PROPOSTA DE UM *GATEWAY IoT* EM COMPUTAÇÃO *FOG* COM TÉCNICAS DE ACELERAÇÃO *WAN*

FRANCISCO LOPES DE CALDAS FILHO

Brasília, Outubro de 2019

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO

PROPOSTA DE UM *GATEWAY IoT* EM COMPUTAÇÃO *FOG* COM TÉCNICAS DE ACELERAÇÃO *WAN*

FRANCISCO LOPES DE CALDAS FILHO

Dissertação de Mestrado submetida ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica

Banca Examinadora

Prof. Rafael Timóteo de Sousa Júnior, Ph.D, FT/UnB ________________________
Orientador

Elder Oroski ________________________
Examinador externo

Georges Daniel Amvame Nze ________________________
Examinador interno

FICHA CATALOGRÁFICA

CALDAS FILHO, FRANCISCO LOPES DE
PROPOSTA DE UM *GATEWAY IoT* EM COMPUTAÇÃO *FOG* COM TÉCNICAS DE ACELERAÇÃO *WAN* [Distrito Federal] 2019.
xvi, 113 p., 210 x 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2019).
Dissertação de Mestrado - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. Internet of Things (IoT)	2. Gateway IoT
3. Fog Computing	4. Aceleração WAN
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

CALDAS FILHO, F.L. (2019). *PROPOSTA DE UM GATEWAY IoT EM COMPUTAÇÃO FOG COM TÉCNICAS DE ACELERAÇÃO WAN*. Dissertação de Mestrado, Departamento de Engenharia Elétrica, Publicação PPGENE.DM 727/19, Universidade de Brasília, Brasília, DF, 113 p.

CESSÃO DE DIREITOS

AUTOR: FRANCISCO LOPES DE CALDAS FILHO
TÍTULO: PROPOSTA DE UM *GATEWAY IoT* EM COMPUTAÇÃO *FOG* COM TÉCNICAS DE ACELERAÇÃO *WAN*.
GRAU: Mestre em Engenharia Elétrica   ANO: 2019

É concedida à Universidade de Brasília permissão para reproduzir cópias desta Dissertação de Mestrado e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. Os autores reservam outros direitos de publicação e nenhuma parte dessa Dissertação de Mestrado pode ser reproduzida sem autorização por escrito dos autores.

____

FRANCISCO LOPES DE CALDAS FILHO
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

Agradecimentos

Agradeço a minha esposa Awatef, por todo apoio e suporte incondicional, cuidando da nossa família para que eu pudesse seguir com o meu sonho do Mestrado. Aos meus filhos Júlia e Miguel que perderam horas de convivência para que eu pudesse estudar, mas sempre me recebiam com carinho e amor.

Agradeço a minha mãe, Maria Madalena e as minhas irmãs Janeina e Eliete, pelo apoio e incentivo que sempre me deram para ir mais longe.

Agradeço a minha Guru Espiritual, Shri Mataji, cujos ensinamentos me ajudaram a manter a mente calma e concentrada.

Agradeço ao meu orientador, Prof. Dr. Rafael T. de Sousa Jr., todos os ensinamentos, conselhos e oportunidades oferecidos.

Agradeço ao meu companheiro de pesquisa Lucas Martins, por dividir comigo o trabalho de produzir diversos artigos científicos.

Agradeço à pesquisadora Dayanne Fernandes, por sempre arrumar soluções criativas e inovadoras para os problemas relacionados à programação e arquitetura de software que encontrávamos no Laboratório UnB-IoT (UIoT).

Agradeço ao Prof. Dr. Robson de Oliveira Albuquerque pelas orientações e correções de textos acadêmicos.

Agradeço ao Professor Fábio Mendonça pelo apoio na condução das pesquisas conduzidas no Laboratório UIoT.

Agradeço aos pesquisadores que se dedicaram para que pudéssemos fazer nove publicações ao longo do mestrado: Cassio Fabius, Thales Von Sperling, Bruno Justino, Bruno França, Guilherme Kfouri, Daniel Gomes, Bruno Dutra, Daniel Alves e Ingrid Palma.

Agradeço ao time de pesquisadores que trabalhou comigo no Laboratório UIoT, sempre dispostos a ir além e que foram fundamentais para o desenvolvimento do nosso laboratório, entre eles Cláudio Santoro, Rodrigo de Lima, Caio Vitor, Flávio Hugo, Rafael Moraes e Mateus Carvalho.

O Laboratório UIoT conta com apoio da Fundação de Apoio à Pesquisa do Distrito Federal FAPDF (Projetos UIoT 0193.001366/2016 e SSDDC 0193.001365/2016), bem como do Laboratório LATITUDE/UnB (Projeto SDN 23106.099441/2016-43). Agradeço às duas instituições pelo apoio recebido.

Durante o desenvolvimento deste mestrado, fui bolsista do TED 002/2017, entre o Gabinete de Segurança Institucional da Presidência da República (GSI/PR) e a Universidade de Brasília. Agradeço também às duas instituições pelo apoio recebido.

RESUMO

O crescimento do número de dispositivos de Internet das Coisas (IoT), especialmente daqueles considerados como dispositivos inteligentes (*smart devices*), se, por um lado, representa um desafio para a pesquisa e o desenvolvimento tecnológico, por outro lado, estimula um paradigma computacional distribuído conhecido como computação em nevoeiro (*Fog Computing*). Este paradigma aproveita a capacidade de processamento e armazenamento dos dispositivos inteligentes, assim como dos equipamentos de comunicação e servidores, instalados na borda da rede, ou seja, provendo serviços computacionais mais próximo ao usuário, reduzindo o volume de dados que precisa ser transmitido para o conjunto de servidores centrais que constituem denominada nuvem computacional da Internet.

A computação em nevoeiro se combina e complementa com o paradigma de computação em nuvem, levando também à redução da latência e do tempo de resposta, o que é particularmente de interesse para aplicações de IoT, pois no intervalo entre as percepções obtidas pelos sensores IoT e a ativação de comandos enviados para o controle de atuadores IoT, as tomadas de decisão ocorrem na área de borda da rede, encurtando consideravelmente o tempo necessário para transmissões, com relação ao mesmo processo realizado com intermediação de entidades em nuvem computacional.

Com base nessa ideia geral, esta dissertação tem como objetivo criar uma arquitetura fim a fim de IoT em *fog*, composta com módulos próprios dos dispositivos inteligentes de IoT, bem como *gateways* e *middleware* de IoT, sob coordenação de um módulo orquestrador. Tal arquitetura tem como requisito ser capaz de realizar o processamento, armazenamento e tomada de decisão de forma distribuída para dar suporte a instâncias de redes IoT.

Além da proposição da arquitetura em *fog* para IoT, para efeito de validação da proposta, foram desenvolvidos ao longo deste trabalho módulos de software utilizando técnicas de aceleração *WAN*, ou seja, técnicas de compressão na comunicação entre os dispositivos inteligentes, *gateways*, *middleware* de IoT e módulos em nuvem.

Como resultado, além de mostrar o efetivo funcionamento geral da arquitetura proposta, a validação mostra também como o *gateway fog* proposto no trabalho permite o armazenamento e processamento de parte dos dados próximo aos sensores/atuadores de IoT, o que, conjuntamente com a aceleração *WAN*, leva a reduzir a necessidade de transmissão para a nuvem. Testes de validação do conjunto de protótipos de software permitiram mostrar uma redução superior 70% no encaminhamento de dados, com relação a uma solução pura de nuvem, ou seja, sem computação em *fog*.

ABSTRACT

The growth in the number of Internet of Things (IoT) devices, especially those considered smart devices, while on the one hand represents a challenge for research and technological development, on the other hand, stimulates a distributed computational paradigm known as fog computing. This paradigm takes advantage of the processing and storage capacity of smart devices, as well as communication equipment and servers, installed at the edge of the network, ie providing computational services closer to the user, reducing the amount of data that needs to be transmitted to the network set of central servers that make up the Internet's computational cloud.

Fog computing combines and complements the cloud computing paradigm, also leading to reduced latency and response time, which is particularly of interest to IoT applications, as in the interval between perceptions obtained by IoT sensors and the activation of commands sent to control IoT actuator, the decision-making takes place in the network edge area, considerably shortening the time required for transmissions, in relation to the same process intermediated by the computational cloud.

Based on this general idea, this dissertation aims to create an end-to-end IoT architecture in fog, composed with IoT intelligent devices own modules, as well as IoT gateways and middleware, under the coordination of an orchestrator module. Such an architecture is required to be able to perform distributed processing, storage, and decision making to support instances of IoT networks.

In addition to the proposed IoT fog architecture, for the purpose of validation of the proposal, software modules were developed throughout this work using WAN acceleration techniques, ie compression techniques in communication between smart devices, gateways, IoT middleware. and cloud modules.

As a result, in addition to showing the overall effective functioning of the proposed architecture, the validation also shows how the proposed fog gateway allows storing and processing part of the data near the IoT sensors/actuators, which, together with WAN acceleration, leads to reduce the need for the transmission to the cloud. Validation tests of the software prototype set showed a reduction in data forwarding of over 70% compared to a pure cloud solution, ie without fog computing.

SUMÁRIO

1 INTRODUÇÃO **1**
1.1 MOTIVAÇÃO 4
1.2 OBJETIVOS DO TRABALHO 6
1.2.1 OBJETIVO GERAL 6
1.2.2 OBJETIVOS ESPECÍFICOS 6
1.3 TRABALHOS PUBLICADOS 6
1.4 METODOLOGIA DE PESQUISA 8
1.4.1 FASE 1 8
1.4.2 FASE 2 8
1.4.3 FASE 3 8
1.5 CONTRIBUIÇÕES DO TRABALHO 8
1.6 ORGANIZAÇÃO DO TRABALHO 9

2 ESTADO DA ARTE E REVISÃO BIBLIOGRÁFICA **10**
2.1 INTERNET DAS COISAS 10
2.1.1 PROTOTIPAÇÃO DE DISPOSITIVOS IOT 11
2.1.2 ARDUINO 12
2.1.3 ESPS 14
2.1.4 COMPUTADORES DE PLACA ÚNICA 16
2.1.5 REDE DE SENSORES SEM FIO UTILIZANDO *ZigBee* 17
2.1.6 *Gateways* LORA 19
2.2 *Gateways* IOT 20
2.3 ACELERAÇÃO WAN 22
2.3.1 DEFINIÇÃO 22
2.3.2 TÉCNICAS DE COMPACTAÇÃO 24
2.4 MODELOS COMPUTACIONAIS 25
2.4.1 *Cloud Computing* 25
2.4.2 A CAMADA DE HARDWARE / *Data center* 26
2.4.3 CAMADA DE INFRAESTRUTURA 26
2.4.4 CAMADA DE PLATAFORMA 27
2.4.5 CAMADA DE SOFTWARE 27
2.4.6 MODELOS DE NUVEM 28
2.4.7 NUVEM PÚBLICA 28
2.4.8 NUVEM PRIVADA 28
2.4.9 NUVEM HÍBRIDA 29
2.4.10 CARACTERÍSTICAS DOS SERVIÇOS EM NUVEM 29
2.4.11 EDGE COMPUTING 30

2.5 *Fog Computing* 33
2.6 Novos desafios de arquitetura IoT 36
2.6.1 Restrições de largura de banda 37
2.6.2 Conectividade intermitente com a nuvem 37
2.6.3 Aumento da capacidade computacional dos dispositivos 39
2.6.4 Evolução da arquitetura IoT com *Fog* 39

3 Proposta de Gateway FOG com Aceleração WAN 42
3.1 Arquitetura Proposta 42
3.2 *Gateway* Cliente 43
3.2.1 Sensor Interface 43
3.2.2 Interface HTTP 45
3.2.3 Interface MQTT 45
3.2.4 Interface UDP 45
3.2.5 Interface TCP 46
3.2.6 Interface ZigBee 46
3.3 Controle de admissão de *Smart Devices* na rede UIoT 46
3.3.1 Controle de admissão de serviços na rede UIoT 46
3.3.2 Formato das mensagens transmitidas entre os *Smart Devices* e o *gateway* 47
3.4 Base de Dados Local 47
3.5 Interface de interação local com o usuário 47
3.6 Interfaces de comunicação *Gateway - Middleware* 48
3.6.1 Envio de dados para o *middleware* 48
3.6.2 Interface de comunicação com o orquestrador 49
3.7 Orquestrador 49
3.7.1 Descarte de amostras na origem 49
3.7.2 Envio de dados com supressão para o *middleware* 50
3.7.3 Aplicação de filtros 52
3.7.4 Compressão de Dados 55
3.7.5 Envio de dados para o *middleware* com compressão 56
3.7.6 Regras para aplicação de filtros 56
3.7.7 Fluxo do *gateway* 57
3.8 *Gateway* em Nuvem 59
3.9 Interface Remota de Usuário 60
3.10 Base de dados remota 60

4 Resultados 61
4.0.1 Validação da Transmissão 61
4.0.2 Validação das regras cadastradas no orquestrador 63
4.0.3 Compactação com perdas (filtros) 64

4.0.4 Testes realizados 64
4.0.5 compactação sem perdas 65
4.0.6 Desenvolvimento do Ambiente IoT 65
4.0.7 Gateway ZigBee 66
4.0.8 Gateway IP IoT 66
4.0.9 Interface de interação com o usuário 67
4.0.10 Interface de comando e controle 67
4.0.11 Controle individual dos dispositivos 67

5 Conclusão 73
5.1 Trabalhos futuros 74

Referências Bibliográficas 75

LISTA DE FIGURAS

2.1 *Smart garden.(ARDUINO, 2019)* ... 11
2.2 Impressora 3D controlada por um Arduino Mega. (ARDUINO, 2019) ... 12
2.3 Arduino Uno. (ARDUINO, 2019) ... 13
2.4 MKR1000. (ARDUINO, 2019) ... 14
2.5 Linha ESP8266. (ESPRESSIF, 2019) ... 15
2.6 *Smart Home IoT.* (HOME..., 2014) ... 16
2.7 Raspberry Pi 3 (RASPBERRY..., 2019) ... 17
2.8 Diferentes topologias em redes ZigBee (JOSHI, 2017) ... 18
2.9 Arquitetura LoRA (CHEONG et al., 2017) ... 19
2.10 Topologia LoRA (AUGUSTIN et al., 2016) ... 20
2.11 Gateway IoT. (KANG; KIM; CHOO, 2017) ... 21
2.12 Aceleração WAN Cisco (CISCO..., 2019). ... 23
2.13 Aceleração WAN Fortinet (FORTINET..., 2019). ... 24
2.14 Camadas Cloud Computing (KALIM, 2013). ... 26
2.15 Arquitetura *Cloud Computing.* (CLOUD, 2017) ... 27
2.16 Arquitetura do OpenNebula (OPEN..., 2019). ... 29
2.17 Industria 4.0 (SHROUF; ORDIERES; MIRAGLIOTTA, 2014). ... 31
2.18 Arquitetura *Edge* ... 32
2.19 Arquitetura *Fog* (TANEJA; DAVY, 2016). ... 33
2.20 Arquitetura *Fog* e *Edge* (SCHENFELD, 2017) ... 35
2.21 Redes Oportunísticas (GUO et al., 2013). ... 38
2.22 Cenários de *Fog Computing* (VARGHESE et al., 2018) ... 39

3.1 Componentes da Solução UIoT. ... 42
3.2 Modelagem do conjunto *gateway*, *middleware* e interface. O *gateway* recebe dados de dispositivos de diferentes protocolos de comunicação. ... 44
3.3 Hierarquia de cadastro do *smart device* no *gateway* para ser aceito como membro da rede. ... 48
3.4 Fluxo de cadastro de regra de supressão. ... 51
3.5 Fluxo de dados com barramento para compressão de dados sem perda. ... 52
3.6 Interação entre os componentes ... 53
3.7 Filas de transmissão*gateway* ... 55
3.8 Fluxo assíncrono de envio de dados brutos sensíveis de um dispositivo ao *middleware DIMS* através do *gateway*. ... 57
3.9 Fluxo assíncrono de envio de dados brutos não sensíveis com verificação de regras de um dispositivo ao *no orquestrador* através do *gateway*. ... 57

4.1 Cenário de teste ... 62

4.2 Gráfico dos resultados do cenário 4, compactação sem perdas, apresentado na Seção 4.0.5. 66
4.3 Estação Meteorológica ZigBee. 69
4.4 Sistema de irrigação controlado de acordo com a umidade do solo. 70
4.5 Régua inteligente controlada pelo *Middleware*. 70
4.6 Visão via *Browser* para o controle da Régua Inteligente. 71
4.7 Tela no *Gateway* local para visualização das regras definidas no Orquestrador. 71
4.8 Visualização dos dados na nuvem. 71
4.9 Dispositivos associados a um determinado *gateway*. 71
4.10 Serviços associados a determinado dispositivo. 72
4.11 Visualização dos dados em formato tabular. 72
4.12 Visualização dos dados em gráficos. 72

LISTA DE TABELAS

2.1 Comparação entre Soluções *Cloud vs Edge*. 41

3.1 Tabela de Resultados.... 53

4.1 Quantidade de mensagens que foram transmitidas nos dispositivos simulados e foram recebidas no *middleware*. 62

4.2 Quantidade de mensagens que foram transmitidas nos dispositivos simulados e foram recebidas no *middleware* e *gateway* após a criação de regras 63

4.3 Quantidade de mensagens que foram transmitidas nos dispositivos simulados e foram recebidas no *middleware* e *gateway* após a criação de regras 64

4.4 Resultados do cenário 4 de compactação sem perdas de dados. 65

4.5 Proporção relativa das etapas 2, 3 e 4 da quantidade e tamanho total de pacotes em comparação com envio sem compactação. 65

LISTA DE SIGLAS E ABREVIAÇÕES

IoT	*Internet of Things* (Internet das Coisas)
OTA	*Over the air*
WAN	*Wide Area Network*
UIoT	*Universal Internet of Things*
UIMS	*User Interface Managment System*
OSI	*Open System Interconnection*
M2M	*Machine To Machine*
WSN	*Wireless Sensor Network*
IDE	*Integrated Development Environment*
USB	*Universal Serial Bus*
SOC	*System On-Chip*
SBC	*Single Board Computing*
MQTT	*Message Queuing Telemetry Transport*
TCP	*Transmission Control Protocol*
HTTP	*Hypertext Transfer Protocol*
UDP	*User Datagram Protocol*
NIST	*National Institute of Standards and Technology*
IaaS	*Infrastructure as a Service*

1 INTRODUÇÃO

A expressão Internet das Coisas (do inglês *Internet of Things* – IoT) refere-se em geral a um paradigma de organização de sistemas que combina aspectos e tecnologias de diferentes abordagens, integrando por exemplo a computação ubíqua, protocolos e tecnologias de comunicação, sensores e atuadores, o que permite formar sistemas em que objetos do mundo real e do mundo digital interagem de forma simbiótica (BORGIA, 2014).

O termo computação ubíqua foi criado por Mark Weiser em 1991 para se referir a dispositivos conectados em todos os lugares de forma tão transparente que o ser humano acabaria por não perceber que eles estão lá.

As realizações tecnológicas nesse sentido vêm tendo um sucesso tal que há cada vez mais dispositivos com os quais o seres humanos interagem cotidianamente via Internet, o que vem sendo comprovado em levantamentos de situação do mercado de dispositivos. Por exemplo, segundo o levantamento Gartner (2015) havia em 2015 4,9 bilhões de dispositivos IoT conectados, com uma projeção de crescimento superior a 500 por cento, podendo chegar a 25 bilhões em 2020.

Segundo esse mesmo levantamento, o aumento expressivo de dispositivos se deve em parte ao fato de consumidores domésticos adquirirem cada vez mais equipamentos sensíveis às variáveis do ambiente, o que inclui relógios inteligentes com sensores de batimentos cardíacos, televisores inteligentes ligados à Internet, sistemas que controlam a iluminação residencial, carros conectados e outros elementos que estão caindo no gosto desses consumidores domésticos.

Mas também o paradigma de IoT está revolucionando os processos industriais e comerciais, por exemplo promovendo rastreabilidade dos produtos em todo o processo de produção, tornando possível um maior controle durante a cadeia produtiva, evitando desperdícios e permitindo maior integração entre fornecedores, industrias e consumidores. Desse modo, há inclusive argumentos de que os dispositivos inteligentes estão estimulando a Quarta Revolução industrial (SHROUF; ORDIERES; MIRAGLIOTTA, 2014) que é focada no consumo consciente de recursos naturais e no uso racional de energia.

A IoT também tem promovido o encontro entre o mundo real e digital na construção de vários outros exemplos de aplicações práticas, tais como cidades e estradas inteligentes, com sensores espalhados pelas rodovias e ruas que conseguem detectar engarrafamentos, realizando a reprogramação de semáforos (TOMÁS, 2017), permitindo melhor fluidez do trânsito, e com novos serviços como por exemplo aqueles em que os usuários podem acompanhar a localização de ônibus, reduzindo o tempo de espera nas paradas. Tais aplicações permitem ainda tratar fatores ambientais, como polução sonora, poluição do ar, volume pluviométrico, temperatura, umidade do ar, que podem ser medidos por sensores espalhados ao longo das cidades e cujos resultados podem ser acompanhados em tempo real (SANCHEZ et al., 2014).

O sucesso de tais aplicações práticas de IoT e suas possibilidades de uso trazem como con-

sequência o aumento exponencial do volume de dados que precisa ser transmitido pela Internet e armazenado em grandes *data centers*. As medições realizadas pelos sensores precisam ser transmitidas e processadas, de forma que se tornem uma informação útil, possibilitando que os dados possam ser utilizadas por usuários em aplicações de visualização, *big data* e mobilidade.

Nessas aplicações de IoT, é importante observar que há várias situações em que o tempo de resposta das comunicações e do processamento precisa ser muito curto. Por exemplo, as aplicações em tempo real para veículos precisam de respostas rápidas, pois a sensibilidade ao ambiente, em carros e aviões, por exemplo, contendo centenas e até milhares de sensores, não pode esperar o processamento de dados enviados para servidores remotos, para somente depois tomar decisões os controles de movimento (frenagem, altura de voo, etc.). Da mesma forma, sistemas de vigilância e segurança, além de produzir um grande volume de dados, também precisam de respostas e análises rápidas para detectar possíveis falhas.

Tais requisitos conjugados de tempo de processamento e volume de dados, que resultam do sucesso da IoT com utilização de serviços avançados de nuvem *cloud computing*, levaram, segundo Shi e Dustdar (2016), à necessidade transferir o processamento de dados ocorre, pelo menos em parte, justamente para mais perto da borda da rede, ao invés de ser completamente realizado em nuvem.

A computação de borda pode resolver problemas como a latência das comunicações, a duração limitada da bateria dos dispositivos móveis, os custos de largura de banda, além de contribuir para a segurança e a privacidade.

Uma das formas de trazer o processamento para mais próximo dos sensores e atuadores de IoT consiste em configurar *data centers* descentralizados e mais próximos dos pontos de acesso da rede, em um cenário conhecido como de *edge computing*, termo que corresponde a uma tradução de computação de borda. Nesta forma de organização, a transferência de dados entre os sensores/atuadores e a nuvem ocorre com baixa latência, mas as decisões continuam sendo tomadas por uma entidade centralizada, que pode trabalhar em orquestração com outros *data centers*.

Entretanto, o avanço da capacidade de processamento dos dispositivos IoT permitiu o surgimento de outro paradigma, denominado de computação em nevoeiro (*fog computing*), em que os dispositivos nos segmentos de acesso à rede realizam o processamento dos dados e a tomada de decisão, muitas vezes nos próprios sensores e objetos inteligentes, permitindo a realização de operações diretamente entre dispositivos, descentralizando o armazenamento e o processamento de dados gerados por tais objetos de IoT (DASTJERDI; BUYYA, 2016).

Embora estejam se tornando cada vez mais comuns, os cenários de *edge* e *fog computing* continuam e devem continuar convivendo com o modelo tradicional de sistemas de IoT em que sensores capturam os dados do ambiente e os repassam diretamente para um *middleware* IoT ou para aplicações em nuvem, ou alternativamente usam um componente de intermediação, *gateway* IoT, para a interação com tais aplicações e *middleware*. Adotamos aqui a definição de que o *middleware* IoT é o componente responsável por promover a interface de comunicação entre os dispositivos IoT e os outros componentes da rede IoT (GUBBI et al., 2013), enquanto o *gateway*

IoT é um componente compacto do *middleware* que se destina a realizar funções de intermediação, como a adaptação de protocolos ou a concentração de conexões de dispositivos de IoT.

Em todo caso, considerando essa mescla de infra-estruturas de computação, de nuvem (*cloud*), de borda (*edge*) e de nevoeiro (*fog*), que servem à construção de sistemas de IoT, colocam-se problemas que são de interesse para o desenvolvimento da presente dissertação de mestrado. Tais problemas são vividos na atualidade das redes e sistemas de IoT e é relevante que se possa trazer soluções, haja vista a perspectiva de continuidade e aumento da utilização dessas redes e sistemas. Mais especificamente, podem ser descritos aspectos caracterizadores dos referidos problemas, tais como:

- Volume de dados transmitidos entre os dispositivos de IoT (sensores, atuadores, objetos inteligentes) e a nuvem computacional. Tal volume varia de acordo com a quantidade de dispositivos instalados na rede IoT, os tipos de dados de interesse e a frequência em que estão sendo coletados, bem como o protocolo de comunicação utilizado entre os dispositivos e o *middleware*. Com o continuado aumento na quantidade de dispositivos e possíveis aplicações, em geral a quantidade de dados cresce, sempre necessitando de um tratamento que permita a resposta em limites de tempo bastante restritivos da interação com objetos reais.
- Tempo de resposta. O tempo de resposta varia em particular em função da tecnologia de transmissão utilizada pelos dispositivos. Por exemplo, uma rede satélite pode apresentar até 600 ms de latência apenas para transmissão dos dados. A distância, tanto física, quanto em termos de saltos de roteamento, entre os dispositivos e a nuvem computacional também influencia diretamente no tempo de resposta que pode impactar os limites de tempo bastante restritivos da interação com objetos reais.
- Dados repetidos. Em um ambiente controlado, onde não há variações bruscas dos indicadores que estão sendo monitorados, há diversas amostras com valores repetidos ao longo do tempo. Uma sala cofre, com climatização controlada terá ter poucas alterações nas leituras de temperatura. Os sensores poderão continuar transmitindo seus dados de coleta, independentemente se houve ou não alteração, ocupando a banda de transmissão de forma desnecessária. Sensores redundantes, instalados na mesma sala, também irão produzir leituras semelhantes ou até mesmo idênticas. Nesses casos, há ocupação de banda para transmitir o mesmo dado. Sensores do mesmo tipo, dentro do mesmo ambiente podem ler valores muito próximos ou até mesmo idênticos e repassá-los para a nuvem ou para o *Gateway*, sem se preocupar se o dado é repetido. Técnicas de otimização e deduplicação podem ser aplicadas no armazenamento da nuvem afim de economizar espaço, entretanto o recurso de banda para transmissão não é poupado (CAI et al., 2017).

Para resolver tais tipos de problemas, um conjunto de técnicas vem se desenvolvendo para melhoria da transmissão de dados em redes de longa distância, o que se denomina em geral de

otimização WAN (*wide area network*), assim como para definir dinamicamente regras de processamento e encaminhamento que possam ser aplicados em nevoeiro ou em borda de rede visando racionalizar as interações com o módulos de software em nuvem.

A presente dissertação apresenta contribuições no sentido de agregar otimização WAN com regras dinâmicas de processamento em nevoeiro, com base em motivações técnicas que são descritas a seguir e que levam a definição dos objetivos geral e específicos do trabalho de mestrado descritos mais adiante.

1.1 MOTIVAÇÃO

Conforme inicialmente apresentado, uma mescla de infra-estruturas de computação, de nuvem (*cloud*), de borda (*edge*) e de nevoeiro (*fog*) vêm servindo à construção de sistemas de IoT. Entretanto, nessa mescla colocam-se problemas que são causados pela natureza distribuída da computação em ambiente com limitações da capacidade de comunicação para um volume de dados que se apresenta crescente e cujo processamento tem requisitos de tempo real.

Técnicas de otimização WAN conjugadas com a regulagem do processamento em nevoeiro parecem promissoras para trazer soluções inovadoras para os citados problemas.

Soluções para aceleração WAN já estão presentes em diversos equipamentos como roteadores, *firewalls* ou *apliances* dedicados para tal finalidade, visando em geral melhorar a experiência de aplicações para o usuário final. Os três tipos comuns de otimização WAN são (JR; NO et al., 2012):

- Otimização de uso do protocolo TCP;
- Supressão de dados;
- Compactação de dados.

Essas técnicas podem reduzir o consumo de capacidade de comunicações, havendo casos de redução em até 54% para algumas aplicações (JR; NO et al., 2012), o que traz benefícios em situações nas quais a largura de banda é escassa. Nesses casos, *gateways* IoT convencionais encaminham os dados transmitidos pelos sensores ou dispositivos inteligentes imediatamente para o *middleware*.

Já os sistemas com computação em nevoeiro, por sua vez, têm como premissa processar e armazenar parte dos dados no ambiente local de cada processador, reduzindo o volume de dados que precisa ser transmitido para a nuvem, seja pelo uso de filtros, que limitam quais dados devem ser enviados, seja pelo processamento local, enviando apenas os dados que interessam para a nuvem.

Tais razões motivam conjugar as duas técnicas para propor neste trabalho de mestrado um mó-

dulo de *Gateway* IoT que, além de aplicar técnicas para otimização da transmissão em ambientes WAN, tem a capacidade de receber regras e fórmulas do *Middleware* IoT e aplicá-las aos dados recebidos dos dispositivos sensores, de modo a distribuir parte do processamento para a borda da rede, aproveitando assim o poder de processamento do próprio *Gateway* e evitando a transmissão de dados desnecessários para a nuvem.

Nessa proposta, as regras do processamento de borda são configuradas na interface do usuário do *Gateway* e podem ser aplicadas para um ou mais serviços associados a sensores, permitindo:

- Redução da frequência com que determinadas leituras são repassadas para a nuvem;
- Sumarização de valores dentro de um intervalo de tempo, aplicando fórmulas matemáticas de mínimo, máximo, média ou quartil;
- Criação de alarmes para dados de situações críticas, como por exemplo, o aumento inesperado da temperatura de um ambiente em que tal grandeza seja considerada um indicador crítico.

Ainda na proposta, mais de uma regra poderá ser aplicada para um conjunto de dados, permitindo por exemplo, a sumarização de leituras de temperatura dentro do intervalo de uma hora e o envio de um alerta imediato caso a temperatura exceda um limiar pré estabelecido.

Assim, a proposta apresentada nesta dissertação visa reduzir o volume de dados transmitidos pela Internet por redes IoT com a utilização de um *Gateway* IoT em computação *fog*, capaz de realizar as seguintes funcionalidades:

- Repasse seletivo de dados de sensores pelo *Gateway*, para a nuvem, obedecendo a filtros previamente estabelecidos;
- Armazenamento local dos dados não transmitidos para a nuvem, possibilitando o acompanhamento destes pelo usuário;
- Aplicação de técnicas de otimização de transmissão dos dados entre o *Gateway* e o *Middleware* IoT;

Com tais funcionalidades, parte do processamento que seria feito na nuvem fica no *Gateway* IoT proposto e técnicas de otimização WAN são usadas no conjunto de *Gateways* IoT para compactação e supressão de dados, reduzindo assim o volume total de dados transmitidos, mas sem que haja prejuízo de perda de informação para as aplicações de IoT. Tais pressupostos levam à definição dos objetivos da dissertação a seguir apresentados.

1.2 OBJETIVOS DO TRABALHO

1.2.1 Objetivo Geral

Desenvolver uma arquitetura computação em nevoeiro aberta, para utilização em redes IoT, na forma de um módulo de *gateway* IoT capaz de otimizar o processamento, armazenamento e transmissão dos dados para o ambiente em nuvem onde tais dados estejam disponíveis para o *middleware* e as aplicações de IOT.

1.2.2 Objetivos Específicos

Detalhando o objetivo geral, os objetivos específicos englobam a concepção, o desenvolvimento e a validação dos itens da propostas, incluindo:

- Concepção de *gateway* IoT aberto, capaz de receber dados de sensores com diferentes tecnologias como ZigBee, Lora, *Bluetooth*, *WiFi*;
- Criar um controlador central, que proverá o cadastro das regras aplicáveis aos diferentes serviços associados aos sensores;
- Integrar módulos que controlem a transmissão de dados entre o *Gateway* e o *Middleware* IoT, de modo a obedecer às regras aplicadas na nuvem, realizando assim a compactação e supressão dos dados.
- Integrar módulos que utilizem técnicas de compactação de dados nas comunicações de suporte à proposta;
- Implementar um protótipo da proposta para efeito de validação, usando um ambiente de IoT fim a fim, que permita realizar baterias de teste e obter resultados para análise e discussão. Com este objetivo específico, é utilizado o *Middleware* UIoT (FERREIRA et al., 2014), projeto de pesquisa que desenvolve soluções IoT fim a fim no próprio Departamento de Engenharia Elétrica da UnB,

1.3 TRABALHOS PUBLICADOS

Durante o desenvolvimento desta dissertação, foram publicados artigos científicos propondo uma solução IoT interoperável e segura, onde dispositivos IoT podem se comunicar de forma autônoma e utilizando diferentes semânticas. A pesquisa, que também foi direcionada para aperfeiçoar a segurança de dispositivos e redes IoT, resultou assim nas seguintes publicações:

1. SILVA, C. C. d. M.; CALDAS, F. d.; MACHADO, F. D.; MENDONÇA, F. L.; DE SOUSA JÚNIOR, R. T. Proposta de auto-registro de serviços pelos dispositivos em ambientes de

iot. In: *34º Simpósio Brasileiro de Telecomunicações e Processamento de Sinais*. Santarém, PA: SBrT, 2016.

2. CALDAS FILHO, F. L. d.; MARTINS, L. M. C. e.; ARAÚJO, I. P.; MENDONÇA, F. L. L. d.; COSTA, J. P. C. L. d.; DE SOUSA JÚNIOR, R. T. Gerenciamento de Serviços IoT com Gateway Semântico. In: *Atas das Conferências IADIS Ibero-Americanas WWW/Internet 2017 e Computação Aplicada 2017*. Vilamoura, Algarve, Portugal: IADIS Press, 2017. p. 199–206. ISBN 978-989-8533-70-8.

3. MARTINS, L. M. C. e.; CALDAS FILHO, F. L. d.; DE SOUSA JÚNIOR, R. T.; GIOZZA, W. F.; COSTA, J. P. C. L. d. Proposta de Adoção de Microsserviços em IoT. In: *Atas das Conferências IADIS Ibero-Americanas WWW/Internet 2017 e Computação Aplicada 2017*. Vilamoura, Algarve, Portugal: IADIS Press, 2017. p. 63–70. ISBN 978-989-8533-70-8.

