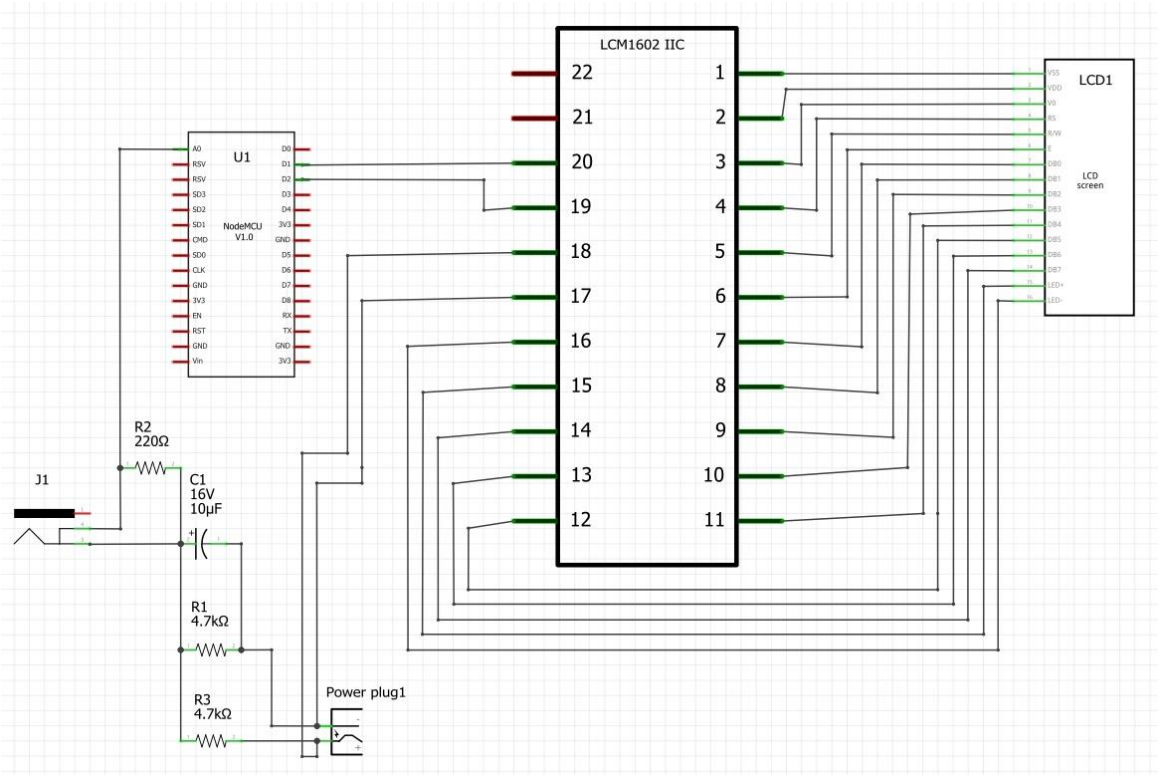
4.2.3.1 Mote medidor do consumo de energia (E-Mote)

O dispositivo medidor de energia (aqui chamado de E-Mote) foi implementado com base no sensor de corrente não invasivo 100A SCT-013 e uma placa NodeMCU para coleta, processamento e envio dos dados coletados. O circuito é alimentado por porta USB, que também é usada para gravação do *firmware* responsável pelo funcionamento pleno do dispositivo.

A Figura 9 ilustra o esquemático do circuito medidor de energia. A fonte de energia de 5V deve ser conectada em *power plug1*. A tensão é aplicada paralelamente ao módulo adaptador I2C (chip LCM1602), responsável pelo interfaceamento com o visor de cristal líquido (LCD –

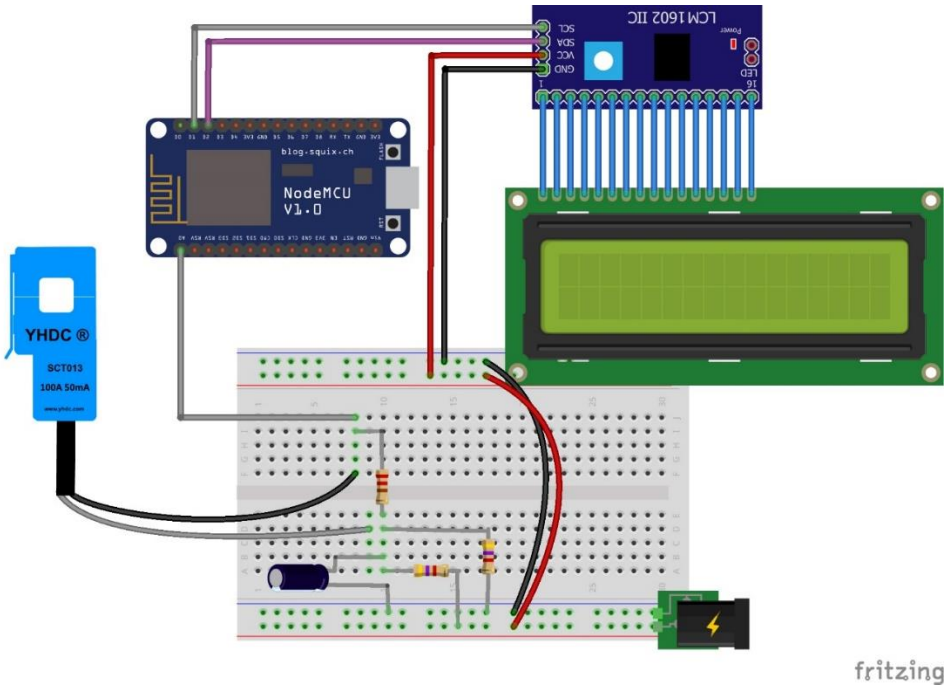
Lyquid Crystal Display) e ao circuito que interliga o NodeMCU na porta A0 e o sensor de corrente conectado ao *jumper* J1. Os dados coletados pelo sensor são enviados ao pino de entrada do conversor analógico digital AD no NodeMCU. Este por sua vez transmite as informações por meio de conexão sem-fio para o SysMorea na *fog*, bem como por meio das conexões GPIO (*General Purpose Input/Output*) nos pinos D1 e D2 para o LCM1602 nos pinos 19 e 20 respectivamente para visualização no LCD. A prototipação do *mote* é ilustrada na Figura 10, já o código fonte do *firmware* que foi gravado no NodeMCU pode ser consultado no APÊNDICE B.3.

Figura 9 - Esquemático do EMote - Medidor de Eletricidade Inteligente



Fonte: Próprio autor, 2019

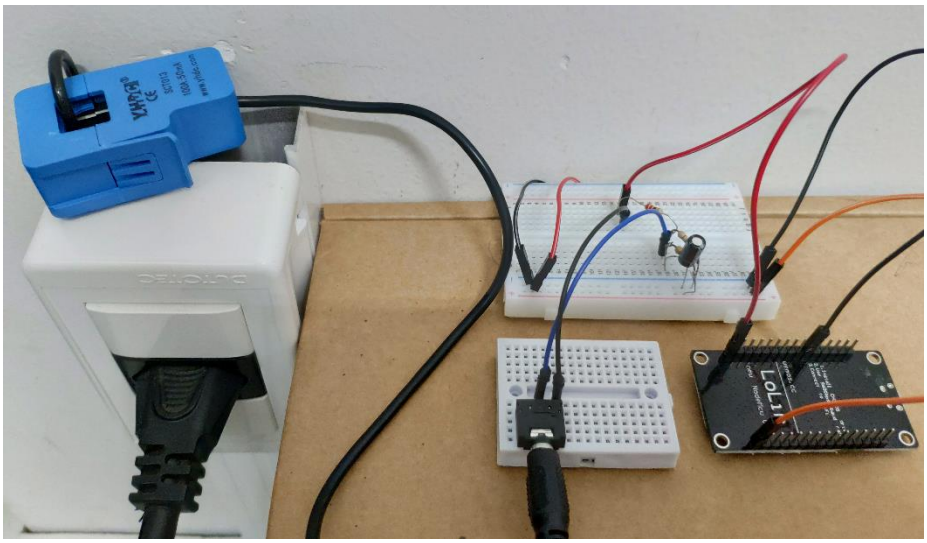
Figura 10 - Prototipação Mote IoT medidor de eletricidade



Fonte: Próprio autor, 2019

A Figura 11 ilustra o *mote* fazendo a leitura das variáveis de energia do ambiente. O dispositivo faz medições da intensidade de corrente elétrica (em *ampéres*) a cada 10 segundos, realizando também os cálculos de potência (em *watts*) e consumo instantâneos (em *watts*-hora). Os dados são enviados para *fog* em intervalos de 1 minuto com a somatória do consumo no intervalo mensurado.

Figura 11 - *Mote* medidor de eletricidade em produção



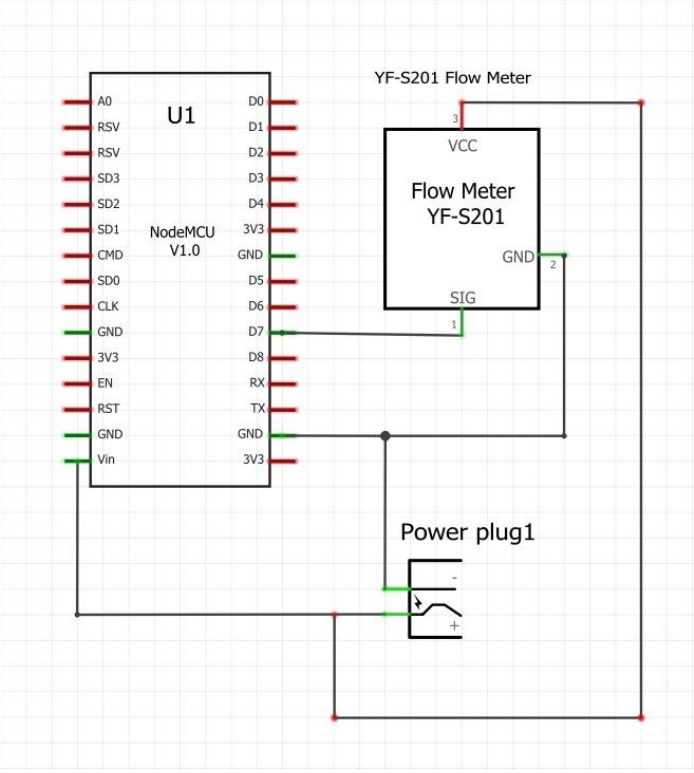
Fonte: Próprio autor, 2019

4.2.4 Mote medidor do consumo de água

O dispositivo para medir a vazão de água foi implementado com base no sensor de fluxo de líquido de ½” (meia polegada) (YF-S201) e uma placa NodeMCU. O YF-S201 utiliza o efeito *Hall* como princípio de funcionamento, cuja função é detectar a rotação de um rotor de nylon com um imã acoplado, gerando pulsos PWM (*Pulse Width Modulation*), de 2,25mm aproximadamente, proporcionais à velocidade com que o líquido passa pelo rotor, assim é possível mensurar a vazão. Já a placa NodeMCU é utilizada para coleta, processamento dos dados e envio das informações ao sistema principal SysMorea, o código do *firmware* pode ser consultado no APÊNDICE B.4.

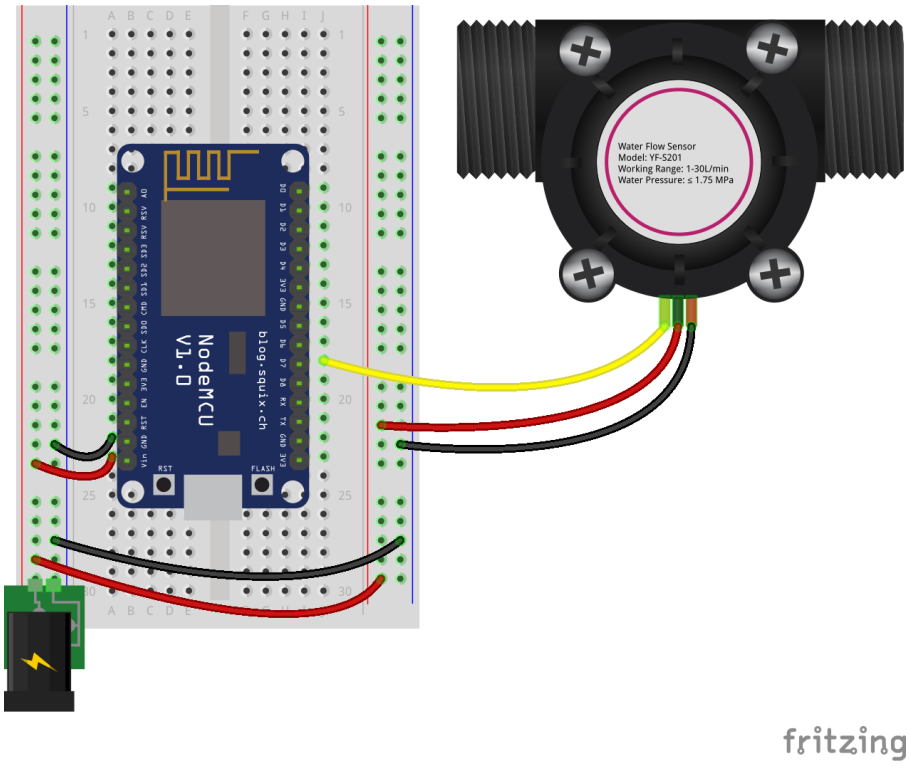
A Figura 12 ilustra o esquemático do circuito medidor de água. A prototipação do *mote* é ilustrada na Figura 13. A fonte de energia de 5V deve ser conectada em *power plug1*. A tensão é aplicada ao pino Vin do NodeMCU e ao VCC (fio vermelho) do sensor YF-S201. Já os valores referentes às variáveis de vazão são enviados pelo sensor de fluxo de líquido por meio da conexão (fio amarelo) ao pino D7 do NodeMCU. Importante que o GND tanto do sensor quanto do NodeMCU esteja conectado para o funcionamento pleno do dispositivo.

Figura 12 - Esquemático do WMote - Medidor de Água Inteligente



Fonte: Próprio autor, 2019

Figura 13 - Prototipação WMote - Medidor de Água Inteligente



Fonte: Próprio autor, 2019

5

Arquitetura de micro-PaaS em *fog computing*

Este capítulo apresenta de forma detalhada aspectos de concepção da abordagem proposta, desenvolvimento e implementação do ambiente experimental, os quais constituem a contribuição central desta dissertação de mestrado. Neste sentido, são discutidas as premissas de concepção da arquitetura de micro-PaaS em *fog computing* detalhando cada elemento e suas respectivas funcionalidades.

5.1 Premissas de Concepção

Uma das premissas de concepção da arquitetura de micro-PaaS em *fog computing* consiste na necessidade de obter um ambiente experimental que atenda às demandas inerentes às infraestruturas computacionais providas pela Internet das Coisas. Neste sentido a solução apresentada disponibiliza um ambiente isolado e seguro utilizando *containers* para a execução das aplicações. Este tipo de solução abstrai dos dispositivos detalhes do sistema operacional, bibliotecas, serviços de inicialização, gestão de memória, sistema de arquivos, entre outros, provendo uma maneira muito mais simples e prática de subir e escalar as aplicações.

A perspectiva é prover uma solução de alto nível, neste ambiente, e oferecer recursos como, alta disponibilidade, balanceamento de carga, orquestração, monitoramento e suporte ao processamento distribuído dos dados obtidos no meio, através de dispositivos inteligentes, os quais gerenciam a coleta e atuação, caracterizando uma abordagem de *fog computing*.

Como principais aspectos a serem providos neste cenário destacam-se: (i) suporte e orquestração das aplicações IoT utilizando tecnologias de *containers*, (ii) módulo de

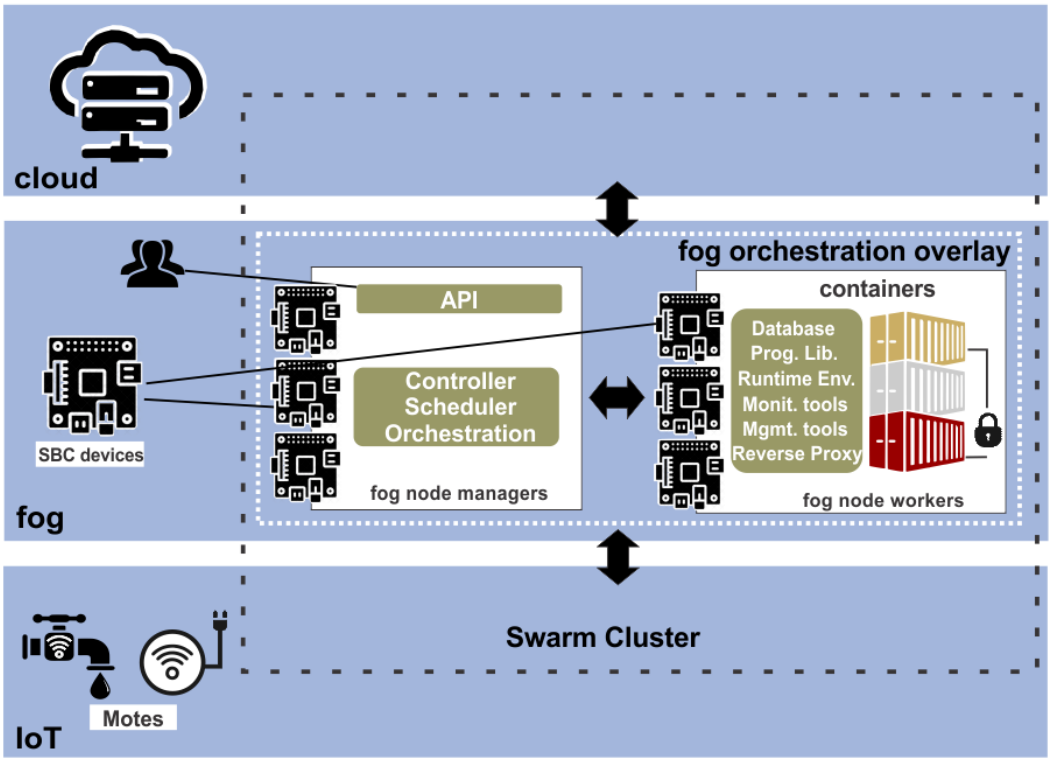
monitoramento em tempo real na *fog computing*, (iii) alta disponibilidade do *cluster*, (iv) distribuição vertical (i.e., entre diferentes níveis da arquitetura) da carga de processamento dos *motes*, (v) distribuição horizontal (i.e., entre *containers* da camada *fog overlay*) da carga de processamento gerada pelos motes, (vi) escalabilidade, (vii) balanceamento de carga.

Na concepção da arquitetura proposta, toda a funcionalidade do ambiente independe da *cloud* para funcionar. Esta somente será utilizada em períodos predeterminados visando a otimização da comunicação do ambiente como um todo, sobretudo nos aspectos relacionados a latência e no custo financeiro dos recursos da PaaS da nuvem como, memória, CPU, rede, etc; ou em situações extremas, onde os recursos na *fog* não sejam suficientes. O armazenamento em nuvem pública dos dados coletados pelos motes, acaba sendo também uma opção de *backup*, já que as informações podem ser consultadas na *fog* através de acesso local ou via VPN por meio de um link de internet convencional resguardados os devidos cuidados com os controles de acesso.

5.2 Modelo Arquitetural

A arquitetura desenvolvida é constituída por 4 (quatro) camadas; 3 (três) delas distribuídas em domínios verticais, e uma quarta camada de sobreposição, rede *overlay*, responsável pela comunicação das aplicações para Internet das Coisas em domínios horizontais. A camada de rede *overlay* contém os *hosts* do *cluster*, formado por dispositivos *Raspberry Pi*, conforme ilustrado na Figura 14. No cenário de estudo de caso implementado, como prova-de-conceito para a arquitetura proposta, a micro-PaaS fornece mecanismos para suportar e orquestrar, na *fog*, aplicações de monitoramento do consumo de água e energia de setores específicos do Instituto Federal de Sergipe por meio de dispositivos inteligentes. No entanto, a arquitetura em estudo pode ser utilizada em diversos cenários para atender às variadas demandas da Internet das Coisas.

Figura 14 - Arquitetura de micro-PaaS em *fog computing*



Fonte: Próprio autor, 2019

Camada *Cloud*

A camada *cloud* localiza-se na camada superior e interage com a camada imediatamente inferior. Esta camada recebe da camada *fog* informações de leitura das variáveis do ambiente por intervalos de tempos predefinidos. A aplicação executada na micro-PaaS Fog deverá comunicar-se com a parte da aplicação hospedada na *cloud*. Na *cloud* também podem ser executadas tarefas que exijam recursos computacionais além dos que estão disponíveis na *fog*, sejam eles relacionados ao hardware (e.g., poder de processamento e memória principal), *software* (e.g., bibliotecas e APIs de programação, recursos para inteligência artificial) ou dados (e.g., grandes repositórios).

É importante ressaltar que a *cloud* aqui pode ser de natureza pública, privada, híbrida, ou comunitária, sem impactos consideráveis para o modelo arquitetural proposto.

