Pós-Graduação em Ciência da Computação

João Victor Lucena do Monte

IOTALKER: Aprimorando a integração de dispositivos IoT

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

João Victor Lucena do Monte

IOTALKER: Aprimorando a integração de dispositivos IoT

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Redes de Computadores
Orientador: Djamel Hadj Fawzi Sadok

Recife
2019

Catalogação na fonte
Bibliotecária Mariana de Souza Alves CRB4-2106

M772i Monte, João Victor Lucena do
IOTALKER: aprimorando a integração de dispositivos IoT – 2019.
65f.: il., fig., tab.

Orientador: Djamel Hadj Fawzi Sadok
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da computação. Recife, 2019.
Inclui referências.

1. Redes de Computadores. 2. IoT. 3. Sensores. 4. Middlewares. I. Sadok, Djamel Hadj Fawzi (orientador). II. Título.

004.6 CDD (22. ed.) UFPE-MEI 2019-142

João Victor Lucena do Monte

“IOTALKER: APRIMORANDO A INTEGRAÇÃO DE DISPOSITIVOS IOT”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 13/03/2019.

BANCA EXAMINADORA

_____

Prof. Dr. Odilon Maroja da Costa Pereira Filho
Centro de Informática/UFPE

_____

Prof. Dr. Rafael Roque Aschoff
Instituto Federal de Pernambuco / Campus Palmares

_____

Prof. Dr. Djamel Fawzi Hadj Sadok
Centro de Informática/UFPE

AGRADECIMENTOS

Agradeço primeiramente a Deus, que me deu saúde e força necessárias para chegar até aqui.

Aos meus pais, Carlos e Ilane, por todo apoio, incentivo, carinho e amor que sempre me deram. Além da compreensão nos momentos em que estive ausente. Apesar das dificuldades sempre se esforçaram para nunca deixar faltar nada e me deram ótimos exemplos a serem seguidos. Sempre por perto nos momentos fáceis e difíceis.

Ao meu irmão Caio, por tanta amizade e companhia. Sempre partilhando todos os momentos sejam eles quais forem ao longo da vida cotidiana. Desde criança seguindo essa jornada juntos.

À minha namorada, Marília, que com muito amor também se tornou alguém sempre presente em todos os momentos da minha jornada. Sempre dispensando atenção, carinho, cuidado e sendo ainda melhor que tudo o que eu esperava de uma companheira.

A todos os professores que contribuíram ao longo da minha jornada, especialmente aos professores Djamel Sadok e Judith Kelner, por todas orientações, recomendações e oportunidades que foram dadas a mim, não apenas durante o desenvolvimento deste Trabalho, mas ao longo de quase sete anos em que trabalho no Grupo de Pesquisa em Redes e Telecomunicações.

A todos os meus colegas e amigos do colégio, da faculdade e do GPRT, que dividiram comigo não apenas momentos de estudo, projetos e aprendizagem, mas também de descontração e bom humor.

RESUMO

No contexto de Internet das coisas, que vem ganhando cada vez mais força nos dias atuais, a heterogeneidade de dispositivos e protocolos de comunicação dificulta o trabalho de desenvolvedores, acrescentando mais etapas no processo de desenvolvimento de aplicações baseadas nesses dispositivos. Este trabalho tem por objetivo a implementação e testes de uma plataforma de software destinada a desenvolvedores, capaz de prover a integração de dispositivos IoT de maneira simplificada: o IoTalker. A plataforma será capaz de se comunicar com estes dispositivos através da utilização de diferentes protocolos. Será possível configurar diversas operações e parâmetros nos dados obtidos e por fim torná-los disponíveis através de outros protocolos para IoT. Todas as etapas, até as mais complexas, do processo de utilização do IoTalker podem ser realizadas rapidamente e sem a necessidade de programação. O IoTalker conta com suporte a um conjunto inicial de protocolos, mas foi implementado de modo a ser expansível, permitindo sua comunicação com cada vez mais dispositivos IoT.

Palavras-chaves: IoT. Sensores. Middlewares. Gateway. ABNT14522.

ABSTRACT

In the ever-increasing context of the Internet of Things, the heterogeneity of devices, communication and protocols makes it difficult for developers to work, adding more steps in the process of developing applications based on these sensors and actuators. The objective of this work is the implementation and testing of a software platform designed for developers, able to provide the integration of IoT devices in a simplified way: IoTalker. The platform will be able to communicate with these devices using different protocols. It will be possible to configure various operations and parameters in the data obtained and finally make them available through other protocols for IoT. Every step, even the most complex, in the process of using IoTalker can be performed quickly and without the need for programming. IoTalker supports a set of protocols, but was implemented to be scalable, allowing them to communicate with more and more IoT devices.

Key-words: IoT. Sensors. Middleware. Gateway. ABNT14522.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BLE	Bluetooth Low Energy
CoAP	Constrained Application Protocol
DPWS	Devices Profile for Web Services
GPRT	Grupo de Pesquisa em Redes e Telecomunicações
GUI	Graphical user interface
HTML	Hypertext Markup Language
HTTP	HyperText Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
MIB	Management Information Base
MQTT	Message Queue Telemetry Transport
REST	Representational state transfer
SNMP	Simple Network Management Protocol
SSU	Saída Serial de Usuário
UFPE	Universidade Federal de Pernambuco
url	Uniform Resource Locator
USB	Universal Serial Bus
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

SUMÁRIO

1 INTRODUÇÃO

Segundo (GARTNER,), até 2020 haverão bilhões de dispositivos conectados à internet. Cada vez mais, o mundo caminha para a era da IoT. Os dispositivos conectados a internet não se limitarão aos tradicionais que temos hoje, como computadores, tablets e smartphones. Também estarão conectados equipamentos e eletrodomésticos, como: geladeiras, aparelhos de ar-condicionado, sistemas de lâmpadas e máquinas de lavar-roupa. Alguns destes, inclusive, já contam com sistemas e interfaces de controle remoto.

Muitos destes dispositivos originalmente não possuíam conectividade com a rede de *Internet*. A tecnologia IoT lhes deu poder de processamento e comunicação. Além de suas funcionalidades habituais, serão capazes de gerar, processar e consumir informação. A interface de controle oferecida por tais dispositivos já é um conceito pré-estabelecido e aceito pelos usuários, mas essa evolução abre um novo leque de possibilidades. Desde o aumento da eficiência em seu uso até mesmo à comodidade ao controlar seus utensílios.

Com um sistema de controle automatizado, um usuário pode remotamente programar sua máquina de lavar para operar durante períodos do dia em que a energia elétrica é mais barata, economizando na conta de luz, por exemplo.

Em épocas muito quentes, seria possível também ligar os aparelhos de ar condicionado de casa antes mesmo de chegar, permitindo aos usuários o conforto de ter um ambiente já refrigerado à sua disposição.

Uma vez que esses dispositivos tornam-se inteligentes, são acrescidos à nova miríade de dispositivos já existentes, contribuindo para o aumento de sua diversidade e heterogeneidade. É possível encontrar dispositivos nos mais diversos contextos como: Telefonia, eletrodomésticos, computação, indústria, saúde, mobilidade, etc. Todos trazendo suas particularidades em termos de formas de controle, protocolos suportados, comandos, programações, modos de operação e funcionamento. Na mesma medida que essa heterogeneidade facilita a vida dos usuários, traz consigo um grande desafio aos desenvolvedores que trabalham nessa área: lidar com um grande número de protocolos e tipos de comunicação. Sejam esses desenvolvedores os responsáveis pelo desenvolvimento direto dos equipamentos, ou desenvolvedores terceiros que por algum motivo necessitam se comunicar com tais dispositivos via software.

Cada protocolo de comunicação utilizado por esses equipamentos funciona de modo particular. Um *nobreak* que aceita requisições SNMP exigirá *snippets* de código e bibliotecas diferentes de um medidor de energia que aceita conexão RS485. E ainda os que compartilham o mesmo protocolo de acesso, podem ter, cada um, uma API de interação diferente. Dois *smartplugs Xbee* podem ter valores diferentes para a mesma porta. Enquanto na porta D0 um *smartplug* da marca X apresenta o valor de tensão, o da marca Y pode apresentar o valor de corrente. Cada fabricante usa e manipula os protocolos da

forma que quiser durante o desenvolvimento de seus dispositivos inteligentes. Salvo exceções, como por exemplo o padrão especificado pela norma ABNT 14522 (ABNT,), que determina como e quais grandezas os medidores de energia brasileiros devem armazenar e disponibilizar as informações.