4. CALDAS FILHO, F. L. d.; MARTINS, L. M. C. e.; ARAÚJO, I. P.; MENDONÇA, F. L. L. d.; COSTA, J. P. C. L. da; DE SOUSA JÚNIOR, R. T. Design and evaluation of a semantic gateway prototype for IoT networks. In: *Companion Proceedings of the 10th International Conference on Utility and Cloud Computing*. Austin, TX, EUA: ACM, 2017. (UCC '17 Companion), p. 195–201. ISBN 978-1-4503-5195-9.

5. MARTINS, L. M. C. e.; CALDAS FILHO, F. L. d.; DE SOUSA JÚNIOR, R. T.; GIOZZA, W. F.; COSTA, J. P. C. da. Increasing the dependability of IoT middleware with cloud computing and microservices. In: *Companion Proceedings of the 10th International Conference on Utility and Cloud Computing*. Austin, TX, EUA: ACM, 2017. (UCC '17 Companion), p. 203–208. ISBN 978-1-4503-5195-9.

6. SPERLING, T. L. von; CALDAS FILHO, F. L. de; DE SOUSA JÚNIOR, R. T.; MARTINS, L. M. C. e; ROCHA, R. L. Tracking intruders in IoT networks by means of DNS traffic analysis. In: IEEE. *2017 Workshop on Communication Networks and Power Systems (WCNPS)*. Brasília, DF, 2017. p. 1–4.

7. RIBEIRO, C. F. C.; CALDAS FILHO, F. L. de; MARTINS, L. M. C. e; ABBAS, C. J. B.; DE SOUSA JÚNIOR, R. T. Protocolos de Redundância de Gateway Aplicados em Redes IoT. In: SBRT. *XXXVI Simpósio Brasileiro de Telecomunicações e Processamento de Sinais - SBrT2018*. Campina Grande, PB, 2018.

8. SPERLING, T. L. von; FRANÇA, B. de A.; CALDAS FILHO, F. L. de; MARTINS, L. M. C. e; ALBUQUERQUE, R. de O.; DE SOUSA JÚNIOR, R. T. Evaluation of an IoT device designed for transparent traffic analysis. In: IEEE. *2018 Workshop on Communication Networks and Power Systems (WCNPS)*. Brasília, DF, 2018. p. 1–5.

9. DUTRA, B. V.; ALENCASTRO, J. F. de; FILHO, F. L. de C.; MARTINS, L. M. C. e; DE SOUSA JÚNIOR, R. T.; ALBUQUERQUE, R. de O. HIDS by signature for embedded

devices in IoT networks. In: UNIVERSIDAD DE EXTREMADURA. *Actas de las V Jornadas Nacionales de Investigación en Ciberseguridad (JNIC 2019)*. Cáceres, Spain, 2019. p. 53–61. ISBN 978-84-09-12121-2.

1.4 METODOLOGIA DE PESQUISA

Nesta dissertação, a proposta de pesquisa foi dividida em três etapas para facilitar a compreensão do trabalho. Essa metodologia visa a aprofundar o estudo relacionado ao tema e aos problemas relacionados, identificando os assuntos abordados pela comunidade acadêmica e os desafios na construção de uma arquitetura que permita a aceleração de dados em um ambiente IoT, conforme detalhado nas fases a seguir apresentadas.

1.4.1 Fase 1

Realizar pesquisa bibliográfica para identificar e analisar artigos que abordem o problema de transmissão de dados em redes IoT, com modelos de computação em redes IoT tanto em *Fog* com em *Edge*, assim como técnicas de aceleração *WAN*.

1.4.2 Fase 2

Obter informações sobre tecnologias relacionáveis ao tema e sobre como elas podem auxiliar na resolução de problemas encontrados na Fase 1.

1.4.3 Fase 3

Projetar, e implementar para fins de validação, uma arquitetura IoT com sensores, atuadores, *Gateways* e *Middleware*, que permita a transmissão de dados otimizada, obedecendo as regras definidas pelo controlador central.

1.5 CONTRIBUIÇÕES DO TRABALHO

O presente trabalho busca trazer as seguintes contribuições:

- Apresentação de um modelo fim a fim de arquitetura IoT capaz de receber dados de sensores e realizar parte do processamento nas extremidades da rede;
- Desenvolvimento de uma solução IoT *Fog* fim a fim, de baixo custo, na qual o *gateway* opere como um nó *Fog*, recebendo tarefas para processamento e armazenamento local;
- Implementação funcional da proposta de arquitetura IoT com posterior disponibilização dos

códigos e dos modelos de hardware e de software.

1.6 ORGANIZAÇÃO DO TRABALHO

Este trabalho foi ordenado em cinco capítulos, sendo este primeiro o de Introdução.

Para facilitar o entendimento da pesquisa, os demais capítulos estão organizados como descrito a seguir.

O Capítulo 2 oferece uma revisão atual das principais tecnologias utilizadas na construção de sensores e na transmissão de dados e como esses são aplicados nas grandes arquiteturas de IoT existentes. O capitulo 2 contempla também uma revisão bibliográfica sobre os novos paradigmas computacionais de *Fog*, *Edge* e *Mist Computing*, assim como sobre técnicas de otimização WAN, utilizadas para obter um melhor aproveitamento da capacidade de comunicação.

O Capítulo 3 apresenta a arquitetura proposta, composta pelos dispositivos inteligentes, *gateway*, *middleware* e orquestador central, detalhando como ocorre a interação entre esses componentes.

O capitulo 4 apresenta os resultados obtidos pelo *gateway Fog*, aplicando as técnicas de filtro e compactação de dados, assim como os resultados práticos.

Finalmente, O Capítulo 5 conclui este trabalho, sintetizando os resultados encontrados e sinalizando caminhos futuros, que podem ser seguidos para dar prosseguimento a este trabalho.

2 ESTADO DA ARTE E REVISÃO BIBLIOGRÁFICA

Este capítulo contém a revisão dos principais conceitos abordados nesta dissertação, como Internet das Coisas, Transmissão de dados em redes IoT, e arquiteturas *Cloud Computing*, *Edge Computing* e *Fog Computing*.

2.1 INTERNET DAS COISAS

De acordo com (ARAÚJO; DE SOUSA JÚNIOR, 2017), o termo Internet das Coisas foi definido concomitantemente ao desenvolvimento da tecnologia RFID, para utilização em áreas de rastreamento de objetos, pessoas e animais, particularmente em um artigo publicado em 1999 por Ashton (2009). Nesse artigo, partindo do princípio de que cada objeto possui um endereço único, conclui-se que os objetos podem se comunicar entre si sem nenhuma intervenção de pessoas, constituindo uma comunicação de máquina para máquina (M2M). Argumentava-se então que a interação entre os objetos possibilita a prestação de serviços sem intervenção humana.

Já a evolução das redes *Wireless Sensor Network* (WSN) vem permitindo que objetos inteligentes, instalados em diferentes ambientes, possam coletar e enviar dados. Nesse sentido, segundo Gubbi et al. (2013):

> A detecção onipresente, habilitada pelas tecnologias *Wireless Sensor Network* (WSN), atravessa várias áreas da vida moderna, oferecendo a capacidade de medir, inferir e compreender indicadores ambientais, desde ecologias delicadas e recursos naturais até ambientes urbanos. A proliferação desses dispositivos em uma rede de comunicação rede de atuação cria a Internet das Coisas (IoT), em que sensores e atuadores se combinam perfeitamente com o ambiente ao nosso redor, e as informações são compartilhadas entre as plataformas para desenvolver uma imagem operacional comum (GUBBI et al., 2013)."

De fato, vem se verificando que as aplicações de IoT crescem de forma exponencial, sendo utilizadas na agricultura, na indústria, (SHROUF; ORDIERES; MIRAGLIOTTA, 2014), em cidades inteligentes, (SANCHEZ et al., 2014) e até mesmo para auxiliar os cuidados com a saúde pessoal, monitorando indicadores como qualidade do sono, quantidade de passos caminhados diariamente e pressão arterial. Sensores, atuadores e dispositivos inteligentes não estão limitados apenas aos oferecidos pela indústria e mesmo usuários com conhecimentos básicos em eletrônica e programação podem construir *hardwares* para atender a uma determinada demanda, com custo de produção inferior de dispositivos IoT vendidos no mercado.

Sendo assunto de interesse para esta dissertação e que permite uma descrição geral das solu-

ções de IoT, o processo de prototipação de dispositivos e aplicações de IoT é descrito nas próximas subseções.

2.1.1 Prototipação de dispositivos IoT

A prototipação de dispositivos inteligentes permite que usuários menos experientes e com poucos conhecimentos em eletrônica, programação e robótica possam criar seus próprios dispositivos inteligentes. O surgimento de plataformas como Arduino, Raspberry Pi, BBC Micro, permitiu que usuários pudessem criar projetos para resolver problemas do dia a dia, como automação residencial de lâmpadas, sistemas para irrigação automática do jardim, sistemas de segurança residencial e uma infinidade de outras soluções. A comunidade de *makers* trocam informações a apresentam os resultados das suas atividades em *blogs* e plataformas de compartilhamento de vídeos, estimulando outras pessoas a criarem soluções criativas e inovadoras.

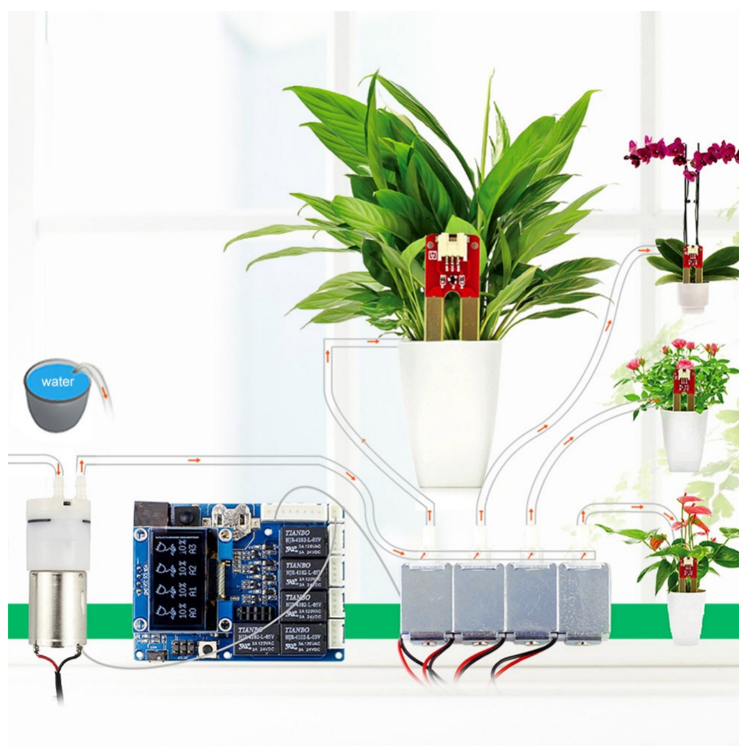

Figura 2.1: *Smart garden.(ARDUINO, 2019)*

Os dispositivos IoT podem receber acabamento com invólucros produzidos em impressoras 3D. Impressoras 3D permitem que o usuário final crie produtos personalizados e únicos, utilizando filamentos plásticos para produzir peças completas de diferentes tamanhos e custo baixo. A impressão 3D aliada a conhecimentos de eletrônica permite a criação de produtos únicos, como brinquedos, dispositivos IoT inteligentes, peças de decoração e qualquer objeto capaz de ser modelado. A prototipação de dispositivos e a impressão 3D faz parte de um novo modelo proposto de indústria conhecido como indústria 4.0 (SHROUF; ORDIERES; MIRAGLIOTTA, 2014).

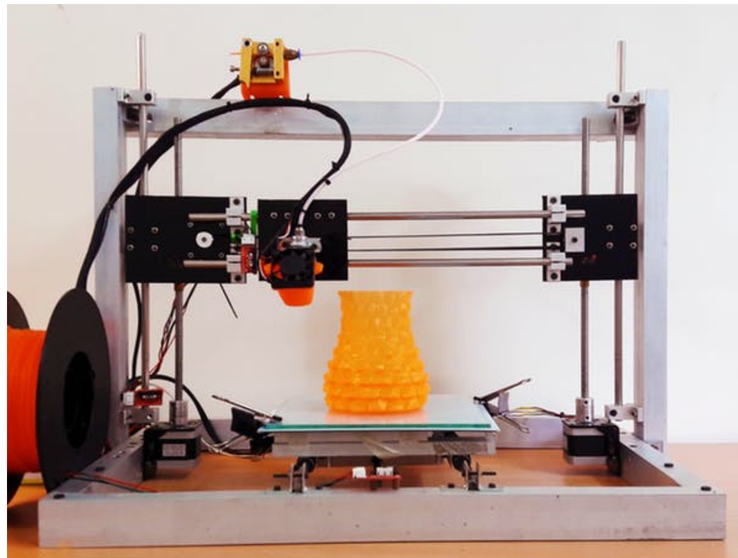

Figura 2.2: Impressora 3D controlada por um Arduino Mega. (ARDUINO, 2019)

As próximas sub-seções 2.1.2 e 2.1.3 detalham o uso de microcontroladores para prototipação de dispositivos IoT.

2.1.2 Arduino

Arduino é uma plataforma de eletrônica aberta de hardware e software, na qual usuários podem desenvolver protótipos e até mesmo produtos finais de forma simples, possibilitando que pessoas com pouco ou nenhum conhecimento em eletrônica e programação criem projetos das mais variáveis complexidades. A empresa Arduino foi fundada em 2005, criando placas com microcontroladores de baixo custo, que podem ter as suas funcionalidades expandidas com aquisição de *Shields*, placas que permitem a expansão de funcionalidades do Arduino. A programação é realizada com uma interface de desenvolvimento IDE utilizando a linguagem de programação própria, derivada do C. (ARDUINO, 2019)

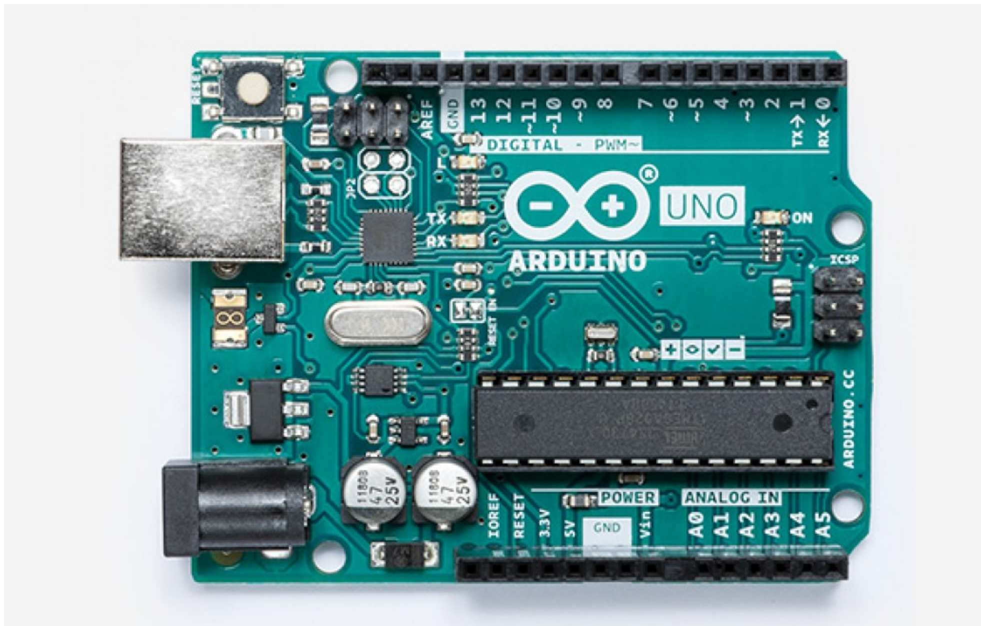

Figura 2.3: Arduino Uno. (ARDUINO, 2019)

A primeira versão comercializada pela empresa foi a placa Arduino Uno, composta por um microcontrolador ATmega328P de 8 bits, operando a 16 MHz, com 14 pinos de entrada e saída utilizado por dispositivos analógicos e portas digitais e analógicas, que podem ser utilizadas para leitura de sensores digitais ou controle de atuadores. Para programar o microcontrolador basta conectar a placa a um computador pessoal, utilizando uma interface USB. A empresa tem evoluído o seu portólio, criando diversas placas focadas para criação de dispositivos IoT, como o modelo MKR1000, com microcontrolador SAMD21 Cortex-M0 de 32 Bits, interface *Wireless* embutida, Clock de 48 MHz, 8 portas analógicas, 12 digitais, capacidade de programação via rede sem fio. O modelo MKR WAN 1300 possui configurações similares ao MKR1000 porém com interface *LoRA*, operando a 915 MHz, permitindo a comunicação com *gateways LoRA* a distancias superiores a 10 Km. O MKR WAN 1300 tem baixo consumo energético, podendo ser alimentado com duas pilhas AA, ou baterias de 5V, sendo utilizada em projetos de baixo consumo de energia. Soluções IoT utilizando Arduino como microcontrolador, associado a redes de sensores sem fio ZigBee tem sido amplamente difundidas, dado o baixo consumo energético, baixo custo de aquisição e manutenção (FERREIRA; CANEDO; DE SOUSA JÚNIOR, 2013).

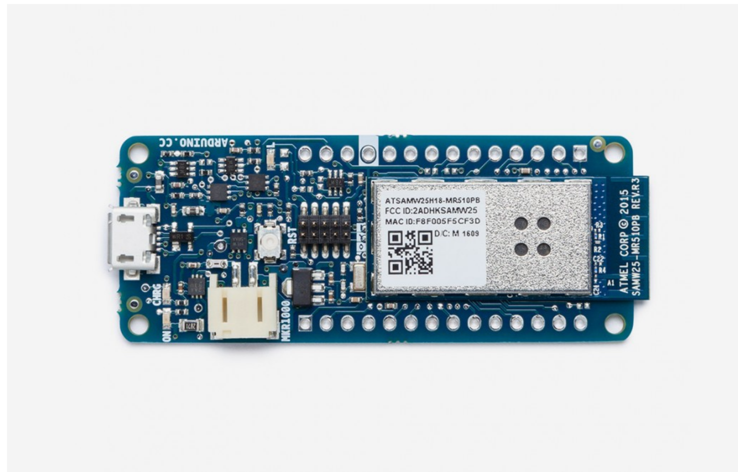

Figura 2.4: MKR1000. (ARDUINO, 2019)

A plataforma Arduino vem sendo amplamente utilizada na construção de dispositivos IoT como em Ferreira (2014) para criação de sensores e atuadores inteligentes, capazes de interagir com *Middleware* e serem comandados remotamente.

A plataforma Arduino foi desenvolvida para facilitar a prototipação e criação de sistemas embarcados. Sistemas embarcados são construídos para executar uma tarefa pré-determinada, realizando a leitura de variáveis do ambiente com sensores, executando um processamento e entregando uma saída programada, estando limitada apenas ao conjunto de ações e entradas previamente estabelecidos (BALL, 2002). Diferente dos computadores que rodam sistemas operacionais e softwares para as mais variadas aplicações, os sistemas embarcados são construídos para executar uma tarefa específica. Logo na maioria dos projetos para estes sistemas não há flexibilidade de software ou hardware que lhes permita realizar outras tarefas que não sejam aquelas para as quais foram desenhados e desenvolvidos. (BALL, 2002)

2.1.3 ESPs

Aproveitando a popularidade criada pela plataforma Arduino, empresa Espressif desenvolveu microcontroladores que pudessem ser utilizados pelo público com baixo conhecimento em eletrônica e programação, porém com funções extras, rádio *Wireless 802.11*, maior poder de processamento, e compatível com as linguagens de programação Lua, MicroPython e C, permitindo também utilizar a IDE do Arduino para programação. Um dos microcontroladores mais populares lançados pela Espressif foi o ESP8266, que possui as seguintes características (ESPRESSIF, 2019):

- *System-On-Chip* com Wi-Fi 802.11g
- Processador de 80 MHz com arquitetura RISC 32 Bits
- Memória RAM de 32 kBytes para instruções e 96 kBytes para dados

O microcontrolador ESP8266 foi encapsulado em diferentes módulos, com quantidade de portas de entrada e saída variando conforme o modelo descrito na Figura 2.5.

Figura 2.5: Linha ESP8266. (ESPRESSIF, 2019)

O baixo custo, associado a maior capacidade de processamento e o *WIFI* embutido, popularizou a utilização dos módulos ESP8266, permitindo que usuários com pouco conhecimento em eletrônica pudessem criar projetos de automação comercial, robótica e controle industrial. A fabricante criou a versão ESP8266EX, voltada para a indústria, mais robusto, podendo operar em temperaturas de -40 e + 125°C. A linha ESP8266 também se destaca pelo baixo consumo energético

Apesar da evolução das capacidades de processamento e armazenamento, os microcontroladores não são adequados para aplicações cliente-Servidor mais complexas. Estas necessidades podem ser suprimidas com a utilização de computadores de placa única, em inglês Single Board Computing (SBC). Com tamanho um pouco maior que um cartão de crédito, as SBCs possuem poder de processamento suficiente para criar servidores de armazenamento, Servidores WEB, *Media Centers*, e uma serie de outros projetos voltados para área de Internet das Coisas, automação residencial. Uma SBC pode ser utilizada para construção de um *gateway* IoT, ficando responsável pelo recebimento de dados e controle de diferentes sensores e atuadores em uma rede, ou operando como ponte na interlocução para redes de Sensores sem Fio com um *middleware* em nuvem.

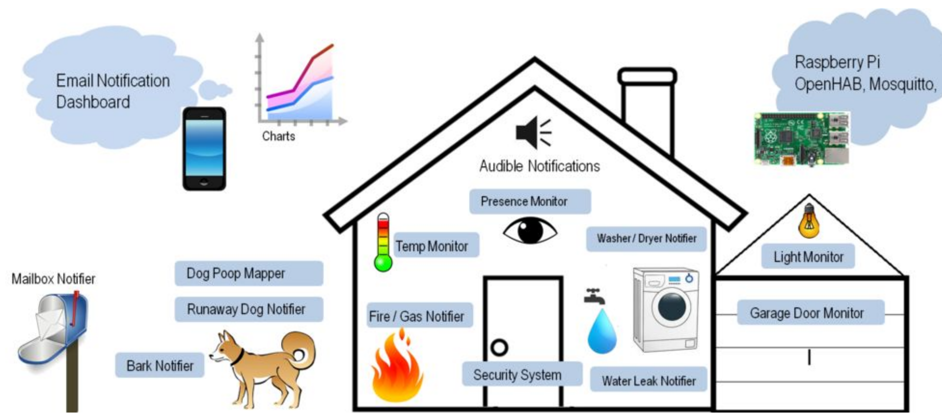

Figura 2.6: *Smart Home IoT.* (HOME. . . , 2014)

2.1.4 Computadores de placa Única

Single-Board Computer (SBC) são computadores completos, dotados de microprocessador, memória, e dispositivos de entrada e saída integrados em uma única placa de circuito impresso. São utilizados na indústria, em sistemas embarcados, na educação de crianças para ensinar conceitos de programação e robótica (MITCHELL, 2012, p. 2), e na e criação de dispositivos IoT (KUMAR; RAJASEKARAN, 2016) (SHETE; AGRAWAL, 2016). A utilização de SBCs cresceu consideravelmente nos últimos anos, apresentando baixo custo para aquisição e grande poder de processamento. Diversos fabricantes apresentaram as suas versões como *Orange Pi* (SBC. . . , 2018) A empresa BeagleBone (BEAGLEBOARD, 2019). A mais conhecida a é a *Raspberry Pi*. A última versão da SBC Raspberry Pi tem processador *Quad-Core* ARM Cortex-A53, 1,2 GHz, 1 GB de memória RAM, GPU integrada, interfaces Ethernet, Wireless 802.11ac, Bluetooth, saída HDMI e 40 pinos GPIO, que podem ser utilizados para conectar sensores, botões e atuadores. A placa é compatível com diversos sistemas operacionais baseados em *Linux* e *Windows*. O custo inicial desta placa é 35 dólares e com capacidade computacional suficiente para ser utilizada como computador pessoal ou para criação de dispositivos IoT inteligentes.

Figura 2.7: Raspberry Pi 3 (RASPBERRY. . . , 2019)

As SBCs tem um consumo de energia superior ao de microcontroladores, não sendo indicadas para utilização de dispositivos alimentados por baterias. A interface *Wireless* também possui baixo alcance, devendo operar perto do roteador sem fio. Há outras tecnologias que permitem que os dispositivos finais operem em uma distancia maior do *gateway* como *ZigBee* e LoRA, que serão abordadas nas próximas subseções.

2.1.5 Rede de sensores sem fio utilizando *ZigBee*

O padrão 802.15.4 especifica um conjunto de padrões para comunicação sem fio entre sensores, atuadores e dispositivos eletrônicos no geral, com ênfase no baixo consumo de energia, baixo custo de implementação e baixa potência de sinal. Transmissores *ZigBee* também transmitem dados utilizando uma conexão sem fios, semelhante ao *Bluetooth*, porém com alcance muito maior. O *Bluetooth* tem evoluído rapidamente ao longo do tempo, porém algumas versões permitem a comunicação em um raio inferior a 10 metros. Outra vantagem do *ZigBee* em relação ao *Bluetooth* é a capacidade de formar redes *Mesh*, onde dispositivos não precisam se comunicam apenas com um ponto central, aumentando consideravelmente o alcance e a permeabilidade da rede.

O *ZigBee* opera na frequência de 2.4 GHz no Brasil, o que limita um pouco o seu alcance de comunicação. Essa limitação pode ser contornada pelo uso de redes *Mesh*. Por se tratar de uma rede que preza pelo baixo consumo de energia, as taxas de transmissão variam entre 20 e 900 kb/s.

Dispositivos utilizando interfaces *ZigBee* podem operar como dispositivo finais (*End-Devices*), roteadores (*Routers*) e coordenadores (*Coordinator*). Os *End-Devices* podem ser sensores ou atuadores, apenas encaminhando pacotes para os roteadores ou coordenadores, geralmente apre-

sentam baixo consumo energético. Os nós do tipo *router* podem atuar como dispositivos finais, encaminhando pacotes para outros elementos do tipo *router* ou para o coordenador. O coordenador fica responsável por montar a rede *ZigBee*, armazenando informações de rotas e chaves de segurança. Existe apenas um único coordenador a rede.

Dispositivos ZigBee podem operar em diferentes topologias, como estrela, árvore, *mesh* ou híbrida, conforme descreve a Figura 2.8.

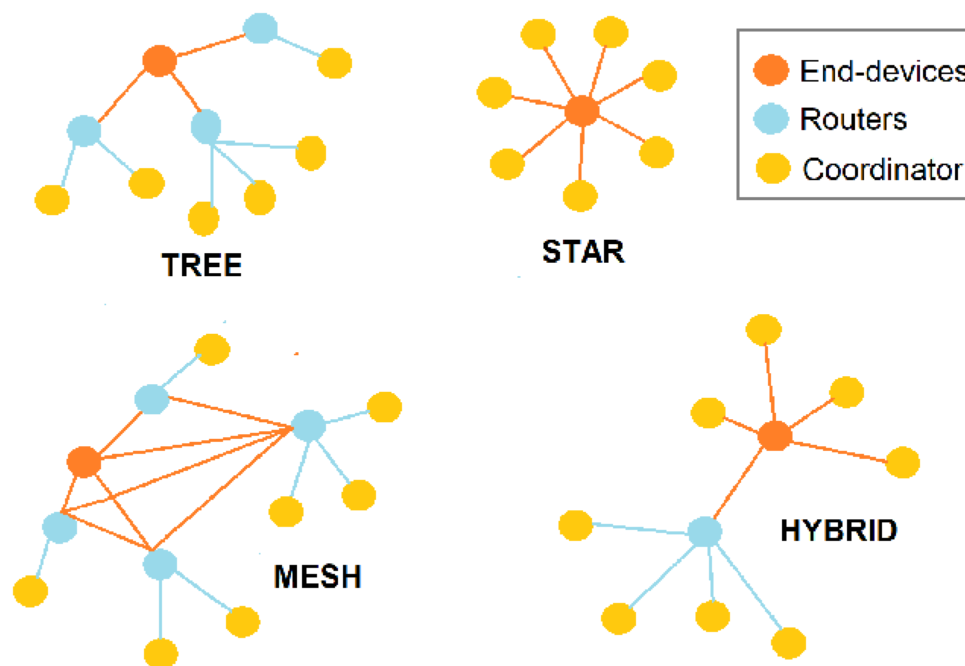

Figura 2.8: Diferentes topologias em redes ZigBee (JOSHI, 2017)

2.1.5.1 LoRA

O protocolo *LoRA* é uma solução de transmissão sem fio Sub-GHz que permite comunicação a longas distâncias com baixo consumo de energia e com possibilidade de transmitir os dados cifrados entre todas as pontas. A topologia da rede *LoRA* é baseada em estrela, semelhante às redes locais sem fio 802.11, porém com alcance que varia entre 4 km, em áreas urbanas, e até 15 km em áreas rurais e taxa de transferência máxima de 50 kb/s.

O protocolo *LoRA* opera na camada física modelo OSI. Os dispositivos remotos enviam os dados para um *gateway* dentro do alcance, que irá repassar os dados utilizando protocolo TCP/IP até os servidores em nuvem.

O protocolo que define a arquitetura do sistema, bem como os parâmetros de comunicação utilizando a tecnologia *LoRA* é conhecido como LORaWAN™.

O protocolo *LoRaWAN* define as especificações de segurança, funcionamento, priorização de mensagens, controle de potência do sinal, visando aumentar o tempo de vida útil da bateria, além da semântica e sintaxe em que a troca de mensagens deve ocorrer. *LoRAWAN* opera na camada

lógica do modelo OSI. A rede LoRA é composta pelos seguintes componentes:

- Dispositivos finais;
- *Gateways*;
- Servidores de rede;
- Servidores de Aplicação.

Os dispositivos finais podem ser sensores ou atuadores, geralmente alimentados com uso de baterias e possuem um rádio que atua como transmissor e receptor *LoRA*. A comunicação ocorre apenas com o *gateway* LoRA, que irá encaminhar os dados recebidos para os servidores de rede.

Os *gateways LORA* podem receber dados dos dispositivos finais instalados a quilômetros de distancia e fazendo a ponte entre os dispositivos e a rede IP.

Servidores de rede são os responsáveis pelo gerenciamento dos dados repassados pelo *gateway*, controlando a velocidade de transmissão e a necessidade de retransmissão dos pacotes. Quando há mais de um *gateway* na rede os servidores assumem um papel fundamental, concentrando os dados transmitidos pelo *gateway* em um ou mais servidores antes de repassar para a aplicação (WIXTED et al., 2016).

Os servidores de aplicação recebem os dados e realizam o processamento, armazenamento e apresentação dos dados (AUGUSTIN et al., 2016).

A utilização da rede *LoRA* está crescendo rapidamente devido ao baixo custo de aquisição e manutenção. Dispositivos finais estão sendo empregados na agricultura, e em estações meteorológicas, controle de frotas do transporte público (ADELANTADO et al., 2017).

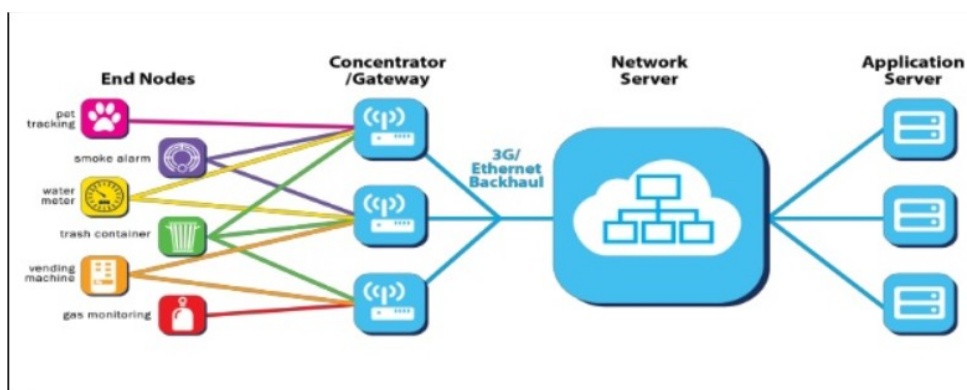

Figura 2.9: Arquitetura LoRA (CHEONG et al., 2017)

2.1.6 *Gateways* LoRA

Apesar de coexistirem, o protocolo LoRA tem ganhado espaço, pela capacidade de transmissão em distâncias ainda maiores das alcançadas pelo *ZigBee*, aliado ao baixo consumo energético.

Gateways LoRA são uma solução viável para receber dados de sensores espalhados em grandes áreas e transmiti-los para rede IP. A arquitetura da rede é basicamente em estrela, com o *gateway* recebendo os dados de centenas e até mesmo milhares de sensores e fazendo o encaminhamento para rede IP.

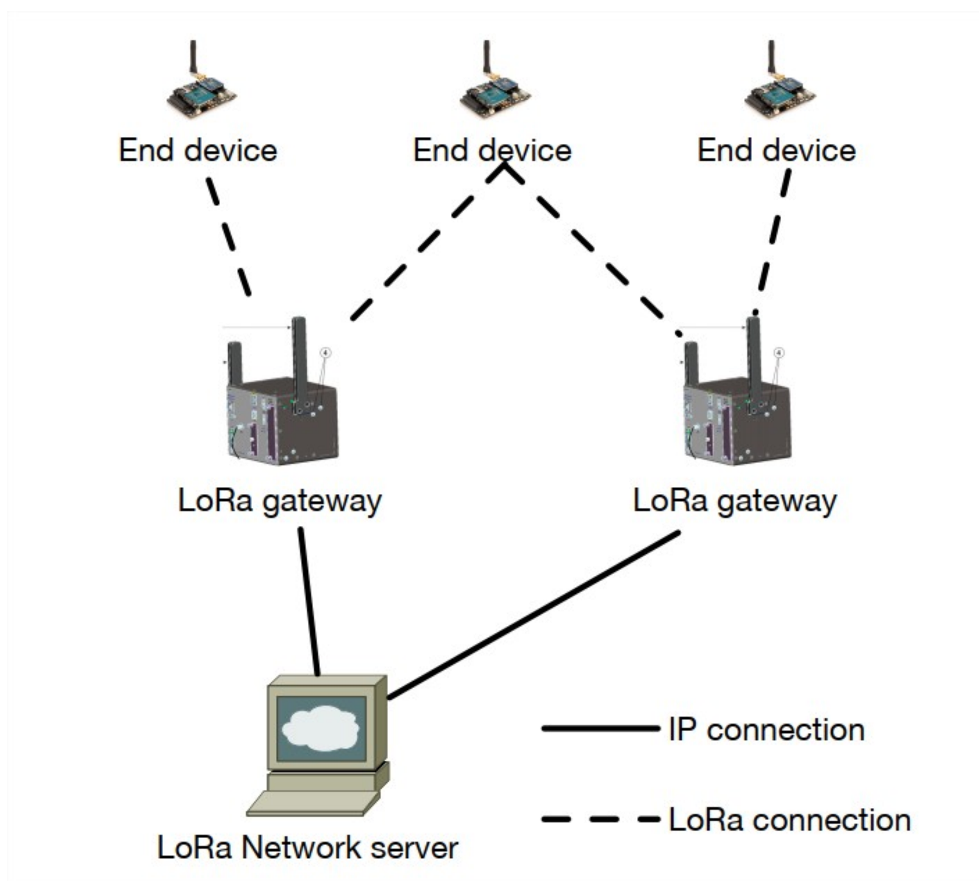

Figura 2.10: Topologia LoRA (AUGUSTIN et al., 2016)

Há várias soluções de *gateways* disponíveis no mercado. Desde soluções comerciais completas, porém podem ser desenvolvidas soluções de baixo custo, utilizando SBCs e *transivers* (PHAM, 2017) (GW. . . , 2019). Os dados recebidos são encaminhados para *middlewares* IoT ou aplicações em nuvem, ou até mesmo armazenados em um banco de dados local, evitando que os dados sejam perdidos caso haja indisponibilidade da internet e o acesso aos dados apenas pela rede local. (GW. . . , 2019)

2.2 *GATEWAYS* IOT

Os *Gateways* IoT têm como função primária, a interconexão de redes de sensores sem fio a rede IP (ZHU et al., 2010)(CHEN; JIA; LI, 2011). As Redes de sensores sem fio são caracterizadas pelo baixo consumo energético, pouca necessidade de transmissão e recepção de dados, e pela capacidade de montar uma rede *Mesh*, fazendo com que dispositivos se comuniquem com

o *gateway* a partir de outros sensores, expandindo o tamanho da rede. Os *Gateways* IoT podem possuir interfaces de comunicação com diferentes tipos de sensores, permitindo por exemplo que dados enviados por sensores ZigBee possam ser encapsulados em protocolos de aplicação TCP/IP como chamadas REST/Full, MQTT, COAP ou qualquer outro protocolo que permita a correta comunicação do *gateway* com o *middleware* (CALDAS FILHO et al., 2017b) .

