UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

TECNÓLOGO EM REDES DE COMPUTADORES

RANDEL SOUZA ALMEIDA

UMA COMPARAÇÃO ENTRE CONTROLADORES DE REDES DEFINIDAS POR *SOFTWARE* EM CENÁRIOS DE INTERNET DAS COISAS

QUIXADÁ

2019

RANDEL SOUZA ALMEIDA

UMA COMPARAÇÃO ENTRE CONTROLADORES DE REDES DEFINIDAS POR *SOFTWARE* EM CENÁRIOS DE INTERNET DAS COISAS

Trabalho de Conclusão de Curso apresentado ao Curso de Redes De Computadores da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de Tecnólogo em Redes De Computadores. Área de concentração: Computação.

Orientador: Prof. Me. Marcos Dantas Ortiz

QUIXADÁ

2019

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A45c Almeida, Randel Souza.
Uma comparação entre controladores de redes definidas por software em cenários de internet das coisas / Randel Souza Almeida. – 2019.
97 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Redes de Computadores, Quixadá, 2019.
Orientação: Prof. Me. Marcos Dantas Ortiz.

1. Software-defined network (computer network technology). 2. Internet das coisas. I. Título.
CDD 004.6

RANDEL SOUZA ALMEIDA

UMA COMPARAÇÃO ENTRE CONTROLADORES DE REDES DEFINIDAS POR *SOFTWARE* EM CENÁRIOS DE INTERNET DAS COISAS

Trabalho de Conclusão de Curso apresentado ao Curso de Redes De Computadores da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de Tecnólogo em Redes De Computadores. Área de concentração: Computação.

Aprovada em: ___/___/____.

BANCA EXAMINADORA

Prof. Me. Marcos Dantas Ortiz (Orientador)
Universidade Federal do Ceará – UFC

Prof. Dr. Arthur de Castro Callado
Universidade Federal do Ceará - UFC

Prof. Dr. Paulo Antonio Leal Rego
Universidade Federal do Ceará - UFC

Dedico em especial, aos meus pais Maria Stênia e José Eremito, ao meu irmão Daniel, às minhas avós Luíza Lopes e Maria Ancelmo e a minha querida namorada Maria Jocivânia.

AGRADECIMENTOS

Agradeço antes de tudo a Deus, por me dar ânimo nos momentos mais difíceis e por permitir seguir em busca dos meus sonhos e objetivos com saúde, alegria e determinação.

Agradeço em especial, aos meus pais Maria Stênia e José Eremito, por todo amor, carinho, suporte e base de valores repassados. Ao meu irmão Daniel pelos momentos engraçados, que me proporcionaram boas risadas. A minha avó Luíza Lopes, que sempre me ajudou com o dinheiro do jantar do RU (Restaurante Universitário) e do transporte até Quixadá quando mais necessitei, além de ser sempre bastante receptiva, alegre e fazer um café delicioso! A minha avó Maria Ancelmo que mesmo estando mais distante sempre queria saber como estava meu rendimento na universidade. E a minha querida namorada Maria Jocivânia, por todo apoio, motivação e atenção quando mais precisei.

Agradeço ao Prof. Me. Marcos Dantas Ortiz, pela sua orientação, paciência e didática empregada. Por toda motivação que foi realizada nos momentos mais complicados deste trabalho, possibilitando que seguisse adiante na pesquisa. Sem a sua ajuda nada disso seria possível, obrigado por tudo!

Agradeço aos professores Prof. Dr. Arthur de Castro Callado e Prof. Dr. Paulo Antonio Leal Rego, por aceitarem ser membros da banca avaliadora e por todos os comentários e correções propostas para a melhoria deste trabalho.

Agradeço ao suporte dos criadores do Mininet-WiFi, respondendo a todas as dúvidas sobre os problemas que surgiram em relação ao emulador.

Agradeço a turma de Programador de Sistemas do SENAC Quixadá, foi uma experiência muito boa e repleta de novos ensinamentos e amigos.

Agradeço a todos os amigos e amigas pelos momentos de descontração, pelo aprendizado que foi repassado nos trabalhos em dupla e em equipe. Agradeço pela confiança que empregaram nos momentos que me pediam ajuda e também agradeço pela disponibilidade quando precisava de ajuda, enfim muito obrigado!

"O impossível não é um fato, ele é só uma opinião."

(Mario Sergio Cortella)

RESUMO

A Internet das Coisas (do inglês IoT) mostra-se um paradigma promissor ao possibilitar a conexão de objetos comuns à Internet. No entanto, surgem limitações de capacidade de memória bem como de processamento desses objetos. Ainda, a gestão e a otimização de desempenho da rede e a programabilidade de IoT tornaram-se complexas. Por conta disso, faz-se necessária uma nova abordagem para integrar redes IoT, tal como Redes Definidas por *Software* (do inglês SDN), pois esta arquitetura possibilita uma maior flexibilidade ao separar o plano de dados e o de controle. O objetivo geral do trabalho é comparar os controladores SDN: Ryu; POX e Floodlight; em cenários IoT, e com isso destacar qual controlador se saiu melhor em relação às métricas: latência; memória; consumo de processador; e tempo de execução dos experimentos. Espera-se que o presente trabalho possa ser utilizado por pesquisadores ou estudantes da área no processo de seleção do controlador SDN a ser usado em estudos ou testes em ambientes de IoT. Um dos resultados obtidos foi que o controlador Floodlight apresentou a menor latência média com até 104 nós IoT.

Palavras-chave: *Software-defined network (computer network technology)*. Internet das coisas.

ABSTRACT

The Internet of Things (IoT) is a promising paradigm in making it possible to connect common objects to the Internet. However limitations of memory capacity as well as processing of these objects arise. Also, the management and optimization of network performance and IoT programmability have become complex. Because of this, a new approach to integrating IoT networks, such as Software Defined Networks (SDN), is needed as this architecture allows for greater flexibility in separating the data plane from the control plane. The overall objective of the paper is to compare SDN controllers: Ryu; POX and Floodlight; in IoT scenarios, and thus highlight which controller did the best with respect to the metrics: latency; memory; processor consumption; and execution time of the experiments. It is expected that the present work can be used by researchers or students in the field in the process of selecting the SDN controller to be used in studies or tests in IoT environments. One of the results obtained was that the Floodlight controller presented the lowest average latency with up to 104 IoT nodes.

Keywords: Software-defined network (computer network technology). Internet of things.

LISTA DE FIGURAS

Figura 1 – Blocos de construção IoT 22
Figura 2 – Arquitetura do protocolo MQTT 24
Figura 3 – Abstração das camadas do CoAP 25
Figura 4 – Mensagens CoAP 26
Figura 5 – Implementação da arquitetura Web com HTTP e o CoAP 27
Figura 6 – Rede 6LoWPAN interligada a Internet 28
Figura 7 – Rede RPL com 2 DODAGs e 2 instâncias 29
Figura 8 – Arquitetura dos dispositivos de rede tradicional 31
Figura 9 – Arquitetura SDN 33
Figura 10 – Componentes de uma rede baseada em NOX 35
Figura 11 – Arquitetura do controlador OpenDaylight 36
Figura 12 – Visão arquitetural do controlador Ryu 38
Figura 13 – Modelo de programação de aplicações do Ryu 39
Figura 14 – Visão arquitetural do controlador Floodlight 40
Figura 15 – Controlador POX 41
Figura 16 – A estrutura geral do controlador Maestro 43
Figura 17 – Diagrama funcional reduzido do controlador Trema 45
Figura 18 – Visão geral da arquitetura do controlador Beacon 46
Figura 19 – Exemplo de uma rede de *switches* e roteadores comerciais com *OpenFlow* habilitado 49
Figura 20 – Arquitetura do emulador Mininet-WiFi 52
Figura 21 – Principais componentes do Mininet-WiFi 53
Figura 22 – Modelo de automatização dos experimentos 56
Figura 23 – Topologia utilizada 58
Figura 24 – Configuração da arquitetura MQTT 58
Figura 25 – Resultado do experimento de latência média 61
Figura 26 – Latência média 61
Figura 27 – Resultado do experimento de memória utilizada 62
Figura 28 – Memória utilizada 63
Figura 29 – Resultado do experimento do percentual de CPU utilizado 64
Figura 30 – Percentual de CPU utilizado 64

Figura 31 – Resultado do experimento do tempo de execução 65
Figura 32 – Tempo de execução 66
Figura 33 – Espaço de armazenamento interno da VM utilizando o comando fdisk . . . 74
Figura 34 – Espaço de armazenamento interno da VM utilizando o comando df 75
Figura 35 – Comando *ping6* 75
Figura 36 – Comando *free* 75
Figura 37 – Comando *time* 76
Figura 38 – Atualizando o sistema operacional 77
Figura 39 – Atualizando o *kernel* do sistema 78
Figura 40 – Comandos instalação do Ryu 79
Figura 41 – Comando instalação do POX 79
Figura 42 – Instalação Floodlight *master* 80
Figura 43 – Inicializando o controlador Floodlight 81
Figura 44 – Rodando a topologia no Mininet-WiFi com o Floodlight 81
Figura 45 – Teste do *ping* entre *sta1* e *sta2* 82
Figura 46 – Visualizando a troca de mensagens pelo *Wireshark* 82
Figura 47 – Inicializando o controlador POX 83
Figura 48 – Rodando a topologia no Mininet-WiFi com o controlador remoto POX . . . 83
Figura 49 – Conexão com o controlador efetuada 84
Figura 50 – Teste de conectividade entre *sta1* e *sta2* 84
Figura 51 – Troca de mensagens entre *sta1* e *sta2* vista no *Wireshark* 85
Figura 52 – Inicializando o controlador Ryu 86
Figura 53 – Rodando a topologia no Mininet-WiFi com o controlador remoto Ryu . . . 86
Figura 54 – Teste de Conectividade utilizando o controlador Ryu 87
Figura 55 – Visualização das conexões através da *web* 87
Figura 56 – Latência média com 26 nós IoT 88
Figura 57 – Latência média com 52 nós IoT 88
Figura 58 – Latência média com 104 nós IoT 89
Figura 59 – Memória utilizada com 26 nós IoT 90
Figura 60 – Memória utilizada com 52 nós IoT 90
Figura 61 – Memória utilizada com 104 nós IoT 91
Figura 62 – Percentual de CPU utilizado com 26 nós IoT 92

Figura 63 – Percentual de CPU utilizado com 52 nós IoT 92
Figura 64 – Percentual de CPU utilizado com 104 nós IoT 93
Figura 65 – Tempo de execução com 26 nós IoT 94
Figura 66 – Tempo de execução com 52 nós IoT 94
Figura 67 – Tempo de execução com 104 nós IoT 95

LISTA DE QUADROS

Quadro 1 – Comparação dos trabalhos relacionados 20
Quadro 2 – Controladores SDN 48
Quadro 3 – Fatores e níveis 55
Quadro 4 – Tempo de execução real dos experimentos 66
Quadro 5 – Comparação dos resultados 67

LISTA DE ABREVIATURAS E SIGLAS

6LoWPAN	*IPv6 over Low power Wireless Personal Area Networks*
ACK	*Aknowledgement*
API	*Application Programming Interface*
CBE	*Container-Based Emulation*
CoAP	*Constrained Application Protocol*
CPU	*Central Processing Unit*
DAG	*Direct Acyclic Graphs*
DAO ACK	*Destination Advertisement Object Acknowledgment*
DAO	*Destination Advertisement Object*
DIO	*DODAG Information Object*
DIS	*DODAG Information Solicitation*
DODAG	*Destination Oriented Acyclic Graph*
GUI	*Graphical User Interface*
HTTP	*Hypertext Transfer Protocol*
ICMP	*Internet Control Message Protocol*
ICMPv6	*Internet Control Message Protocol Version 6*
IETF	*Internet Engineering Task Force*
IETF ROLL	*Internet Engineering Task Force Routing Over Lowpower and Lossy networks*
IoT	*Internet of Things*
IP	*Internet Protocol*
LLNs	*Low-power and Lossy Networks*

M2M	*Machine-to-Machine*
MQTT	*Message Queuing Telemetry*
NETCONF	*Network Configuration Protocol*
NOS	*Network Operating System*
OF-CONFIG	*OpenFlow Managementand Configuration Protocol*
ONF	*Open Networking Fundation*
ONOS	*Open Network Operation System*
P&G	*Procter & Gamble*
PLC	*Power Line Communication*
RAM	*Random Access Memory*
REST	*Representational State Transfer*
RFID	*Radio-Frequency IDentification*
RPL	*IPv6 Routing Protocol for Low Power and Lossy Networks*
RTT	*Round Trip Time*
SDIoT	*Software Defined Internet of Things*
SDN	*Software Defined Networking*
SSH	*Secure SHell*
UDP	*User Datagram Protocol*
URI	*Uniform Resource Identifier*
VANETs	*Veicular Ad Hoc Networks*
VM	*Virtual Machine*

SUMÁRIO

1	**INTRODUÇÃO**	16
1.1	**Objetivos**	18
1.1.1	***Objetivo geral***	18
1.1.2	***Objetivos específicos***	18
2	**TRABALHOS RELACIONADOS**	19
3	**FUNDAMENTAÇÃO TEÓRICA**	21
3.1	**Internet das Coisas**	21
3.1.1	***Protocolos da camada de aplicação***	23
3.1.1.1	*MQTT*	23
3.1.1.2	*CoAP*	24
3.1.2	***Protocolos da camada de rede (encapsulamento e roteamento)***	27
3.1.2.1	*6LoWPAN*	28
3.1.2.2	*RPL*	29
3.2	**Redes Definidas por *Software***	31
3.2.1	***Controladores***	34
3.2.1.1	*NOX*	34
3.2.1.2	*OpenDaylight*	35
3.2.1.3	*Ryu*	36
3.2.1.4	*Floodlight*	39
3.2.1.5	*POX*	41
3.2.1.6	*Maestro*	43
3.2.1.7	*Trema*	44
3.2.1.8	*Beacon*	46
3.2.1.9	*Principais características dos controladores citados*	47
3.2.2	***OpenFlow***	48
3.2.3	***Mininet-WiFi***	51
3.2.3.1	*Arquitetura*	51
3.2.3.2	*Funcionamento*	52
4	**METODOLOGIA**	54
4.1	**Ambiente de trabalho**	54
4.2	**Automatização dos experimentos**	54

4.3	**Cenários**	57
5	**RESULTADOS E DISCUSSÃO**	60
5.1	**Latência média**	60
5.2	**Memória utilizada**	62
5.3	**Consumo de processador**	63
5.4	**Tempo de execução**	65
6	**CONSIDERAÇÕES FINAIS**	67
	REFERÊNCIAS	69
	APÊNDICE A – COMANDOS	74
	APÊNDICE B – INSTALAÇÃO DO MININET-WIFI	77
	APÊNDICE C – INSTALAÇÃO DOS CONTROLADORES SDN	79
	APÊNDICE D – TESTANDO O CONTROLADOR FLOODLIGHT	81
	APÊNDICE E – TESTANDO O CONTROLADOR POX	83
	APÊNDICE F – TESTANDO O CONTROLADOR RYU	86
	APÊNDICE G – LATÊNCIA MÉDIA	88
	APÊNDICE H – MEMÓRIA UTILIZADA	90
	APÊNDICE I – CONSUMO DE PROCESSADOR	92
	APÊNDICE J – TEMPO DE EXECUÇÃO	94

1 INTRODUÇÃO

De acordo com Borgia (2014), o termo Internet das Coisas, do inglês *Internet of Things* (IoT), foi introduzido a princípio por Kevin Ashton em 1999 durante uma apresentação realizada para a empresa *Procter & Gamble* (P&G). Desde então o termo IoT é empregado para caracterizar a possibilidade de conectar objetos comuns do cotidiano à Internet. Desse modo, IoT mostra-se um paradigma promissor, sobretudo ao integrar um grande número de objetos heterogêneos que apresentam meios de conexão distintos e capacidades computacionais variadas, com o propósito de permitir uma visão do mundo físico através da rede (NITTI et al., 2016).

A IoT viabiliza que objetos comuns do cotidiano realizem atividades como transferência de informações, processamento e análise de dados. Segundo Xu, He e Li (2014), o principal objetivo do paradigma IoT é a capacidade de conectar diferentes coisas (objetos inteligentes) através da rede, para que ocorra integração entre sistemas ou dispositivos heterogêneos.

Visto que o paradigma IoT propicia a integração de diversos objetos que possuem características distintas, tanto em relação à tecnologia de conexão à Internet como em relação à capacidade computacional, é esperado que a quantidade de objetos inteligentes IoT alcance a marca de 212 bilhões de entidades implantadas globalmente até o final de 2020 (AL-FUQAHA et al., 2015).

Ainda, de acordo com Santos et al. (2016), a IoT pode ser percebida como uma extensão da Internet atual, permitindo que objetos do dia-a-dia estejam conectados a ela. Por estarem conectados à Internet, tais objetos possibilitam atividades como transferir dados para outros objetos, controle remoto e monitorização de ambientes (SANTOS et al., 2016).

No entanto, Santos et al. (2016) afirmam que as redes IoT são formadas por objetos heterogêneos, que apresentam limitações tanto no que diz respeito à capacidade de memória como de processamento. Segundo Bedhief, Kassar e Aguili (2016) o desenvolvimento de redes e dispositivos IoT está impulsionando a evolução de aplicações como casas inteligentes (Salman et al., 2016a), cidades inteligentes, saúde inteligente, carros inteligentes (Ruengittinun; Paisalwongcharoen; Watcharajindasakul, 2017) e indústria. Por outro lado, a gestão e a otimização de desempenho da rede e a programabilidade de IoT tornaram-se complexas (BEDHIEF; KASSAR; AGUILI, 2016).

Por tudo isso, faz-se necessária uma nova abordagem para integrar redes IoT, tal como Redes Definidas por *Software*, do inglês *Software Defined Networking* (SDN), pois esta

arquitetura possibilita uma maior flexibilidade ao separar o plano de dados e o de controle (HU; HAO; BAO, 2014; XIA et al., 2015; NUNES et al., 2014), sendo o controlador SDN o responsável pela centralização do plano de controle.

O paradigma SDN surgiu em virtude do alto grau de complexidade da arquitetura de rede tradicional. Segundo Karakus e Durresi (2017), em arquiteturas de redes tradicionais a reconfiguração ou reinstalação de equipamentos ou aplicações de rede só podem ser realizadas fisicamente. Para isso, é necessária mão de obra altamente qualificada. Em SDN, há um desacoplamento do plano de dados e do de controle permitindo assim que os administradores de rede facilmente mudem as suas características já que há programabilidade e configurabilidade (KARAKUS; DURRESI, 2017). O elemento central da rede SDN é o controlador, também denominado de Sistema Operacional de Rede, do inglês *Network Operating System* (NOS) (CARAGUAY et al., 2014). Através dele, a rede é logicamente centralizada, servindo como a ponte de comunicação entre o plano de gerenciamento e o plano de dados, o que possibilita uma visão global da rede, além de flexibilidade, pois as políticas da rede podem ser facilmente atualizadas, adicionadas ou removidas.

Tendo em vista a importância do paradigma SDN, em especial o controlador, e também os desafios inerentes de redes IoT, este trabalho consiste em realizar uma comparação entre os controladores SDN que dão suporte a cenários IoT. Levando em conta as limitações apresentadas pelos dispositivos IoT no que se refere a capacidade de memória, ao poder de processamento e ao consumo energético, ademais, apresenta heterogeneidade de tecnologias de conexão. Considerando tais limitações do paradigma IoT, o objetivo central do trabalho consiste em comparar os controladores SDN em cenários IoT, e a partir disso, elencar qual controlador sobressai em cada métrica de interesse, tendo como foco as seguintes métricas: latência; memória; consumo de processador; e tempo de execução dos experimentos (em cada cenário).

É esperado que este trabalho possa ser utilizado por pesquisadores ou estudantes da área no processo de seleção do controlador SDN a ser usado em estudos ou testes em ambientes de IoT. Isto facilitará a tomada de decisão em relação a qual controlador utilizar para realizar o estudo ou teste em cenários ou ambientes de IoT.

O presente trabalho está organizado da seguinte forma: na Seção 1.1 encontram-se os objetivos geral e específicos. O objetivo geral norteia a questão de pesquisa e os objetivos específicos pontuam questões a serem elucidadas para se chegar à resposta da questão de pesquisa do objetivo geral. O Capítulo 2 aborda os trabalhos relacionados, tendo assim o intuito

de descrever alguns trabalhos que tratam dos temas chave deste trabalho e também descreve-se o que é similar ou não ao atual trabalho. No Capítulo 3 encontra-se a fundamentação teórica, nela os temas chave do vigente trabalho são explicados. Ao fazer isso, aborda-se sobre os conceitos relacionados a IoT e sobre os conceitos relacionados a SDN. A metodologia do trabalho é apresentada no Capítulo 4, este Capítulo descreve o caminho percorrido para a obtenção das métricas de interesse definidas para o trabalho. No Capítulo 5 apresenta-se os resultados e a discussão, os resultados são apresentados em forma de gráficos, e faz-se a discussão em cima destes gráficos. Por fim, o Capítulo 6 apresenta as considerações finais e os trabalhos futuros.

1.1 Objetivos

Esta Seção aborda o objetivo norteador do trabalho na Subseção 1.1.1. A Subseção em questão é a responsável por delimitar o escopo do trabalho. Por fim, os objetivos específicos são contemplados na Subseção 1.1.2, os objetivos específicos trazem uma série de pontos a serem resolvidos. Tais pontos, quando resolvidos, esclarecem a questão de pesquisa apresentada no objetivo geral.

1.1.1 Objetivo geral

O objetivo do presente trabalho consiste em comparar os controladores de Redes Definidas por Software (SDN): Ryu; POX e Floodlight; em cenários de Internet das Coisas (IoT), a fim de descobrir qual dos controladores apresenta os melhores resultados em relação a latência, memória, consumo de processador e tempo de execução dos experimentos.

1.1.2 Objetivos específicos

- Fazer levantamento sobre os controladores SDN;
- Indicar qual controlador possui menor latência média;
- Indicar qual controlador apresenta os melhores valores para memória utilizada;
- Indicar qual controlador apresenta menor consumo de processador;
- Indicar qual controlador apresenta menor tempo de execução dos experimentos;

2 TRABALHOS RELACIONADOS

Neste Capítulo são apresentados alguns trabalhos que abordam os temas chave do vigente trabalho. Nos parágrafos subsequentes serão expostos trabalhos com foco no tema IoT e também trabalhos com foco tanto em IoT quanto em SDN, sempre fazendo-se um contraponto com o que é semelhante ou diferente em relação ao presente trabalho.

O trabalho de Bondkovskii et al. (2016) faz uma comparação qualitativa entre dois controladores SDN de código aberto, o OpenDaylight[1] e o *Open Network Operation System*[2] (ONOS). O estudo foca na interface *Northbound* destes dispositivos, em que tal interface foi configurada para realizar espelhamento de porta. O presente trabalho também realiza uma comparação entre controladores SDN, dessa forma foram escolhidos controladores SDN de código fonte aberto a fim de realizar a comparação dos mesmos em redes IoT. Porém, não usou-se o tráfego de acesso dos controladores nas interfaces *Nortbound* espelhadas como critério de avaliação, pois o foco principal consiste na análise dos controladores SDN que se encontram em cenários de IoT. As métricas utilizadas neste trabalho são latência, memória, consumo de processador e tempo de execução dos experimentos.

O trabalho de Rastogi e Bais (2016), realiza uma análise comparativa em termos da capacidade do tráfego. O objetivo central do trabalho consiste em apresentar uma análise entre dois controladores denominados Pox[3] e Ryu[4] respectivamente, em termos da capacidade de manuseio de tráfego. O emulador Mininet[5] foi utilizado para emular o ambiente dos controladores SDN e assim monitorar o desempenho do tráfego. Assim como no trabalho de Rastogi e Bais (2016), este trabalho realiza uma análise comparativa entre controladores SDN, no entanto o presente trabalho compara controladores SDN que se encontram em uma rede IoT. Neste trabalho utilizou-se o emulador Mininet-WiFi que possibilita a emulação de uma rede IoT.

O trabalho de Jararweh et al. (2015), propõe um *framework* para IoT utilizando o paradigma SDN. Este *framework* foi então denominado de *Software Defined Internet of Things* (SDIoT). Tem o propósito de testar diferentes formas e tipos de topologias IoT, e também o intuito de poder acomodar grandes fluxos de dados provenientes da rede IoT. Diferentemente do trabalho de Jararweh et al. (2015), o presente trabalho não tem a finalidade de propor um

[1] https://www.opendaylight.org/
[2] http://onosproject.org/
[3] https://github.com/noxrepo/pox
[4] https://osrg.github.io/ryu/
[5] http://mininet.org/walkthrough/

framework para IoT. Porém, este trabalho, assim como o trabalho de Jararweh et al. (2015) utilizou do paradigma SDN para integrar a rede IoT aos controladores SDN e assim ser possível desenvolver uma análise comparativa dos controladores em cenários IoT.

O trabalho de Wu et al. (2015), apresenta um sistema para controle de fluxo de dados ubíquos e gerenciamento de mobilidade em redes heterogêneas urbanas utilizando o paradigma de SDN em conjunto com o paradigma IoT, o sistema em questão foi nomeado de *UbiFlow*. Igualmente ao trabalho de Wu et al. (2015), este trabalho utiliza do paradigma SDN em conjunto com o paradigma IoT, no entanto não foi desenvolvido um sistema para gerenciamento do fluxo de dados ubíquos pois o foco deste trabalho reside em realizar uma comparação entre controladores SDN em cenários de IoT.

Na bibliografia estudada foram vistas comparações entre controladores de código fonte aberto com foco na interface *Northbound* que foi espelhada e também uma análise comparativa em termos da capacidade do tráfego. Tais testes não utilizaram cenários de IoT integrados aos respectivos controladores SDN. Ainda foi mostrado um *framework* denominado SDIoT cujo o propósito era o teste de topologias variadas de IoT utilizando a tecnologia SDN com o intuito de acomodar grandes fluxos de dados e também foi exposto um sistema para controle de fluxo de dados ubíquos e gerenciamento de mobilidade em redes heterogêneas urbanas de nome *UbiFlow* que também integrava SDN e IoT. No entanto, estes não tinham foco em realizar comparações dos controladores SDN. Com isso, o presente trabalho visa realizar uma comparação entre controladores SDN em cenários de IoT e desta forma determinar qual controlador melhor se adequa aos cenários IoT. O Quadro 1 apresenta uma comparação dos trabalhos relacionados em relação ao vigente trabalho, nela consta o **Autor**, se o trabalho fazia uso de **IoT**, se o trabalho empregava **SDN** e se o trabalho fazia **Comparação de controladores SDN em IoT**.

Quadro 1 – Comparação dos trabalhos relacionados

Autor	**IoT**	**SDN**	**Comparação de controladores SDN em IoT**
Bondkovskii et al. (2016)	Não	Sim	Não
Rastogi e Bais (2016)	Não	Sim	Não
Jararweh et al. (2015)	Sim	Sim	Não
Wu et al. (2015)	Sim	Sim	Não
Próprio autor (2019)	Sim	Sim	Sim

Fonte: Próprio autor (2019).

3 FUNDAMENTAÇÃO TEÓRICA

Neste Capítulo apresenta-se a explanação dos temas chave do atual trabalho. A Seção 3.1 apresenta os conceitos relacionados a IoT, dessa maneira, aborda-se sobre os seguintes assuntos: blocos formadores de IoT; componentes arquiteturais; protocolos da camada de aplicação; e protocolos da camada de rede. Por fim, na Seção 3.2 é apresentada a conceituação de SDN assim como suas principais características, outros assuntos importantes que são fundamentados são: os controladores SDN; o protocolo *OpenFlow*; e o emulador Mininet-WiFi.

3.1 Internet das Coisas

De acordo com Borgia (2014), o termo IoT, foi introduzido a princípio por Kevin Ashton em 1999 durante uma apresentação realizada para a empresa P&G. Esta conotação foi empregada por Kevin Ashton, para descrição da internet conectada ao mundo físico, através, de sensores onipresentes e através de uma plataforma baseada em *feedbacks* em tempo real, com enorme capacidade para melhorar o conforto, a segurança e a qualidade de vida (BORGIA, 2014)

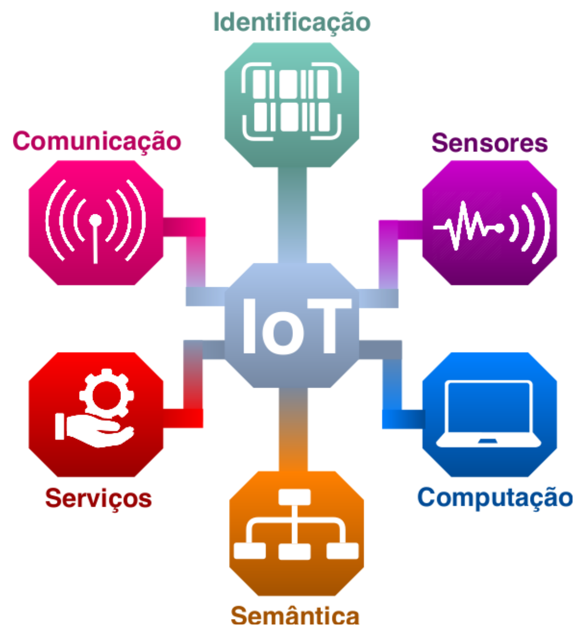
A IoT, como o próprio nome já é bem sugestivo, possibilita que objetos heterogêneos estejam conectados à Internet. De acordo com Islam et al. (2015), qualquer pessoa pode estar conectada a qualquer coisa (objeto) a qualquer momento em qualquer rede. A IoT é constituída por um grande número de tecnologias a fim de integrar estes objetos ao ambiente físico. A Figura 1 apresenta seis blocos básicos definidos para sua construção.

Dessa forma, segundo Xu, He e Li (2014), em IoT pode-se ter variados dispositivos, e tais dispositivos podem utilizar diferentes tecnologias de comunicação, rede, capacidade de armazenamento de dados, capacidade de processamento e potência de transmissão.

Segundo Santos et al. (2016), são seis os blocos formadores de IoT. São eles: identificação; sensores ou atuadores; comunicação; computação; serviços; e semântica. O bloco de identificação é usado para identificar os demais objetos IoT e posteriormente prover comunicação entre eles. O bloco de sensores ou atuadores é responsável por coletar as informações pertinentes que podem ser processadas ou utilizadas para reagir de acordo com os dados coletados. Ainda o bloco de comunicação possibilita que objetos inteligentes se conectem e troquem informações entre si. Já o bloco de computação realiza o processamento dos dados, este bloco é quem dá a inteligencia dos objetos IoT. O bloco de serviços provê diversas classes de serviços como comunicação e identificação. Por fim, o bloco de semântica trata da descoberta

e uso inteligente dos recursos.

Figura 1 – Blocos de construção IoT



Fonte: Adaptado de Santos et al. (2016).