Diante de toda essa problemática em específico, surgiram os *middlewares* IoT. Essas plataformas tem por objetivo principal trazer toda essa miríade a um padrão comum para os desenvolvedores (e até usuários convencionais) que desejam utilizá-las em suas aplicações ou para criar sistemas integrados e programáveis. São disponibilizadas opções já prontas para comunicação com tais dispositivos, facilitando essa tarefa. Algumas dão suporte a uma grande gama de dispositivos diferentes e/ou disponibilizam repositórios para aumentar ainda mais essa quantidade. As ferramentas oferecidas podem envolver Graphical user interface (GUI), bibliotecas, linguagens de *script*, *drivers* e outras.

Quando o dispositivo não é suportado pelo *middleware* IoT, o problema é um pouco maior. O desenvolvedor não conta com muitos recursos e o conhecimento em programação torna-se mais necessário para tal tarefa. Por mais que alguns ainda ofereçam linguagens de *script* próprias, algum conhecimento de programação ainda é necessário.

Considerando a contextualização apresentada e as questões levantadas pode-se enumerar as seguintes perguntas de pesquisa, as quais essa pesquisa de mestrado poderá, ao seu fim, responder:

- Quais os tipos de plataformas que permitem a comunicação com dispositivos IoT?
- Quais as funcionalidades já oferecidas pelas principais plataformas que permitem a integração de dispositivos IoT?
- Como pode ser a arquitetura de uma plataforma para integração de dispositivos IoT?
- Que funcionalidades uma plataforma pode oferecer para facilitar a integração de dispositivos IoT?

Este trabalho tem por objetivo geral apresentar a proposta e implementação de uma plataforma de software que oferece facilidades nesse contexto, combinando as melhores funcionalidades das ferramentas e técnicas relacionadas da literatura e trazendo praticidade e simplicidade em seu uso.

O IoTalker consiste em gateway IoT no qual o usuário pode criar e padronizar uma ponte de comunicação com seus diversos dispositivos através da manipulação de arquivos de configuração, e futuramente, uma GUI.

Como objetivos específicos, pode-se citar:

- Fazer um estudo dos trabalhos relacionados presentes na literatura.
- Planejar a arquitetura de software da plataforma.

- Implementar a arquitetura planejada.
- Validar o funcionamento da plataforma em casos de uso.

A apresentação do trabalho nessa dissertação está organizada da seguinte forma nas seções que se seguem:

1. Introdução, que trouxe uma contextualização sobre IoT e integração de dispositivos heterogêneos.
2. Fundamentação teórica, trazendo conceitos importantes que são necessários para o compreendimento do trabalho.
3. Trabalhos relacionados, trazendo e categorizando trabalhos que apresentam características semelhantes.
4. Desenvolvimento, contendo os detalhes arquiteturais da implementação e manipulação do IoTalker.
5. Casos de uso, contendo detalhes sobre a aplicação do IoTalker em dois cenários.
6. Conclusão e considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo descreve um conjunto de tecnologias de software, protocolos de comunicação e bancos de dados relevantes ao desenvolvimento de sistemas IoT.

2.1 PARADIGMA CLIENTE-SERVIDOR

É uma das arquiteturas mais antigas desenvolvidas para o projeto de redes de computadores. Tem-se duas entidades envolvidas na comunicação propriamente dita: o cliente e o servidor (KUROSE; ROSS, 2012). O cliente realiza requisições ao servidor, que as responde. O servidor nunca deverá enviar requisições ao cliente.

Esse padrão é bastante utilizado quando se tem o servidor como central de processamento. Máquinas robustas que realizam manipulações complexas e atendem muitas requisições de diversos clientes.

Considerando o contexto de IoT, encontram-se algumas dificuldades relacionadas à utilização desse paradigma. Tem-se muitos dispositivos que atuarão como cliente, realizando milhares de requisições por segundo e podendo causar sobrecarga em seus servidores. A medida que a quantidade de dispositivos aumenta, essa sobrecarga cresce.

O contexto IoT também demanda mais flexibilidades, no sentido de clientes poderem se comunicar entre si. Esse contexto também demanda mais estruturas intermediárias na arquitetura, como *middlewares* e *gateways*. Essas demandas tornam o tradicional cliente-servidor não muito adequado à IoT.

2.2 MQTT

Message Queue Telemetry Transport (MQTT) (MQTT,) é um protocolo para comunicação *machine-to-machine* e IoT. Foi projetado para o envio de mensagens no paradigma *publish/subscribe* de forma leve e simples, tornando-o ideal para cenários que dispõem de pouca banda e de máquinas com poucos recursos computacionais.

O padrão de comunicação MQTT se baseia no conceito de tópicos. Existem *publishers* e *subscribers*. Os *subscribers* se registram no serviço de mensageria informando qual seu(s) tópico(s) de interesse. Os *publishers* publicam mensagens no serviço de mensageria atribuindo-as a um tópico específico. Todos os *subscribers* que tem interesse nesse tópico receberão as mensagens publicadas, e assim acontece a comunicação entre ambos, embora não diretamente.

Existe mais uma entidade envolvida nessa comunicação, além dos *publishers* e *subscribers*: o *broker*. Este *broker* é um modulo de software que fornece a base para o serviço de mensageria em si. Permite que *publishers* publiquem suas mensagens e *subscribers* as

recebam, ou seja, ele faz a ponte entre *publishers* e *subscribers* ou *assinantes*. Ao longo do desenvolvimento do projeto, foi utilizado o *broker mosquitto* (MOSQUITTO...,).

São permitidas muitas opções como a exigência de autenticação e armazenamento de mensagens enviadas e recebidas.

2.3 BANCO DE DADOS NOSQL E MONGODB

Bancos de dados NoSQL (AMAZON,) são bancos que fogem do padrão relacional. Enquanto os bancos relacionais (clássicos) se limitam ao armazenamento de dados em tabelas, os bancos NoSQL permitem o armazenamento em formas diferentes e mais versáteis, como documentos e grafos.

Esses bancos são mais flexíveis e facilitam o desenvolvimento de aplicações modernas, podendo adequar-se melhor à suas necessidades. São conhecidos pela facilidade de desenvolvimento e boa performance. Muitos deles foram implementados tendo-se o contexto IoT como objetivo. Como exemplos, tem-se o MongoDB (MONGODB,) e OrientDB (ORIENTDB,).

No projeto IoTalker foi utilizado o banco de dados MongoDB. Esse banco apresenta uma arquitetura simples para o armazenamento dos dados. Os dados são armazenados no formato de JSON, em unidades chamadas documentos. Esses documentos são agrupados em *collections* e ficam disponíveis para realização de operações. Cada documento deve possuir uma chave *"_id"*, usada pelo banco para a indexação.

Existem APIs para trabalhar com MongoDB em muitas linguagens. Nesse trabalho, foi utilizada a biblioteca PyMongo (PYMONGO,), para Python3.

2.4 SNMP

Protocolo para gerenciamento de dispositivos em redes IP e um dos componentes do conjunto de protocolos da internet. É utilizado para verificar se equipamentos estão operando em suas condições normais. Permite a execução de comandos entre os nós da rede IP, requisitando e definindo valores.

Um dos nós, o gerente, é responsável pelo gerenciamento, e os outros nós disponibilizam informações sobre si ao(s) gerente(s). As informações disponíveis são listadas em uma espécie de catálogo, chamado MIB.

Como estudo de base para a elaboração do projeto, foi analisado o passo a passo para comunicação com dispositivos SNMP utilizando Python3. Foi utilizada a biblioteca PySNMP (PYSNMP,), que permite entre outras funcionalidades, a requisição de valores armazenados nos campos da MIB do dispositivo.