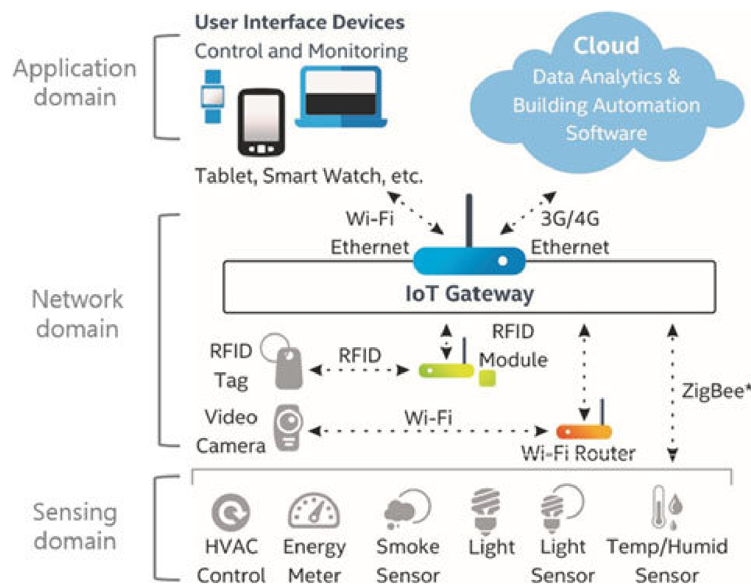

Figura 2.11: Gateway IoT. (KANG; KIM; CHOO, 2017)

Antes de o dado ser encaminhado para a nuvem deve ser encapsulado em uma semântica própria, compreendida pelo *gateway* e pelo *middleware*. Não há um padrão único de semântica comum a todos os ambientes IoT e este fato acaba sendo decisivo para a criação de silos, onde cada fabricante segue a sua proposta, dificultando a interoperabilidade entre os dispositivos IoT (DESAI; SHETH; ANANTHARAM, 2015).

Os *Gateways* IoT devem permitir a correta tradução dos dados recebidos para o *midleware*, permitindo que as aplicações de usuários possam ter acesso a estes dados a apresentá-los aos usuários (CALDAS FILHO et al., 2017a).

Gateways IoT podem ser hardwares dedicados para tal aplicação, realizando a comunicação com até centenas de sensores, ou operar via software, embutido em *smartphones* ou computadores. (ZACHARIAH et al., 2015), permitindo que dispositivos inteligentes vestíveis consigam enviar e receber dados da internet através de um celular (RAHMANI et al., 2015).

Por se tratar de um elemento crítico para garantir a disponibilidade da rede, os *gateways* IoT podem ser instalados em uma configuração que garanta a redundância ou o balanceamento de carga, nos quais mais de um *gateway* é instalado na mesma rede, permitindo a comutação automática e em um curto intervalo de tempo em caso de falha do elemento principal, reduzindo os intervalos de indisponibilidade (RIBEIRO et al., 2018). O tipo de *link* utilizado para comunicação do *gateway* com a internet irá variar de acordo com o volume de sensores a ele associado (KANG; KIM; CHOO, 2017). Os cenários de larga escala e com grande volume de dados exigirão *links* de

maior capacidade. Já os cenários com poucos sensores e baixo volume de tráfego podem utilizar *links* sem fio 3G, satélite.

2.3 ACELERAÇÃO WAN

2.3.1 Definição

A aceleração WAN tem como finalidade reduzir o volume de dados transmitidos em redes de longa distância. As aplicações trabalham mais rapidamente priorizando os dados críticos, otimizando o tráfego TCP por meio da rede com base na utilização de algorítimos de compressão.

Os *appliances* de aceleração WAN, dispositivos com hardware dedicado e software integrado, especificamente projetados para fornecer um recurso de computação para aceleram WAN, trabalham em pares, sendo aplicado um em cada ponta do *link WAN* de comunicação.

Os dispositivos de aceleração que utilizam a camada de rede funcionam em todos os tipos de tráfego e não apenas para o TCP devido ao fato de não utilizarem técnicas de *proxy* e consequentemente nesta modalidade é possível otimizar os arquivos de voz e vídeo que usualmente utilizam o protocolo UDP e podendo ser utilizados em circuitos de alta velocidade pois não possuem limites de sessões como os acelerados de camada 7.

Entre os benefícios que trazidos pela utilização da aceleração WAN estão a redução da latência devido a utilização de protocolos que diminuem as idas e vindas do tráfego de dados, reduzindo a utilização da banda e possível congestionamento com o recurso do cache local, por meio da compressão dos dados e redução da quantidades das informações transmitidas. Assim se reduz a quantidade de retransmissões de pacotes TCPs com erros através da rede superando também a perda de pacotes.

Segundo a Cisco... (2019), as organizações possuem dois desafios opostos: fornecer altos níveis de desempenho de aplicativos cada vez mais distribuídos e uma infraestrutura dispendiosa de uma rede de área ampla (WAN), que apresenta atraso significativo, perda de pacotes, congestionamento e limitações de largura de banda.

O Modelo de topologia da solução de aceleração de WAN cisco Waas (CISCO..., 2019) :

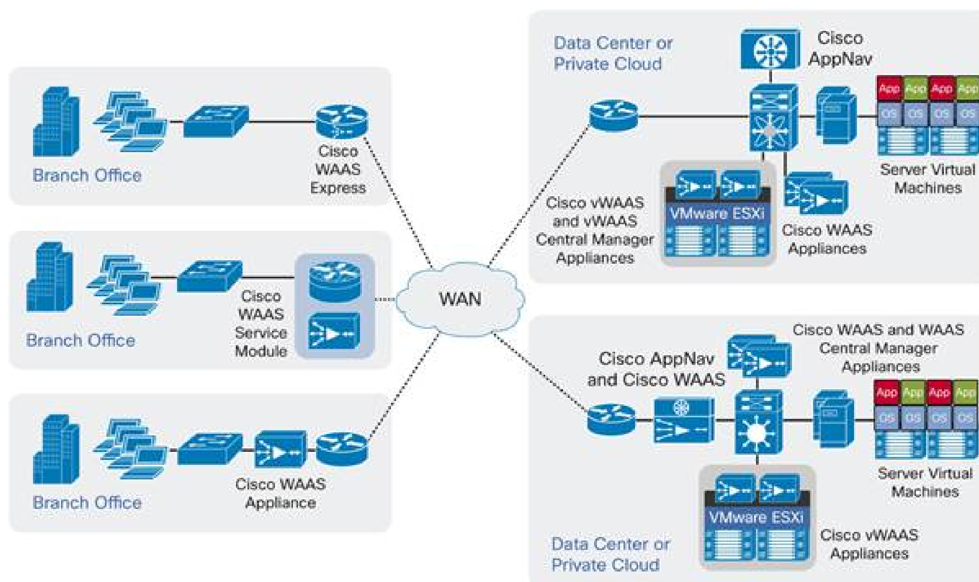

Figura 2.12: Aceleração WAN Cisco (CISCO. . . , 2019).

A técnica de otimização de rede WAN pode utilizar diversos métodos para tornar a aceleração possível tais como:

- otimização por fluxo de transporte;
- compressão;
- aceleração de aplicações;
- serviços de arquivo para aplicações de computadores;
- *mist computing*, realizando o processamento e armazenamento dos dados na borda da rede;
- Virtualização.

As técnicas de otimização WAN não são aplicadas apenas pela Cisco, diversos fabricantes de equipamentos de rede e Firewall oferecem tal funcionalidade. O Fabricante Fornet oferece por exemplo a otimização da transmissão entre *data centers*.

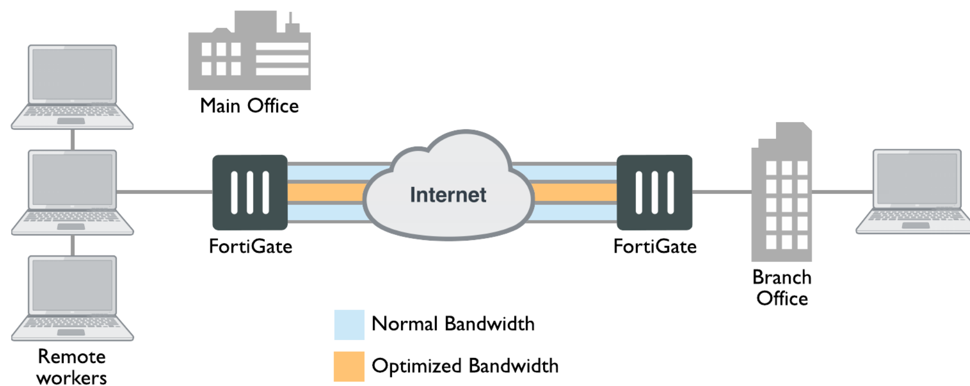

Figura 2.13: Aceleração WAN Fortinet (FORTINET. . . , 2019).

2.3.2 Técnicas de compactação

A compactação de dados trata da redução de um conjunto de símbolos em um tamanho de bytes que ocupará um menor espaço de armazenamento, a esse conjunto de símbolos é dado o nome de dicionário. A compactação promove o agrupamento dos dados que estejam separados reduzindo o seu tamanho consequentemente o espaço que ocupa em um dispositivo e seu tempo de transmissão. No processo de compressão de dicionario são excluídos os símbolos redundantes e é guardada a referência das posições onde o símbolos aparecem duplicados, também reduzindo o tamanho do arquivo porém por meio da exclusão de símbolos e não apenas pelo agrupamento. O dicionário deve ser incluído junto com o arquivo compactado.

A compressão é realizada por meio da utilização de um algorítimo de compactação. Um dos algoritimos utilizado para compactação de dados é o *Huffman*. A codificação de *Huffman* é baseada na frequência de ocorrência de um item de dados (pixel nas imagens). O princípio é usar menos bits para codificar os dados que ocorrem com mais frequência. Os códigos são armazenados em um livro de códigos que pode ser construído para cada imagem ou conjunto de imagens.O livro de códigos e os dados codificados devem ser transmitidos para habilitar a decodificação.

Com a compactação dos quadros de dados, o tamanho desses arquivos é reduzido, diminuindo também o tempo de transmissão dos pacotes por meio da rede poupando recursos de processamento, assim os quadros condensados utilizam menos largura de banda podendo ser transmitido um volume maior de dados em uma única vez.

Segundo (BRAYNER DOROTÉA KARINE D. PITOMBEIRA, 2016), métodos de compactação clássicos como a codificação *Huffman*, a codificação aritmética proposta em Brayner Dorotéa Karine D. Pitombeira (2016) ou a *Lempel-Ziv* não são suficientemente rápidos, pois apresentam um *overhead* de processamento que anula os ganhos de performance obtidos sendo esses métodos eficientes apenas sobre grandes blocos de dados, sendo incompatíveis, portanto, com acessos aleatórios a pequenas quantidades de dados. *Overhead,* em ciência da computação, é geralmente considerado qualquer processamento ou armazenamento em excesso, seja de tempo de compu-

tação, de memória, de largura de banda ou qualquer outro recurso que seja requerido para ser utilizado ou gasto para executar uma determinada tarefa.

Um ambiente ideal para a utilização da compactação ou compreensão de dados é a rotina de *backup*. Para essa atividade é importante que o espaço de um dispositivo seja aproveitado com eficiência visto que um backup fica armazenado por um período de tempo razoável. Redes IoT também podem obter benefícios de técnicas de compactação, haja vista que o volume de dados transmitidos pode ser muito alto e com pouca variabilidade, cenário que possibilita obter grandes taxas de compactação.

2.4 MODELOS COMPUTACIONAIS

2.4.1 *Cloud Computing*

A Computação em Nuvem (*Cloud Computing*) surgiu a partir da demanda emergente de empresas e usuários finais em utilizar aplicações e serviços computacionais na internet sem ter que arcar com os altos custos de infraestrutura, utilizando recursos de armazenamento, processamento, e conectividade hospedados em servidores instalados nos *data centers* de provedores. Antes da oferta desse tipo de serviço, usuários que tivessem demandas de serviços como hospedagem de sites, *e-mail* ou compartilhamento de arquivos precisavam criar e manter uma infraestrutura conectada à internet, ligada vinte e quatro horas por dia, sete dias por semana. Manter um ambiente de *data center* de alta disponibilidade envolve altos custos, tornando inviável para pequenas empresas e usuários domésticos. O mesmo se aplica a outros serviços como *e-mail*, comunicação instantânea, compartilhamento de arquivos, e qualquer outro que precise de conectividade permanente com a nuvem. Para uma pequena empresa manter um servidor com o serviço de *e-mail* rodando 24 horas por dia tem um custo de infra estrutura alto para suportar apenas o seu serviço. Procurando atender à crescente demanda de serviços que precisavam ser hospedados fora do ambiente do cliente, provedores de internet começaram a oferecer em seus *data centers*, a possibilidade de o usuário utilizar os seus recursos computacionais de forma compartilhada, sem ter os custos iniciais de compra de servidores, criação do ambiente com conectividade e manutenção, pagando apenas por recursos consumidos, como espaço em disco, bytes transmitidos e recebidos, alocação de CPUs e memória RAM.

Conforme definição da National Institute of Standards and Technology (NIST), a computação em nuvem é um modelo que permite o acesso conveniente e sob demanda a um conjunto compartilhado de recursos de computação configuráveis (por exemplo, redes, servidores, armazenamento, aplicativos e serviços) que podem ser rapidamente provisionados e liberados com o mínimo esforço de gerenciamento ou interação com o provedor de serviços (MELL; GRANCE et al., 2011).

Segundo Zhang, Cheng e Boutaba (2010), os elementos que compõem a arquitetura de *Cloud Computing* podem ser explicados segundo o modelo de quatro camadas, detalhadas nas subseções

seguintes:

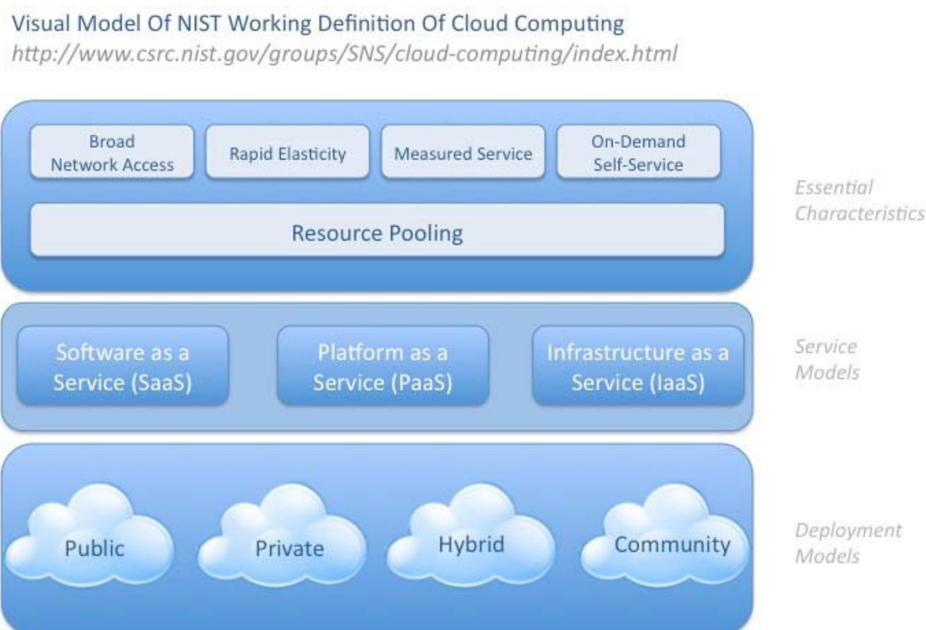

Figura 2.14: Camadas Cloud Computing (KALIM, 2013).

2.4.2 A camada de Hardware / *Data center*

Essa camada comporta todos os equipamentos necessários para garantir o funcionamento e a conectividade do *data center* tais como *nobreaks*, fontes de alimentação, sistema de refrigeração, servidores, roteadores, *switches*, *links* de conexão com a internet. Além dos elementos físicos estão separados nesta camada tarefas e rotinas operacionais necessárias para garantir o funcionamento do ambiente, como a configuração dos elementos de rede, o gerenciamento de tráfego e a gestão da capacidade para adequar a infraestrutura à demanda.

2.4.3 Camada de infraestrutura

Esta camada permite que os recursos disponibilizados na camada de *data center* possam ser utilizados a partir de *Virtual Machines* (VMs) em que cada VM poderá utilizar uma quantidade especifica de núcleos de CPU, memória RAM, espaço em disco e ser destinada para uma aplicação ou sistema específico. Provedores de *Cloud* trabalham com o modelo de negócios chamado de IaaS, *Infrastruture as a Service*, no qual o cliente contrata uma máquina virtual especificando os recursos computacionais que deseja, pagando um valor fixo por mês e podendo expandir a capacidade da maquina de acordo com a sua necessidade.

2.4.4 Camada de plataforma

A camada intermediária da nuvem é utilizada para o desenvolvimento de aplicações e ferramentas para *Web*. Provedores de nuvem oferecem um serviço chamado PaaS, *Plataform as as Service*, em que o usuário aluga a utilização de um hardware com uma quantidade especifica de memória RAM, espaço em disco, núcleos de CPU e largura de banda em uma maquina virtual, com sistema operacional de preferencia do contratante e um conjunto de ferramentas de desenvolvimento como banco de dados, servidor *web* entre outros.

2.4.5 Camada de Software

A última camada é a mais utilizada pelo público de maneira geral. Nela estão todos os aplicativos e programas disponibilizados pela web e acessados por usuários a partir de celulares, *tablets* e em computadores. Nela são oferecidos serviços como editores de documentos como *Google Drive, Office 365*. SaaS ou *Software as a Service* é a camada mais alta da nuvem. SaaS serve como a camada da nuvem que a grande maioria dos consumidores utiliza. Construído sobre o IaaS e o PaaS, o *Software as a Service* fornece aplicativos, programas, software e ferramentas da *Web* ao público, gratuitamente ou cobrando valores de assinatura mensal ou anual.

Ao utilizar aplicações como *Google Play Store*, *App Store*, *Dropbox*, *Salesforce*, *Adobe Cloud Suite*, *Spotify* ou qualquer outro software baseado em nuvem armazenado em um servidor da *Web* localizado em um *data center* do outro lado do mundo, está sendo acessada a camada de nuvem SaaS. A premissa básica do SaaS é o software amigável acessado por meio de um dispositivo de computação de escolha armazenado em um servidor de todo o mundo.

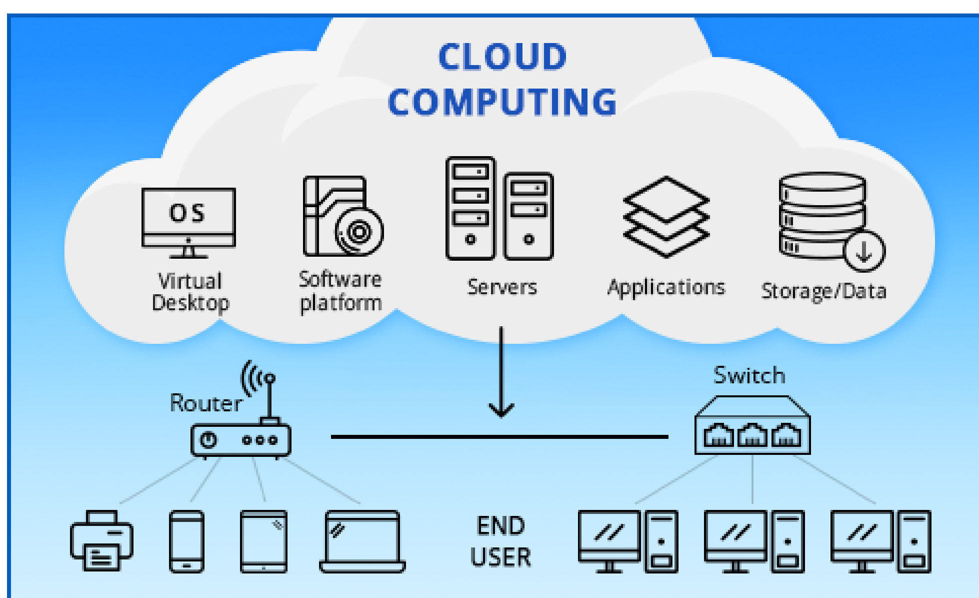

Figura 2.15: Arquitetura *Cloud Computing.* (CLOUD, 2017)

2.4.6 Modelos de nuvem

As nuvens podem ser classificadas de acordo com o modelo de implementação da infraestrutura, podendo ser públicas, privadas ou híbridas (MELL; GRANCE et al., 2011). Esta subseção irá detalhar um pouco mais cada uma delas.

2.4.7 Nuvem Pública

Os recursos de infraestrutura da nuvem pertencem a uma organização, que ofertam os seus serviços para o público em geral acessá-los a partir de conexões com a internet (GOYAL, 2014). Os provedores podem oferecer serviços gratuitos tais como aqueles oferecidos pelo *Google* como *e-mail*, compartilhamento de arquivos e interação com outros usuários em rede sociais. A remuneração pode ocorrer com o pagamento de mensalidades fixas como *Spotify* ou de acordo com a demanda de utilização. Serviços em nuvem pública costumam ter preços menores, pois o custo é rateado com um grande número de assinantes. No entanto, questões envolvendo utilização dos dados de usuários pelos provedores e terceiros, e o vazamento de informações sigilosas faz com que esse modelo de nuvem não seja atrativo para grandes empresas e entidades governamentais. Em 2015 a União Europeia lançou um conjunto de normas *General Data Protection Regulation* (GDPR) que determina como os provedores devem lidar, armazenar e proteger os dados dos usuários, fazendo com que provedores de nuvem publica de todo mundo se adequassem a nova legislação (EU GDPR, 2019).

2.4.8 Nuvem privada

Segundo Mell, Grance et al. (2011), a infraestrutura em nuvem é provisionada para uso exclusivo por uma única organização, sendo que os recursos de *hardware*, armazenamento e rede não são compartilhados. Esses recursos podem pertencer à própria organização ou ser alugado por provedores de *data centers*. Nesse último caso, o fornecedor disponibiliza toda a infraestrutura e gerencia a disponibilidade e os recursos de hardware, porém os dados armazenados são de propriedade do locador. A nuvem privada fornece maior segurança em relação a nuvem pública, dado o seu acesso restrito, e costuma ser utilizada por entidades governamentais, instituições financeiras e comerciais. A infraestrutura em nuvem é provisionada para uso exclusivo por uma única organização, composta por vários consumidores (por exemplo, unidades de negócios). Ele pode ser de propriedade, gerenciada e operada pela organização, por terceiros ou por alguma combinação deles, e pode existir dentro ou fora das instalações.

Uma modalidade de serviços oferecidos em nuvem privada é CaaS, *Colocation as a Service*, as empresas compram os servidores, softwares e equipamentos de rede e os hospedam em um *rack*, residente dentro dentro do *data center* de um terceiro, deixando para o provedor a responsabilidade da energia, controle de temperatura e conectividade (ISHAKIAN et al., 2010).

2.4.9 Nuvem Híbrida

A infraestrutura da nuvem híbrida é composta por duas ou mais nuvens privadas ou públicas (MELL; GRANCE et al., 2011) beneficiando-se da flexibilização,facilidade de acesso e capacidade de expansão de recursos oferecida pela nuvem pública, aliada a robustez e segurança de um ambiente privados. Dados e aplicações sensíveis e sigilosas podem ser mantidas na nuvem privada, enquanto informações não sensíveis podem ser armazenadas na nuvem pública. A orquestração de onde os dados serão gravados e quais recursos serão utilizados fica a cargo de plataformas de gerenciamento de Nuvem como Open Nebula (OPEN. . . , 2019). Essa plataforma possui *drivers* para se conectar a máquinas virtuais locais e a diferentes provedores de nuvem externos, dando liberdade para o administrador escolher onde e como os dados serão armazenados.

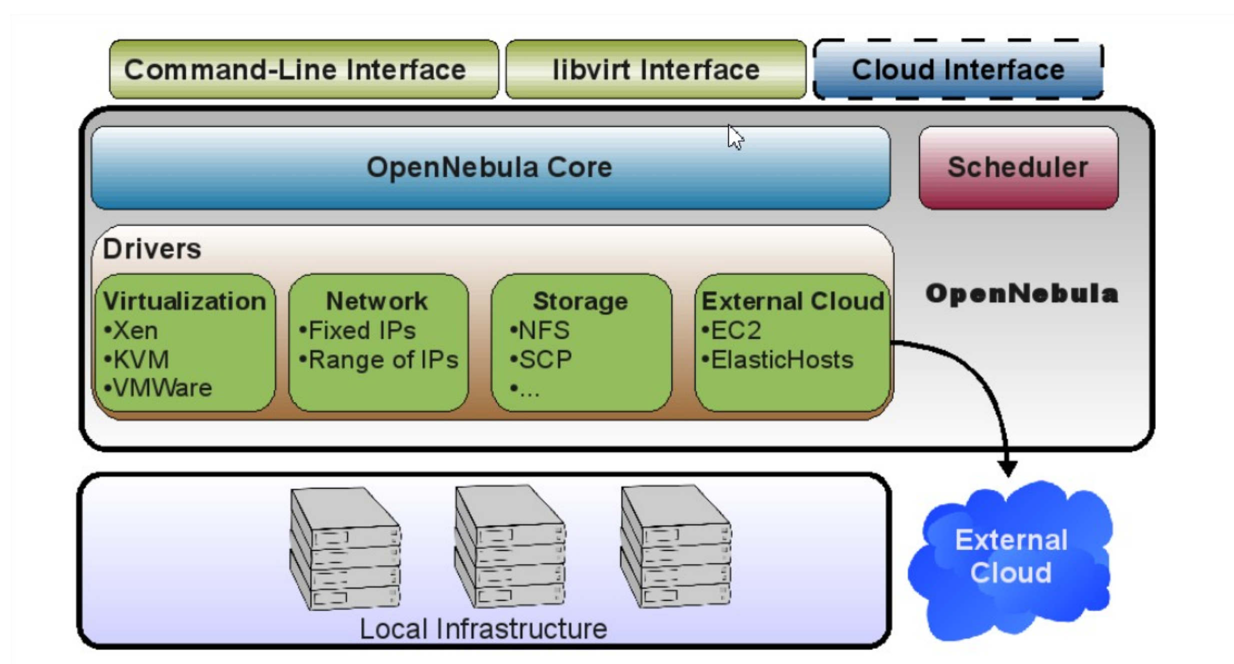

Figura 2.16: Arquitetura do OpenNebula (OPEN. . . , 2019).

Conforme Osman et al. (2017), o modelo de nuvem híbrido fornece baixo custo de aquisição e manutenção para os usuários, permitindo a elaboração de estratégias para melhoria da educação, disponibilizando aos estudantes, recursos virtualmente ilimitados de processamento e armazenamento.

2.4.10 Características dos serviços em nuvem

Conforme Zhang, Cheng e Boutaba (2010), o modelo da computação em nuvem permite um melhor aproveitamento de recursos computacionais graças a características intrínsecas desse serviço, que serão descritas nas próximas subseções

2.4.10.1 Multi-inquilino

Os serviços em nuvem fornecidos ao usuário final podem ser sub locados por diferentes provedores, que se especializam em cada camada. O proprietário de cada camada precisa focar nos objetivos específicos da sua camada, prestando serviço a outros provedores das demais camadas. Grandes provedores podem se especializar em disponibilizar máquinas virtuais (IaaS) para provedores de conteúdo (SaaS). Serviços de *streaming* de vídeo como Netflix fazem uso do modelo de multi-inquilinos e não precisam se preocupar com a infraestrutura na qual sua aplicação está hospedada, apenas em desenvolver e atualizar o software.

2.4.10.2 Conjunto de recursos compartilhados

Os provedores de infraestrutura alocam para um conjunto de consumidores recursos computacionais de CPU, memória, espaço em disco dinamicamente e conforme sua demanda, permitindo melhor aproveitamento da IaaS, maximizando a utilização dos recursos entre os usuários e minimizando os custos de consumo de energia e resfriamento. (ZHANG; CHENG; BOUTABA, 2010)

2.4.10.3 Geo distribuição e acesso de rede onipresente

Provedores de nuvem pública costumam ofertar os seus serviços através da internet, instalando seus *data centers*, em diferentes países, para garantir a disponibilidade e reduzir a latência de acesso. A distribuição geográfica dos *data centers* com objetivo de redução da latência e aumento da disponibilidade dos serviços impulsionou o desenvolvimento de um novo modelo computacional chamado *Edge Computing*, que será abordado nesse capítulo.

2.4.10.4 Provisionamento dinâmico de recursos

Uma das principais vantagens do modelo de computação em nuvem é a capacidade de alocar os recursos dinamicamente, conforme a demanda do usuário, sem a necessidade de intervenção manual. Essa característica beneficia usuários da nuvem em momentos de pico da sua aplicação, ficando a cargo do provedor redimensionar a sua infraestrutura para atender a momentos de grande demanda.

2.4.11 Edge Computing

O modelo de computação em nuvem permitiu a racionalização do processamento e do armazenamento de dados, levando essas tarefas para grandes *data centers* compartilhados, com alta capacidade computacional mas, na maioria das vezes, distantes do usuário final. Construir e manter um *data center* operacional envolve altos custos. O maior *data center* do mundo, *Lakeside Technology Center*, localizado em Chicago, foi construído em uma área de 102 mil metros quadra-

dos, abrigando milhares de servidores, com energia gerada utilizando 50 geradores, alimentado por tanques com 30 mil litros de combustível (WORLD'S..., 2010). Apesar de ser muito potente, todo recurso computacional está concentrado dentro do mesmo ambiente, e usuários de outros continentes que acessam aplicações desse *data center* irão ter problemas com a alta latência. Aplicações em tempo real, vídeo vigilância e *streaming* são sensíveis a atraso na entrega, demandando pontos de armazenamento e processamento mais próximos à origem de onde o dado é gerado.

A solução de *Edge Computing* busca descentralizar parte do processamento e armazenamento de informações, espalhando pequenos *data centers*, nas bordas do *backbone* de grandes operadoras de Telecomunicações, reduzindo assim a latência de comunicação entre as pontas e o volume de dados transmitidos no *core* do *backbone*, favorecendo aplicações IoT que necessitam de respostas rápidas para controle de máquinas industriais, cidades e casas inteligentes (XU et al., 2017).

A Indústria 4.0 (SATURNO; PERTEL; DESCHAMPS, 2017) é um movimento na qual a mudança na forma de produzir está sendo impulsionada pelas novas tecnologias, como IoT, *Cloud Computing*, e a comunicação maquina para máquina (M2M). Essas novas tecnologias demandam uma comunicação de baixa latência e alta disponibilidade, caraterísticas podem ser ofertadas por soluções de *Edge Computing*.

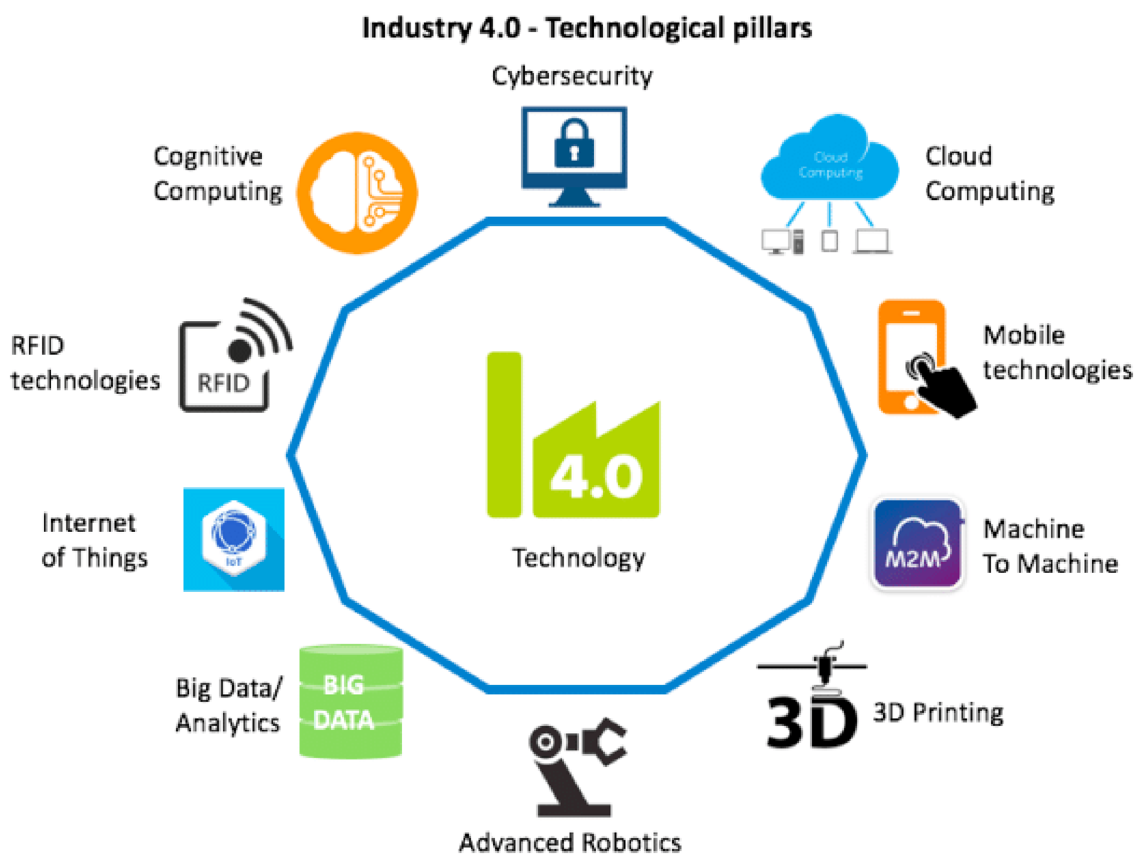

Figura 2.17: Industria 4.0 (SHROUF; ORDIERES; MIRAGLIOTTA, 2014).