De acordo com Tayyaba et al. (2016), os componentes arquiteturais de uma rede IoT são divididos em quatro camadas. São elas: camada de percepção; camada de rede; camada de aplicação; e camada de *middleware*.

- **Camada de percepção**: é a camada formada pelos dispositivos físicos. Constituída por sensores, Identificação por Radiofrequência, do inglês *Radio-Frequency IDentification* (RFID) e dispositivos móveis. Esta camada é responsável pela coleta dos dados do ambiente onde estes dispositivos estão inseridos. Após a coleta, os dados podem ser transmitidos para a borda da rede ou para outro dispositivo.
- **Camada de rede**: fica responsável pela transmissão dos dados coletados provenientes dos objetos físicos. Estes dados são enviados para a borda da rede para posterior processamentos da informação coletada. Neste contexto existem diferentes tecnologias que podem ser utilizadas para a transmissão dos dados, como *ZigBee*, *bluetooth* e Wi-Fi. Essa variedade de tecnologias de transmissão de dados contribui para a heterogeneidade do paradigma IoT.
- **Camada de aplicação**: lida com os serviços ou aplicações dependendo da

demanda do usuário e de acordo com a manipulação dos dados obtidos da camada de percepção que foram posteriormente processados.

- **Camada de *middleware***: é encarregada de realizar a tradução das mensagens informativas de um serviço, sem a necessidade de conhecimento detalhado do *hardware* dos dispositivos que estão trocando as mensagens. A camada de *middleware* está associada com o gerenciamento dos serviços, endereçamento e nomeação dos serviços requisitados pelos dispositivos heterogêneos.

3.1.1 Protocolos da camada de aplicação

Esta Seção aborda alguns protocolos da camada de aplicação que são utilizados no âmbito de IoT. As Subseções estão organizadas da seguinte forma: a Subseção 3.1.1.1 trata de descrever o protocolo MQTT, mostrando sua arquitetura e o modelo de funcionamento empregado por tal protocolo; por fim, a Subseção 3.1.1.2 tem o intuito de abordar sobre o protocolo CoAP, descrevendo o modelo utilizado e as mensagens que este protocolo suporta.

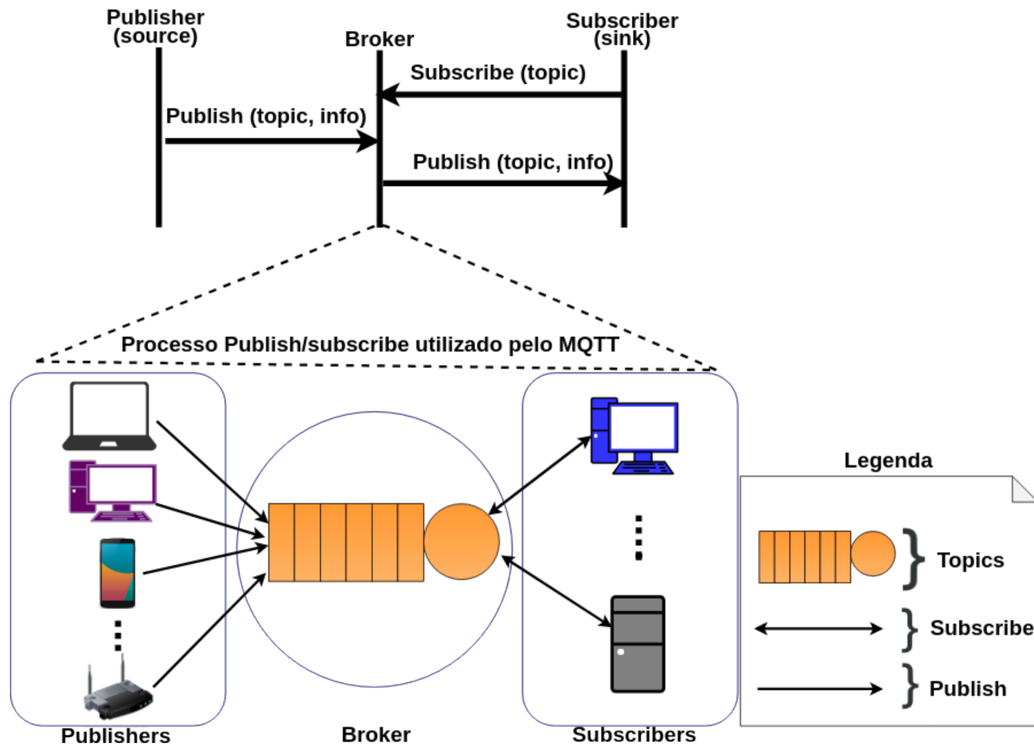
3.1.1.1 MQTT

O protocolo MQTT, do inglês *Message Queuing Telemetry*, é um protocolo destinado a troca de mensagens, que foi apresentado por Andy Stanford-Clark da IBM e Arlen Nipper da Arcom (ANTHRAPER; KOTAK, 2019), no ano de 1999 e que foi posteriormente padronizado no ano de 2013 pela OASIS (Wukkadada et al., 2018). O objetivo do protocolo MQTT é o de conectar dispositivos embarcados e redes com aplicações e *middlewares* (AL-FUQAHA et al., 2015). Para o processo de conexão utiliza-se mecanismos de roteamento *One-to-One* (Um-para-Um), *One-to-Many* (Um-para-Muitos) e *Many-to-Many* (Muitos-para-Muitos) tornando o MQTT um protocolo viável para IoT e para M2M (AL-FUQAHA et al., 2015).

O protocolo MQTT utiliza o modelo *publish/subscribe*, para Al-Fuqaha et al. (2015), este modelo fornece flexibilidade de transição e simplicidade de implementação. A arquitetura do protocolo MQTT apresenta três componentes (YASSEIN et al., 2017), assim como visto na Figura 2. O ***publisher*** é um editor, o ***broker*** é um corretor ou intermediário e o ***subscriber*** é um assinante. Quando um dispositivo está interessado em um determinado tópico registra-se como um assinante para aquele tópico, para desta forma ser informado quando os editores publicam seus tópicos de interesse pelo intermediário. O editor trabalha como um gerador de dados de interesse. Por fim, o editor transfere as informações para os assinantes (partes de interesse)

através do intermediário.

Figura 2 – Arquitetura do protocolo MQTT



Fonte: Adaptado de Al-Fuqaha et al. (2015), Yassein et al. (2017).

Neste trabalho utilizou-se do protocolo MQTT. Sendo o MQTT um dos principais protocolos da camada de aplicação utilizados em IoT. A escolha do MQTT Teve como principal objetivo permitir que os nós IoT trocassem informações entre si. Com o intuito de gerar muito tráfego de aplicação nos cenários utilizados, a fim de tornar os cenários mais realísticos.

3.1.1.2 CoAP

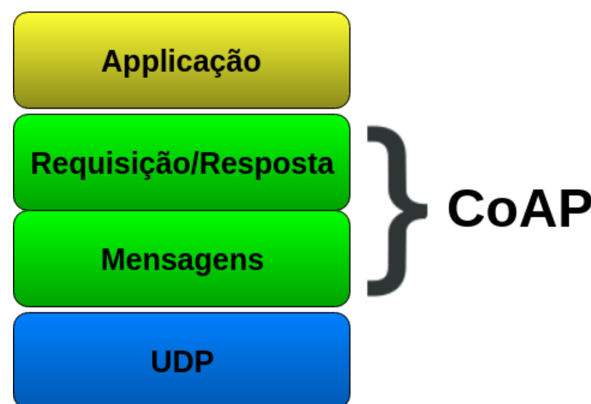
O Protocolo de Aplicação Restrita, do inglês *Constrained Application Protocol* (CoAP), foi criado pela Força-Tarefa de Engenharia da Internet, do inglês *Internet Engineering Task Force* (IETF) (KARAGIANNIS et al., 2015). É categorizado como um protocolo de transferência da *Web* utilizado por dispositivos com recursos limitados, isto é, com baixa capacidade energética e que apresentam elevada perda de pacotes (SHELBY; HARTKE, 2014; IGLESIAS-URKIA; ORIVE; URBIETA, 2017).

Conforme ressalta Frigieri, Mazzer e Parreira (2015), o CoAP é um protocolo da camada de aplicação baseado no Protocolo de Transferência de Hipertexto, do inglês *Hypertext*

Transfer Protocol (HTTP), foi otimizado com foco em dispositivos que apresentam potência e capacidade de processamento limitados e em geral é aplicado a objetos inteligentes em ambientes de IoT. O CoAP trabalha com o protocolo de transporte UDP, do inglês *User Datagram Protocol*, e especifica um conjunto mínimo de requisições de Transferência de Estado Representacional, do inglês *Representational State Transfer* (REST), que inclui os métodos HTTP: POST; GET; PUT; e DELETE. Ainda mantém suporte para o armazenamento e a descoberta de recursos internos.

A Figura 3 ilustra a abstração das camadas presentes no protocolo CoAP. Destaca-se o modelo utilizado pelo protocolo CoAP que é o de Requisição/Resposta. Ainda na Figura 3, visualiza-se duas camadas distintas que constituem o protocolo CoAP são elas: Mensagens; e Requisição/Resposta. A camada de mensagens lida com o UDP e com mensagens assíncronas. A camada Requisição/Resposta fica encarregada de gerenciar a interação das Requisições/Respostas baseando-se nas mensagem advindas da camada de Aplicação (AZZOLA, 2018).

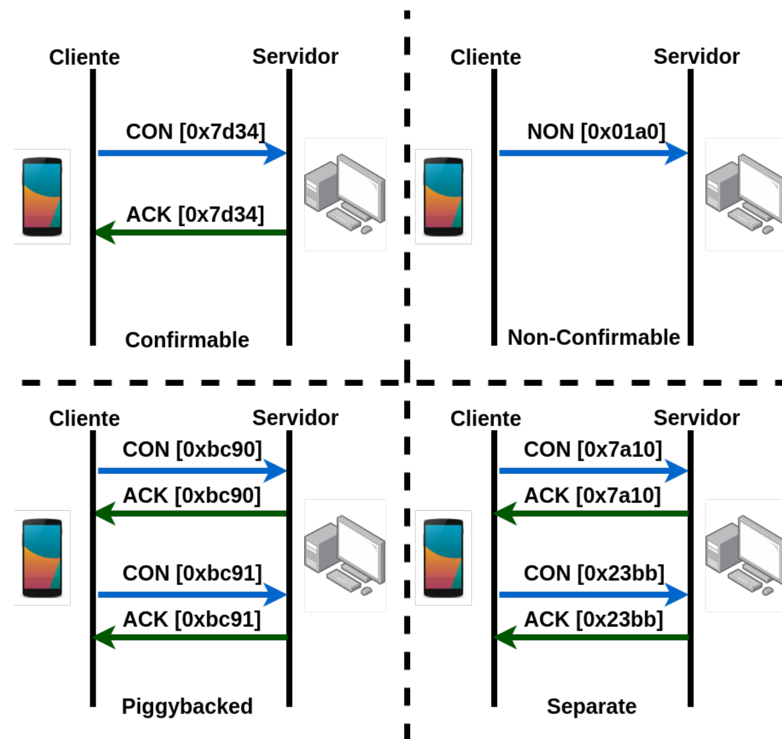
Figura 3 – Abstração das camadas do CoAP



Fonte: Adaptado de Azzola (2018).

Desta forma os dispositivos podem atuar como clientes ou como servidores e os recursos podem ser acessados por meio de Identificador Uniforme de Recursos, do inglês *Uniform Resource Identifier* (URI). No entanto, ao contrário do que ocorre no HTTP a conexão não é estabelecida antes da troca de mensagens. O processo de comunicação ocorre de maneira assíncrona (Bormann; Castellani; Shelby, 2012; FRIGIERI; MAZZER; PARREIRA, 2015).

Figura 4 – Mensagens CoAP



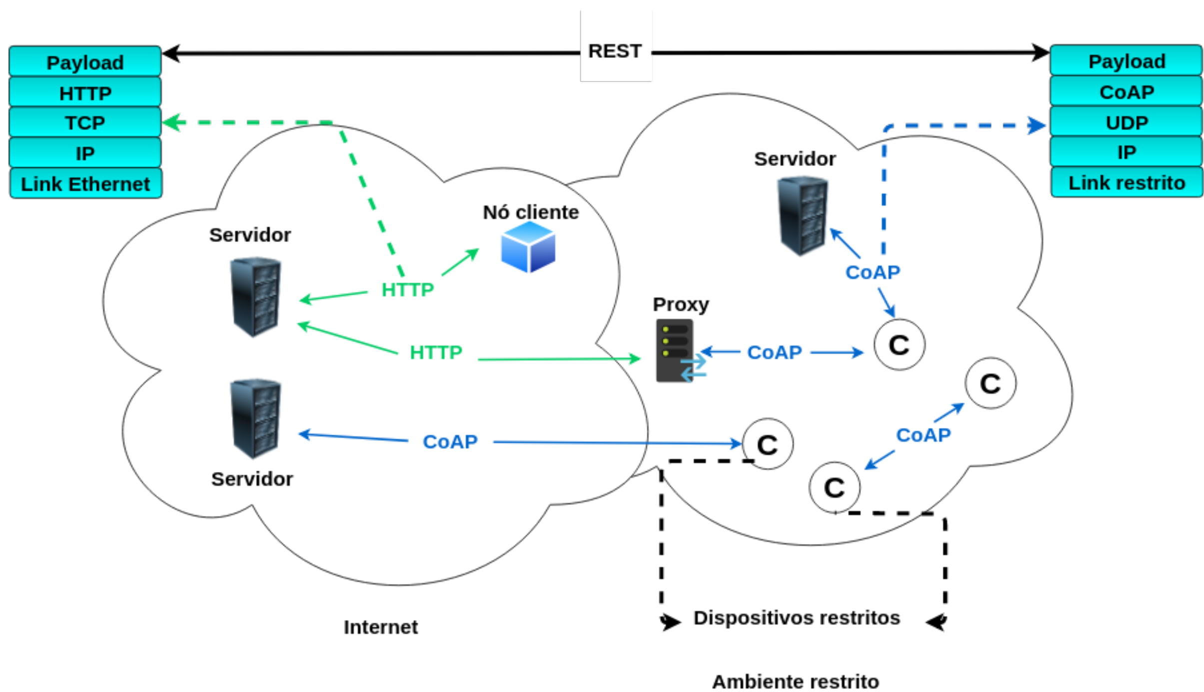
Fonte: Adaptado de Salman e Jain (2019), Al-Fuqaha et al. (2015).

A Figura 4 mostra alguns tipos de mensagens que o CoAP possui, a seguir os tipos de mensagens do protocolo CoAP são descritas levando em conta os trabalhos (ABDELFADEEL, 2016; Bormann; Castellani; Shelby, 2012):

1. ***Confirmable***: a mensagem enviada (***Send***) deve ser confirmada pelo receptor através de um **ACK**, do inglês ***Aknowledgement***.
2. ***Non-confirmable***: Podem ser enviadas mensagens que não exigem confirmação.
3. ***Acknowledgment***: as mensagens de confirmação são retransmitidas com limites de tempo exponenciais, até serem reconhecidas pelo receptor ou atingirem um número máximo de retransmissões.
4. ***Reset***: é indicativo de que uma mensagem do tipo *Confirmable* ou do tipo *Non-Confirmable* foi recebida, porém o receptor não consegue processa lá adequadamente.
5. ***Piggybacked***: uma resposta está inclusa em uma mensagem de confirmação ACK.
6. ***Separate***: quando uma mensagem do tipo *Confirmable* transporta uma solicitação e é confirmada com uma mensagem vazia, uma resposta *Separate* é enviada em uma troca de mensagens separada.
7. ***Empty***: mensagem com código 0,00. Não é nem um pedido e nem uma resposta, é uma

mensagem vazia contendo apenas um cabeçalho de 4 bytes.

Figura 5 – Implementação da arquitetura Web com HTTP e o CoAP



Fonte: Adaptado de Bormann, Castellani e Shelby (2012), Nascimento (2018).

A Figura 5 mostra a implementação da arquitetura da Web com HTTP e o CoAP. Os protocolos HTTP e CoAP trabalham juntos em ambientes restritos e em ambientes tradicionais da Internet. Também mostra-se a pilha do protocolo CoAP que é similar a pilha do protocolo HTTP, mas menos complexa. A Figura 5, em questão, também é composta por servidores e por dispositivos restritos, isto é, com capacidade limitada e por um *proxy*. O *proxy* é basicamente um intermediário entre o cliente é o servidor. Ficando responsável essencialmente em receber as solicitações dos clientes e encaminhá-las e também de receber as respostas das solicitações e encaminhá-las ao cliente adequado (Bormann; Castellani; Shelby, 2012).

Este trabalho não utilizou o protocolo CoAP. Pelo pouco tempo disponível para testar a viabilidade deste protocolo no emulador Mininet-WiFi. No entanto, para trabalhos futuros seria uma boa ideia adicionar o tráfego de aplicação CoAP nos cenários IoT, caso tenha viabilidade.

3.1.2 Protocolos da camada de rede (encapsulamento e roteamento)

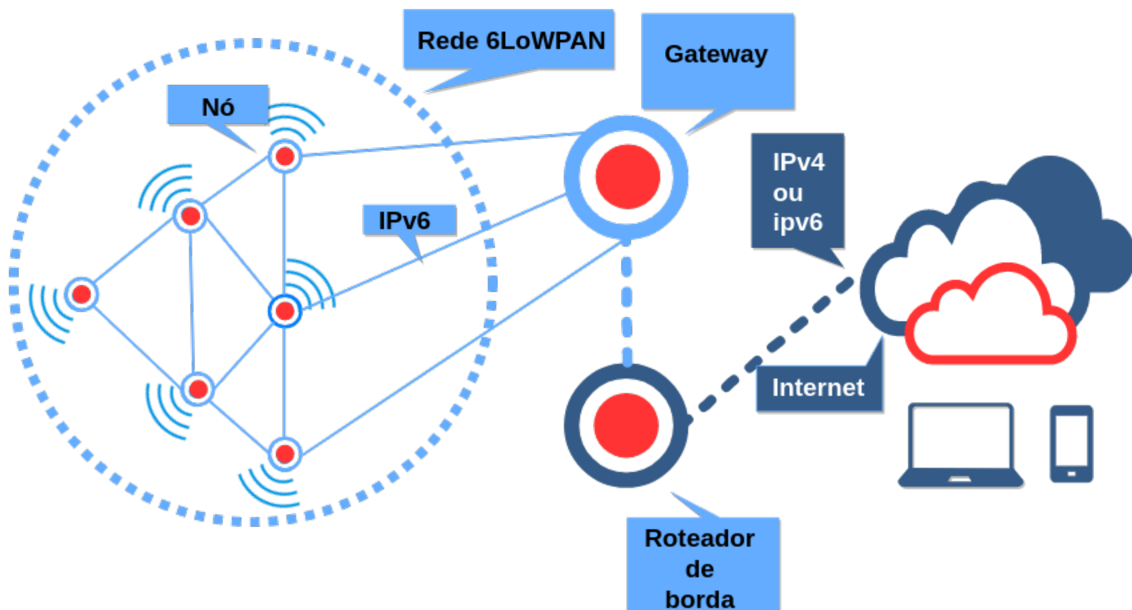
Nesta Seção, aborda-se sobre os protocolos da camada de rede, incluindo encapsulamento e roteamento. As Subseções estão organizadas da seguinte forma: a Subseção

3.1.2.1, que está ligada a camada de encapsulamento, fala sobre o 6LoWPAN; já a Subseção 3.1.2.2 explica o funcionamento do protocolo de roteamento RPL.

3.1.2.1 6LoWPAN

O 6LoWPAN, do inglês IPv6 *over Low power Wireless Personal Area Networks* é uma camada de adaptação, para redes de área pessoal sem fio de baixa potência. Tal concepção segundo Mulligan (2007), nasceu da ideia de que o Protocolo da Internet, do inglês *Internet Protocol* (IP), poderia e deveria ser aplicado até mesmo nos menores dispositivos. Permitindo, assim, que pacotes IPv6 fossem transportados em pequenos quadros sobre enlaces IEEE 802.15.4. Um dos requisitos da IoT é o uso de arquitetura flexível em camadas que pode fornecer interconexão a objetos heterogêneos (Lamkimel et al., 2018).

Figura 6 – Rede 6LoWPAN interligada a Internet



Fonte: Adaptado de Zolertia (2019), Lamkimel et al. (2018).

A Figura 6 expõe uma rede 6LoWPAN interligada à Internet. A rede 6LoWPAN é constituída por nós. O 6LoWPAN remove muitas sobrecargas do IPv6 viabilizando que os nós enviem pacotes IPv6. Ainda, os nós conectam-se a um *Gateway* que por sua vez está interligado ao roteador de barda da rede. O roteador de borda da rede conecta-se à Internet. Permitindo a troca de informações dos nós 6LoWPAN com a rede externa.

Neste trabalho se utilizou do 6LoWPAN, visto que as redes de sensores apresentam

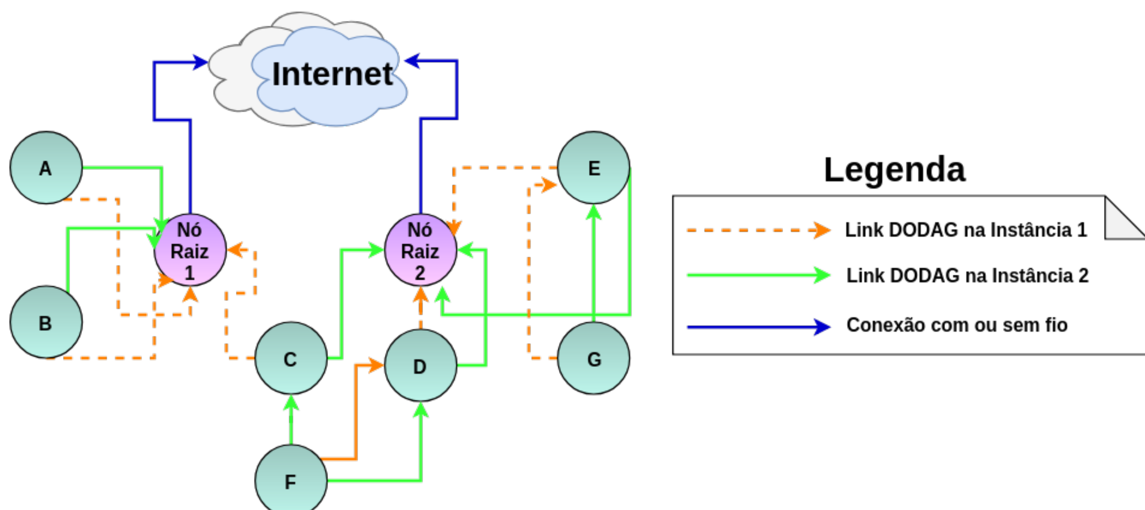
limitações energéticas, baixo alcance e baixas taxas de transmissão (FONTES; ROTHENBERG, 2019). O suporte ao 6LoWPAN através do emulador Mininet-WiFi é possível por meio do módulo denominado de *fakelb*[1], possibilitando a criação de redes IEEE 802.15.4 (FONTES; ROTHENBERG, 2019).

3.1.2.2 RPL

As LLNs, do inglês *Low-power and Lossy Networks*, consistem em sua maioria de nós que apresentam poder de processamento limitado, baixa capacidade de memória e também pouca capacidade energética (WINTER et al., 2012). Os roteadores deste tipo de rede são conectados por *links* com perdas, que geralmente suportam baixa taxa de dados. A partir disso, o grupo de trabalho IETF ROLL, do inglês *Internet Engineering Task Force Routing Over Low power and Lossy networks*, definiu os requisitos para o protocolo RPL, do inglês IPv6 *Routing Protocol for Low-Power and Lossy Networks* (WINTER et al., 2012).

O protocolo RPL foi projetado para operar em uma variedade de tecnologias de camada de enlace, como redes sem fio de baixa potência ou PLC, do inglês *Power-Line Communication*. Este protocolo foi aprimorado até mesmo para ser usado em Redes Ad Hoc Veiculares, do inglês *Veicular Ad Hoc Networks* (VANETs) (IOVA, 2014).

Figura 7 – Rede RPL com 2 DODAGs e 2 instâncias



Fonte: Adaptado de Iova (2014).

O RPL é um protocolo do tipo vetor de distância, baseado em Grafo Acíclico

[1] https://github.com/torvalds/linux/blob/master/drivers/net/ieee802154/fakelb.c

Direcionado, do inglês *Direct Acyclic Graphs* (DAG) (KIM et al., 2017). Foi otimizado para ser capaz de coletar dados de diversos sensores, criando assim um Grafo Acíclico Direcionado Orientado ao Destino, do inglês *Destination-Oriented Acyclic Graph* (DODAG), no nó raiz ou roteador de borda. A rede pode conter várias instâncias do protocolo RPL, e essas instâncias podem conter vários DODAG, sendo um DODAG para cada nó raiz ou roteador de borda. É importante esclarecer que de acordo com Braga et al. (2018), em uma instância específica, um nó só pode pertencer a um DODAG. A Figura 7, apresenta uma rede RPL na qual é formada por 2 DODAGs e 2 instâncias.

Ainda, segundo Iova (2014), Braga et al. (2018), o protocolo RPL utiliza quatro tipos de mensagens de controle usando o ICMPv6, do inglês *Internet Control Message Protocol Version 6*, com o objetivo de manter e atualizar as rotas, são as seguintes:

- DODAG *Information Object* (DIO): é o tipo de mensagem de controle que contém informações sobre o DODAG. Informações como, o identificador do DODAG, qual a instância em que se encontra o DODAG ou ainda as métricas utilizadas para calcular a rota. As mensagens são enviadas por *broadcast*. O nó raiz ou roteador de borda é o único que pode iniciar o envio desta mensagem de controle.
- DODAG *Information Solicitation* (DIS): o objetivo deste tipo de mensagem é solicitar informações de configuração. Partindo disso, um nó pode enviar esse tipo de mensagem de forma *unicast* ou *multicast*. O retorno desta mensagem será um pacote DIO.
- *Destination Advertisement Object* (DAO): o DAO é a única mensagem de controle que pode ser reconhecida pelo destinatário. No entanto, as mensagens DAO são opcionais e somente utilizadas quando rotas descendentes são necessárias, como por exemplo em aplicações que requerem tráfego ponto-multiponto ou ainda ponto-a-ponto. Esse tipo de mensagem é enviada de modo *unicast* aos vizinhos ou ao nó raiz.
- *Destination Advertisement Object Acknowledgment* (DAO-ACK): esse tipo de mensagem indica se os vizinhos que anteriormente receberam o DAO querem porta-se como um salto (hop) na rota descendente, para o nó que está enviando a mensagem.

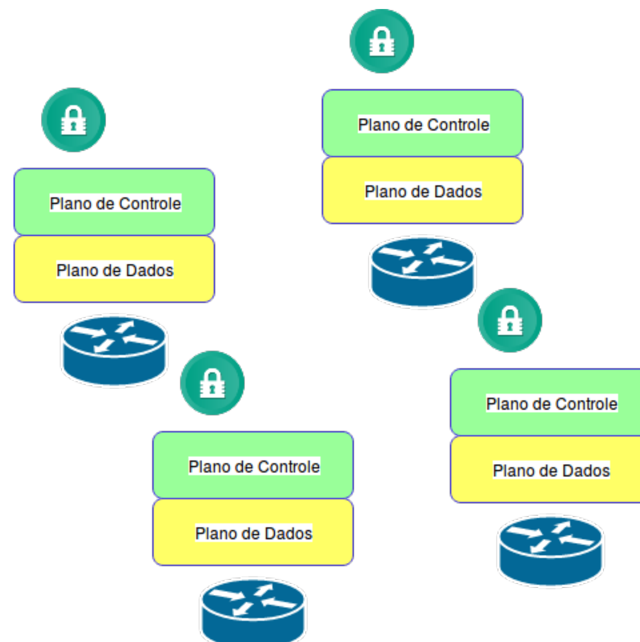
Neste trabalho não foi usado o protocolo RPL. Pelo pouco tempo disponível para testar se era possível a implementação deste protocolo nos roteadores do emulador Mininet-WiFi. Dito isto, seria muito interessante utilizar de nós habilitados com o protocolo RPL em trabalhos futuros.

3.2 Redes Definidas por *Software*

O paradigma SDN, fornece a habilidade de modificação do comportamento da rede dinamicamente, dependendo da necessidade do usuário (CARAGUAY et al., 2014). As características inerentes de SDN trazem benefícios significativos para o paradigma IoT. Segundo Sood, Yu e Xiang (2016), a integração de SDN e IoT trazem grandes benefícios, pois SDN tem potencial para rotear de forma inteligente o tráfego de dados e ainda utilizar de forma proveitosa os recursos de rede subutilizados. Aumentando assim a capacidade da rede, e portanto facilitando a preparação das redes para o grande volume de dados de IoT. A integração de SDN com IoT simplificará a coleta de informações, a análise das informações e a tomada de decisão em IoT (SOOD; YU; XIANG, 2016).

A dinamicidade do comportamento da rede é possível, segundo Nunes et al. (2014), pela ideia de "redes programáveis", proporcionando assim um meio de facilitar a evolução da rede e ainda simplificar consideravelmente o gerenciamento da mesma. Em contraste, a arquitetura de rede convencional é distribuída, e os elementos que a constituem são proprietários. A Figura 8 ilustra a arquitetura dos dispositivos de rede tradicionais, mostrando como o plano de controle e o de dados estão organizados em tais dispositivos.

Figura 8 – Arquitetura dos dispositivos de rede tradicional



Fonte: Adaptado de Caraguay et al. (2014).

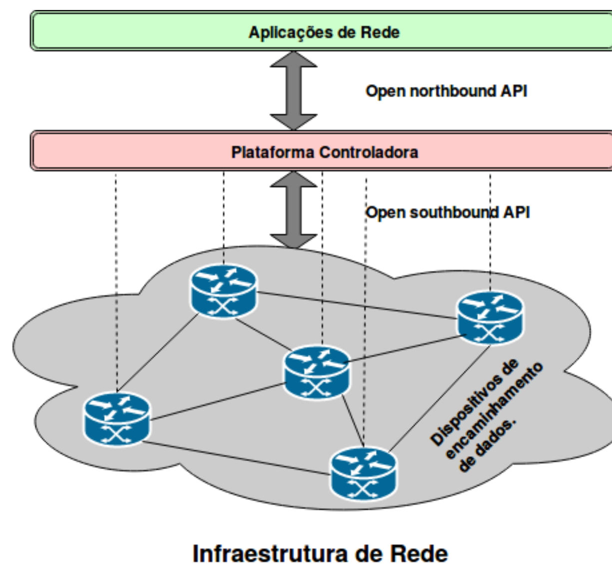
Em arquiteturas de rede convencionais, o plano de dados e o plano de controle encontram-se embarcados nos dispositivos de rede. Desta maneira, tais equipamentos são responsáveis por encaminhar os pacotes, além de possuírem um *software* de controle que realiza o processo de tomada de decisão, gerando assim tabelas de roteamento. Isso caracteriza uma estrutura descentralizada. Estes dispositivos são de domínio proprietário (código fonte fechado), o que dificulta o teste e implementação de novas ideias em arquiteturas convencionais. Em contrapartida a arquitetura SDN centraliza o plano de controle, através do controlador.