O dispositivo SNMP utilizado nessa etapa foi um *nobreak* instalado no GPRT. Utilizando PySNMP foi elaborada uma requisição solicitando alguns valores do *nobreak* e foram recebidas as respostas corretas. A requisição é mostrada na figura 1.

```
argsQuery = (SnmpEngine(),
             CommunityData('public', mpModel=0),
             UdpTransportTarget((serverIp,serverPort)),
             ContextData(),
             ObjectType(ObjectIdentity('UPS-MIB', 'upsIdentName', 0)), #upsId
             ObjectType(ObjectIdentity('UPS-MIB', 'upsBatteryStatus', 0)), #Battery Status
             ObjectType(ObjectIdentity('UPS-MIB', 'upsSecondsOnBattery', 0)), #Time using the battery (sec)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsEstimatedMinutesRemaining', 0)), #Time remaining (min)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsEstimatedChargeRemaining', 0)), #Charge remaining (percent)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsBatteryVoltage', 0)), #Battery voltage (0.1 volts)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsBatteryCurrent', 0)), #Battery current (0.1 amper)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsBatteryTemperature', 0)), #Battery temperature (C)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsInputFrequency', 1)), #Input frequency (0.1 hz)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsInputVoltage', 1)), #Input voltage (RMS Volt)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsInputCurrent', 1)), #Input Current (RMS 0.1 Amper)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsInputTruePower', 1)), #Input power (watts)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsOutputFrequency', 0)), #Output frequency (0.1 Hz)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsOutputVoltage', 1)), #Output voltage (RMS volts)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsOutputPercentLoad', 1)), #Output usage
             ObjectType(ObjectIdentity('UPS-MIB', 'upsOutputCurrent', 1)), #Output current (0.1 RMS Amper)
             ObjectType(ObjectIdentity('UPS-MIB', 'upsOutputPower', 1))) #Output power(Watts)
```

Figura 1 – Requisição para o *nobreak* solicitando valores para alguns dos campos de sua MIB utilizando PySNMP.

2.5 MODBUS

Um antigo protocolo para comunicação de dados. É uma solução facilmente adaptável e barata. O *modbus* opera em um modelo *mestre-escravo*. Nesse modelo, o nó escravo apenas aguarda os comandos do nó mestre, assemelhando-se ao modelo *cliente-servidor*. Quando solicitado, os escravos enviam ao mestre os valores lidos em seus registradores, independente das circunstâncias e de sua natureza.

O dispositivo utilizado para o estudo da comunicação *modbus* foi um medidor de energia modelo 7550E da marca CCK, mostrado na figura 2. Utilizando a biblioteca MinimalModbus (MINIMALMODBUS,) (Python3) bem como a referência fornecida pelo fabricante informando os endereços onde o medidor armazena cada valor, foi possível o desenvolvimento de um código que solicita ao medidor um conjunto de valores selecionados, atuando como mestre.

Figura 2 – Medidor 7550E da marca CCK utilizado para o estudo da comunicação *modbus*.

2.6 ABNT 14522

Norma que define os padrões que os medidores de energia elétrica devem obedecer para facilitar sua comunicação com diferentes tipos de sistemas de medição de energia. ABNT 14522 especifica vários modelos e padrões diferentes que podem ser seguidos. Nesse trabalho foram utilizados dois destes: porta óptica e Saída Serial de Usuário (SSU) (ABNT,).

O padrão porta-óptica possui uma série de comandos que podem ser enviados ao medidor de energia e o mesmo deve respondê-las, tratando-se de uma comunicação bidirecional. Um dos principais comandos de interesse considerando o contexto IoT é o comando 14 da norma, que solicita as grandezas instantâneas mensuradas pelo medidor de energia. A figura 3-A mostra a sintaxe desse comando.

Após a realização de testes, foi observado que nem todos os medidores respondem corretamente a esse comando, embora todos pertençam ao padrão. Utilizando softwares para monitoramento da comunicação entre os medidores e aplicativos dos fabricantes, foi descoberto o envio de outro comando, com a sintaxe expressa na figura 3-B.

O padrão SSU segue outras regras. A comunicação é assíncrona, unidirecional e o medidor envia um conjunto de mensagens contendo o número de pulsos de energia ativa, número de pulsos de energia reativa e um contador regressivo para o fechamento do intervalo de demanda.

O número de pulsos de energia ativa e reativa é uma marca da energia consumida. Cada medidor possui um valor específico de energia para o qual quando consumida, incrementa esse contador. Multiplicando-se esse valor pelo número de pulsos obtém-se a energia consumida.

Durante o tempo em que o contador regressivo é reduzido a zero, os pulsos de energia

3.1.2.1.9 Leitura das grandezas instantâneas – Comando com resposta simples

Comando

Octeto 001: 14 - Grandezas instantâneas
Octeto 002: Número de série do leitor MSB
Octeto 003: Número de série do leitor
Octeto 004: Número de série do leitor LSB
Octeto 005 até
Octeto 064: NULL
Octeto 065: CRC LSB
Octeto 066: CRC MSB

```
command = bytearray()
command.append(0x6)  #1 - Field indicating the command packet beginning
command.append(0x14) #2 - Field indicating the command number
command.append(0)    #3 - Null field
command.append(0x11) #4 - Reader id
command.append(1)    #5 - Reader id

for i in range(6,66):
    command.append(0) # - Parameters (null for the command 14)

command.append(0xf7) #66 - Crc LSB
command.append(0xde) #67 - Crc MSB
```

Figura 3 – A) Estrutura do comando que solicita as grandezas instantâneas do medidor, segundo a norma ABNT14522 B) Comando compatível com todos os medidores de energia testados e que responde com as grandezas instantâneas corretamente.

ativa e reativa são incrementados. Quando o contador regressivo chega a zero, a energia ativa e reativa podem ser calculadas (baseadas na quantidade de pulsos emitidos) e em seguida os valores dos contadores de pulsos são zerados.

O medidor repete a mensagem contendo o valor final de pulsos três vezes e o intervalo de demanda geralmente é de 15 minutos.

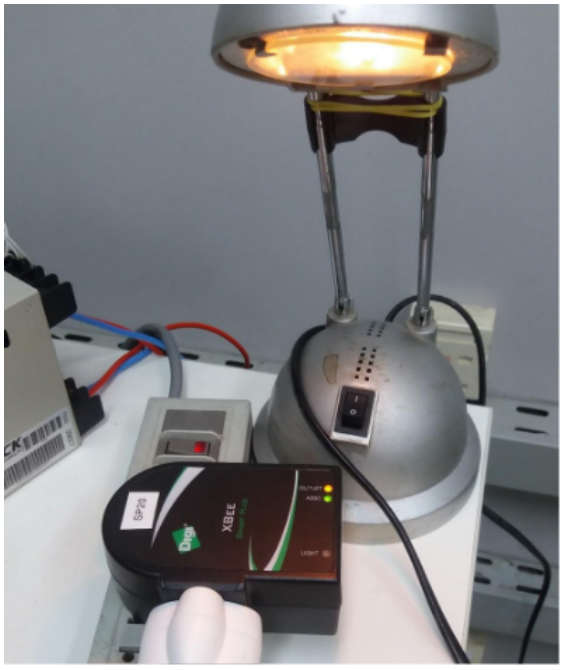

Figura 4 – *Smartplug* da fabricante Digi utilizado no estudo da comunicação Xbee.

2.7 XBEE

Módulos capazes de se enviar e receber mensagens através do protocolo sem fio ZigBee. A comunicação ZigBee (DIGI,) se caracteriza por baixa potência de operação, baixa taxa de transmissão de dados e baixo consumo de energia. ZigBee apresenta um perfil da tecnologia IEEE 802.15.4 operando na frequência de 2.4 Ghz e foi planejada para IoT.