Conforme estimativas de Shi e Dustdar (2016), haverá 50 bilhões de dispositivos inteligentes conectados à internet até 2020. Aplicações IoT podem exigir um tempo de resposta muito curto e gerar um volume considerável de dados, e a computação em nuvem pode não fornecer o desempenho necessário.

As técnicas de *Edge Computing* relacionam a utilização dos recursos de nuvem nos perímetros da rede com o objetivo de processar dados prioritários mais próximo à borda da rede, transmitindo as informações a um dispositivo próximo a exemplo de um *gateway* que pode processar os dados de forma mais rápida e posteriormente enviar para armazenamento na nuvem.

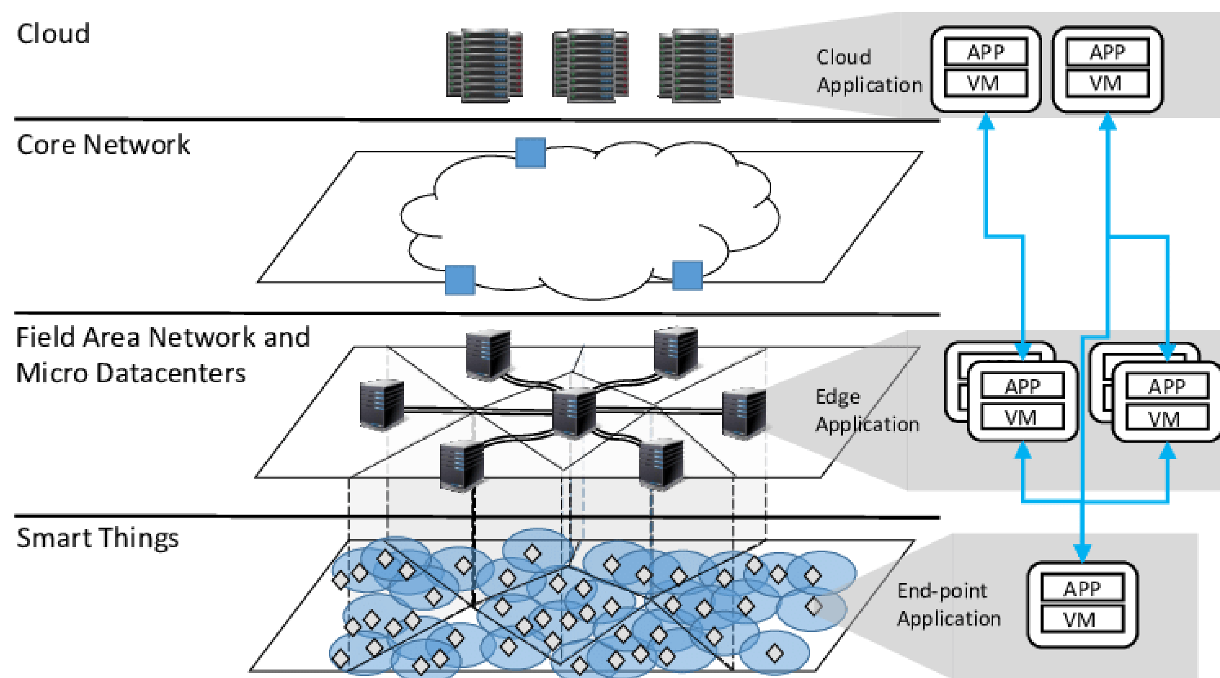

Figura 2.18: Arquitetura *Edge*

Para (SHI et al., 2016), a computação de borda refere-se às tecnologias de ativação que permitem com que a computação seja executada na borda da rede, em dados *downstream* em nome de serviços em nuvem, e *upstream* dados em nome dos serviços IoT. As aplicações IoT utilizam desses recursos para maior rapidez e processamento dos dados, visto que é possível utilizar de recursos como programação, em linguagem de máquina entre outros recursos com menor distância.

Segundo Gartner, Inc. (2016), a borda é posicionada como um *design* topológico para entregar a *Internet of Things* (IoT), uma alternativa descentralizada para a nuvem, ou como uma construção de topologia de alto nível para economizar dinheiro e reduzir a latência.

Taleb et al. (2017) se refere a um serviço em nuvem mais próximo ao usuário, no qual é abordado a otimização do processamento e armazenamento com redução do consumo dos recursos e menor latência, devido à proximidade com a borda da rede. Os autores utilizaram como base dois estudos de caso demostrado pela arquitetura descrita, baseada numa estrutura autônoma de serviços para permitir acessos a usuários de qualquer localização por meio de diferentes recursos

de *edge computing*.

2.5 *FOG COMPUTING*

A Internet das Coisas possui grande dependência da Computação em Nuvem. Os dispositivos inteligentes coletam os dados utilizando diferentes tipos de sensores e os enviam para os servidores na nuvem, que se encarregam de processar e persistir as informações. Os servidores hospedados em *data centers* possuem poder computacional infinitamente superior ao de dispositivos inteligentes e conseguem atender à demanda de vários dispositivos simultaneamente. Entretanto, o processo de envio e processamento remoto incrementa a latência, prejudicando operações de tempo real e desperdiçando as capacidades de processamento e armazenamento que tanto avançaram nos últimos anos nos aparelhos dos usuários como celulares, TVs inteligentes, Câmeras de vigilância e dispositivos IoT inteligentes. O modelo *Fog Computing* traz a proposta de deixar parte das tarefas de armazenamento e processamento mais próximo dos usuários, em *data centers* instalados em rede local mais próximo possível da fonte de dados.

Esta arquitetura permite o melhor aproveitamento de recursos computacionais distribuídos, além da redução da latência, dado que boa parte do processamento e comunicação ocorre perto dos dispositivos (TANEJA; DAVY, 2016).

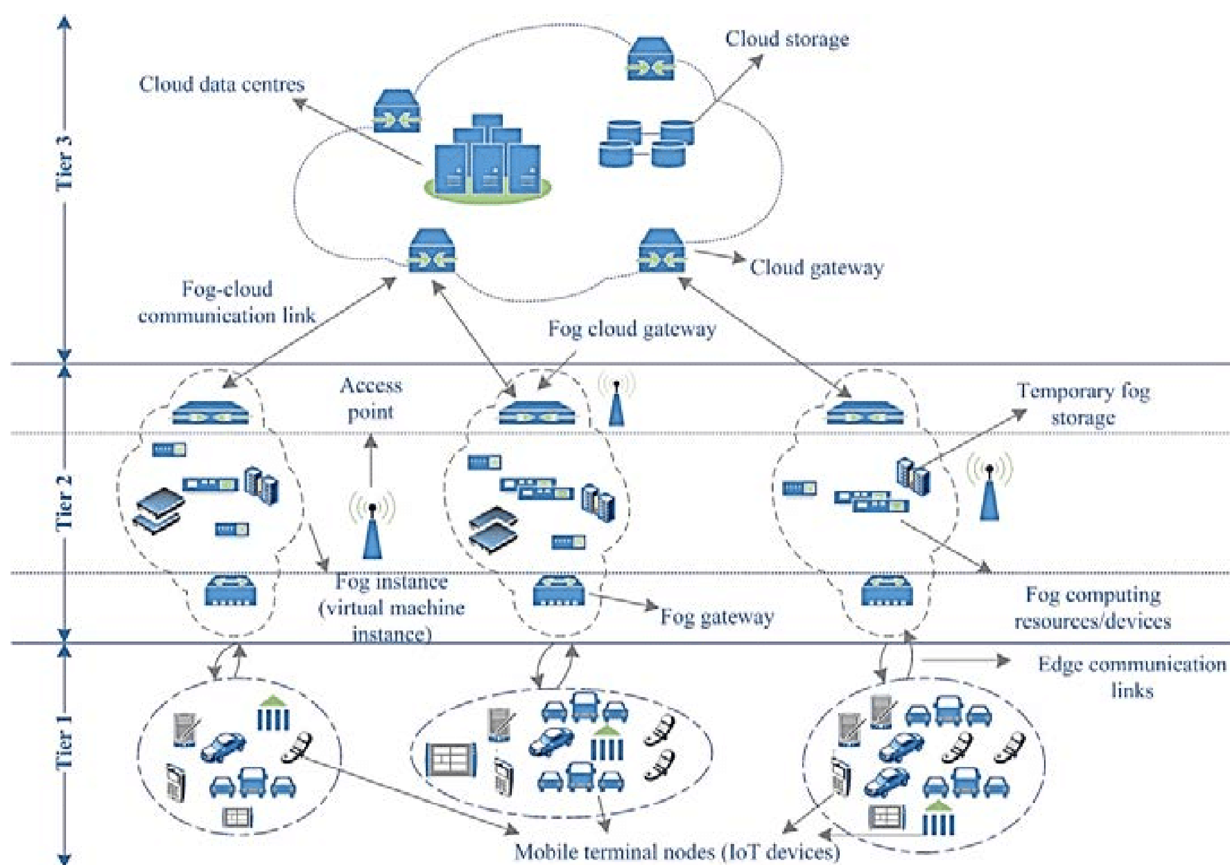

Figura 2.19: Arquitetura *Fog* (TANEJA; DAVY, 2016).

Fog computing em sua tradução "computação em nevoeiro", trata uma rede descentralizada e mais próxima a fonte de transmissão dos dados estando localizada entre os dispositivos de hardware, dispositivos/sensores IoT e a nuvem.

Segundo Veerraju e Kumar (2018), os componentes de rede permitem que a computação Fog crie grandes distribuições geográficas de serviços baseados em nuvem. A computação *Fog* facilita a localização, suporte de mobilidade, interações em tempo real, escalabilidade e interoperabilidade.

Segundo Stojmenovic e Wen (2014), na computação de nevoeiro, os serviços podem ser hospedados em dispositivos finais como receptores de TV *set-top-boxes* ou pontos de acesso. A infraestrutura dessa nova computação distribuída permite que os aplicativos sejam executados o mais próximo possível de dados acionáveis e massivos detectados,de pessoas, processos e coisas. A computação em névoa, na verdade acaba se tornando uma computação em nuvem perto do "solo".

Schenfeld (2017) descreve *Fog computing* como uma estrutura computacional que possui como principais finalidades a melhora da eficiência e redução do volume de dados transmitidos que serão processados, analisados e, após conclusão desses processos, armazenados na nuvem. Nessa tecnologia o processamento é realizado no próprio dispositivo que gera os dados onde o recurso de*Fog computing* está localizado dentro de um ambiente distribuído que utiliza diversos recursos de integração de hardware e software. A arquitetura de *Fog computing* está representada na Figura 2.20.

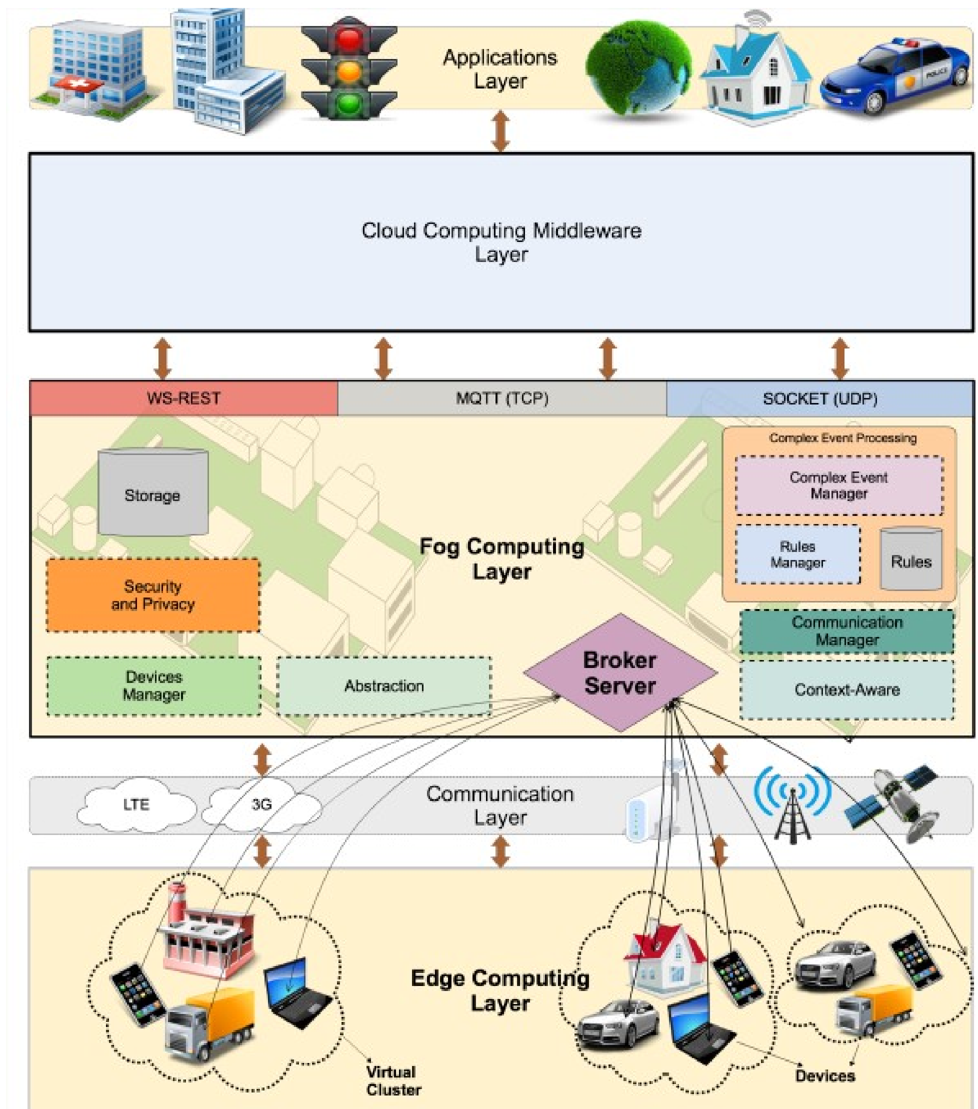

Figura 2.20: Arquitetura *Fog* e *Edge* (SCHENFELD, 2017)

O termo *Fog Computing* foi criado pela Cisco em 2012 é um paradigma de computação distribuída, que aproveita a capacidade de processamento e armazenamento dos dispositivos de rede em diferentes níveis hierárquicos, reduzindo a necessidade do envio de dados para serem processados na nuvem, ocasionando a redução da latência, pois o processamento e armazenamento ocorre mais próximo das "coisas", beneficiando aplicações em tempo real como processamento de imagens,reconhecimento facial, monitoração do trânsito, entre outros.

Sistemas de controle industrial, como *Smart Grids*, óleo e gás precisam que a latência entre os sensores e o nó central de controle seja de alguns ms (WEINER et al., 2014) e apesar do avanço dos sistemas de telecomunicações uma latência tão baixa não se tornou factível para todos os lugares os lugares do planeta.

Segundo Sarkar, Chatterjee e Misra (2018), os dispositivos de computação *Fog* fazem tarefas de escopo local e armazenamento em pequena escala, reduzindo a dependência de consultas e

gravações de dados na nuvem. No entanto, os pedidos que exigem armazenamento permanente ou análises envolvendo conjuntos de dados históricos (por exemplo, dados de mídia social, fotos, vídeos, histórico médico, *backups* de dados, etc.) são encaminhados para os servidores remotos, agindo como *gateways* para redirecionar os pedidos para o núcleo da estrutura de computação em nuvem, assim sendo, a computação de nevoeiro não é um substituto de computação em nuvem, em vez disso, essas duas tecnologias complementam uma outra. As funções complementares de nuvem e *Fog* permitem que usuários finais experimentem uma nova geração de computação, tecnologia que atende aos requisitos dos aplicativos IoT em tempo real e de baixa latência executados na borda da rede.

2.6 NOVOS DESAFIOS DE ARQUITETURA IOT

Segundo (CHIANG; ZHANG, 2016), as mudanças trazidas pela Internet das Coisas impõe alguns desafios que apenas o cenário tradicional de computação em nuvem não consegue resolver. O volume de dados transmitidos por sensores está crescendo de forma exponencial e o fato de precisar transmitir todos esses dados para nuvem pode se tornar insustentável. Ambientes industriais, onde os sensores e atuadores precisam de respostas em ms para poder tomar decisões, podem não conseguir essa latência aguardando a resposta de um servidor na nuvem.A Internet, principal canal utilizado para comunicação entre os *smart devices* e a nuvem, não está onipresente em todos os cantos do planeta. A indústria do agronegócio está se utilizando cada dia mais dispositivos IoT, entretanto os ambientes rurais costumam ficar afastados dos grandes centros urbanos, e a conexão com a Internet possui baixa capacidade de transmissão e alto tempo de resposta. Campos de petróleo e gás remotos costumam ser atendidos por *links* satélites, porém esta tecnologia de transmissão costuma ser sensível a chuvas e pode não estar disponível o tempo todo. Nesta seção serão expostos alguns dos problemas que estão reforçando a utilização dos cenários de processamento distribuído para ambientes IoT.

A Internet das Coisas traz consigo obstáculos a serem superados e desafios a serem vencidos através da conexão de dispositivos do cotidiano utilizando sensores, microeletrônica e linguagens de programação de forma a garantir a monitoração e uma melhor utilização dos recursos a fim de atender as necessidades de diversos seguimentos como cidades e carros inteligentes (LATRE et al., 2016).

Segundo (PERERA et al., 2014), "a Internet das Coisas tem o potencial de mudar o mundo, assim como a Internet fez, talvez até mais.”A arquitetura de IoT possui diversas possibilidades de conexão cada uma com suas necessidades de adequação e seus próprios desafios, entre elas pode-se mencionar o *bluetooth* que fornece comunicação sem fio, porém como muitas outras tecnologias possui limitações a exemplo da distância entre os dispositivos para realizar sua conexão, o *wifi*, o 4G ou 4,5 G, O ethernet, que possui a necessidade de ser cabeado entre diversas outras possibilidades com os seus respectivos desafios a serem superados.

De acordo com Gluhak et al. (2011), a maioria dos experimentos em IoT seguem uma estrutura de duas ou três camadas. Os projetos em IoT costumam possuir duas camadas nas quais a primeira é basicamente composta de sensores que coletam os dados e a segunda camada formada por servidores nos quais os dados são processados, armazenados e utilizados pelas aplicações.

As redes IoT possuem desafios pertinentes a seus diversos segmentos. As *Smart Cities* e a IoT têm características particulares do ponto de vista das infraestruturas e das aplicações. (ATZORI; IERA; MORABITO, 2010)

A infraestrutura de internet das coisas é formada por uma heterogeneidade de dispositivos que contem capacidade e características diferentes e recursos restritos que devem de alguma forma conectar entre si permitindo a comunicação contínua entre equipamentos eletrônicos com os seus sensores, atuadores e dispositivos a uma rede que permite recolher, trocar e interpretar as informações geradas.(BORGIA, 2014)

2.6.1 Restrições de largura de banda

Atualmente há mais dispositivos inteligentes conectados à Internet do que seres humanos no planeta, e esse crescimento exponencial está pressionando operadoras de Telecomunicações para disponibilizar aos usuários finais soluções de conectividade mais robustas e onipresentes, com maior largura de banda (KELLY, 2015). Carros conectados geram dezenas de Megabytes por segundo, transmitindo informações dos sensores, rotas por onde passam, velocidade instantânea, consumo de combustível e outros dados de monitoração do veiculo. Sistemas de vídeo vigilância em tempo real que utilizam câmeras de alta resolução também podem consumir centenas de Megabytes por segundo para enviar as gravações em tempo real para uma central remota. O Serviço de *Smart Grid* dos EUA gera um valor aproximado de 1.000 petabytes a cada ano (CHIANG; ZHANG, 2016) a Biblioteca do Congresso dos EUA gerou cerca de 2,4 petabytes de dados por mês, o tráfego do *Google* cerca de um petabyte por mês e a rede da AT e T consumida 200 petabytes por ano em 2010 (COCHRANE, 2010).

Esse grande volume de dados não precisa necessariamente ser transmitido para servidores na internet. ABI Pesquisas estimam que 90 % dos dados gerados pelos dispositivos finais serão armazenados e processados localmente, e não na nuvem (KELLY, 2015), reforçando a utilização de paradigmas de *Fog* e *Edge Computing*. Parte destes dados que precisam ser transmitidos, podem ter o volume reduzido por meio de técnicas de aceleração WAN.

2.6.2 Conectividade intermitente com a nuvem

Segundo Taneja e Davy (2016) provedores de nuvem tem dificuldade em fornecer ininterruptamente serviços para dispositivos e sistemas que tenham conectividade de rede intermitente com a Internet. Tais dispositivos incluem veículos, *drones* e plataformas de petróleo. Por exemplo, uma plataforma de petróleo no oceano pode utilizar como canais de comunicação apenas *links*

satélites. Este tipo de canal apresenta baixa disponibilidade em locais nublados e em dias de chuva, deixando o ambiente remoto isolado nos momentos de intempérie. No entanto, aplicativos como coleta de dados, análise de dados, e controles para a plataforma de petróleo têm que estar disponíveis mesmo quando o equipamento não tem conectividade de rede com a nuvem.

Grandes produtoras agropecuárias, instaladas em fazendas longe dos centros urbanos, monitoram os animais utilizando sensores que medem o deslocamento e a saúde do animal. A comunicação de cada sensor diretamente com a internet se torna inviável, seja pelos altos custos associados, seja pela falta de infraestrutura adequada no campo para tal. Assim sendo, os sensores enviam dados para um *gateway*, que por sua vez também possui restrições no envio de dados e pode operar por um bom tempo sem se comunicar com a nuvem. Esses *gateways*, diferentes dos utilizados em redes convencionais, que apenas repassam a informação e possuem um *buffer* de dados limitado, precisa persistir o dado até que se tenha conectividade novamente. Outra opção adotada, é o armazenamento na rede local, que permita o armazenamento e sincronismo com a nuvem após o retorno da conectividade.

Entretanto, há um conjunto de dispositivos que não necessitam conectividade constante com a internet e foram criados para interagir com o contato com serem humanos. *Totens* de auto atendimento em lojas, serviços como aluguel de bicicletas e patinetes utilizam redes de dados oportunísticas (GUO et al., 2013) em que a ação do objeto inteligente depende da interação com um usuário. Após ocorrer o primeiro contato, diversas ações e interações podem ser feitas no ambiente em nuvem ou no dispositivo, como a liberação de algum produto ou serviço.

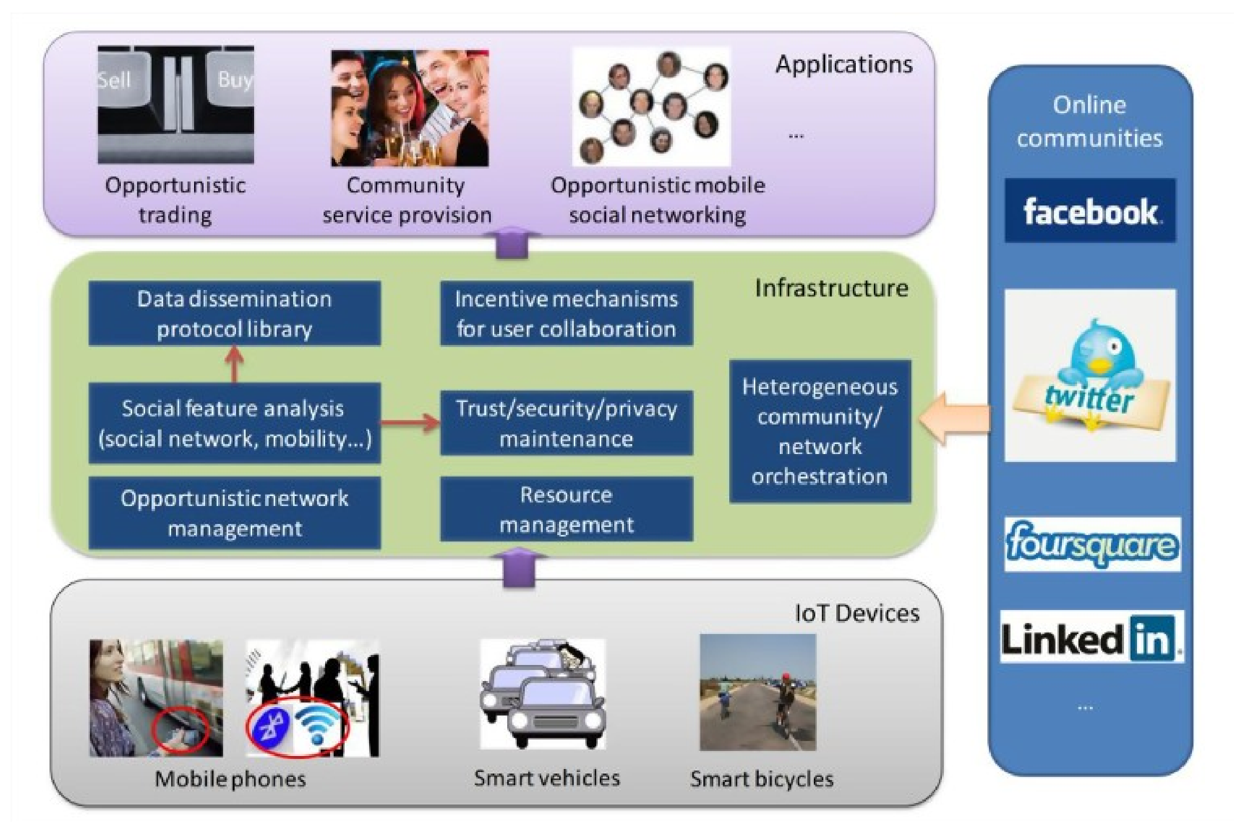

Figura 2.21: Redes Oportunísticas (GUO et al., 2013).

2.6.3 Aumento da capacidade computacional dos dispositivos

A capacidade de armazenamento e processamento dos aparelhos eletrônicos vem crescendo ao longo dos anos. Há uma década celulares conseguiam armazenar alguns megabytes, hoje, podem vir de fábrica com gigabytes de memória RAM e CPU de vários núcleos, superando o poder de processamento e armazenamento de computadores pessoais e até servidores de décadas atrás. Essa evolução não trouxe melhorias apenas para celulares, dispositivos de Internet das coisas também estão equipados com CPUs mais rápidas e com mais núcleos e maior capacidade de armazenamento. Estes recursos podem ser aproveitados, montando assim uma rede distribuída.

Conforme Varghese et al. (2018), a distribuição de tarefas computacionais de forma orquestrada, aliando processamento e armazenamento local e a baixa latência na comunicação entre os dispositivos beneficia cenários como o de cidades inteligentes, cuidados pessoais com a saúde, indústria 4.0.

Figura 2.22: Cenários de *Fog Computing* (VARGHESE et al., 2018)

2.6.4 Evolução da arquitetura IoT com *Fog*

Conforme Chiang e Zhang (2016), a arquitetura em névoa pode contribuir preenchendo lacunas tecnológicas encontradas em ambientes IoT, distribuindo a computação, controle e armazenamento para dispositivos mais próximos do usuário, reduzindo a dependência de comunicação com a nuvem, a latência e aproveitando a capacidade dos equipamentos próximos aos usuários. Assim sendo, as técnicas baseadas em *Fog* poderão aproximar parte das tomadas de decisão do usuário, realizando o pré-processamento dos dados, o controle e atuação de dispositivos mais próximo das

bordas. As técnicas baseadas em *Fog* permitem o aumento da disponibilidade do serviço, pois falhas na nuvem ou na comunicação entre elas não irão afetar diretamente o serviço.

Assim sendo, os serviços de Nuvem e *Fog* são interdependentes e complementares, pois a grande massa de dados armazenados e o volume analítico precisa ser processado em grandes computadores.

Segundo Chaouchi, Bourgeau e Kirci (2013), a IoT faz referência a interconexão de objetos do nosso cotidiano por meio da web, para que isso ocorra é necessário a adaptação dos protocolos existentes, considerando que existem paradigmas a serem superados para que a rede de objetos inteligentes atenda aos padrões impostos pela rede global. A internet das coisas possibilita controlar dispositivos do nosso dia a dia utilizando recursos como sensores e o emprego de microeletrônica. Promove dessa forma, novos conceitos e novas possibilidades de implantação de recursos, usufruindo de elementos que fazem parte da rotina diária da grande maioria da população mundial. Quando são conectados objetos que utilizam diferentes elementos a uma única rede, são abertas portas para o surgimento de novas aplicações e tecnologias de maneira geral, Internet das coisas significa unir objetos à Internet e esta ação gera uma gama de possibilidade de sua utilização na qual os objetos podem fornecer comunicação entre usuários e dispositivos.

O IoT em um ambiente de nuvem acarreta uma utilização maior da largura de banda por uma grande quantidade de dispositivos à borda da nuvem, causando grandes congestionamento na rede. Diante deste fato, compor a internet das coisas com a inclusão de *Fog computing* determina uma redução no envio de dados à nuvem, consequentemente também diminui seu processamento. A proximidade dos dispositivos IoT da borda da nuvem, "nevoeiro", reduz consequentemente, os pontos de falhas e a latência Bonomi et al. (2012).

Bonomi et al. (2012) defende que a visão de nevoeiro foi concebida para endereçar aplicativos e serviços que não se encaixam bem no paradigma da nuvem. A *Fog computing* e a internet das coisas realizam em conjunto a distribuição adequada e eficiente dos recursos entre a rede local e a nuvem. Os dispositivos e sensores realizam o processamento no qual os dados são gerados e coletados mas, esses equipamentos não possuem recursos de armazenamento e é nesta função que se se faz o emprego do *Fog computing* em que Bonomi et al. (2012) conceitua que a evolução da combinação de IoT e *Fog Computing* resulta em baixa latência na rede, distribuição geográfica ampla¸ mobilidade, grande número de nós, além da heterogeneidade de dispositivos.

Conforme descrito em Chiang e Zhang (2016), as arquiteturas de *Fog* e *Cloud* são complementares e a escolha de qual paradigma deve ser utilizado depende dos requisitos da aplicação. Os principais fatores que devem ser levados em consideração são descritos na Tabela 2.1.

Tabela 2.1: Comparação entre Soluções *Cloud vs Edge.*

Características	*Cloud*	*Fog*
Localização e modelo computacional	Centralizado em um pequeno numero de grandes *data centers*	Distribuído em diferentes localidades
		ocupa diversas áreas geográficas
		próximo aos usuários
Tamanho	Grandes *data centers* podem conter até milhares de servidores	Pequenos nós instalados próximo a origem dos dados
Instalação	Requer grande planejamento, com trabalho de equipe multidisciplar e especializada	Baixo planejamento e em alguns casos basta ligar e usar
Operação	Operação complexa, exigindo mão de obra qualificada e cara	Operação pode ser autonoma
Aplicação	Aplicações não sensíveis a latência	Aplicações sensíveis a latência
	Aplicações que consomem grandes recursos computacionais	
Conectividade com a internet e requisitos de banda	Conexão constante com a Nuvem	Conexão intermitente com a rede
	Grande exigência de largura de banda	Baixo consumo de dados

Chiang e Zhang (2016).

3 PROPOSTA DE GATEWAY FOG COM ACELERAÇÃO WAN

3.1 ARQUITETURA PROPOSTA

A arquitetura proposta nesta dissertação descreve uma rede IoT fim a fim, com sensores que captam as informações do ambiente, um *gateway* capaz de receber mensagens em diversos protocolos de comunicação encaminhando os dados através de API REST para o *middleware*. O *middleware* pode estar instalado tanto na nuvem ou em um ambiente de rede local. Cabe ao *middleware* a responsabilidade de persistir os dados e permitir que o usuário controle atuadores remotamente. A capacidade do *gateway* receber dados por diferentes canais e repassar para um *middleware* em nuvem foi detalhada em (CALDAS FILHO et al., 2017a) (CALDAS FILHO et al., 2017b) (MARTINS et al., 2017b)

Os componentes eco sistema UIoT estão detalhados conforme arquitetura detalhada na figura abaixo:

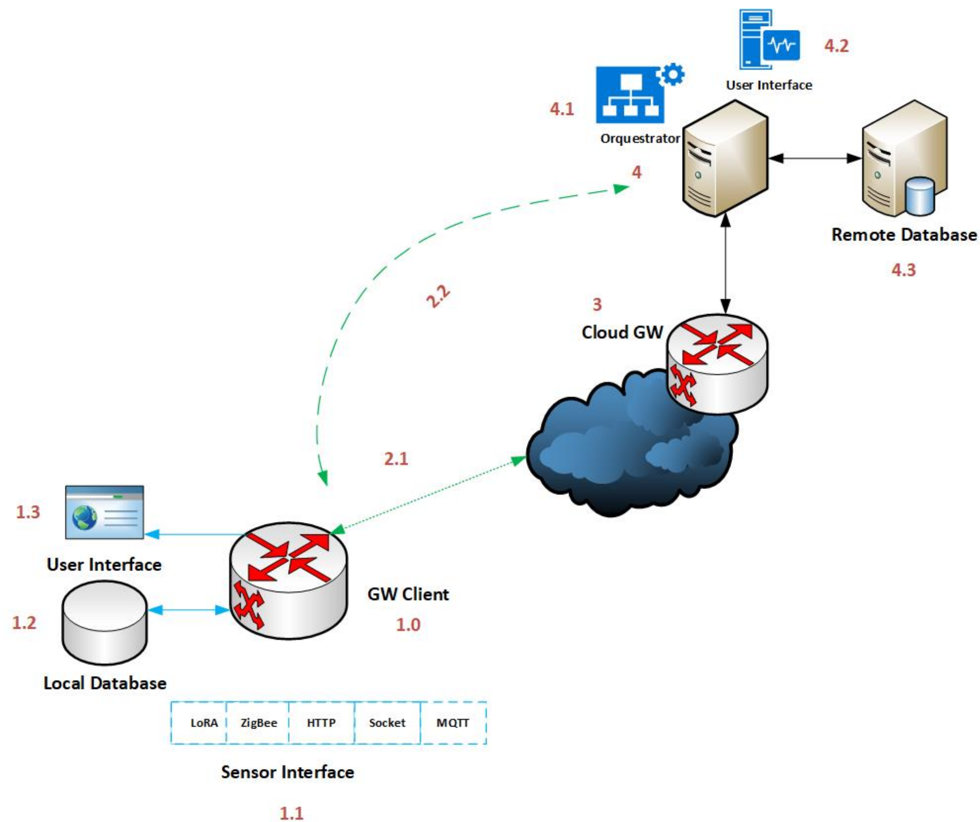

Figura 3.1: Componentes da Solução UIoT.