SDN é uma nova abordagem que veio em resposta às limitações impostas pela arquitetura de rede tradicional. Essa arquitetura convencional tornou-se "ossificada", o que significa que é muito complexo e caro criar novas aplicações para a rede. Isso se dá porque os dispositivos que constituem uma rede tradicional são de posse proprietária e de código fonte fechado. Nestas redes, para ser adicionada uma nova aplicação, é necessário um longo processo engessado de desenvolvimento e testes para só então ser adicionada a nova aplicação no dispositivo de rede pretendido (KARAKUS; DURRESI, 2017).

Em resposta ao modelo anteriormente citado, as redes SDN permitem uma maior flexibilidade e configurabilidade, uma vez que sua arquitetura é composta por três planos: plano de gerenciamento; de controle; e de dados.

Neste novo modelo, o plano de encaminhamento de dados passou a ser desacoplado do plano de controle. Com isso, as redes SDN possibilitam um controle centralizado da rede e consequentemente uma visão global da mesma (KREUTZ et al., 2015). Tal modelo difere da arquitetura tradicional, em que o plano de dados e de controle estão presentes no mesmo dispositivo, e dessa forma somente possibilitando uma visão local da rede. A Figura 9 apresenta a arquitetura de uma rede SDN composta pelo plano de gerenciamento, pelo plano de dados, pelo plano de controle.

Figura 9 – Arquitetura SDN



Fonte: Adaptado de Kreutz et al. (2015).

Pode-se perceber na Figura 9 que existem duas interfaces de comunicação, uma intitulada *northbound* e outra de nome *southbound*. A interface *southbound* é a responsável por determinar os protocolos de comunicação utilizados entre o plano de controle e o de dados. Já a interface *northbound* define as instruções de nível de máquina (baixo nível) que a interface *southbound* utiliza para prover a programabilidade dos dispositivos de encaminhamento de dados. A seguir os elementos da arquitetura SDN serão detalhados.

- **Plano de gerenciamento**. Neste plano, são definidas as políticas de alto nível que serão capazes de configurar o plano de dados. Existem APIs, do inglês *Application Programming Interface*, desenvolvidas em várias linguagens de programação, como *Python*, *Java* e *C++*. A API é uma Interface de Programação de Aplicativos.
- **Plano de controle**. É responsável por aplicar as políticas de encaminhamento, definindo por onde os dados serão repassados. Na arquitetura SDN, o plano de controle é separado do plano de dados, permitindo um controle centralizado da rede e uma visão global da mesma. O dispositivo responsável por essa centralização do plano de controle é o controlador SDN.
- **Controlador**. É a peça central de uma rede SDN, em que são programadas as funções de rede. A partir do controlador é possível configurar ou reconfigurar os dispositivos presentes no plano de dados de modo que se tenha uma visão

completa da rede.

- **Plano de dados**. É onde de fato os dados irão trafegar. Este plano é responsável por encaminhar os fluxos de dados de acordo com as politicas de encaminhamento estabelecidas pelo plano de controle.

3.2.1 Controladores

Também conhecido como NOS, o controlador é uma peça importante para o paradigma SDN. A arquitetura SDN é composta por um conjunto de equipamentos como roteadores e *switches*, assim como nas arquiteturas de rede tradicionais. No entanto o diferencial é que em SDN estes dispositivos são somente utilizados para o encaminhamento de dados, sem possuírem o *software* de controle embarcado nos mesmos. Desse modo a "inteligência" da rede é removida do plano de dados dos dispositivos físicos para ser centralizada logicamente pelo controlador da rede SDN (KREUTZ et al., 2015).

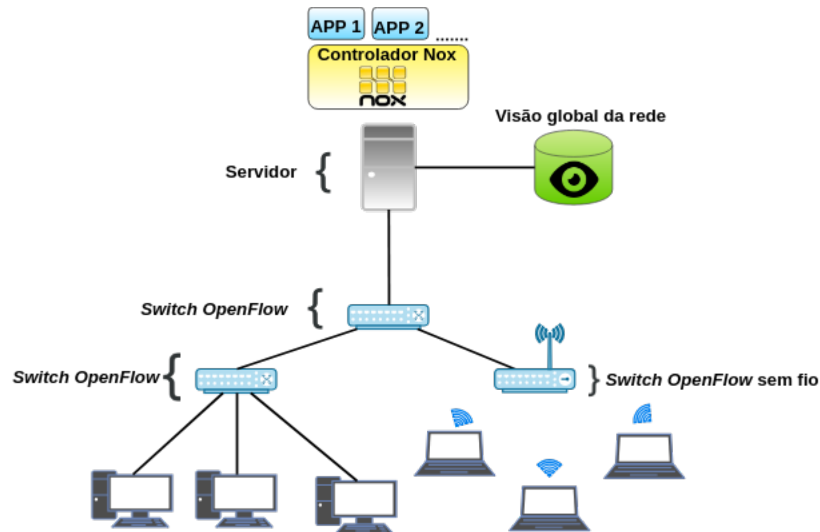
Dada a importância do controlador em SDN, este trabalho utiliza controladores SDN, a fim de manter uma estrutura centralizada e também ter um controle e gerenciamento mais eficiente dos dispositivos IoT e desta forma realiza-se uma análise comparativa de tais controladores integrados em cenários IoT. Pode-se afirmar que existem diversos controladores *openSource*, alguns deles são: NOX[2], OpenDaylight, Ryu, Floodligth[3], POX, Maestro[4], Trema[5] e Beacon[6]. Os controladores citados serão descritos nos parágrafos subsequentes.

3.2.1.1 NOX

O primeiro controlador SDN foi o NOX, desenvolvido pela *Nicira Networks*, em conjunto com o protocolo *OpenFlow*. O controlador NOX clássico é disponibilizado sob uma licença pública geral e tem suporte às linguagens de programação *C++* e *Python*. Já o "novo NOX" apenas tem suporte a *C++* e possui menos aplicações que a versão clássica, no entanto é mais rápida comparada à clássica.

[2] https://github.com/noxrepo/nox
[3] http://www.projectfloodlight.org/floodlight/
[4] https://www.sdxcentral.com/projects/maestro/
[5] https://github.com/trema/trema
[6] https://openflow.stanford.edu/display/Beacon/Home

Figura 10 – Componentes de uma rede baseada em NOX



Fonte: Adaptado de Gude et al. (2008).

A Figura 10 ilustra os componentes primários de uma rede baseada em NOX. Contendo um conjunto de *switches* habilitados com o protocolo *OpenFlow* e um servidor conectado na rede. O *software* do controlador NOX esta sendo executado no servidor, assim como as aplicações de gerenciamento que rodam no NOX. A visão global da rede contém os resultados das observações realizadas pelo NOX na rede, as aplicações utilizam esses resultados para tomar decisões de gerenciamento da rede (GUDE et al., 2008).

Na literatura pesquisada o controlador NOX já encontra-se obsoleto e é incentivado que se utilize o controlador POX ao invés dele. Por este motivo o controlador NOX não entrou nem na fase de instalação. Consequentemente não foi utilizado neste trabalho.

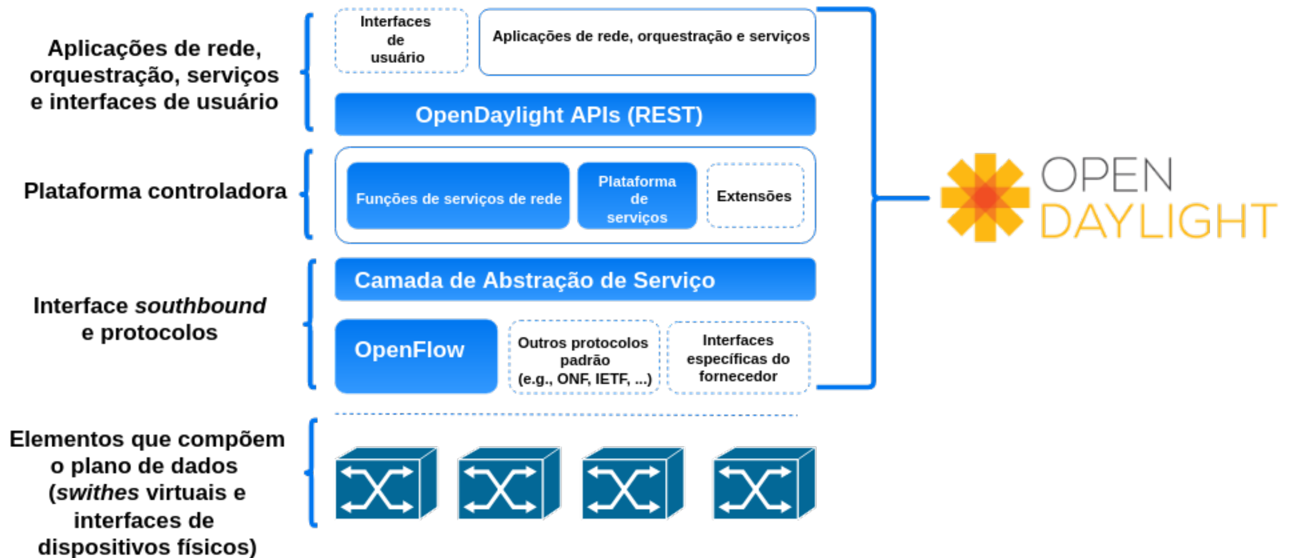
3.2.1.2 OpenDaylight

O controlador OpenDaylight é um projeto de código fonte aberto hospedado pela *Linux Foundation* e destinado ao aprimoramento de SDN, oferecendo suporte para a comunidade e para a indústria. Tem suporte ao protocolo *OpenFlow* e é baseado na linguagem de programação Java. O controlador OpenDaylight apresenta uma interface web para facilitar a visualização dos elementos como *hosts* e *switches*.

Segundo Badotra e Singh (2017), para fornecer o controle centralizado da rede, de maneira programática, e também para fornecer monitoração dos dispositivos de rede o controlador OpenDaylight usa protocolos abertos. Um dos principais protocolos abertos utilizados é o

OpenFlow.

Figura 11 – Arquitetura do controlador OpenDaylight



Fonte: Adaptado de Badotra e Singh (2017), Machado (2018).

A Figura 11 mostra a arquitetura do controlador OpenDaylight. Sua arquitetura é constituída por camadas, sendo a camada plataforma controladora (camada de controle) a mais importante (BADOTRA; SINGH, 2017), pois é onde de fato o controlador reside e atua como um cérebro para a rede. Pois gerencia o fluxo de tráfego dos *swithes* utilizando tabelas de fluxo.

A camada de abstração de serviço, encontra-se abaixo da camada plataforma controladora de acordo com a Figura 11. Nela encontra-se a interface *southbound* e protocolos, que tem suporte para vários protocolos e fornece serviços consistentes para módulos e aplicações. As aplicações de rede, orquestração, serviços e interfaces de usuário e Opendaylight APIs fazem parte da camada de aplicação. Por fim na camada de dados estão os elementos que compõem o plano de dados, como por exemplo, *swithes* virtuais e interfaces de dispositivos físicos.

O controlador OpenDaylight, não foi utilizado por que o processo de instalação na máquina virtual não foi finalizado com sucesso.

3.2.1.3 Ryu

Ryu é um controlador de código aberto, implementado em sua totalidade na linguagem *Python*. Este controlador fornece componentes de *software* com interfaces de

programação de aplicação bem definidas, facilitando a criação, o gerenciamento e o controle da rede. Tem suporte a vários protocolos para gerenciar dispositivos de rede, como *OpenFlow*, *Netconf* e *OF-config*. Sobre o *OpenFlow*, o Ryu suporta totalmente as extensões 1.0, 1.2, 1.3, 1.4, 1.5 e *Nicira* (OSRG, 2014b).

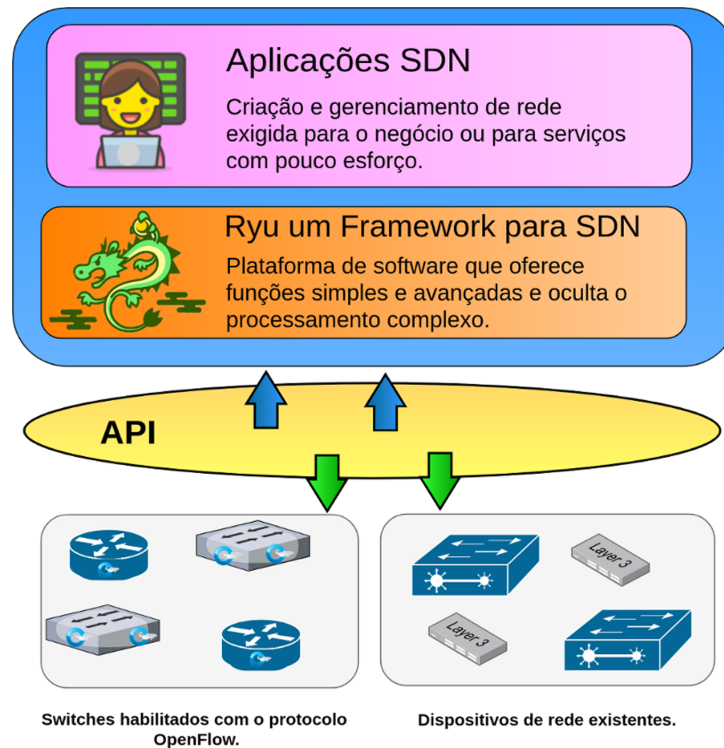
Segundo Kubo et al. (2014), o controlador Ryu fornece a funcionalidade de coleta de informações de dispositivos de rede convencionais, bem como de dispositivos habilitados com o protocolo *OpenFlow*. Dessa forma, evitando a divisão do sistema de gerenciamento em dois sistemas distintos, um para dispositivos convencionais e outro para dispositivos habilitados com *OpenFlow*.

O Ryu suporta tanto o protocolo OF-CONFIG, do inglês *OpenFlow Management and Configuration Protocol*, como o protocolo NETCONF, do inglês *Network Configuration Protocol*, na função de gerenciamento de configurações de *switch*. Possibilitando assim, a criação de ferramentas para a unificação de alteração de configurações, tanto dos *switches* habilitados com o *OpenFlow* como para os dispositivos de rede convencionais (KUBO et al., 2014).

Segundo Kubo et al. (2014), o controlador Ryu fornece principalmente ferramentas e bibliotecas que facilitam o processo de desenvolvimento de aplicações SDN. Aplicações que são comumente utilizadas estão inclusas nos exemplos de aplicações. Os exemplos incluem aplicações que implementam *firewalls* e roteadores, também incluem funções que são frequentemente utilizadas na construção de redes, como *link aggregation* e *spanning trees*.

A Figura 12 mostra a visão arquitetural do controlador Ryu. Apresentando a camada de aplicações SDN que é responsável pela criação e pelo gerenciamento de rede, exigida para o negócio ou para os serviços de maneira a reduzir esforço (KUBO et al., 2014). Na camada de controle está o controlador Ryu, que é uma plataforma de *software* que oferece funções simples e também funções avançadas e oculta o processamento complexo (KUBO et al., 2014). E por fim, a camada de dados onde podem estar presentes tanto os *switches* habilitados com o protocolo *OpenFlow* como também os dispositivos de rede existentes (convencionais). A comunicação entre as camadas citadas ocorre por meio de uma API.

Figura 12 – Visão arquitetural do controlador Ryu



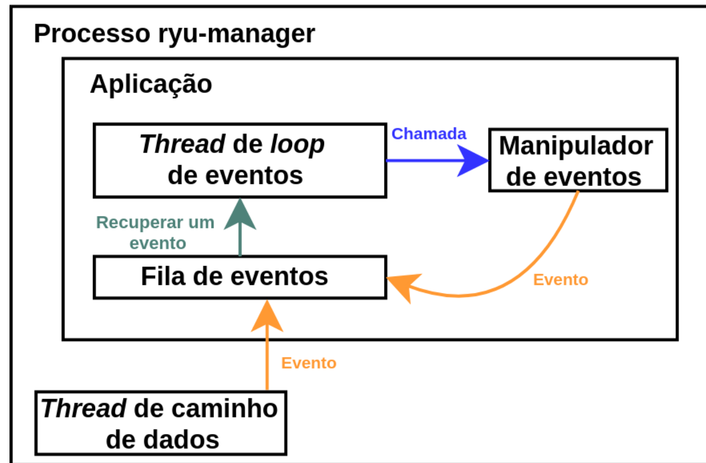
Fonte: Adaptado de Kubo et al. (2014).

A Figura 13 mostra o modelo de programação de aplicações do controlador Ryu. As aplicações herdam de uma classe denominada *RyuApp* que é encontrada em *ryu.base.app_manager.RyuApp* (OSRG, 2014a). Já os eventos são herdados da classe *EventBase* presente em *ryu.controller.event.EventBase*.

A comunicação entre as aplicações é realizada através da transmissão e recebimento dos eventos e cada aplicação tem uma fila única para o recebimento dos eventos. O controlador Ryu apresenta um modelo de execução *multithread*, e uma *thread* é criada automaticamente para cada aplicação. Esta *thread* executa em um *loop* de eventos, caso haja algum evento na fila de eventos, o *loop* de eventos carregará o evento e vai chamar o manipulador de eventos correspondente.

O controlador Ryu permite a criação de novos manipuladores de eventos para as aplicações, então após a criação, quando ocorre um evento do tipo especificado pelo manipulador de eventos, o manipulador de eventos correspondente para aquele evento é chamado a partir do *loop* de eventos da aplicação (OSRG, 2014a).

Figura 13 – Modelo de programação de aplicações do Ryu



Fonte: Adaptado de osrg (2014a).

O controlador Ryu foi utilizado neste trabalho. Devido ao fato de ser muito fácil o processo de instalação e de não ocorrer erros ao testar a conectividade com o emulador Mininet-WiFi.

3.2.1.4 *Floodlight*

O Floodlight é um controlador SDN de código aberto. É um controlador *OpenFlow* baseado em *Java* de classe empresarial e licença *Apache*, tem suporte a *swithes* físicos e virtuais. Também é possível executar este controlador em redes que não utilizem o protocolo *OpenFlow*. A linguagem suportada por este controlador é a *Java*. É apoiado por uma comunidade de desenvolvedores, incluindo vários engenheiros da *Big Switch Networks* (NETWORKS, 2019).

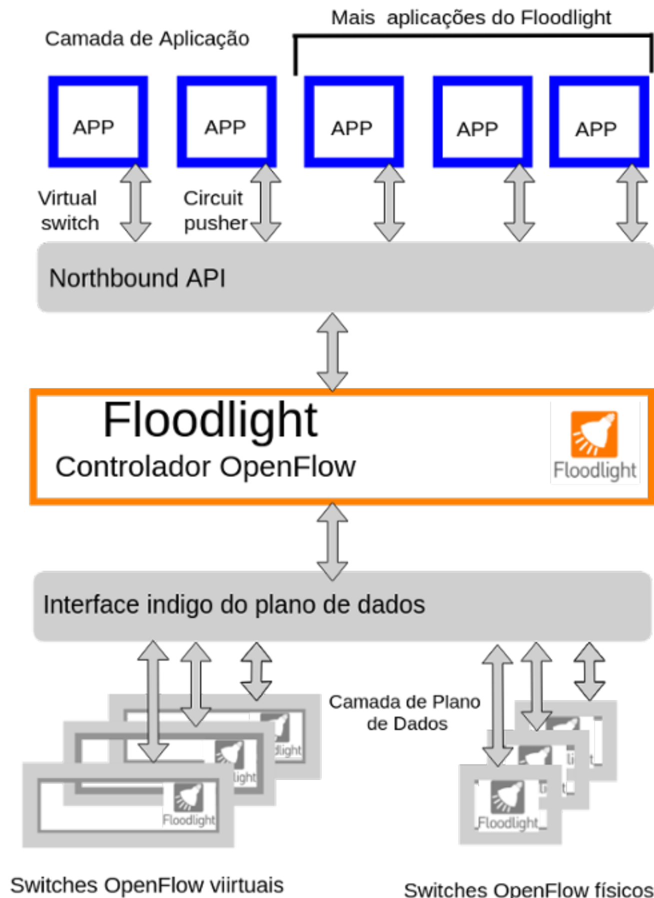
O Floodlight foi projetado para funcionar com o crescente número de *switches*, roteadores, comutadores virtuais e pontos de acesso que suportam o padrão *OpenFlow*.

Segundo Networks (2019), os motivos para se utilizar o controlador Floodlight são:

- Funciona com switches físicos e virtuais que entendam o protocolo *OpenFlow*.
- Licenciado com *Apache*, isso permite usar o Floodlight para praticamente qualquer finalidade.
- O Floodlight é desenvolvido por uma comunidade aberta de desenvolvedores. Possibilitando contribuições de código vindas de diversas pessoas ativas, de modo que, as informações sobre o projeto, sobre o roteiro de desenvolvimento e sobre os *bugs* são abertamente compartilhados.

- Fácil de usar, o Floodlight é simples de instalar e executar.
- Testado e suportado, o Floodlight é testado e aprimorado ativamente por uma comunidade de desenvolvedores profissionais.

Figura 14 – Visão arquitetural do controlador Floodlight



Fonte: Adaptado de Networks (2019).

Ainda, de acordo com Networks (2019), o controlador Floodlight oferece um sistema de carregamento de módulos que simplifica a extensão e o aprimoramento, é fácil de configurar apresentando poucas dependências. Suporta uma ampla gama de comutadores *OpenFlow* virtuais e físicos. Pode lidar com redes mistas *OpenFlow* e não *OpenFlow*, capacidade de gerenciar múltiplas "ilhas" de *switches* de *hardware OpenFlow*. Projetado para ser de alto desempenho e *multithread*. Suporte para plataforma de orquestração em nuvem *OpenStack*.

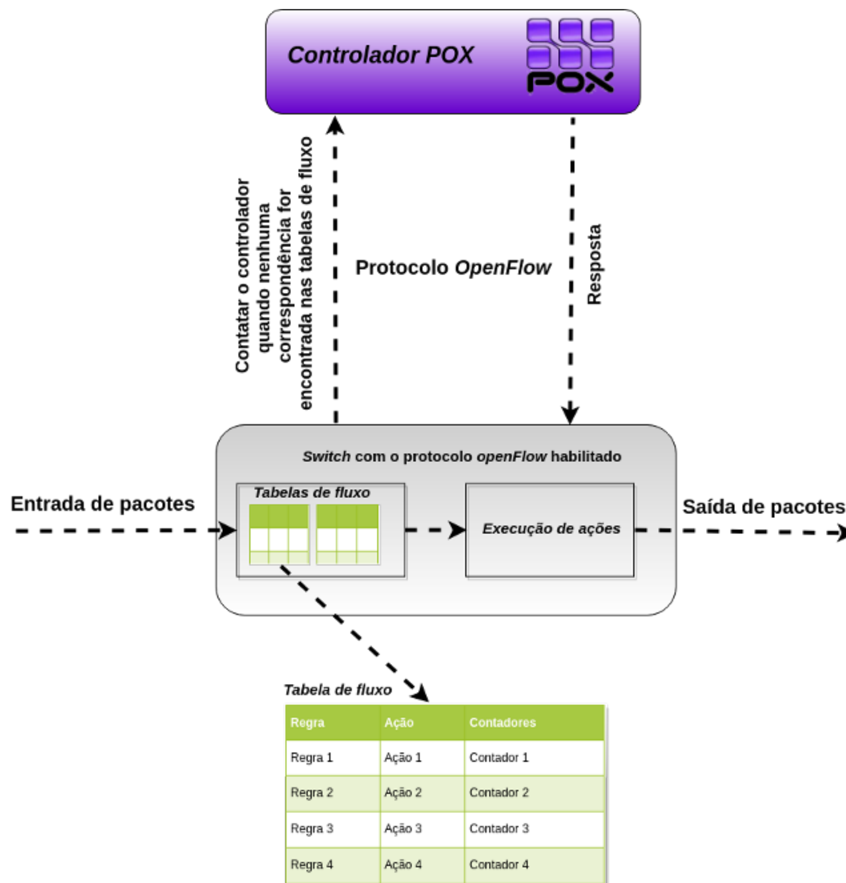
O controlador Floodlight foi utilizado neste trabalho, pelo êxito na instalação. O processo de instalação não foi fácil como o processo do controlador Ryu. Por apresentar alguns erros e por isso ser escolhido utilizar a versão 1.2. O número de comandos também foi superior

do que o do controlador Ryu.

3.2.1.5 POX

O POX é um controlador de rede *OpenFlow* de código aberto, escrito na linguagem *Python*. Contém interface *OpenFlow* e componentes reusáveis para seleção e descoberta de topologias (PRIYADARSINI; BERA; BHAMPAL, 2017). No inicío o POX funcionava somente como um controlador *OpenFlow*, mas agora também pode funcionar como um *switch OpenFlow*, e pode ser útil para escrever *software* de rede em geral (MCCAULEY KYRIAKOS ZARIFIS, 2009).

Figura 15 – Controlador POX



Fonte: Adaptado de Kaur, Singh e Ghumman (2014).

O POX atualmente se comunica com os *switches OpenFlow* 1.0 e inclui suporte especial para as extensões *Open vSwitch/Nicira*. Algumas das principais características deste controlador são descritas em (BHOLEBAWA; DALAL, 2018), são elas:

- Pode fornecer uma interface "*Pythonic OpenFlow*".
- Possuir componentes de simples reutilização, como por exemplo para descoberta de caminho e descoberta de topologia.
- Capacidade de rodar em qualquer ambiente de sistema operacional.
- Pode suportar a mesma Interface Gráfica de Usuário, do inglês *Graphical User Interface* (GUI) do NOX e também a arquitetura virtual semelhante ao do NOX.
- Podendo fornecer um melhor desempenho em comparação ao NOX escrito em *Python*.

O controlador POX oferece um modo eficiente para a implementação do protocolo *OpenFlow*, que é o protocolo responsável pela comunicação entre os controladores e os *switches*. Este controlador possibilita a execução de aplicações tais como balanceamento de carga e de *firewall*. A ferramenta *Tcpdump* pode ser usada para a captura dos pacotes e para visualização do tráfego dos pacotes entre o controlador POX e os dispositivos habilitados com o protocolo *OpenFlow* (KAUR; SINGH; GHUMMAN, 2014).

Segundo Kaur, Singh e Ghumman (2014), após o *switch* conectar-se ao controlador a sua tabela de fluxo encontra-se vazia, dessa maneira quando um pacote chega ao *switch*, ele não sabe como deve manipular aquele pacote. Desta forma, o *switch* envia uma mensagem contatando o controlador pois nenhuma correspondência foi encontrada em sua tabela de fluxo (Figura 15). Para que o *switch* saiba como manipular o pacote o controlador insere uma entrada de fluxo na tabela de fluxo do *switch*. A entrada de fluxo na tabela de fluxo é constituída por três partes, regra (campo de correspondência), ação e contadores. Para cada pacote que passar por um *switch*, será preciso instalar uma entrada de fluxo na tabela de fluxo do *switch* para que o tráfego de pacotes seja devidamente encaminhado sem a necessidade de intervenção adicional do controlador (KIM; FEAMSTER, 2013).

A Figura 15 mostra o processo descrito anteriormente. Quando um pacote entra no *switch*, o *switch* verifica em suas tabelas de fluxo se há alguma regra para aquele fluxo de entrada, se houver o *switch* executa as ações relacionadas a regra daquele fluxo de entrada. Caso não haja nenhuma regra associada para aquele fluxo de entrada o *switch* envia uma mensagem contatando o controlador. Em seguida o controlador como resposta envia uma entrada de fluxo para o *switch*. Esta entrada será instalada na tabela de fluxo do *switch* para que os fluxos daquela categoria sejam encaminhados corretamente.

O controlador POX foi utilizado neste trabalho, por ter sido o controlador mais fácil

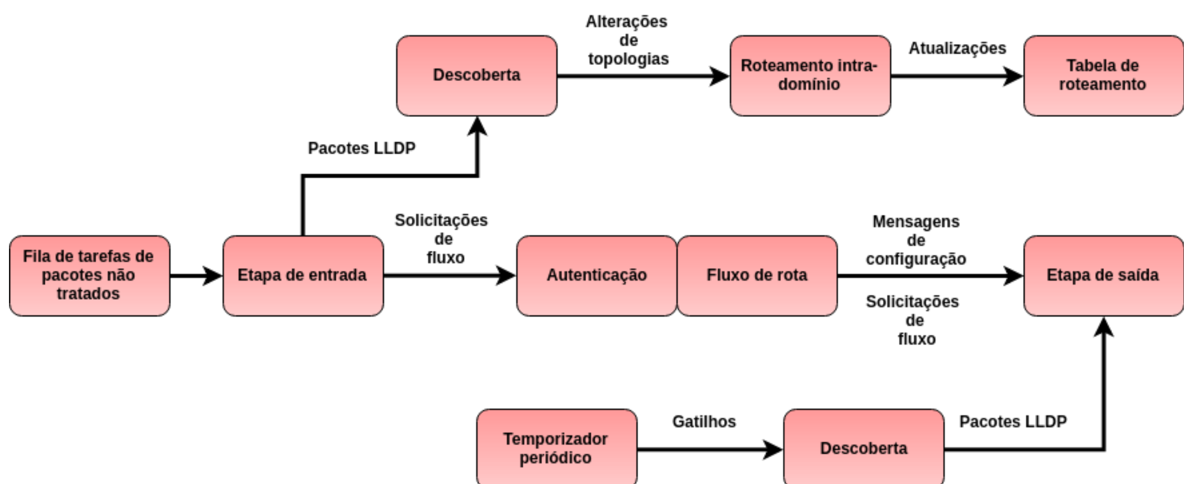
de se instalar e não apresentar arquivos extras de instalação após o *clone* de seu repositório.

3.2.1.6 Maestro

O Maestro é um “Sistema Operacional” para a orquestração do controle de aplicações de rede. Fornecendo interfaces para implementação de aplicações modulares de controle de rede para acessar e modificar o estado da rede e coordenar suas interações (UNIVERSITY, 2019). Desse modo, o Maestro é uma plataforma para obter funções de controle de rede automáticas e programáveis utilizando-se de aplicações modularizadas. Embora o foco do Maestro seja a construção de um controlador *OpenFlow*, este não se limita apenas às redes *OpenFlow*. A estrutura de programação do Maestro fornece interfaces para:

- Introdução de novas funções de controle personalizadas, adicionando componentes de controle modularizados.
- Manter o estado da rede a cargo dos componentes de controle.
- Compor os componentes de controle especificando o sequenciamento de execução e o estado de compartilhamento de rede dos componentes.

Figura 16 – A estrutura geral do controlador Maestro



Fonte: Adaptado de Cai, Cox e Ng (2010).