O padrão técnico IEEE 802.15.4 define a operação de redes sem fios de baixa taxa de transmissão (LR-WPANs). Ele especifica a camada física e o controle de acesso à mídia (MAC) para LR-WPANs. Esta padrão é muito importante no contexto das soluções de comunicação IoT já que é a base para as especificações de várias tecnologias como Zigbee, ISA100.11a (usado na indústria), WirelessHART (usado na indústria), 6LoWPAN e Thread. Cada uma das tecnologias estende ainda mais o padrão, desenvolvendo as camadas superiores que não estão definidas no IEEE 802.15.4. Por exemplo, o 6LoWPAN define uma ligação para a versão IPv6 do Protocolo da Internet (IP) sobre os WPANs e é usado pelas camadas superiores, como o Thread.

Os medidores utilizados para o estudo dessa comunicação são *smartplugs* da marca DIGI, mostrados na figura 4. A comunicação com os mesmos foi possível utilizando a biblioteca Xbee (XBEE...,) (Python3) e as informações fornecidas pelo fabricante, contendo equações de calibração.

2.8 *MIDDLEWARES* IOT

Middlewares são camadas de software que possuem a função de trazer abstrações e facilitar o trabalho de desenvolvedores de software (PUDER; RÖMER; PILHOFER, 2011). Essas abstrações podem ser em diversos sentidos, como na comunicação entre redes de computadores, fornecendo APIs para manipular funcionalidades complexas e fornecendo transparências de muitos tipos.

Existem muitos tipos de *middlewares*, como o orientado a mensagens e baseado em tuplas. Essa variedade permite aos desenvolvedores selecionar o que melhor se adequa à arquitetura de sua aplicação.

Acompanhando a evolução, os *Middlewares* IoT também possuem a capacidade de abstrair ao desenvolvedor as complexidades envolvidas na manipulação de seus dispositivos.

São oferecidas funções como abstração de dispositivos em forma de serviços, armazenamento de medições, atuação e compartilhamento. As *features* de alguns *middlewares* IoT serão melhor descritas na seção de trabalhos relacionados.

2.9 ARQUITETURA EM MICROSSERVIÇOS

Padrão arquitetural onde ao invés de se ter um único módulo responsável por todas as funcionalidades, processamentos e tarefas do sistema, tem-se diversos módulos menores e com funcionalidades específicas. Esses módulos menores podem se comunicar e são responsáveis por apenas uma tarefa específica. É adequado ao *deployment* em ambientes Cloud e apresenta facilidades em termos de escalabilidade. Os microsserviços ainda podem ser implementados em diferentes linguagens de programação, escalados de maneiras diferentes, o que facilita o desenvolvimento entre diferentes equipes de software (Balalaie; Heydarnoori; Jamshidi, 2016).

2.10 APIS RESTFUL

Representational state transfer (REST) é um conjunto de regras e um padrão que podem ser utilizadas para criação de serviços web, de modo que a comunicação com, ou entre os mesmos aconteça mais facilmente. Serviços ou APIs que seguem esse padrão são chamados RESTful.

O padrão REST permite que, utilizando um conjunto de operações, recursos especificados textualmente pelos serviços sejam manipulados.

Recursos são identificados por *endpoints* (similares aos endereços), e respondem a essas operações com informações estruturadas em JSONs, Hypertext Markup Language (HTML) e Extensible Markup Language (XML). REST independe do protocolo utilizado, mas se tratando de HyperText Transfer Protocol (HTTP), as requisições são as mesmas envolvidas no protocolo, como GET e POST.

Todas as APIs envolvidas no projeto do IoTalker são RESTful.

3 TRABALHOS RELACIONADOS

A primeira atividade realizada para iniciar a pesquisa foi o levantamento dos trabalhos relacionados. O objetivo dessa etapa foi entender e encontrar as lacunas existentes, referentes ao tema da pesquisa. Assim, através de uma pesquisa empírica no Google Scholar, que acessa artigos de diferentes engenhos como: IEE, ACM, Springer, entre outras, foi possível encontrar trabalhos que envolvessem a integração de dispositivos IoT.

Nessa seção, será realizado um estudo sobre os trabalhos relacionados. A comparação com o IoTalker e influências no design de sua arquitetura serão abordados após o detalhamento da mesma.

Os trabalhos encontrados foram organizados em três categorias principais de acordo com suas características estruturais. Cada uma dessas categorias apresenta aspectos que servirão de base para o planejamento da arquitetura do IoTalker. Essas categorias são:

1. *Middlewares* IoT: são alguns trabalhos que atuam como uma camada de abstração para dispositivos IoT, integrando-os a outras plataformas. Embora o IoTalker não seja um sistema *middleware* completo, este opera com integração de dispositivos, tal como acontece nesses sistemas, havendo convergência de funcionalidades.

2. *Gateways* IoT: trabalhos que possuem características semelhantes como *gateways* extensíveis e geradores de código para comunicação de protocolos. Esses trabalhos são mais voltados diretamente para a conversão de protocolos e comunicação com dispositivos IoT. *Middlewares* IoT são soluções mais completas que implementam muitas funcionalidades além de apenas a conversão de protocolos, que é o trabalho realizado pelos *gateways*.

3. Aplicações proprietárias: aplicações desenvolvidas normalmente pelos próprios fabricantes do dispositivo IoT ou desenvolvedores do projeto. São aplicações normalmente fechadas para operar apenas com os dispositivos para os quais foram implementadas e normalmente oferecem poucas possibilidades de comunicação ou integração com outras aplicações (APIs). Isso dificulta seu uso em um contexto deferente de IoT. Desse modo, a revisão de literatura se focará nas duas categorias anteriores de trabalhos relacionados.

As subseções seguintes terão como foco listar e apresentar alguns dos trabalhos nas categorias de *middlewares* IoT e *gateways* IoT.

3.1 *MIDDLEWARES* IOT

Em (NGU et al., 2017a) é realizado um levantamento e classificação dos principais *middlewares* IoT, atentando para seus pontos fortes e fracos. Os autores os classificam em

três categorias:

- *Service-based*: *Middlewares* que necessitam de muitos recursos computacionais e que executam em Clouds ou máquinas situadas numa camada acima dos dispositivos. Não podem ser executados em dispositivos IoT com poucos recursos computacionais. Fornecem serviços mais básicos com poucas opções para facilitar a integração com outras aplicações. Os autores dão como exemplo de *middlewares* deste tipo o Hydra (Linksmart) (EISENHAUER; ROSENGREN; ANTOLIN, 2009) e Global Sensor Networks (GSN) (ABERER; HAUSWIRTH; SALEHI, 2006).

- *Cloud-based*: *Middlewares* compostos por serviços funcionais acessados normalmente por um conjunto de APIs. Podem ser utilizados para diversos fins, porém possuem as mesmas desvantagens de uma Cloud, como dependência do fornecimento do serviço, boa conexão com a Internet e cobranças por requisições e uso. Os autores dão como exemplo de *middlewares* deste tipo o Google Fit (GOOGLE...,), Xively (XIVELY,) e Paraimpu (PINTUS; CARBONI; PIRAS, 2012).

- *Actors-based*: *Middlewares* com uma estrutura mais versátil e leve. *Actors* podem estar presentes em todas as camadas arquiteturais, desde o dispositivo IoT até a aplicação. Os usuários podem manusear seus módulos e adicionar componentes com facilidade. Os autores dão como exemplo de *middlewares* deste tipo o Calvin (PERSSON; ANGELSMARK, 2015), Node-RED (NODERED,) e Ptolemy Acessor Host (PTOLEMY,).

O Hydra (Linksmart) (EISENHAUER; ROSENGREN; ANTOLIN, 2009) provê um serviço para atuar como abstração para diferentes tipos de dispositivos, facilitando sua integração em aplicações. Sua principal característica é a capacidade de visualizar os dispositivos como serviços. Suas capacidades são expressas em ontologias, facilitando buscas. O projeto é focado em três domínios de aplicação principais: automação residencial, saúde e agricultura.

O sistema de desenvolvimento SDK do Hydra exige muito conhecimento de programação, tornando sua utilização complicada para usuários finais ou como ponte para outras aplicações. Para atingir o objetivo de visualizar os dispositivos como serviços, faz-se necessário que os mesmos tenham capacidade de executar protocolos e processamentos mais pesados, tornando esse um processo complicado para dispositivos mais limitados.