Os componentes estão enumerados na figura para facilitar seu detalhamento ao longo do Capitulo.

3.2 *GATEWAY* CLIENTE

O *gateway* cliente fica instalado no ambiente remoto e tem como função básica receber os dados dos sensores e atuadores e repassa-los para a nuvem. As funcionalidades e interfaces de comunicação com sensores estão sendo incrementadas desde 2015. Em (RIBEIRO et al., 2018) foi desenvolvida uma estrutura para aumentar a disponibilidade da comunicação entre os sensores e a nuvem, acrescentando um *gateway* redundante e permitindo a comutação automática em caso de falha. Visualizamos entretanto, que a arquitetura de redundância garantia a disponibilidade do serviço porém não aproveitava plenamente os recursos computacionais do *gateway* de Backup, deixando para esse, apenas a responsabilidade de monitorar o estado da rede e assumir o encaminhamento de pacotes em caso de falha. Desenvolvemos então uma arquitetura em *Cluster*, onde o elemento de *backup* realiza tarefas de processamento e armazenamento solicitadas pelo orquestrador, deixando o *Gateway* principal apenas com a responsabilidade de encaminhar os pacotes e se comunicar com o *Middleware*.

A forma como os dados são transmitidos ou armazenados são definidos por regras criadas no orquestrador em nuvem e propagadas para o *gateways* remotos. O orquestrador determina o período que os dados são armazenados localmente e pode alterar a forma como os dados são enviados, impondo as seguintes modificações:

- Alterar a frequência de envio de dados por cada serviço, associado a um sensor ou dispositivo inteligente, reduzindo o volume de dados transmitidos e armazenados na nuvem;
- Processar os dados de um período, enviando média, mínimos, máximos ou quartil de um período pré-determinado, reduzindo a carga de processamento na nuvem;
- Criar alertas a partir dos dados dos sensores, indicando situações anormais que seriam ignoradas pelas regras de sumarização.

O orquestrador pode determinar, por exemplo, que os dados de temperatura de um determinado sensor, sejam representados pela média das amostras recebidas na última hora, porém se os valores lidos estiverem acima de um limiar pré-estabelecido, uma mensagem de alerta deve ser enviada imediatamente para o orquestrador, avisando um eventual problema. As regras que poderão ser aplicadas serão detalhadas na seção correspondente.

3.2.1 Sensor Interface

O *gateway* proposto tem a capacidade de receber dados oriundos de redes de sensores sem fio, *ZigBee* e *LoRA*, assim como de dispositivos inteligentes com capacidade de transmissão de

dados em redes IP, utilizado para isto os protocolos HTTP, MQTT e *Sockets* TCP e UDP. A comunicação entre os dispositivos e o *gateway* utilizando esses canais, são descritas nas seções correspondentes.

Para este experimento foram montados diversos protótipos funcionais de dispositivos inteligentes que foram integrados à arquitetura proposta. Estes são compostos por microcontroladores, sensores, transmissor / receptor de dados. O microcontrolador será responsável por fazer a leitura dos sensores, processar o dados e transmitir para o *gateway*. Entre os sensores utilizados neste protótipo estão os de temperatura e umidade do ambiente (DHT11), Pressão atmosférica (BMP180), luminosidade (LDR), microfones para medir o ruido do ambiente, detectores de chuva, sensores de corrente elétrica não invasivos (ST-013), de poluição atmosférica, entre outros. Estes dispositivos foram espalhados pelo *Campus* e se comunicam utilizando diferentes tecnologias de transmissão como *Ethernet*, *Wirelles 802.11*, *ZigBee*, Lora, *Bluetooth*.

O *gateway*, foi implantado em um computador de placa única Raspberry Pi 3, modelo B. Foram adicionados ao *gateway* interfaces para comunicação ZigBee, *LoRA* e um receptor 433 MHz capaz de receber dados dos sensores. O sistema operacional instalado no *gateway* foi o Raspbian, uma versão do Linux adaptada para o *Raspberry*. A aplicação converte os dados recebidos pelos sensores em chamadas *REST* Full, requisições utilizando o protocolo HTTP, para o *Middleware* e foi desenvolvida em *Python* 3.

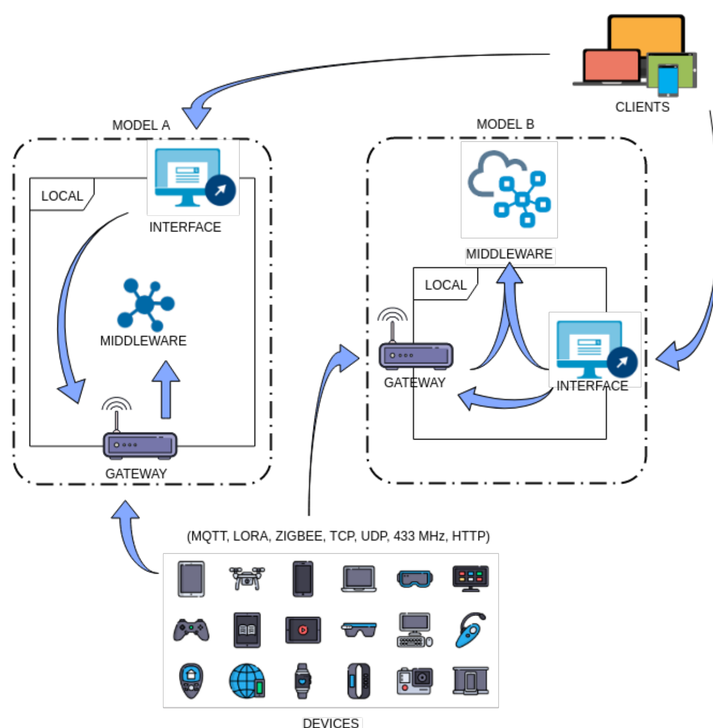

Figura 3.2: Modelagem do conjunto *gateway*, *middleware* e interface. O *gateway* recebe dados de dispositivos de diferentes protocolos de comunicação.

3.2.2 Interface HTTP

Dispositivos inteligentes com cliente HTTP instalado podem repassar os valores das suas leituras ou receber comandos oriundos do *Middleware* utilizando o protocolo o protocolo *HyperText Transfer Protocol*. O *gateway* funciona como um intermediário entre o objeto inteligente e *Middleware* remoto, retirando do dispositivo as responsabilidades da autenticação na rede IoT, realizando *cache* dos dados, e obedecendo as regras estabelecidas pelo controlador para envio dos dados.

A aplicação do *gateway* levanta um serviço HTTP local e aguarda o recebimento de requisições em caminhos específicos para realizar o registro de novos sensores e serviços e o repasse dos dados para o *Middleware*. Abaixo estão listados os caminhos acessíveis no *gateway* para este protocolo:

- **/client** [**POST**] : Registra um cliente no *middleware*;
- **/service** [**POST**] : Registra um serviço no *buffer* do *gateway* e no *middleware*;
- **/data** [**POST**] : Registra um dado no *middleware*. Se o dado não for categorizado como sensível e conter regras de compressão / supressão então este dado é mantido no *buffer* do *gateway*;
- **/rules** [**POST**] : Registra uma regra de algum determinado serviço no *buffer* do *gateway*;
- **/list/rules** [**GET**] : Lista regras a partir dos parâmetros de busca informados através da *URL*.

3.2.3 Interface MQTT

O *gateway* UIoT implementa uma interface MQTT, desenvolvida em python, responsável pela comunicação com os dispositivos inteligentes na rede local. A troca de mensagens é realizada com operações de *publisher* e *subscriber* em tópicos específicos. As mensagens recebidas serão enfileiradas para serem repassadas ao *Middleware* em conjunto com outras mensagens recebidas de outras interfaces. Para dar suporte ao protocolo *MQTT* é utilizada a ferramenta *Mosquitto* (LIGHT, 2017). O *gateway* recebe dados válidos publicados na porta 1883 de cliente, serviço e dado bruto.

3.2.4 Interface UDP

O *Gateway* UIoT implementou uma interface UDP como forma de garantir a agilidade na transmissão dos dados críticos. Dispositivos inteligentes podem marcar os seus dados como urgentes e esses serão encaminhados para a nuvem imediatamente, sem passar pelo processo de enfileiramento descrito anteriormente. A transmissão de pacotes UDP não é orientada a conexão, sendo mais ágil que a transmissão HTTP ou socket TCP. O *gateway* recebe dados válidos enviados na porta 5005 de cliente, serviço e dado bruto.

3.2.5 Interface TCP

O *Gateway* UIoT também possui uma interface TCP, responsável por receber dados dos sensores na porta 5010 e repassar para o *Middleware*. Importante ressaltar que independente da interface utilizada, a sintaxe das mensagens são averiguadas antes de encaminha-las para o *Middleware*. Caso haja um preenchimento indevido dos campos, a mensagem é descartada.

3.2.6 Interface ZigBee

O *gateway* recebe dados válidos através de um módulo conectado no *gateway* por uma porta USB serial a uma taxa de transmissão de 9600 bps. Os sensores ZigBee são previamente configurados para enviar os dados para a interface conectada ao *Gateway*.

3.3 CONTROLE DE ADMISSÃO DE *SMART DEVICES* NA REDE UIOT

Os dispositivos inteligentes precisam ser aceitos na rede UIoT para que possam enviar os dados das suas leituras ou serem controlados remotamente. O processo de admissão e controle ocorre por intermédio do *gateway*, que recebe os atributos informados pelos dispositivos e os repassa para o *middleware*. Com a posse dessas informações, o *middleware* irá gerar um `ID` único de identificação.

Esta chave irá servir para associar os dispositivos aos seus respectivos serviços. São atributos obrigatórios:

- Nome do dispositivo. Exemplo: Raspberry PI;
- Chipset. Exemplo: AMD 790FX;
- Endereço MAC. Exemplo:FF:FF:FF;
- Serial do dispositivo. Exemplo: C210;
- Nome do processador. Exemplo: Intel I3;
- Canal de comunicação. Exemplo: Ethernet.

3.3.1 Controle de admissão de serviços na rede UIoT

Após realizar o registro do dispositivo, será realizada a identificação dos serviços contidos no dispositivo inteligente. Cada serviço representa um tipo de leitura que pode ser oferecido pelo sensor. Um serviço precisa conter as seguintes informações:

- Nome do serviço. Exemplo: Get temp;

- Chipset. Exemplo: AMD 790FX;
- Endereço MAC. Exemplo: FF:FF:FF;
- Número do serviço. Exemplo: 3;
- Tipo de dado que o serviço irá coletar. Exemplo: temperatura.

3.3.2 Formato das mensagens transmitidas entre os *Smart Devices* e o *gateway*

O padrão de mensagens transmitidas entre os dispositivos e o *gateway* obedece ao padrão listado abaixo:

- Informação sobre a sensibilidade deste dado. Exemplo: 1;
- Chipset. Exemplo: AMD 790FX;
- Endereço MAC. Exemplo: FF:FF:FF;
- Número do serviço. Exemplo: 3;
- Valores brutos. Exemplo: [20.3, 50.0].

3.4 BASE DE DADOS LOCAL

A base de dados local tem a responsabilidade de persistir os dados recebidos pelos sensores inteligentes, servindo como cache temporária para a nuvem em caso de falhas de transmissão, permitindo o envio dos dados solicitados pelo *Middleware* de acordo com as regras cadastradas no orquestrador, beneficiando links de internet intermitentes. Outro beneficio direto é a divisão de responsabilidades em armazenar informações do sensores. O *Middleware* poderá armazenar os dados sumarizados, aplicando regras de mínimo, média, máximos e quartis, enquanto o *gateway* poderá armazenar o dado bruto, sem aplicar qualquer tipo sumarização, por um intervalo de tempo determinado pelo *Middleware*. O padrão do armazenamento no *gateway* são 24 horas, podendo ser maior mas não sendo tão interessante devido a limitações de armazenamento das SBCs.

3.5 INTERFACE DE INTERAÇÃO LOCAL COM O USUÁRIO

A interface do usuário desenvolvida no *gateway* permite que o usuário visualize os dispositivos inteligentes e serviços que estão enviando dados para ele. Também permite a visualização das regras de envio cadastradas no orquestrador e atribuídas a sensores e serviços de responsabilidade do *gateway*, além do controle de atuadores.

3.6 INTERFACES DE COMUNICAÇÃO *GATEWAY - MIDDLEWARE*

Há dois modelos de utilização do conjunto *gateway* e *middleware*, no primeiro modelo, mostrado na Figura 3.2 como modelo A, o *gateway* recebe os dados dos sensores na rede local e estabelece comunicação com o *middleware* no mesmo ambiente. No modelo B, os dados dos sensores são enviados para um *middleware* na nuvem.

Logo, caso o *middleware* esteja instalado na internet a comunicação ocorrerá com a utilização de um *link* de Internet, utilizando chamadas REST.

3.6.1 Envio de dados para o *middleware*

Os dispositivos IoTs precisam atender algumas premissas para transmitir os dados para o *middleware*, são elas:

1. Possuir um transmissor e receptor compatível com uma das tecnologias de transmissão suportadas pelo *gateway*;
2. Encapsular os dados lidos pelos sensores em formato `JSON`;
3. Ser aceito como membro da rede pelo *gateway* a partir do processo de auto-registro (SILVA et al., 2016a).

Para alcançar o terceiro passo é necessário seguir a hierarquia apresentada na Figura 3.3. Após atender as premissas listadas acima, o *smart device* está apto a transmitir os dados dos sensores ou de ser controlado remotamente pelo *middleware*.

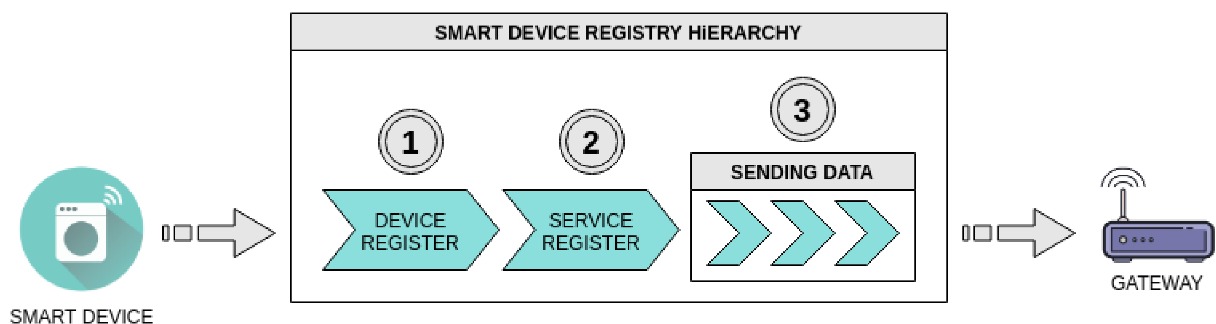

Figura 3.3: Hierarquia de cadastro do *smart device* no *gateway* para ser aceito como membro da rede.

O encaminhamento dos dados é feito utilizando chamadas REST, que são validadas pelo *Gateway* público na sua sintaxe. Dispositivos ligados diretamente na Internet podem enviar dados diretamente para o *gateway* público. Foi desenvolvido algumas estações meteorológicas autônomas que operam por 3G, que enviam dados diretamente para a nuvem.

3.6.2 Interface de comunicação com o orquestrador

Além de enviar e encaminhar os dados recebidos dos sensores e dispositivos inteligentes, o *gateway* consulta periodicamente o orquestrador para averiguar se há alguma nova regra de transmissão associada a determinado sensor ou ao *gateway*. Regras de sumarização de dados são aplicadas a serviços de um determinado *smart device*, regras de compactação de dados estão associadas ao *gateway*. Nesse último caso o orquestrador informa o numero de mensagens que devem ser enfileiradas antes de serem compactadas.

3.7 ORQUESTRADOR

Conforme explicado em (SHI; DUSTDAR, 2016), uma das funções da arquitetura em *Edge Computing* é realizar parte do processamento mais próximo das bordas da rede, delegando a *Cloudlets*, pequenos *Data Centers* espalhados na rede ou a elementos com maior capacidade computacional, (Gartner, Inc., 2016) as tarefas de processar e armazenar parte das informações dos sensores, reduzindo assim a latência para executar tarefas de leitura e escrita nos dados, possibilitando tomadas de decisão mais ágeis pelos sensores e atuadores nas pontas. A decisão de onde o dado estará armazenado é determinada no controlador central, entidade responsável por orquestrar as tomadas de decisão por parte das outras entidades, trazendo flexibilidade para toda a arquitetura.

O controlador central é a entidade que contem as regras de comportamento que os *gateways* UIoT devem obedecer. Na solução proposta, podemos cadastrar um conjunto de regras que serão detalhadas nas seções seguintes.

3.7.1 Descarte de amostras na origem

Podemos cadastrar no *orquestrador* a frequência com que os dados de um ou mais serviços serão repassados para a nuvem pelo *gateway*. Dessa maneira podemos reduzir a quantidade de amostras que um determinado serviço entrega, descartando os dados na origem, economizando assim banda de transmissão. O encaminhado dos dados utilizando regras definidas no orquestrador podem ocorrer de duas maneiras, com perdas ou sem perdas de amostras. No cenário com perdas, os dados recebidos na rede local pelo *gateway* durante o período sugerido de supressão não são encaminhados para *middleware* e são descartados na origem. Supondo que um sensor envie dados de temperatura a cada minuto e foi atribuída uma regra determinando que os dados sejam repassados apenas a cada 10 minutos, as nove amostras recebidas antes do intervalo de 10 minutos serão descartadas. Já no cenário 2, caso o sensor envie o dado a cada minuto e a regra esteja cadastrada para envia-la em 10, o Gateway UIoT irá enfileirar e transmitir as 10 mensagens em uma única troca de comunicação com o *Middleware*. O encaminhamento de várias amostras em uma única mensagem trás alguns benefícios como a redução do processo de negociação TCP

e *Three Way HandeShake*. As amostras de um determinado período podem ser resumidas pelo *gateway* com fórmulas matemáticas antes de serem repassadas conforme será detalhado na próxima subseção.

3.7.2 Envio de dados com supressão para o *middleware*

A rede IoT desenvolvida neste projeto permite que o usuário cadastre regras de comunicação entre o *gateway* e o *middleware*, alterando a forma como essas duas entidades se comunicam, podendo modificar a frequência que um determinado serviço envia os dados e / ou definir fórmulas matemáticas que serão aplicadas a um conjunto de amostras antes destas serem transmitidas, sumarizando a informação antes do seu envio e realizando, processamento e armazenamento dos dados nas pontas.

Para permitir o encaminhamento imediato de determinadas mensagens trocadas entre os sensores e o *gateway*, sem que essas passem por técnicas de compressão ou supressão, onde pode ocorrer aumento da latência de transmissão, foi criado um campo de prioridade. Mensagens de serviços preenchidos com o numeral um nesse campo são imediatamente transmitidas para o *middleware*. Esta técnica, semelhante a aplicada em pacotes `IP` em redes com QoS (IETF, 2006) permite que dados críticos ou alarmes enviados pelos sensores cheguem à *Cloud* e possam ser processados com maior agilidade.

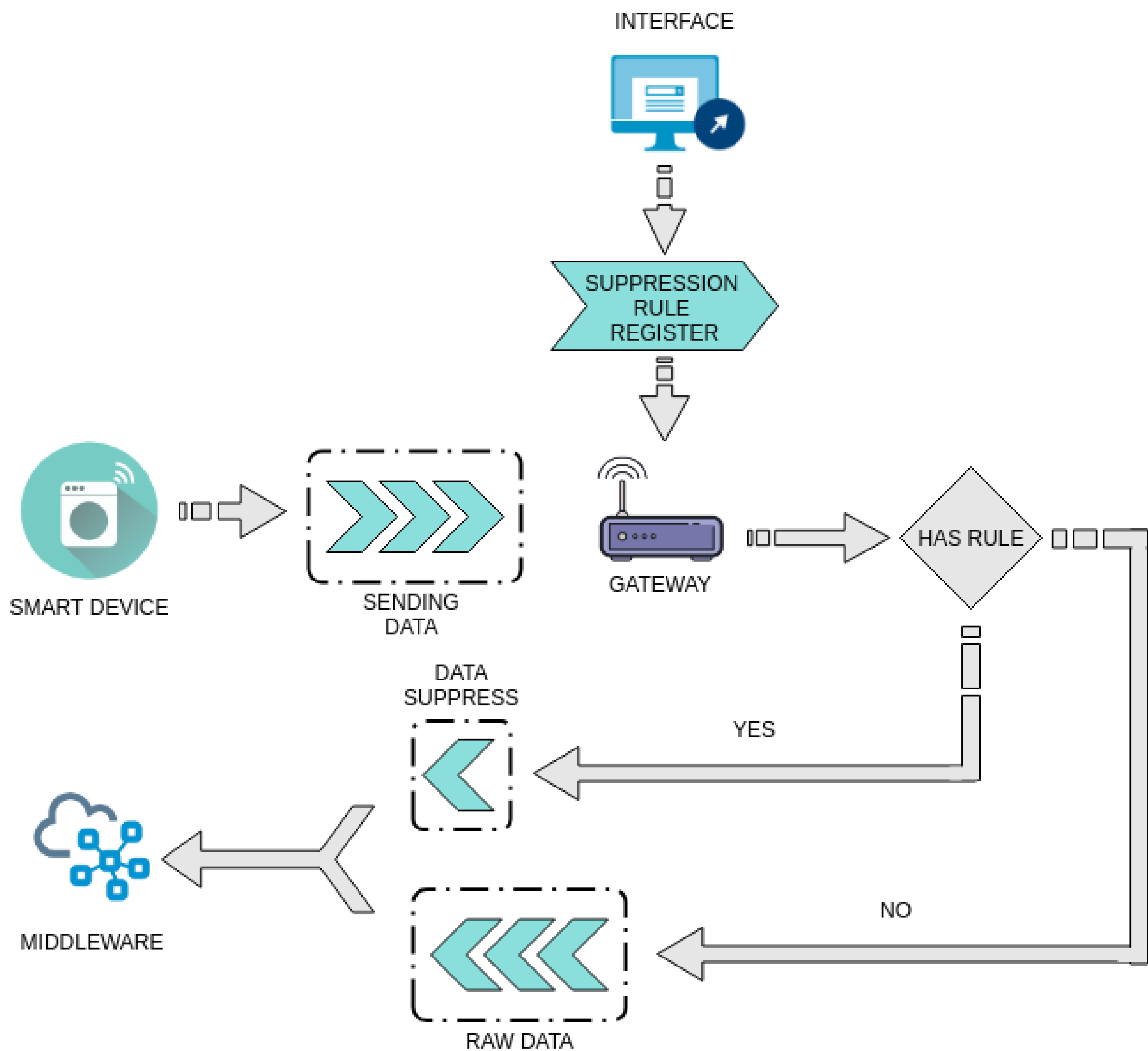

Figura 3.4: Fluxo de cadastro de regra de supressão.

As regras de supressão estão associadas a um determinado serviço e podem ser cadastradas na interface gráfica do orquestrador (Figura 3.4).

- **Cadastro de regras pela interface**
  As regras listadas abaixo acumulam dados no *buffer* do *gateway* e as enviam comprimidas para o *middleware* depois de alcançado seus objetivos. As regras 1 e 2 não podem ser enviadas juntas, pois são regras excludentes.

  1. Regra de envio de dados por frequência de unidade. Exemplo: 10 unidades;
  2. Regra de envio de dados por frequência de tempo (unidade de minuto). Exemplo: 5 minutos;
  3. Regra de envio de dados por data/hora. Existem 2 formatos para esta regra suprimir sem perda:

- Dia específico sem intervalo. Exemplo: 28-07-2018T15:55Z;
- Dia da semana sem intervalo. Exemplo: satT12:00Z.

- **Número prefixado no *gateway***
  É possível configurar um número para compressão sem perda no *gateway*. Antes de enviar qualquer dado para o *middleware* através de regras pré-cadastradas ou dados não sensíveis sem regras, o *gateway* verifica se há algum número prefixado para compressão e o mantém no *buffer* do *gateway*. Quando este número é alcançado ele comprime todos os dados encontrados no *buffer* e os envia em uma requisição HTTP POST para o *middleware* como é mostrado na Figura 3.5.

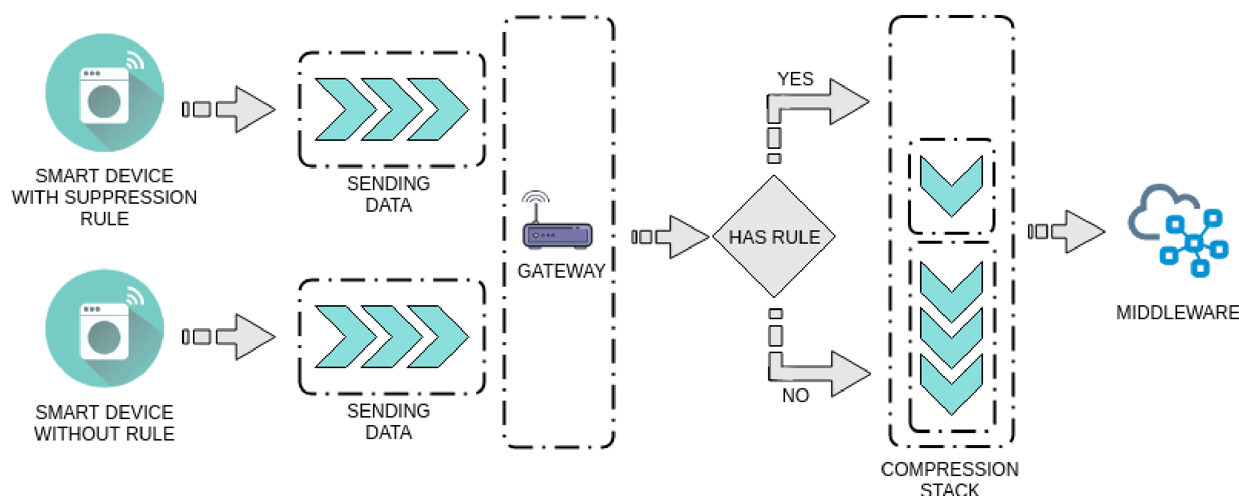

Figura 3.5: Fluxo de dados com barramento para compressão de dados sem perda.

3.7.3 Aplicação de filtros

A solução proposta permite que dados associados a um determinado serviço sejam reduzidos, obedecendo a critérios estabelecidos pelo administrador da rede, antes da transmissão. Os filtros permitem que os dados encaminhados para nuvem estejam dentro de politicas pré-estabelecidas. Caso o administrador da rede entenda que valores médios da ultima hora podem representar o conjunto de dados de um determinado sensor, este pode cadastrar uma regra a associar ao serviço desejado.

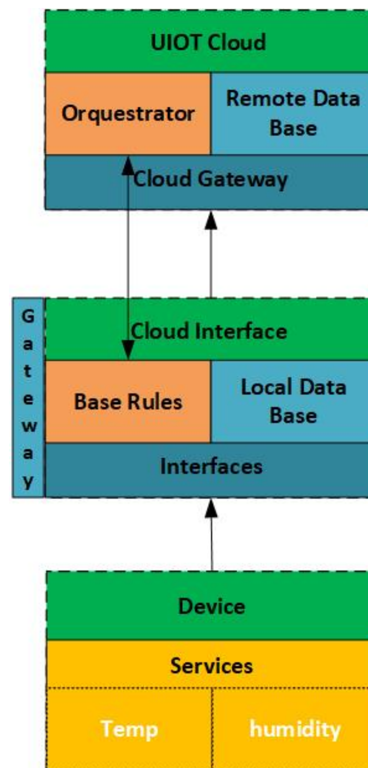

Figura 3.6: Interação entre os componentes

Assim sendo, as fórmulas estão associadas a serviços específicos e são aplicadas a um conjunto de dados em um intervalo de tempo. As fórmulas matemáticas determinam como um conjunto de dados deverá ser resumido antes de ser enviado ao *Middleware*, podendo ser:

1. Média - Apresenta a média aritmética simples de um conjunto de dados;
2. Mínimo - Apresenta o menor valor do intervalo selecionado;
3. Máximo - Apresenta o maior valor do intervalo selecionado;
4. Terceiro Quartil- O valor do terceiro quartil do conjunto de amostras no intervalo selecionado.

O tamanho do conjunto de amostras de um determinado serviço está diretamente relacionado à frequência com que os dados são enviados pelos dispositivos. Um dispositivo inteligente, com um serviço de temperatura que envia o dado no intervalo de dois minutos, teremos as respectivas quantidades dentro de diferentes intervalos:

	60 min	**180 Min**	**600 Min**	**1440 Min**
Qte amostras	30	60	300	720

Tabela 3.1: Tabela de Resultados.

seja $S(t)$ a quantidade total de amostras recebidas pelo *gateway* dentro de um intervalo de tempo t, na frequência x temos:

$$S(t) = \frac{t}{x} \tag{3.1}$$

seja $C(t)$ o conjunto amostral de tamanho S recebido pelo *gateway* em um intervalo de tempo [0,t] em que:

$$C(t) = \{c_1, c_2, ...c_S(t)\} \tag{3.2}$$

A quantidade de amostras que serão enviadas para a nuvem obedecem a seguinte formula:

1. $M_t = \sum_{i=1}^{n} C_n/S$
2. $\min_t = \min C(t)$
3. $\max_t = \max C(t)$
4. $Quartil_3 = CQ_3 = X$

Assim sendo, o conjunto de dados transmitidos será o resultado das operações matemáticas, e não mais o conjunto com todos valores enviados pelo dispositivo inteligente, reduzindo o volume de dados transmitidos para a nuvem apenas a uma mensagem. A agregação dos dados pelo *gateway* reduz a quantidade de mensagens que precisam ser transmitidas e consequente redução na utilização do *link* de comunicação com a nuvem, beneficiando cenários que o *link* WAN é cobrado por bytes transmitidos. A agregação entretanto pode trazer perda de informação caso seja aplicada em longos períodos onde há grandes variações das leituras. Seja C o conjunto de dados obtidos do sensor de temperatura de uma sala no intervalo de uma hora, enviados em uma frequência de cinco minutos, assim sendo:

$$S(5) = \frac{60}{5} = 12 \tag{3.3}$$

O conjunto C é formado por 12 amostras listadas abaixo:

$$C = \{20, 21, 22, 20, 23, 20, 20, 20, 20, 21, 20, 20\}$$

Assim sendo:

1. $MC(t) = 20,58$;
2. $C(t)Q_3 = 21$;
3. Desvio padrão C(t) = 0,99

As formulas de media e terceiro quartil conseguem representar de forma próxima os valores contidos em C ao longo do tempo. Este fato pode ser constado a partir do desvio padrão das amostras com valor próximo a zero. Entretanto, caso C tenha os valores:

$$C = \{20, 21, 23, 24, 24, 25, 25, 26, 27, 29, 30, 32\}$$

1. $MC(t) = 25,5$
2. $C(t)Q_3 = 28,5$
3. Desvio padrão $C(t) = 3,55$

Verificamos que os resumos de média e terceiro quartil apresentam variação maior em relação as leituras de C, o que é evidenciado pelo maior valor de desvio padrão. A diferença entre os valores lidos e as suas representações irão apresentar maior distorção para leituras com grandes variações e com intervalos de tempo amostral maiores. Por isso é importante que o administrador consiga escolher o tamanho amostral corretamente. Um dos trabalhos futuros desta solução é a escolha de qual fórmula utilizar para agregar que represente menos perda de informação dentro de um intervalo de tempo.

3.7.4 Compressão de Dados

O usuário poderá cadastrar no orquestrador regras que agrupem um conjunto de mensagens, aplicando técnicas de compressão antes de fazer o envio para o *Middleware*. Várias mensagens são agrupadas em pacotes de tamanho fixo, que podem ter como origem o mesmo serviço ou um conjunto de serviços. O tamanho dos pacotes pode variar entre 100 a 300 mensagens que serão compactadas e transmitidas ao Middleware. Caso a quantidade de mensagens pre determinadas não seja atingida, os pacotes serão formados e enviados com a quantidade que estiver em fila,depois que o tempo limite de fila for excedido.

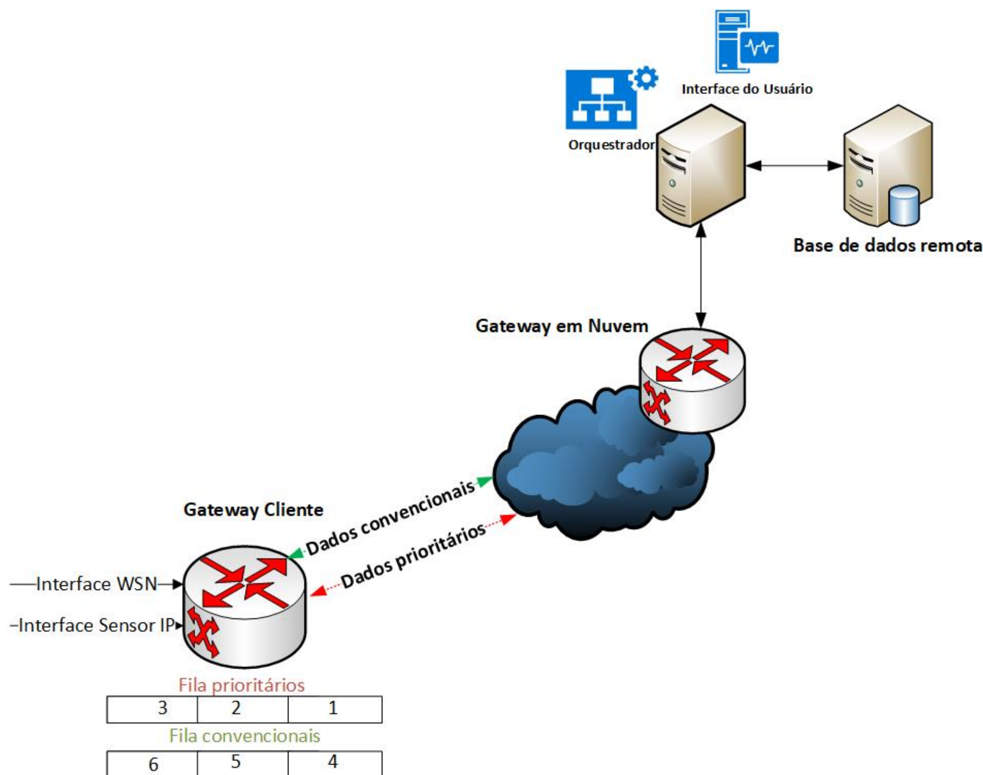

Figura 3.7: Filas de transmissão*gateway*

3.7.5 Envio de dados para o *middleware* com compressão

As mensagens trocadas entre o *gateway* e os *Smart Devices* seguem uma semântica e ontologia pré estabelecida, com pouca variação nos caracteres utilizados. Arquivos de texto com dados semelhantes são excelentes candidatos para aplicar algorítimos de compressão, apresentando taxas altas de compactação.