Além disso, o Maestro tenta explorar o paralelismo dentro de uma única máquina para melhorar o desempenho de rendimento do sistema. O recurso fundamental de uma rede *OpenFlow* é que o controlador é responsável pelo estabelecimento inicial de todos os fluxos, entrando em contato com os *switches* relacionados (UNIVERSITY, 2019). Assim, o desempenho

do controlador pode ser o ponto de falha. Os criadores do Maestro tentaram exigir o mínimo de esforço possível dos programadores para gerenciar o paralelismo. Em vez disso, o Maestro lida com a maior parte da tarefa tediosa e complicada de gerenciar a distribuição de carga de trabalho e o trabalho de agendamento de *Threads*. Por *design*, o Maestro é portável e escalável:

- Desenvolvido em *Java* (a plataforma e os componentes). É altamente portável, pois pode ser usado em vários sistemas operacionais e em várias arquiteturas.
- É *multithread*, aproveitando ao máximo os processadores *multi-core*.

A Figura 16 mostra a estrutura geral do controlador Maestro, de acordo com Cai, Cox e Ng (2010), a etapa de entrada e a etapa de saída lidam com os detalhes de baixo nível de leitura e gravação dos *buffers* de soquete e na conversão de mensagens *OpenFlow* não tratadas em estruturas de dados de alto nível. Outras funcionalidades de alto nível podem ser implementadas em um módulo denominado aplicações no Maestro. Os programadores podem modificar as aplicações existentes de forma flexível, isto é, modificar o comportamento das aplicações ou ainda adicionar novas aplicações, a fim de atender aos objetivos do desenvolvedor. Na Figura 16 as aplicações verificadas são descoberta, autenticação, fluxo de rota e roteamento intra-domínio.

O controlador Maestro não foi usado neste trabalho pela dificuldade que encontrou-se para instalá-lo no ambiente de trabalho.

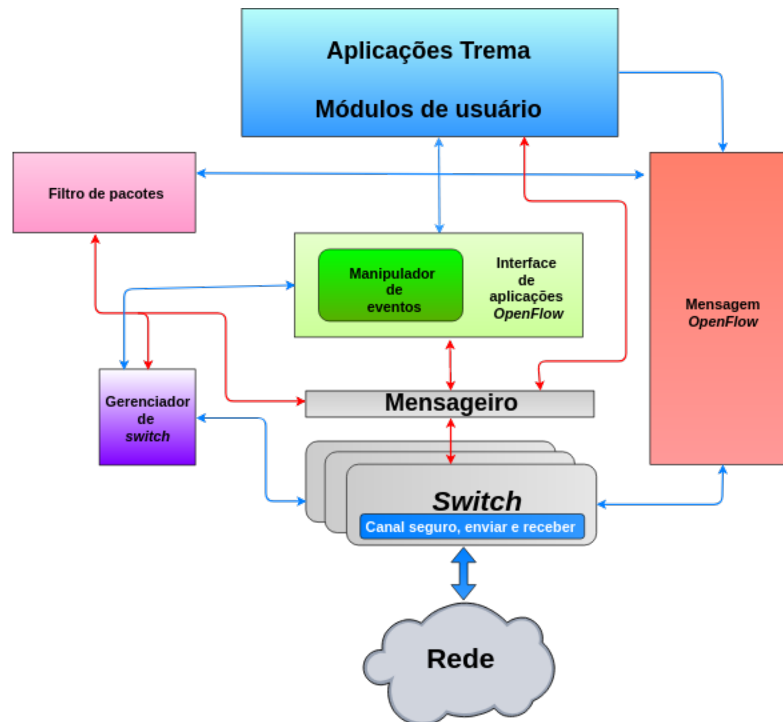
3.2.1.7 Trema

Trema fornece tudo o que é necessário para a criação de controladores *OpenFlow* na linguagem de programação *Ruby* e *C*. Fornecendo uma biblioteca *OpenFlow* de alto nível e também um emulador de rede que pode criar redes baseadas no protocolo *OpenFlow* (SHIMONISHI Y. TAKAMYA, 2018).

Os módulos principais do controlador Trema incluem, em especial, os módulos essenciais do protocolo *OpenFlow* e normalmente outros módulos que são úteis para várias aplicações (RAO, 2014). Os módulos *switch* e gerenciador de *switch* implementam as funcionalidades necessárias para a comunicação correta com os *switches OpenFlow*, mantendo ainda todas as informações importantes sobres os *switches* (RAO, 2014). O módulo de filtro de pacotes de entrada permite ao *switch OpenFlow* enviar pacotes capturados ao controlador Trema. O *switch* envia os pacotes ao controlador somente quando é solicitado pelo controlador ou quando não existe uma entrada adequada na tabela de fluxo do *switch*. Em suma, o módulo de filtro de pacotes de entrada é o responsável pela entrega dos pacotes que vem do módulo

gerenciador de *switch* ao controlador Trema (RAO, 2014). A Figura 17 representa o diagrama funcional reduzido do controlador Trema.

Figura 17 – Diagrama funcional reduzido do controlador Trema



Fonte: Adaptado de Rao (2014).

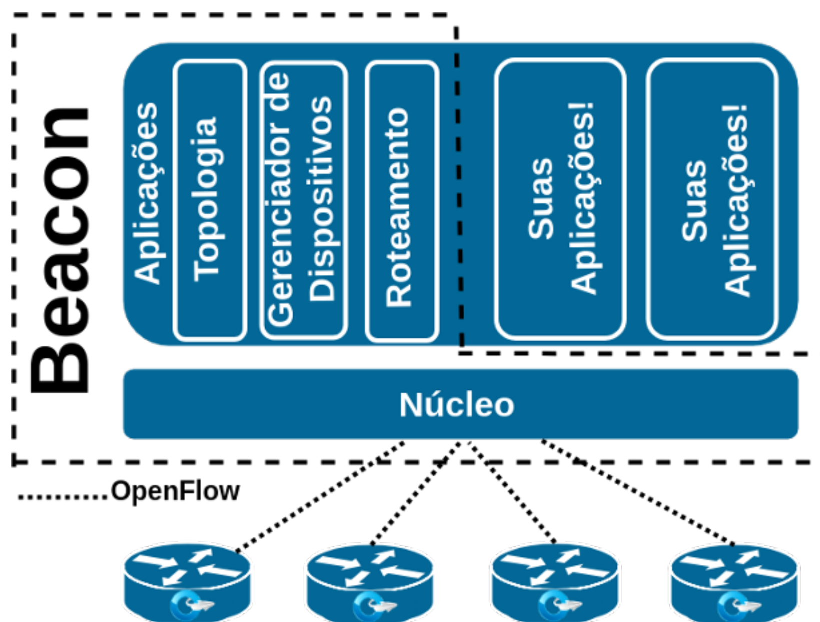
Em continuação a respeito dos módulos, o módulo *TremaShark* é um *plugin* do *Wireshark* para o rastreamento de eventos de comunicação entre processos dos módulos funcionais. Os eventos podem variar de mensagens para o status do canal seguro ou para o status da fila (RAO, 2014). O módulo de bibliotecas do Trema, abrange as categorias de protocolo (*OpenFlow*), interfaces (aplicação *OpenFlow*, *switch* e gerenciamento), estruturas de dados (como exemplo lista, *hash* e *buffer*), utilitários (*logs* e estatística) e protocolos de redes. Segundo Rao (2014), as aplicações Trema, denominadas de módulos de usuário, incluem cerca de 15 a 20 aplicações distintas, que implementam funcionalidades de alto nível e que podem ser usadas como ponto de referência para a criação de novas aplicações pelos desenvolvedores.

O controlador Trema não foi utilizado neste trabalho. Não conseguiu-se instalá-lo. Ademais, o controlador Trema de acordo com a literatura somente é usado para modo de pesquisa e não possui integração com o Mininet-WiFi.

3.2.1.8 Beacon

O Beacon é um controlador *OpenFlow* de *software* livre baseado em *Java*, este controlador já vem sendo utilizado para o ensino e pesquisa em SDN (ERICKSON, 2013b). Desenvolvido no ano de 2010, pela universidade de *Stanford* como um projeto de pesquisa (HOANG; PHAM, 2015), tal controlador foi projetado para atender a três objetivos principais de acordo com Hoang e Pham (2015), o desempenho, a facilidade de desenvolvimento de aplicações e a configuração de tempo de execução.

Figura 18 – Visão geral da arquitetura do controlador Beacon



Fonte: Adaptado de Erickson (2013b).

Uma visão mais ampla da arquitetura do Beacon é mostrada na Figura 18. O Beacon fornece uma estrutura para controlar dispositivos de rede usando o protocolo *OpenFlow* e um conjunto de aplicações integradas que fornecem a funcionalidade de plano de controle comumente necessária ao paradigma SDN.

De acordo com Erickson (2013a), o Beacon é um controlador *OpenFlow* baseado em *Java*, rápido, multiplataforma e modular que suporta tanto operações baseadas em eventos quanto operações com *Threads*. Sendo suas principais características:

- **Estável**: está em desenvolvimento desde o início de 2010 e tem sido usado em vários projetos de pesquisa, em classes e implantações experimentais. O Beacon

atualmente alimenta um *data center* experimental com 100 *Switches* virtuais e 20 *Swithes* físicos e funciona há meses sem nenhuma queda.

- **Multiplataforma**: é escrito em *Java* e pode ser executado em várias plataformas, desde servidores *Linux* até telefones *Android*.
- **Código aberto**: é licenciado sob uma combinação[7] da licença GPL v2 e da licença *Stanford University FOSS License Exception v1.0*.
- **Dinâmico**: os pacotes de códigos no Beacon podem ser iniciados, interrompidos, atualizados e instalados durante a execução (em ambiente de produção), sem interromper outros pacotes não dependentes.
- **Desenvolvimento rápido**: é fácil de criar e de executar. O *Java* e o *Eclipse* simplificam o desenvolvimento e a depuração de aplicativos.
- **Rápido**: foi desenvolvido para suportar o modelo *multithread*.
- **Interface de Usuário *Web***: Opcionalmente utiliza o servidor *web* corporativo *Jetty* e um *framework* de Interface de Usuário extensível e personalizável.
- ***Frameworks***: foi construido com base em *frameworks* atuais como *Spring*[8] e *Equinox*[9] (OSGi).

O controlador Beacon também não foi utilizado neste trabalho. Pelo mesmo fato de não ter sucesso no processo de instalação do mesmo.

3.2.1.9 Principais características dos controladores citados

O Quadro 2 mostra as principais características presentes nos controladores que foram apresentados na Seção 3.2.1. Os controladores em questão são: POX; Ryu; Trema; Floodlight; OpenDaylight; NOX; Maestro e Beacon.

Todos os controladores do Quadro 2 são de código aberto. Metade deles tem suporte a **REST API**. Os que possuem suporte são Ryu, Floodlight e OpenDaylight. Em relação à **linguagem** de programação **implementada**, os controladores POX e Ryu foram implementados na linguagem Python. Já os controladores Floodlight, OpenDaylight, Maestro e Beacon são implementados em Java. Por fim, o controlador Trema é implementado em Ruby e C.

Outra característica importante é a **versão** do protocolo ***OpenFlow***. Alguns controladores suportam somente a versão 1.0: POX, Trema, OpenDaylight, NOX, Maestro e

---
[7] https://openflow.stanford.edu/display/Beacon/License.html
[8] https://spring.io/
[9] https://www.eclipse.org/equinox/

Beacon. O controlador Ryu tem suporte da versão 1.0 à 1.5. E o controlador Floodlight suporta a versão 1.3 do protocolo *Openflow*.

Quadro 2 – Controladores SDN

	Linguagem implementada	***OpenFlow***	**REST API**	**GUI**	**Documentação**	**Plataforma suportada**	**Suporte Multithread**	**Domínio de aplicação**	**Distribuído ou centralizado**
POX	Python	v1.0	Não	Sim	Pouca	Linux, MAC OS e Windows	Não	Campus	Centralizado
Ryu	Python	v1.0 à v1.5	Sim	Sim	Média	Linux	Sim	Campus	Centralizado
Trema	Ruby e C	v1.0	Não	Não	Média	Linux	*	*	*
Floodlight	Java	v1.3	Sim	Sim	Muita	Linux, MAC OS e Windows	Sim	Campus	Centralizado
OpenDaylight	Java	v1.0	Sim	Sim	Muita	Linux, MAC OS e Windows	Sim	Datacenter	Distribuído
NOX	C++	v1.0	Não	Sim	Pouca	Linux	*	Campus	Centralizado
Maestro	Java	v1.0	Não	*	Pouca	Linux, MAC OS e Windows	Sim	Pesquisa	Centralizado
Beacon	Java	v1.0	Não	Sim	Média	Linux, MAC OS e Windows	Sim	Pesquisa	Centralizado

Fonte: Adaptado de Nunes et al. (2014), Kaur, Singh e Ghumman (2014), Rastogi e Bais (2016), Salman et al. (2016b), Khondoker et al. (2014).

Em relação ao suporte a GUI, somente o Trema não está habilitado. Os demais controladores possuem suporte a GUI. Os controladores POX, NOX e Maestro, tem pouca documentação. Os controladores que apresentam uma documentação média são: Ryu; Trema; e Beacon. Os controladores Floodlight e OpenDaylight apresentam muita documentação.

Todos os controladores do Quadro 2 tem suporte as plataformas *Linux, MAC OS* e *Windows*, exceto o Ryu que apresenta suporte somente ao *Linux*. Na categoria suporte a *multthread*, apenas o POX não apresentou suporte, todos os outros tem suporte a *multthread*. Apenas o controlador OpenDaylight e caracterizado como distribuído os outros são caracterizados como centralizados. Por fim, a respeito do domínio de aplicação, os controladores que tem o domínio de aplicação em campus são: POX; Ryu; Floodlight; e NOX. Os que tem domínio de aplicação em pesquisa são o Maestro e o Beacon. Por último, o OpenDaylight apresenta o domínio de aplicação em *Datacenter*.

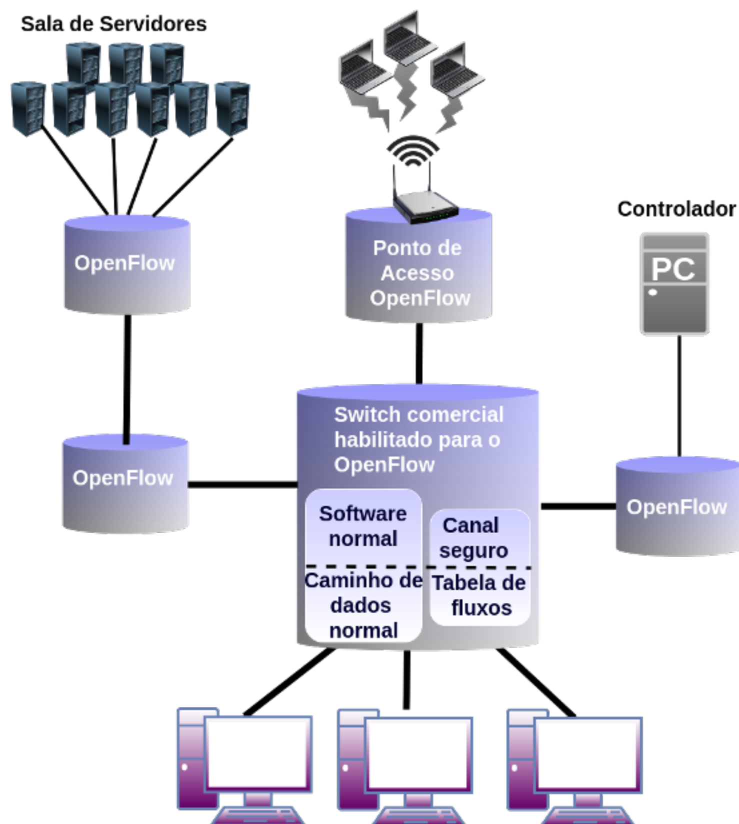
3.2.2 OpenFlow

O protocolo *OpenFlow* é o mais utilizado para a realização da comunicação entre o plano de dados e o plano de controle em SDN. Este protocolo foi desenvolvido a partir de um projeto de pesquisa no ano de 2008, pela universidade de *Stanford*, e posteriormente padronizado

pela *Open Networking Fundation* (ONF) em 2011 (LAISSAOUI et al., 2015).

A Figura 19 mostra uma rede de *switches* comerciais e um ponto de acesso habilitados com o protocolo *OpenFlow*. No exemplo de rede *OpenFlow* da Figura 19 todas as tabelas de fluxo presentes são gerenciadas somente por um controlador, no entanto, o protocolo *OpenFlow* permite que um *switch* seja gerenciado por mais de um controlador, e dessa forma consegue-se obter maior desempenho ou ainda evitar um ponto único de falha (obter robustez) (MCKEOWN et al., 2008).

Figura 19 – Exemplo de uma rede de *switches* e roteadores comerciais com *OpenFlow* habilitado



Fonte: Adaptado de McKeown et al. (2008).

Na arquitetura do protocolo *OpenFlow*, os dispositivos encarregados pelo encaminhamento, ou *switches OpenFlow*, contém uma ou mais tabelas de fluxo e também uma camada de abstração que se comunica com o controlador (NOS) através do protocolo *OpenFlow*. As tabelas de fluxo consistem em entradas de fluxo, para cada entrada destas tabelas é determinado como os pacotes que pertencem a um fluxo específico serão processados e encaminhados, deste modo as entradas de fluxo consistem tipicamente de campo de

correspondência ou regras de correspondência, contadores usados para coletar estatísticas para determinados fluxos e o conjunto de instruções ou ações a serem realizados (NUNES et al., 2014).

McKeown et al. (2008), Hu, Hao e Bao (2014), relatam que a arquitetura *OpenFlow* normalmente inclui 3 importantes componentes, como segue:

1. ***Switches***: o protocolo *OpenFlow*, que é de código aberto, especifica as regras gerais para monitorar ou alterar as tabelas de fluxo em diferentes *switches* e roteadores. Um *switch OpenFlow* é constituido de pelo menos 3 componentes:
   a) tabela de fluxo: cada tabela de fluxo possui um campo de ação associado a cada campo de entrada de fluxo.
   b) canal de comunicação: possui um canal de comunicação que disponibiliza o *link* necessário para a transmissão de comandos e transmissão de pacotes entre o controlador da rede e o *switch*.
   c) o protocolo *OpenFlow*: este protocolo habilita o controlador a ser capaz de comunicar-se com qualquer *switch* ou roteador da rede SDN.
2. ***Controladores***: o controlador tem a autonomia de adicionar, de corrigir, de atualizar ou de remover os fluxos de entrada das tabelas de fluxos. Desse modo, um controlador pode ser por exemplo uma simples aplicação, rodando em um computador, a fim de estabelecer estaticamente os fluxos para a interconexão de um conjunto de computadores em um determinado ambiente de testes.
3. ***Entrada de fluxo***: cada entrada de fluxo contém ao menos uma ação simples a ser realizada, ou seja, envolve uma operação de rede a ser desempenhada para este determinado fluxo de entrada. A maior parte dos *switches OpenFlow* tem suporta as seguintes ações:
   a) enviar os pacotes do fluxo de entrada para uma porta especifica pré configurada para aquele determinado fluxo.
   b) encapsular os pacotes do fluxo de entrada e enviá-los para um controlador.
   c) eliminar os pacotes do fluxo de entrada.

Este trabalho utilizou do protocolo *OpenFlow*. Pois os controladores utilizam dele como principal protocolo de gerenciamento para SDN.

3.2.3 Mininet-WiFi

Segundo Fontes e Rothenberg (2019), o Mininet-WiFi[10] é um emulador desenvolvido na linguagem *Python*, dando suporte a emulação de redes sem fio e redes sem fio definidas por *software*. O módulo nativo do *kernel Linux* mac80211hwsim, é o responsável pelo suporte ao Wi-Fi, tendo também, suporte a uma vasta gama de variedades de protocolos 802.11, como IEEE 802.11a,b,g,n,ac,ax.

Ainda de acordo Fontes e Rothenberg (2019), este emulador vem sendo desenvolvido em cima do conceito de CBE, do inglês *Container-Based Emulation*, sendo esta uma das principais categorias de emulação de redes utilizando de *containers* leves. O nó que é emulado consiste em suma dos processos denominados de *user space* (espaço do usuário) do sistema operacional. Tendo em vista que há um compartilhamento entre o *kernel* do sistema operacional e o nó emulado, tendo em mente isso, a aplicação que for compatível com o sistema operacional *Linux* também será compatível com o nó emulado, como vantagem não é necessário codificar ou adaptar aplicações, pois as aplicações reais são suportadas desde que o sistema suporte-as (FONTES; ROTHENBERG, 2019).

O Mininet-WiFi foi criado justamente para o suporte ao WiFi (FONTES; ROTHENBERG, 2019), mesmo assim pesquisas utilizam-se do Mininet-WiFi com outras tecnologias de redes sem fio, haja visto que o Mininet-WiFi também tem suporte a redes LTE, do inglês *Long Term Evolution* e também já iniciaram extensões para tecnologias como o 6LoWPAN, sendo implementado nativamente para sistemas Linux através do módulo mac802154hwsim. De acordo com Fontes e Rothenberg (2019), o objetivo é habilitar mais tecnologias sem fio para ser possível que tendências tecnologicas, assim como IoT possam ser experimentadas através do Mininet-WiFi.

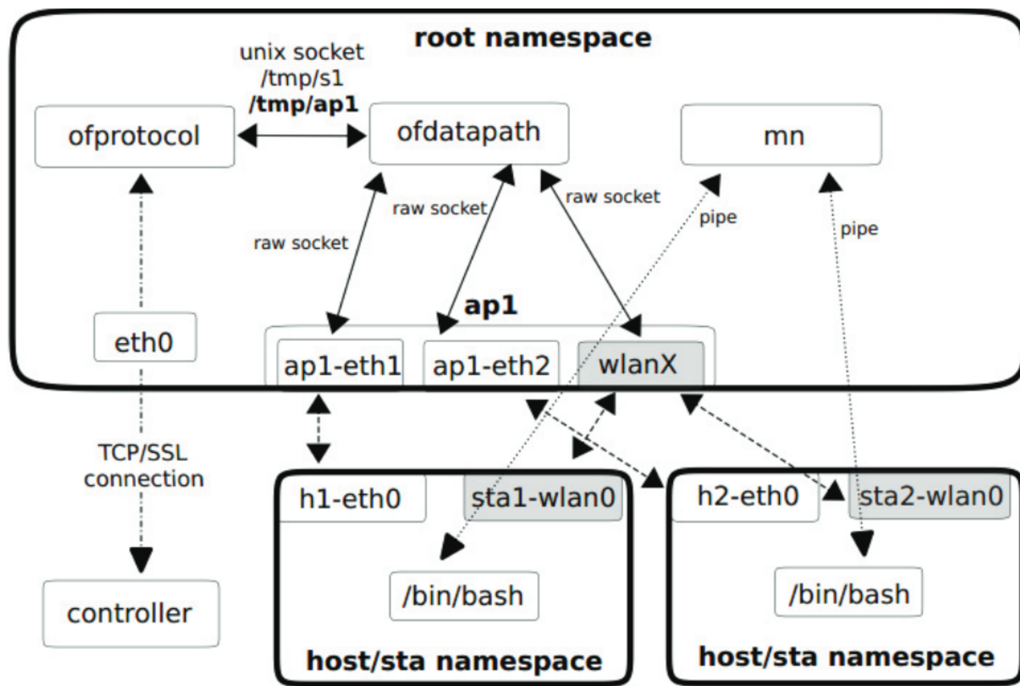
3.2.3.1 Arquitetura

O Mininet-WiFi utiliza o mesmo processo de virtualização do Mininet (Fontes et al., 2015; FONTES; ROTHENBERG, 2019) sendo baseado em processos que são executados em *Linux Network namespace* e placas de redes virtuais. Segundo Fontes e Rothenberg (2019), *Linux Network namespace* representam, de maneira lógica, uma cópia da pilha de rede do sistema operacional *Linux*, incluindo assim suas próprias rotas, regras de *firewall* e dispositivos de rede.

[10] O repositório do Mininet-WiFi é https://github.com/intrig-unicamp/mininet-wifi/.

Atuando como verdadeiros computadores detendo das mesmas propriedades de rede, do mesmo modo que observaria-se em um computador físico.

Figura 20 – Arquitetura do emulador Mininet-WiFi



Fonte: Fontes et al. (2015).

A Figura 20 mostra como está organizada a arquitetura do emulador Mininet-WiFi. Conceitualmente os pontos de acesso empregados pelo Mininet-WiFi são os mesmo empregados pelo Mininet, o diferencial é que no Mininet-WiFi os pontos de acesso são equipados com placas de rede WiFi operando em modo *master* (Fontes et al., 2015; FONTES; ROTHENBERG, 2019). A virtualização dos pontos de acesso é possível por intermedio do *daemon hostpad*[11] que tem como finalidade prover capacidade de pontos de acesso por meio de interfaces virtuais WiFi (Fontes et al., 2015; FONTES; ROTHENBERG, 2019).

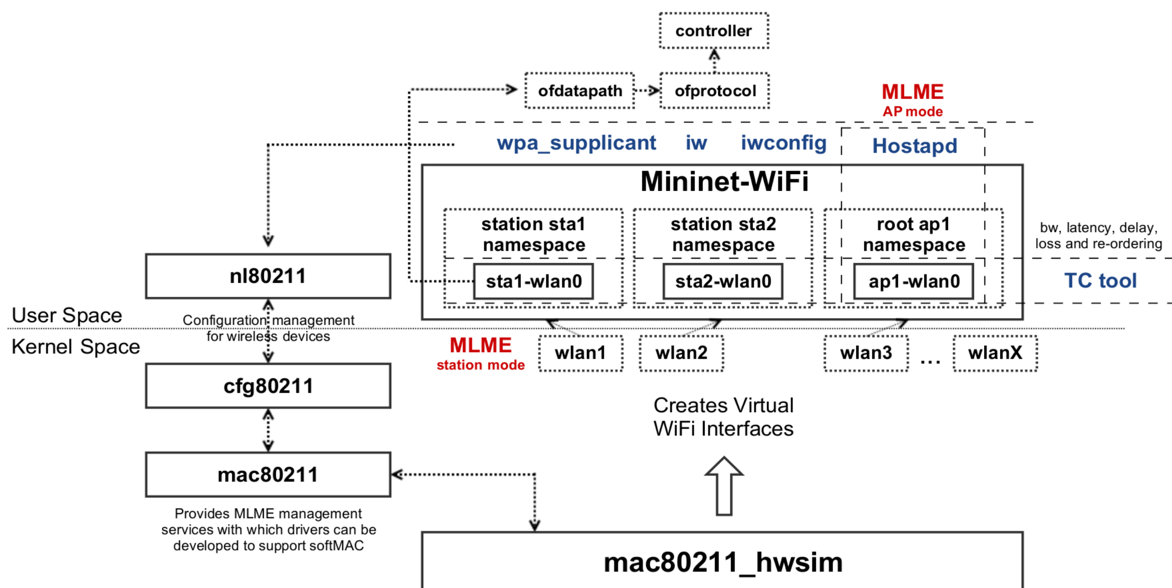
3.2.3.2 Funcionamento

A Figura 21 apresenta os componentes que integram a arquitetura do Mininet-WiFi. De acordo com Fontes et al. (2015), Fontes e Rothenberg (2016), Fontes e Rothenberg (2019), a comunicação entre os módulos ocorre da seguinte maneira: ao inicializar, o módulo

[11] Segundo Fontes et al. (2015), Fontes e Rothenberg (2019), Hostpad, do inglês, *Host Acess Point Daemon* é um *software* a nível de usuário que tem a capacidade de tornar uma interface de rede sem fio em pontos de acesso e servidores de autenticação.

denominado *mac80211_hwsim*, que é o responsável pela virtualização das placas WiFi, é então carregado contendo o número de interfaces sem fio suficientes para os demais nós que foram antecipadamente definidos pelo usuário.

Figura 21 – Principais componentes do Mininet-WiFi



Fonte: Fontes e Rothenberg (2016).

Todos os recursos suportados pelo *mac80211_hwsim* estão localizados no espaço do kernel do sistema operacional *Linux*, sendo provenientes do *mac80211*, que é um *framework* utilizado por desenvolvedores para a escrita de *drivers* para dispositivos sem fio baseados no *SoftMAC*.

Este trabalho utilizou o emulador Mininet-WiFi para a criação do ambiente dos nós IoT, através do *fakelb*. O emulador Mininet-WiFi permitiu a integração dos cenários IoT aos controladores SDN. Viabilizando o trabalho comparativo dos controladores SDN em cenários IoT.

4 METODOLOGIA

Este Capítulo apresenta as etapas percorridas para que as métricas de interesse do trabalho fossem devidamente coletadas. Após a coleta de todas as métricas espera-se que o objetivo do presente trabalho seja alcançado. O objetivo deste trabalho é o de comparar os controladores SDN em cenários IoT, a fim de descobrir qual dos controladores apresenta os melhores resultados em relação a latência, memória, consumo de processador e tempo de execução dos experimentos. A Seção 4.1 descreve o ambiente de trabalho utilizado para rodar os experimentos. Por fim, a Seção 4.2 descreve o processo de automatização dos experimentos.

4.1 Ambiente de trabalho

O ambiente de trabalho utilizado foi uma máquina virtual, do inglês *Virtual Machine* (VM), que foi criada e disponibilizada pela Universidade Federal do Ceará Campus Quixadá. A VM foi habilitada para acesso remoto através do protocolo SSH, do inglês *Secure SHell*, permitindo assim que fosse possível acessá-la por meio de um terminal de forma remota fora das imediações da Universidade.

Por padrão, a VM foi criada com o sistema operacional *Ubuntu* 14.04 LTS, contendo a versão do *kernel* padrão 4.4.0-31-*generic*. A memória RAM, do inglês *Random Access Memory*, definida foi de 2 *Gigabytes* e o espaço de armazenamento interno definido para a VM foi de 40 *Gigabytes*.

4.2 Automatização dos experimentos

Antes da implementação dos *scripts* de automatização dos experimentos, existiu a fase de definição dos fatores e níveis do trabalho. Os fatores podem ser entendidos como as configurações que variam dentro do experimento e os níveis representam cada valor que o fator pode ter. O Quadro 3 mostra os fatores e os níveis definidos para os experimentos.

O primeiro fator é o controlador SDN tendo os níveis Ryu, Floodlight e POX. Foram utilizados estes controladores pela facilidade de instalação. O Segundo fator é a quantidade de nós IoT contendo os níveis 26, 52 e 104. O motivo da escolha destas quantidades foi para ter muitos nós trafegando informações entre si nos cenários, e com isso, ter um grande tráfego de aplicação. O terceiro fator é o número de repetições de cada experimento com o nível 100. O quarto fator é a quantidade de pacotes *ping* enviados de nó para nó com o nível 500, outro fator

definido foi o tamanho do pacote *ping* com o nível 9999 *bytes* e por fim o último fator é o tráfego de aplicação com o nível MQTT.

A escolha dos fatores e níveis do Quadro 3 foram em virtude de criar cenários IoT que fossem parecidos com cenários reais. Isso é, tivessem um número significativamente grande de nós IoT e também houvesse tráfego de aplicação entre os nós.