Camada *Fog*

A camada *fog* compreende os dispositivos estacionários da rede localizados na camada intermediária atuando como uma ponte entre os dispositivos inteligentes e os serviços de

computação em nuvem. Os *nodes* dessa camada foram classificados em dois tipos, *Fog Node Manager* (FNM) e *Fog Node Worker* (FNW), conforme observado na Figura 14.

Camada IoT

A camada IoT é composta por *motes* formando o alicerce da Internet das Coisas dando origem ao fluxo de dados distribuídos. Enquanto os dispositivos IoT estão sob o nível inferior, a camada *fog* e *cloud* compreendem o segundo e terceiro nível respectivamente, além dessas a camada *overlay* se sobrepõe ao segundo nível criando uma camada de sobreposição. Os dispositivos que compõem esta camada fazem a leitura de informações do ambiente onde estão inseridos (e.g., temperatura, umidade, consumo de água, consumo de energia), e também podem controlar objetos pertencentes a este ambiente (e.g., aparelhos de ar condicionado, pontos de iluminação). Os motes comunicam-se com a camada imediatamente superior por meio de rede sem fio, passando por um dispositivo intermediário, como um access point ou roteador.

Camada *fog orchestration overlay*

A camada *fog orchestration overlay* é uma camada de sobreposição à camada *fog*. Esta camada é uma abstração implementada no *Docker* para o gerenciamento da comunicação de dados entre os containers com segurança e que permitem resolver problemas de escalabilidade, balanceamento de carga e sobreposição de endereçamento no *cluster*. Através desta camada é possível anexar vários serviços de *containers* na mesma rede, como, por exemplo, o serviço de descoberta interno do *Swarm* que entrega um endereço IP exclusivo e uma entrada DNS (também por um serviço interno) para cada *container* na mesma rede.

Dessa forma cria-se a rede *overlay* com o objetivo de tratar o tráfego de controle de dados relacionados aos serviços do *cluster*, tornando transparente o roteamento dos pacotes de cada *container* independente do *node* que requisitou o serviço. Isto fará com que o cliente acesse os recursos através de um único endereço (URL), permitindo implementar alta disponibilidade, escalabilidade, balanceamento de carga, resiliência e sobretudo a orquestração do ambiente. Tudo isso, através de túneis para comunicação segura utilizando o protocolo IPSEC (*IP Security Protocol*).

5.3 Desenvolvimento do ambiente experimental

Esta seção detalha os passos realizados para o desenvolvimento e implantação de uma micro-PaaS em *fog computing* de acordo com o modelo arquitetural proposto na seção 5.2. Este ambiente experimental foi planejado como prova de conceito de modo a prover infraestrutura com baixo TCO e boa performance necessária às aplicações IoT utilizando *containers*, em um ambiente orquestrado, com balanceamento de carga, escalável, com alta disponibilidade e baixa latência utilizando *Raspberry Pi*.

5.3.1 Implementação da *Fog Computing*

Para implementação da camada *fog* foram utilizados seis *Raspberry Pi 3* modelo B e um *switch* HP modelo A5120 *layer* 2, ilustrado na Figura 15. Primeiramente instalou-se em cada um dos dispositivos *Raspberry PI* o HypriotOS[5], um sistema operacional, de código aberto, muito leve baseado no Debian, otimizado para arquitetura ARM (*Advanced Risk Machine*) e pronto para executar o *Docker* e suas ferramentas, como, *Docker-Compose*, *Docker-Swarm*, entre outras. O passo seguinte foi realizar algumas configurações para padronização, como, por exemplo, definição do *hostname*, endereços estáticos, etc.

Figura 15 - *Cluster Fog*

Fonte: Próprio autor, 2019

[5] https://blog.hypriot.com/

A Tabela 1 traz detalhes das definições realizadas, bem como algumas especificações físicas e relacionadas aos sistemas. O *cluster* que compõe a *fog*, formado pelos 6 (seis) *Raspberry Pi* foram interligados à VLAN IFSLAB, do Instituto Federal de Sergipe Campus Lagarto, por meio de conexões *GigabitEthernet* cabeadas. A comunicação entre os dispositivos da camada IoT e o *cluster* da camada *fog*, ocorreu por meio de rede sem fio padrão IEEE 802.11n.

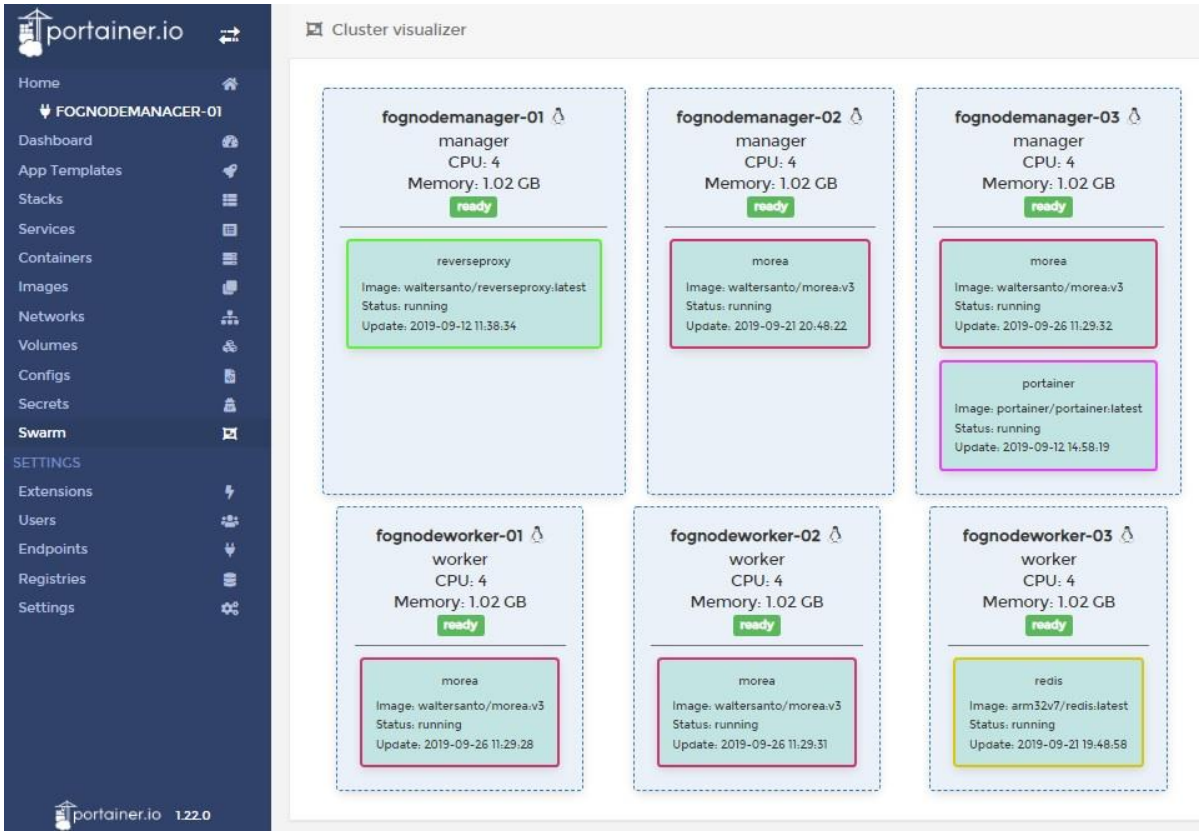
Tabela 1- Especificações do ambiente

Hardware	**S.O.**	**Versão do Docker engine**	**Hostname**	**Rede IFS LAB (172.29.0.0 / 21) IP / Máscara**
CPU 4 CORE 1200MHz ARM Cortex-A53 1GB Memory RAM	HypriotOS v1.10.0	18.06.03-ce	fognodemanager-01	172.29.0.30 / 21
			fognodemanager-02	172.29.0.31 / 21
			fognodemanager-03	172.29.0.32 / 21
			fognodeworker-01	172.29.0.33 / 21
			fognodeworker-02	172.29.0.34 / 21
			fognodeworker-03	172.29.0.35 / 21

Fonte: Próprio autor, 2019

Após as configurações iniciais e definição dos parâmetros de rede dos SBCs foi necessária a configuração do *cluster* com o *Docker-Swarm*, ou simplesmente *Swarm*. O *Swarm* é uma camada de *software* responsável pelo gerenciamento e a orquestração dos *containers* sobre *clusters* de *Docker hosts*, atuando de forma conjunta provendo alta disponibilidade e escalabilidade. A Figura 16 traz uma visão geral do *cluster* por meio da interface de usuário web *Portainer* (PORTAINER, 2019), onde pode-se observar informações importantes como a quantidade de *nodes*, total de CPU, total de memória, bem como, a versão da API do *Docker Engine*. Os comandos utilizados para a configuração do *cluster Swarm* estão detalhados no apêndice A.1.

Figura 16- Tela de visão geral do *cluster* no Portainer



Fonte: Próprio autor, 2019

Para que seja plenamente possível ao *Docker Swarm* realizar a orquestração dos *containers* na *fog* que irão rodar as instâncias das aplicações como serviços, existem quatro elementos essenciais: ***node***, ***task***, ***service*** e ***load balancing***.

Os *nodes* na arquitetura proposta são instâncias físicas implementadas com seis *Raspberry Pi*, no entanto estas instâncias também podem ser representadas de forma virtualizada. Estas instâncias podem assumir basicamente três papéis: *managers*, *workers* ou ambos. A Figura 17 ilustra todos os FNM e FNW do *cluster* e respectivas informações de *status* e disponibilidade.

Figura 17 - *Fog Node Manager* e *Fog Node Worker*

```
root@fognodemanager-01:/home/walter# docker node ls
ID                            HOSTNAME            STATUS  AVAILABILITY  MANAGER STATUS
jzs6szyuek9q8uwihlgr9l8on *   fognodemanager-01   Ready   Active        Leader
kywr2xj2y2shpl3l4nye8kjst     fognodemanager-02   Ready   Active        Reachable
fh79uhmneazv2bl0ll7t7oupr     fognodemanager-03   Ready   Active        Reachable
smbua76nfawke996vhhsiwtj8     fognodeworker-01    Ready   Active
m8ou5eg39ij2atgfhffk5far4     fognodeworker-02    Ready   Active
scligyapq85s6nlq3tnqnkof9     fognodeworker-03    Ready   Active
root@fognodemanager-01:/home/walter#
```

Fonte: Próprio autor, 2019

FNM são *nodes managers* responsáveis pela gestão do *cluster* como um todo, pela orquestração das aplicações, pelo monitoramento de todos os *nodes* do *cluster*, cuidam das rotinas internas quando um novo serviço é publicado no *cluster*, recebem e aplicam atualizações impostas pelo administrador do *cluster*. Além disso, são responsáveis por controlar todo o ciclo de vida dos *containers*, conciliação entre os estados, atribuição de *workloads* para elementos abaixo do *cluster*, dentre outras operações. Para que a arquitetura proposta pudesse prover alta disponibilidade foram necessários a implementação de pelo menos 3 *managers*, esse quantitativo se dá pela implementação utilizada no algoritmo de consenso *raft* intrínseco no *cluster Swarm*, o qual será explicado adiante. Os FNM podem assumir a função tanto de *managers* como *workers*, conforme detalhado na Figura 18. Dos 3 *managers* implementados o *Swarm* elege um destes como *leader* (*node* através do qual o administrador do sistema se comunica com os demais *nodes* do *cluster*), os demais serão réplicas do *leader* e ficam com o *status reachable*. Desse modo, quando os *managers* estão com o *status reachable* desempenham a função de *workers*, e em caso de falha do *leader* o *Swarm* elegerá um novo *leader* dentre os dois *managers* com o *status reachable*. *Nodes* que assumem ambas as funções priorizam sempre a função *manager* em detrimento à função *worker*.

FNW como o próprio nome sugere, são *nodes workers*, apenas recebem as cargas de trabalho enviadas pelo *manager leader* e as processam, não possuindo qualquer função administrativa.

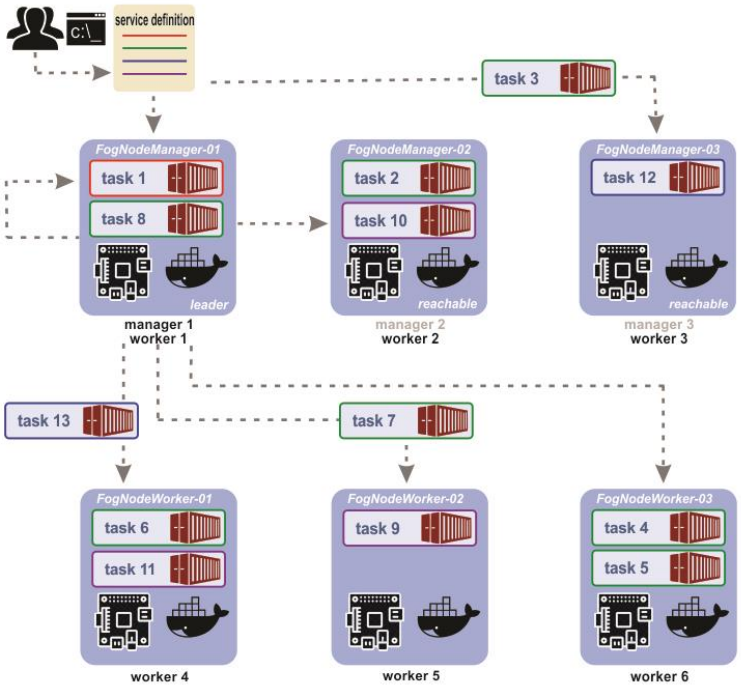
A *task*, de maneira pragmática, é um ambiente de execução de um único *container* que "responde" para a gestão dos *nodes managers* do *Swarm*. Diferente de instâncias *Docker standalone*, as *tasks* não possuem qualquer tipo de autonomia, as configurações são pré-definidas pelos *managers* e vão sempre executar apenas o que lhe foi designado. Uma *task* será sempre criada e atribuída a um *node worker* pelo *manager*. Outro aspecto importante sobre *tasks* é que esta nunca poderá iniciar sua execução em um *node* e terminar em outro. O *Swarm* utiliza *tasks* como unidades atômicas de distribuição de *workloads*, a Figura 18 apresenta uma visão geral das *tasks* distribuídas ao longo dos *nodes*.

Enquanto as *tasks* são os elementos através dos quais os *containers* são distribuídos ao longo dos *nodes*, o *service* é o elemento responsável por todas as definições relacionadas ao ambiente, sendo a única forma de se interagir com o *Swarm.* Quando um *service* é criado pode-se definir o número de réplicas para os *nodes*, a imagem que deve ser considerada para *deploy* dos *containers*, comandos que serão executados dentro dos *containers*, camadas de redes,

dentre tantas outras configurações. Portanto, a distribuição das *tasks* ocorre depois, quando o *service* já definiu quando e como tudo deverá funcionar.

Para o ambiente *fog computing* apresentado neste estudo foram instanciados 4 serviços, são eles: *SysMorea*, *Redis*, *Reverse Proxy* e o *Portainer*, observados na Figura 18 em, *service definition*, pelos contornos nas cores verde, azul, roxo e vermelho respectivamente.

Figura 18 - Visão geral do processo de orquestração da *Fog*



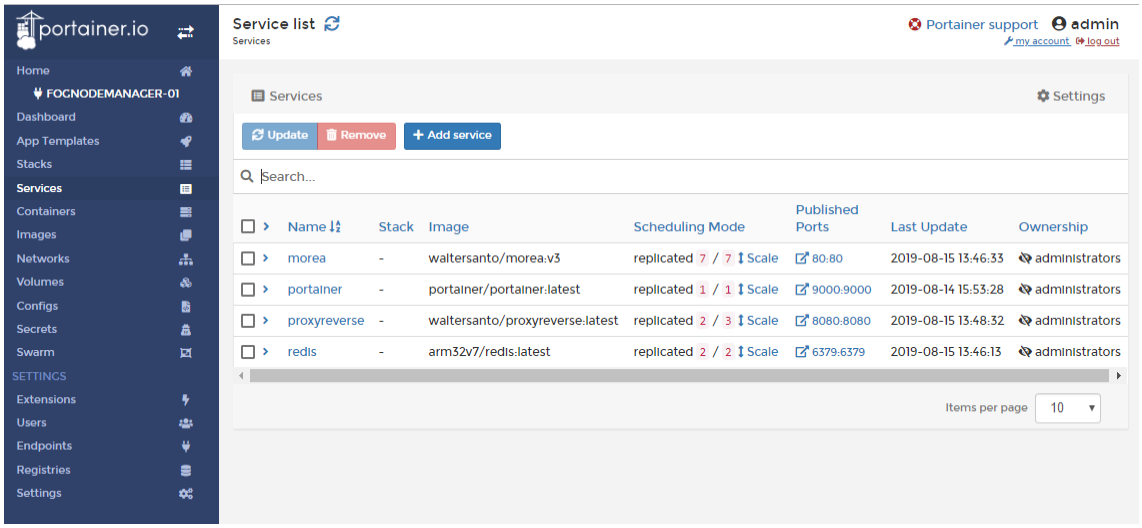
Fonte: Próprio autor, 2019

Já o *load balancing* é vital à orquestração no *Swarm*, é o elemento primário de distribuição de carga entre as *tasks* executadas em cada *node*. Cada *node* do *cluster Swarm* utiliza o método *ingress load balancing* para expor publicamente os serviços desejados ao acesso externo. Dessa forma, é atribuído um valor de porta disponível ao parâmetro "*PublishedPort*", e cujo valor é definido no momento em que o serviço é criado. Caso não seja atribuído um valor manualmente, o *Swarm* automaticamente atribui uma porta disponível entre as portas 30000 a 32767. Outro importante aspecto relacionado ao *load balancing* está atrelado ao componente de DNS interno do *Swarm* que atribui automaticamente a cada *service* em execução uma entrada na tabela interna de DNS. Assim, o *load balancing* utiliza o DNS para distribuir as cargas entre os diferentes *services* em execução.

5.3.2 Aplicações e Serviços

A arquitetura de micro-PaaS em *fog computing* proposta foi capaz de prover orquestração, balanceamento de carga, escalabilidade e o instanciamento de serviço das seguintes aplicações para um ambiente da Internet das Coisas: *SysMorea, Redis, Reverse Proxy* e o *Portainer*. Na Figura 19 são observadas 13 *tasks* as quais correspondem aos 13 serviços escalados e devidamente replicados na *fog*.

Figura 19 - Escalabilidade de serviços na *Fog*

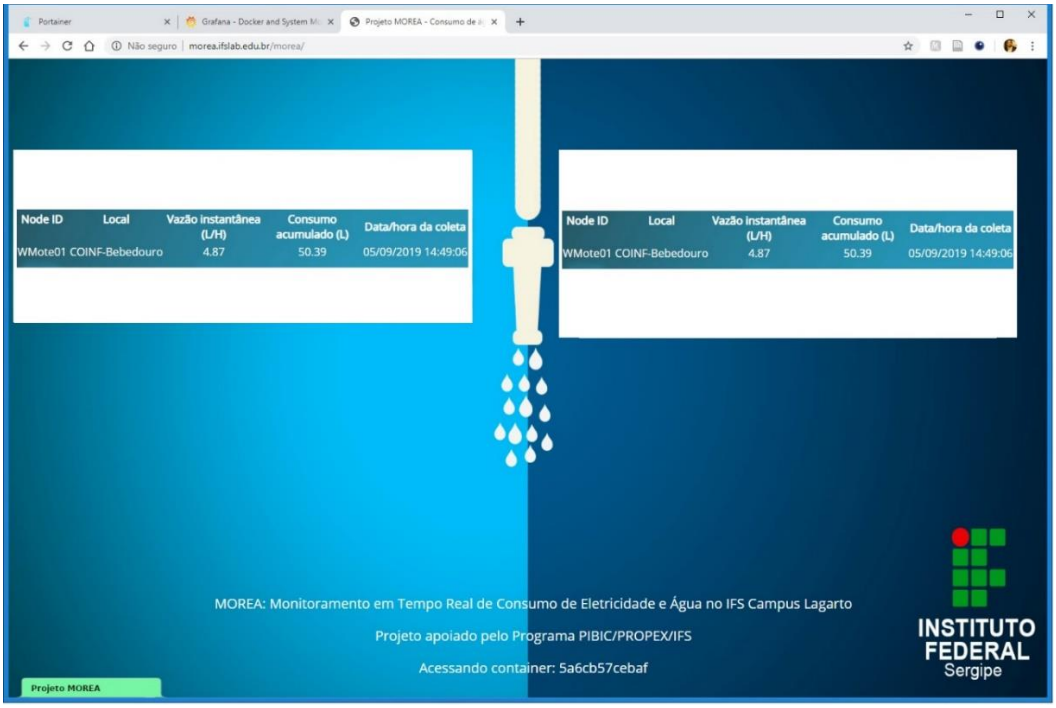


Fonte: Próprio autor, 2019

SysMorea e Redis

O Sistema de Monitoramento em tempo real do consumo de Eletricidade e Água (SysMorea) (MENEZES, VIEIRA, MATOS JUNIOR, 2019), tem como objetivo fazer a leitura do consumo de eletricidade e água de pontos estratégicos do Instituto Federal de Sergipe Campus Lagarto de forma inteligente. As informações de consumo são obtidas por meio da atuação de dispositivos IoT que fazem a leitura em tempo real das variáveis do ambiente e as transmitem ao SysMorea, via *wireless*, instanciado como serviço na *fog orquestration overlay*. A Figura 20 ilustra a tela do sistema.