No modelo de transmissão de dados com compressão o *gateway* recebe as mensagens dos diferentes *Smart Devices* associados, armazenando em um *buffer* lógico de tamanho pré fixado, antes de realizar o encaminhamento para a *Cloud* formando filas de dados a serem transmitidos. As mensagens que chegam são comprimidas utilizando o algoritmo *LempelZiv* (ZIV; LEMPEL, 1978) e encaminhadas em sua totalidade para o *middleware*.

Além da redução dos dados transmitidos, há um outro benefício em agrupar os dados em filas e transmiti-los de forma simultânea. A comunicação entre o *gateway* e o *middleware* é realizada via REST HTTP, e ao encaminhar uma mensagem ao *middleware*, o *gateway* precisa encapsular o dado nesse protocolo, estabelecer uma conexão com o servidor e no final realizar a transmissão. Este processo precisa ser realizado para todo e qualquer dado recebido pelo *gateway*, fazendo com que este precise abrir diversas conexões simultâneas com a nuvem. Ao transmitir mensagens em pequenos blocos, o *gateway* reduz a quantidade de conexões abertas com o *middleware*.

3.7.6 Regras para aplicação de filtros

As regras cadastradas no orquestrador permitem a redução no envio de dados enviados pela nuvem, possibilitando que a informação de um determinado sensor seja recuperada na nuvem, sem a necessidade de envio de todos os dados e, sem haver perda de informação.

1. Regras de compressão sem perda listadas na Seção 3.7.5 com fórmula aplicada. As seguintes regras são disponibilizadas: cálculo da média dos valores, valor máximo, valor mínimo e terceiro quartil (75%). Exemplo: 1 para média;

2. Regra de envio de dados por gatilho de valor. Esta regra associa um operador e uma métrica para envio ao *middleware*. Exemplo: {“”! = ”‘ : “”30”‘}.

3. Regra de envio de dados por data/hora. Determina quais horários devem ocorrer o encaminhamento de mensagens de um determinado serviço para o *Middleware*. Os dados recebidos pelo *gateway* fora do horário pré-estabelecido serão descartados. Existem 2 formatos para esta regra comprimir com perda:

   - Dia específico com intervalo. Exemplo: `28-07-2018T15:55Z16:55`;
   - Dia da semana com intervalo. Exemplo: `satT12:00Z13:00`.

3.7.7 Fluxo do *gateway*

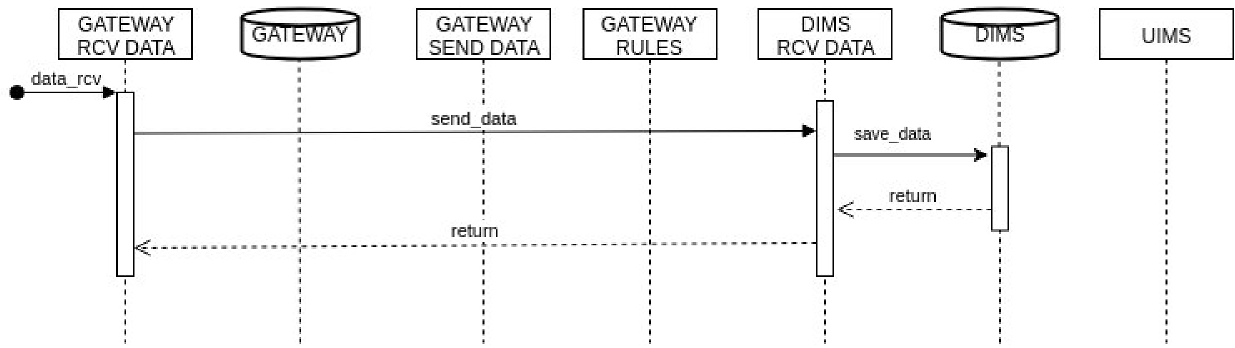

Figura 3.8: Fluxo assíncrono de envio de dados brutos sensíveis de um dispositivo ao *middleware DIMS* através do *gateway*.

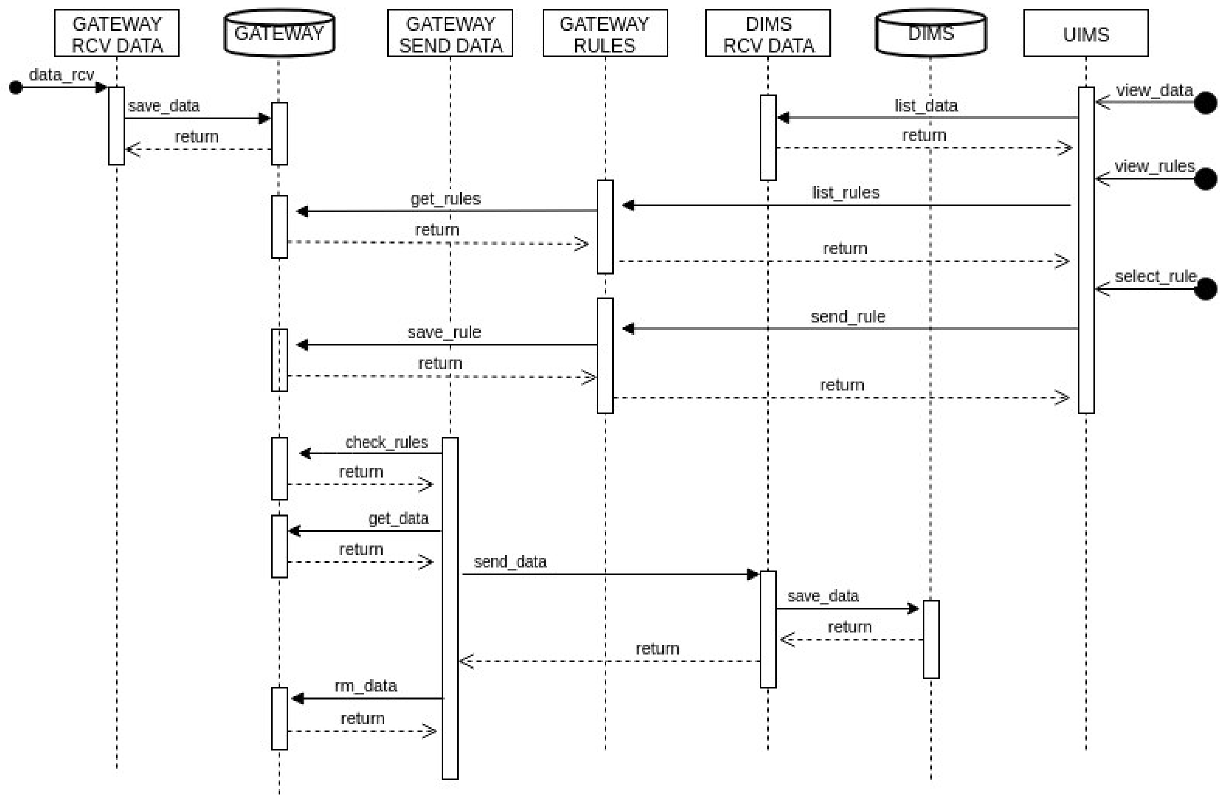

Figura 3.9: Fluxo assíncrono de envio de dados brutos não sensíveis com verificação de regras de um dispositivo ao *no orquestrador* através do *gateway*.

Algorithm 1 Factory

```
1:  procedure INIT(listeners)
2:      start_controllers()
3:      start_db()
4:      update_dispatcher()
5:      start_listeners(listeners)
6:      for all listener ∈ listeners do
7:          queue_up ← Queue
8:          queue_down ← Queue
9:          listener.start(queue_up, queue_down)
10:         StartThreadRcv(listener, queue_up, queue_down)
11:     end for
12: end procedure
```

Algorithm 2 Factory

```
1: procedure STARTTHREADRCV(listener, queue_up, queue_down)
2:     while running do
3:         message ← queue_up.get()
4:         Gate(message, queue_down)
5:     end while
6: end procedure
```

Algorithm 3 Factory

```
1: procedure GATE(message, queue_down)
2:     entity ← get_entity(message)
3:     if entity = list_rules then
4:         response ← get_rules(message)
5:         queue_down.put(response)
6:     else if entity = rules then
7:         response ← get_rules(message)
8:         queue_down.put(response)
9:         if response.code = 200 then
10:            schedule_rule(message)
11:        end if
12:    else if entity = data then
13:        if message.sensitive = True then
14:            send_to_dims(message)
15:        else if check_rule(message) = True then
16:            save_data(message)
17:            check_unit_rule(message)
18:        else if check_dispatcher() = True then
19:            save_data(message)
20:            update_dispatcher()
21:        else
22:            send_to_dims(message)
23:        end if
24:    else if entity = service then
25:        result ← send_to_dims(message)
26:        if result.code = 200 then
27:            save_service(message)
28:        end if
29:    else if entity = client then
30:        send_to_dims(message)
31:    else
32:        log_error_entity_msg()
33:    end if
34: end procedure
```

3.8 *GATEWAY* EM NUVEM

A aplicação *Gateway* em Nuvem tem como responsabilidade validar os dados recebidos pelos *Gateways* e dispositivos inteligentes que se comunicam diretamente com a Nuvem. O *Cloud Gateway* verifica se os dados enviados via chamadas REST obedecem os padrões estabelecidos no auto registro (SILVA et al., 2016b), validando se a semântica pre determinada foi obedecida e o conteúdo do *JSON* contido nas chamadas *REST* atende as regras detalhadas neste capítulo.

3.9 INTERFACE REMOTA DE USUÁRIO

A interface remota do usuário permite a visualização dos dados armazenados na nuvem, detalhando quais dispositivos inteligentes e serviços a ele associado estão enviando dados. Nesse ambiente é possível visualizar tanto dados repassados por *Gateways*, quanto dispositivos inteligentes que se comunicam diretamente com a nuvem. A visualização pode ocorrer em relatórios gráficos ou tabulares, permitindo que o usuário visualize a frequência e os valores de cada serviço, na frequência determinada pelas regras de transmissão.

3.10 BASE DE DADOS REMOTA

A base de dados remota armazena todos os valores recebidos pela nuvem, assim como as regras cadastradas para cada serviço ou dispositivo. O banco de dados utiliza *MongoDB*, amplamente utilizado em solução IoT (KANG et al., 2016) que apresentou bons resultados para apresentar atender as características intrínsecas das aplicações IoT distribuídas. (PAETHONG; SATO; NAMIKI, 2016)
Foi utilizado *MongoDB* com sucesso nos *Gateways* remotos e estes apresentaram desempenho aceitável.

4 RESULTADOS

Os testes de validação foram divididos em duas partes, a primeira valida as técnicas de compactação de dados e aplicação de filtros cadastrados no *Middleware* . Essa fase valida a capacidade do *gateway* obedecer as regras aos filtros definidos no controlador central, determinando como os dados devem ser transmitidos pelo *Gateway* para a Nuvem. A segunda fase do conjunto de testes propõe validar a economia na transmissão de dados, agrupando dados não críticos em filas de transmissão de tamanho fixo e posterior compactação de dados antes de realizar o envio para a nuvem, otimizando assim a capacidade de transmissão do *link* de internet.

A segunda parte do trabalho apresenta os dispositivos e elementos da arquitetura IoT criadas para validar a proposta de uma rede contendo dispositivos, *gateway* e *Middleware* em produção.

A figura 4.1 apresenta a arquitetura utilizada para validação deste trabalho. O *gateway* cliente foi configurado em uma SBC *Raspberry Pi* 3 responsável por receber os dados dos clientes simulados e dispositivos inteligentes reais. O *gateway* possui uma interface local de usuário que permite a visualização dos dados recebidos e armazenados no contexto local, antes de ser enviado para a nuvem. Foram criados clientes IoT em PCs, com capacidade para simular o comportamento de centenas e até mesmo milhares de sensores, permitindo uma maior escala dos testes.

Foi criado um ambiente em nuvem remota de baixa latência, instalando os componentes de *middleware*, de Orquestrador, de base de dados local e de Interface de usuário. O *gateway* realiza comunicação direta com dois componentes em nuvem, o *Middleware* que recebe os dados enviados e realiza a persistência na base, e o Orquestrador, no qual o usuário determina como ele quer receber os dados para cada sensor associado ao *gateway*. Foram elaborados quatro cenários de testes para validar a transmissão e monitorar os indicadores de desempenho de transmissão.

Em cada etapa dos cenários foram avaliados os seguintes itens:

- Perda de mensagens;
- Integridade dos dados;
- Tamanho das mensagens;
- Intervalo de tempo entre os envios das mensagens.

As sub-seções seguintes descrevem os testes realizados em cada cenário.

4.0.1 Validação da Transmissão

Este ambiente foi elaborado para validar a comunicação entre os diferentes sensores simulados, que enviam os dados para o *gateway* local, que por sua vez os recebe, analisa se estes

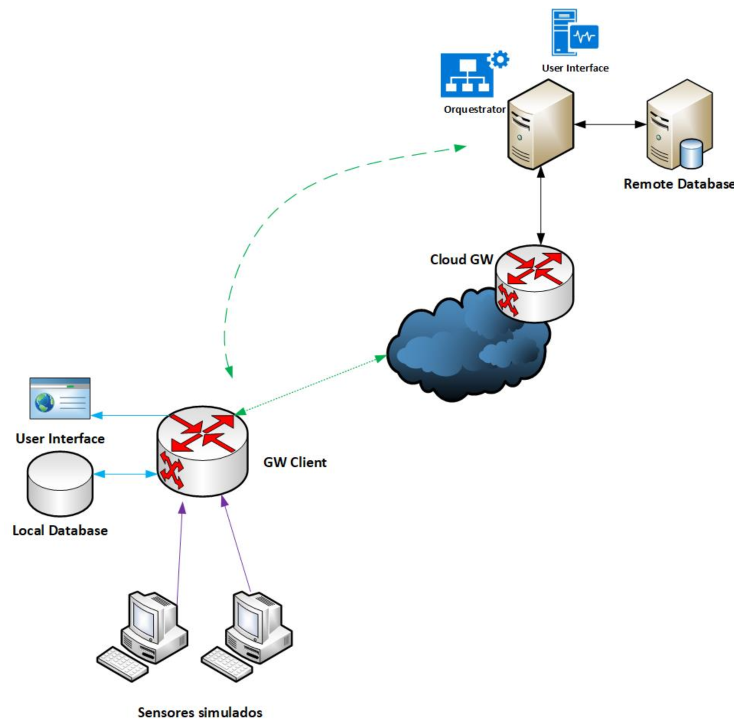

Figura 4.1: Cenário de teste

obedecem a uma semântica pre estabelecida e os encaminha para o *middleware*. Os testes foram divididos em três etapas, são elas:

1. Simulação de um sensor, com um serviço enviando dados a cada dez segundos durante o período de um minuto.
2. Simulação de dez sensores, em computadores distintos, com um serviço enviando dados a cada dez segundos durante o período de cinco minutos.
3. Simulação de cem sensores, distribuídos em computadores distintos, com um serviço enviando dados a cada dez segundos durante o período de cento e oitenta minutos.

Tabela 4.1: Quantidade de mensagens que foram transmitidas nos dispositivos simulados e foram recebidas no *middleware*.

Nro	Qte sensores	Qte Serviços	Frequência(Seg)	Duração(Seg)	Mensagens recebidas (GW)	Mensagens recebidas (MD)
1	1	1	10	60	6	6
2	10	1	10	360	360	360
3	100	1	10	10800	1080	1080

Elaborado pela autor

Os testes foram gerados utilizando computadores conectados na mesma rede do *gateway* IoT com um programa em Python que simula o comportamento do sensor, enviando dados aleató-

rios na frequência descrita na tabela. O objetivo deste teste é validar a capacidade do *gateway* de receber e encaminhar um volume de dados crescente sem apresentar falhas ou travamento. Nesse cenário não houve perdas de mensagens, o *gateway* conseguiu repassar integralmente sem apresentar perdas, mantendo o percentual de CPU abaixo de oito por cento durante todo o teste e memória RAM alterando entre 43 e 69 por cento.

4.0.2 Validação das regras cadastradas no orquestrador

Para validar a comunicação entre os sensores, *gateway* e *Midleware* foram elaborados cenários de testes simulados, utilizando *scripts* em Python para simular uma quantidade massiva de sensores enviado dados para o *gateway*, obedecendo as regras previamente definidas no controlador. O primeiro cenário de testes envolveu os seguintes componentes e testes:

1. Simulação de quatro sensores, com um serviço enviando dados a cada minuto para o *gateway* por dez minutos. Foram criadas regras associadas a cada serviço para que para o *gateway* aplique as formulas de média, valor máximo, mínimo ou terceiro quartil para cada duas amostras.

2. Simulação de quatro sensores, com um serviço enviando dados a cada minuto para o *gateway* por dez minutos. Foram criadas regras associadas a cada serviço para que para o *gateway* aplique as formulas de média, valor máximo, mínimo ou terceiro quartil para cada três amostras.

3. Simulação de quatrocentos sensores, com um serviço enviando dados a cada minuto para o *gateway* por dez minutos. Foram criadas regras associadas a cada serviço para que para o *gateway* aplique as formulas de média, valor máximo, mínimo ou terceiro quartil para cada duas amostras.

Tabela 4.2: Quantidade de mensagens que foram transmitidas nos dispositivos simulados e foram recebidas no *middleware* e *gateway* após a criação de regras

Nro	Qte sensores	Qte Serviços	Frequência	Frequência fórmula (min)	Duração	Mensagens recebidas (GW)	Mensagens recebidas (MD)
1	4	1	1	2	10	10	5
2	4	1	1	3	10	10	3
3	400	1	1	2	10	400	200

Elaborado pela autor

As mensagens transmitidas pelos sensores simulados foram corretamente recebidas pelo *gateway*, e repassadas para o *Midleware* conforme as regras atribuídas no controlador Central sem que houvessem perdas de mensagens. Os dados produzidos pelos serviços foram gerados aleatoriamente e dessa forma não foram feitas analises estatísticas sobre a sua representatividade

4.0.3 Compactação com perdas (filtros)

Além das operações matemáticas apresentadas, o controlador permite que limiares sejam cadastrados para envio imediato de dados para o *Middleware*. O *gateway* compara se determinado serviço apresentou valor maior, menor ou igual ao limiar estabelecido, e se a condição for atendida, a leitura será imediatamente repassada para o *midleware*. Um serviço pode possuir regras de agregação e limiares associados, reduzindo o volume de mensagens trocadas com a nuvem, porem avisando em situações consideradas criticas. O mecanismo de alertas permite que eventos criticos sejam informados imediatamente, sem aguardar o restante dos dados para realizar a agregação, encaminhando imediatamente os dados recebidos quando houver valores acima dos limiares cadastrados.

1. Simulação de um sensor, com um serviço enviando dados a cada minuto para o *gateway* por uma hora;
2. Simulação de um sensor, com um serviço enviando dados a cada minuto para o *gateway* por uma hora. Regra de envio a cada 10 minutos aplicando a média dos valores;
3. Simulação de um sensor, com um serviço enviando dados a cada minuto para o *gateway* por uma hora. Regra de envio a cada 10 minutos aplicando a média dos valores e regra de alerta para valores maiores ou iguais a 30;

Tabela 4.3: Quantidade de mensagens que foram transmitidas nos dispositivos simulados e foram recebidas no *middleware* e *gateway* após a criação de regras

Nro	Qte sensores	Qte Serviços	Frequência	Frequência formula (min)	Duração	Mensagens recebidas (GW)	Mensagens recebidas (MD)
1	1	1	1	2	60	60	30
2	1	1	1	10	60	60	6
3	1	1	1	10	60	60	12

Elaborado pela autor

Conforme pode ser visualizado na tabela, não houve perdas de mensagens nos testes. Foi definido um limiar padrão para eventos superiores a trinta serem enviados imediatamente para o *Middleware* e isto aumentou o volume de mensagens.

4.0.4 Testes realizados

Para validar a economia de envio de pacotes usando compactação sem perdas foi preciso utilizar o *TCPDUMP* (JACOBSON; LERES; MCCANNE, 1989) para monitorar o tráfego dos pacotes de dados da camada de transporte. Para comprimir os dados sem perda de informação do *gateway* para o *middleware* foi utilizado a compressão por número prefixado, método explicado na Seção 3.7.5. Foram criadas 4 etapas para validar estes cenário:

1. Simulação de 100 sensores, com um serviço enviando dados a cada 2 segundos para o *gateway* por 10 minutos sem número prefixado de compressão;
2. Simulação de 100 sensores, com um serviço enviando dados a cada 2 segundos para o *gateway* por 10 minutos. Compressão prefixada de 100 mensagens;
3. Simulação de 100 sensores, com um serviço enviando dados a cada 2 segundos para o *gateway* por 10 minutos. Compressão prefixada de 200 mensagens;
4. Simulação de 100 sensores, com um serviço enviando dados a cada 2 segundos para o *gateway* por 10 minutos. Compressão prefixada de 300 mensagens.

4.0.5 compactação sem perdas

Para cada etapa de simulação do cenário 4 de compactação sem perdas, detalhada na Subseção 4.0.5, foram extraídos arquivos do formato `PCAP` da ferramenta *TCPDUMP* para análise quantitativa do tamanho total de cada pacote trocado entre *gateway* e *middleware*.

Tabela 4.4: Resultados do cenário 4 de compactação sem perdas de dados.

	Total de pacotes	Tamanho total (bytes)
Etapa 1	363612	38305889
Etapa 2	15710	7977357
Etapa 3	12628	6729054
Etapa 4	11288	6177641

Elaborado pela autor

Tabela 4.5: Proporção relativa das etapas 2, 3 e 4 da quantidade e tamanho total de pacotes em comparação com envio sem compactação.

	Tamanho da pilha	Quantidade de pacotes (%)	Tamanho total (%)
Etapa 2	100	95.68%	79,17%
Etapa 3	200	96.53%	82,43%
Etapa 4	300	96.90%	83,87%

Elaborado pela autor

Como podemos ver pela Tabela 4.4, Tabela 4.5 e o gráfico da Figura 4.2, quando os dados são armazenados em blocos de 100 e enviados comprimidos ao *middleware* sem perdas, etapa 2, a quantidade de pacotes trocados e o tamanho total dos pacotes tiveram uma diminuição significativa em comparação à etapa 1 que não houve compressão, 95.68% e 79,17% respectivamente.

4.0.6 Desenvolvimento do Ambiente IoT

Para produzir os resultados esperados, além de validar a solução proposta, a equipe do Laboratório UIoT da UnB desenvolveu um ambiente IoT composto por sensores e atuadores, dispositivos

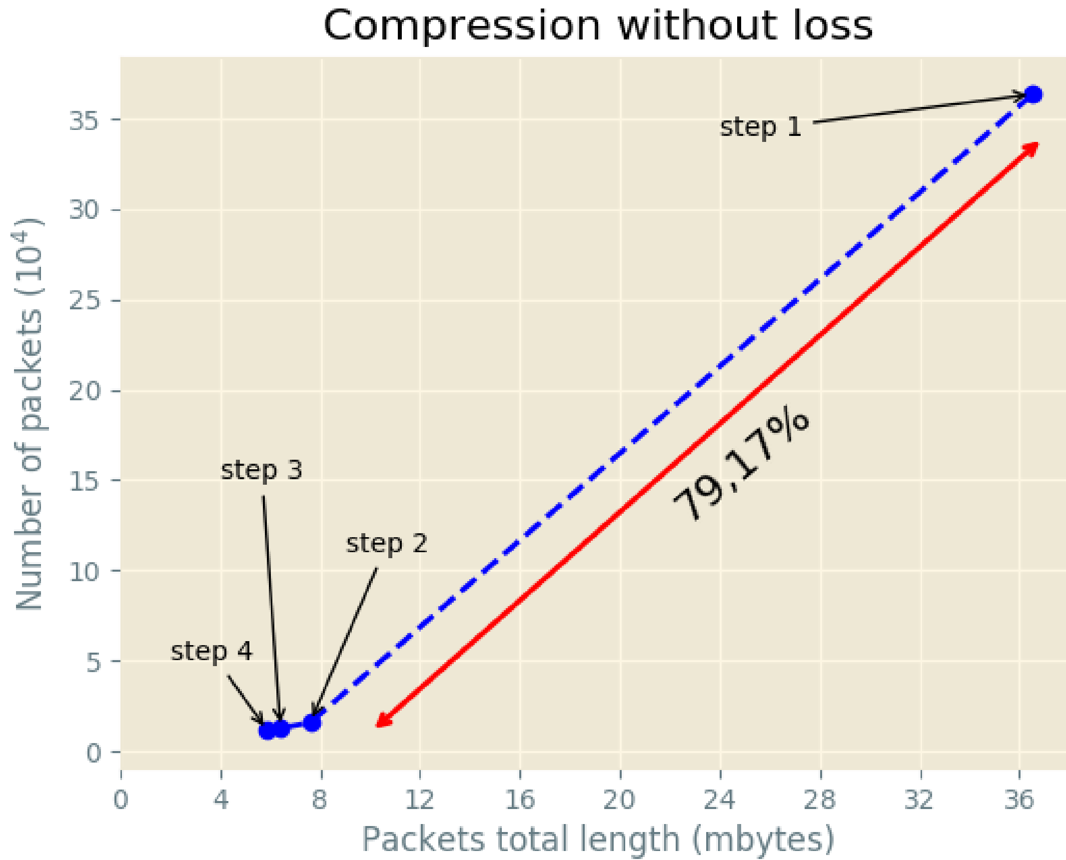

Figura 4.2: Gráfico dos resultados do cenário 4, compactação sem perdas, apresentado na Seção 4.0.5.

inteligentes, *Gateways*, *Middleware* e aplicações que permitem a visualização e controle dos dispositivos inteligentes. As sessões seguintes detalham algumas das atividades desenvolvidas no Laboratório UIoT ao longo do desenvolvimento da dissertação e com contribuições mútuas entre os trabalhos.

4.0.7 Gateway ZigBee

O desenvolvimento de um módulo de transmissão e recepção ZigBee permitiu a interoperabilidade do *Middleware UIoT* com dispositivos em redes de sensores sem fio ZigBee. Houve desdobramentos deste trabalho, com duas pesquisas sendo conduzidas e finalizadas. Diversos dispositivos com transmissor *ZigBee* foram criados para permitir a prova de conceito.

4.0.8 Gateway IP IoT

Foram desenvolvidos dispositivos com arduino, responsáveis apenas por fazer a leitura dos sensores e controle do atuadores, transferindo atividades complexas como autenticação, tradução semântica com o *gateway*. A comunicação com o *middleware* era feita a partir de chamadas

REST Full via HTTPs.

O *hardware* exibido na figura 4.4 realiza a irrigação automática do solo quando este está seco. Ele também monitora a quantidade de água no reservatório. Informações sobre quantidade de água no reservatório, status da bomba e percentual de umidade do solo são enviadas a cada minuto para o *gateway*

Outro dispositivo inteligente desenvolvido durante a dissertação foi uma régua elétrica inteligente, que o usuário pode ligar e desligar remotamente, utilizando *Browser* em rede local, a tela do *Middleware* pela Internet, ou até mesmo via *Telegram*. As informações de consumo são medidas em tempo real e disponibilizadas via *Browser*

4.0.9 Interface de interação com o usuário

O cadastro e a visualização de regras ocorrem no ambiente em nuvem que o *Gateway* UIoT está associado. Nele podemos criar e visualizar regras e filtros criados para determinado serviço.

Na interface de interação com o usuário na nuvem podemos também, visualizar os dados enviados pelos diferentes *gateways*

A tela permite visualizar quais sensores estão associados a determinado *gateway*

Cada dispositivo inteligente possui serviços associados a ele, conforme tela seguinte:

Os dados dos serviços podem ser visualizados em formato de tabela ou gráfico

4.0.10 Interface de comando e controle

A solução criada nesta dissertação também permite o controle de dispositivos em ambiente de rede local, via aplicação instalada no Gateway. O portal WEB (Dash) permite a visualização de dados dos sensores, além do controle de atuadores remotamente. O *Dash* permite que as lampadas sejam ligadas e desligadas via WEB, além da visualização dos dados enviados pelos sensores

4.0.11 Controle individual dos dispositivos

As bibliotecas para Arduino e ESP8266 criadas permitem o controle individual dos dispositivos via WEB. Por meio de um portal simples é possível fazer o controle dos atuadores além de visualizar os dados dos sensores.

Os dispositivos criados no projeto permitem a mobilidade de redes sem fio através da biblioteca WIFI Manager. Caso o dispositivo seja retirado da rede sem fio em que está conectado, ele entrar em modo de ponto de acesso, na qual o usuário pode se conectar, informar as credenciais da sua rede e após isto visualizar os dados ou controlar os dispositivos na sua rede sem fio.

Os dispositivos podem obter atualização de software remotamente, utilizando a biblioteca

OTA, permitindo a correção ou inclusão de novas funcionalidades sem a necessidade de desmontar o hardware.

Para facilitar a construção de novos dispositivos, desenvolvemos uma biblioteca para comunicação dos dispositivos projetados em ESP e Arduino com Gateway UIoT. Com apenas 5 linhas de código é possível configurar o dispositivo para enviar dados para o Gateway. Esta biblioteca foi integrada aos módulos WIFI Manager e OTA, trazendo todas as funcionalidades citadas para um único módulo.

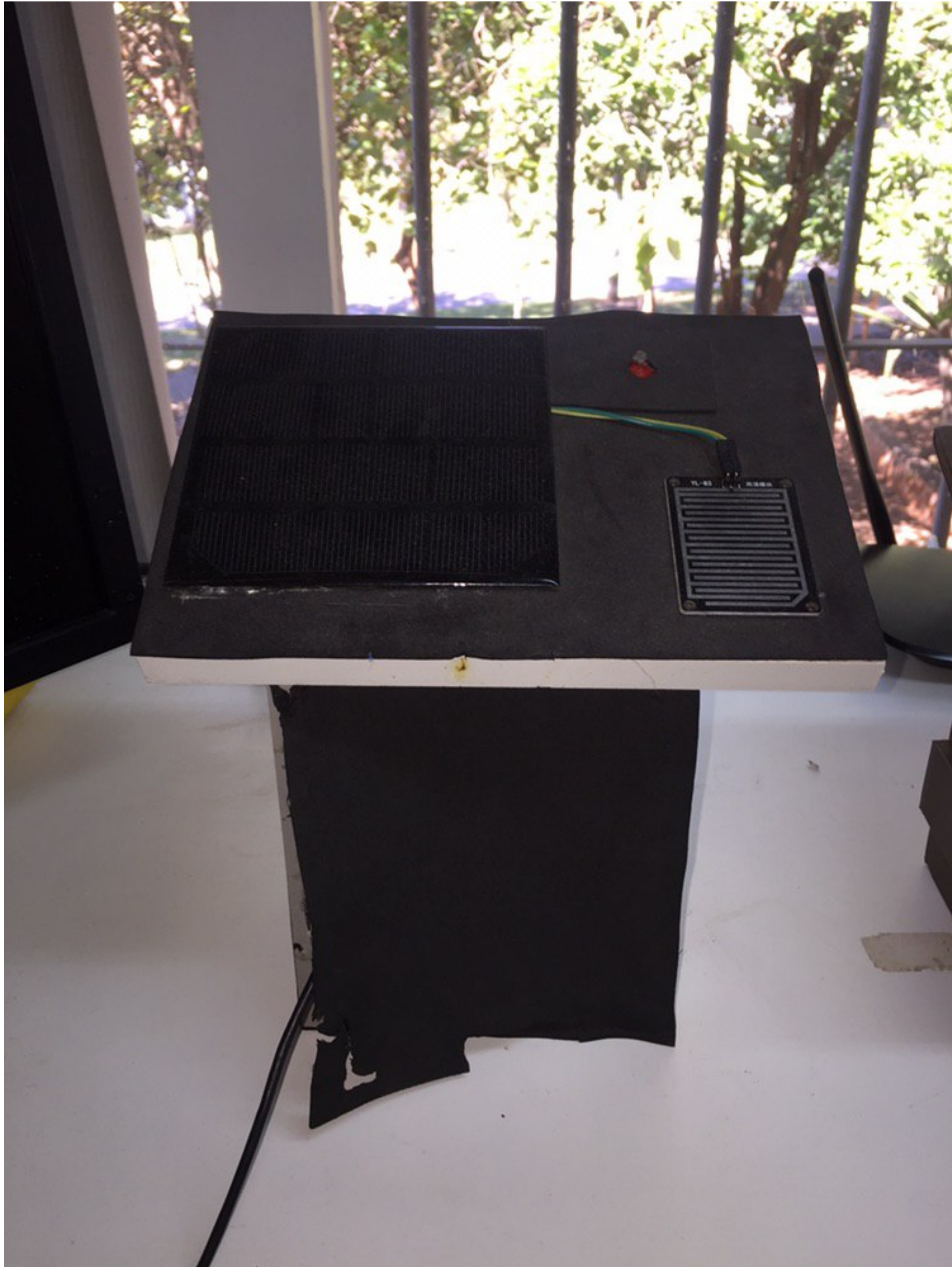

Figura 4.3: Estação Meteorológica ZigBee.

Figura 4.4: Sistema de irrigação controlado de acordo com a umidade do solo.

Figura 4.5: Régua inteligente controlada pelo *Middleware*.

UIoT Smart Outlet

Smart Outlet ligada

Desligar

Corrente medida: 30.76 mA

Potência consumida: 6.77 W

Tensão da Rede Elétrica = 220V

◉ 220 V ○ 110 V

172.16.9.69

Atualizar IP do Gateway UIoT

Created by UIoT

Figura 4.6: Visão via *Browser* para o controle da Régua Inteligente.

Search

name ↑	chipset	mac	serviceNumber	type	Applied Time	Unity
regra1	AMD 790FX	FF:FF:FF:FF:FF:FF	4	expurgo	25	min

Rows per page: 10 1-1 of 1

Figura 4.7: Tela no *Gateway* local para visualização das regras definidas no Orquestrador.

Service Cloud Average

Search

Dispositivo ↑	Nome do Serviço	Frequência média de atualização	Horário das Coletas
Raspberry PI	Get temp	52.694971257990055	Tue - 11 Jun 2019 20:27:27
Raspberry PI	Get temp	52.694971257990055	Tue - 11 Jun 2019 20:27:27
Raspberry PI	LarissaePrado	129.85257800420126	Tue - 11 Jun 2019 20:33:57
Raspberry PI	Get temp	52.694971257990055	Tue - 11 Jun 2019 20:27:27
Raspberry PI	Get temp	52.694971257990055	Tue - 11 Jun 2019 20:27:27

Rows per page: 5 1-5 of 7

Figura 4.8: Visualização dos dados na nuvem.

Clients

Search

Name ↑	Interface	MAC	
Raspberry PI	Ethernet	FF:FF:FF:FF:FF:FF	SERVICES

Figura 4.9: Dispositivos associados a um determinado *gateway*.

Services

Search

Nome ↑	Number	Parameter	
Get hum	5	humidity	DATA
Get temp	1	temperature	DATA
Get temp	3	temperature	DATA
Get temp	3	temperature	DATA

Rows per page: 5 1-4 of 4

Figura 4.10: Serviços associados a determinado dispositivo.