Quadro 3 – Fatores e níveis

Fator	**Nível**
Controlador SDN	Ryu, Floodlight e POX
Quantidade de nós IoT	26, 52 e 104
Repetições	100
Pacotes *Ping*	500
Tamanho do pacote *Ping*	9999 *bytes*
Tráfego de aplicação	MQTT

Fonte: Próprio autor (2019).

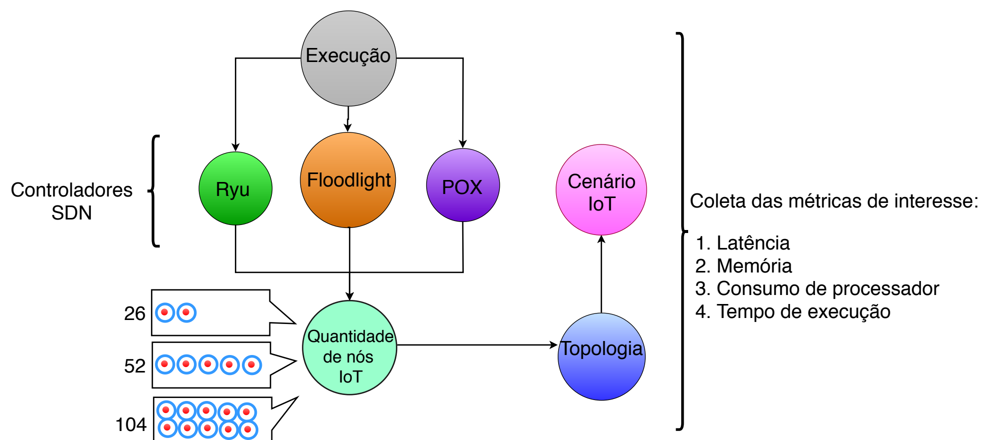
Definidos os fatores e os níveis, o modelo de automatização dos experimentos [1] foi estruturado como apresentado na Figura 22, em que apresentam-se os passos que os *scripts* percorrem para a coleta das métricas de interesse. O controlador Ryu será utilizado para exemplificar o funcionamento do modelo de automatização dos experimentos da Figura 22. Quando o *script* principal é executado, o controlador Ryu é selecionado e então verifica-se qual a quantidade de nós IoT escolhida para aquele experimento. A primeira quantidade são 26 nós IoT, o próximo passo é a passagem de todas as informações de configurações até o momento realizadas (nome do controlador, ip do controlador e quantidade de nós IoT) para o *script* de criação da topologia. O *script* de criação da topologia executa o programa em *python* responsável pela criação e execução do cenário IoT, repassando as informações de configurações para este programa em *python*.

Cada cenário IoT será executado 100 vezes, tendo tráfego de aplicação MQTT em todo o período de execução do experimento. O tráfego de aplicação MQTT foi utilizado para simular o ambiente de *smart homes*. O ambiente é formado por nós que simulam lâmpadas e portas inteligentes. Para cada execução dos experimentos as 4 métricas de interesse serão coletadas (latência, memória, consumo de processador e tempo de execução).

[1] Todos os *scripts .sh* e os códigos em *Python* estão disponíveis para a comunidade no seguinte repositório do *GitHub*: https://github.com/RandelSouza/TCC.

Para a coleta da métrica latência utilizou-se o comando *ping6* que é a versão do *ping* para o IPv6. O *ping* utiliza o *Internet Control Message Protocol* (ICMP), para verificar a conectividade entre dois *hosts*, enviando pacotes ICMP *echo request* e aguardando os pacotes de resposta, ICMP *echo reply*. O tempo de ida e volta, do inglês *Round Trip Time* (RTT), vai ser o dado relevante para posterior criação dos gráficos de latência, serão criados gráficos para a latência média. A quantidade de pacotes *echo request* configurados para envio foram de 500 para todos os experimentos. Outros parâmetros configurados no comando *ping6* nos experimentos foram o intervalo de envio entre os pacotes ICMP *echo request*, atribuindo-se com o argumento -i o valor de 0.0 segundos. Já o tamanho do pacote foi configurado com a opção -s, sendo atribuído um valor de 9999 *bytes*.

Figura 22 – Modelo de automatização dos experimentos



Fonte: Próprio autor (2019).

Vale ressaltar que os dados da última linha resultantes do comando *ping6* empregado nos experimentos, e referentes ao RTT foram escolhidos para a geração dos gráficos de latência. O campo utilizado foi o *avg* (latência média).

A métrica memória foi coletada a partir do comando *free* -m, que possibilita a visualização dos dados sobre memória são distribuídos em 6 colunas. São elas:

1. *total*: representa a memória total instalada (*MemTotal* e *SwapTotal* presentes em */proc/meminfo*). Na Seção 4.1 relata-se que a quantidade de memória RAM definida para A VM foi de 2 *Gigabytes* e pelo resultado do comando *free* -m da Figura 36 percebe-se que a quantidade utilizável da VM é na verdade de 1,9 *Gigabytes*.
2. *used*: mostra a memória usada, é calculada seguindo a formula *total* - *free* - *buffers* - *cache*.

3. *free*: é a memória não utilizada (*MemFree* e *SwapFree* presentes em */proc/meminfo*).
4. *shared*: memória usada por *tmpfs* (para o compartilhamento de memória).
5. *buff/cache*: *buff* é a memória usada pelos *buffers* do *kernel* (informação dos *buffers* presentes em */proc/meminfo*), o *cache* como o acesso à memória RAM é mais rápido que o acesso ao disco, a implementação de um sistema de *cache* funciona para que a leitura frequente de dados do disco ocorra com melhor performance, através do uso de um espaço de *cache* de memória, também chamado de *Page Cache*. Logo *buff/cache*, representa a soma das memória de *buffers* e do *cache*.
6. *available*: é uma estimativa (uma conta interna) de quanta memória está disponível, sem o uso do espaço de *swap*, para ser utilizada por novas aplicações que se iniciem ou por aplicações em execução que necessitem de mais memória.

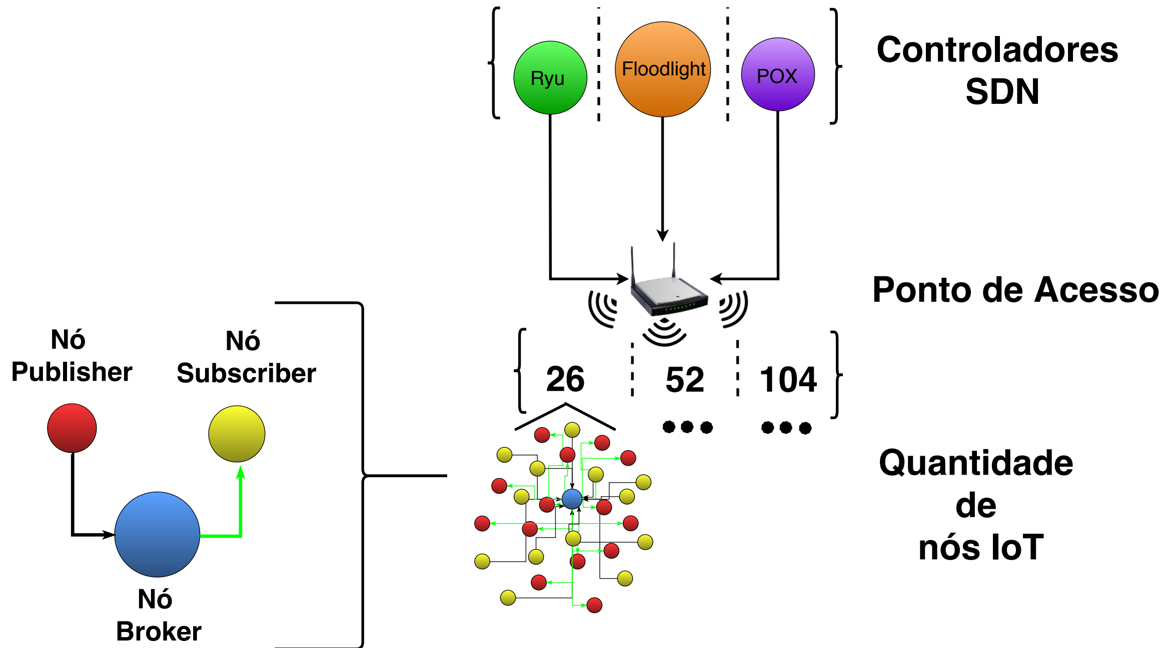
Os dados que foram escolhidos para serem trabalhados foram os da coluna *used* para a geração dos gráficos de memória utilizada. Por fim, para a coleta da métrica tempo de execução e também consumo de processador, usou-se o comando */usr/bin/time*. Com este comando foram coletados dados sobre o tempo de execução dos experimentos e também foi possível a coleta do percentual de CPU, do inglês *Central Processing Unit*, utilizado para cada experimento executado, para a métrica consumo de processador.

O argumento -f permite que a saída do comando *time* seja formatada, em que %E retornará os valores para o tempo de duração do comando em hora, minuto e segundo. Já o %P retornará o percentual de CPU que o comando usou em porcentagem. A opção %e também retorna o tempo decorrido da execução, no entanto, somente em segundos. Os dados escolhidos para serem utilizados na geração dos gráficos foram os relacionados as opções %P (percentual da CPU) e %e (Tempo decorrido).

4.3 Cenários

A conexão dos nós foi realizada utilizando-se do 6LoWPAN. O suporte ao 6LoWPAN pelo Mininet-WiFi foi possível graças ao módulo *fakelb*, presente em versões mais recentes do *kernel linux* (FONTES; ROTHENBERG, 2019). Segundo Fontes e Rothenberg (2019), mesmo que não exista uma implementação adequada do meio sem fio para o 6LoWPAN no Mininet-WiFi, o seu uso já é possível, por conta do módulo *fakelb*, e testes já podem ser realizados. Tendo em mente tudo isso, os nós foram configurados com 6LoWPAN. Permitindo a troca de informações entre os nós, pelo meio sem fio.

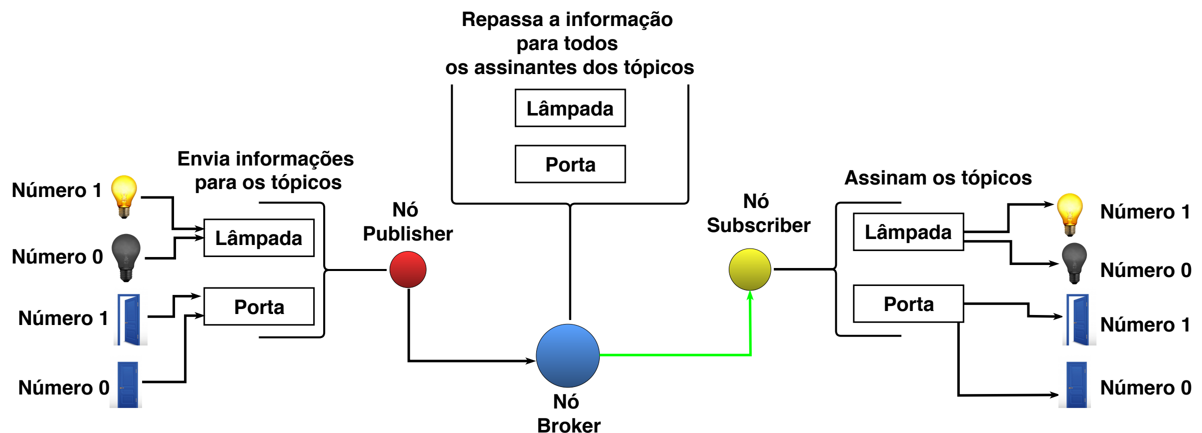
Figura 23 – Topologia utilizada



Fonte: Adaptado de Bedhief, Kassar e Aguili (2018).

A Figura 23 mostra a topologia utilizada neste trabalho. A topologia apresentada é a estrela que foi baseada na *single topology* de Bedhief, Kassar e Aguili (2018). A Figura 23 é uma representação condensada das configurações de todos os experimentos. É visto que o controlador encontra-se conectado a um ponto de acesso que por sua vez conecta-se a rede de nós configurada com 6LoWPAN. Cada rede IoT é formada por metade de nós *publishers* e a outra metade de nós *subscribers*.

Figura 24 – Configuração da arquitetura MQTT



Fonte: Próprio autor (2019).

A divisão é feita de acordo com a quantidade de nós definida. Cada rede tem um nó extra que é o *broker* da rede. Onde o *mosquitto*[2] fica executando e esperando as ações de publicação ou assinatura dos tópicos lâmpada e porta (Figura 24).

A Figura 24 mostra como o protocolo MQTT foi configurado nos cenários. O nó *publisher* envia informações para os tópicos lâmpada e porta. O número 1 é enviado para que a ação de ligar a lâmpada ou abrir a porta seja realizada, dependendo do tópico. Já o número 0 informa o inverso, lâmpada apagada ou porta fechada, dependendo do tópico escolhido.

O nó *broker* fica encarregado de repassar as informações advindas do nó *publisher* para todos os nós *subscribers* (assinantes) dos tópicos lâmpada e porta. Por fim, o nó *subscriber* assina os tópicos lâmpada e porta. Com isso toda vez que um *publisher* enviar informações para os tópicos lâmpada ou porta os *subscribers* receberão essas informações por meio do *broker*. O MQTT foi utilizado para que houvesse muito tráfego de aplicação nos cenários, com o intuito de tornar os cenários IoT mais realistas.

[2] https://mosquitto.org

5 RESULTADOS E DISCUSSÃO

Após a execução de todos os experimentos, foram gerados os gráficos de acordo com as métricas de interesse do trabalho. Desse modo, o intervalo de confiança empregado para a geração dos gráficos foi de 95%. Cada Seção deste Capítulo é constituída por um gráfico de pontos e um gráfico de barras. Cada gráfico de barras foi criado utilizando-se os mesmos dados do gráfico de pontos, no entanto o intuito do gráfico de barras é de dar uma perspectiva visual diferente, possibilitando uma comparação quantitativa mais aparente. O parágrafo seguinte informa as Seções compreendidas neste Capítulo.

A Seção 5.1 apresenta o resultado da latência média. A seção 5.2 apresenta o resultado de memória utilizada. Por fim, a seção 5.3 apresenta o resultado de consumo de processador dos experimentos e a Seção 5.4 apresenta o resultado do tempo de execução dos experimentos.

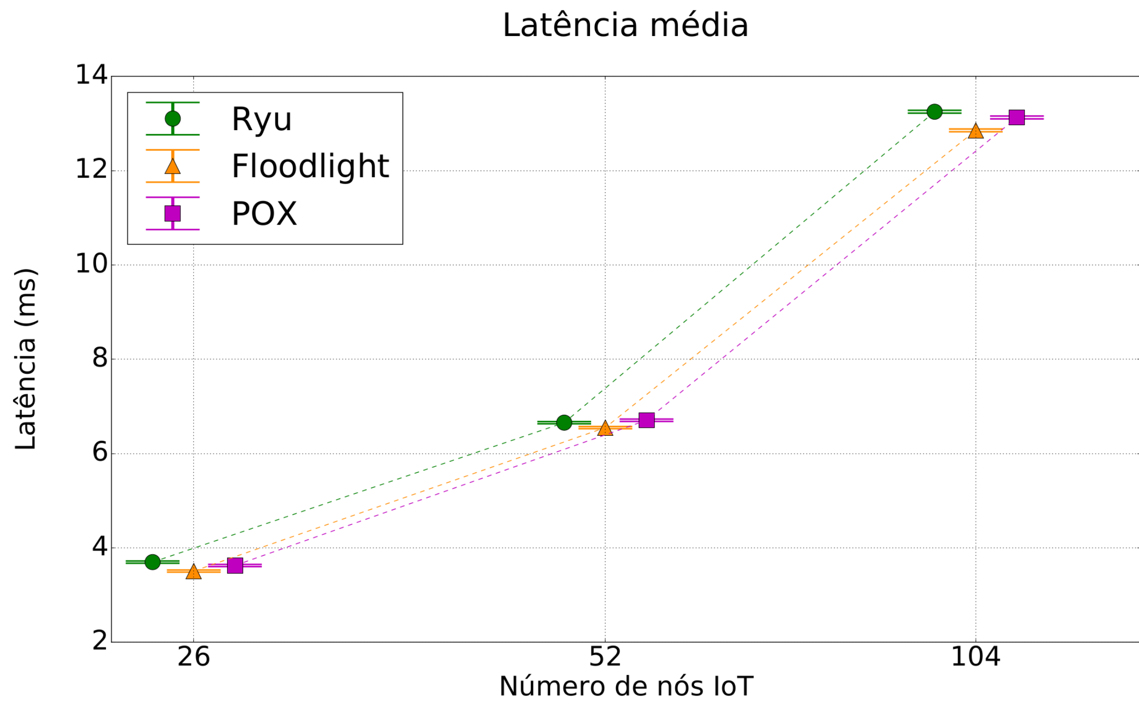
5.1 Latência média

A Figura 25 mostra o resultado do experimento de latência média. No cenário que apresentava 26 nós IoT, o controlador Floodlight apresentou uma latência média menor do que os demais controladores. O segundo menor valor de latência média foi o do controlador POX e o maior resultado foi o do controlador Ryu.

No cenário que continha 52 nós IoT (Figura 25), os resultados foram similares ao do cenário de 26 nós IoT, neste cenário, assim como no primeiro cenário, o controlador Floodlight sobressaiu em relação aos demais controladores ao apresentar a menor latência média. Já os controladores POX e Ryu apresentaram valores bem similares de latência média, no entanto o POX teve um resultado um pouco melhor do que o Ryu e o Ryu, por sua vez, teve o menor resultado entre os controladores comparados.

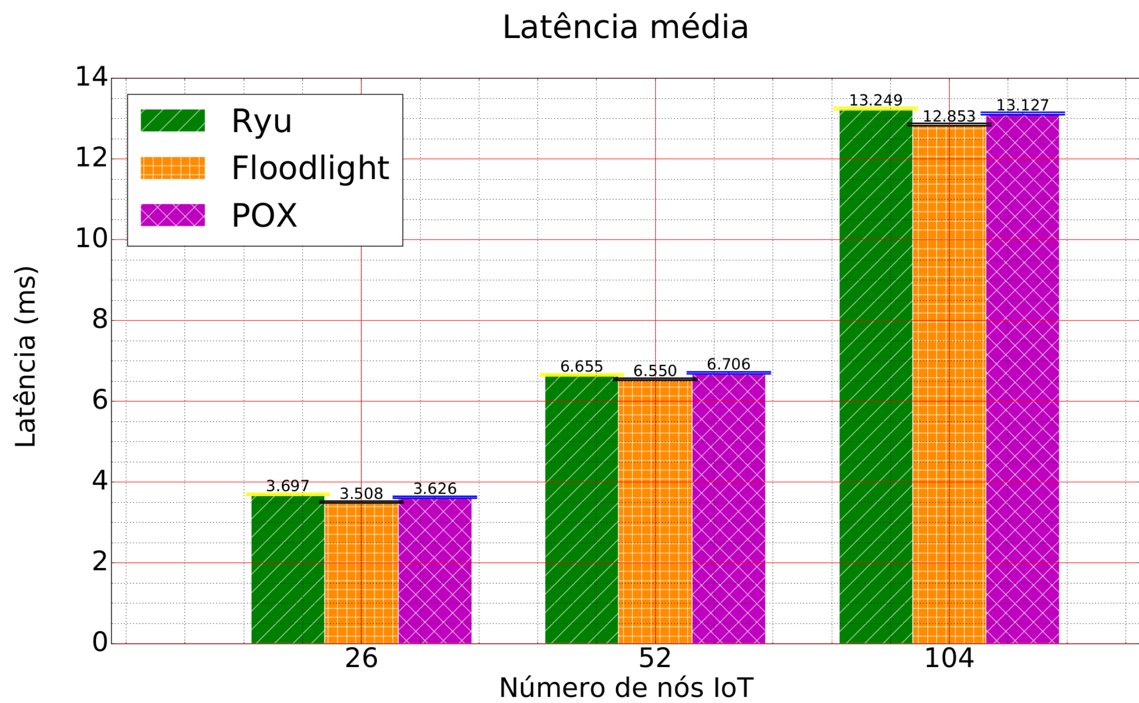
No último cenário (Figura 25), novamente o mesmo comportamento notado nos cenários anteriores persistiu. O controlador Floodlight foi bem melhor do que os outros controladores, o controlador POX foi o segundo melhor e o último controlador, com o pior resultado, foi o Ryu. A Figura 26 mostra o gráfico de barras da métrica de latência média.

Figura 25 – Resultado do experimento de latência média



Fonte: Próprio autor (2019).

Figura 26 – Latência média



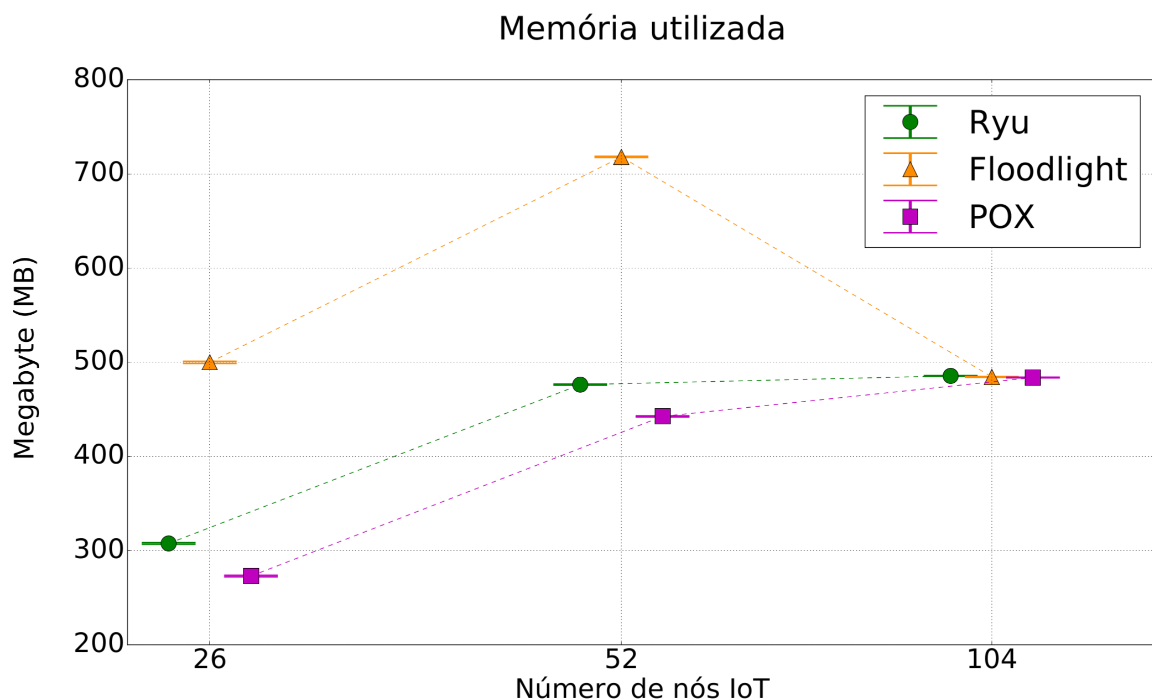
Fonte: Próprio autor (2019).

5.2 Memória utilizada

A Figura 27 mostra o resultado da coleta da métrica de memória utilizada dos experimentos. No cenário que apresentava 26 nós IoT o controlador POX foi o que obteve a menor utilização de memoria sendo bem melhor do que os controladores Ryu e Floodlight. O controlador Ryu por sua vez, apresentou uma utilização de memória bem menor do que o controlador Floodlight. Já o Floodlight teve a maior utilização de memória no cenário contendo 26 nós IoT.

Ainda na Figura 27, agora no cenário contendo 52 nós nós IoT, os resultados mantiveram-se semelhates ao do cenário com 26 nós IoT. O controlador POX foi o que obteve a menor utilização de memória em comparação com o controlador Ryu e o controlador Floodlight. O controlador Ryu, assim como no cenário com 26 nós IoT, apresentou utilização de memória menor do que o Floodlight. O controlador Floodlight teve a maior utilização de memória entre os 3 controladores.

Figura 27 – Resultado do experimento de memória utilizada

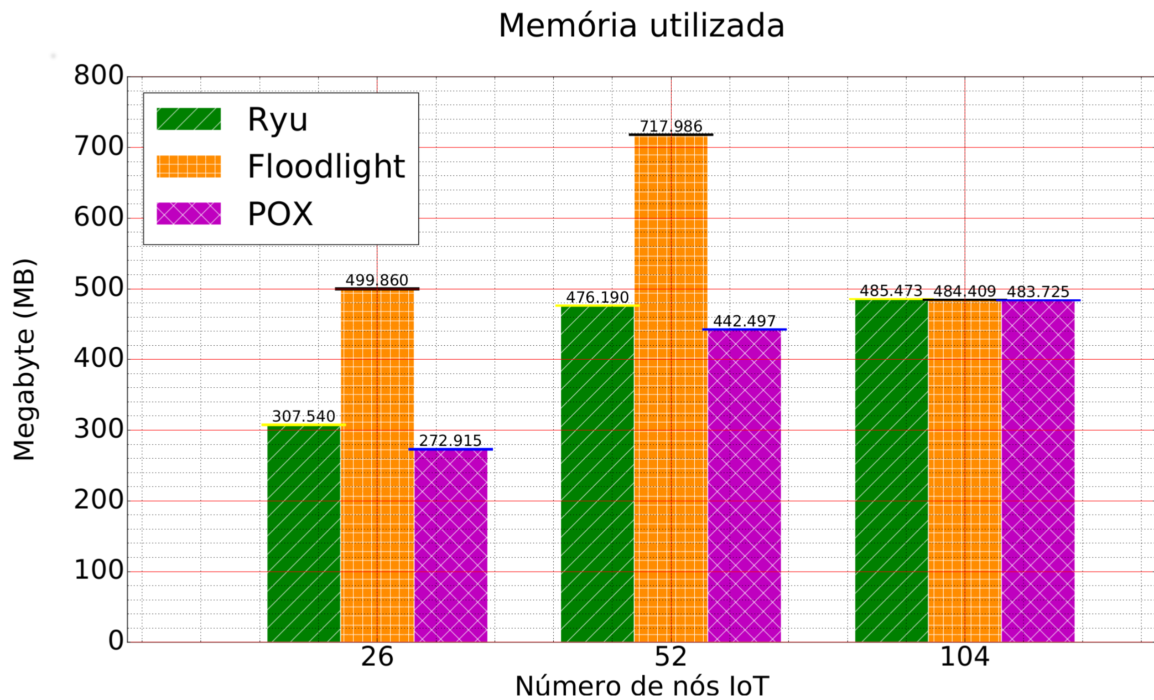


Fonte: Próprio autor (2019).

Finalmente, no cenário com 104 nós IoT (Figura 27), os resultados foram muito idênticos, mesmo assim pelos dados da Figura 28 o POX teve 483.725 MB, o Floodlight

apresentou 484.409 MB e o Ryu teve 485.473 MB sendo estas respectivamente da menor para a maior utilização.

Figura 28 – Memória utilizada



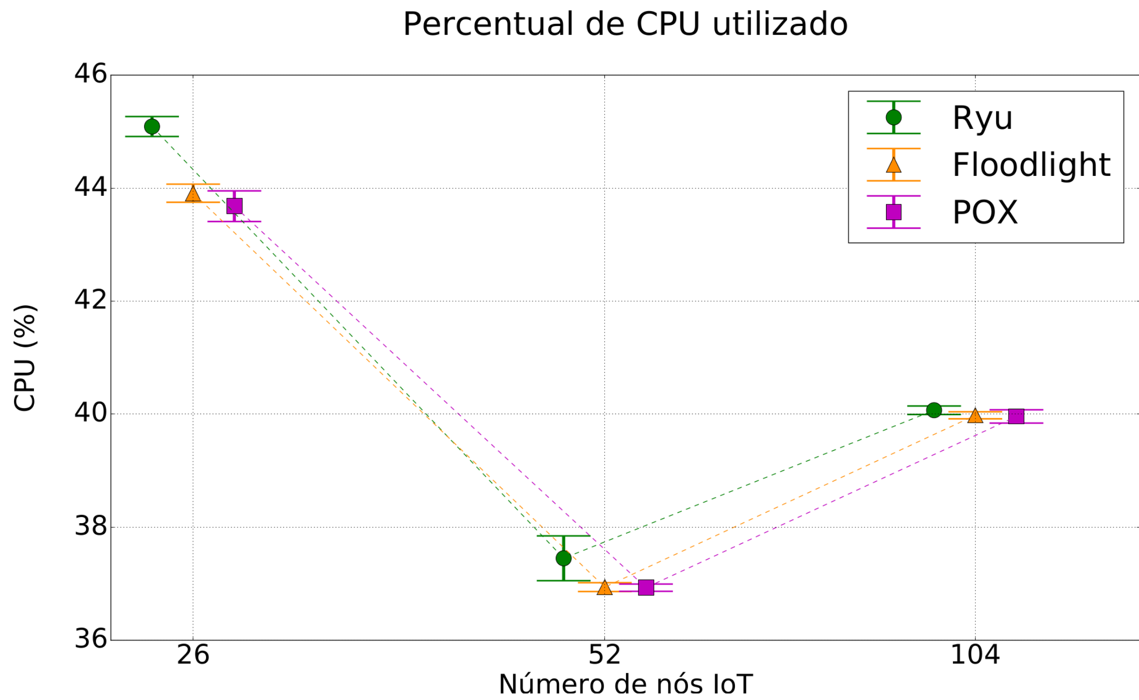
Fonte: Próprio autor (2019).

5.3 Consumo de processador

A Figura 29 mostra o consumo de processamento em forma de percentual de utilização da CPU. No cenário que continha 26 nós IoT o controlador POX teve o resultado levemente melhor do que o Floodlight, isto é, obteve o menor percentual de utilização da CPU. No entanto, suas margens de erro tocam-se com a do controlador Floodlight. O controlador Ryu apresentou o maior percentual de utilização. Por fim, o controlador Floodlight apresentou um índice de utilização de CPU intermediário em comparação aos demais.