Figura 20 - SysMorea



Fonte: Próprio autor, 2019

O SysMorea foi desenvolvido na linguagem de programação PHP e com integração ao Redis, um banco de dados não relacional do tipo chave-valor (REDIS, 2019). O código do sistema pode ser consultado no apêndice B.1, bem como os comandos que instanciaram o serviço Redis no *cluster* no apêndice C.2.

Na Figura 21 é possível visualizar os identificadores das *tasks*, os *nodes* onde os *containers* executando a aplicação foram instanciados, e a quantidade de réplicas sendo 7 para o SysMorea e 2 para o Redis, o *status* do serviço, a porta utilizada pelo serviço, entre outras informações.

Figura 21 - Orquestração dos serviços SysMorea e Redis

Name	Stack	Image	Scheduling Mode	Published Ports	Last Update	Ownership
morea	-	waltersanto/morea:v3	replicated 7 / 7 Scale	80:80	2019-08-15 13:46:33	administrators

Status Filter	Task	Actions	Slot	Node	Last Update
running	zjts47gz3x60hjh4tl1g2p1lx		1	fognodemanager-02	2019-08-14 16:05:22
running	wb194rr7dg1onhjdu2f7so46z		2	fognodemanager-03	2019-08-15 13:46:37
running	14nkk62yk7jx7ygo1937takje		3	fognodeworker-03	2019-08-14 16:09:18
running	jv7xqctf3j9m9sq7h8i917ifh		4	fognodeworker-03	2019-08-15 13:46:37
running	9y4dt5gwzi1phbcdhzjfsh7w9		5	fognodeworker-01	2019-08-14 16:09:13
running	r1ifbzo53hjnx0ufhvk11jf1u		6	fognodeworker-02	2019-08-14 16:09:13
running	yv1xgjbrvzosw4fvaq1k9v13c		8	fognodemanager-01	2019-08-14 16:09:22

redis	-	arm32v7/redis:latest	replicated 2 / 2 Scale	6379:6379	2019-08-15 13:46:13	administrators

Status Filter	Task	Actions	Slot	Node	Last Update
running	acxje4qdrqaau816hjsxqap1s		1	fognodemanager-03	2019-08-12 19:06:06
running	d36icbees1bb8q09ad7zhaxdp		2	fognodeworker-01	2019-08-15 13:46:17

Fonte: Próprio autor, 2019

Para instanciar o serviço contendo a aplicação SysMorea no *cluster Swarm* fez-se necessário primeiramente a criação de um *container* contendo os sistemas a saber:

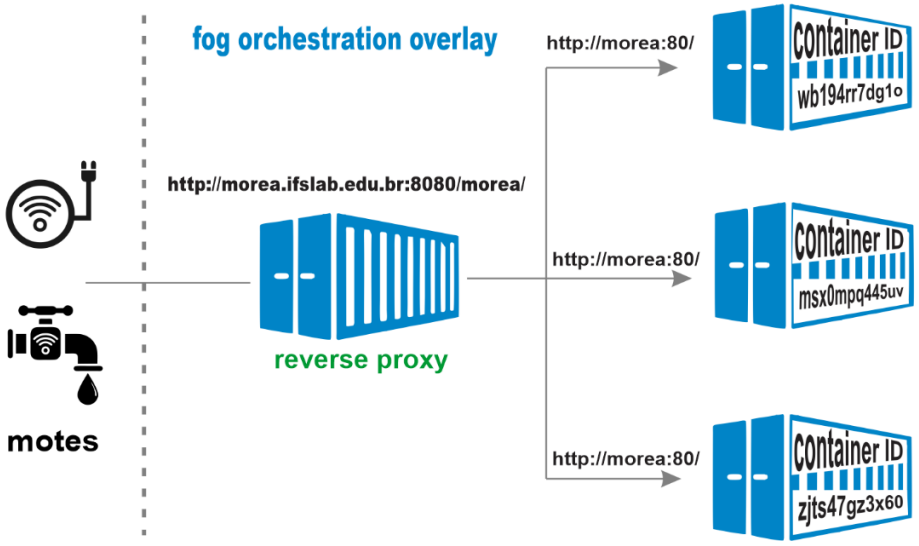
- Sistema Operacional Debian GNU/Linux 7 (wheezy);
- Servidor web Nginx 1.7.10
- PHP 5.4

Após a criação do *container* contendo os sistemas acima e as devidas configurações, foi gerada a imagem waltersanto/morea e a realização do *deployment* da mesma para o *docker hub*. Todos os passos contendo os comandos utilizados, bem como o link do docker hub contendo a aplicação podem ser verificados no apêndice C.1.

Reverse Proxy

Em uma arquitetura de microsserviços, é comum dividir uma API em várias aplicações de implantação independente. Embora tecnicamente os serviços que são executados em frente aos *containers* que contém as aplicações possam se expor diretamente à Internet por meio de portas, a melhor prática é atender ao tráfego de um *cluster* nas portas 80/443 e rotear internamente via DNS. A Figura 22 ilustra o *reverse proxy* implementado na camada *fog* da arquitetura proposta.

Figura 22 – Reverse Proxy



Fonte: Próprio autor, 2019

Para isso, foi utilizado o serviço nginx como *reverse proxy*. Dessa forma, quando o tráfego chega ao serviço nginx, ele é roteado para serviços na *fog orchestration overlay* pelo

nome DNS. No apêndice C.3 é possível verificar a configuração que direciona o tráfego com base na URL de entrada: “http://morea.ifslab.eud.br:8080/morea”.

A Figura 23 ilustra as 3 réplicas do *reverse proxy* que foram devidamente escaladas no ambiente de orquestração proposto.

Figura 23 - Orquestração do serviço *Reverse Proxy*

Fonte: Próprio autor, 2019

5.3.3 Alta disponibilidade

A alta disponibilidade no ambiente implementado parte de duas premissas: a tolerância a falhas dos FNMs, e a alta disponibilidade dos FNWs.

No tocante aos *nodes managers* o *cluster* implementado consiste de três FNM. O *node manager* contém as informações necessárias para gerenciar o *cluster*, mas se este ficar inativo, o *cluster* poderá deixar de funcionar caso não haja uma quantidade mínima de *node manager* necessários para prover a tolerância a falha do ambiente.

Quando o *Docker Engine* é executado no modo *Swarm*, os *nodes managers* implementam o algoritmo de consenso *Raft* para gerenciar o estado do *cluster* global. A razão em utilizar este tipo de algoritmo é certificar-se de que todos os *nodes managers* encarregados de gerenciar e agendar tarefas no *cluster* estejam armazenando o mesmo estado consistente.

Ter o mesmo estado consistente em todo o *cluster* significa que, em caso de falha, qualquer *node manager* pode selecionar as tarefas e restaurar os serviços para um estado estável. Por exemplo, se o *leader manager*, que é responsável pelo agendamento de tarefas no *cluster*, deixa de responder, qualquer outro *manager* pode pegar a tarefa de agendar e reequilibrar tarefas para corresponder ao estado desejado.

O *Raft* tolera até (N-1)/2 falhas, onde N = número total de *node managers*, e requer uma maioria ou quórum de (N/2)+1 membros para concordar com os valores propostos ao *cluster*. Isso significa que, na arquitetura implementada, se 2 dos 3 FNMs estiverem indisponíveis, o sistema não poderá processar solicitações para agendar tarefas adicionais. As tarefas existentes

continuam em execução, mas o planejador não pode reequilibrar tarefas para lidar com falhas se o conjunto de *managers* não estiver íntegro.

Para que a tolerância a falha ocorra é necessário que seja mantido um número ímpar de *managers* no *Swarm* maior ou igual a três, conforme demonstrado na Tabela 2**.** Ter um número ímpar de *managers* garante que, durante um desmembramento da rede em dois conjuntos, por exemplo, haja uma chance maior de que o quórum permaneça disponível para processar solicitações.

Tabela 2- Tolerância a falhas na *Fog Orchestration Overlay*

Tamanho do cluster	**Quórum**	**Tolerância a falhas**
1	1	0
2	2	0
3	2	1
4	2	1
5	3	2

Fonte: Próprio autor, 2019

Já no que diz respeito aos *nodes workers* o impacto por uma indisponibilidade pode ser bem menos relevante já que estes apenas processam cargas de trabalhos e não executam função de gerenciamento e administração. Estes impactos são detalhados no Capítulo 6 que trata da análise dos resultados obtidos.

5.3.4 Interfaces de monitoramento e gerenciamento do ambiente

A arquitetura de Micro-PaaS em *fog computing*, objetivo principal desse trabalho de pesquisa, também foi capaz de prover monitoramento e gerenciamento, eficiente, por meio de *containers*, das aplicações do ambiente, das métricas, dos próprios *containers* e serviços do *cluster*. O conjunto de ferramentas implementadas com essa finalidade foram: Portainer (PORTAINER, 2019), o *dashboard Grafana* (GRAFANA, 2019), e o *data source Prometheus* (PROMETHEUS, 2019) todos de código aberto.

Portainer

Para o gerenciamento e manutenção do ambiente *Docker*, incluindo os *nodes*, *end points*, o próprio *cluster Swarm* utilizou-se o *Portainer* (PORTAINER, 2019). No apêndice

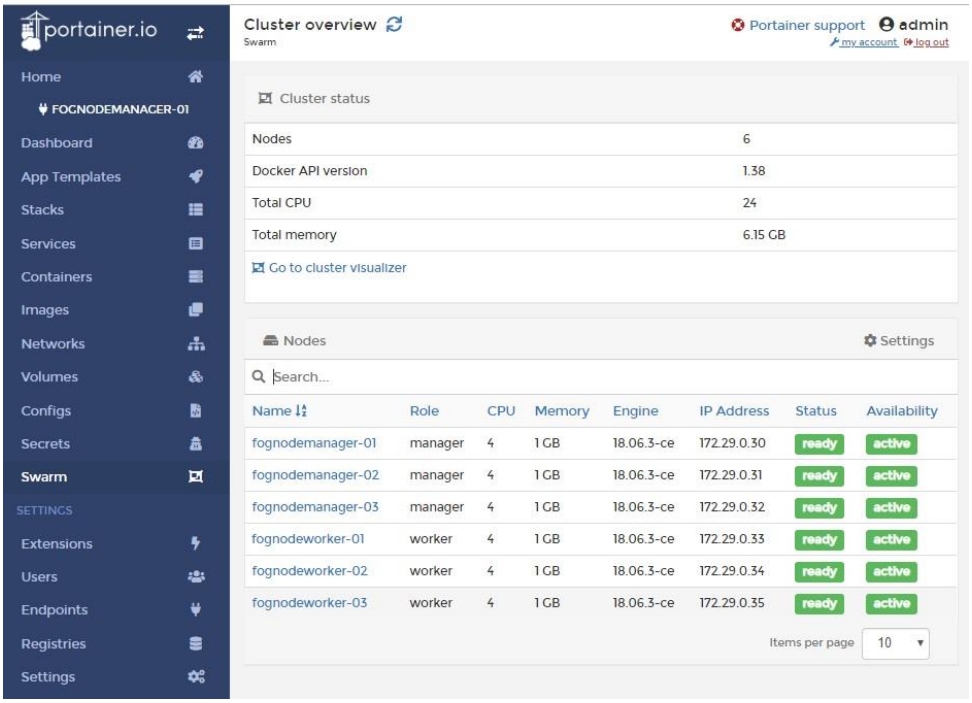
A.2 é possível verificar com detalhe o processo de instanciação do serviço *Portainer*, apenas em *nodes managers*, visto não fazer sentido que um ambiente de gerenciamento seja acessado de um *node worker*. Na Figura 24 é possível verificar a alocação do serviço ao *container* correspondente.

Figura 24 - Orquestração do serviço Portainer

Fonte: Próprio autor, 2019

O *Portainer*, conforme ilustra a Figura 25, é uma *interface web* de gerenciamento, bastante intuitiva, leve e *open source*, conforme afirmado anteriormente. Este serviço foi integrado ao *cluster* com o objetivo de fornecer uma visão geral e detalhada do ambiente *Docker*, e consequentemente uma melhor administração da *fog* provendo de forma eficiente o gerenciamento e a manutenção de *containers*, imagens, redes e volumes.

Figura 25 - Portainer



Fonte: Próprio autor, 2019

Grafana

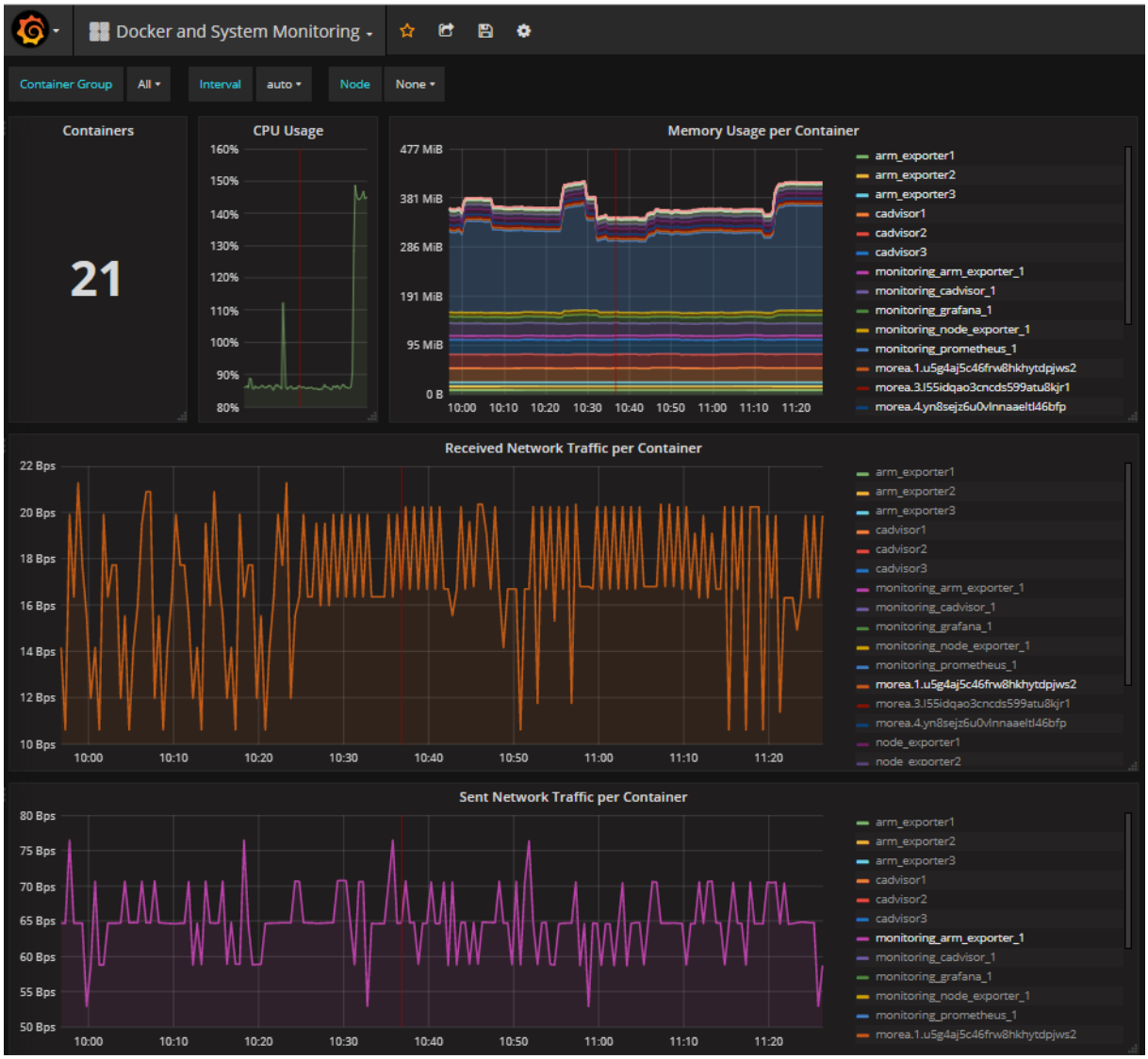
O *Grafana* é uma aplicação de *dashboard* o qual permite consultar, visualizar, alertar e interpretar métricas. A visualização dos dados pode ser integrada a diferentes mecanismos para prover monitoramento em tempo real de aplicações, ambientes de infraestrutura e outros serviços. Entre as principais características, destacam-se:

- Monitoramento *on-line*;
- Visualização enquanto as aplicações são executadas com tempo em ordem de 1 segundo;
- Interatividade com interface web;
- Filtros elaborados;

Os dados visualizados no *Grafana*, apresendados na Figura 26 foram utilizados no estudo do comportamento do *cluster* como um todo, sobretudo, no que diz respeito à utilização de recursos de rede e computacionais. Além disso, foi através desse *dashboard* que as métricas foram exportadas para subsidiar as análises realizadas no capítulo posterior. Dentre as métricas avaliadas estão:

- Total de carga da CPU, memória e uso de armazenamento dos *hosts* no *cluster;*
- Gráfico de uso da CPU, memória e do tráfego de entrada e saída de rede dos nodes da *fog;*
- Gráfico de uso da CPU, memória e do tráfego de entrada e saída de rede dos *containers*;

Figura 26 - Grafana

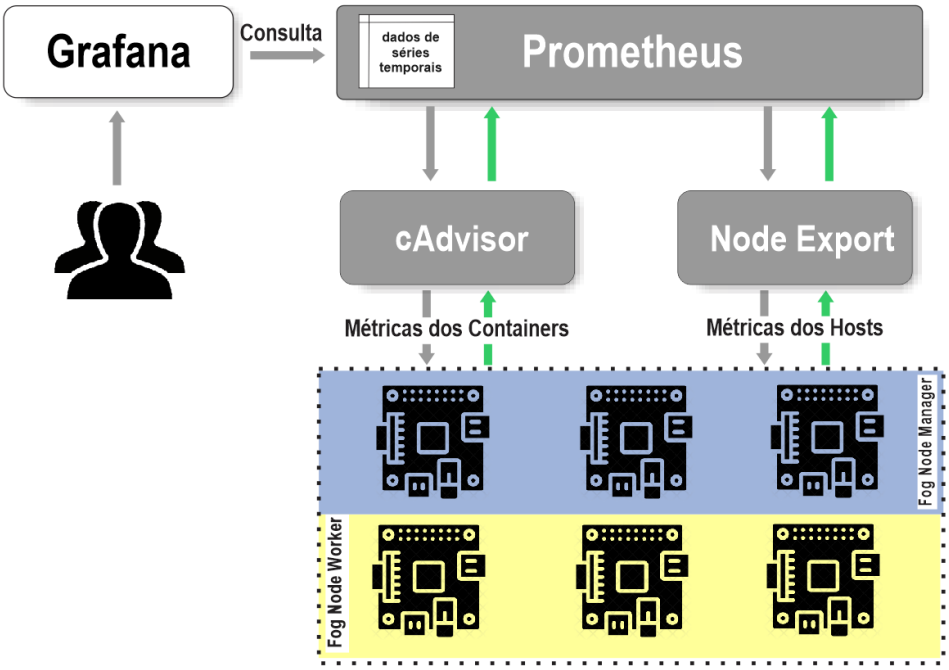


Fonte: Próprio autor, 2019

Prometheus

O Prometheus (PROMETHEUS, 2019) é uma ferramenta *open source* de monitoramento, adaptada ao atual modelo de TI e focada em serviços. Essa ferramenta foi desenvolvida pela SoundCloud em 2012 e logo seu projeto foi abraçado por outras empresas, inclusive pelo Docker, um dos principais contribuidores do projeto (Silva, D. S., 2018). O Prometheus lê métricas de trabalhos, diretamente ou por meio de um *gateway* intermediário armazenando todas as amostras coletadas localmente e executando regras sobre esses dados para agregar e registrar novas séries temporais ou gerar alertas. A Figura 27 ilustra a integração do *Prometheus* com *Grafana*.

Figura 27 – Integração monitoramento com Prometheus e Grafana



Fonte: Próprio autor, 2019

As informações dos *containers* são obtidas utilizando-se o cAdvisor, que consiste em um *daemon* em execução que coleta, agrega, processa e exporta informações sobre a execução de *containers* (CADVISOR, 2019). Por outro lado, a ferramenta *Node Exporter*, fica responsável pelo envio das métricas dos *hosts* para o *Prometheus*. Por fim, o *Grafana* consulta a base de dados do *Prometheus* e exibe as métricas do ambiente em formato de gráficos compondo um painel de visualização.

6

Resultados e Discussão

Este capítulo apresenta os detalhes dos experimentos realizados na arquitetura implementada, de modo a identificar na *fog computing*, as vantagens em termos de otimização da qualidade de serviço para aplicações IoT relacionados à latência, alta disponibilidade, escalabilidade, balanceamento de carga e economia no valor gasto com a aquisição e operação da infraestrutura necessária.