Em (ABERER; HAUSWIRTH; SALEHI, 2006) os autores descrevem o Global Sensor Networks (GSN), um *middleware* IoT cuja principal *feature* é a abstração de dispositivos físicos em virtuais. Os usuários integram dispositivos IoT à plataforma especificando um arquivo de descrição XML. Esse XML contém detalhes sobre o tipo dos dados e como acessar o *stream* dos mesmos. Cada sensor é associado a um *wrapper*, que especifica os protocolos utilizados e o que fazer com os dados recebidos. Caso o *wrapper* para o tipo

de dispositivo / protocolo ainda não esteja disponível, o usuário deverá implementa-lo. Os dados e sensores podem ser acessados via APIs e serviços web. O GSN também conta com um banco de dados para o armazenamento de medições.

A tabela 1 elenca as principais funcionalidades oferecidas pelos *middlewares* IoT do tipo *service-based*, bem como suas desvantagens.

Middleware IoT	**Funcionalidades**	**Desvantagens**
Linksmart	Dispositivos como serviços, ontologias, domínios específicos	SDK complexa e necessidade de dispositivos robustos
GSN	Dispositivos virtuais, XML, Wrapper, APIs e armazenamento	Conceitos de programação Java e necessidade de dispositivos robustos

Tabela 1 – *Middlewares* IoT do tipo *service-based*.

O Google Fit (GOOGLE...,) é um *middleware* IoT que trabalha com dados esportivos de *fitness*. O sistema contém um banco onde esses dados são armazenados e expostos para o desenvolvimento de aplicações através de APIs. Esse *middleware* é voltado para o domínio de *fitness*.

Os dados são gerados pelos dispositivos dos usuários, como celulares e *smartwatches*. O Google Fit é compatível com dispositivos que utilizam Bluetooth Low Energy (BLE). Para integrar um dispositivo não previamente compatível são necessários conhecimentos não básicos de programação Java.

No *middleware* IoT Xively (XIVELY,), os dispositivos físicos são transformados em virtuais através da conexão com uma aplicação web. IoT Xively oferece um banco de dados de séries temporais escalável que armazena os dados e permite sua consulta com baixo tempo de resposta. A integração com dispositivos, especialmente os não suportados, requer conhecimentos avançados de programação. Não são fornecidas muitas opções para integração com outras plataformas.

O Paraimpu (PINTUS; CARBONI; PIRAS, 2012) oferece a abstração de dispositivos os categorizando em sensores e atuadores. Ambas as categorias são no formato de serviços, que podem ser interconectadas pelo desenvolvedor utilizando Javascript. São oferecidos serviços como banco de dados (é utilizado MongoDb) e balanceamento de carga. Uma das principais *features* do Paraimpu é a possibilidade de compartilhar dados e serviços com outras pessoas via redes sociais.

A tabela 2 reúne as principais funcionalidades oferecidas pelos *middlewares* IoT do tipo *cloud-based*, bem como suas desvantagens.

Calvin (PERSSON; ANGELSMARK, 2015) oferece um modelo arquitetural mais leve, adequando-se a execução tanto em dispositivos com poucos recursos quanto numa Cloud que oferece muito poder de processamento. Seu modelo tem como base módulos de software chamados *Actors*. Os *Actors* desempenham funções especificas e possuem *inputs* e *outputs*, que podem ser combinados, permitindo sua conexão.

Middleware IoT	Funcionalidades	Desvantagens
Google Fit	Dados esportivos, armazenamento, APIs, BLE	Integração utilizando Java
Xively	Aplicação web, Banco de séries temporais	Integração complexa, poucas opções para integrar com outras plataformas
Paraimpu	Sensores e atuadores, Javascript, MongoDB, compartilhamento em redes sociais	Recursos limitados na configuração de sensores e atuadores

Tabela 2 – *Middlewares* IoT do tipo *cloud-based*.

A biblioteca nativa do Calvin já conta com suporte a dispositivos e protocolos mais populares. Os desenvolvedores podem compor novos *Actors* em uma linguagem de programação própria, o CalvinScript, embora esse ainda não seja um processo simples. Novos *Actors* compostos podem ser adicionados à biblioteca da plataforma para serem utilizados por outros programadores. O middleware não apresenta interface gráfica.

O Node-RED (NODERED,) possui características similares. Baseado em node.js, tem arquitetura versátil para trabalhar em máquinas com pouco ou muito recurso computacional. Sua estrutura é baseada na composição e utilização de *Nodes*, que seriam análogos aos *Actors* do Calvin.

Os *Nodes* são módulos escritos em Javascript que abstraem a comunicação com um dispositivo IoT e que podem ser interconectados, compondo aplicações. Esse processo pode ser realizado via uma interface gráfica, fornecida pelo *middleware*. Para integração de novos dispositivos, faz-se necessário que hajam bibliotecas em node.js que permitam a comunicação, ou que os mesmos sejam módulos ou serviços acessíveis ao sistema.

Seguindo o padrão dos *middlewares* IoT *actors-based*, o Ptolemy (PTOLEMY,) abstrai os dispositivos e serviços em *Accessors*, expondo APIs Javascript para comunicação. As APIs são leves o suficiente para executar em máquinas sem muitos recursos.

O Ptolemy se diferencia do Node-RED e Calvin por possuir uma arquitetura orquestrada, por um módulo chamado *Director*. O Ptolemy possui uma interface gráfica e abriga um extenso repositório de *Accessors* à disposição dos usuários, minimizando as necessidades de integração e facilitando o desenvolvimento de aplicações.

A tabela 3 reúne as principais características e desvantagens dos *middlewares* IoT do tipo *actor-based*, bem como suas desvantagens.

Middleware IoT	Funcionalidades	Desvantagens
Calvin	Actors, CalvinScript, arquitetura versátil, biblioteca	Sem GUI
Node-RED	Nodes, arquitetura versátil, Node.js, GUI	Necessidade de suporte Node.js
Ptolemy	Accessors, javascript, Director, biblioteca, GUI	-

Tabela 3 – *Middlewares* IoT do tipo *actor-based*.

3.2 *GATEWAYS* IOT

Em (ZHU et al., 2010) os autores propõem um sistema em várias camadas para um cenário de *smart home*.

1. Camada de percepção: Camada onde se encontram presentes os dispositivos IoT e redes de sensores. Que processam e geram informações.

2. Camada de transmissão: Camada que faz a conversão dos protocolos dos dispositivos para a camada de aplicação (acima). Suporta protocolos como ZigBee e WirelessHART, e converte as mensagens para a *Internet* e Redes móveis. O módulo núcleo dessa camada é o *IoT Gateway*, foco do trabalho.

3. Camada de aplicação: Fornece algumas funcionalidades num nível de abstração maior, permitindo aos usuários controlarem e visualizarem seus equipamentos.

No trabalho (DESAI; SHETH; ANANTHARAM, 2015) os autores apresentam um *gateway* que aceita conexões de dispositivos nos protocolos MQTT, Extensible Messaging and Presence Protocol (XMPP) e Constrained Application Protocol (CoAP). As mensagens enviadas pelos dispositivos são convertidas em JSONs através de uma etapa de processamento para em seguida serem enviadas à serviços de Cloud ou outros *gateways*.

Em (GUOQIANG et al., 2013) os autores propõem um *gateway* IoT configurável de acordo com as necessidades do usuário. O *gateway* suporta algumas tecnologias de comunicação, como Zigbee e RS485. Possui *slots* onde o usuário especifica quais protocolos serão convertidos. Sua saída para o envio das mensagens processadas são *Ethernet*, redes móveis e RS485.

É realizado um estudo sobre a relação entre Cloud-IoT e uma proposta de um *gateway* IoT no trabalho (AAZAM; HUNG; HUH, 2014). Os autores propõem um *gateway* capaz de se comunicar com dispositivos IoT em diferentes protocolos como Zigbee e WirelessHART. O *gateway* possui a capacidade de processar e remodelar as mensagens antes de as enviarem a seu destino. Os autores também propõem que o *gateway* ofereça serviços adicionais, como armazenamento de medições.