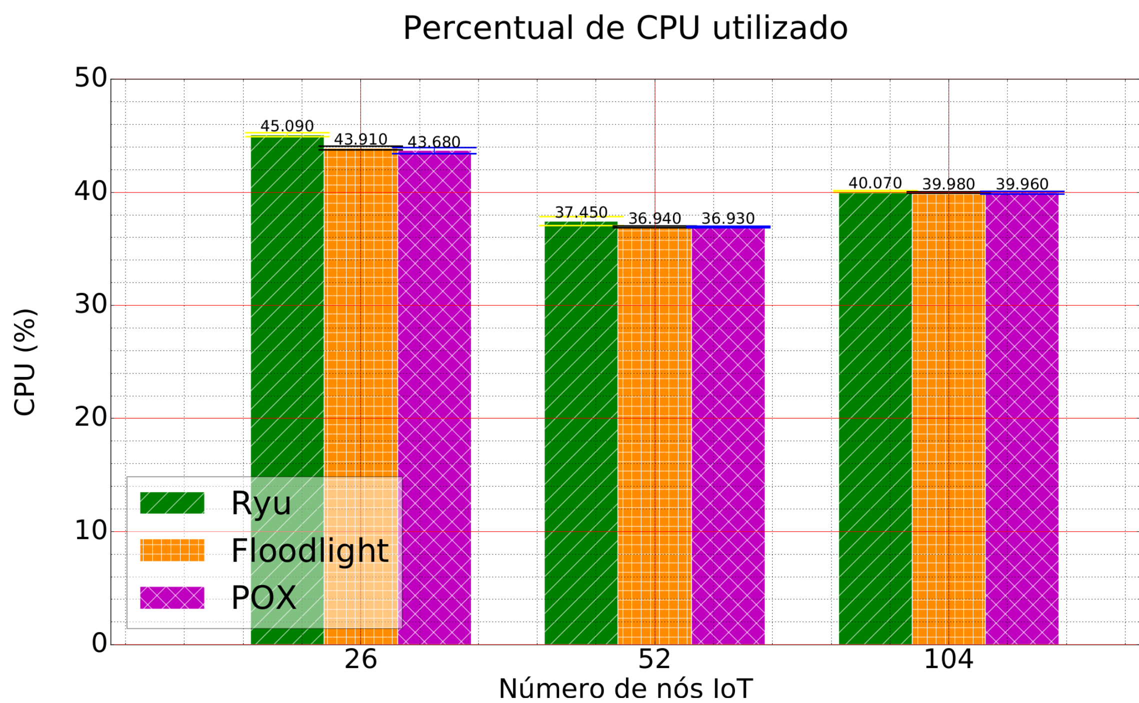
No cenário com 52 nós IoT (Figura 29), os resultados foram bem parecidos ficando os 3 controladores empatados tecnicamente. O mesmo ocorreu no cenário com 104 nós IoT os controladores tiveram resultados bem parecidos e também ficaram empatados tecnicamente por conta da sobreposição das margens de erro.

Figura 29 – Resultado do experimento do percentual de CPU utilizado



Fonte: Próprio autor (2019).

Figura 30 – Percentual de CPU utilizado

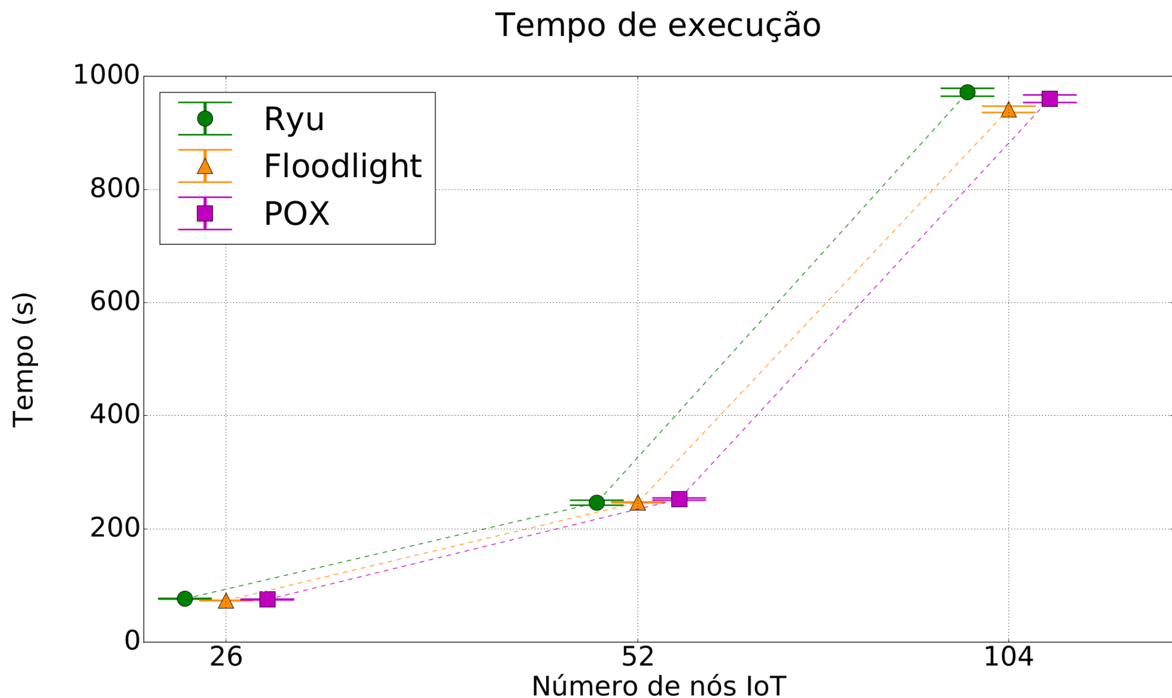


Fonte: Próprio autor (2019).

5.4 Tempo de execução

A Figura 31 mostra o resultado da coleta da métrica de tempo de execução dos experimentos. No cenário contendo 26 nós IoT, o tempo de execução dos controladores POX e Ryu foram bem parecidos, no entanto o controlador Floodlight foi o que apresentou um tempo de execução menor em relação ao POX e ao Ryu.

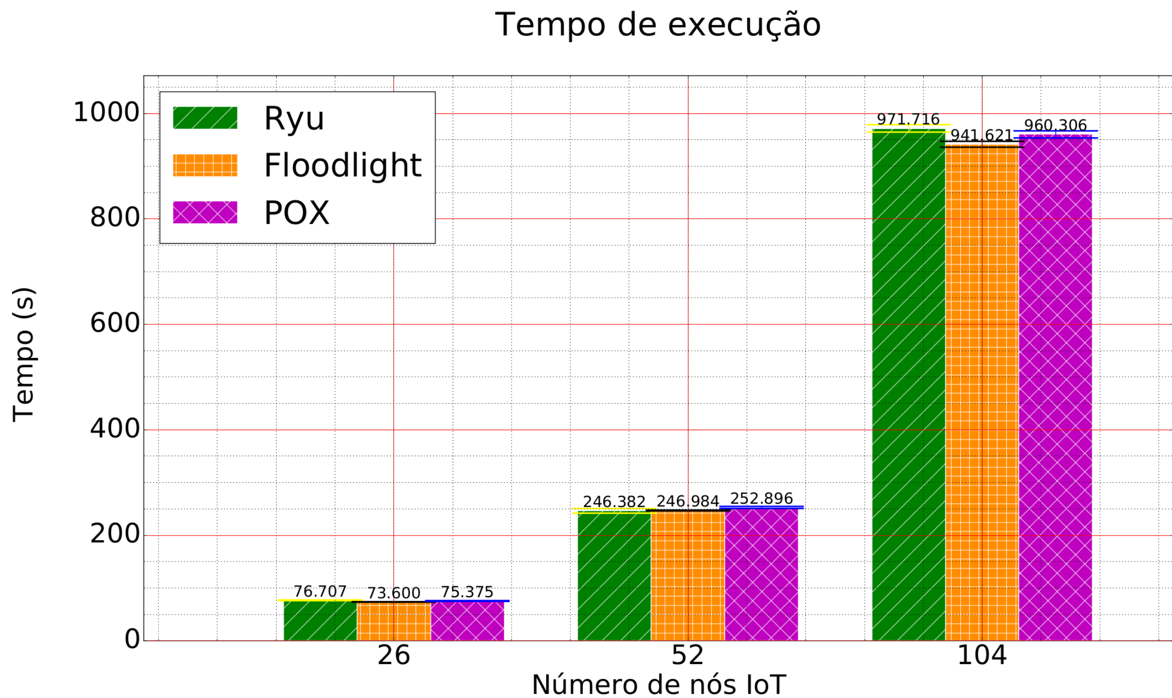
Figura 31 – Resultado do experimento do tempo de execução



Fonte: Próprio autor (2019).

No cenário com 52 nós IoT (Figura 31), o controlador POX apresentou o pior resultado. Já os controladores Ryu e POX tiveram valores bem próximos. No último cenário com 104 nós IoT (Figura 31), foi o cenário em que mais houve diferença, tendo como menor tempo de execução o do controlador Floodlight. O Segundo menor tempo de execução foi o do controlador POX e o maior tempo foi o do controlador Ryu. A Figura 32 mostra os gráficos de barras referentes ao tempo de execução dos experimentos. Pode-se supor que no cenário com 104 nós o Floodlight teve o tempo de execução menor por ser desenvolvido em Java. Tendo assim um desempenho melhor do que Python (linguagem de desenvolvimento do Ryu e POX).

Figura 32 – Tempo de execução



Fonte: Próprio autor (2019).

A Figura 32 mostra os dados do tempo real decorrido relacionado ao processo. Por este gráfico é possível saber quanto tempo realmente durou cada experimento. Para isso, foi retirada a média simples de cada cenário (soma de todos os tempos em segundos e divididos pelo número de controladores). Então o valor da média foi dividido por 60 para obter o valor em minutos. Com isso, no cenário com 26 nós IoT o tempo real de execução de cada experimento foi de 1 minuto e 25 segundos. Nó cenário com 52 nós IoT o tempo foi de 4 minutos e 14 segundos. No último cenário de 104 nós IoT o tempo de execução foi de 16 minutos e 36 segundos. O Quadro 4 apresenta o tempo real em minutos dos experimentos.

Quadro 4 – Tempo de execução real dos experimentos

Quantidade de nós IoT	**Soma (segundos)**	**Média (segundos)**	**Tempo real (minutos)**
26	225.682	75.2273333	1:25
52	746.262	248.754	4:14
104	2873.643	957.881	16:36

Fonte: Próprio autor (2019).

6 CONSIDERAÇÕES FINAIS

IoT é um paradigma promissor ao possibilitar que objetos comuns conectem-se à Internet, mas que apresenta algumas limitações como capacidade de memória e de processamento. A partir disso este trabalho utilizou de outro paradigma denominado SDN a fim de manter uma maior flexibilidade da rede IoT, ao separar o plano de dados e o de controle, mantendo uma visão global da rede. Após a definição dos fatores e níveis foi criado e implementado o modelo de automatização dos experimentos. Permitindo que as métricas de interesse fossem coletadas. Todos os códigos[1] deste trabalho estão disponíveis para a comunidade.

Ao analisar os resultados obtidos, percebe-se que o controlador Floodlight foi o que apresentou a menor latência média com até 104 nós IoT. Já em relação à memória utilizada o controlador POX sobressaiu com até 52 nós IoT.

O consumo de processador mostrou que as porcentagem nos 3 cenários foram bastante parecidas, exceto no primeiro cenário com 26 nós IoT, onde o Ryu apresentou o maior percentual de CPU utilizado. Por fim, o tempo de execução dos experimentos mostrou que com até 52 nós o tempo foi quase que igual, no entanto com 104 nós o controlador Floodlight sobressaiu com o menor tempo entre os demais.

O Quadro 5 contém a síntese dos resultados obtidos. Para criar o Quadro 5 foi empregada a lógica de pontos. Para cada métrica em questão nos cenários foi atribuído 1 ponto em cada cenário que o controlador sobressaiu. Por exemplo, o controlador Floodlight foi melhor do que os demais em relação a métrica latência média com 26, 52 e 104 nós. Por isso o controlador teve 3 pontos na métrica em questão. A lógica descrita foi feita para as demais métricas.

Quadro 5 – Comparação dos resultados

	POX	**Ryu**	**Floodlight**
Latência média	*	*	3
Memória utilizada	2	*	*
Percentual de CPU utilizado	*	*	*
Tempo de execução	*	*	1
Total	2	0	4

Fonte: Próprio autor (2019).

[1] Os códigos deste trabalho então disponíveis na seguinte página do *Github*: https://github.com/RandelSouza/TCC.

Como trabalhos futuros pode-se aumentar o número de nós IoT, adicionar tráfego de aplicação CoAP nos cenários e adicionar outros controladores SDN como o OpenDaylight. Além de adicionar modelos de perda e utilizar o consumo energético como uma nova métrica.

Em continuação, pode-se modificar a topologia. Adicionar um ponto de acesso para cada *smart home* e adicionar *switches* para conectar o controlador às *smart homes*. Outra sugestão seria a modificação dos *scripts*, para que, o tempo de execução dos experimentos possa ser fixo, por exemplo, 5 minutos para cada experimento. Com o tempo de execução fixo pode-se adicionar nós de maneira dinâmica, a fim de gerar mais tráfego para o controlador SDN. Por fim, pode-se adicionar nós configurados com o protocolo RPL.

REFERÊNCIAS

ABDELFADEEL, K. **A Service Discovery Framework in IPv6 over Low-power Wireless Personal Area Network**. 115 p. Tese (Doutorado), 04 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**, IEEE, v. 17, n. 4, p. 2347–2376, 2015.

ANTHRAPER, J. J.; KOTAK, J. Security, privacy and forensic concern of mqtt protocol (march 19, 2019). In: **Proceedings of International Conference on Sustainable Computing in Science, Technology and Management (SUSCOM-2019)**. [S.l.: s.n.], 2019.

AZZOLA, F. **CoAP Protocol**: Step-by-step guide. 2018. Disponível em: https://dzone.com/articles/coap-protocol-step-by-step-guide. Acesso em: 16 maio. 2019.

BADOTRA, S.; SINGH, J. Open daylight as a controller for software defined networking. **International Journal of Advanced Computer Research**, v. 8, 05 2017.

BEDHIEF, I.; KASSAR, M.; AGUILI, T. Sdn-based architecture challenging the iot heterogeneity. In: IEEE. **Smart Cloud Networks & Systems (SCNS)**. [S.l.], 2016. p. 1–3.

Bedhief, I.; Kassar, M.; Aguili, T. From evaluating to enabling sdn for the internet of things. In: **2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)**. [S.l.: s.n.], 2018. p. 1–8. ISSN 2161-5322.

BHOLEBAWA, I. Z.; DALAL, U. D. Performance analysis of sdn/openflow controllers: Pox versus floodlight. **Wireless Personal Communications**, Springer, v. 98, n. 2, p. 1679–1699, 2018.

BONDKOVSKII, A.; KEENEY, J.; MEER, S. van der; WEBER, S. Qualitative comparison of open-source sdn controllers. In: IEEE. **Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP**. [S.l.], 2016. p. 889–894.

BORGIA, E. The internet of things vision: Key features, applications and open issues. **Computer Communications**, Elsevier, v. 54, p. 1–31, 2014.

Bormann, C.; Castellani, A. P.; Shelby, Z. Coap: An application protocol for billions of tiny internet nodes. **IEEE Internet Computing**, v. 16, n. 2, p. 62–67, March 2012. ISSN 1089-7801.

BRAGA, A. B. et al. **Análise de desempenho dos protocolos CTP e RPL em ambiente móvel**. 58 p. Monografia (Trabalho de Conclusão de Curso) — Faculdade de Engenharia Elétrica, Universidade Federal de Uberlândia, Patos de Minas, 2018.

CAI, Z.; COX, A. L.; NG, T. **Maestro: a system for scalable openflow control**. [S.l.], 2010. Disponível em: https://hdl.handle.net/1911/96391. Acesso em: 20 abr. 2019.

CARAGUAY, Á. L. V.; PERAL, A. B.; LÓPEZ, L. I. B.; VILLALBA, L. J. G. Sdn: Evolution and opportunities in the development iot applications. **International Journal of Distributed Sensor Networks**, SAGE PublicationsSage UK: London, England, 2014.

ERICKSON, D. **Beacon OpenFlow Controller**. 2013. Disponível em: https://openflow.stanford.edu/display/Beacon/Home.html. Acesso em: 15 fev. 2019.

ERICKSON, D. The Beacon OpenFlow Controller. In: ACM. **HotSDN**. [S.l.], 2013.

FONTES, R. d. R.; ROTHENBERG, C. E. Mininet-wifi: A platform for hybrid physical-virtual software-defined wireless networking research. In: ACM. **Proceedings of the 2016 ACM SIGCOMM Conference**. [S.l.], 2016. p. 607–608.

FONTES, R. dos R.; ROTHENBERG, C. E. Mininet-wifi: Plataforma de emulação para redes sem fio definidas por software. In: SBC. **Anais Estendidos do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. [S.l.], 2019. p. 201–208.

Fontes, R. R.; Afzal, S.; Brito, S. H. B.; Santos, M. A. S.; Rothenberg, C. E. Mininet-wifi: Emulating software-defined wireless networks. In: **2015 11th International Conference on Network and Service Management (CNSM)**. [S.l.: s.n.], 2015. p. 384–389.

FRIGIERI, E. P.; MAZZER, D.; PARREIRA, L. M2m protocols for constrained environments in the context of iot: A comparison of approaches. In: **International Telecommunications Symposium**. [S.l.: s.n.], 2015. p. 5.

GUDE, N.; KOPONEN, T.; PETTIT, J.; PFAFF, B.; CASADO, M.; MCKEOWN, N.; SHENKER, S. Nox: towards an operating system for networks. **ACM SIGCOMM Computer Communication Review**, ACM, v. 38, n. 3, p. 105–110, 2008.

HOANG, D. B.; PHAM, M. On software-defined networking and the design of sdn controllers. In: IEEE. **2015 6th International Conference on the Network of the Future (NOF)**. [S.l.], 2015. p. 1–3.

HU, F.; HAO, Q.; BAO, K. A survey on software-defined network and openflow: From concept to implementation. **IEEE Communications Surveys & Tutorials**, IEEE, v. 16, n. 4, p. 2181–2206, 2014.

IGLESIAS-URKIA, M.; ORIVE, A.; URBIETA, A. Analysis of coap implementations for industrial internet of things: A survey. **Procedia Computer Science**, Elsevier, v. 109, p. 188–195, 2017.

IOVA, O.-T. **Standards optimization and network lifetime maximization for wireless sensor networks in the Internet of things**. Tese (Doutorado) — Strasbourg, 2014.

ISLAM, S. R.; KWAK, D.; KABIR, M. H.; HOSSAIN, M.; KWAK, K.-S. The internet of things for health care: a comprehensive survey. **IEEE Access**, IEEE, v. 3, p. 678–708, 2015.

JARARWEH, Y.; AL-AYYOUB, M.; BENKHELIFA, E.; VOUK, M.; RINDOS, A. et al. Sdiot: a software defined based internet of things framework. **Journal of Ambient Intelligence and Humanized Computing**, Springer, v. 6, n. 4, p. 453–461, 2015.

KARAGIANNIS, V.; CHATZIMISIOS, P.; VáZQUEZ-GALLEGO, F.; ALONSO-ZARATE, J. A survey on application layer protocols for the internet of things. **Trans. IoT Cloud Comput.**, v. 3, p. 11–17, 01 2015.

KARAKUS, M.; DURRESI, A. A survey: Control plane scalability issues and approaches in software-defined networking (sdn). **Computer Networks**, 2017.

KAUR, S.; SINGH, J.; GHUMMAN, N. S. Network programmability using pox controller. In: **ICCCS International Conference on Communication, Computing & Systems, IEEE**. [S.l.: s.n.], 2014. p. 138.

Khondoker, R.; Zaalouk, A.; Marx, R.; Bayarou, K. Feature-based comparison and selection of software defined networking (sdn) controllers. In: **2014 World Congress on Computer Applications and Information Systems (WCCAIS)**. [S.l.: s.n.], 2014. p. 1–7. ISSN null.

KIM, H.; FEAMSTER, N. Improving network management with software defined networking. **IEEE Communications Magazine**, Citeseer, v. 51, n. 2, p. 114–119, 2013.

KIM, H.-s.; KO, J.; CULLER, D.; PAEK, J. Challenging the ipv6 routing protocol for low-power and lossy networks (rpl): A survey. **IEEE Communications Surveys & Tutorials**, v. 19, p. 2502–2525, 09 2017.

KREUTZ, D.; RAMOS, F. M.; VERISSIMO, P. E.; ROTHENBERG, C. E.; AZODOLMOLKY, S.; UHLIG, S. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, IEEE, v. 103, n. 1, p. 14–76, 2015.

KUBO, R.; FUJITA, T.; AGAWA, Y.; SUZUKI, H. Ryu sdn framework: Open-source sdn platform software. **NTT Technical Review**, v. 12, n. 8, p. 1–5, 2014.

LAISSAOUI, C.; IDBOUFKER, N.; ELASSALI, R.; BAAMRANI, K. E. A measurement of the response times of various openflow/sdn controllers with cbench. In: IEEE. **Computer Systems and Applications (AICCSA), 2015 IEEE/ACS 12th International Conference of**. [S.l.], 2015. p. 1–2.

Lamkimel, M.; Naja, N.; Jamali, A.; Yahyaoui, A. The internet of things: Overview of the essential elements and the new enabling technology 6lowpan. In: **2018 IEEE International Conference on Technology Management, Operations and Decisions (ICTMOD)**. [S.l.: s.n.], 2018. p. 142–147. ISSN 2159-5119.

MACHADO, M. T. G. P. **Controlador OpenDaylight**. 2018. Disponível em: https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2018_2/opendaylight/. Acesso em: 19 mar. 2019.

MCCAULEY KYRIAKOS ZARIFIS, A. T. S. S. M. **The POX network software platform**. 2009. Disponível em: https://github.com/noxrepo/pox. Acesso em: 12 mar. 2019.

MCKEOWN, N.; ANDERSON, T.; BALAKRISHNAN, H.; PARULKAR, G.; PETERSON, L.; REXFORD, J.; SHENKER, S.; TURNER, J. Openflow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, ACM, v. 38, n. 2, p. 69–74, 2008.

MULLIGAN, G. The 6lowpan architecture. In: ACM. **Proceedings of the 4th workshop on Embedded networked sensors**. [S.l.], 2007. p. 78–82.

NASCIMENTO, R. S. d. **Inspeção de transportadores de correia: arquitetura integrada para uma plataforma de inspeção com uso de VANTs.** 188 p. Dissertação (Mestrado) — Mestrado Profissional em Instrumentação, Controle e Automação de Processos de Mineração, Universidade Federal De Ouro Preto, Ouro Preto, Minas Gerais, Brasil, 2018.

NETWORKS, A. B. S. **Project Floodlight**: Open source software for building software-defined networks. 2019. Disponível em: http://www.projectfloodlight.org/floodlight/. Acesso em: 14 mar. 2019.

NITTI, M.; PILLONI, V.; COLISTRA, G.; ATZORI, L. The virtual object as a major element of the internet of things: a survey. **IEEE Communications Surveys & Tutorials**, IEEE, v. 18, n. 2, p. 1228–1240, 2016.

NUNES, B. A. A.; MENDONCA, M.; NGUYEN, X.-N.; OBRACZKA, K.; TURLETTI, T. A survey of software-defined networking: Past, present, and future of programmable networks. **IEEE Communications Surveys & Tutorials**, IEEE, v. 16, n. 3, p. 1617–1634, 2014.

OSRG. **Framework using OpenFlow 1.3**. 2014. Disponível em: http://osrg.github.io/ryu-book/en/Ryubook.pdf. Acesso em: 20 mar. 2019.

OSRG. **Ryu component-based software defined networking framework**. 2014. Disponível em: https://osrg.github.io/ryu/. Acesso em: 13 mar. 2019.

PRIYADARSINI, M.; BERA, P.; BHAMPAL, R. Performance analysis of software defined network controller architecture—a simulation based survey. In: IEEE. **Wireless Communications, Signal Processing and Networking (WiSPNET), 2017 International Conference on**. [S.l.], 2017. p. 1929–1935.

RAO, S. **SDN Series Part Two**: Trema, a framework for developing openflow controllers in ruby and c. 2014. Disponível em: https://thenewstack.io/sdn-series-part-ii-trema-a-framework-for-developing-openflow-controllers-in-ruby-and-c/. Acesso em: 4 abr. 2019.

RASTOGI, A.; BAIS, A. Comparative analysis of software defined networking (sdn) controllers—in terms of traffic handling capabilities. In: IEEE. **Multi-Topic Conference (INMIC), 2016 19th International**. [S.l.], 2016. p. 1–6.

Ruengittinun, S.; Paisalwongcharoen, J.; Watcharajindasakul, C. Iot solution for bad habit of car security. In: **2017 10th International Conference on Ubi-media Computing and Workshops (Ubi-Media)**. [S.l.: s.n.], 2017. p. 1–4.

Salman, L.; Salman, S.; Jahangirian, S.; Abraham, M.; German, F.; Blair, C.; Krenz, P. Energy efficient iot-based smart home. In: **2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)**. [S.l.: s.n.], 2016. p. 526–529.

Salman, O.; Elhajj, I. H.; Kayssi, A.; Chehab, A. Sdn controllers: A comparative study. In: **2016 18th Mediterranean Electrotechnical Conference (MELECON)**. [S.l.: s.n.], 2016. p. 1–6. ISSN 2158-8481.

SALMAN, T.; JAIN, R. A survey of protocols and standards for internet of things. **Advanced Computing and Communications**, p. 20, 2019.

SANTOS, B. P.; SILVA, L.; CELES, C.; BORGES, J. B.; NETO, B. S. P.; VIEIRA, M. A. M.; VIEIRA, L. F. M.; GOUSSEVSKAIA, O. N.; LOUREIRO, A. Internet das coisas: da teoria à prática. **Minicursos SBRC-Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**, 2016.

SHELBY, Z.; HARTKE, C. B. K. **The Constrained Application Protocol (CoAP)**. 2014. Disponível em: https://tools.ietf.org/html/rfc7252. Acesso em: 15 maio. 2019.

SHIMONISHI Y. TAKAMYA, K. S. Y. C. K. S. H. **Full-Stack OpenFlow Framework in Ruby**. 2018. Disponível em: https://github.com/trema/trema. Acesso em: 4 abr. 2019.

SOOD, K.; YU, S.; XIANG, Y. Software-defined wireless networking opportunities and challenges for internet-of-things: A review. **IEEE Internet of Things Journal**, IEEE, v. 3, n. 4, p. 453–463, 2016.

TAYYABA, S. K.; SHAH, M. A.; KHAN, N. S. A.; ASIM, Y.; NAEEM, W.; KAMRAN, M. Software-defined networks (sdns) and internet of things (iots): A qualitative prediction for 2020. **network**, v. 7, n. 11, 2016.

UNIVERSITY, R. **Rice University Project Maestro**. 2019. Disponível em: https://www.sdxcentral.com/projects/maestro/. Acesso em: 12 mar. 2019.

WINTER, T.; THUBERT, A. B. P.; HUI, R. K. J. W.; LEVIS, K. P. P.; STRUIK, J. V. R.; ALEXANDER, R. K. **RPL**: Ipv6 routing protocol for low-power and lossy networks. 2012. Disponível em: https://tools.ietf.org/html/rfc6550. Acesso em: 23 maio. 2019.

WU, D.; ARKHIPOV, D. I.; ASMARE, E.; QIN, Z.; MCCANN, J. A. Ubiflow: Mobility management in urban-scale software defined iot. In: IEEE. **Computer Communications (INFOCOM), 2015 IEEE Conference on**. [S.l.], 2015. p. 208–216.

Wukkadada, B.; Wankhede, K.; Nambiar, R.; Nair, A. Comparison with http and mqtt in internet of things (iot). In: **2018 International Conference on Inventive Research in Computing Applications (ICIRCA)**. [S.l.: s.n.], 2018. p. 249–253.

XIA, W.; WEN, Y.; FOH, C. H.; NIYATO, D.; XIE, H. A survey on software-defined networking. **IEEE Communications Surveys & Tutorials**, IEEE, v. 17, n. 1, p. 27–51, 2015.

XU, L. D.; HE, W.; LI, S. Internet of things in industries: A survey. **IEEE Transactions on industrial informatics**, IEEE, v. 10, n. 4, p. 2233–2243, 2014.

YASSEIN, M. B.; SHATNAWI, M. Q.; ALJWARNEH, S.; AL-HATMI, R. Internet of things: Survey and open issues of mqtt protocol. In: IEEE. **2017 International Conference on Engineering & MIS (ICEMIS)**. [S.l.], 2017. p. 1–6. Disponível em: https://ieeexplore.ieee.org/document/8273112. Acesso em: 23 maio. 2019.

ZOLERTIA. **WHAT IS 6LOWPAN AND WHY SHOULD I TRY IT IN MY IOT PROJECT?** 2019. Disponível em: https://zolertia.io/6lowpan-iot-protocol/. Acesso em: 22 maio. 2019.

APÊNDICE A – COMANDOS

Este APÊNDICE mostra alguns comandos úteis que foram utilizados para visualizar as informações da VM. O APÊNDICE B informa o processo de instalação do emulador Mininet-WiFi. O APÊNDICE C explica o processo de instalação dos controladores SDN. Os APÊNDICES D, E e F apresentam os testes dos controladores Floodlight, POX e Ryu, respectivamente. Por fim, os APÊNDICES G, H, I e J mostram os gráficos das métricas utilizadas neste trabalho que são: latência média, memória utilizada, consumo de processador e tempo de execução dos experimentos, respectivamente.

O programa *fdisk* é usado para a criação e manipulação de tabelas de partição. A execução do comando com o argumento -l "*sudo fdisk -l*" lista as tabelas de partição descrevendo varias informações dentre elas a capacidade interna de armazenamento. Em continuação, ainda na Figura 33 é possível visualizar o particionamento do disco /dev/xvda em 3 outras partições, sendo denominadas de /dev/xvda1 que apresenta 37,9 *Gigabytes*, /dev/xvda2 e /dev/xvda5 apresentando cada uma 2,1 *Gigabytes*.

Figura 33 – Espaço de armazenamento interno da VM utilizando o comando fdisk

```
randel@ipatingao:~$ sudo fdisk -l
Disk /dev/xvda: 40 GiB, 42949672960 bytes, 83886080 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xbe60f63b

Dispositivo Inicializar    Start      Fim  Setores  Size Id Tipo
/dev/xvda1                  2048 79448063 79446016 37,9G 83 Linux
/dev/xvda2              79450110 83884031  4433922  2,1G  5 Estendida
/dev/xvda5              79450112 83884031  4433920  2,1G 82 Linux swap / Solaris
```

Fonte: Próprio autor (2019).

Outro comando que foi utilizado para visualizar a capacidade de armazenamento interno da VM foi o "df -h", que exibe a quantidade de espaço em disco disponível e a opção -h permite que os valores sejam legíveis para humanos, isto é, mostra os valores em potência de 1024. No entanto este comando não mostrou todas as partições e nem a capacidade total do disco (Figura 34).

Figura 34 – Espaço de armazenamento interno da VM utilizando o comando df

```
randel@ipatingao:~$ df -h
Sist. Arq.      Tam. Usado Disp. Uso% Montado em
udev            963M     0  963M   0% /dev
tmpfs           199M   21M  178M  11% /run
/dev/xvda1       38G   23G   13G  64% /
tmpfs           992M     0  992M   0% /dev/shm
tmpfs           5,0M     0  5,0M   0% /run/lock
tmpfs           992M     0  992M   0% /sys/fs/cgroup
tmpfs           199M     0  199M   0% /run/user/1001
```

Fonte: Próprio autor (2019).

A Figura 35 mostra o final da execução de um comando *ping6* no experimento do controlador Ryu com 26 nós IoT.