6.1 Alta disponibilidade e balancemanto de carga

A arquitetura de micro-PaaS em *fog computing* implementada não só provê a possibilidade de fazer com que os *nodes* atuem de forma conjunta e coordenada, ou seja, orquestrada, como oferece recursos de alta disponibilidade, escalabilidade e balanceamento de carga.

6.1.1 Metodologia da avaliação

Para a obtenção das métricas de disponibilidade do ambiente bem como a ratificação do balanceamento de carga, durante experimentos controlados de injeção de falhas, foi implementado no *reverse proxy* um *script shell*, o CheckMorea cujo código é exibido no Quadro 5.

Quadro 5 - *Script shell CheckMorea*

Check Morea
#!/bin/bash # Script to check every 0.2s the availability of the Morea # Author: Rubens Matos e Walter E. Santo # Date: 2019-21-08

```
outfile="log-morea-availability.txt"

while [ True ]
do
    touch $outfile
    timestamp=`date +"%d-%m-%Y %H:%M:%S"`
    timeout 1 curl morea/morea/index.php > output-web-service.txt 2>&1
    status=$?
    ContainerID=`grep Acessando output-web-service.txt | cut -d: -f2`
    if [[ "$status" -eq 0 ]]
    then
      echo $timestamp" Up"$ContainerID>>$outfile
    else
      echo $timestamp" Down"$ContainerID>>$outfile
    fi
    sleep 0.2
done
```

Fonte: Próprio autor, 2019

O *script* CheckMorea verifica a disponibilidade do SysMorea, realizando uma requisição para a página web do sistema a cada 200 milissegundos. O *script* registra o status *Up* caso a requisição tenha um retorno com sucesso, e *Down* caso haja falha. Ele também registra o identificador do *container* que respondeu à requisição.

Após a definição do *script*, uma outra etapa imprescindível que envolva qualquer estudo empírico em que o objetivo é fazer inferências sobre uma população a partir de uma amostra é determinar o tamanho da mesma, a fim de oferecer uma representação estatísticas adequada e suficiente. Determinar o tamanho adequado da amostra é essencial, já que amostras muito grandes podem desperdiçar tempo e recursos, enquanto amostras muito pequenas podem levar a resultados imprecisos. Nesse sentido, para determinar o tamanho da amostra utilizou-se a equação (i) (JAIN, 1990), para um intervalo de confiança de 95%, aplicada a uma amostra preliminar.

$$n = \left[\frac{z_{\frac{\alpha}{2}}\,\sigma}{E}\right]^2 \rightarrow n = 15{,}1 \text{ (i)}$$

onde:

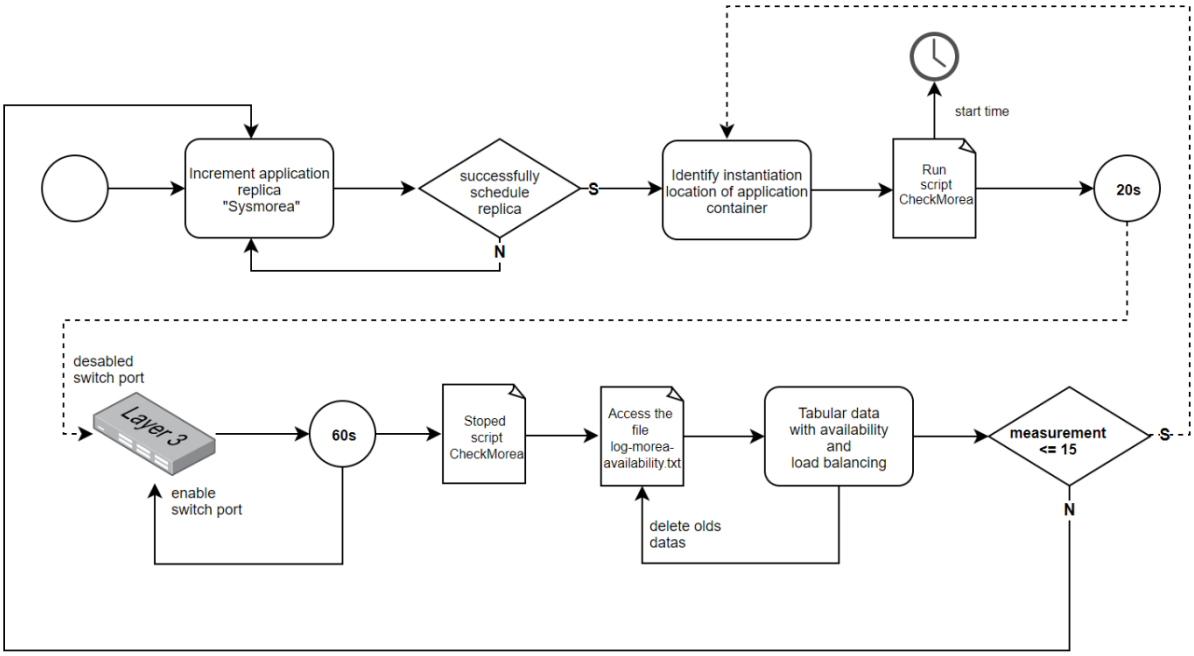
- n = número de amostras
- $z_{\frac{\alpha}{2}}$ = constante 1,96 para Intervalo de Confiança (IC) de 95%, considerando a distribuição normal

- σ = desvio padrão (2,20 – maior desvio padrão encontrado na amostra), conforme Tabela 3.
- E = erro máximo aceitável de 5% multiplicado pela maior média, da amostragem preliminar conforme Tabela 3.

Dessa forma, depois de definido o tamanho ideal da amostra, realizou-se 15 (quinze) testes de injeção de falha, cada um destes com duração mínima de 1 (um) minuto, para cada um dos cenários exibidos no Quadro 6. Com a aplicação em *status* de execução, o *script* CheckMorea era iniciado. Decorridos 20 segundos da execução do *script*, a fim de provocar a indisponibilidade da aplicação, desabilitava-se a porta do *switch* na qual o *fog node* responsável pela aplicação naquele momento estava conectado. A simulação da indisponibilidade via de regra seguia sempre o serviço instanciado mais antigo, ou seja, o *fog node* que continha o *container* com este serviço seria desconectado do *cluster*. Ao final dos 60 segundos era encerrada a execução do *script* e os dados do arquivo "log-morea-availability.txt" eram então analisados e interpretados.

Após a realização dos 15 (quinze) testes escalonava-se uma nova réplica, com isso, era obtido um novo cenário e os 15 (quinze) testes eram novamente repetidos. A partir do 2º cenário as réplicas foram escalonadas de forma incremental, de 2 (duas) até 5 (cinco) réplicas. O último cenário foi implementado com 10 (dez) réplicas, com o objetivo de verificar o comportamento do sistema em uma escala maior que as anteriores. A Figura 28 ilustra de forma sequencial o fluxograma de execução dos experimentos de disponibilidade do sistema SysMorea hospedado na micro-PaaS fog.

Figura 28 – Fluxograma do experimento de disponibilidade do SysMorea na micro-PaaS fog



Fonte: Próprio autor, 2019

6.1.2 Carga de trabalho

Os cenários estabelecidos que recebiam a carga de trabalho e que foram simulados, afim de validar a alta disponibilidade e o balanceamento de carga do ambiente são exibidos no Quadro 6. A identificação dos FNMs e FNWs contidos em cada um dos cenários seguem a seguinte padronização:

- FNM**x**[**y**] – onde **x** é a identificação do host *Fog Node Manager* e **y** o número de réplicas instanciadas da aplicação;
- FNW**x**[**y**] – onde **x** é a identificação do host *Fog Node Worker* e **y** o número de réplicas instanciadas da aplicação;
- **FNMx[y] –** comunicação interrompida do *Fog Node Manager* com a *fog*;
- **FNWx[y] –** comunicação interrompida do *Fog Node Worker* com a *fog*;
- Quando houver omissão do termo **[y]** significa que a quantidade de réplica do serviço escalonado é igual a 1 (um).

Quadro 6 – Cenários utilizados pela carga de trabalho

	Cenário I [1] réplica	Cenário II [2] réplicas	Cenário III [3] réplicas	Cenário IV [4] réplicas	Cenário IV [5] réplicas	Cenário VI [10] réplicas
1	**FNM2**	**FNW2** FNW3	FNM1 FNM2 **FNW2**	FNM2 FNW1 FNW2 **FNW3**	FNM2 FNM3 **FNW1**-FNW2 FNW3	**FNM1** FNM2 FNM3[2] FNW1[2] FNM2[2] FNW3[2]
2	**FNW3**	FNM1 **FNW3**	FNM1 FNM2 **FNM3**	FNM1 FNM2 **FNW1** FNW2	FNM1 FNM2 FNM3 FNW1 **FNW2**	FNM1[2] FNM2[2] FNM3[2] FNW2[2] **FNW 3[2]**
3	**FNW1**	FNM1 **FNW2**	FNM1 FNM2 **FNW2**	FNM1 FNM2 **FNW2** FNW3	FNM1 FNM2 FNM3 FNW1 **FNW3**	FNM1[2] FNM2[2] FNM3[2] **FNW1[2]** FNW3[2]
4	**FNM2**	FNM1 **FNM2**	FNM1 **FNM2** FNM3	FNM1 FNM2 FNW1 **FNW3**	FNM1 FNM2 FNM3 FNW1 **FNW2**	FNM1[2] FNM2[2] FNM3[2] FNW1[2] **FNW3[2]**
5	**FNW1**	FNM1 **FNW2**	FNM1 FNM2 **FNM3**	FNM1 FNM2 **FNW1** FNW2	FNM1 FNM2 FNM3 **FNW1** FNW3	FNM1[2] FNM2[2] FNM3[2] **FNW1[2]** FNW2[2]
6	**FNM2**	FNM1 **FNM2**	FNM1 FNM2 **FNW2**	FNM1 FNM2 **FNW2** FNW3	FNM1 FNM2 FNM3 FNW2 **FNW3**	FNM1[2] FNM2[2] FNM3[2] **FNW2[2]** FNW3[2]
7	**FNM3**	FNM1 **FNW2**	FNM1 FNM2 **FNW2**	FNM1 FNM2 FNW1 **FNW3**	FNM1 FNM2 FNM3 FNW1 **FNW2**	FNM1[2] FNM2[2] FNM3[2] FNW1[2] **FNW3[2]**
8	**FNM2**	FNM2 **FNW2**	FNM1 **FNM2** FNM3	FNM1 FNM2 FNW3 **FNW1**	FNM1 FNM2 FNM3 FNW1 **FNW3**	FNM1[2] FNM2[2] FNM3[2] **FNW1[2]** FNW2[2]
9	**FNW1**	FNM1 **FNM2**	**FNW1** FNW2 FNW3	FNM1 FNM2 FNM3 **FNW3**	FNM1 FNM3 FNW1 FNW2 **FNW3**	FNM1[2] FNM2[2] FNM3[2] **FNW2[2]** FNW3[2]
10	**FNM3**	FNM1 **FNW2**	FNM2 **FNW2** FNW3	FNM1 FNM2 **FNW2**-FNW3	FNM1 FNM2 FNM3 **FNW1** FNW2	FNM1[2] FNM2[2] FNM3[2] FNW1[2] **FNW3[2]**
11	**FNM2**	FNM1 **FNM2**	FNM2 FNW2 **FNW3**	FNM1 FNM2 FNM3 **FNW3**	FNM1 FNM2 FNM3 **FNW2** FNW3	FNM1[2] FNM2[2] FNM3[2] **FNW1[2]** FNW2[2]
12	**FNM3**	FNM1 **FNW2**	FNM2 FNW2 **FNW1**	FNM1 FNM2 FNM3 **FNW1**	FNM1 FNM2 FNM3 FNW1 **FNW3**	FNM1[2] FNM2[2] FNM3[2] **FNW2[2]** FNW3[2]
13	**FNW3**	FNM1 **FNM2**	FNM2 **FNW2** FNW3	FNM1 FNM2 FNM3 **FNW3**	FNM1 FNM2 FNM3 **FNW1** FNW2	FNM1[2] FNM2[2] FNM3[2] FNW1[2] **FNW3[2]**
14	**FNM2**	FNM2 **FNW2**	FNM2 FNW1 **FNW3**	FNM1 FNM2 FNM3 **FNW1**	FNM1 FNM2 FNM3 **FNW2** FNW3	FNM1[2] FNM2[2] FNM3[2] **FNW1[2]** FNW2[2]
15	**FNM3**	FNM1 **FNW2**	FNM2 **FNW2** FNW3	FNM1 FNM2 FNM3 **FNW3**	FNM1 FNM2 FNM3 FNW1 **FNW3**	FNM1[2] FNM2[2] FNM3[2] **FNW2[2]** FNW3[2]

Fonte: Próprio autor, 2019

6.1.3 Resultados obtidos

Um dos principais desafios deste trabalho de pesquisa foi implementar um ambiente em *fog computing* com recursos de hardware com poder computacional discretos e que fossem suficientemente eficientes na orquestração de aplicações sem comprometer o desempenho do ambiente e a qualidade de serviço para as aplicações IoT. Com esse objetivo, conduzimos o experimento de modo a identificar a quantidade mínima de réplicas que deveriam ser

escalonadas e orquestradas no ambiente capaz de atender aos requisitos de indisponibilidade e balanceamento de carga sem degradar a performance da *fog*. A Tabela 3 exibe a média, desvio padrão e coeficiente de variação do tempo de indisponibilidade (também conhecido como *downtime*), considerando a injeção artificial de uma falha e um tempo de monitoramento de 60 segundos de acordo com os números de réplicas específicos.

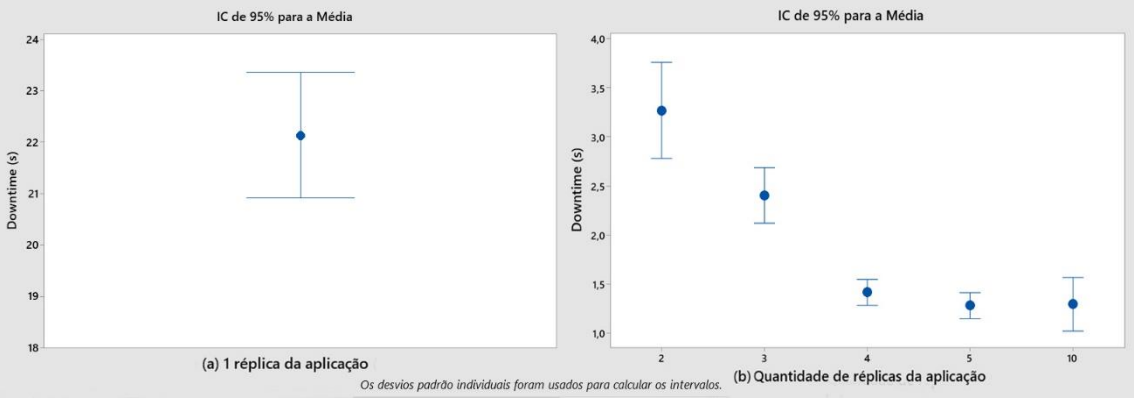
Tabela 3 – Métricas da indisponibilidade

Total de Réplicas	Média (s)	Desvio padrão	Coeficiente de variação
1	22,13	2,20	0,10
2	3,27	0,88	0,27
3	2,40	0,51	0,21
4	1,33	0,49	0,37
5	1,41	0,23	0,16
10	1,29	0,49	0,38

Fonte: Próprio autor, 2019

Para demonstrar o quanto os resultados apresentados são confiáveis calculou-se o IC para a média obtida na amostra de cada cenário, obtendo-se um IC de 95% para a média, baseado na distribuição normal, conforme ilustrado na Figura 29.

Figura 29 – Gráfico intervalo de confiança



Fonte: Próprio autor, 2019

Para uma melhor visualização a Figura 29 foi segmentada em duas partes: (a) 1 (uma) réplica e (b) os demais quantitativos de réplicas. Dessa forma, permitiu-se uma melhor resolução entre as quantidades de réplicas de cada cenário, já que a diferença entre o IC do ambiente escalonado com (1) uma réplica, Figura 29 (a); com os demais cenários, Figura 29 (b), é relativamente maior se comparado com a diferença dos valores do IC entre os demais cenários.

É possível ainda identificar que entre os cenários escalonados com 1, 2 e 3 réplicas não há intersecção dos IC entre estes. No entanto, como, afirmado anteriormente, os resultados dos cenários contendo 4, 5, e 10 réplicas, respectivamente, não possuem diferenças estatísticas relevantes entre si.

Os valores obtidos das medições realizadas na amostra revelaram que escalonando o ambiente com 4 (quatro) réplicas é suficiente para orquestração de todo o ambiente de forma eficiente, levando-se em consideração redundância e balanceamento de carga visando à alta disponibilidade. Escalonar o ambiente além disso vai exigir custos, como, por exemplo, memória e processamento desnecessários.

A Tabela 4 exibe a análise dos dados referentes à medição 01 do "Cenário III", que corresponde a um ambiente orquestrado com 3 réplicas instanciadas nos nodes FNM1, FNM2 e NFW2 respectivamente. O FNW2 foi selecionado para simular o *status "down"* na *fog*. Na coluna **Container ID** é possível visualizar o identificador de cada *container* responsável pela resposta da aplicação no momento da consulta, demonstrando o efetivo funcionamento do balanceamento de carga através da alternância às requisições de forma lógica e sequencial. Ainda na mesma coluna após os 20 segundos é possível notar que quando o *status* da aplicação encontra-se *down* o identificador do *container* "c7587c8cf100" deixa de responder às requisições. O sistema fica indisponível por 3 segundos e depois volta a responder com as duas réplicas restantes por meio dos identificadores "8777417c5bc4" e "5312a34f9c1e" até a recuperação total do ambiente que orquestra de forma automática uma nova réplica através do identificador "4d59b5e87d36" em substituição ao que falhou.

Tabela 4 – Indisponibilidade e balanceamento de carga

Medição 01			
node: UP	FNM1 / FNM2 / FNW2		
node:Down	FNW2		
Data	**Hora**	**Status**	**Container ID**
26-08-2019	17:53:46	Up	c7587c8cf100
26-08-2019	17:53:47	Up	8777417c5bc4
26-08-2019	17:53:48	Up	5312a34f9c1e
26-08-2019	17:53:49	Up	c7587c8cf100
26-08-2019	17:53:50	Up	8777417c5bc4
26-08-2019	17:53:51	Up	5312a34f9c1e
26-08-2019	17:53:52	Up	c7587c8cf100
26-08-2019	17:53:53	Up	8777417c5bc4
26-08-2019	17:53:54	Up	5312a34f9c1e
26-08-2019	17:53:55	Up	c7587c8cf100
26-08-2019	17:53:56	Up	8777417c5bc4
26-08-2019	17:53:57	Up	5312a34f9c1e

26-08-2019	17:53:58	Up	c7587c8cf100
26-08-2019	17:53:59	Up	8777417c5bc4
26-08-2019	17:54:00	Up	5312a34f9c1e
26-08-2019	17:54:01	Up	c7587c8cf100
26-08-2019	17:54:02	Up	8777417c5bc4
26-08-2019	17:54:03	Up	5312a34f9c1e
26-08-2019	17:54:04	Up	c7587c8cf100
26-08-2019	17:54:05	Up	8777417c5bc4
26-08-2019	17:54:06	Up	5312a34f9c1e
26-08-2019	17:54:07	Down	
26-08-2019	17:54:08	Up	8777417c5bc4
26-08-2019	17:54:09	Up	5312a34f9c1e
26-08-2019	17:54:10	Down	
26-08-2019	17:54:11	Up	8777417c5bc4
26-08-2019	17:54:12	Up	5312a34f9c1e
26-08-2019	17:54:13	Down	
26-08-2019	17:54:14	Up	8777417c5bc4
26-08-2019	17:54:15	Up	5312a34f9c1e
26-08-2019	17:54:16	Up	8777417c5bc4
26-08-2019	17:54:17	Up	5312a34f9c1e
26-08-2019	17:54:18	Up	8777417c5bc4
26-08-2019	17:54:19	Up	5312a34f9c1e
26-08-2019	17:54:20	Up	8777417c5bc4
26-08-2019	17:54:21	Up	5312a34f9c1e
26-08-2019	17:54:22	Up	8777417c5bc4
26-08-2019	17:54:23	Up	5312a34f9c1e
26-08-2019	17:54:24	Up	8777417c5bc4
26-08-2019	17:54:25	Up	5312a34f9c1e
26-08-2019	17:54:26	Up	8777417c5bc4
26-08-2019	17:54:27	Up	5312a34f9c1e
26-08-2019	17:54:28	Up	8777417c5bc4
26-08-2019	17:54:29	Up	5312a34f9c1e
26-08-2019	17:54:30	Up	**4d59b5e87d36**
26-08-2019	17:54:31	Up	8777417c5bc4
26-08-2019	17:54:32	Up	5312a34f9c1e
26-08-2019	17:54:33	Up	**4d59b5e87d36**
26-08-2019	17:54:34	Up	8777417c5bc4
26-08-2019	17:54:35	Up	5312a34f9c1e
26-08-2019	17:54:36	Up	**4d59b5e87d36**
26-08-2019	17:54:37	Up	8777417c5bc4
26-08-2019	17:54:38	Up	5312a34f9c1e
26-08-2019	17:54:39	Up	**4d59b5e87d36**
26-08-2019	17:54:40	Up	8777417c5bc4
26-08-2019	17:54:41	Up	5312a34f9c1e
26-08-2019	17:54:42	Up	**4d59b5e87d36**
26-08-2019	17:54:43	Up	8777417c5bc4
26-08-2019	17:54:44	Up	5312a34f9c1e
26-08-2019	17:54:45	Up	**4d59b5e87d36**
26-08-2019	17:54:46	Up	8777417c5bc4
Tempo de Indisponibilidade			**3 segundos**

Fonte: Próprio autor, 2019

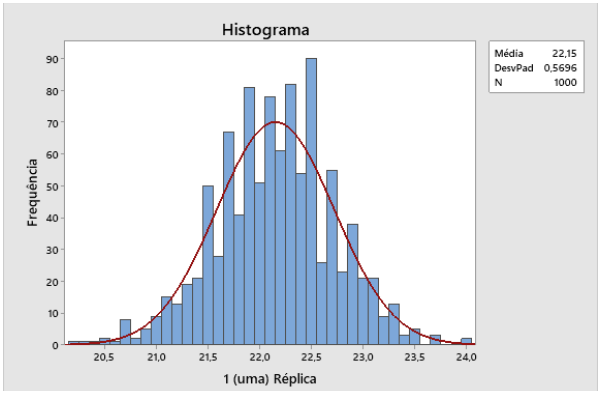
As planilhas contendo os resultados de todos os 90 testes realizados (15 repetições para cada um dos 6 cenários) podem ser consultados no endereço: https://is.gd/iW5wki.