TABLE GRAPH

Search

Time ↑	Sensitive	Values
Mon Dec 03 2018 13:14:05	1	["20.2"]
Mon Dec 03 2018 13:14:05	1	["20.2", "30.0"]

Rows per page: 10 1-2 of 2

Figura 4.11: Visualização dos dados em formato tabular.

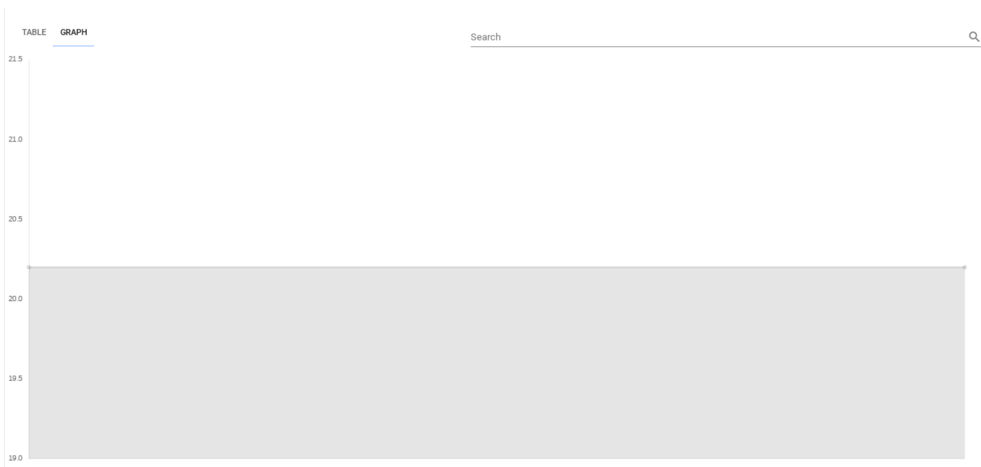

Figura 4.12: Visualização dos dados em gráficos.

5 CONCLUSÃO

A motivação para o desenvolvimento do trabalho de mestrado que deu origem à presente dissertação foi contribuir para um racionalização do trasporte de dados em redes e sistemas de IoT, utilizando técnicas de aceleração do tráfego em redes WAN e regras de organização do processamento em nevoeiro (*fog*) para filtragem e sumarização de dados a serem enviados para módulos de processamento em nuvem (*cloud*) usados nessas redes e sistemas de IoT.

Com base nessa ideia geral, esta dissertação descreve os passos para criar uma arquitetura fim a fim de IoT em *fog*, composta com módulos próprios dos dispositivos inteligentes de IoT, bem como *gateways* e *middleware* de IoT, sob coordenação de um módulo orquestrador. O requisito funcional para Tal arquitetura foi definido em termos de capacidade de realizar o processamento, armazenamento e tomada de decisão de forma distribuída para dar suporte a instâncias de redes IoT.

Feita a proposição da arquitetura em *fog* para IoT, para efeito de validação da proposta, foram desenvolvidos ao longo deste trabalho módulos de software utilizando técnicas de aceleração *WAN*, ou seja, técnicas de compressão na comunicação entre os dispositivos inteligentes, *gateways*, *middleware* de IoT e módulos em nuvem. Foram também definidos os formatos e os processos de cadastro, propagação e execução de regras de processamento em nevoeiro, estas voltadas a economizar os dados a serem intercambiados com módulos de computação em nuvem, usados em um ambiente fim-a-fim de IoT.

Como resultado, além de mostrar o efetivo funcionamento geral da arquitetura proposta, a validação mostra também como o *gateway fog* proposto no trabalho permite o armazenamento e processamento de parte dos dados próximo aos sensores/atuadores de IoT, o que, conjuntamente com a aceleração *WAN*, leva a reduzir a necessidade de transmissão para a nuvem. Testes de validação do conjunto de protótipos de software permitiram mostrar uma redução superior 70% no encaminhamento de dados, com relação a uma solução pura de nuvem, ou seja, sem computação em *fog*.

Conforme discussão apresentada na seção de resultados, a solução proposta criou um ambiente de processamento e armazenamento distribuído composto por dispositivos inteligentes, *Gateways*, *Middleware*, visualizador e Controlador Central.

Concomitantemente ao desenvolvimento da ideia central da dissertação, em atividades do Laboratório UIoT da UnB, como benefícios mútuos entre tais atividades e este tarbalho de mestrado, dispositivos inteligentes foram criados com *hardware* de baixo custo, aberto, e os códigos para comunicação com o *gateway*, assim como bibliotecas que permitem a atualização remota e portabilidade que estão disponíveis no anexo deste documento. Assim, para efeito da validação do trabalho de mestrado, houve aumento na quantidade e variabilidade dos dispositivos, assim como as formas de comunicação entre eles e o *gateway*, indicando a flexibilidade e a escalabilidade da

proposta.

Especificamente, ficou demonstrado que o *gateway fog* proposto na dissertação foi capaz de receber dados de redes de sensores sem fio, repassando para nuvem de acordo com as regras atribuídas pelo orquestrador, sendo assim uma extensão do ambiente em *cloud*, capaz de armazenar e processar os dados antes do envio. Assim, contribuímos com a implementação de *middleware* IoT com suporte à compactação e aplicação de filtros, de modo a que os dados enviados são compactos mas suficientes para que módulos em nuvem sejam capazes de responder aos dispositivos e *gateways*, tal como configurados Laboratório UIoT, aplicando filtros e formulas aos dados antes de serem enviados.

Finalmente, a prototipação usada na validação do trabalho resultou em visualizadores de dados na nuvem e em rede local (*Dash*), capazes de controlar remotamente atuadores e visualizar os dados de sensores.

5.1 TRABALHOS FUTUROS

A partir do trabalho realizado, é possível evoluir no sentido de aplicar regras de maior alcance e envolvendo processamento avançado da informação tradada em aplicações de IoT que envolvam multimídia e relacionamentos mais complexos entre os dados.

Por exemplo, como parte de projetos de reconhecimento facial integradas a solução descrita nesta dissertação, criam-se dispositivos de IoT inteligentes com Raspberry e câmeras, que captam imagens do ambiente e reconhecem os rostos das pessoas que entram no Laboratório UIoT. Ao identificar um rosto, é gerada uma fotografia e enviada para um aplicativo de rede social usado como ambiente de monitoração comunitária do laboratório, em particular pela rede Telegram.

As trocas de informação nesse sistema que está em desenvolvimento incluem o cadastro de pessoas no ambiente em nuvem. Ao cadastrar as imagens, há possibilidade treinar o algorítimo em nuvem e associar este a um determinado dispositivo inteligente. O dispositivo inteligente realiza o *download* do algorítimo treinado, reconhecendo e nomeando as fotos de pessoas cadastradas. Tal informação tem uma estrutura mais complexa que poderia ser explorada na definição de regras de processamento em *fog* mais aptas aos requisitos do sistema.

Outras possibilidades se configuram quanto à aceleração de tráfego e regras dinâmicas de processamento com mesclagem *cloud-edge-fog*, ou seja de processamento distribuído, para os casos por exemplo em que no sistema de reconhecimento proposto se possa associar fotos públicas, de pessoas desaparecidas, permitindo a geração de alertas na incidência de eventos.

É possível ainda considerar outras dimensões informacionais como a localização e o movimento, cuja dinâmica requer uma maior flexibilidade e adaptabilidade da IoT, como no caso de outro projeto do Laboratório UIoT que visa criar um carro autônomo como uma segunda fonte de dados, por circular pela sala capturando imagens e enviando para a central de reconhecimento.

REFERÊNCIAS BIBLIOGRÁFICAS

ADELANTADO, F.; VILAJOSANA, X.; TUSET-PEIRO, P.; MARTINEZ, B.; MELIA-SEGUI, J.; WATTEYNE, T. Understanding the Limits of LoRaWAN. *IEEE Communications Magazine*, v. 55, n. 9, p. 34–40, Setembro 2017. ISSN 0163-6804.

ARAÚJO, I. P.; DE SOUSA JÚNIOR, R. T. Proposta de Abordagem Quadridimensional e Integrada para Internet as Coisas Embasada na Revisão dos Conceitos de Smart Object. In: *Atas das Conferências IADIS Ibero-Americanas WWW/Internet 2017 e Computação Aplicada 2017*. Vilamoura, Algarve, Portugal: IADIS Press, 2017. p. 111–118. ISBN 978-989-8533-70-8.

ARDUINO. 2019. Disponível em: <www.arduino.cc>.

ASHTON, K. *That 'Internet of Things' Thing*. 2009. Acesso em 05 maio 2019. Disponível em: <https://www.rfidjournal.com/articles/view?4986>.

ATZORI, L.; IERA, A.; MORABITO, G. The Internet of Things: A Survey. *Computer Networks*, Elsevier BV, v. 54, n. 15, p. 2787–2805, Outubro 2010. ISSN 1389-1286.

AUGUSTIN, A.; YI, J.; CLAUSEN, T.; TOWNSLEY, W. M. A Study of LoRa: Long Range & Low Power Networks for the Internet of Things. *Sensors*, v. 16, n. 9, 2016. ISSN 1424-8220. Disponível em: <http://www.mdpi.com/1424-8220/16/9/1466>.

BALL, S. *Embedded Microprocessor Systems: Real World Design*. 3. ed. Burlington, MA, EUA: Elsevier, 2002. ISBN 978-0750675345.

BEAGLEBOARD. 2019. Disponível em: <https://beagleboard.org/bone>.

BONOMI, F.; MILITO, R.; ZHU, J.; ADDEPALLI, S. Fog Computing and Its Role in the Internet of Things. In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. Helsinki, Finlândia: ACM, 2012. (MCC '12), p. 13–16. ISBN 978-1-4503-1519-7.

BORGIA, E. The Internet of Things vision: Key features, applications and open issues. *Computer Communications*, v. 54, p. 1–31, Dezembro 2014. ISSN 01403664.

BRAYNER DOROTÉA KARINE D. PITOMBEIRA, R. W. C. B. A. *Uma Arquitetura Eficiente para Armazenamento, Compressão e Acesso a Dados em Dispositivos Móveis com Recursos Computacionais Limitados*. Dissertação (Dissertação) — UNIFOR, Fortaleza, 2016.

CAI, H.; XU, B.; JIANG, L.; VASILAKOS, A. V. IoT-Based Big Data Storage Systems in Cloud Computing: Perspectives and Challenges. *IEEE Internet of Things Journal*, IEEE, v. 4, n. 1, p. 75–87, 2017.

CALDAS FILHO, F. L. d.; MARTINS, L. M. C. e.; ARAÚJO, I. P.; MENDONÇA, F. L. L. d.; COSTA, J. P. C. L. da; DE SOUSA JÚNIOR, R. T. Design and evaluation of a semantic gateway prototype for IoT networks. In: *Companion Proceedings of the 10th International Conference on Utility and Cloud Computing*. Austin, TX, EUA: ACM, 2017. (UCC '17 Companion), p. 195–201. ISBN 978-1-4503-5195-9.

CALDAS FILHO, F. L. d.; MARTINS, L. M. C. e.; ARAÚJO, I. P.; MENDONÇA, F. L. L. d.; COSTA, J. P. C. L. d.; DE SOUSA JÚNIOR, R. T. Gerenciamento de Serviços IoT com Gateway Semântico. In: *Atas das Conferências IADIS Ibero-Americanas WWW/Internet 2017 e Computação Aplicada 2017*. Vilamoura, Algarve, Portugal: IADIS Press, 2017. p. 199–206. ISBN 978-989-8533-70-8.

CHAOUCHI, H.; BOURGEAU, T.; KIRCI, P. Internet of Things: From Real to Virtual World. In: CHILAMKURTI, N.; ZEADALLY, S.; CHAOUCHI, H. (Ed.). *Next-Generation Wireless Technologies: 4G and Beyond*. Londres, Reino Unido: Springer, 2013. p. 161–188. ISBN 978-1-4471-5164-7.

CHEN, H.; JIA, X.; LI, H. A brief introduction to IoT gateway. In: *IET International Conference on Communication Technology and Application (ICCTA 2011)*. Pequim, China: Institution of Engineering and Technology (IET), 2011. p. 610–613.

CHEONG, P. S.; BERGS, J.; HAWINKEL, C.; FAMAEY, J. Comparison of lorawan classes and their power consumption. In: . [S.l.: s.n.], 2017.

CHIANG, M.; ZHANG, T. Fog and IoT: An Overview of Research Opportunities. *IEEE Internet of Things Journal*, v. 3, n. 6, p. 854–864, Dezembro 2016. ISSN 2327-4662.

CISCO Wan Aceleration Guide. 2019. Disponível em: <https://www.cisco.com/c/en/us/td/docs/app_ntwk_services/waas/waas/v501/quick/guide/waasqcg.html>.

CLOUD. 2017. Disponível em: <https://thinkitsolutions.com/what-i-think-i-need/increased-productivity-and-efficiency/cloud-computing-solutions>.

COCHRANE, N. *US smart grid to generate 1000 petabytes of data a year*. 2010. Disponível em: <https://www.itnews.com.au/news/us-smart-grid-to-generate-1000-petabytes-of-data-a-year-170290>.

DASTJERDI, A. V.; BUYYA, R. Fog computing: Helping the Internet of Things realize its potential. *Computer*, IEEE, v. 49, n. 8, p. 112–116, Agosto 2016.

DESAI, P.; SHETH, A.; ANANTHARAM, P. Semantic Gateway as a Service Architecture for IoT Interoperability. In: IEEE. *Mobile Services (MS), 2015 IEEE International Conference on*. New York, NY, EUA, 2015. p. 313–319.

DUTRA, B. V.; ALENCASTRO, J. F. de; FILHO, F. L. de C.; MARTINS, L. M. C. e; DE SOUSA JÚNIOR, R. T.; ALBUQUERQUE, R. de O. HIDS by signature for embedded devices in IoT networks. In: UNIVERSIDAD DE EXTREMADURA. *Actas de las V Jornadas Nacionales de Investigación en Ciberseguridad (JNIC 2019)*. Cáceres, Spain, 2019. p. 53–61. ISBN 978-84-09-12121-2.

ESPRESSIF. *ESP8266 Technical Reference*. Xangai, China, 2019. Disponível em: <https://www.espressif.com/sites/default/files/documentation/esp8266-technical_reference_en.pdf>.

EU GDPR. *EU GDPR Portal - Home Page*. 2019. Disponível em: <https://eugdpr.org/>.

FERREIRA, H. G. C. *Arquitetura de Middleware para Internet das Coisas*. Dissertação (Dissertação) — Universidade de Brasília, Brasília, 2014.

FERREIRA, H. G. C.; CANEDO, E. D.; DE SOUSA JÚNIOR, R. T. IoT Architecture to Enable Intercommunication Through REST API and UPnP Using IP, ZigBee and Arduino. In: *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. Lyon, França: [s.n.], 2013. p. 53–60.

FERREIRA, H. G. C.; DE SOUSA JÚNIOR, R. T.; DEUS, F. E. G. d.; CANEDO, E. D. Proposal of a Secure, Deployable and Transparent Middleware for Internet of Things. In: *2014 9th Iberian Conference on Information Systems and Technologies (CISTI)*. Barcelona, Espanha: [s.n.], 2014. p. 1–4.

FORTINET Wan Aceleration. 2019. Disponível em: <https://help.fortinet.com/fos60hlp/60/Content/FortiOS/fortiOS-HTML5-v2/InsideFOS/WANOpt.htm>.

GARTNER. *Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015*. [S.l.], 2015. Disponível em: <https://www.gartner.com/en/newsroom/press-releases/2015-11-10-gartner-says-6-billion-connected-things-will-be-in-use-in-2016-up-30-percent-from-2015>. Acesso em: 2019-02-22.

Gartner, Inc. *How Edge Computing Completes Cloud*. [S.l.], 2016. Disponível em: <https://www.gartner.com/en/webinars/3887383/how-edge-computing-completes-cloud>. Acesso em: 2019-02-22.

GLUHAK, A.; KRCO, S.; NATI, M.; PFISTERER, D.; MITTON, N.; RAZAFINDRALAMBO, T. A survey on facilities for experimental internet of things research. *IEEE Communications Magazine*, v. 49, n. 11, p. 58–67, Novembro 2011. ISSN 0163-6804.

GOYAL, S. Public vs private vs hybrid vs community - cloud computing: A critical review. *International Journal of Computer Network and Information Security*, MECS Publisher, v. 6, n. 3, p. 20–29, Fevereiro 2014.

GUBBI, J.; BUYYA, R.; MARUSIC, S.; PALANISWAMI, M. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, v. 29, n. 7, p. 1645–1660, 2013. ISSN 0167-739X.

GUO, B.; ZHANG, D.; WANG, Z.; YU, Z.; ZHOU, X. Opportunistic IoT: Exploring the harmonious interaction between human and the internet of things. *Journal of Network and Computer Applications*, v. 36, n. 6, p. 1531 – 1539, 2013. ISSN 1084-8045. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1084804513000052>.

GW LoRA. 2019. Disponível em: <https://www.thethingsnetwork.org/docs/gateways/start/list.html>.

HOME Automation with Arduinoand OpenHAB. 2014. Disponível em: <https://electronichamsters.files.wordpress.com/2014/08/system_design.pdf>.

IETF. *Configuration Guidelines for DiffServ Service Classes*. 2006. Disponível em: <https://tools.ietf.org/html/rfc4594>.

ISHAKIAN, V.; SWEHA, R.; LONDONO, J.; BESTAVROS, A. Colocation as a Service: Strategic and Operational Services for Cloud Colocation. In: *2010 Ninth IEEE International Symposium on Network Computing and Applications*. [S.l.: s.n.], 2010. p. 76–83.

JACOBSON, V.; LERES, C.; MCCANNE, S. The tcpdump manual page. *Lawrence Berkeley Laboratory, Berkeley, CA*, v. 143, 1989.

JOSHI, P. *Introduction to ZigBee: Architecture, Networks and AT Commands*. 2017. Disponível em: <https://circuitdigest.com/article/zigbee-introduction-architecture-at-commands>.

JR, T. G.; NO, J. C. C. et al. *Application acceleration and wan optimization fundamentals*. [S.l.]: Cisco Press, 2012.

KALIM, A. *Clouds on the Academic Horizon*. 2013.

KANG, B.; KIM, D.; CHOO, H. Internet of everything: A large-scale autonomic iot gateway. *IEEE Transactions on Multi-Scale Computing Systems*, v. 3, p. 206–214, Julho 2017.

KANG, Y.; PARK, I.; RHEE, J.; LEE, Y. MongoDB-Based Repository Design for IoT-Generated RFID/Sensor Big Data. *IEEE Sensors Journal*, v. 16, n. 2, p. 485–497, Janeiro 2016. ISSN 1530-437X.

KELLY, R. *Internet of Things Data To Top 1.6 Zettabytes by 2020*. 2015. Disponível em: <https://campustechnology.com/articles/2015/04/15/internet-of-things-data-to-top-1-6-zettabytes-by-2020.aspx>.

KUMAR, R.; RAJASEKARAN, M. P. An iot based patient monitoring system using raspberry pi. In: *2016 International Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE'16)*. [S.l.: s.n.], 2016. p. 1–4.

LATRE, S.; LEROUX, P.; COENEN, T.; BRAEM, B.; BALLON, P.; DEMEESTER, P. City of things: An integrated and multi-technology testbed for iot smart city experiments. In: *2016 IEEE International Smart Cities Conference (ISC2)*. [S.l.: s.n.], 2016. p. 1–8.

LIGHT, R. A. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, The Open Journal, v. 2, n. 13, 2017.

MARTINS, L. M. C. e.; CALDAS FILHO, F. L. d.; DE SOUSA JÚNIOR, R. T.; GIOZZA, W. F.; COSTA, J. P. C. da. Increasing the dependability of IoT middleware with cloud computing and microservices. In: *Companion Proceedings of the 10th International Conference on Utility and Cloud Computing*. Austin, TX, EUA: ACM, 2017. (UCC '17 Companion), p. 203–208. ISBN 978-1-4503-5195-9.

MARTINS, L. M. C. e.; CALDAS FILHO, F. L. d.; DE SOUSA JÚNIOR, R. T.; GIOZZA, W. F.; COSTA, J. P. C. L. d. Proposta de Adoção de Microsserviços em IoT. In: *Atas das Conferências IADIS Ibero-Americanas WWW/Internet 2017 e Computação Aplicada 2017*. Vilamoura, Algarve, Portugal: IADIS Press, 2017. p. 63–70. ISBN 978-989-8533-70-8.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011.

MITCHELL, G. The raspberry pi single-board computer will revolutionise computer science teaching [for against]. *Engineering Technology*, v. 7, n. 3, p. 26–26, Abril 2012. ISSN 17509637.

OPEN Nebula. 2019. Disponível em: <https://opennebula.org/>.

OSMAN, A.; ABDULATIF, H.; ELGELANY, A.; ALIAS, A. D. N. Private cloud: Effective strategy in developed countries case study: Sudanese universities. *IOSR Journal of Computer Engineering (IOSR-JCE)*, v. 19, p. 73–79, Março 2017.

PAETHONG, P.; SATO, M.; NAMIKI, M. Low-power distributed NoSQL database for IoT middleware. In: *2016 Fifth ICT International Student Project Conference (ICT-ISPC)*. [S.l.: s.n.], 2016. p. 158–161.

PERERA, C.; ZASLAVSKY, A.; CHRISTEN, P.; GEORGAKOPOULOS, D. Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys Tutorials*, v. 16, n. 1, p. 414–454, 2014. ISSN 1553-877X.

PHAM, C. Building Low-Cost Gateways and Devices for Open LoRa IoT Test-Beds. In: *Testbeds and Research Infrastructures for the Development of Networks and Communities*. Cham: Springer International Publishing, 2017. p. 70–80. ISBN 978-3-319-49580-4.

RAHMANI, A.-M.; THANIGAIVELAN, N. K.; GIA, T. N.; GRANADOS, J.; NEGASH, B.; LILJEBERG, P.; TENHUNEN, H. Smart e-health gateway: Bringing intelligence to internet-of-things based ubiquitous healthcare systems. In: IEEE. *Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE*. [S.l.], 2015. p. 826–834.

RASPBERRY Pi. 2019. Disponível em: <https://www.raspberrypi.org/>.

RIBEIRO, C. F. C.; CALDAS FILHO, F. L. de; MARTINS, L. M. C. e; ABBAS, C. J. B.; DE SOUSA JÚNIOR, R. T. Protocolos de Redundância de Gateway Aplicados em Redes IoT. In: SBRT. *XXXVI Simpósio Brasileiro de Telecomunicações e Processamento de Sinais - SBrT2018*. Campina Grande, PB, 2018.

SANCHEZ, L.; MUÑOZ, L.; GALACHE, J. A.; SOTRES, P.; SANTANA, J. R.; GUTIERREZ, V.; RAMDHANY, R.; GLUHAK, A.; KRCO, S.; THEODORIDIS, E.; PFISTERER, D. SmartSantander: IoT experimentation over a smart city testbed. *Computer Networks*, Elsevier BV, v. 61, p. 217–238, Março 2014. Disponível em: <https://doi.org/10.1016/j.bjp.2013.12.020>.

SARKAR, S.; CHATTERJEE, S.; MISRA, S. Assessment of the Suitability of Fog Computing in the Context of Internet of Things. *IEEE Transactions on Cloud Computing*, v. 6, n. 1, p. 46–59, Janeiro 2018. ISSN 2168-7161.

SATURNO, M.; PERTEL, V. M.; DESCHAMPS, F. Proposal of an automation solutions architecture for Industry 4.0. In: ICPR POZNAN. *Proceedings of the 24th International Conference on Production Research*. Poznań, Polônia, 2017.

SBC OrangePi. 2018. Disponível em: <http://www.orangepi.org/>.

SCHENFELD, M. C. *Fog e edge computing : uma arquitetura híbrida em um ambiente de internet das coisas*. Dissertação (Dissertação) — Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, RS, Março 2017.

SHETE, R.; AGRAWAL, S. IoT based urban climate monitoring using Raspberry Pi. In: *2016 International Conference on Communication and Signal Processing (ICCSP)*. [S.l.: s.n.], 2016. p. 2008–2012.

SHI, W.; CAO, J.; ZHANG, Q.; LI, Y.; XU, L. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, v. 3, n. 5, p. 637–646, Outubro 2016. ISSN 2327-4662.

SHI, W.; DUSTDAR, S. The promise of edge computing. *Computer*, IEEE, v. 49, n. 5, p. 78–81, 2016.

SHROUF, F.; ORDIERES, J.; MIRAGLIOTTA, G. Smart factories in industry 4.0: A review of the concept and of energy management approached in production based on the internet of things paradigm. In: IEEE. *Industrial Engineering and Engineering Management (IEEM), 2014 IEEE International Conference on*. [S.l.], 2014. p. 697–701.

SILVA, C. C. d. M.; CALDAS, F. d.; MACHADO, F. D.; MENDONÇA, F. L.; DE SOUSA JÚNIOR, R. T. Proposta de auto-registro de serviços pelos dispositivos em ambientes de iot. In: *34º Simpósio Brasileiro de Telecomunicações e Processamento de Sinais*. Santarém, PA: SBrT, 2016.

SILVA, C. C. d. M.; FERREIRA, H. G. C.; DE SOUSA JÚNIOR, R. T.; BUIATI, F.; VILLALBA, L. J. G. Design and Evaluation of a Services Interface for the Internet of Things. *Wireless Personal Communications*, Janeiro 2016. ISSN 0929-6212, 1572-834X.

SPERLING, T. L. von; CALDAS FILHO, F. L. de; DE SOUSA JÚNIOR, R. T.; MARTINS, L. M. C. e; ROCHA, R. L. Tracking intruders in IoT networks by means of DNS traffic analysis. In: IEEE. *2017 Workshop on Communication Networks and Power Systems (WCNPS)*. Brasília, DF, 2017. p. 1–4.

SPERLING, T. L. von; FRANÇA, B. de A.; CALDAS FILHO, F. L. de; MARTINS, L. M. C. e; ALBUQUERQUE, R. de O.; DE SOUSA JÚNIOR, R. T. Evaluation of an IoT device designed for transparent traffic analysis. In: IEEE. *2018 Workshop on Communication Networks and Power Systems (WCNPS)*. Brasília, DF, 2018. p. 1–5.

STOJMENOVIC, I.; WEN, S. The fog computing paradigm: Scenarios and security issues. In: *2014 Federated Conference on Computer Science and Information Systems*. [S.l.: s.n.], 2014. p. 1–8.

TALEB, T.; DUTTA, S.; KSENTINI, A.; IQBAL, M.; FLINCK, H. Mobile Edge Computing Potential in Making Cities Smarter. *IEEE Communications Magazine*, IEEE, v. 55, n. 3, Março 2017.

TANEJA, M.; DAVY, A. Resource Aware Placement of Data Analytics Platform in Fog Computing. *Procedia Computer Science*, v. 97, p. 153–156, Dezembro 2016.

TOMÁS, J. P. *Three smart traffic case studies*. [S.l.], 2017. Disponível em: <https://enterpriseiotinsights.com/20171108/channels/fundamentals/three-smart-traffic-case-studies-tag23-tag99>. Acesso em: 2019-02-22.

VARGHESE, B.; VILLARI, M.; RANA, O.; JAMES, P.; SHAH, T.; FAZIO, M.; RANJAN, R. Realizing Edge Marketplaces: Challenges and Opportunities. *IEEE Cloud Computing*, v. 5, p. 9–20, Novembro 2018.

VEERRAJU, T.; KUMAR, D. K. K. A survey on fog computing: research challenges in security and privacy issues. *International Journal of Engineering & Technology*, v. 7, p. 335, Março 2018.

WEINER, M.; JORGOVANOVIC, M.; SAHAI, A.; NIKOLIÉ, B. Design of a low-latency, high-reliability wireless communication system for control applications. In: *2014 IEEE International Conference on Communications (ICC)*. [S.l.: s.n.], 2014. p. 3829–3835. ISSN 1550-3607.

WIXTED, A. J.; KINNAIRD, P.; LARIJANI, H.; TAIT, A.; AHMADINIA, A.; STRACHAN, N. Evaluation of LoRa and LoRaWAN for wireless sensor networks. In: *2016 IEEE SENSORS*. [S.l.: s.n.], 2016. p. 1–3.

WORLD'S Largest Data Center: 350 E. Cermak. 2010. Disponível em: <https://www.datacenterknowledge.com/special-report-the-worlds-largest-data-centers/worlds-largest-data-center-350-e-cermak/>.

XU, J.; PALANISAMY, B.; LUDWIG, H.; WANG, Q. Zenith: Utility-aware resource allocation for edge computing. In: . [S.l.: s.n.], 2017.

ZACHARIAH, T.; KLUGMAN, N.; CAMPBELL, B.; ADKINS, J.; JACKSON, N.; DUTTA, P. The internet of things has a gateway problem. In: ACM. *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. [S.l.], 2015. p. 27–32.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, v. 1, n. 1, p. 7–18, Maio 2010. ISSN 1869-0238.

ZHU, Q.; WANG, R.; CHEN, Q.; LIU, Y.; QIN, W. IOT Gateway: BridgingWireless Sensor Networks into Internet of Things. In: *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. [S.l.: s.n.], 2010. p. 347–352.

ZIV, J.; LEMPEL, A. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, IEEE, v. 24, n. 5, p. 530–536, 1978.

Apêndices

BaseProtocol_esp8266.cpp

```
#include "BaseProtocol_esp8266.h"

int BaseProtocol_esp8266::create_service(int number, const char *name, String unit
    , bool numeric, String parameter){
  this->service[number] = Service(number, name, unit, numeric, parameter);
  return this->service[number].number;
  // return Service(number, name, unit, numeric, parameter);
}

bool BaseProtocol_esp8266::send_data(int service, float *data, int array_size, int
    sensitive=0) {
  char * char_data = float_to_char(data, array_size);
  this->DEVICE_REGISTERED = (!this->DEVICE_REGISTERED)? this->register_all(service
    , char_data, sensitive) : this->register_service_data(service, char_data,
    sensitive);
  free(char_data);
  return this->DEVICE_REGISTERED;
}

bool BaseProtocol_esp8266::send_data(int service, char *char_data, int sensitive=0
    ) {
  this->DEVICE_REGISTERED = (!this->DEVICE_REGISTERED)? this->register_all(service
    , char_data, sensitive) : this->register_service_data(service, char_data,
    sensitive);
  return this->DEVICE_REGISTERED;
}

void BaseProtocol_esp8266::device_identificator(){
 Serial.println("Indenticator");
 int address = 0;
 int bytes[8];
 bool result = SPIFFS.begin();
 Serial.println("SPIFFS opened: " + result);

  File f = SPIFFS.open("/UIOT.txt", "r");

 if (!f){
    Serial.println("Criando arquivo");
    File f = SPIFFS.open("/UIOT.txt", "w");

    int iterator = 0xFF;
    for (int i = 0; i < 8; i++){
      int a = millis();
```

```
      int b = rand() % 123 + 1;
      int unique = a * b;
      bytes[i] = unique & iterator;
    }


      this->mac_byte[4] = bytes[0];
      this->mac_byte[5] = bytes[1];

      for (int i = 0; i < 6; i++){
        this->chipset_byte[i] = bytes[i+2];
      }

      for(int i = 0; i < 6; i++){
        byte b = this->chipset_byte[i];
        this->chipset += this->nibble_to_char((b & 0xF0)>>4);
        this->chipset += this->nibble_to_char(b & 0xF);
        this->chipset += ":";
      }
      this->chipset.remove(this->chipset.length()-1,1);

      for(int i = 0; i < 6; i++){
        byte b = this->mac_byte[i];
        this->mac += this->nibble_to_char((b & 0xF0)>>4);
        this->mac += this->nibble_to_char(b & 0xF);
        this->mac += ":";
      }
      this->mac.remove(this->mac.length()-1,1);

      f.println(this->mac);
      f.println(this->chipset);

      while(f.available()) {
      //Lets read line by line from the file
      String line = f.readStringUntil("\n");
      Serial.println(this->mac);
      Serial.println(this->chipset);
    }

      // EEPROM.write(i + 2,bytes[i]);
      //unique = unique >> 8;
    // }
  } else{
    this->mac = f.readStringUntil("\r");
    f.readStringUntil("\n");
    this->chipset = f.readStringUntil("\r");
    this->mac_byte[4] = this->get_value_from_char(this->mac.charAt(12))*0x10;
    this->mac_byte[4] += this->get_value_from_char(this->mac.charAt(13));

    this->mac_byte[5] = this->get_value_from_char(this->mac.charAt(15))*0x10;
    this->mac_byte[5] += this->get_value_from_char(this->mac.charAt(16));
```

```
    Serial.println(this->mac_byte[4]);
    Serial.println(this->mac_byte[5]);


  }
  f.close();

  // String mac = f.readStringUntil("\n");
  // Serial.println(mac);

    Serial.print("MAC: ");
    Serial.println(this->mac);

    Serial.print("CHIPSET: ");
    Serial.println(this->chipset);



  // for (int i = 0; i< 6; i++){
  //    Serial.print(this->mac[i]);
  //    Serial.print(", ");
  // }

  // Serial.print("CHIPSET: ");
  // Serial.println(this->chipset);
  // for (int i = 0; i< 6; i++){
  //    Serial.print(this->chipset[i]);
  //    Serial.print(", ");
  // }

  // Serial.println("");

}


bool BaseProtocol_esp8266::register_all(int service, char *data, int sensitive){
  return this->register_device() && this->register_service_data(service, data,
     sensitive);
}

bool BaseProtocol_esp8266::register_service_data(int service,  char* data, int
     sensitive){
  Serial.println(this->service[service].registered);
  if (this->service[service].registered){
    if(this->register_data(service, data, sensitive)){
      return true;
    }
    else{
      this->service[service].registered = false;
      return false;
    }
  }
```

```
    else{
      Serial.println("Else");
      if(this->register_service(service) && this->register_data(service, data,
      sensitive)){
        Serial.println("AKI");
        Serial.println(this->service[0].registered);
        this->service[service].registered = true;
        Serial.println(this->service[service].registered);
        return true;
      }
      else{
        Serial.println("OIE");
        this->service[service].registered = false;
        return false;
      }
    }
  }


char *BaseProtocol_esp8266::make_client_data(char* json){
    // Serial.println("Registering Device");

    // Serial.println(strlen(json));
    json = (char*)malloc(2*sizeof(char));
    json[0] = "{";
    json[1] = "\0";
    json = this->append_json(json, "name", this->name.c_str(),0);
    json = this->append_json(json, "chipset", this->chipset.c_str(),0);
    json = this->append_json(json, "mac", this->mac.c_str(),0);
    json = this->append_json(json, "serial", this->serial.c_str(),0);
    json = this->append_json(json, "processor", this->processor.c_str(),0);
    json = this->append_json(json, "channel", this->channel.c_str(),0);
    json[strlen(json)-1] = "}";
    // Serial.println(json);
    return json;
}


char *BaseProtocol_esp8266::make_service_data(Service service, char* json){
    // Serial.println("Registering Service");

    // Serial.println(strlen(json));
    json = (char*)malloc(2*sizeof(char));
    json[0] = "{";
    json[1] = "\0";
    json  = this->append_json(json, "name", service.name,0);
    // Serial.println(json);
    // delay(1000);
    json = this->append_json(json, "chipset", this->chipset.c_str(),0);
    // Serial.println(json);
    // delay(1000);
    json = this->append_json(json, "mac", this->mac.c_str(),0);
```

```
    // Serial.println(json);
    // delay(1000);
    json = this->append_json(json, "parameter", service.parameter.c_str(),0);
    // Serial.println(json);
    // delay(1000);
    json = this->append_json(json, "number", String(service.number).c_str(), 1);
    // Serial.println(json);
    // delay(1000);
    json = this->append_json(json, "unit", service.unit.c_str(),0);
    // Serial.println(json);
    // delay(1000);
    json = this->append_json(json, "numeric", String(service.numeric).c_str(),1);
    json[strlen(json)-1] = "}";
    // Serial.println(json);
    // delay(1000);
    return json;
}

char *BaseProtocol_esp8266::make_raw_data(Service s, char *data, int sensitive,
    char* json){
    // Serial.println("Raw Data");
    // Serial.println(strlen(json));
    json = (char*)malloc(2*sizeof(char));
    json[0] = "{";
    json[1] = "\0";
    json = this->append_json(json, "chipset", this->chipset.c_str(), 0);
    json = this->append_json(json, "mac", this->mac.c_str(), 0);
    json = this->append_json(json, "sensitive", String(sensitive).c_str(),1);
    json = this->append_json(json, "serviceNumber", String(s.number).c_str(),1);
    json = this->append_json(json, "values", data, s.numeric);
    json[strlen(json)-1] = "}";
    // Serial.println(json);
    return json;
}


char* BaseProtocol_esp8266::append_json(char *json, const char* key, const char*
    value, int numeric){
    // Serial.println(strlen(json));
    if(key == "values" && !numeric){
      json = (char*) realloc (json, (strlen(key) + strlen(value) + 9 + strlen(json))
       * sizeof(char)); // 9 because "" and : and ,
      strcat(json, "\"");
      strcat(json, key);
      strcat(json, "\":[\"");
      strcat(json, value);
      strcat(json, "\"],");
      json[strlen(json)] = "\0";
    }else if(key == "values" && numeric){
      json = (char*) realloc (json, (strlen(key) + strlen(value) + 7 + strlen(json))
       * sizeof(char)); // 5 because "" and : and ,
      strcat(json, "\"");
```

```
      strcat(json, key);
239   strcat(json, "\":[");
      strcat(json, value);
241   strcat(json, "],");
      json[strlen(json)] = "\0";
243 }else if(key != "values" && !numeric){
      json = (char*) realloc (json, (strlen(key) + strlen(value) + 7 + strlen(json))
       * sizeof(char)); // 5 because "" and : and ,
245   strcat(json, "\"");
      strcat(json, key);
247   strcat(json, "\":\"");
      strcat(json, value);
249   strcat(json, "\",");
      json[strlen(json)] = "\0";
251 }else if(key != "values" && numeric){
      json = (char*) realloc (json, (strlen(key) + strlen(value) + 5 + strlen(json))
       * sizeof(char)); // 5 because "" and : and ,
253   strcat(json, "\"");
      strcat(json, key);
255   strcat(json, "\":");
      strcat(json, value);
257   strcat(json, ",");
      json[strlen(json)] = "\0";
259 }
    return json;
261 }

263 char* BaseProtocol_esp8266::float_to_char(float* float_array, int array_size){
    char *values;
265 values = (char*)malloc(1*sizeof(char));
    // values[0] = "[";
267 values[0] = "\0";
    String b;
269 int contador = 1;
    for(int i = 0; i < array_size; i++){
271   // Serial.println(dtostrf(float_array[i], 10, 3, b));
      b = String(float_array[i]);
273   contador += b.length() + 1;
      values = (char*) realloc (values, (contador ) * sizeof(char));
275   strcat(values, b.c_str());
      strcat(values, ",");
277   // Serial.print("values: ");
      // Serial.println(values);
279 }
    values[contador-2] = "\0";
281
    // strcat(values, "]");
283
    return values;
285 }

287 char BaseProtocol_esp8266::nibble_to_char(int nibble){
```

```
    return nibble + 48 + 7*(nibble > 9); //ascii based
}

byte BaseProtocol_esp8266::get_value_from_char(char hexa){
      return hexa - 48  - 7*(hexa>="A");

}
bool BaseProtocol_esp8266::register_service(int service){
    return false;
}
bool BaseProtocol_esp8266::register_device(){
    return false;
}
bool BaseProtocol_esp8266::register_data(int svc, char* dt, int stv){
    return false;
}
```