Figura 35 – Comando *ping6*

```
877 bytes from 2001::1: icmp_seq=9995 ttl=64 time=0.040 ms
877 bytes from 2001::1: icmp_seq=9996 ttl=64 time=0.050 ms
877 bytes from 2001::1: icmp_seq=9997 ttl=64 time=0.029 ms
877 bytes from 2001::1: icmp_seq=9998 ttl=64 time=0.029 ms
877 bytes from 2001::1: icmp_seq=9999 ttl=64 time=0.029 ms
877 bytes from 2001::1: icmp_seq=10000 ttl=64 time=0.029 ms

--- 2001::1 ping statistics ---
10000 packets transmitted, 10000 received, 0% packet loss, time 442ms
rtt min/avg/max/mdev = 0.026/0.034/0.301/0.011 ms, ipg/ewma 0.044/0.032 ms
```

Fonte: Próprio autor (2019).

Figura 36 – Comando *free*

```
randel@ipatingao:~/TCC/Experimento$ free -m
              total        used        free      shared  buff/cache   available
Mem:           1983         114         436          20        1432        1616
Swap:          2164          11        2153
```

Fonte: Próprio autor (2019).

A Figura 37 mostra, como exemplo, o comando *time* sendo executado para sumarizar o tempo decorrido de execução do comando *ls* e também o percentual de CPU.

Figura 37 – Comando *time*

```
randel@ipatingao:~/TCC/Experimento$ /usr/bin/time -f "Tempo decorrido: %E\nPercentua
l da CPU: %P\nTempo real decorrido relacionado ao processo: %e" ls > teste.txt
Tempo decorrido: 0:00.00
Percentual da CPU: 50%
Tempo real decorrido relacionado ao processo: 0.00
```

Fonte: Próprio autor (2019).

APÊNDICE B – INSTALAÇÃO DO MININET-WIFI

Tendo a VM pré-configurada e em funcionamento, foi iniciado o passo de instalação do emulador Mininet-WiFi. O código fonte do Mininet-WiFi pode ser baixado pelo *github* conforme a seguinte url "*https://github.com/intrig-unicamp/mininet-wifi*". Para clonar o Mininet-Wifi através da linha de comando é necessário ter o *git* instalado na VM, para isso utilizou-se o comando "*sudo apt-get install git*". Com o *git* instalado foi realizado o comando para clonar o Mininet-WiFi para a VM. O comando para tal foi "*git clone https://github.com/intrig-unicamp/mininet-wifi*". Após isso, o passo subsequente foi entrar no diretório clonado com o comando "*cd mininet-wifi*" e dentro do respectivo diretório existe um arquivo de instalação que fica em "*util/install.sh*". Este arquivo foi executado com o comando "*sudo util/install.sh -Wlnfv6*". Os argumentos passados configuram o *script* de instalação para instalar as dependências *wireless* com o -W, para instalar as dependencias do Mininet-WiFi com o -n, para instalar o *OpenFlow* com o -f, para instalar o *OpenvSwitch* com o -v, para instalar *wmediumd* com -l e para instalar as ferramentas *wpan* com o -6.

Depois do término da execução do arquivo de instalação do Mininet-WiFi, foi realizado um teste para verificar se o emulador estava realmente em pleno funcionamento. Entretanto, ao executar os exemplos de teste, houve um erro que persistia em todas as tentativas, o seguinte erro acontecia "*command failed: Invalid argument* (-22)". Para tentar reparar o erro, foi decidido fazer atualização na versão do sistema operacional da versão vigente até o momento que era a *Ubuntu* 14.04 LTS para a versão *Ubuntu* 16.04 LTS a atualização foi por motivo empírico de já ter uma máquina real com a versão *Ubuntu* 16.04 LTS com o emulador Mininet-WiFi em perfeito funcionamento. Então o processo de atualização foi realizado com os comandos presentes na Figura 38, e ao final da execução dos comandos da Figura 38 a VM foi reinicializada com o comando "*sudo reboot*".

Figura 38 – Atualizando o sistema operacional

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get dist-upgrade
sudo apt-get install update-manager-core
sudo do-realease-upgrade
```

Fonte: Próprio autor (2019).

Mesmo após a atualização do sistema operacional, o erro ainda acontecia, com isso, foi pesquisado o erro e encontrada a *issue : command failed for 6LowPAN example* nela os desenvolvedores deram instruções para reexecutar o arquivo "*util/install.sh*" utilizando somente o argumento -6, caso não tivesse passado este argumento por padrão, mas, também não funcionou. Então foi verificado pelos comentários que estavam conseguindo resolver o problema ao atualizar a versão do *kernel*. Tendo isso em vista, foi feito o processo de atualização do *kernel*. Os comandos utilizados estão presentes na Figura 39. A versão do *kernel* depois da atualização ficou a 4.15.0-59-*generic*, e realizando novamente os testes verificou-se um funcionamento normal dos exemplos sem o erro em questão.

Figura 39 – Atualizando o *kernel* do sistema

```
sudo apt-get install --install-recommends linux-generic-hwe-16.04
sudo reboot
```

Fonte: Próprio autor (2019).

Possuindo então o ambiente de trabalho pré-configurado e tendo o emulador Mininet-WiFi instalado e sem erros, o passo seguinte foi a instalação dos controladores SDN, a descrição do processo de instalação dos controladores Floodlight, POX e Ryu é apresentada no APÊNDICE C e o teste de conectividade dos controladores citados pode ser consultado nos APÊNDICES D, E e F, respectivamente.

APÊNDICE C – INSTALAÇÃO DOS CONTROLADORES SDN

Dando continuidade aos passos de execução, chegou-se ao ponto de instalação dos controladores SDN, em ordem foram instalados o Ryu, POX e Floodlight. A instalação do controlador Ryu não apresentou problema, necessitando de apenas 3 comandos. O primeiro comando foi utilizado para clonar o repositório do Ryu presente no *github* com o comando "*git clone git://github.com/osrg/ryu.git*", após isso, com o comando "*cd ryu*" entrou-se na pasta clonada e por fim executou-se o arquivo de instalação com o seguinte comando "*python ./setup.py install*". Os comando estão exposto na Figura 40.

Figura 40 – Comandos instalação do Ryu

```
git clone git://github.com/osrg/ryu.git
cd ryu
python ./setup.py install
```

Fonte: Próprio autor (2019).

O segundo controlador a ser instalado foi o POX. Diferentemente do primeiro controlador instalado não contêm arquivos de instalação, tendo somente que fazer um clone do repositório presente no *github*. O comando empregado para clonar o POX foi "*git clone https://github.com/noxrepo/pox.git*" (Figura 41).

Figura 41 – Comando instalação do POX

```
git clone https://github.com/noxrepo/pox.git
```

Fonte: Próprio autor (2019).

Por fim, foi realizada a instalação do controlador Floodlight. De início, tentou-se instalar a versão *master* do Floodlight com os comandos da Figura 42, porém ocorriam erros ao executar os comandos "*git submodule init*", "*git submodule update*" e "*ant*". Ainda tentou-se pesquisar uma solução para resolver os erros de instalação, mas não houve sucesso nas pesquisas. Então partiu-se para a instalação da versão 1.2 do controlador Floodlight, para isso foi preciso somente modificar o primeiro comando apresentado na Figura 42, o comando ficou da seguinte forma "*git clone -b v1.2 git://github.com/floodlight/floodlight.git*". A versão 1.2 do controlador Floodlight foi instalada sem a ocorrência de erros. Em comparação aos dois controladores

anteriormente instalados, o Floodlight apresentou um número maior de comandos para sua instalação.

Figura 42 – Instalação Floodlight *master*

```
git clone git://github.com/floodlight/floodlight.git
cd floodlight
git submodule init
git submodule update
ant
sudo mkdir /var/lib/floodlight
sudo chmod 777 /var/lib/floodlight
```

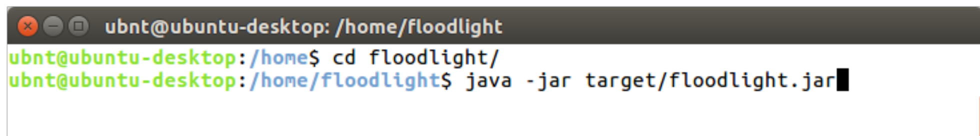
Fonte: Próprio autor (2019).

Finalizado o processo de instalação dos controladores Ryu, POX e Floodlight, o passo posterior foi a criação dos *scripts* de automatização dos experimentos, a Seção 4.2 discute, o modelo empregado para a automatização dos experimentos, discorrendo também de informações importantes sobre os fatores e níveis empregados para o presente trabalho.

APÊNDICE D – TESTANDO O CONTROLADOR FLOODLIGHT

Após a instalação do controlador Floodlight, pode-se executá-lo com o comando "*java -jar target/floodlight.jar*". A Figura 43 mostra o comando utilizado para entrar no diretório do controlador e em seguida é mostrado o comando para a execução do controlador. O Floodlight por padrão fica escutando requisições na porta 6653.

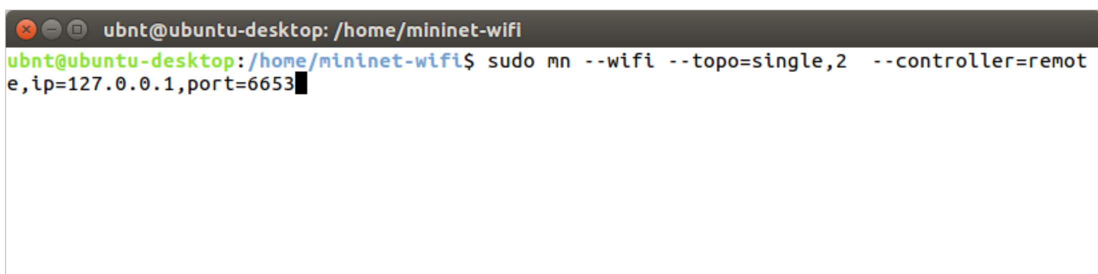
Figura 43 – Inicializando o controlador Floodlight



Fonte: Próprio autor (2019).

Em seguida, dentro do diretório do Mininet-WiFi será executado o comando para criar uma topologia simples, como é visto na Figura 44, contendo duas estações de trabalho, um Ponto de Acesso, do inglês *Access Point* (AP) e o controlador remoto que será o Floodlight. Como este controlador roda na mesma máquina que reside o Mininet-WiFi, o IP associado a ele é 127.0.0.1 que é o *localhost*. A porta usada foi a 6653 que é a porta padrão do Floodlight.

Figura 44 – Rodando a topologia no Mininet-WiFi com o Floodlight



Fonte: Próprio autor (2019).

Neste ponto a topologia simples do Mininet-WiFi foi executada e o controlador remoto Floodlight já está integrado. Para testar a conectividade das estações de trabalho criadas, será utilizado o comando "*sta1 ping sta2*", como visto na Figura 45. E por fim, o capturador de pacotes *Wireshark* foi usado para a visualização da troca de mensagens decorrente do teste de conectividade do comando "*sta1 ping sta2*", Figura 46.

Figura 45 – Teste do *ping* entre *sta1* e *sta2*

```
ubnt@ubuntu-desktop: /home/mininet-wifi
mininet-iot>
mininet-iot> nodes
available nodes are:
ap1 c0 sta1 sta2
mininet-iot> sta1 ping sta2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.279 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.150 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.225 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.234 ms
```

Fonte: Próprio autor (2019).

Figura 46 – Visualizando a troca de mensagens pelo *Wireshark*

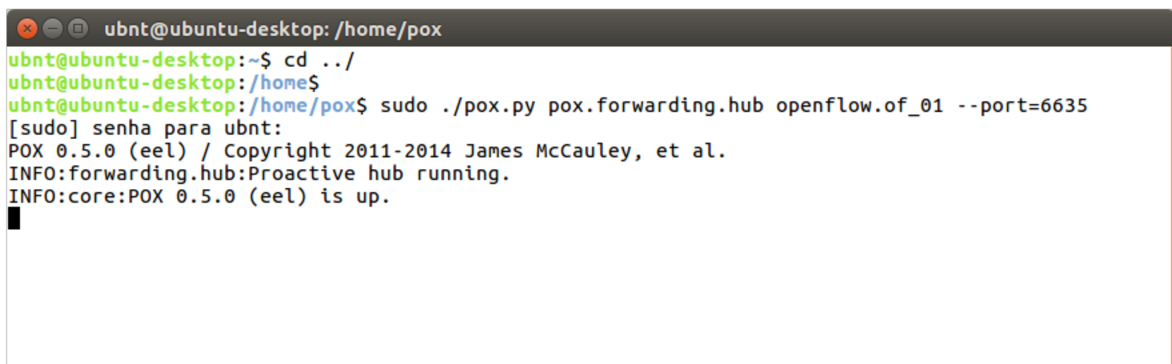
```
Capturing from ap1-wlan1
File  Edit  View  Go  Capture  Analyze  Statistics  Telephony  Wireless  Tools  Help
Apply a display filter ... <Ctrl-/>                                   Expression...  +
No.  Time          Source    Destination  Protocol  Length Info
497 229.375448507 10.0.0.1  10.0.0.2     ICMP      98 Echo (ping) request  id=0x164d, seq=439/46849, tt
498 229.375597182 10.0.0.2  10.0.0.1     ICMP      98 Echo (ping) reply    id=0x164d, seq=439/46849, tt
499 230.399382994 10.0.0.1  10.0.0.2     ICMP      98 Echo (ping) request  id=0x164d, seq=440/47105, tt
500 230.399473270 10.0.0.2  10.0.0.1     ICMP      98 Echo (ping) reply    id=0x164d, seq=440/47105, tt
501 231.423628625 10.0.0.1  10.0.0.2     ICMP      98 Echo (ping) request  id=0x164d, seq=441/47361, tt
502 231.423699838 10.0.0.2  10.0.0.1     ICMP      98 Echo (ping) reply    id=0x164d, seq=441/47361, tt
503 232.447480809 10.0.0.1  10.0.0.2     ICMP      98 Echo (ping) request  id=0x164d, seq=442/47617, tt
504 232.447566359 10.0.0.2  10.0.0.1     ICMP      98 Echo (ping) reply    id=0x164d, seq=442/47617, tt
505 233.471399421 10.0.0.1  10.0.0.2     ICMP      98 Echo (ping) request  id=0x164d, seq=443/47873, tt
506 233.471502584 10.0.0.2  10.0.0.1     ICMP      98 Echo (ping) reply    id=0x164d, seq=443/47873, tt
507 234.495874378 10.0.0.1  10.0.0.2     ICMP      98 Echo (ping) request  id=0x164d, seq=444/48129, tt
508 234.495993441 10.0.0.2  10.0.0.1     ICMP      98 Echo (ping) reply    id=0x164d, seq=444/48129, tt
509 235.519363360 10.0.0.1  10.0.0.2     ICMP      98 Echo (ping) request  id=0x164d, seq=445/48385, tt
510 235.519445211 10.0.0.2  10.0.0.1     ICMP      98 Echo (ping) reply    id=0x164d, seq=445/48385, tt
511 236.543337823 10.0.0.1  10.0.0.2     ICMP      98 Echo (ping) request  id=0x164d, seq=446/48641, tt
512 236.543409548 10.0.0.2  10.0.0.1     ICMP      98 Echo (ping) reply    id=0x164d, seq=446/48641, tt
0000  02 00 00 00 01 00 02 00  00 00 00 00 08 00 45 00
0010  00 54 87 3d 40 00 40 01  9f 69 0a 00 00 01 0a 00
ap1-wlan1: <live capture in progress>   Packets: 512 · Displayed: 512 (100.0%)   Profile: Default
```

Fonte: Próprio autor (2019).

APÊNDICE E – TESTANDO O CONTROLADOR POX

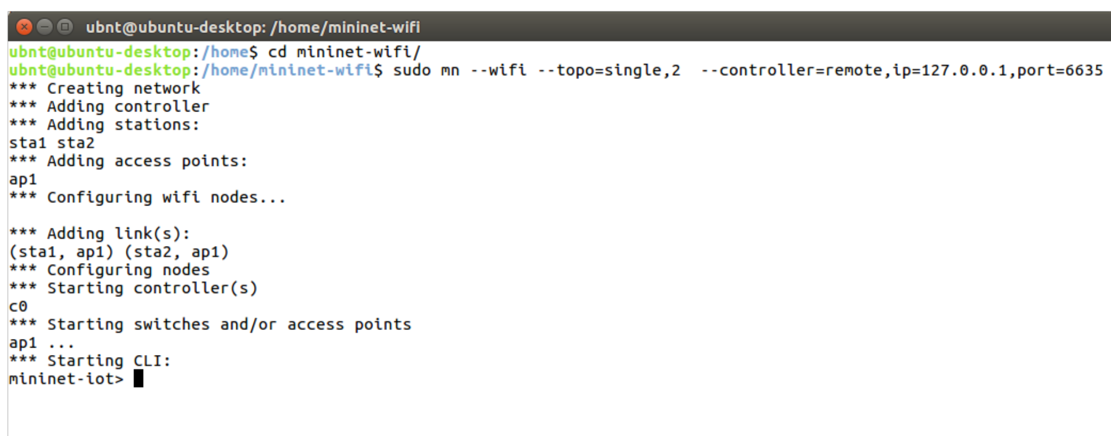
Inicialmente o controlador POX será inicializado. Para isso é necessário entrar na pasta "*pox/*" e rodar o comando "*sudo ./pox.py pox.forwarding.hub openflow.of_01 –port=6635*", Figura 47. Ao término deste passo, em outro terminal, no diretório onde está o Mininet-WiFi será executado o comando de criação de uma topologia simples com o seguinte comando "*sudo mn –wifi –topo=single,2 –controller=remote,ip=127.0.0.1,port=6635*". Neste comando é informado que o controlador será remoto "*–controller=remote*" e que o IP do controlador é o *localhost* "*ip=127.0.0.1*". Por fim, é informada a porta que o controlador POX escuta as conexões "*port=6635*", o comando de criação da topologia simples é visto na Figura 48.

Figura 47 – Inicializando o controlador POX



Fonte: Próprio autor (2019).

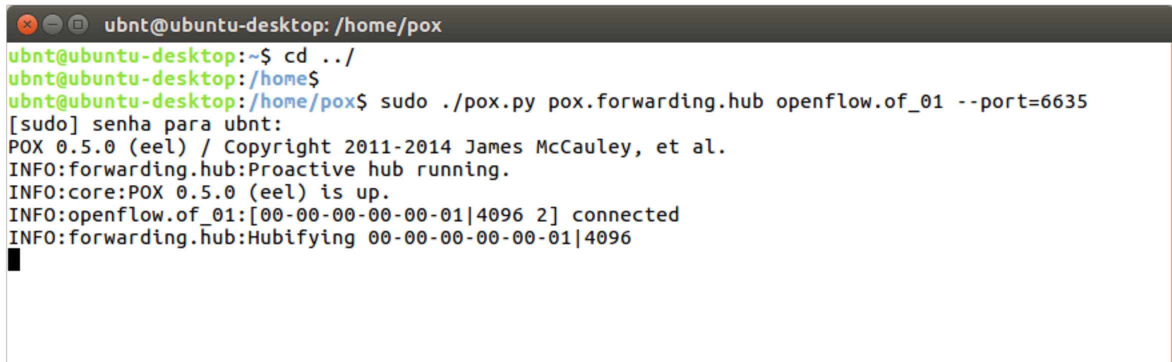
Figura 48 – Rodando a topologia no Mininet-WiFi com o controlador remoto POX



Fonte: Próprio autor (2019).

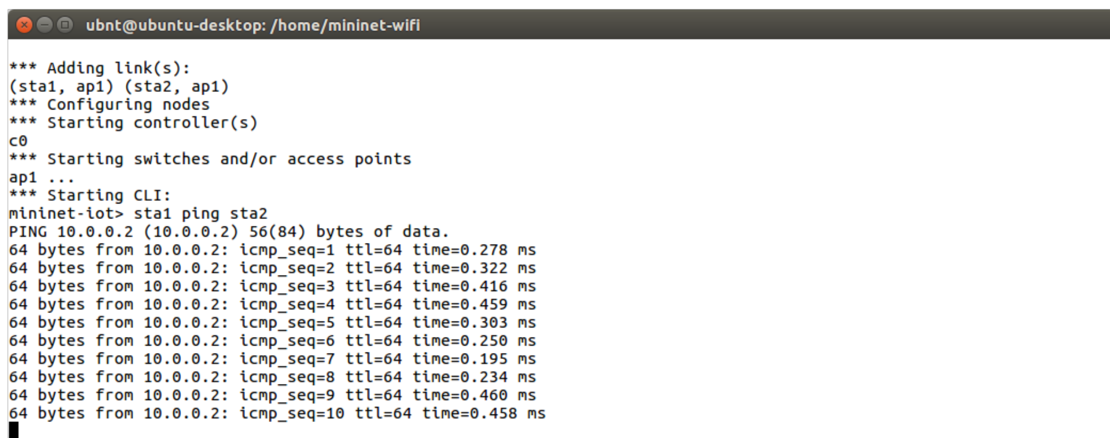
A Figura 49 mostra a conexão efetuada com êxito ao executar o comando da Figura 6. Na Figura 50 é visto o resultado do teste de conectividade entre *sta1* e *sta2*, ao utilizar o comando "*sta1 ping sta2*". A Figura 51 mostra a troca de mensagens entre *sta1* e *sta2*, através da ferramenta *Wireshark*.

Figura 49 – Conexão com o controlador efetuada



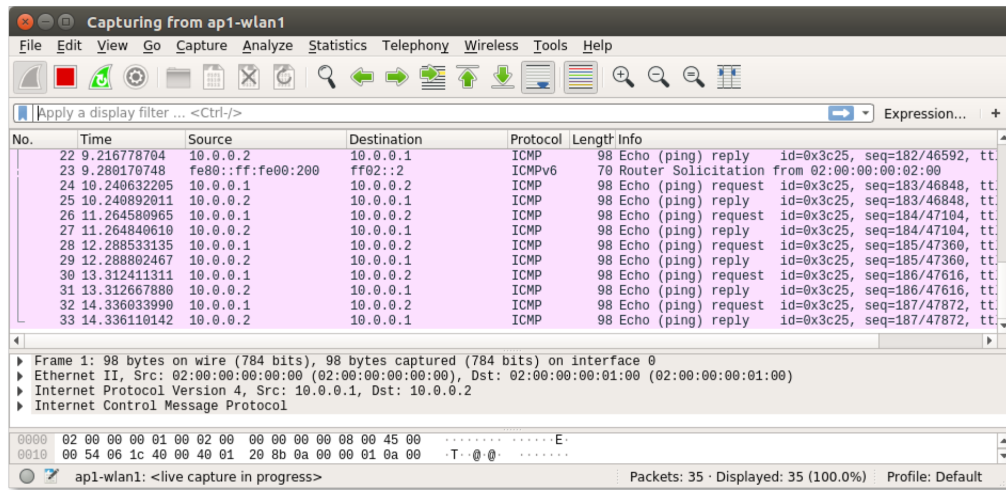
Fonte: Próprio autor (2019).

Figura 50 – Teste de conectividade entre *sta1* e *sta2*



Fonte: Próprio autor (2019).

Figura 51 – Troca de mensagens entre *sta1* e *sta2* vista no *Wireshark*

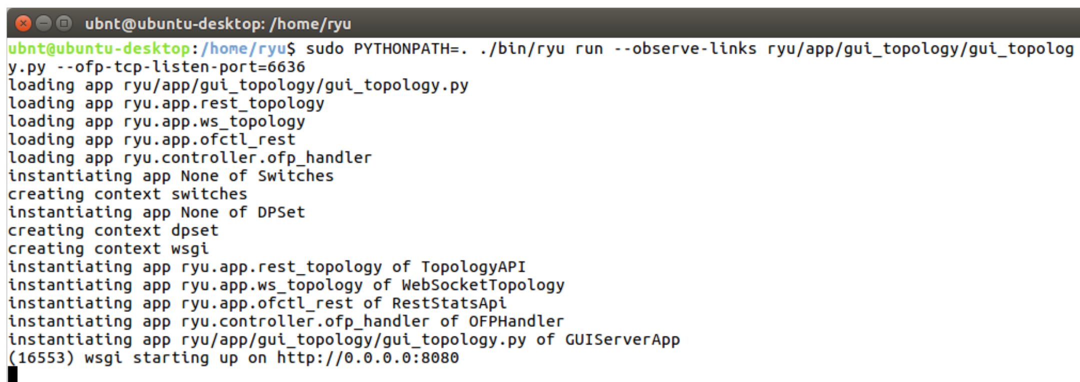


Fonte: Próprio autor (2019).

APÊNDICE F – TESTANDO O CONTROLADOR RYU

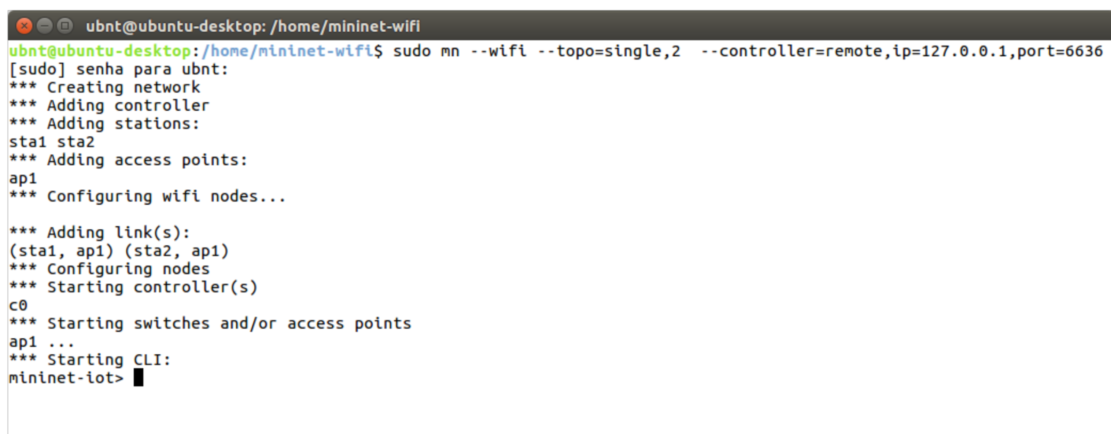
Para inicializar o controlador Ryu é necessário rodar o seguinte comando "*sudo PYTHONPATH=. ./bin/ryu run –observe-links ryu/app/gui_topology/gui_topology.py –ofp-tcp-listen-port=6636*". A Figura 52 mostra tanto o comando sendo executado quanto as mensagens de conexão após os comandos de criação de topologia no Mininet-WiFi serem realizados. E por fim, a Figura 53 mostra o comando de criação da topologia igualmente o da Figura 48 e Figura 44 modificando somente a porta onde o controlador Ryu está escutando conexões que nesse caso é 6636.

Figura 52 – Inicializando o controlador Ryu



Fonte: Próprio autor (2019).

Figura 53 – Rodando a topologia no Mininet-WiFi com o controlador remoto Ryu



Fonte: Próprio autor (2019).

Um teste de conectividade entre *sta1* e *sta2* é visto na Figura 54. Ainda, o

controlador Ryu permite a visualização dos nós ligados ao controlador através de uma aplicação *web* que pode ser acessada a partir da seguinte URL, do inglês *Uniform Resource Locator*, "*http://0.0.0.0:8080/*", na Figura 55 pode-se visualizar a estrutura de conexão criada.

Figura 54 – Teste de Conectividade utilizando o controlador Ryu

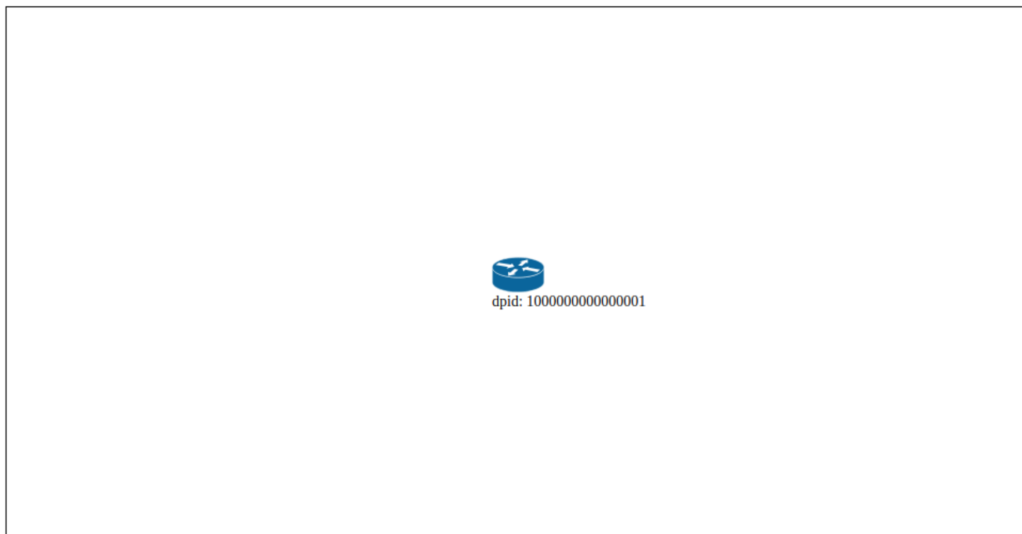
```
ubnt@ubuntu-desktop: /home/mininet-wifi
*** Configuring wifi nodes...