Para que fosse verificado o comportamento das métricas de indisponibilidade de todos os cenários em uma escala relativamente superior para comparação com os cenários reais, utilizou-

se como estratégia de reamostragem a técnica estatística *Bootstrap* (Statdisk, 2019). Dessa forma, conseguiu-se estimar uma reamostragem com 1000 valores, baseados na média e desvio padrão dos dados coletados nos cenários reais. Com os dados obtidos das reamostragens gerou-se gráficos de histogramas com curva de distribuição normal relacionando a frequência com o total de réplicas, conforme ilustrado nas Figuras de 30 a 35 para cada um dos cenários.

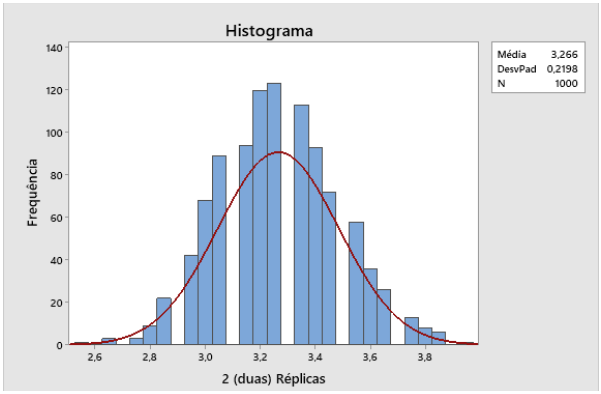
Os valores gerados pelo *Bootstrap* nas posições 25 e 975 correspondem respectivamente a 2,5% (limite inferior) e 97,5% (limite superior) do intervalo de confiança coincidindo com os valores encontrados na amostragem real.

Figura 31 – Gráfico Histograma 1 Réplicas



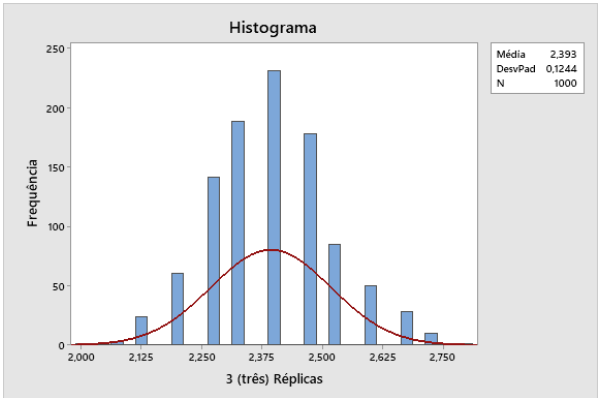
Fonte: Próprio autor, 2019

Figura 30 – Gráfico Histograma 2 Réplicas



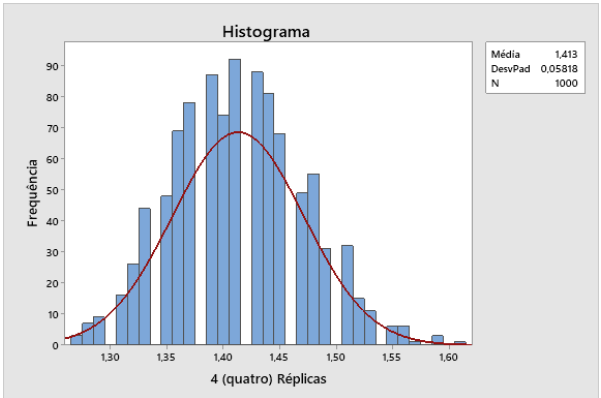
Fonte: Próprio autor, 2019

Figura 33 – Gráfico Histograma 3 Réplicas



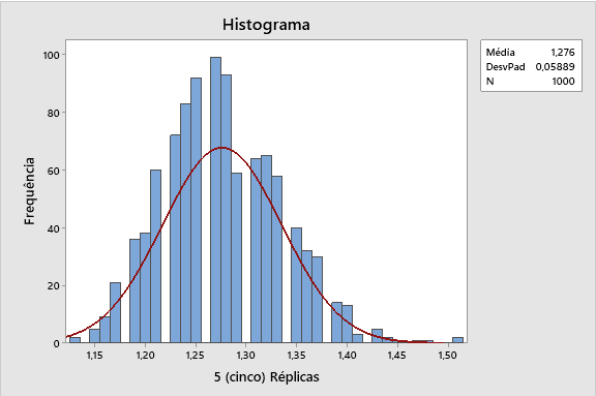
Fonte: Próprio autor, 2019

Figura 32 – Gráfico Histograma 4 Réplicas



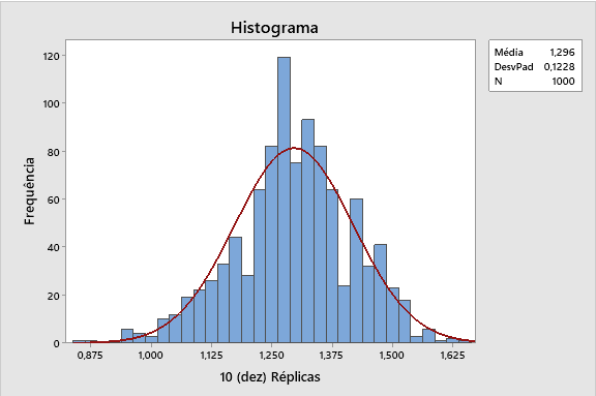
Fonte: Próprio autor, 2019

Figura 35 – Gráfico Histograma 5 Réplicas



Fonte: Próprio autor, 2019

Figura 34 – Gráfico Histograma 10 Réplicas



Fonte: Próprio autor, 2019

6.2 Custo total de propriedade micro-PaaS Fog e PaaS Heroku

O custo total de propriedade, do inglês, *total cost of ownership* (TCO) é o cálculo geral de todos os custos que envolvem a aquisição e implementação de um serviço ou produto. Tem como objetivo calcular os custos totais de um produto ou ferramenta durante sua vida útil e determinar se um investimento é válido ou não. Nesse sentido, realizou-se um estudo comparativo do TCO da micro-PaaS Fog e da PaaS Heroku (HEROKU, 2019). O TCO encontrado da micro-PaaS Fog representou apenas 23% da versão profissional de entrada da Heroku, visualizada no Quadro 7.

Heroku é um provedor de PaaS em nuvem baseado em um sistema de *containers* (*dynos*) gerenciado, com serviços de dados integrados e um poderoso ecossistema, para implantar e executar aplicativos modernos. A experiência do desenvolvedor Heroku é uma abordagem centrada no aplicativo para entrega de software, integrada às ferramentas e fluxos de trabalho mais populares da atualidade.

O Quadro 7 consiste na comparação do TCO da micro-PaaS implementada com a TCO de 3 (três) tipos de serviços disponibilizados pela plataforma Heroku. Destes serviços, um possui configuração básica (Free) e dois configurações profissionais (Standard 1x e Standard 2x).

A versão Free, conforme informações da própria plataforma Heroku, é utilizada como ambiente de teste em nuvem como uma *sandbox* limitada. Por ter um custo zero para o usuário a plataforma é bastante econômica nos recursos oferecidos, como, por exemplo, apenas uma aplicação web e um único *worker* com apenas 512MB de memória RAM. Comporta apenas 2 tipos de processos, não escala de forma horizontal, também não traz relatórios das métricas da aplicação e nem pode ser combinada com múltiplos *containers* (dynos). Mostrando-se dessa forma ineficiente para o ambiente de monitoramento proposto.

Já a versão Standard 1x é a versão de entrada profissional, traz 512MB de memória RAM, destina-se a visibilidade, desempenho e disponibilidade aprimorados para alimentar aplicações de produção, suporta tipos de processos ilimitados e pode ser escalada. As métricas das aplicações podem ser consultadas a partir de 2 horas com resolução de 1 minuto até no máximo 7 dias com resolução de 2 horas, pode ser combinada com múltiplos *containers* de desempenho. No entando o seu custo de 25 dólares ao mês por cada *container* implementado pode inviabilizar projetos com orçamentos modestos.

A versão Standard 2x diferencia-se da anterior apenas no tocante a quantidade de memória RAM, aumentando-se de 512MB para 1GB e no preço que também dobrou. Nesta versão o usuário terá um custo mensal de 50 dólares por cada *container*.

Quadro 7 – TCO Heroku e micro-PaaS Fog

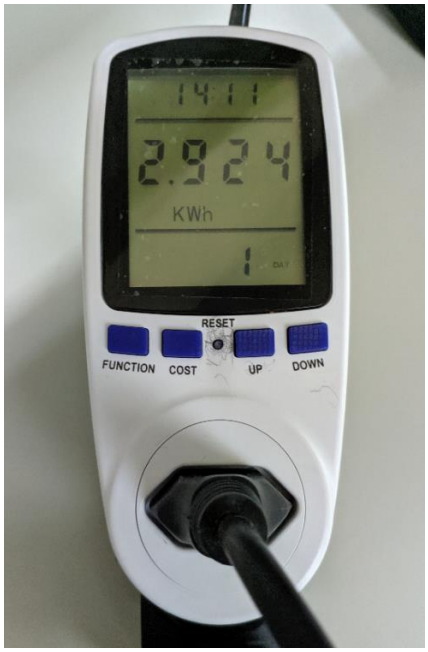
	Básica	Profissional		micro-PaaS Fog
	Free	Standard 1x	Standard 2x	
Para que serve?	Testes de aplicações em nuvem em uma sandbox limitada	Visibilidade, desempenho e disponibilidade aprimorados para alimentar aplicações de produção.		Visibilidade, desempenho, alta disponibilidade, orquestração, escalabilidade independentes e ilimitadas para aplicações IoT
RAM	512MB	512MB	1GB	1GB
Nº de tipos de processos	2	ilimitado	ilimitado	ilimitado condicionado a expansão de hardware
Sempre ligado	adormece após 30 minutos de inatividade; sempre depende do restantes de horas do dyno mensal	✓	✓	✓
Escala horizontal	✘	✓	✓	✓
Métricas da aplicação	✘	2 horas com resolução de 1 minuto,		sem limite de dias ou horas, resolução dempendente da infraestrutura de monitoramento adotada

		24 horas com resolução de 10 minutos, 3 dias com resolução de 1 hora, 7 dias com resolução de 2 horas		
Dedicado	✘	✘	✘	✓
Combinação de múltiplos dyno	✘	Pode ser combinado com dynos de desempenho [6]		Pode ser combinado com múltiplos containers
Custos	0	R$ 103,25* p/ dyno mensal	R$ 206,50 * p/ dyno mensal	R$ 1635,00 (6 Raspberry Pi), ilimitados números de containers condicionado a expansão de hardware R$ 32,00 de energia da micro-PaaS Fog p/ mês

Fonte: Próprio autor, 2019

Para o TCO da micro-PaaS fog exibido no Quadro 7, considerou-se os custos de aquisição dos dispositivos *Raspberry Pi* que compõem o *cluster* do ambiente experimental utilizado. Além disso, incluiu-se o custo da energia elétrica consumida por estes dispositivos. Para estimar o consumo de energia da micro-PaaS Fog utilizou-se o monitor ilustrado na Figura 36, que fez a aferição em KWh por um período de 72 horas do consumo de energia do *cluster fog* e todos os serviços em execução. Neste período o equipamento registrou um consumo total de 5,53KWh, e uma média de 0,078KWh a cada 1(uma) hora de consumo.

Figura 36 – Medidor do Consumo de Energia



Fonte: Próprio autor, 2019

[6] https://devcenter.heroku.com/articles/dyno-types

* taxa de conversão real / dólar = 4,13

A Tabela 5 traz o demonstrativo em KWh consumido mensalmente pela micro-PaaS Fog. O consumo mensal estimado foi de 55,25 KWh e um custo aproximado de 32 reais. Este valor foi composto por dois tipos de tarifas (novembro de 2019) praticados pela concessionária de energia que atende o IFS campus Lagarto:

- Tarifa ponta – período de utilização das 17:30h às 20:30h;
- Tarifa fora ponta – demais horários.

Tabela 5 – Consumo de energia mensal da micro-PaaS Fog

	Tarifa KWh		KWh (mês)		Total
Tarifa Ponta	R$	2,13	6,9	R$	14,70
Tarifa Fora Ponta	R$	0,36	48,35	R$	17,26
TOTAL GASTO POR MÊS				**R$**	**31,96**

Fonte: Próprio autor, 2019

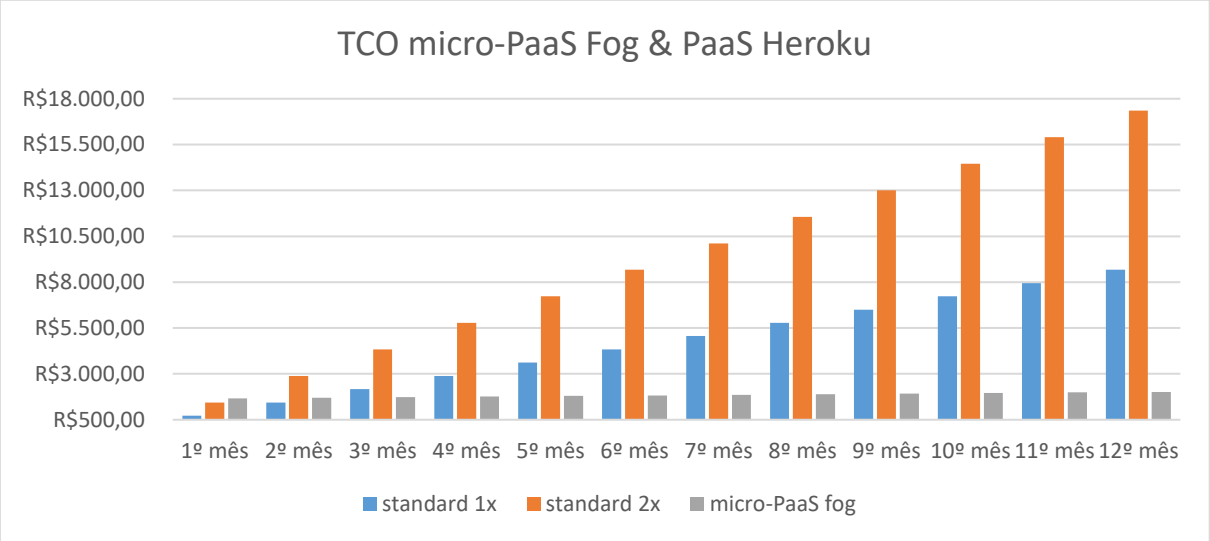
A Figura 37 ilustra as aplicações em execução no *cluster fog* por meio do gerenciador Portainer. Observa-se um total de 7 (sete) aplicações em execução, cada uma instanciada em um único *container*, para uma estrutura mínima necessária. Destas, observam-se 4 (quatro) instâncias do SysMorea; 1 (uma) instância do Reverseproxy; 1 (uma) instância do Redis e 1 (uma) instância do Portainer.

Figura 37 – Aplicações em execução cluster fog

Fonte: Próprio autor, 2019

Com base na quantidade de *containers* em execução no *cluster fog* e a quantidade de KWh estimada no mês, gerou-se um gráfico comparativo dos custos mensais entre a micro-PaaS Fog e a PaaS Heroku, ilustrado na Figura 38. É possível observar que no 2º mês o custo da micro-PaaS Fog é bem próximo da versão standard 1x, e 41,23% menor do que a versão standard 2x. A partir do 3º mês fica evidente a economia do custo da micro-PaaS Fog em relação as duas versões profissionais da Heroku. Ao final de 12 meses o custo com micro-PaaS Fog representou 23% do TCO da versão standard 1x e 12% do TCO da versão standard 2x.

Figura 38 – TCO micro-PaaS Fog Vs PaaS Heroku



Fonte: Próprio autor, 2019

A micro-PaaS fog implementada nesta pesquisa mostrou-se extremamente superior a versão free. Se comparada as versões profissionais Standard 1x e Standard 2X possui o dobro de memória da primeira e o mesmo da segunda. As métricas das aplicações podem ser consultadas por um período maior e com menor resolução o que possibilita uma leitura mais fidedigna das informações. Além disso, apresentou TCO muito menor do que a Heroku, atendendo de forma satisfatória às aplicações implementadas no *cluster fog* de forma eficiente ao custo fixo de aproximadamente 1635 reais relativos aos SBCs e mais um custo mensal médio de energia no valor de 32 reais pelo consumo de todo o cluster.

É importante ressaltar que essa comparação do TCO reforça nossa hipótese de que a micro-PaaS fog é uma solução viável, e economicamente vantajosa para ambientes de IoT, pois permite que dependamos menos dos recursos dos provedores de nuvem, e consequentemente pagando menos por seus serviços.

6.3 Latência

Uma das principais características da *fog* é a redução da latência em razão dos serviços estarem localizados mais próximos dos usuários finais. Para suportar baixa latência deve haver uma organização hierárquica na arquitetura, sendo a *cloud* representada na camada superior, a *fog* na camada intermediária, e os dispositivos inteligentes na camada inferior. Nesse sentido, para atestarmos a eficiência da micro-PaaS fog no tocante a latência utilizou-se os três tipos de situações encontradas no ambiente:

- latência mote – Heroku (L_{mh}) , que consiste no tempo decorrido para envio de dados do *mote* diretamente para o provedor Heroku na nuvem, sem utilização da micro-PaaS fog.
- latência mote – *fog* (L_{mf}), que consiste no tempo decorrido para envio de dados do *mote* para a micro-PaaS fog.
- latência *fog* – Heroku (L_{fh}), que consiste no tempo decorrido para envio de dados da micro-PaaS fog para o provedor Heroku na nuvem.

Para gerar as métricas que possibilitaram a análise da latência L_{fh} utilizou-se o *script shell CheckLatencia* conforme visualizado no Quadro 8. Este *script* realiza a medição do tempo decorrido entre a requisição do envio dos dados e a resposta de confirmação por parte do servidor. Por tanto, a latência medida pode ser considerada equivalente ao *Round Trip Time* (RTT)

Quadro 8 – *Script shell CheckLatencia*

Check Latencia
#!/bin/bash # Script to send data to Heroku # Author: Rubens Matos e Walter E. Santo # Date: 2019-09-20 outfile="log-heroku-availability.txt" while [True] do touch $outfile timestamp=`date +"%d-%m-%Y %H:%M:%S"`

```
timeout 5 curl -s -o /dev/null -w "%{time_starttransfer}\n"
morea.herokuapp.com/conn.php?localColeta=CTI&nodeID=EMote01&consumoAtual=3&consumo
Total=100
timeout 5 curl -s -o /dev/null -w "%{time_starttransfer}\n"
morea.herokuapp.com/conn.php?localColeta=CTI&nodeID=WMote01&consumoAtual=5&consum
oTotal=70
status=$?
if [[ "$status" -eq 0 ]]
then
  echo $timestamp" Up">>$outfile
else
  echo $timestamp" Down">>$outfile
fi

sleep 60
done
```

Fonte: Próprio autor, 2019

Já para L_{mh} e L_{mf} adcionou-se às *firmwares* constantes nos APÊNDICES B.3 (Emotes) e B.4 (Wmotes) comandos para registrar a diferença de tempo entre o instante imediatamente anterior ao envio dos dados e o instante em que o *mote* recebe a confirmação. A única diferença entre a forma como é realizada a medição de L_{mh} e a medição de L_{mf} está na URL utilizada para envio dos dados. O Quadro 9 ilustra o código adicionado ao *firmware*.