O trabalho (KIM; CHOI; RHEE, 2015) apresenta a proposta de um *gateway* para o cenário de uma *smart home*. O *gateway* abstrai a heterogeneidade dos dispositivos integrados. Este se comunica com dispositivos MQTT, Devices Profile for Web Services (DPWS) e outros protocolos como Bluetooth e Zigbee. É montado um *testbed* para validação do projeto.

4 DESENVOLVIMENTO

Esse trabalho tem por objetivo lançar a proposta, bem como a versão inicial de uma plataforma de software que facilita o processo de comunicação com dispositivos heterogêneos. O IoTalker atua como um *gateway*, convertendo um protocolo de entrada em um protocolo de saída, com a adição de *features* extras condizentes com o contexto IoT. IoTalker pode ser visto como um *gateway* genérico e configurável, apto à interagir com vários protocolos e dispositivos diferentes.

Quando desejam integrar seus dispositivos IoT, os desenvolvedores precisam buscar referências, bibliotecas e informações relativas de como programar e fazer aquisição dos dados. Esse trabalho geralmente não é tarefa fácil, seja por fabricantes disponibilizarem poucas informações à respeito, APIs complexas (ou até mesmo ausência de APIs) ou protocolos sem bibliotecas de suporte.

Outra alternativa seria o uso dos já citados *middlewares* IoT. Ferramentas que vem ganhando popularidade nesse contexto. *Middlewares* atuam como ponte, conectando dispositivos à aplicações maiores ou particulares. Como já mencionado, desempenham um trabalho similar ao IoTalker, porém a maior dificuldade surge na hora de integrar dispositivos que não são suportados pelo *middleware*. Os *middlewares* oferecem então ferramentas para que o programador realize a integração manualmente, processo esse que pode ser complexo e potencialmente errôneo.

Tendo esse cenário como base, o IoTalker se destaca. A pluralidade de protocolos compatíveis, aliada a sua simplicidade de uso o tornam uma abordagem poderosa a ser utilizada por desenvolvedores que desejam integrar seus dispositivos e sensores IoT a outras plataformas, como mostrado na figura 5.

A metodologia de desenvolvimento do projeto foi estruturada nas seguintes etapas:

1. Seleção e comunicação com um conjunto de dispositivos com os quais o Iotalker será, inicialmente, compatível. Os dispositivos são os que foram listados no capítulo 2. Nessa etapa foram encontradas grandes dificuldades, pois alguns dos dispositivos listados, especialmente os medidores de energia do padrão ABNT14522, não possuem bibliotecas para facilitar a comunicação e apresentam descrições não muito claras dos protocolos e mensagens trocadas. Apesar das dificuldades, foi possível a comunicação com todos os dispositivos listados.

2. Planejamento e implementação da arquitetura de software da plataforma. Essa etapa será o foco desse capítulo como um todo. O modelo arquitetural escolhido, módulos componentes da plataforma, aspectos de implementação, exemplos, APIs e outros detalhes serão abordados nas subseções seguintes.

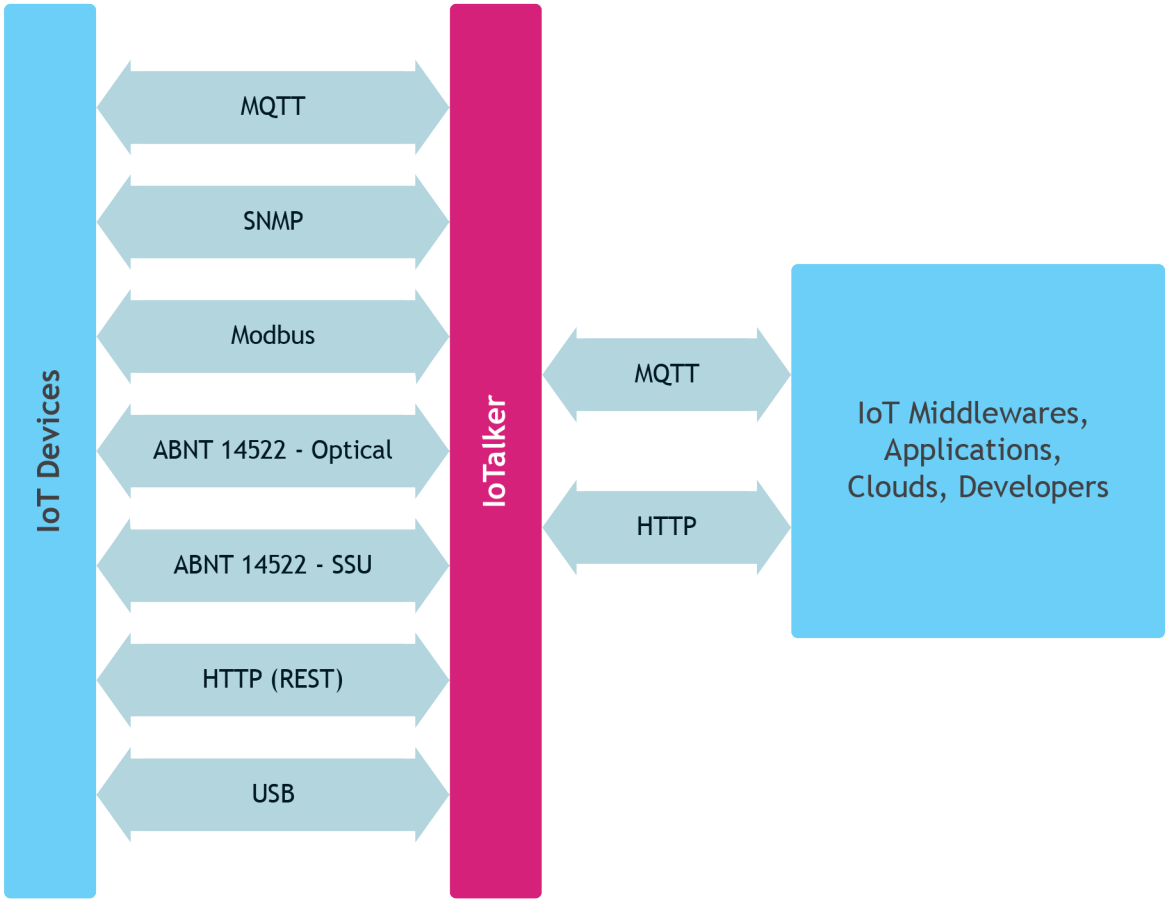

Figura 5 – Atuação do IoTalker como um *gateway* IoT suportando comunicação com dispositivos IoT e permitindo sua integração à outras plataformas e sistemas

3. Testes de validação. Finalizada a implementação, a plataforma será posta a prova com dois casos de uso que explorem suas funcionalidades, permitindo sua validação. Os casos de uso utilizados para validação serão abordados no capítulo 5.

Seu modo de funcionamento e operações é todo definido por arquivos de configuração simples. Listando corretamente os parâmetros, o desenvolvedor pode criar um servidor web HTTP que disponibiliza as últimas medições de seus *smartplugs* em pouco tempo.

A arquitetura de software do IoTalker foi implementada em microsserviços e é mostrada na figura 6. Cada microsserviço funciona de modo independente e possui propriedades de comunicação com os demais, processando informações e desempenhando funções únicas e específicas. O protocolo de comunicação entre os microsserviços é HTTP.