BaseProtocol_esp8266.h

```
#ifndef BaseProtocol_esp8266_H
#define BaseProtocol_esp8266_H

#include <FS.h>

class Service {
  public:
    Service (int number, const char *name, String unit, int numeric, String
    parameter){
        this->number = number;
        this->name = name;
        this->unit = unit;
        this->numeric = numeric;
        this->parameter = parameter;
    }
    Service(){}
    int number;
    const char * name;
    String *tags;
    String unit = "";
    int numeric;
    String parameter = "";
    bool registered = false;
  };

```

```
28 class BaseProtocol_esp8266 {
     public:
30         bool send_data();
           void device_identificator();
32         bool register_all(int, char*, int);
           bool register_service_data(int, char*, int);
34         virtual bool register_device() = 0;
           virtual bool register_service(int)= 0;
36         virtual bool register_data(int, char*, int)= 0;
           bool send_data(int , float*, int, int);
38         bool send_data(int , char*, int);
           char* float_to_char(float*, int);
40         int create_service(int , const char *, String, bool, String);
           char nibble_to_char(int);
42         char *make_client_data(char*);
           char *make_service_data(Service, char*);
44         char *make_raw_data(Service, char*, int, char*);
           byte get_value_from_char(char);
46         char *append_json(char*, const char*, const char*, int);

48     private:
           bool DEVICE_REGISTERED = false;
50

52     protected:
           byte mac_byte[6] = {0x9A, "I", "O", "T", 0, 0};
54         byte chipset_byte[6];
           String name = "";
56         String serial = "CA124";
           String processor = "ATMega";
58         String channel = "Ethernet";
           String chipset = "";
60         String mac = "";
           Service service[5];
62         int service_index = 0;
   };
64
   #endif
```

UHttp_esp8266.cpp

```
2 #include "UHttp_esp8266.h"

4 UHttp_esp8266::UHttp_esp8266(){}
```

```
void UHttp_esp8266::init(String device_name){
  delay(1000);
  this->device_identificator();
  this->name = device_name;

  this->OTA_config(device_name, false);
  WiFiManager wifimanager;
  wifimanager.autoConnect(device_name.c_str());
  Serial.println("Connection Successfull");
}


void UHttp_esp8266::handle(){
  ArduinoOTA.handle();
}

// void UHttp_esp8266::init(String device_name, bool force_erase_all){
//   delay(1000);
//   this->device_identificator();
//   this->name = device_name;
//
//   this->OTA_config(device_name, force_erase_all);
//   WiFiManager wifimanager;
//   wifimanager.autoConnect(device_name.c_str());
//   Serial.println("Connection Successfull");
// }

void UHttp_esp8266::OTA_config(String ota_name, bool force_erase_all){
  ArduinoOTA.onStart([force_erase_all]() {
    String type;
    if (ArduinoOTA.getCommand() == U_FLASH) {
      type = "sketch";
    } else { // U_SPIFFS
      type = "filesystem";
    }

    // NOTE: if updating SPIFFS this would be the place to unmount SPIFFS using
    SPIFFS.end()
    if(force_erase_all){
      SPIFFS.end();
    }

    Serial.println("Start updating " + type);
  });

  ArduinoOTA.onEnd([]() {
    Serial.println("\nEnd");
  });
  ArduinoOTA.onProgress([](unsigned int progress, unsigned int total) {
    Serial.printf("Progress: %u%%\r", (progress / (total / 100)));
  });
  ArduinoOTA.onError([](ota_error_t error) {
```

```
    Serial.printf("Error[%u]: ", error);
    if (error == OTA_AUTH_ERROR) {
      Serial.println("Auth Failed");
    } else if (error == OTA_BEGIN_ERROR) {
      Serial.println("Begin Failed");
    } else if (error == OTA_CONNECT_ERROR) {
      Serial.println("Connect Failed");
    } else if (error == OTA_RECEIVE_ERROR) {
      Serial.println("Receive Failed");
    } else if (error == OTA_END_ERROR) {
      Serial.println("End Failed");
    }
  });
  ArduinoOTA.setHostname(ota_name.c_str());
  ArduinoOTA.setPassword((ota_name + "_password").c_str());
  ArduinoOTA.begin();
}


void UHttp_esp8266::set_server(String server){
  Serial.println("set_server");
  this->server = server;
}

bool UHttp_esp8266::register_device(){
  char *data = NULL;
  data = this->make_client_data(data);
  this->http.begin(this->server + "/client");
  this->http.addHeader("Content-Type", "application/json");
  int code = this->http.POST(data);
  String payload = this->http.getString();
  Serial.println(data);
  Serial.println(code);
  Serial.println(payload);
  free(data);
  this->http.end();
  return (code==200);
}
bool UHttp_esp8266::register_service(int s){
  char *data = NULL;
  // data = this->make_service_data(s, data);
  data = this->make_service_data(this->service[s], data);

  this->http.begin(this->server + "/service");
  this->http.addHeader("Content-Type", "application/json");
  int code = this->http.POST(data);
  String payload = this->http.getString();
  Serial.println(code);
  Serial.println(payload);
  free(data);
  this->http.end();
  return (code==200);
```

```
}
bool UHttp_esp8266::register_data(int s, char* value, int sensitive){
    char *data = NULL;
    // data = this->make_raw_data(s, value, sensitive, data);
    data = this->make_raw_data(this->service[s], value, sensitive, data);
    this->http.begin(this->server + "/data");
    this->http.addHeader("Content-Type", "application/json");
    int code = this->http.POST(data);
    String payload = this->http.getString();
    Serial.println(code);
    Serial.println(payload);
    free(data);
    this->http.end();
    return (code==200);
}
```

UHttp_esp8266.h

```
#ifndef UHttp_esp8266_H
#define UHttp_esp8266_H

#include <ESP8266HTTPClient.h>
#include "BaseProtocol_esp8266.h"
#include <WiFiManager.h>
#include <ESP8266WiFi.h>
#include <ArduinoOTA.h>

class UHttp_esp8266 : public BaseProtocol_esp8266{
    public:
        UHttp_esp8266();
        UHttp_esp8266(String);
        void OTA_config(String);
        void OTA_config(String, bool);
        void init(String);
    void handle();
        void set_server(String);
        bool register_device();
        bool register_service(int);
        bool register_data(int, char*, int);

    private:
        // PubSubClient mqtt_client;
        String server;
        HTTPClient http;
        int mqtt_port = 1883;
};
#endif
```

gateway/gateway/core/database.py

```
from pymongo import MongoClient, errors
from bson.objectid import ObjectId
from bson import json_util
import json
import os


class Database:
    def __init__(self, dbname="gateway"):
        self._dbname = dbname
        self._protocols_id = 0

    def connect(self):
        if os.environ.get("MONGODB_URI") is None:
            mongodb = "mongodb://localhost:27017"
        else:
            mongodb = os.environ.get("MONGODB_URI")
            self._dbname = mongodb.rsplit("/", 1)[1]
        self._mongo_client = MongoClient(mongodb)
        self._collections = self._mongo_client[self._dbname]
        try:
            self._collections.command("ping")
        except errors.ServerSelectionTimeoutError:
            return False
        return True

    def start(self, blank=False):
        if blank:
            self._mongo_client.drop_database(self._dbname)
        if self._collections["config"].count_documents({}) ==
    0:
            self._collections["config"].insert_one({"protocols
    ": []})
```

```
33              if not os.environ.get("HEROKU"):
                    self._collections["config"].insert_one({"pid":
    0})
35          return True

37      def field_exists(self, collec, field=None):
            return True if self._collections[collec].find_one(
39                                                          {field: {"
   exists": True}) elseFalsedeffind_value(self, collec, field = None, value = None, id = False) :
   exists"

41


43

 : True}})
45              if res and "_id" in res:
                    res.pop("_id")
47              return res[field] if res else []

49          response = []
            cursor = self._collections[collec].find(value)
51          for doc in cursor:
                doc_parse = json.loads(json_util.dumps(doc))
53              if not  id:
                    doc_parse.pop("_id")
55              response.append(doc_parse)
            return response

57
        def add_value(self, collec, value):
59          value = [value] if  type(value) is not
    list else value
            for v in value:
61              if "_id" in v:
                    v.pop("_id")
63              self._collections[collec].insert_one(v)
                v.pop("_id")

65
            return True

67
        def update_value(self, collec, field, value):
69          self._collections[collec].update(
```

```
            {field: {"exists": True, "set": {field: value}})
71        return True

73    def update_obj(self, collec, query, update):
          self._collections[collec].update(query, update)
75        return True

77    def delete_values(self, collec, value=None, ids=None):
          count = 0
79        if value:
              return self._collections[collec].delete_one(value)
    .deleted_count
81        if ids:
              for  id in ids:
83                count += self._collections[collec].delete_one(
                                                        {"_id": ObjectId(
    id)}).deleted_count
85        return count

87    def drop_collec(self, collec):
          self._collections[collec].drop()
89        if collec == "config":
              # drop and restart
91            self._collections["config"].insert_one({"protocols
    ": []})
              if not os.environ.get("HEROKU"):
93                self._collections["config"].insert_one({"pid":
     0})

95    def close(self, delete=False):
          if delete:
97            self._mongo_client.drop_database(self._dbname)
  self._mongo_client.close()
```

gateway/gateway/core/factory.py

```
2 from queue import Queue
```

```
import threading
import time
import sys
import os

from datetime import datetime
from pathlib import Path
import logging.config
import requests
import logging
import yaml

from .database import Database
from .utils import responses


class Factory:
    def __init__(self, debug=False):
        self._active_protocols = []
        self._active_listeners = []
        self._last_response = Queue()
        self._running = True
        self._db_conn = None
        self._dims_url = None
        self._debug = debug
        self.logger = logging.getLogger("gateway.factory")

        self.start_middleware_connection()

    def start_middleware_connection(self):
        # config dims url reference
        if not os.environ.get("DIMS_URL"):
            self.logger.debug("DIMS url reference couldn\ t be found. "
                              "Setting default.")
            os.environ["DIMS_URL"] = "http://http://uiot-dims.herokuapp.com/"

        self._dims_url = os.environ["DIMS_URL"]
        self.logger.info(f"DIMS url point to : {self._dims_url}")
```

```
42    def start(self, protocols, dbname="gateway", blank=False):
          # database
44        self._db_conn = Database(dbname=dbname)
          self._db_conn.connect()
46        self._db_conn.start(blank)
          self.logger.info(f"Database started.")
48
          self._db_conn.update_value(collec="config", field="pid
     ",
50                                   value=os.getpid())

52        # setting protocols
          self._active_protocols = protocols
54
          self._db_conn.update_value(collec="config", field="
     protocols",
56                                   value=protocols)

58        self.logger.info("Settings Startup: Active protocols -
      "
                          f"{self._active_protocols}.")
60
          self.start_listeners()
62
      def start_listeners(self):
64        for listener in self._active_protocols:
              module = __import__(f"gateway.listeners.{listener}
     ",
66                                fromlist=[listener.upper()])
              mod_class =  getattr(module, listener.upper())
68
              listener_queue, sender_queue = Queue(), Queue()
70            try:
                  current_protocol = mod_class()
72                current_protocol.start_listener(listener_queue
     , sender_queue)
                  self.logger.info(f"Listener {listener} started
     .")
74            except ValueError as err:
                  self.logger.warning(err)
```

```
76             else:
                    self._active_listeners.append(current_protocol
    )
78                  threading.Thread(target=self.rcv_from_listener
    ,
                                     args=(current_protocol,
    listener_queue,
80                                         sender_queue),
                                     daemon=True).start()
82                  self.logger.debug(f"Listener {listener}
    started receive.")

84         self.logger.info("All listeners started.")

86         if not self._debug:
               while self._running:
88                 try:
                       self.logger.info("Factory alive.")
90                     time.sleep(900)   # 15 minutes
                   except KeyboardInterrupt:
92                     self._db_conn.update_value(collec="config"
    , field="pid",
                                                  value=0)
94                     quit()

96     def rcv_from_listener(self, listener, listener_queue,
    sender_queue):
           while self._running:
98             self.logger.debug(f"Waiting {listener.__class__.
    __name__} "
                                 "receive a message...")
100            message = listener_queue.get()
               message["channel"] = listener.__class__.__name__
102            self.logger.info(f"{listener.__class__.__name__}
    received a "
                                "message.")
104            result = self.gate(message, sender_queue)
               if self._debug:
106                self._last_response.put(result)

108    def stop_listeners(self):
```

```
        self.logger.info("Stopping listeners...")
110     for listener in self._active_listeners:
            listener.stop_listener()
112     self.logger.info("All listeners stopped.")

114 def gate(self, msg, sender_queue):
        entity = self.get_entity(msg)
116
        if not entity:
118         self.logger.warning("No entity could be found.")
            result = responses("WRONG_REQ")
120         sender_queue.put(result)
            return result
122
        self.logger.info(f"Message of type {entity} arrived.")
124     if entity == "data":
            if "time" not in msg:
126             msg["time"] = datetime.now().isoformat()

128     self.logger.debug(f"Try to send {entity} to dims...")
        result = self.send_to_dims(entity, msg)
130     sender_queue.put(result)
        self.logger.debug(f"{entity} sent!")
132     return result

134 def get_entity(self, msg):
        base = ["chipset", "mac"]
136     client_req = [*base, "name", "serial", "processor", "
    channel"]
        service_req = [*base, "name", "number", "parameter"]
138     data_req = [*base, "sensitive", "serviceNumber", "
    values"]
        all_req = [client_req, service_req, data_req]
140
        def validate(msg,  type): return
    all(key in msg for key in  type)
142
        def get_type_and_validate(msg):
144         res = {0: "client", 1: "service", 2: "data"}
            ret_type =  list(
    map(lambda typ: validate(msg, typ), all_req))
146         ret_loc =  list( filter(lambda it: ret_type[it],
```

```
                                    range( len(ret_type))))
148             return res[ret_loc[0]] if ret_loc else ret_loc

150         entity = msg.pop("entity") if "entity" in msg \
                else get_type_and_validate(msg)
152 
            return entity
154 
        def send_to_dims(self, entity, msg):
156         feedback = {"details": "Failed to establish a 
    connection with "
                                "DIMS. Try again later."}
158         try:
                ret = requests.post(f"{self._dims_url}{entity}",
    json=msg)
160             # 500 internal server error and 502 bad gateway
                if ret.status_code != 500 and ret.status_code != 
    502:
162                 self.logger.info(f"Data sent to dims to {self.
    _dims_url}. "
                                     f"Status code received : {ret
    .status_code}")
164                 self.logger.debug(f"Message sent : {msg}.")
                    return ret.json()
166         except requests.exceptions.ConnectionError:
                pass
168 
            self.logger.warning(feedback["details"])
170         resp =  dict(responses("INT_ERROR_REQ"))
            resp.update(feedback)
172         return resp

174     @property
        def active_protocols(self):
176         return self._active_protocols

178     @property
        def active_listeners(self):
180         return self._active_listeners

182     def close(self):
```

```
            self.logger.info("Closing factory...")
            self._running = False
            self.stop_listeners()
            self._db_conn.close()
            self.logger.info("Factory closed.")


def run():
    path = f"{Path(__file__).parents[2]}/logging.yaml"
    with  open(path, "rt") as f:
        config = yaml.safe_load(f.read())
    logging.config.dictConfig(config)

    logging.getLogger("gateway.factory").info(
        "UIoT Gateway started in background.")

    if  len(sys.argv) > 1:
        protocols = sys.argv[1: len(sys.argv)]
        Factory().start(protocols)
    else:
        msg = "Not enough arguments given in configuration to
   start process."
        print(msg)
logging.getLogger("gateway.factory").error(msg)
```

gateway/gateway/core/utils.py

```
SUCCESS_REQ = {"code": 200, "message": "Success"}
BAD_REQ = {"code": 400, "message": "Bad Request", "details":
           "The request could not be understood by the server
   due"
           " to malformed syntax. The client SHOULD NOT repeat
    the"
           " request without modifications."}
NFOUND_C_REQ = {"code": 422, "message": "Client not found", "
   details":
                "The client passed in service body could not
   be found. "
```

```
                    "Check the chipset and mac values."}
NFOUND_CS_REQ = {"code": 422, "message": "Client or Service not found",
                    "details": "The client or service passed in data body could "
                    "not be found. Check the chipset, mac and serviceNumber "
                    "values."}
INT_ERROR_REQ = {"code": 500, "message": "Internal Server Error",
                    "details": "The execution of the service failed "
                    "in some way."}
WRONG_REQ = {"code": 400, "message": "Bad Request", "details":
                "Was not found an entity pattern, try reading the api "
                "documentation in /docs."}


def responses(status):
    return {"SUCCESS_REQ": SUCCESS_REQ, "BAD_REQ": BAD_REQ,
            "NFOUND_C_REQ": NFOUND_C_REQ, "NFOUND_CS_REQ": NFOUND_CS_REQ,
            "INT_ERROR_REQ": INT_ERROR_REQ,
"WRONG_REQ": WRONG_REQ}.get(status, None)
```

gateway/gateway/listeners/base.py

```
class BaseListener:
    """Abstract class to guide the future protocols listeners.
    This class is a model to implement protocols listeners to gateway.
    Raises:
        NotImplementedError: If some protocol doesn t implement the required
        methods.
    """
    def start_listener(self, queue):
```

```
        """
        Start the listener from factory.
        Returns:
                True   if listener starts with success and
    False   otherwise.
        """
        raise NotImplementedError("Start listener not
    implemented.")

    def stop_listener(self):
        """
        Stop the listener from factory.
        Returns:
                True   if the listener completely stops and
    False   otherwise.
        """
raise NotImplementedError("Stop listener not implemented.")
```

gateway/gateway/listeners/http.py

```
from .http_endpoints.register import build_register_blueprint
from .http_endpoints.errors import error_handler
from .base import BaseListener

from flask import Flask, request, jsonify, Blueprint
from flask_swagger_ui import get_swaggerui_blueprint
from flask_cors import CORS

from werkzeug.serving import make_server
from yaml import Loader, load
from pathlib import Path
import threading
import datetime
import logging
import sys
import os


```

```python
def init_report_config():
    reports_path = f"{Path(__file__).parents[2]}/reports"
    if not os.path.exists(reports_path):
        os.makedirs(reports_path)
    dt = datetime.datetime.now().strftime("%Y-%m-%dT%H:%MZ")
    filename = f"{dt}_report.csv"
    handler = logging.FileHandler(f"{reports_path}/{filename}"
)
    log_formatter = logging.Formatter(
        "%(asctime)s.%(msecs)03d,%(size)s,%(message)s", "%H:%M
:%S")
    handler.setFormatter(log_formatter)
    logging.getLogger("gateway.listener.http").addHandler(
handler)


class FlaskServer(threading.Thread):
    """Flask server thread.
    This class implements HTTP protocol listeners to UIoT
gateway.
    Flask server details: host 0.0.0.0 and port 8000.
    Args:
        app (:obj: Flask ) Flask object reference.
    """
    def __init__(self, app):
        threading.Thread.__init__(self)
        self.daemon = True
        env_port = os.environ.get("PORT")
        local_port = 8000 if not env_port else env_port
        self.srv = make_server(host="0.0.0.0", port=local_port
, app=app,
                              threaded=True)
        self.ctx = app.app_context()
        self.ctx.push()

    def run(self):
        self.srv.serve_forever()

    def shutdown(self):
        self.srv.shutdown()

```

```
56 class HTTP(BaseListener):
       """HTTP protocol listener.
58     This class implements HTTP protocol listeners to UIoT
   gateway.
       """
60     def __init__(self):
           self.queue_fac_up, self.queue_fac_down = None, None
62         # flask app thread
           self.app_thread = None
64         # flask configuration
           self.app = Flask(__name__)
66         self.app.register_blueprint(self.show_docs(),
   url_prefix="/docs")
           self.app.register_blueprint(build_register_blueprint(
   self.app))
68         self.app.register_blueprint(error_handler)
           CORS(self.app)
70
       def start_listener(self, queue_fac_up, queue_fac_down):
72         app = self.app
           self.queue_fac_up, self.queue_fac_down = queue_fac_up,
    queue_fac_down
74
           @app.route("/report", methods=["GET"])
76         def report_data():
               init_report_config()
78             return jsonify({"code": 200, "message": "Success"
   }), 200

80         @app.route("/")
           def home():
82             return jsonify({"code": 200, "message": "Success"
   }), 200

84         @app.route("/message/<entity>", methods=["POST"])
           def server_message(entity):
86             message = request.get_json()
               logging.getLogger("gateway.listener.http").debug(
88                 f"HTTP recv : \"{message}\"")
               if entity == "data" and "counter" in message:
```

```
90                     self.report_logging(message)

92                 message["entity"] = entity
                   # wait response from gateway validation
94                 self.queue_fac_up.put(message)

96                 response = self.queue_fac_down.get()
                   return jsonify(response), response["code"]
98
               # start flask app
100            self.app_thread = FlaskServer(app)
               self.app_thread.start()
102            return True

104        def show_docs(self):
               doc_path = f"{Path(__file__).parents[1]}/docs/swagger.
       yaml"
106            swagger_yml = load(
       open(doc_path, "r"), Loader=Loader)
               return get_swaggerui_blueprint("/docs", doc_path,
108                                                 config={"spec":
       swagger_yml})

110        def report_logging(self, message):
               if message["counter"] == -1:
112                # restart global logging
                   for handler in logging.getLogger("gateway.listener
       .http"
114                                                     ).handlers[:]:
                       logging.getLogger(
116                        "gateway.listener.http").removeHandler(
       handler)
               else:
118                logging.getLogger("gateway.listener.http").debug(
                       f"\"{message}\"", extra={"size": sys.getsizeof
       (message)})
120
           def stop_listener(self):
122            # stop flask app server thread
               self.app_thread.shutdown()
124 return True
```

gateway/gateway/listeners/mqtt.py

```
from gateway.listeners.base import BaseListener

import os
import paho.mqtt.client as paho
import ast


class MQTT(BaseListener):
    """MQTT protocol listener.
    This class implements MQTT protocol listeners to UIoT
  gateway.
    """
    def __init__(self):
        self.client = paho.Client()
        self.queue_fac_up, self.queue_fac_down = None, None

    def start_listener(self, queue_fac_up, queue_fac_down):
        self.queue_fac_up, self.queue_fac_down = queue_fac_up,
    queue_fac_down

        def on_connect(client, userdata, flags, rc):
            self.client.subscribe("#")

        def on_message(client, userdata, msg):
            message = ast.literal_eval(msg.payload.decode())
            # submit message to gate process
            self.queue_fac_up.put(message)

        self.client.on_connect = on_connect
        self.client.on_message = on_message
        if os.environ.get("MQTT_URI") is None:
            mqtt_host = "0.0.0.0"
        else:
            mqtt_host = os.environ.get("MQTT_URI")
```

```
        self.client.connect(mqtt_host)
        self.client.loop_start()

    def stop_listener(self):
        self.client.loop_stop()
self.client.disconnect()
```

gateway/gateway/listeners/tcp.py

```
from gateway.listeners.base import BaseListener

import threading
import socket
import json


class TCP(BaseListener):
    """TCP protocol listener.
    This class implements TCP protocol listeners to UIoT
  gateway.
    """
    def __init__(self):
        self.queue_fac_up, self.queue_fac_down = None, None
        self.run = True
        self.port = 5010
        self.host = ""

    def start_listener(self, queue_fac_up, queue_fac_down):
        self.queue_fac_up, self.queue_fac_down = queue_fac_up,
    queue_fac_down
        # ipv4 (AF_INET) socket obj using tcp protocol (
  SOCK_STREAM)
        self.sock = socket.socket(socket.AF_INET, socket.
  SOCK_STREAM)
        self.sock.bind((self.host, self.port))
        self.sock.listen(5)  # max backlog of connections
        threading.Thread(target=self.rcv, daemon=True).start()
        return True
```

```
26
    def handle_client_connection(self, client_sock):
28      message = client_sock.recv(1024)
        if not message:
30          client_sock.close()
        decod_msg = json.loads(message.decode())
32      self.queue_fac_up.put(decod_msg)  # submit message to
   gate process

34  def rcv(self):
        while self.run:
36          client_sock, address = self.sock.accept()  #
   accept connection
            threading.Thread(
38              target=self.handle_client_connection,
                args=(client_sock,)
40          ).start()

42  def stop_listener(self):
        self.run = False
44 self.sock.close()
```

gateway/gateway/listeners/udp.py

```
from gateway.listeners.base import BaseListener
2
import threading
4 import socket
import json
6

8 class UDP(BaseListener):
    """UDP protocol listener.
10  This class implements UDP protocol listeners to UIoT
   gateway.
    """
12  def __init__(self):
        self.queue_fac_up, self.queue_fac_down = None, None
```

```
14          self.run = True
            self.port = 5005
16          self.host = "0.0.0.0"

18      def start_listener(self, queue_fac_up, queue_fac_down):
            self.queue_fac_up, self.queue_fac_down = queue_fac_up,
        queue_fac_down
20          self.sock = socket.socket(socket.AF_INET, socket.
    SOCK_DGRAM)
            self.sock.setsockopt(socket.SOL_SOCKET, socket.
    SO_REUSEADDR, 1)
22          self.sock.bind((self.host, self.port))
            threading.Thread(target=self.rcv, daemon=True).start()
24          return True

26      def rcv(self):
            while self.run:
28              message, client = self.sock.recvfrom(1024)
                decod_msg = json.loads(message.decode())
30              # submit message to gate process
                self.queue_fac_up.put(decod_msg)
32
        def stop_listener(self):
34          self.run = False
self.sock.close()
```

gateway/gateway/listeners/udp.py

```
1 from gateway.listeners.base import BaseListener

3 import threading
  import socket
5 import json

7
  class UDP(BaseListener):
9     """UDP protocol listener.
```

```
    This class implements UDP protocol listeners to UIoT
   gateway.
    """
    def __init__(self):
        self.queue_fac_up, self.queue_fac_down = None, None
        self.run = True
        self.port = 5005
        self.host = "0.0.0.0"

    def start_listener(self, queue_fac_up, queue_fac_down):
        self.queue_fac_up, self.queue_fac_down = queue_fac_up,
 queue_fac_down
        self.sock = socket.socket(socket.AF_INET, socket.
   SOCK_DGRAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.
   SO_REUSEADDR, 1)
        self.sock.bind((self.host, self.port))
        threading.Thread(target=self.rcv, daemon=True).start()
        return True

    def rcv(self):
        while self.run:
            message, client = self.sock.recvfrom(1024)
            decod_msg = json.loads(message.decode())
            # submit message to gate process
            self.queue_fac_up.put(decod_msg)

    def stop_listener(self):
        self.run = False
self.sock.close()
```

gateway/gateway/listeners/zigbee.py

```
import json
import logging
from digi.xbee.devices import XBeeDevice
from serial.serialutil import SerialException

```

```
from gateway.listeners.base import BaseListener
7

9 logger = logging.getLogger("gateway.listener.zigbee")

11
class ZIGBEE(BaseListener):
13     """ZigBee protocol listener.
    This class implements ZigBee protocol listeners to UIoT
   gateway.
15     """
    def __init__(self):
17         self.port = "/dev/ttyUSB0"
        self.baud_rate = 9600
19         self.device = XBeeDevice(self.port, self.baud_rate)
        try:
21             self.device. open()
        except SerialException as err:
23             logger.error(err)
            raise ValueError("Closing ZigBee listener due to
   error. Restart "
25                              "gateway to try start ZigBee
   again.")

27     def rcv(self, xbee_message):
        address = xbee_message.remote_device.get_64bit_addr()
29         logger.debug(f"Received data from {address}.")
        data = xbee_message.data.decode("utf8")
31         logger.debug(f"Raw data : {data}.")
        try:
33             decod_msg = json.loads(data)
        except ValueError as err:
35             logger.warning("Data received isn\ t format as
   JSON. "
                           "You can only send JSON-
   serializable data.")
37         else:
            logger.debug(f"Data decoded : {decod_msg}.")
39             self.queue_fac_up.put(decod_msg)  # submit message
    to gate process
```

```
41     def start_listener(self, queue_fac_up, queue_fac_down):
           self.queue_fac_up, self.queue_fac_down = queue_fac_up,
       queue_fac_down
43         self.device.add_data_received_callback(self.rcv)

45     def stop_listener(self):
           if self.device is not None and self.device.is_open():
47 self.device.close()
```

Readme.md

```
1
  UIoT Gateway
3 ------------

5 **UIoT** is an *Internet of Things* open middleware. Made to handle and
      store knowledge and be a layer for IoT applications and devices. **UIoT
      ** is maintained by the [Universal Internet of Things](https://uiot.org
      ) and the [University of Bras lia](http://www.unb.br).

7 **Gateway** is a communication bridge between devices and UIoT package.

9 Requirements
  ------------
11
  - Tools required:
13   - [Python3.7](https://www.python.org/downloads/)
     - [MongoDB](https://docs.mongodb.com/manual/installation/)
15
  Run
17 ---
  - Running using docker-compose:
19    bash
  sudo docker-compose build
21 sudo docker-compose up

23 To modify database, mqtt or data interface connection, change the
      environment variables set in the Dockerfile:
     bash
25 ENV MONGODB_URI "mongodb://mongo:27017"
  ENV DIMS_URL "http://homol.redes.unb.br/uiot/dims/"
27 ENV MQTT_URI "mqtt_broker"
```

```
29
- Running using standalone dependencies:
31   bash
sudo service mongod start  # starts mongodb
33 sudo mkdir /var/log/uiot  # creates UIoT logs folder
sudo chown USER:USER /var/log/uiot  # give UIoT logs folder permissions
35 python3 -m venv venv  # creates python 3 virtual environment
source venv/bin/activate  # starts python 3 virtual environment
37 pip install -e .  # install UIoT gateway app and its requirements
gateway  # run UIoT gateway app in background
39

41  UIoT Gateway  will start the process in background, to stop the service
     run the following command:

43   bash
gateway --stop  # closes the gateway
45

47 The command above will stop the gateway and show  UIoT Gateway finished
     if a process gateway was running in background, otherwise will show
    UIoT Gateway is not running .

49 For further options, check:

51   bash
gateway --help
53

55 Protocols
---------
57
There are some configurations according to the protocols your gateway will
     be supporting. These configurations are described bellow:
59
* ZigBee
61
sudo usermod -a -G dialout
```