Quadro 9 – Comandos para medição de L_{mh} e L_{mf}

Check Latencia

```
// Criando uma conexao TCP
WiFiClient client;

// Registra o tempo inicial, imediatamente antes da conexão com o servidor
startTime=millis();
if (!client.connect(host, httpPort)) {
  return;
}

// Calcula o tempo para estabelecimento de conexão
connTime=millis()-startTime;

//Envia dados para o servidor através do método GET do HTTP
//Código específico para Emote ou Wmote
(...)

// Calcula o tempo para o início da confirmação do servidor após envio dos dados
int it=0;
while (client.connected()){
```

```
    String response = client.readStringUntil('\n');
    if (it==0){
      respTime=millis()-startTime;
    }
    it++;
    if (response == "/r"){
      Serial.println("Headers received");
      break;
    }
  }

  // Calcula o tempo total após o fim da mensagem de confirmação do servidor.
  transfTime=millis()-startTime;
  sentSamples++;

 // Exibe quantidade de amostras, tempo de conexão, tempo de início e de fim da resposta
  Serial.print(sentSamples);
  Serial.print(",");
  Serial.print(connTime);
  Serial.print(",");
  Serial.print(respTime);
  Serial.print(",");
  Serial.println(transfTime);
```

Fonte: Próprio autor, 2019

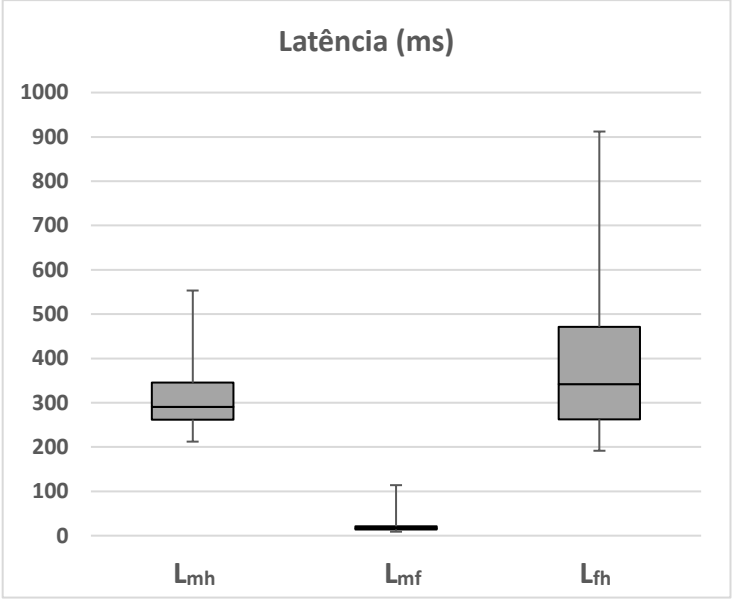
As análises das latências foram representadas através do gráfico *box plot*. Neste tipo de gráfico são representados 5 dados estatísticos: o mínimo, o primeiro quartil (Q1), a mediana, o terceiro quartil (Q3) e o máximo. Além disso, este tipo de representação gráfica pode apresentar *outliers*, ou seja, dados que se diferenciam drasticamente dos demais e que podem afetar decisões a serem tomadas a partir da análise dos dados se não forem devidamente considerados.

O gráfico *box plot* ilustrado na Figura 39 representa L_{mh} e L_{mf} durante o processo de comunicação entre o *mote* e a aplicação hospedada, respectivamente, no Heroku e na *fog*; e por fim, L_{fh} durante o processo de comunicação entre *fog* e Heroku.

Considerando 60 amostras em cada cenário a latência média L_{mf} obtida foi de 26ms, valor que representa 7,41% da média da latência em L_{mh} (362,40ms). A latência máxima em L_{mf} foi menor do que a latência mínima em L_{mh}, já o coeficiente de variação encontrado em L_{mf} foi de 1,58 enquanto em L_{mh} o seu coeficiente de variação foi de 5,25. Vale ressaltar que o baixo coeficiente de variação em L_{mf} indica uma grande estabilidade na comunicação entre *mote* e *fog*.

A média da latência L_{fh} foi de 473ms, a maior entre todas. Isto ocorre devido à virtualização dos processos de rede internos ao *docker* que faz parte da micro-PaaS fog. Apesar disso, este resultado não tem impacto significativo para a aplicação SysMorea, já que apenas dados consolidados serão enviados da *fog* para o Heroku, e a uma frequência de envio baixa.

Figura 39 – Latência micro-PaaS Fog



Fonte: Próprio autor, 2019

O intervalo dos valores encontrados, em cada cenário, entre Q1 e Q3, representam 50% dos valores encontrados na respectiva amostra. Em L_{mh} o valor encontrado em Q1 foi de 311ms e o valor encontrado em Q3 foi de 395ms. Nesse sentido, L_{mf} obteve 19ms como valor de Q1 e Q3 de valor igual a 26ms. Já, L_{fh} obteve em Q1 o valor de 333ms e 543ms como valor de Q3.

Os resultados obtidos demonstram a eficiência da *fog* mesmo utilizando equipamentos com custos reduzidos como os adotados por esta pesquisa. A arquitetura implementada mostra-se satisfatória às aplicações em tempo real destinadas a IoT e que são sensíveis à latência em razão das características intrínsecas de cada ambiente.

7

Conclusão

Neste capítulo são apresentadas as principais conclusões relacionadas ao tema central, as publicações realizadas e submetidas, as limitações e dificuldades encontradas, bem como, os trabalhos futuros.

7.1 Principais Conclusões

Esta dissertação de mestrado propôs e implementou uma arquitetura de micro-PaaS Fog, de baixo custo e desempenho satisfatório, para aplicações da Internet das Coisas que pode ser utilizada como alternativa ou complemento às grandes PaaS proprietárias em nuvem. Como tecnologia principal utilizou-se soluções de *containers*, por serem mais leves do que as virtualizações tradicionais e por serem mais indicadas na utilização com s*ingle board computer* (SBC). Neste trabalho foram apresentados aspectos de concepção da *fog computing* permitindido a flexibilidade necessária para atender aos requisitos de escalabilidade, mecanismos de orquestração e containerização de aplicações para IoT. Além disso, realizou-se o mapeamento sistemático da literatura para obtenção de trabalhos relacionados que pudessem direcionar esta pesquisa, sobretudo, nas métricas avaliadas e na identificação dos critérios de avaliação – heterogeneidade, gerenciamento de QoS, escalabilidade, mobilidade, federação e interoperabilidade – necessários para prover de forma eficaz e eficiente a arquitetura objeto desse estudo.

Tendo em vista os aspectos observados na literatura e por estarem alinhados à proposta central deste trabalho, a camada *fog* da arquitetura foi constituída baseada no *Raspberri Pi* e na tecnologia de *container* Docker. O processo de orquestração que acontece na *fog orchestration*

overlay, camada sobreposta à camada *fog*, é gerenciado pelo *Swarm* que se encarrega, entre outras coisas, na gestão dos nodes *manager* e *workers*, na distribuição de *workloads*, e principalmente, na definição dos atributos dos serviços, como, por exemplo, quantidade de réplicas, configurações de rede, portas de publicação, etc. Esta camada também foi a responsável pela escalabilidade, alta disponibilidade, balanceamento de carga e monitoramento do ambiente, provendo solução de alto nível com suporte ao processamento distribuído de informações do consumo de água e energia do IFS Campus Lagarto, através de dispositivos inteligentes.

A arquitetura implementada mostrou-se eficiente por meio das análises dos resultados encontrados, como, por exemplo, a avaliação das métricas relacionadas à alta disponibilidade, ao balanceamento de carga e à latência. Estas métricas foram obtidas por meio da elaboração de *scripts shell* e alterações no código do *firmware* gravado nos *motes*, devidamente estruturados para coletar da forma mais fidedigna possível todas as informações do ambiente.

Com a análise da alta disponibilidade e balanceamento de carga chegou-se ao cenário ideal, que seria o escalonamento do ambiente de orquestração com 4 (quatro) réplicas instanciadas da aplicação SysMorea, levando-se em consideração um intervalo de confiança (IC) de 95% este cenário obteve tempo médio de recuperação de falhas de 1,33 segundos, desvio padrão de 0,49 e coeficiente de variação de 0,37.

Outra análise que deve ser levada em consideração é o baixo TCO encontrado na micro-PaaS Fog em relação ao TCO da PaaS Heroku. Já a partir do 3º mês a micro-PaaS Fog teria o menor TCO, e ao final de 12 meses, custaria 23% e 12%, respectivamente, do TCO das versões da Heroku standard 1x e standard 2x.

No tocante a redução da latência, uma das principais características dos ambientes *fog computing*, os testes realizados no ambiente implementado também mostraram-se promissores, uma vez que a latência média entre os *motes* e a *fog* foi de 26ms e o coeficiente de variação de 1,58 indicando a estabilidade da comunicação entre *mote e fog*.

Levando-se em consideração as avaliações realizadas com o estudo de caso que exploraram as funcionalidades em relação às 4 camadas da micro-PaaS Fog, as análises caracterizaram a arquitetura proposta como capaz de prover um ambiente estável para aplicações sensíveis a latência, como as destinadas à IoT, e que requeiram tempo reduzido de

recuperação de falhas. Outro dado a ser considerado é que a arquitetura consegue manter sua operação mesmo com eventuais desconexões com a Internet, já que, a comunicação da camada *fog* com a camada *cloud* em casos de falha com a Internet poderá ser realizada quando do restabelecimento da conexão preservando-se as informações geradas na origem dos dados.

Dado o exposto e em razão das avaliações discutidas no Capítulo 6, é possível concluir que todos os objetivos, tanto o geral quantos os específicos, foram atingidos, ratificando a hipótese deste trabalho: a tecnologia de conteinerização *Docker Container* com *Docker Swarm* é capaz de prover uma micro-PaaS orquestrada para aplicações IoT em ambientes *fog computing* baseados em *clusters* de plataformas *Raspberry Pi*, com custo reduzido e respeitando critérios de QoS, como, por exemplo, escalabilidade, alta disponibilidade, balanceamento de carga e latência.

7.2 Limitações e dificuldades encontradas

A orquestração e implementação das aplicações utilizando tecnologia de *containers* no *Raspberry Pi* devido a sua arquitetura ARM foi uma das principais dificuldades encontradas, já que os trabalhos encontrados na literatura utilizando plataformas SBCs ainda são incipientes no campo da *fog computing* o que dificultou o processo de configuração de todo o ambiente.

Outra limitação encontrada está relacionada com a capacidade de uma única rede suportar diferentes aplicações sem comprometer QoS, já que, a micro-PaaS Fog foi concebida para suportar aplicações distintas e consequentemente implicou em vários tipos de tráfego de rede. Isso implicou em um esforço extra para desenvolver mecanismo para analisar as métricas do ambiente e consequentemente a validação da arquitetura. Além disso, fornecer garantias de QoS em redes sem fio, devido as restrições e limitações que os dispositivos IoT possuem também foi um desafio encontrado e que foi superado.

Questões relacionadas com problemas na execução e coletada dos dados dos dispositivos IoT causadas por instabilidade na alimentação elétrica destes dispositivos também foram superadas substituindo a fonte de alimentação por uma de melhor qualidade.

7.3 Trabalhos Futuros

A partir dos resultados expostos nesse trabalho de mestrado, pode-se vislumbrar algumas possibilidades de trabalhos futuros. Uma delas consiste em avaliar a arquitetura proposta por meio da ferramenta de emulação de larga escala para Internet das Coisas – VIoLET. Esta ferramenta é capaz de emular dispositivos *fog* utilizando *container* definindo características de rede (largura de banda e latência entre os dispositivos) e recursos (CPU, memória RAM e disco) de forma flexível e com a utilização de aplicações reais para validação do ambiente (BADIGER, BAHETI E SIMMHAN, 2018).

Uma outra extensão possível ao trabalho aqui exposto seria avaliar o desempenho do *cluster fog* utilizando a ferramenta de *benchmark* RIoTBench desenvolvidas para avaliar sistemas de processamento de fluxo distribuído, do inglês, *Distributed Stream Processing System* (DSPS) para aplicações IoT. O RIoTBench consiste em 27 tarefas frequentemente utilizados por aplicações IoT sendo capaz de gerar cargas de trabalho para auxiliar na avalição de métricas para diversas DSPS, direcionados para aplicações IoT emergentes em infraestrutura para computação em nuvem, contudo, o uso desta técnica também pode ser um campo potencial de pesquisa baseada em *fog computing* (SHUKLA, CHATURVEDI E SIMMHAN, 2017).

Uma outra proposta de trabalho futuro seria otimizar o consumo de energia utilizando técnicas *Dynamic Voltagen Frenquency Scaling* (DVFS). Por meio desta técnica é possível mudar dinamicamente a tensão e a frequência de um *host* em relação à carga da CPU. Técnicas de DVFS são bem consolidados em ambientes de computação na nuvem, contudo, o uso desta técnica ainda é pouco explorado em ambientes de *fog computing*. Nesse sentido, uma proposta de pesquisas futura seria implementar os *nodes fog* no ambiente de acordo com a demanda de carga de trabalho processada na CPU (KOLPE, ZHAI, SAPATNEKAR, 2011).

A arquitetura aqui proposta também pode ser melhorada através da implementação de mecanismos para escalar de forma automática a quantidade de instâncias das aplicações de acordo com a carga de trabalha empreendida no ambiente. Com o escalonamento das aplicações de forma automática, em função da carga de trabalho, os recursos da *fog* poderiam ser melhor aproveitados, sobretudo, os relacionados à processamento, à memória e aos dados trafegados na rede.

7.4 Publicações Realizadas

Nesta seção encontram-se descritos os artigos publicados no andamento desta dissertação de mestrado. Os artigos contemplam os principais esforços alcançados até o momento, bem como sua projeção nas pesquisas realizadas junto ao Grupo de Pesquisa de Sistemas Distribuídos do Departamento de Computação da Universidade Federal de Sergipe.

- **Systematic Mapping on Orchestration of Container-based Applications in Fog Computing** (QUALIS A2)
  SANTO, E. WALTER; MATOS, JÚNIOR, R.S.; RIBEIRO, ADMILSON; SOUZA, DANILO; SANTOS, RENEILSON; **Systematic Mapping on Orchestration of Container-based Applications in Fog Computing**. *15th International Conference on Network and Service Management (CNSM 19)*, 2019, outubro.

A publicação apresenta o Mapeamento Sistemático da Literatura com o objetivo de entender e identificar as métricas e lacunas da literatura atual sobre orquestração de aplicações baseados em *container*, especialmente aqueles hospedados em *clusters* de plataformas *Single Board Computer* (SBC), como o *Raspberry Pi*, que foram usados para implementação de ambientes da *fog computing*. Além disso, o artigo identificou as principais características e requisitos para orquestração eficiente neste ambiente traçando um estudo comparativo dos critérios de avaliação – heterogeneidade, gerenciamento de QoS, escalabilidade, mobilidade, federação e interoperabilidade.

- **Machine learning algorithms to detect DDoS attacks in SDN** (QUALIS A2)
  SANTOS, R., SOUZA, D., SANTO, W., RIBEIRO, A., & MORENO, E. **Machine learning algorithms to detect DDoS attacks in SDN**. Concurrency and Computation: Practice and Experience, e5402.
  https://doi.org/10.1002/cpe.5402

O artigo traz uma análise de ataques *Distributed Denial of Service* (DDoS) e sugere a implementação de quatro algoritmos de aprendizado de máquina (SVM, MLP, Decision Tree e Random Forest) com o objetivo de classificar estes ataques em um ambiente simulado por SDN. Os ataques DDoS foram simulados usando-se a ferramenta Scapy com uma lista de IPs válidos, adquirindo, como resultado, a melhor precisão com o algoritmo Random Forest e o melhor tempo de processamento com o algoritmo Decision Tree. Além disso, são mostrados os recursos

mais importantes para classificar ataques DDoS e algumas desvantagens na implementação de um classificador para detectar os três tipos de ataques DDoS discutidos neste documento (ataque ao controlador, ataque à tabela de fluxo e ataque à largura de banda).

- **Internet of Things: A Survey on Communication Protocol Security** (QUALIS B3)
  SANTO, E. WALTER; RIBEIRO, ADMILSON; MORENO, E. D.; SALGUEIRO, R. J. R. DE B.; SOUZA, D.; SANTOS, RENEILSON; **Internet of Things: A Survey on Communication Protocol Security**. *ACM Conference - Euro American Conference on Telematics and Information Systems (EATIS 18),* 2018, novembro.
  ISBN: 978-1-4503-6572-7 doi>10.1145/3293614.3293644

A publicação apresenta um *survey* sobre os principais problemas de segurança que afetam os protocolos de comunicação no contexto da Internet das Coisas, a fim de identificar possíveis ameaças e vulnerabilidades. Os protocolos RFID, NFC, 6LoWPAN, 6TiSCH, DTSL, CoAP e MQTT, para uma melhor organização, foram explorados e categorizados em camadas de acordo com o modelo de referência TCP / IP. No final, um resumo é apresentado em forma de tabela com os modos de segurança usados para cada protocolo usado.

- **Uma revisão sistemática sobre a Segurança nos Protocolos de Comunicação para Internet das Coisas** (QUALIS B4)
  DO ESPÍRITO SANTO, WALTER, EDWARD ORDOÑEZ, AND ADMILSON RIBEIRO; **Uma revisão sistemática sobre a Segurança nos Protocolos de Comunicação para Internet das Coisas**. *Journal on Advances in Theoretical and Applied Informatics* 4.1 (2018): 1-9
  ISSN 2447-5033 DOI: https://doi.org/10.26729/jadi.v4i1.2482

O artigo apresenta uma revisão sistemática acerca dos aspectos relevantes voltados para segurança da Internet das Coisas, trazendo uma visão macro da necessidade urgente de serem adotadas medidas para mitigar ataques e vulnerabilidades aos protocolos atuantes na comunicação IoT. Foram identificadas as principais ameaças existentes e apresentou-se algumas sugestões como forma de mitigá-las. Foi possível, então, agrupar as ameaças encontradas por camada de acordo com a atuação de cada protocolo.

7.5 Publicações em Andamento

- **Micro PaaS Fog: Container Based Orchestration for IoT Applications Using SBC** (QUALIS A2)
  SANTO, E. WALTER; MATOS, JÚNIOR, R.S.; RIBEIRO, ADMILSON; SOUZA, DANILO; SANTOS, RENEILSON; **Micro PaaS Fog: Container Based Orchestration for IoT Applications Using SBC**. *International Journal of Grid and Utility Computing* (IJGUC) 2019, novembro.

O artigo propõe e implementa uma arquitetura de micro-PaaS em *fog computing*, em *cluster* de plataforma *single board computer* (SBC), para orquestração de aplicações utilizando *containers*, aplicada à Internet das Coisas e que atendam a critérios de QoS, como, por exemplo, alta disponibilidade, escalabilidade, balanceamento de carga e latência. A partir do modelo proposto, a micro-PaaS Fog foi implementada com tecnologia de virtualização em *containers* utilizando serviços de orquestração em um *cluster* formado por Raspberry Pi para monitoramento inteligente do consumo de água e energia à um custo total de propriedade (TCO) equivalente a 23% de uma plataforma como serviço (PaaS) pública.

Referências

AAZAM, M.; HUH, E.-N. Fog computing: The cloud-iot\/ioe middleware paradigm. **IEEE Potentials**, v. 35, n. 3, p. 40–44, 2016.

ABDELSHKOUR, MAHER. **Iot, from cloud to fog computing.** Disponível em: <http://blogs.cisco.com/perspectives/iot-from-cloudto-fog-computing>. Acesso em: 12 nov. 2018.

ABREU, D. P.; VELASQUEZ, K.; ASSIS, M. R. M.; BITTENCOURT, L. F.; CURADO, M.; MONTEIRO, E.; MADEIRA, E. **A Rank Scheduling Mechanism for Fog Environments**2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud). **Anais**...IEEE, 2018

AL FARUQUE, M. A.; VATANPARVAR, K. Energy management-as-a-service over fog computing platform. **IEEE internet of things journal**, v. 3, n. 2, p. 161–169, 2016.

AL-FUQAHA, A.; GUIZANI, M.; MOHAMMADI, M.; ALEDHARI, M.; AYYASH, M. Internet of things: A survey on enabling technologies, protocols, and applications. **IEEE Communications Surveys & Tutorials**, v. 17, n. 4, p. 2347–2376, 2015.

ANTONI, M.; VIVIAN, G. R.; PREUSS, E. Implementaçao de uma nuvem de armazenamento privada usando ownCloud e Raspberry PI. **Encontro Anual de Tecnologia da Informaç ao e Semana Acadêmica de Tecnologia da Informaç ao**, p. 55–62, 2015.

ATZORI, L.; IERA, A.; MORABITO, G. The internet of things: A survey. **Computer networks**, v. 54, n. 15, p. 2787–2805, 2010.