Tal escolha foi tomada devido ao fato das funcionalidades do IoTalker poderem ser divididas em tarefas granulares, além desse padrão arquitetural facilitar sua utilização e escalabilidade. Trata-se de uma plataforma de desenvolvimento, que pode ser ajustada, de acordo com as necessidades dos desenvolvedores. Alterar uma funcionalidade em um microsserviço é bem mais simples que em um software com milhares de linhas reunidas

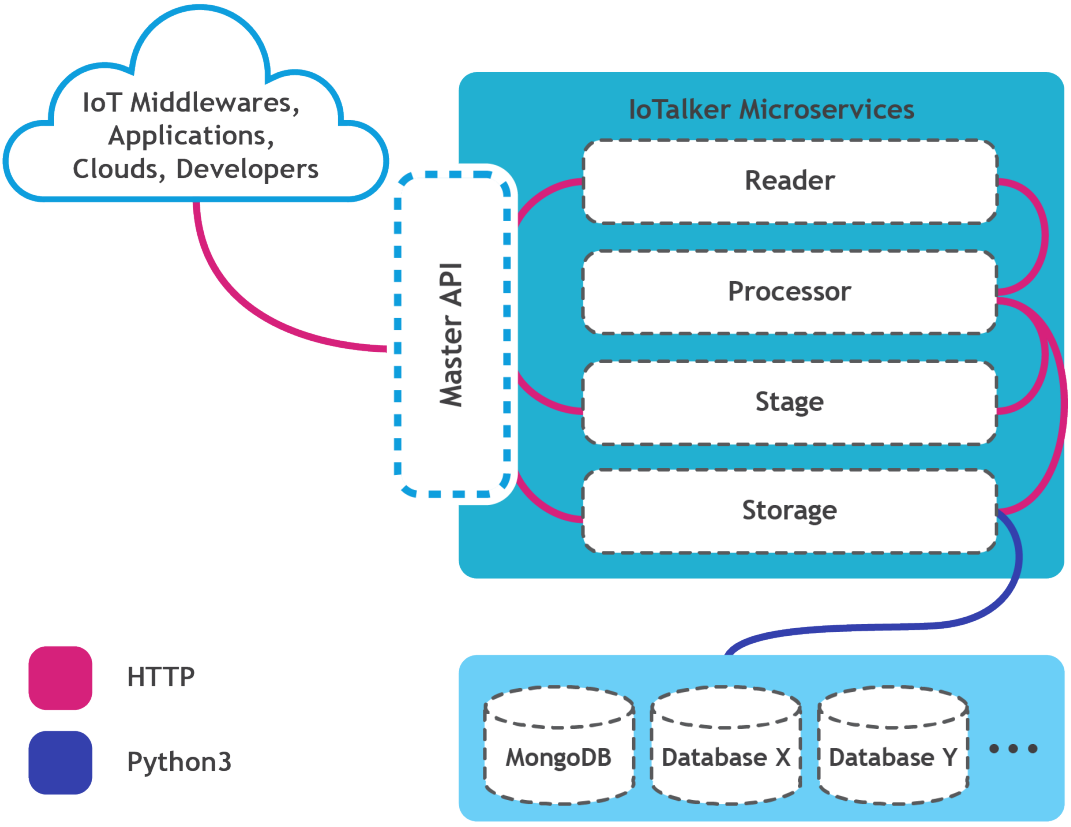

Figura 6 – Arquitetura de software do IoTalker em microsserviços. Suas funcionalidades e APIs serão descritas nas subseções posteriores

em poucos arquivos, seguindo uma arquitetura monolítica. Desde que as mensagens trocadas entre os microsserviços permaneçam uniformes, o desenvolvedor pode editar seu funcionamento da forma que preferir, e o sistema se manterá funcionando corretamente.

O IoTalker é composto por cinco microsserviços: *Reader*, *Processor*, *Stage*, *Storage* e *Master API*. As próximas cinco subseções posteriores os detalham, em termos de elementos, configurações, APIs individuais e formato de mensagens aceitas e enviadas. Em seguida são detalhados também os campos do arquivo de configuração.

4.1 *READER*

O *Reader* estabelece uma ponte de comunicação direta com o dispositivo em questão. Essa ponte é criada de acordo com as especificações do protocolo aceito pelo dispositivo. Caso o dispositivo esteja conectado via Universal Serial Bus (USB) ao computador, por exemplo, onde o IoTalker está executando, podem ser especificados parâmetros como *baudrate*, bits de paridade e porta USB de leitura. Se o dispositivo é acessado por meio de algum tipo de servidor web, também como exemplo, o IoTalker necessitará como parâmetros o endereço IP do servidor web, porta e autenticação. Os parâmetros informados serão relativos ao protocolo na qual o dispositivo aceita comunicação.

A tarefa desse microsserviço consiste em se comunicar com os dispositivos IoT, fazer a aquisição de seus dados / medições e traduzi-las em um formato comum. O microsserviço é composto pela combinação de dois elementos: *Driver(s)* e API.

Um *Driver* é uma abstração da comunicação com um dispositivo em forma de uma classe Python. Todos os *Drivers* possuem as mesmas assinaturas para alguns métodos pre-definidos, que são necessários para o funcionamento do sistema. São esses métodos:

- **__connect**: Método responsável por permitir a conexão propriamente dita com o dispositivo. Seja essa conexão via porta USB ou via um *broker* MQTT, por exemplo. Os parâmetros para o estabelecimento da conexão são obtidos através do arquivo de configuração e são dados como parâmetros do construtor da classe.

- **__read__message**: Dada que a conexão foi criada no método _connect, esse método permite e retorna a leitura de uma mensagem do dispositivo.

- **__generate__message**: Método auxiliar que permite a padronização da mensagem no formato comum para que a mesma possa ser interpretada pelo *Processor* corretamente. Esse método também adiciona o *timestamp* às medições. O método recebe três valores como parâmetro:

    1. *Value*: Valor numérico lido dos sensores e dispositivos IoT.
    2. *Metric code*: Código referente a grandeza física mensurada pela medição lida. O código é utilizado para indexar um dicionário com as possibilidades de grandezas e unidades. O conteúdo para cada chave é uma tupla onde o primeiro elemento é a descrição da grandeza e o segundo a sua unidade. Os desenvolvedores são livres para adicionar novas tuplas, até mesmo com diferentes unidades para uma mesma grandeza. O dicionário pode conter uma tupla *("length","m")* e outra *("length","km")*, por exemplo. Esse dicionário é utilizado por vários microsserviços além do *Reader*, e padroniza a referência à grandezas e unidades na plataforma, impedindo que sejam criadas medições com *strings* diferentes para a mesma variável (como "3m"ou "3 meters", por exemplo).
    3. *Meter id*: Identificador do dispositivo que originou a mensagem. Em alguns casos esse identificador é igual ao identificador do *Driver* (dado pelo arquivo de configuração, que será mostrado abaixo), mas em outros não. A necessidade da criação de um identificador surgiu devido a possibilidade de existirem mais de um sensor ou sub-sensor interagindo com o mesmo *Driver*, sob uma mesma conexão. Exemplificando, tem se o caso de medidores de energia trifásicos. A conexão com o medidor é única, via entrada óptica, mas cada uma das fases tem valores diferentes para corrente, potência e outras variáveis. As fases são visualizadas então como sub-sensores. Seus identificadores *Meter Id* serão diferentes do identificador do *Driver*. Enquanto o identificador do *Driver* seria

URL	GET
iotalker/reader	Todos os *Drivers* em operação
iotalker/reader/{driver_id}	Detalhes sobre um *Driver* específico
iotalker/reader/{driver_id}/measurements	Últimas medições lidas pelo *Driver* especificado

Tabela 4 – Urls e seus retornos esperados para o microsserviço *Reader*. Os identificadores são especificados pelo arquivo de configuração e devem ser únicos para cada *Driver*.

"Medidor_Energia", o das fases individuais seriam *"Medidor_Energia_faseA"*, *"Medidor_Energia_faseB"* e *"Medidor_Energia_faseC"* respectivamente.

Embora os *Drivers* possuam as mesmas assinaturas, a implementação varia de dispositivo para dispositivo, o que torna cada *Driver* único. Um *Driver* pode conter outros métodos auxiliares, mas necessita primordialmente dos três listados acima.

Os parâmetros para a configuração de um *Driver* variam de acordo com o tipo de protocolo com o qual ele lida. Se as mensagens são obtidas via MQTT, os parâmetros envolverão o endereço IP do *broker* MQTT e tópico para subscrição. Caso seja um medidor de energia, conectado via porta óptica USB, os parâmetros envolverão a porta USB e o *baudrate*.

Cada *Driver* possui um identificador único, definido pelo arquivo de configuração. Esse identificador é utilizado para permitir que mais de um *Driver* para um mesmo protocolo esteja em execução, desde que possuindo diferentes parâmetros de configuração. O desenvolvedor pode, por exemplo, configurar um *Driver* para ler medições de um medidor de energia na porta */dev/ttyUSB0* e outro na porta */dev/ttyUSB2*.