*** Adding link(s):
(sta1, ap1) (sta2, ap1)
*** Configuring nodes
*** Starting controller(s)
c0
*** Starting switches and/or access points
ap1 ...
*** Starting CLI:
mininet-iot> sta1 ping sta2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.303 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.263 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.360 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.324 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.329 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.726 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.400 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.372 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.171 ms
```

Fonte: Próprio autor (2019).

Figura 55 – Visualização das conexões através da *web*

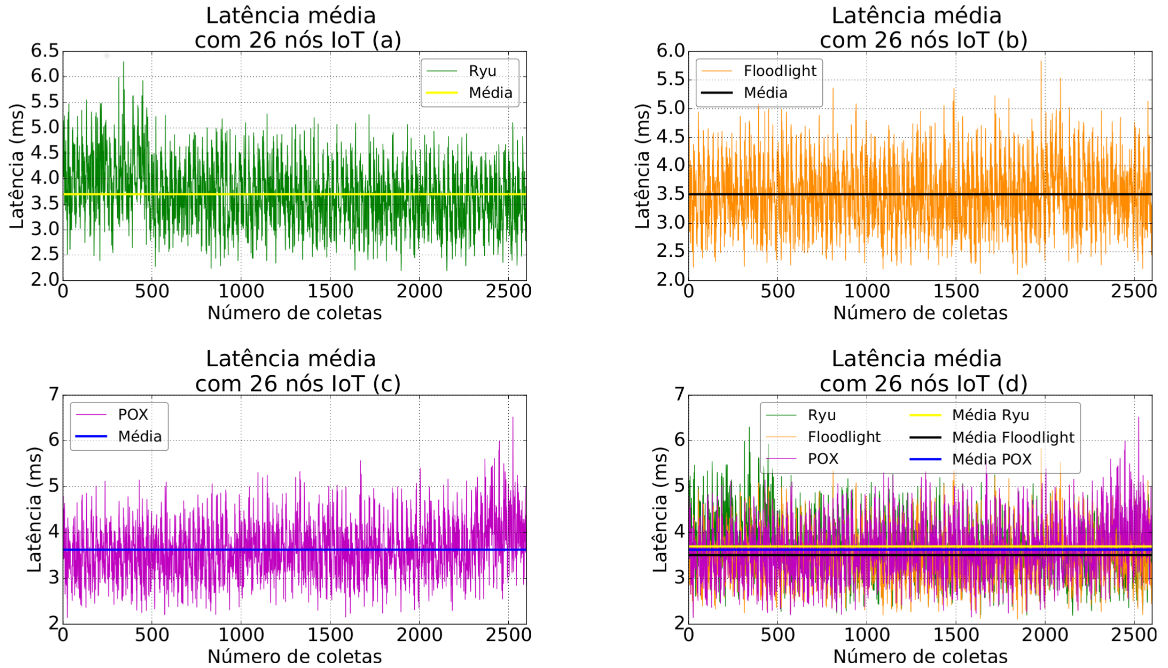
Ryu Topology Viewer



Fonte: Próprio autor (2019).

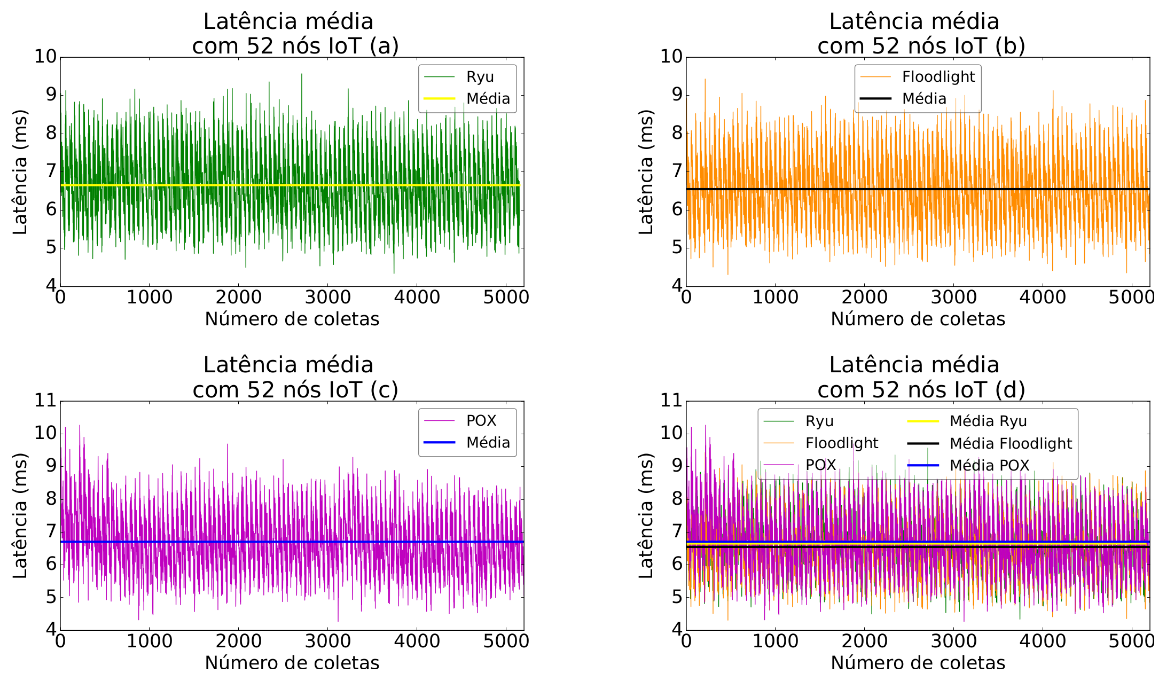
APÊNDICE G – LATÊNCIA MÉDIA

Figura 56 – Latência média com 26 nós IoT



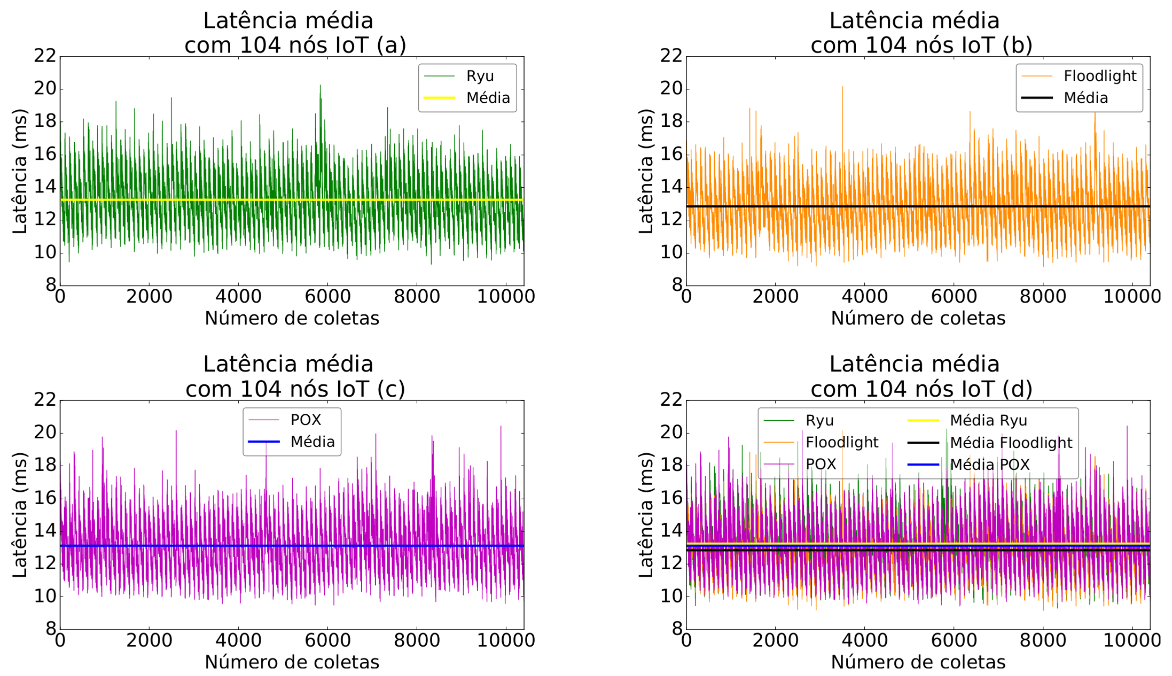
Fonte: Próprio autor (2019).

Figura 57 – Latência média com 52 nós IoT



Fonte: Próprio autor (2019).

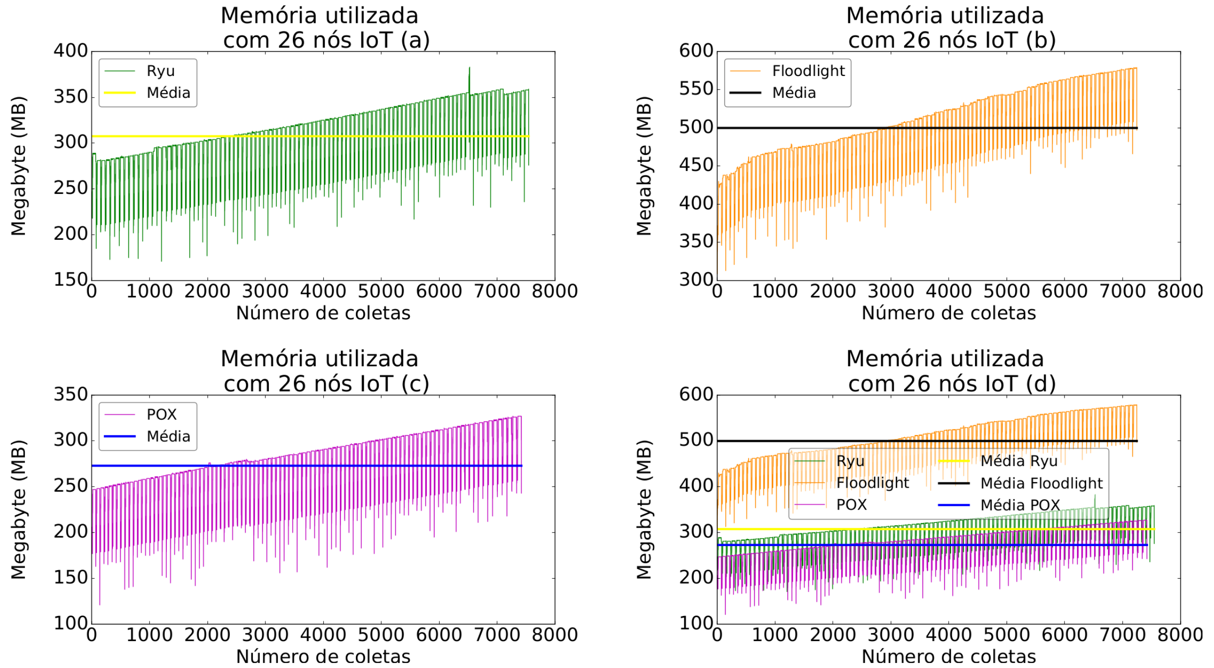
Figura 58 – Latência média com 104 nós IoT



Fonte: Próprio autor (2019).

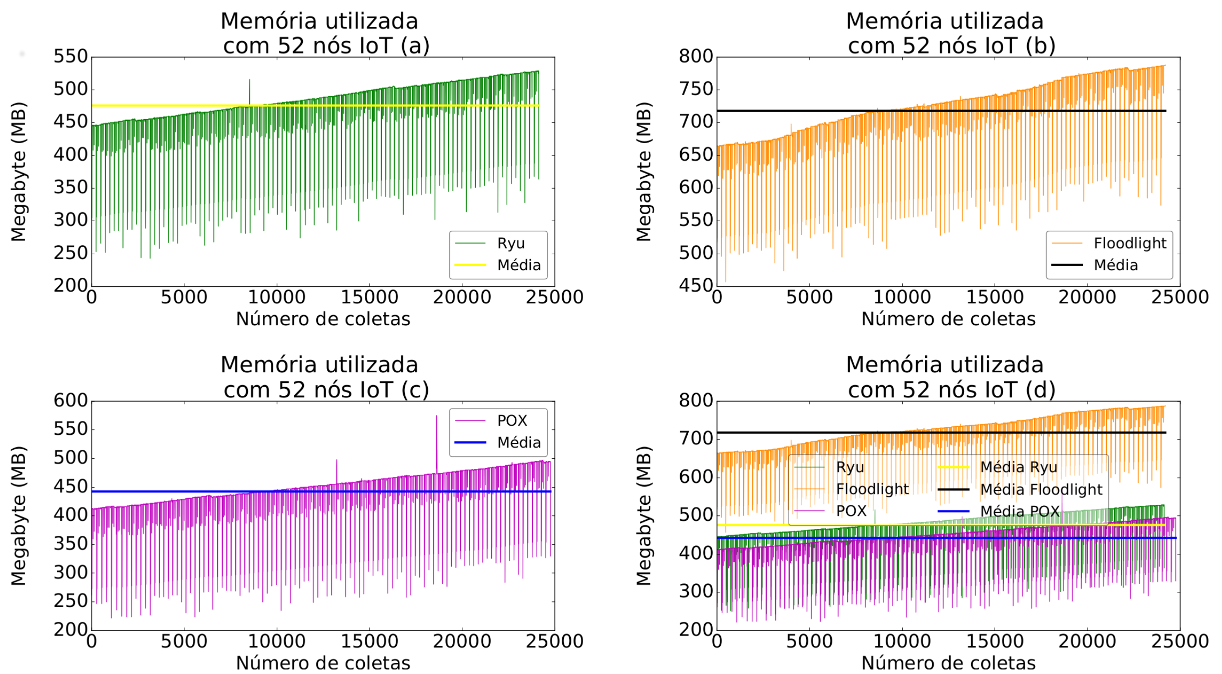
APÊNDICE H – MEMÓRIA UTILIZADA

Figura 59 – Memória utilizada com 26 nós IoT



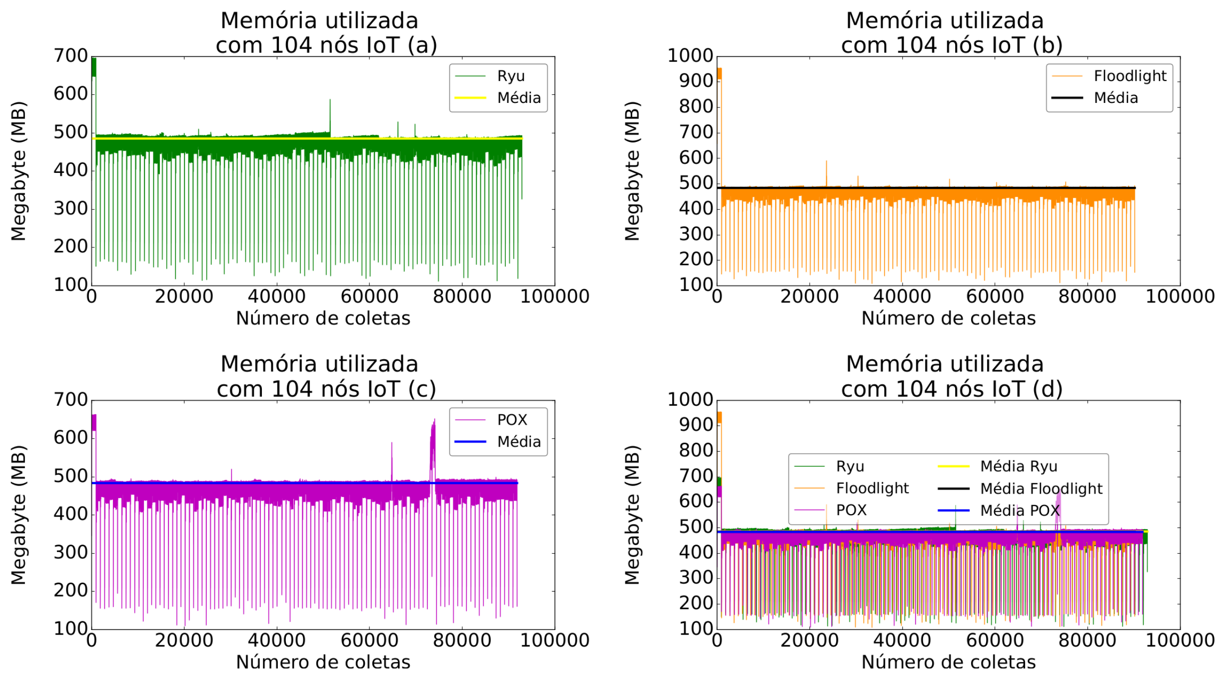
Fonte: Próprio autor (2019).

Figura 60 – Memória utilizada com 52 nós IoT



Fonte: Próprio autor (2019).

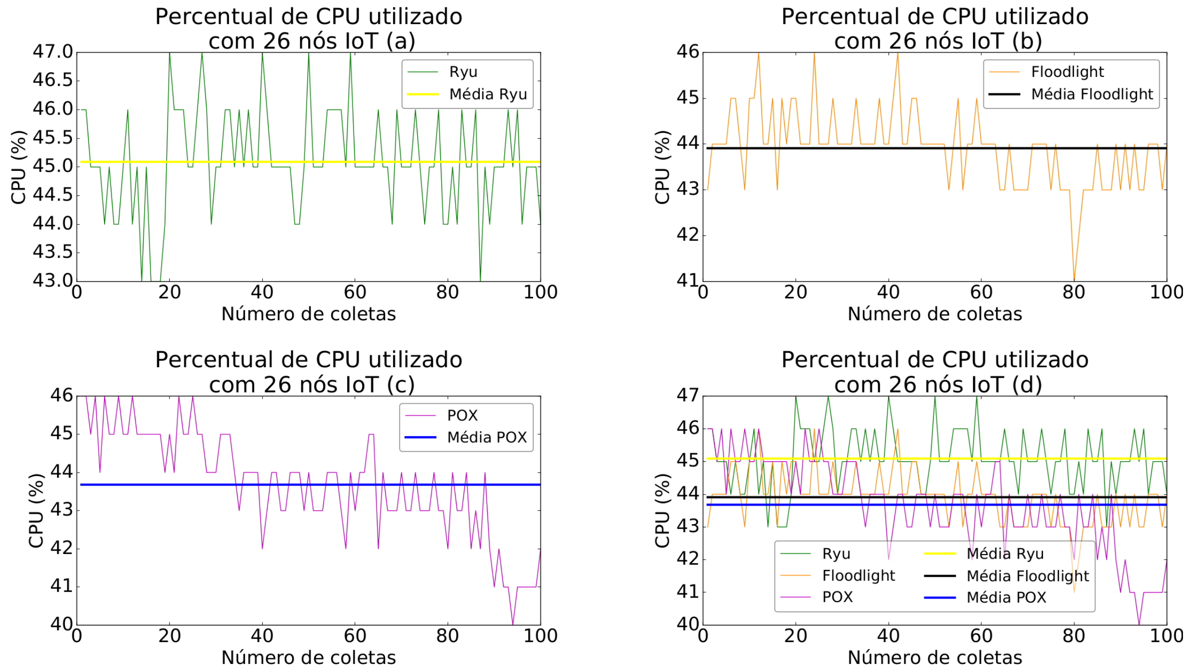
Figura 61 – Memória utilizada com 104 nós IoT



Fonte: Próprio autor (2019).

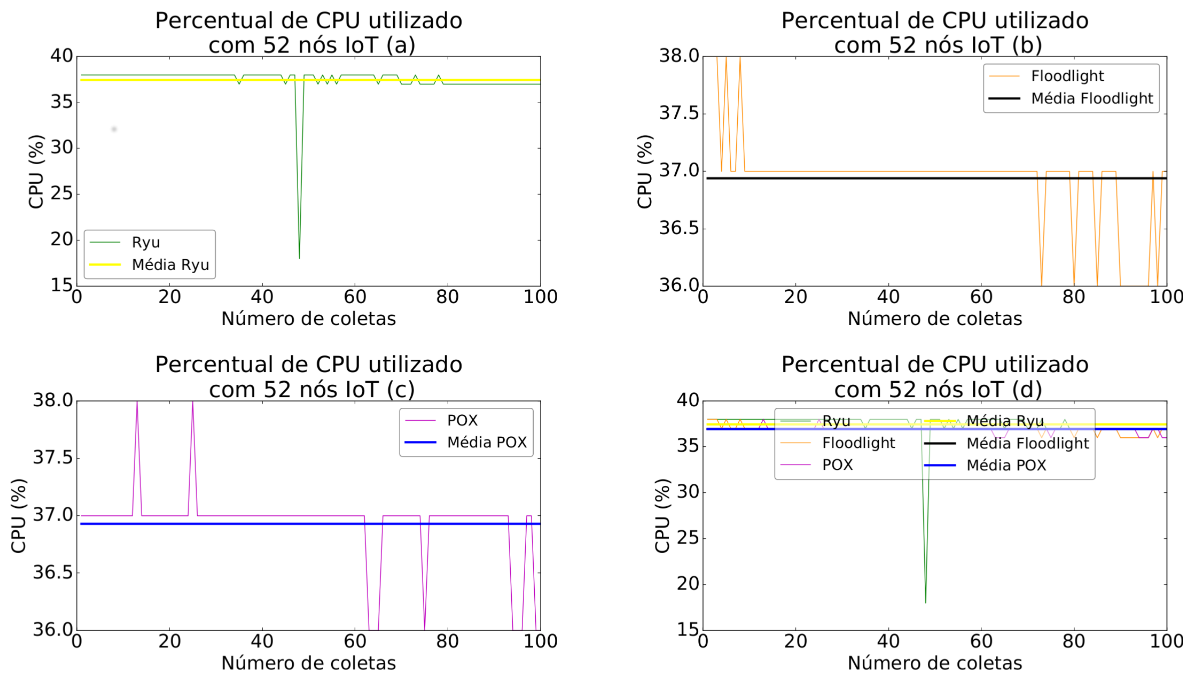
APÊNDICE I – CONSUMO DE PROCESSADOR

Figura 62 – Percentual de CPU utilizado com 26 nós IoT



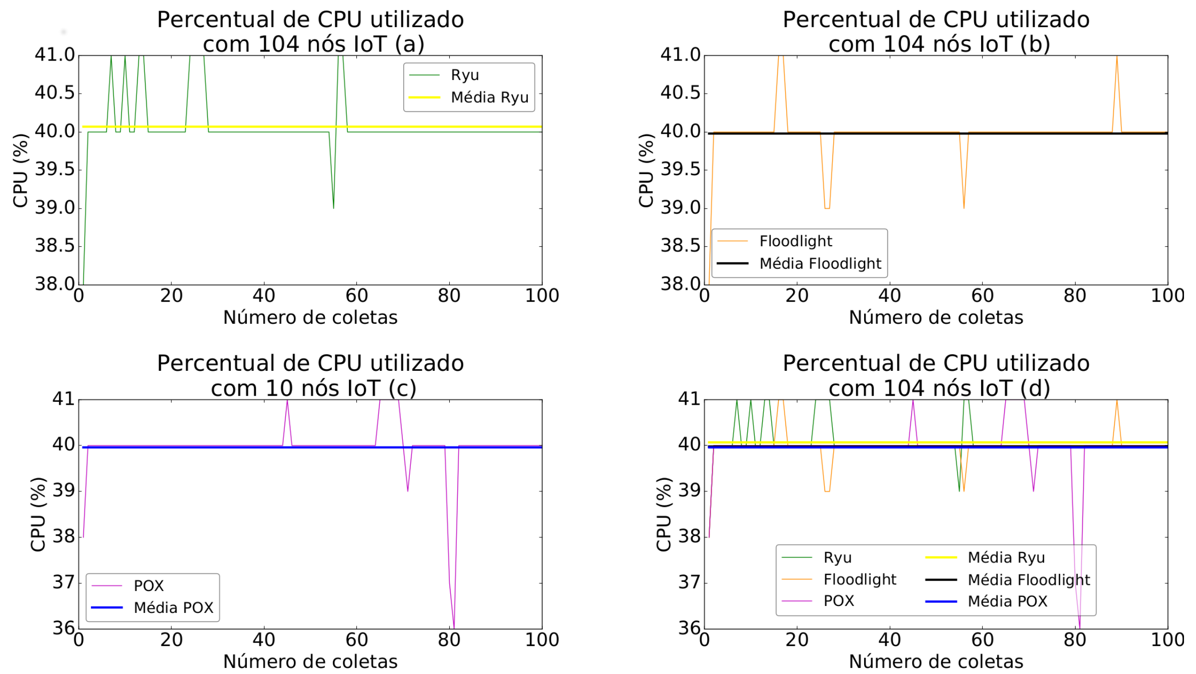
Fonte: Próprio autor (2019).

Figura 63 – Percentual de CPU utilizado com 52 nós IoT



Fonte: Próprio autor (2019).

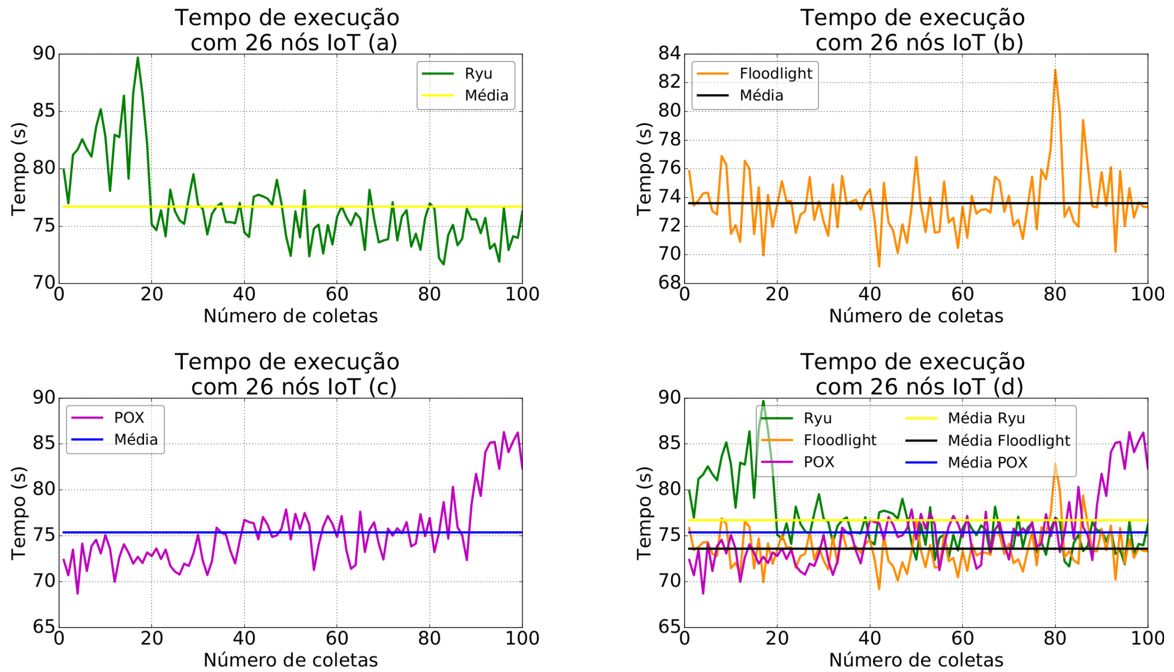
Figura 64 – Percentual de CPU utilizado com 104 nós IoT



Fonte: Próprio autor (2019).

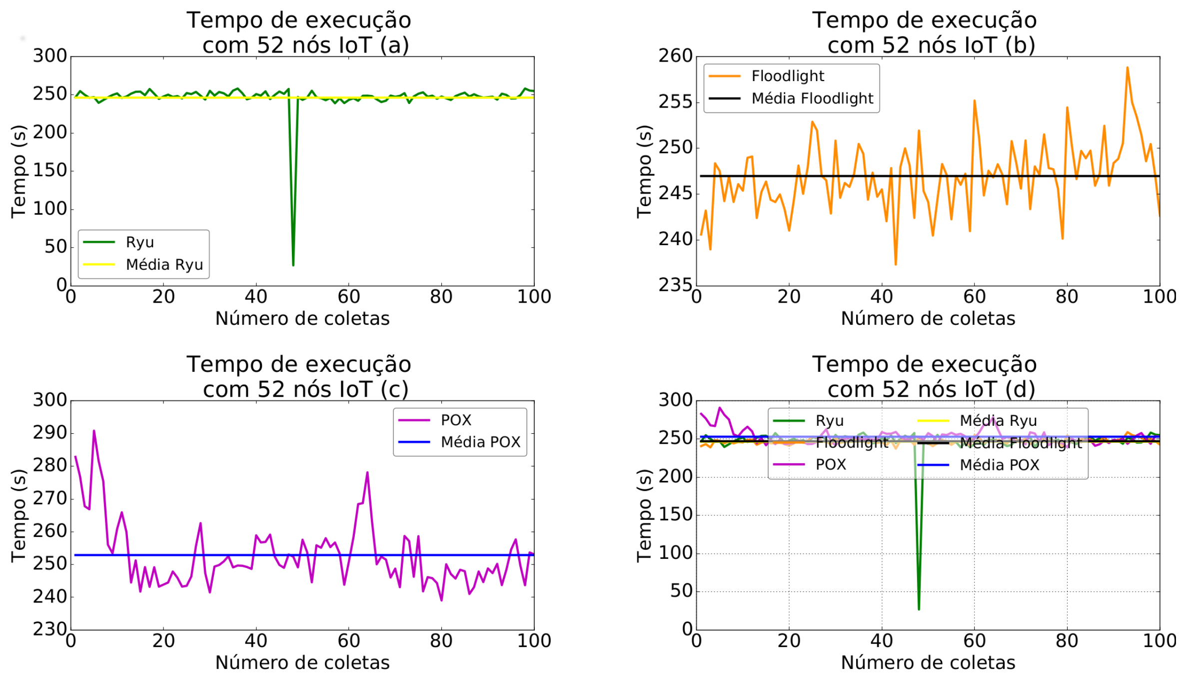
APÊNDICE J – TEMPO DE EXECUÇÃO

Figura 65 – Tempo de execução com 26 nós IoT



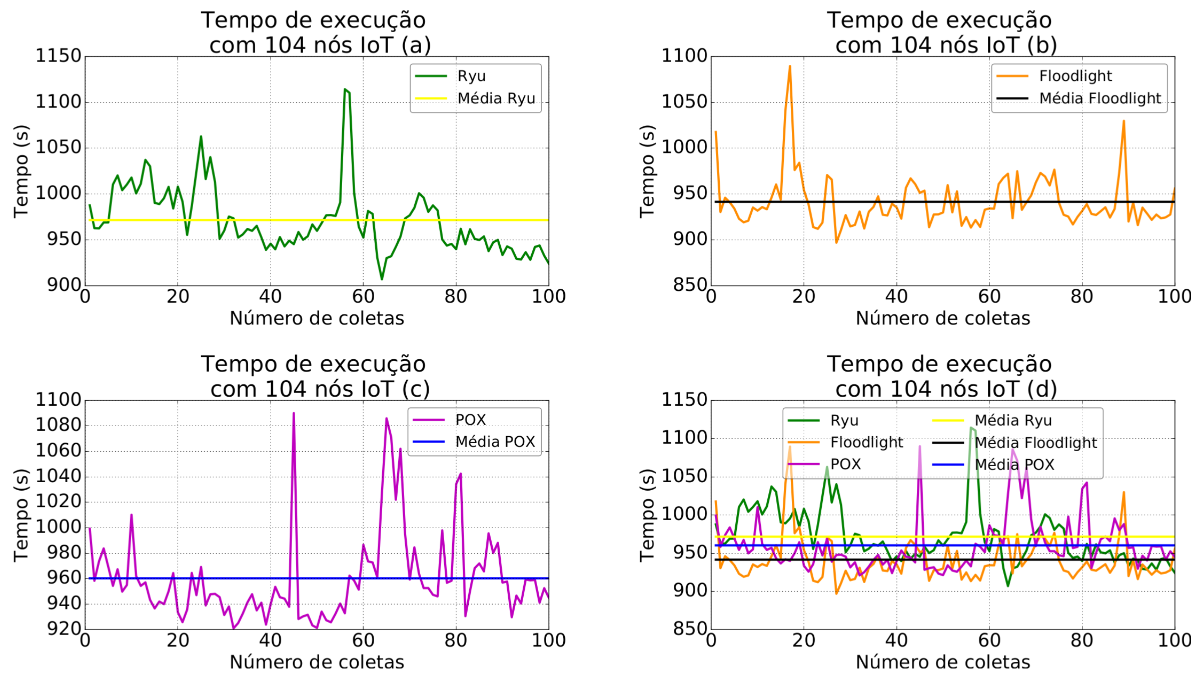
Fonte: Próprio autor (2019).

Figura 66 – Tempo de execução com 52 nós IoT



Fonte: Próprio autor (2019).

Figura 67 – Tempo de execução com 104 nós IoT



Fonte: Próprio autor (2019).