BABU, S. M.; LAKSHMI, A. J.; RAO, B. T. **A study on cloud based Internet of Things: CloudIoT**Communication Technologies (GCCT), 2015 Global Conference on. **Anais**...IEEE, 2015

BADIGER, S.; BAHETI, S.; SIMMHAN, Y. **VIoLET: A Large-scale Virtual Environment for Internet of Things**European Conference on Parallel Processing. **Anais**...Springer, 2018

BANANA PI. , 2019. Disponível em: <https://www.banana-pi.org>. Acesso em: 19 nov. 2019

BELLAVISTA, P.; ZANNI, A. **Feasibility of fog computing deployment based on docker containerization over raspberrypi**Proceedings of the 18th international conference on distributed computing and networking. **Anais**...ACM, 2017

BONOMI, F.; MILITO, R.; ZHU, J.; ADDEPALLI, S. **Fog computing and its role in the internet of things**Proceedings of the first edition of the MCC workshop on Mobile cloud computing. **Anais**...ACM, 2012

BOTTA, A.; DE DONATO, W.; PERSICO, V.; PESCAPÉ, A. Integration of cloud computing and internet of things: a survey. **Future Generation Computer Systems**, v. 56, p. 684–700, 2016.

BRITO, M. S. DE; HOQUE, S.; MAGEDANZ, T.; STEINKE, R.; WILLNER, A.; NEHLS, D.; KEILS, O.; SCHREINER, F. **A service orchestration architecture for fog-enabled infrastructures**Fog and Mobile Edge Computing (FMEC), 2017 Second International Conference on. **Anais**...IEEE, 2017

BROGI, A.; FORTI, S.; IBRAHIM, A. **How to best deploy your Fog applications, probably**Fog and Edge Computing (ICFEC), 2017 IEEE 1st International Conference on. **Anais**...IEEE, 2017

CADVISOR. , 2019. Disponível em: <https://github.com/google/cadvisor>. Acesso em: 19 nov. 2019

CARDELLINI, V.; GRASSI, V.; PRESTI, F. L.; NARDELLI, M. **On QoS-aware scheduling of data stream applications over fog computing infrastructures**Computers and Communication (ISCC), 2015 IEEE Symposium on. **Anais**...IEEE, 2015

CIRANI, S.; FERRARI, G.; IOTTI, N.; PICONE, M. **The iot hub: a fog node for seamless management of heterogeneous connected smart objects**Sensing, Communication, and Networking-Workshops (SECON Workshops), 2015 12th Annual IEEE International Conference on. **Anais**...IEEE, 2015

CISCO. **IOx and fog applications**. Disponível em: <https://www.cisco.com/c/en_in/solutions/internet-of-things/iot-fog-applications.html>. Acesso em: 16 out. 2018.

COMPUTING, F. the Internet of Things: Extend the Cloud to Where the Things are. **Cisco White Paper**, 2015.

DAGALE, H.; ANAND, S. V. R.; HEGDE, M.; PUROHIT, N.; SUPREETH, M. K.; GILL, G. S.; RAMYA, V.; SHASTRY, A.; NARASIMMAN, S.; LOHITH, Y. S. **Cyphys+: A reliable and managed cyber-physical system for old-age home healthcare over a 6lowpan using wearable motes**2015 IEEE International Conference on Services Computing. **Anais**...IEEE, 2015

DIAMOND SYSTEM CORPORATION. COM-Based SBCs: The Superior Architecture for Small Form Factor Embedded Systems. 2016.

DUA, R.; RAJA, A. R.; KAKADIA, D. **Virtualization vs containerization to support paas**Cloud Engineering (IC2E), 2014 IEEE International Conference on. **Anais**...IEEE, 2014

ETSI, N. Etsi gs nfv 002 v1. 1.1 network functions virtualization (nfv). **Architectural Framework. sl: ETSI**, 2013.

FELTER, W.; FERREIRA, A.; RAJAMONY, R.; RUBIO, J. **An updated performance comparison of virtual machines and linux containers**Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On. **Anais**...IEEE, 2015

GIANG, N. K.; BLACKSTOCK, M.; LEA, R.; LEUNG, V. C. **Developing IoT applications in the fog: a distributed dataflow approach**Internet of Things (IOT), 2015 5th International Conference on the. **Anais**...IEEE, 2015

GIL, A. C. Como elaborar projetos de pesquisa. **São Paulo**, v. 5, n. 61, p. 16–17, 2002.

GIUSTO, D. **A. lera, G. Morabito, l. Atzori (Eds.) The Internet of Things**. [s.l.] Springer, 2010.

GOGOUVITIS, S. V.; MUELLER, H.; PREMNADH, S.; SEITZ, A.; BRUEGGE, B. Seamless computing in industrial systems using container orchestration. **Future Generation Computer Systems**, 2018.

GRAFANA. **GRAFANA**. Disponível em: <https://grafana.com>.

GUBBI, J.; BUYYA, R.; MARUSIC, S.; PALANISWAMI, M. Internet of Things (IoT): A vision, architectural elements, and future directions. **Future generation computer systems**, v. 29, n. 7, p. 1645–1660, 2013.

HEROKU. **HEROKU**. Disponível em: <https://www.heroku.com/platform>.

HILL, J.; CULLER, D. **A wireless embedded sensor architecture for system-level optimization**. [s.l.] Citeseer, 2002.

HONG, K.; LILLETHUN, D.; RAMACHANDRAN, U.; OTTENWÄLDER, B.; KOLDEHOFE, B. **Mobile fog: A programming model for large-scale applications on the internet of things**Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing. **Anais**...ACM, 2013

HOQUE, S.; BRITO, M. S. DE; WILLNER, A.; KEIL, O.; MAGEDANZ, T. **Towards container orchestration in fog computing infrastructures**Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual. **Anais**...IEEE, 2017

HOU, X.; LI, Y.; CHEN, M.; WU, D.; JIN, D.; CHEN, S. Vehicular fog computing: A viewpoint of vehicles as the infrastructures. **IEEE Transactions on Vehicular Technology**, v. 65, n. 6, p. 3860–3873, 2016.

ISMAIL, B. I.; GOORTANI, E. M.; AB KARIM, M. B.; TAT, W. M.; SETAPA, S.; LUKE, J. Y.; HOE, O. H. **Evaluation of docker as edge computing platform**Open Systems (ICOS), 2015 IEEE Confernece on. **Anais**...IEEE, 2015

JAIN, R. **The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling**. [s.l.] John Wiley & Sons, 1990.

JIANG, Y.; HUANG, Z.; TSANG, D. H. Challenges and solutions in fog computing orchestration. **IEEE Network**, v. 32, n. 3, p. 122–129, 2018.

JOY, A. M. **Performance comparison between linux containers and virtual machines**Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in. **Anais**...IEEE, 2015

KHAN, A. Key Characteristics of a Container Orchestration Platform to Enable a Modern Application. **IEEE Cloud Computing**, n. 5, p. 42–48, 2017.

KITCHENHAM, B. Procedures for performing systematic reviews. **Keele, UK, Keele University**, v. 33, n. 2004, p. 1–26, 2004.

KOLPE, T.; ZHAI, A.; SAPATNEKAR, S. S. **Enabling improved power management in multicore processors through clustered DVFS**Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011. **Anais**...IEEE, 2011

KOZHIRBAYEV, Z.; SINNOTT, R. O. A performance comparison of container-based technologies for the cloud. **Future Generation Computer Systems**, v. 68, p. 175–182, 2017.

KUBERNETES. **KUBERNETES**. Disponível em: <https://kubernetes.io/pt/>.

KUKLIŃSKI, S.; DINH, K. T.; DESTRÉ, C.; YAHIA, I. G. B. **Design principles of generalized network orchestrators**Communications Workshops (ICC), 2016 IEEE International Conference on. **Anais**...IEEE, 2016

LEVIS, P.; MADDEN, S.; POLASTRE, J.; SZEWCZYK, R.; WHITEHOUSE, K.; WOO, A.; GAY, D.; HILL, J.; WELSH, M.; BREWER, E. TinyOS: An operating system for sensor networks. *In*: **Ambient intelligence**. [s.l.] Springer, 2005. p. 115–148.

LI, Y.; ANH, N. T.; NOOH, A. S.; RA, K.; JO, M. **Dynamic Mobile Cloudlet Clustering for Fog Computing**. *In*: INTERNATIONAL CONFERENCE ON ELECTRONICS, INFORMATION, AND COMMUNICATION (ICEIC). 2018

LXC. **Linux Containers**. Disponível em: <https://linuxcontainers.org>.

MAHMUD, R.; KOTAGIRI, R.; BUYYA, R. Fog computing: A taxonomy, survey and future directions. *In*: **Internet of everything**. [s.l.] Springer, 2018. p. 103–130.

MARATHON. **MARATHON**. Disponível em: <https://mesosphere.github.io/marathon/>.

MENEZES, A. M. S.; VIEIRA, A. M.; MATOS JUNIOR, R. S. Protótipo de sistema automatizado para monitoramento e controle do consumode eletricidade e água em instalações prediais. **dx.doi.org/10.13140/RG.2.2.33313.35681/1**, 2019.

MIKKILINENI, R.; MORANA, G.; ZITO, D. **Cognitive Application Area Networks: A New Paradigm for Distributed Computing and Intelligent Service Orchestration**Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2015 IEEE 24th International Conference on. **Anais**...IEEE, 2015

MOURADIAN, C.; NABOULSI, D.; YANGUI, S.; GLITHO, R. H.; MORROW, M. J.; POLAKOS, P. A. A comprehensive survey on fog computing: State-of-the-art and research challenges. **IEEE Communications Surveys & Tutorials**, v. 20, n. 1, p. 416–464, 2017.

MUNIR, A.; KANSAKAR, P.; KHAN, S. U. IFCIoT: Integrated Fog Cloud IoT: A novel architectural paradigm for the future Internet of Things. **IEEE Consumer Electronics Magazine**, v. 6, n. 3, p. 74–82, 2017.

OPENFOG CONSORTIUM. **OpenFog**. Disponível em: <http://www.openfogconsortium.org/>. Acesso em: 16 out. 2018.

ORANGE PI. , 2019. Disponível em: <https://www.orangepi.org>. Acesso em: 19 nov. 2019

OUEIS, J.; STRINATI, E. C.; BARBAROSSA, S. **The fog balancing: Load distribution for small cell cloud computing**Vehicular Technology Conference (VTC Spring), 2015 IEEE 81st. **Anais**...IEEE, 2015

OUEIS, J.; STRINATI, E. C.; SARDELLITTI, S.; BARBAROSSA, S. **Small cell clustering for efficient distributed fog computing: A multi-user case**Vehicular Technology Conference (VTC Fall), 2015 IEEE 82nd. **Anais**...IEEE, 2015

PAHL, C.; HELMER, S.; MIORI, L.; SANIN, J.; LEE, B. **A container-based edge cloud paas architecture based on raspberry pi clusters**Future Internet of Things and Cloud Workshops (FiCloudW), IEEE International Conference on. **Anais**...IEEE, 2016

PAHL, C.; LEE, B. **Containers and clusters for edge cloud architectures–A technology review**Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on. **Anais**...IEEE, 2015

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. **Systematic Mapping Studies in Software Engineering.**EASE. **Anais**...2008

PETTICREW, M.; ROBERTS, H. **Systematic reviews in the social sciences: A practical guide.** [s.l.] John Wiley & Sons, 2008.

PORTAINER. **PORTAINER**. Disponível em: <https://www.portainer.io>.

PRINCY, S. E.; NIGEL, K. G. J. **Implementation of cloud server for real time data storage using Raspberry Pi**Green Engineering and Technologies (IC-GET), 2015 Online International Conference on. **Anais**...IEEE, 2015

PROMETHEUS. **PROMETHEUS**. Disponível em: <https://prometheus.io>.

REDIS. **REDIS**. Disponível em: <https://redis.io>.

SANTORO, D.; ZOZIN, D.; PIZZOLLI, D.; DE PELLEGRINI, F.; CRETTI, S. **Foggy: a platform for workload orchestration in a Fog Computing environment**Cloud Computing Technology and Science (CloudCom), 2017 IEEE International Conference on. **Anais**...IEEE, 2017

SARKAR, S.; CHATTERJEE, S.; MISRA, S. Assessment of the Suitability of Fog Computing in the Context of Internet of Things. **IEEE Transactions on Cloud Computing**, v. 6, n. 1, p. 46–59, 2018.

SHI, W.; CAO, J.; ZHANG, Q.; LI, Y.; XU, L. Edge computing: Vision and challenges. **IEEE Internet of Things Journal**, v. 3, n. 5, p. 637–646, 2016.

SHUKLA, A.; CHATURVEDI, S.; SIMMHAN, Y. Riotbench: An iot benchmark for distributed stream processing systems. **Concurrency and Computation: Practice and Experience**, v. 29, n. 21, p. e4257, 2017.

SILVA, D. S. **Uma Arquitetura Autonômica para a Alocação de Recursos em ambientes Fog Computing**. Dissertação (Mestrado em Ciências da computação). São Cristóvão: Universidade Federal de Sergipe, 2018.

SOLTESZ, S.; PÖTZL, H.; FIUCZYNSKI, M. E.; BAVIER, A.; PETERSON, L. **Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors**ACM SIGOPS Operating Systems Review. **Anais**...ACM, 2007

STATDISK. **Statdisk Online**. Disponível em: <https://www.statdisk.org/>.

STEINMACHER, I.; CHAVES, A. P.; GEROSA, M. A. Awareness support in distributed software development: A systematic review and mapping of the literature. **Computer Supported Cooperative Work (CSCW)**, v. 22, n. 2–3, p. 113–158, 2013.

SUCHITRA, V. Tecnologias de orquestração de nuvem: explore suas opções. **IBM Developer Works**, 2016.

VAQUERO, L. M.; RODERO-MERINO, L. Finding your way in the fog: Towards a comprehensive definition of fog computing. **ACM SIGCOMM Computer Communication Review**, v. 44, n. 5, p. 27–32, 2014.

VELASQUEZ, K.; ABREU, D. P.; GONÇALVES, D.; BITTENCOURT, L.; CURADO, M.; MONTEIRO, E.; MADEIRA, E. **Service orchestration in fog environments**2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud). **Anais**...IEEE, 2017

VILLARI, M.; CELESTI, A.; TRICOMI, G.; GALLETTA, A.; FAZIO, M. **Deployment orchestration of microservices with geographical constraints for edge computing**Computers and Communications (ISCC), 2017 IEEE Symposium on. **Anais**...IEEE, 2017

WANG, Q.; ZHU, Y.; CHENG, L. Reprogramming wireless sensor networks: challenges and approaches. **IEEE network**, v. 20, n. 3, p. 48–55, 2006.

WEN, Z.; YANG, R.; GARRAGHAN, P.; LIN, T.; XU, J.; ROVATSOS, M. Fog orchestration for internet of things services. **IEEE Internet Computing**, v. 21, n. 2, p. 16–24, 2017.

XAVIER, M. G.; NEVES, M. V.; ROSSI, F. D.; FERRETO, T. C.; LANGE, T.; DE ROSE, C. A. **Performance evaluation of container-based virtualization for high performance computing environments**Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. **Anais**...IEEE, 2013

YI, S.; HAO, Z.; QIN, Z.; LI, Q. **Fog computing: Platform and applications**2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb). **Anais**...IEEE, 2015

ZENG, D.; GU, L.; GUO, S.; CHENG, Z.; YU, S. Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system. **IEEE Transactions on Computers**, v. 65, n. 12, p. 3702–3712, 2016.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. **Journal of internet services and applications**, v. 1, n. 1, p. 7–18, 2010.

Apêndices

APÊNDICE A Configuração do *cluster Swarm* e interface de gerenciamento do usuário

É detalhada a seguir os comandos utilizados na criação de um *cluster* contendo três *nodes managers* e três *nodes workers*. Para fins de redundância é necessário no mínimo três *nodes managers* em razão do algoritmo de consenso raft implementado no *Swarm*.

A.1 Iniciando o *cluster Swarm*

Definindo o *node manager* no *cluster Swarm*:

```
1 docker swarm init --advertise-addr 172.29.0.30:2377
```

Será exibida a informação na primeira linha que *Docker engine* deste *host* passou a atuar no modo *swarm*, especificamente como modo *manager*, bem como, o comando *docker swarm join* associado ao *token* gerado e o endereço IP do *manager*, por exemplo:

```
docker swarm join --token SWMTKN-1-336m3bzo74end5jf03wh64psbhj6x2inde051u
7qla0ecu1ovk-ed6aenzo4ks6l70vhzhws03bq 172.29.0.30:2377
```

É possível, também, visualizar os *tokens* por meio dos comandos:

```
1 docker swarm join-token --quiet worker
```

Após a execução do comando *docker swarm join* será exibida a afirmação:
This node joined a swarm as a worker.

```
1 docker swarm join-token --quiet manager
```

Após a execução do comando *docker swarm join* será exibida a afirmação:
This node joined a swarm as a worker.

Para adcionar *nodes workers* ou *managers* faz-se necessário colar o *token* correspondente seguido do IP:porta (*manager leader*) precedido do comando docker swarm join --token:

```
1 docker swarm join --token SWMTKN-1-336m3bzo74end5jf [...] IP:porta
```

A.2 Instanciando a interface de gerenciamento web Portainer como serviço

Independente de se inicializar uma instância (*container*) da imagem portainer como serviço, um aspecto importante para uso de *container* ou serviço de *containers* é a persistência de dados.

Dessa forma, criou-se um diretório que irá armazenar os dados do *container*, de modo, que ao finalizar um *container* e inicializa-lo novamente, o *Portainer* possa ler os dados das configurações já realizadas:

```
1 mkdir -p /data/orquestrador
```

Ao inicializar um *container* utilizando o recurso de serviço do *Docker Swarm* o ambiente poderá dispor de alta disponibilidade do *container*, deste modo, caso um node *manager* fique indisponível, o *Swarm* irá alocar outro *container* em um dos outros dois nós que estão em modo *manager*. Para que o *Swarm* possa alocar um *container* do *Portainer* no *cluster*, e tal alocação ocorra apenas em *nodes manager*, é necessário utilizar o recurso "*constraint"*, por meio da execução do comando no *node manager leader*:

```
1 docker service create --name portainer --publish 9000:9000 --replicas=1 --constraint 'node.role == manager' --mount type=bind,src=/data/orquestrador,dst=/data portainer/portainer
```

O serviço criado tem o nome portainer, o host manager na qual o *container* for iniciado irá publicar (expor a porta 9000), de modo que, toda conexão na porta 9000 do host será redirecionada para porta 9000 em *listening* no *container*.

O parâmetro --*constraint* informa que o *container* somente será iniciado caso o atributo *role* possua o valor *manager*, o que justamente faz com o que a instância do container inicie apenas em nós do tipo *manager* no *cluster*.

O parâmetro –mount type=bind,src=/data/orquestrador,dst=/data fará com que o diretório local (existente no node *manager* onde o *container* será iniciado) seja mapeado no diretório /data dentro do *container*, permitindo assim que todos os dados gravados ou lidos no diretório /data do *container* na verdade são dados existentes no diretório /data/orquestrador. Deste modo, o *container* terá persistência dos dados no momento que ele for finalizado.

O parâmetro –no-auth é particular dos binários do container portainer, e faz com que o orquestrador não exija usuário e senha em uma primeira execução, o que é interessante para apenas começarmos a explorar um pouco a ferramenta, mas iremos no final do artigo configurar a autenticação no orquestrador.

APÊNDICE B Códigos do Sistema de Monitoramento em tempo real do consumo de Eletricidade e Água (SysMorea) e dos Motes.

É detalhada a seguir o código de programação, desenvolvido na linguagem PHP, do Sistema de Monitoramento em tempo real do consumo de Eletricidade e Água (SysMorea).

B.1 Arquivo index.php

```
<?php
ini_set('display_errors', 1);
require "predis/autoload.php";
Predis\Autoloader::register();
try{
$redis = new Predis\Client(array(
                                        "scheme" => "tcp",
                                        "host" => "redis",
                                        "port" => 6379,
                                        "persistent" => "1"
                                        ));
date_default_timezone_set("America/Recife");
$consumoAtual = "";
$consumoTotal = "";
$localColeta = "";
$horaColeta = "";
}catch (Exception $e) {
        die($e->getMessage());
}
?>
<html>
<title>Projeto MOREA - Consumo de água e eletricidade no IFS Lagarto</title>
<style>
        body{
        background-image:url(fundo.jpg);
        background-attachment: fixed;
        background-size: 100% 100%;
        background-repeat: repeat-y;
        background-color: #000;
        }
</style>
<head>
        <meta http-equiv="refresh" content="60">
        <meta charset="utf-8">
<link href="https://fonts.googleapis.com/css?family=Open+Sans&display=swap" rel="stylesheet">
<style type="text/css">
body,td,th {
        color: #fff;
        font-family:'Open Sans', 'Verdana', sans-serif;
                font-size: 16px;
}
```

```
body p {
	text-align: center;
	font-size: smaller;
	font-family:'Open Sans', 'Verdana', sans-serif;
	padding:0px;
}
body td {
	text-align: center;
}
table {
	border: 3px solid white;
background-color: rgba(100, 100, 100, 0.4);
}
th {
	font-size: smaller;
}
#id {
text-align: left;
}
#id {
	color: #FFF;
}
#id {
	font-family: Verdana, Geneva, sans-serif;
}
#id {
	font-size: 36px;
}
#id {
	font-size: larger;
}
.local {
	font-size: 36px;
	text-align: left;
}
.rodape {
	font-size: smaller;
	text-align: center;
}
.headerGeral {
	vertical-align:top;
}
.headerConsumo {
	max-width: 140px;
	word-wrap:break-word;
}
.total {
	font-size: 36px;
}
#container {
   position: relative;
}
#copyright {
   position: absolute;
   bottom: 0;
}
.data {
```

```
		font-size: 24px;
}
.consumo strong {
		font-family: "Segoe UI Black";
		text-align: left;
}
.containerData {
   width: 95%;
   height: 300px;
   margin: auto;
   padding: 10px;
}
.containerWater {
   width: 45%;
   float: left;
}
.containerEnergy {
   margin-left: 55%;
   width: 45%;
}
.containerFooter {
   display:block !important;
   width: 50%;
   height: 100px;
   margin: 0 auto;
   margin-left: 25%;
   margin-right: 25%;
   padding: 5px;
   background: rgba(100, 100, 100, 0.6);
   text-align: center;
   position: fixed;
   z-index:9999999;
   bottom:0;
}
</style>
</head>
<body>
<div id="container">
<section class="containerData">
<div class="containerWater">
<h3 style="text-align:center;">Dados de consumo de água</h3>
<table>
		<tr>
			<th class="headerGeral">ID</th>
			<th class="headerGeral">Local</th>
			<th class="headerConsumo">Vazão instant. (L/H)</th>
			<th class="headerConsumo">Consumo acumulado (L)</th>
			<th class="headerConsumo">Data/hora da coleta</th>
		</tr>
<?php
		echo('<tr>');
		$wmoteList = array('WMote01','WMote02','WMote03','WMote04','WMote05');
			foreach($wmoteList as &$wmote){
			try{
				$dados = $redis->zrevrange($wmote,0,0);
				if (!empty($dados)){
					$vars=explode(":",$dados[0]);
```

```
                                        $timestamp=$vars[0];
                                        $consumoAtual=$vars[1];
                                        $consumoTotal=$vars[2];
                                        $localColeta=$vars[3];
                                        $horaColeta = date('d/m/y H:i:s',$timestamp);
                                }
                                echo('<td><p>'.$wmote.'</p></td>');
                                echo('<td><p>'.$localColeta.'</p></td>');
                                echo('<td><p>'.$consumoAtual.'</p></td>');
                                echo('<td><p>'.$consumoTotal.'</p></td>');
                                echo('<td><p>'.$horaColeta.'</p></td>');
                                echo('</tr>');
                        }catch (Exception $e) {
                                die($e->getMessage());
                        }finally{
                                $dados="";
                                $consumoAtual="";
                                $consumoTotal="";
                                $localColeta="";
                                $horaColeta="";
                        }
        }
?>
</table>
</div>
<div class="containerEnergy">
<h3 style="text-align:center;">Dados de consumo de eletricidade</h3>
<table>
        <tr>
                <th class="headerGeral">ID</th>
                <th class="headerGeral">Local</th>
                <th class="headerConsumo">Consumo último min. (Wh)</th>
                <th class="headerConsumo">Consumo acumulado (Wh)</th>
                <th class="headerConsumo">Data/hora da coleta</th>
        </tr>
<?php
        echo('<tr>');
        $emoteList = array('EMote01','EMote02','EMote03','EMote04','EMote05');

        foreach($emoteList as &$emote){
                try{
                        $dados = $redis->zrevrange($emote,0,0);

                        if (!empty($dados)){
                                $vars=explode(":",$dados[0]);
                                $timestamp=$vars[0];
                                $consumoAtual=$vars[1];
                                $consumoTotal=$vars[2];
                                $localColeta=$vars[3];
                                $horaColeta = date('d/m/y H:i:s',$timestamp);
                        }
                        echo('<td><p>'.$emote.'</p></td>');
                        echo('<td><p>'.$localColeta.'</p></td>');
                        echo('<td><p>'.$consumoAtual.'</p></td>');
                        echo('<td><p>'.$consumoTotal.'</p></td>');
                        echo('<td><p>'.$horaColeta.'</p></td>');
                        echo('</tr>');
```

```
                    }catch (Exception $e) {
                            die($e->getMessage());
                    }finally{
                            $dados="";
                            $consumoAtual="";
                            $consumoTotal="";
                            $localColeta="";
                            $horaColeta="";
                    }
            }
?>
</table>
</div>
</section>
<div class="containerFooter">
<p class="rodape">MOREA: Monitoramento em Tempo Real de Consumo de Eletricidade e Água no IFS Campus Lagarto</p>
<p class="rodape">Projeto apoiado pelo Programa PIBIC/PROPEX/IFS</p>
<p class="rodape">Acessando container: <?php $docker_cid=file_get_contents("/etc/hostname"); echo($docker_cid); ?></p>
</div>
<div style="text-align:left;position:fixed;z-index:9999999;bottom:0; width: 100%;cursor: pointer;line-height: 0;display:block !important;"><a target="_blank" href="https://www.ifs.edu.br"><img src="morea-project-logo-2.png" alt="Projeto MOREA - IFS Campus Lagarto"></a></div>
</html>
```

B.2 Arquivo Conn.php

Arquivo que recebe os dados do monitoramento de água e energia que são enviados pelos dispositivos IoT.

Armazena os dados no Redis para posterior visualização e processamento

```
<?php
ini_set('display_errors', 1);
require "predis/autoload.php";
Predis\Autoloader::register();
try{
$redis = new Predis\Client(array(
                        "scheme" => "tcp",
                        "host" => "redis",
                                        "port" => 6379,
                                        "persistent" => "1"
                                ));
echo("Connected to Redis");
}catch(Exception $e){
 die($e->getMessage());
}
/*Obtem dados passados através do HTTP GET*/
```

```
$consumoAtual = filter_input(INPUT_GET, 'consumoAtual', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_FRACTION);
$consumoTotal = filter_input(INPUT_GET, 'consumoTotal', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_FRACTION);
$localColeta = filter_input(INPUT_GET, 'localColeta', FILTER_DEFAULT);
$nodeID =  filter_input(INPUT_GET, 'nodeID', FILTER_DEFAULT);
date_default_timezone_set("America/Recife");
$horaData = date('d/m/Y H:i:s');
$format = "d/m/Y H:i:s";
$dateobj = DateTime::createFromFormat($format, $horaData);
$timestamp = $dateobj->getTimestamp();
if (is_null($consumoTotal) || is_null($consumoAtual)) {
  //Gravar log de erros
  die("Dados inválidos");
}
/* Armazena dados no Redis */
try{
$redis->zadd($nodeID,[$timestamp.":".$consumoAtual.":".$consumoTotal.":".$localColeta => 0]);
}catch(Exception $e){
 die($e->getMessage());
}
?>
```

B.3 Firmware EMote

Arquivo com as configurações do *firmware* gravado no NodeMCU que habilita o pleno funcionamento dos dispositivos IoT utilizados para fazer a leitura do consumo de eletricidade de forma inteligente:

```
//Incluindo as bibliotecas necessárias
#include <ESP8266WiFi.h>
#include <EmonLib.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27, 16, 2);


// Nome da sua rede Wifi
const char* ssid = "IFSLAB";
// Senha da rede
const char* password = "********";

// Site que receberá os dados - IMPORTANTE: SEM O HTTP://
const char* host = "morea.ifslab.edu.br";
```

```
EnergyMonitor SCT013;

int pinSCT = A0;   //Pino analógico conectado ao SCT-013
int tensao = 127;
int potencia;
int i=0;
double corrente=0;
double somaPotencia=0;
double somaCorrente=0;
double consumoAtual=0;
double consumoTotal=0;
double mediaPotencia=0;
double mediaCorrente=0;

void setup()
{
  Serial.begin(9600);
  Serial.println("Ligando");

  lcd.init();   // Inicializa o LCD
  lcd.backlight(); // Habilita o backlight
  lcd.print("Medidor Corrente"); // Mensagem de inicialização no LCD

  //Pino e valor de calibração do sensor.
  //Calibração (9.090909) = Número de voltas da bobina interna do sensor (2000), dividido pelo valor do resistor de carga utilizado (220 ohms)
  //Ver https://portal.vidadesilicio.com.br/sct-013-sensor-de-corrente-alternada/
  SCT013.current(pinSCT, 9.090909);

  delay(1000);  // Espera 1 segundo
  lcd.clear();  // Apaga a tela do LCD

  conexaoWiFi(); // Realiza conexão à rede WiFi
}

void loop()
{
  // Calcula o valor da Corrente RMS e converte para corrente instantânea (pico).
  // 1480 é o número de amostras padrão da biblioteca EmonLib. Ver: https://openenergymonitor.org/forum-archive/node/11143.html
  corrente = SCT013.calcIrms(1480)/0.707;
  potencia = corrente * tensao;         // Calcula o valor da Potencia Instantanea

  //lcd.setCursor(0,0);

  //String corr = "Corrente(A): " + String(corrente) + "   ";
  //lcd.print(corr);
  //Serial.println(corr);

  //lcd.setCursor(0,1);

  //String pot = "Potencia(W): " + String(potencia);
```

```
  //lcd.print(pot);
  //Serial.println(pot);

  // Acumula potencia e corrente para calculo da média do minuto
  somaPotencia+=potencia;
  somaCorrente+=corrente;
  i++;
  //Serial.print(i);

  // Envia os dados a cada 60 segundos
  if(i == 6){
    //Calcula media da potência no ultimo minuto
    mediaPotencia=somaPotencia/6;
    mediaCorrente=somaCorrente/6;

    //Calcula consumo em Watt-hora (Wh)
    consumoAtual=mediaPotencia/60;
    consumoTotal+=consumoAtual;

    // Criando uma conexao TCP
    WiFiClient client;
    const int httpPort = 80;
    if (!client.connect(host, httpPort)) {
      return;
    }

    //Envia dados para o servidor através do método GET do HTTP
    client.print("GET /morea/conn.php?consumoAtual=" + String(consumoAtual,2) +
                 "&consumoTotal=" + String(consumoTotal,2) + "&mediaPotencia=" +
                 String(mediaPotencia,2) + "&mediaCorrente=" + String(mediaCorrente,2) +
                 "&localColeta=Datacenter&nodeID=EMote01" + " HTTP/1.1\r\n" +
                 "Host: " + String(host) + "\r\n" + "Connection: close\r\n\r\n");

    resetVariaveis();
    //lcd.clear();
  }

  delay(10000);//Espera dez segundos para fazer próxima medição
}

void conexaoWiFi(){
  // Conectando na rede wifi
  Serial.println("");
  Serial.print("Conectando");
  WiFi.begin(ssid,password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Conectado a rede: ");
  Serial.println(ssid);
```

```
  printWifiData();
}


void printWifiData() {
  // print your WiFi shield's IP address:
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");
  Serial.println(ip);

  // print your MAC address:
  byte mac[6];
  WiFi.macAddress(mac);
  Serial.print("MAC address: ");
  Serial.print(mac[0], HEX);
  Serial.print(":");
  Serial.print(mac[1], HEX);
  Serial.print(":");
  Serial.print(mac[2], HEX);
  Serial.print(":");
  Serial.print(mac[3], HEX);
  Serial.print(":");
  Serial.print(mac[4], HEX);
  Serial.print(":");
  Serial.println(mac[5], HEX);

  // print your subnet mask:
  IPAddress subnet = WiFi.subnetMask();
  Serial.print("NetMask: ");
  Serial.println(subnet);

  // print your gateway address:
  IPAddress gateway = WiFi.gatewayIP();
  Serial.print("Gateway: ");
  Serial.println(gateway);
}

void resetVariaveis(){
  i = 0;
  somaPotencia=0;
  somaCorrente=0;
  mediaPotencia=0;
  mediaCorrente=0;
}
```

B.4 Firmware WMote

Arquivo com as configurações do *firmware* gravado no NodeMCU que habilita o pleno funcionamento dos dispositivos IoT utilizados para fazer a leitura do consumo de água de forma inteligente:

```
//Incluindo as bibliotecas necessárias
#include <ESP8266WiFi.h>

#define PIN_SENSOR  13 //Pino a ser utilizado para o sensor. GPIO 13 (D7 no NodeMCU v3)
#define DEBUG  1 //DEBUG = 1 (habilita mensagens no monitor serial)

//Variaveis necessarias
volatile byte contaPulso; //Variável para a quantidade de pulsos
float freq; //Variável para frequência em Hz (pulsos por segundo)
float vazao=0; //Variável para armazenar o valor em L/min
float litros = 0; //Variável para o volume de água em cada medição
float volume = 0; //Variável para o volume de água acumulado
byte i;
const byte intervalo = 5; //Intervalo de coleta das amostras (em segundos)

// Nome da sua rede Wifi
const char* ssid = "IFSLAB";
//const char* ssid = "";
// Senha da rede
const char* password = "********";
//const char* password = "";
// Site que receberá os dados - IMPORTANTE: SEM O HTTP://
const char* host = "morea.ifslab.edu.br"; //www.site.com.br
//const char* host = "";
const int httpPort = 80;

void ICACHE_RAM_ATTR incpulso();

void setup() {
  // Iniciando o Serial
  Serial.begin(9600);

  pinMode(PIN_SENSOR, INPUT); //Configura o pino do sensor como entrada
  attachInterrupt(PIN_SENSOR, incpulso, RISING); //Associa o pino do sensor para interrupção

  delay(500);
  Serial.println("\nProjeto MOREA - Monitoramento em Tempo Real de Consumo de
Eletricidade e Água");

  conexaoWiFi();
}

void loop() {

  contaPulso = 0;//Zera a variável

  sei(); //Habilita interrupção
  delay (intervalo*1000); //Aguarda um intervalo X (em milisegundos)
  cli(); //Desabilita interrupção
```

```
 freq = (float) contaPulso / intervalo; // Calcula a frequência em pulsos por segundo

 vazao = freq / 8.0 ; //Converte para L/min, sabendo que 8 Hz (8 pulsos por segundo) = 1 L/min
 litros = vazao / (60/intervalo); //Recebe o volume em Litros consumido no intervalo atual.
 volume += litros; //Acumular o volume total desde o início da execução

 i++;

 if (DEBUG){
  //Exibe no monitor serial os dados coletados
  Serial.print("Vazao: ");
  Serial.println(vazao);
  Serial.print("Volume: ");
  Serial.println(volume);
  Serial.print("Número de coletas realizadas no minuto atual: ");
  Serial.println(i);
 }

 // Envia os dados a cada 60 segundos
 if(i == (60/intervalo)){

    i = 0;
    // Criando uma conexao TCP
    WiFiClient client;

    if (!client.connect(host, httpPort)) {
      return;
    }

    //Envia dados para o servidor através do método GET do HTTP
    client.print("GET /morea/conn.php?consumoAtual=" +
                 String(vazao,2) + "&consumoTotal=" + String(volume,2) +
                 "&localColeta=COINF-Bebedouro&nodeID=WMote01" +
                 " HTTP/1.1\r\n" + "Host: " + String(host) + "\r\n" +
                 "Connection: close\r\n\r\n"
                 );

    if (DEBUG){
      //Exibe no monitor serial os dados enviados
      Serial.println("Dados enviados!");
      Serial.print("Vazao: ");
      Serial.println(vazao);
      Serial.print("Volume: ");
      Serial.println(volume);
    }

 }


}
```

```
void ICACHE_RAM_ATTR incpulso ()
{
 contaPulso++; //Incrementa a variável de pulsos
}

void conexaoWiFi(){
  // Conectando na rede wifi
  Serial.print("Conectando");
  WiFi.begin(ssid,password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.print("Conectado a rede: ");
  Serial.println(ssid);

  printWifiData();
}

void printWifiData() {
  // print your WiFi shield's IP address:
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");
  Serial.println(ip);

  // print your MAC address:
  byte mac[6];
  WiFi.macAddress(mac);
  Serial.print("MAC address: ");
  Serial.print(mac[0], HEX);
  Serial.print(":");
  Serial.print(mac[1], HEX);
  Serial.print(":");
  Serial.print(mac[2], HEX);
  Serial.print(":");
  Serial.print(mac[3], HEX);
  Serial.print(":");
  Serial.print(mac[4], HEX);
  Serial.print(":");
  Serial.println(mac[5], HEX);

  // print your subnet mask:
  IPAddress subnet = WiFi.subnetMask();
  Serial.print("NetMask: ");
  Serial.println(subnet);

  // print your gateway address:
  IPAddress gateway = WiFi.gatewayIP();
```

```
  Serial.print("Gateway: ");
  Serial.println(gateway);
}
```

APÊNDICE C Contaneirização das aplicações e Instanciamento de Serviços

É detalhada a seguir os comandos utilizados na API do *Docker* para criação dos *containers* contendo as aplicações utilizadas neste trabalho de dissertação, bem como, os comandos necessários pra instanciamento dos serviços no *Swarm* para orquestração na *fog computing*.

C.1 *Container* Morea (Debian + nginx + PHP)

Definindo o SO. Executando um *container* Debian, nomeando-o como “morea” respondendo na porta 80:80

```
1 docker run --name morea -itd -p 80:80 debian
```

Atualizando os pacotes de instalação do SO

```
1 docker exec morea apt-get update
```

Instalando o servidor web Nginx

```
1 docker exec morea apt-get install nginx -y
```

Iniciando o servidor web

```
1 docker exec morea service nginx start
```

Instalando o Hypertext Preprocessor – PHP
Antes da instalação é necessário verificar quais versões do PHP são compatíveis com a versão do SO instalada.

```
1 docker exec morea apt-cache search php | grep fpm
```

Instalando o php-fmp

```
1 docker exec server apt-get install php5-fpm -y
```

Iniciando o php-fmp

```
1 docker exec morea service php5-fpm start
```

A partir dessa etapa todos os procedimentos de configuração foram realizados de dentro do container por meio do comando:

```
1 docker exec -it morea bash
```

Configurando o NGinx. O arquivo “default.conf” localizado em “etc/nginx/conf.d/” foi configurado da seguinte forma:

```
1 server {      # include /etc/nginx/naxsi.rules
      listen 80;

      server_name localhost; to FastCGI server
      #
      root /usr/share/nginx/html;
      index index.php;split_path_info ^(.+\.php)(/.+)$;
           try_files $uri =404;
      location / {
           # First attempt to serve request as file, then
           # as directory, then fall back to displaying a 404.
           try_files $uri $uri/ =404;AME $document_root$fastcgi_script_name;
           # Uncomment to enable naxsi on this location
      }

      location ~* \.(?:ico|css|js|gif|jpe?g|png)$ {
        expires 30d;
        add_header Pragma public;
        add_header Cache-Control "public";
      }
}
```

Após edição do arquivo de configuração o serviço NGinx foi reiniciado.

```
1 service nginx reload
```

Com o ambiente pronto foi copiado o código SysMorea para o diretório do NGinx /usr/share/nginx/html/morea/ e renomeado para index.php

Na sequência realizou-se o download do agente predis localizado em (https://github.com/nrk/predis/archive/v1.1.1.tar.gz) e o mesmo foi descompactado em /usr/share/nginx/html/morea/

C.2 *Container* Redis

O Redis é um banco de dados NoSQL, ou seja, não relacional do tipo chave-valor. O serviço foi instanciado na *Fog* com parâmetros da rede overlay fognetwork_overlay e a porta de publicação *default* 6379. Foi realizado o *pull* de uma imagem de arquitetura arm em razão da plataforma utilizada no ambiente.

```
1 docker service create -d --name redis --network fognetwork_overlay --publish 6379:6379 arm32v7/redis
```

C.3 *Reverse Proxy*

```
server {
   listen      8080;
   server_name  localhost;

   #charset koi8-r;
   #access_log  /var/log/nginx/host.access.log  main;

   location / {
   proxy_pass http://morea:80;
   #    root   /usr/share/nginx/html;
   #    index  index.html index.htm;
   }

   #error_page  404              /404.html;

   # redirect server error pages to the static page /50x.html
   #
   error_page   500 502 503 504  /50x.html;
   location = /50x.html {
```

```
      root   /usr/share/nginx/html;
    }

    # proxy the PHP scripts to Apache listening on 127.0.0.1:80
    #
    #location ~ \.php$ {
            #    proxy_pass   http://127.0.0.1;f Apache's document root
    # concurs with nginx's one
    #
    #location ~ /\.ht {
    #    deny  all;
    #}
}

    #    proxy_pass   http://127.0.0.1;
    #}

    # pass the PHP scripts to FastCGI server listening on 127.0.0.1:9000
    #
    #location ~ \.php$ {
    #    root html;
    #    fastcgi_pass   127.0.0.1:9000;
    #    fastcgi_index  index.php;
    #    fastcgi_param  SCRIPT_FILENAME  /scripts$fastcgi_script_name;
    #    include fastcgi_params;
    #}
            # deny access to .htaccess files, if Apache's document root
    # concurs with nginx's one
    #
    #location ~ /\.ht {
    #    deny  all;
    #}
}
```