O *Reader* é o encapsulamento de um ou mais *Drivers* em um microsserviço, expondo uma API RESTful, permitindo a comunicação com outros microsserviços. A figura 7 mostra um exemplo de possível configuração do *Reader*. Seus endereços e retornos esperados são listados na tabela 4. Essa API é acessada pelo *Processor*, mas tratando-se o IoTalker de uma plataforma de desenvolvimento, as Uniform Resource Locator (url)s do *Reader* podem ser acessadas livremente por requisições externas ou planejadas pelo desenvolvedor.

A implementação inicial do IoTalker inclui suporte aos protocolos / normas / conexões: MQTT, SNMP, RS485, ABNT14522-Óptica, ABNT14522-SSU, Xbee, HTTP e USB comum. Sua implementação foi realizada de maneira a ser extensível, podendo abranger novos protocolos de maneira simples, através da criação de novos *Drivers*.

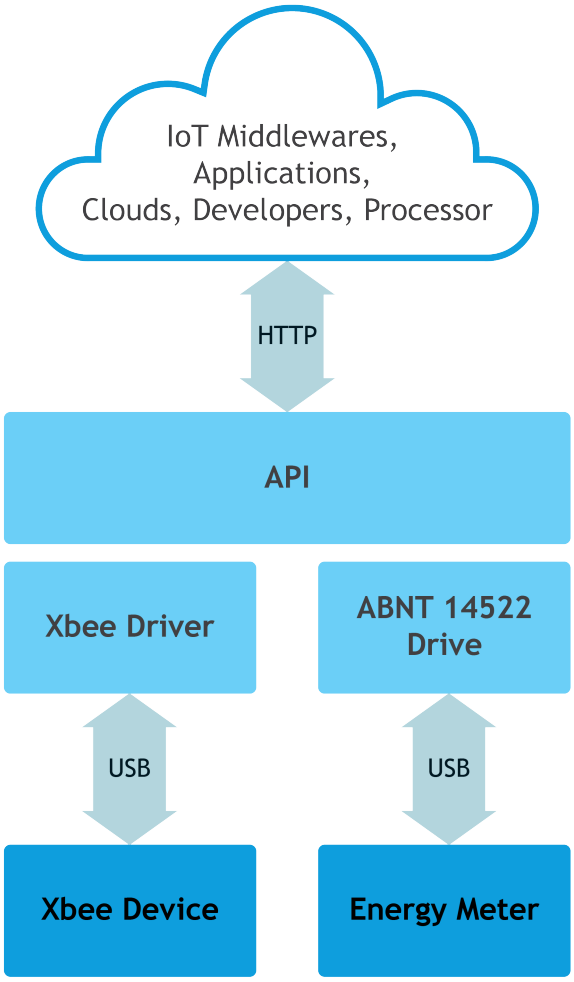

Figura 7 – Exemplo de arquitetura configurada do *Reader* para operar com 2 *Drivers*. Um primeiro exemplo para comunicação com dispositivos XBee e outro para comunicação com medidores de energia padronizados de acordo com a norma ABNT14522.

4.2 *PROCESSOR*

O microsserviço *Processor* atua como um orquestrador. Os microsserviços *Reader*, *Storage* e *Stage* apresentam um comportamento passivo, apenas respondendo e atendendo as requisições do *Processor*.

O *Processor* opera repetindo uma sequência de três passos infinitamente e não possui uma API individual como o *Reader*. Os três passos de operação são os seguintes:

1. Solicitação das medições de todos os *Drivers* em execução no microsserviço *Reader*.
2. Processamento de uma série de operações.
3. Envio dos resultados obtidos no passo 2 ao microsserviço *Stage*.

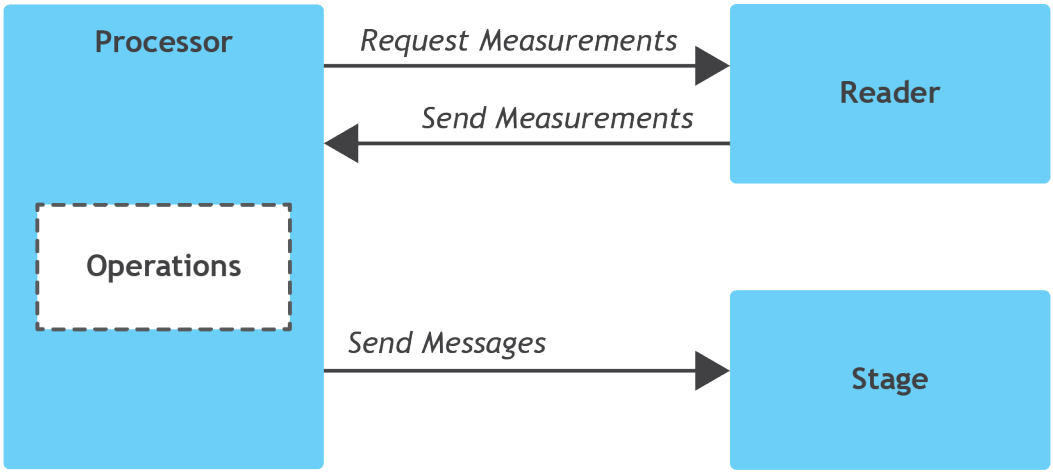

Figura 8 – Fluxo do funcionamento do *Processor* em relação ao *Reader* e *Stage*.

O fluxo de funcionamento é mostrado na figura 8. Os passos 1 e 3 se resumem basicamente à uma comunicação HTTP simples, respeitando-se os padrões estabelecidos. Já o passo 2 consiste na realização de uma série de operações envolvendo os valores obtidos pelo *Reader* e pelas próprias operações anteriores.

Essas operações permitem que os dados sejam manipulados e tratados, da forma que o desenvolvedor desejar. As manipulações realizadas pelo *Processor* são aplicadas apenas às informações reais obtidas. Toda complexidade envolvida na leitura da mesma é tratada pelo *Reader*.

A implementação inicial conta com o suporte a cinco tipo de operações:

- ***Calibration*:** alguns dispositivos necessitam de equações de calibração para obtenção de valores corretos. Esse procedimento é realizado pelo *Processor*, visto que a função do *Reader* é somente a extração da informação e suas configurações. Configurando corretamente, pode-se realizar quaisquer manipulação de caráter matemático envolvendo um ou mais dados lidos de um dispositivo IoT. A necessidade de criação dessa *feature* surgiu no estudo da comunicação com medidores de energia elétrica. Os valores mensurados pelos mesmos correspondem a apenas uma fração do valor real. Visto que dependendo da instalação elétrica montada, podem ter sido utilizados módulos transformadores de tensão e corrente. Logo, o valor lido pelo medidor necessita ser multiplicado por um valor constante, para se chegar a um valor real após a etapa de leitura (figura 9-A).

- ***Aggregation*:** através dessa *feature*, o usuário pode criar dispositivos virtuais por meio da criação e manipulação dos seus dispositivos físicos. Essa funcionalidade pode ser utilizada por usuários que desejam dados de não apenas um medidor, mas de um conjunto. Demos então um cenário que exemplifica essa situação. O usuário possui vários *smartplugs* instalados nos equipamentos eletrônicos do seu

quarto, como TV, som e luminária. Cada um desses *smartplugs* consegue mensurar os valores de consumo do equipamento eletrônico ao qual está conectado. O usuário porém, não tem interesse nos valores individuais, mas sim no todo. Ele deseja saber os valores de consumo total do seu quarto. Para tal, deve-se então ser criado um dispositivo virtual que agrega os dados mensurados por cada um dos *smartplugs* em um único valor, através de um somatório (figura 9-B).

- **Storage:** O IoTalker oferece a possibilidade de armazenar os dados e informações obtidas de seus dispositivos IoT em um banco de dados NoSQL (MongoDB). Embora essa funcionalidade seja oferecida pelo *Processor*, não é o mesmo que a provê. Ela é provida pelo microsserviço *Storage*, que será detalhado nas subseções posteriores. Essa funcionalidade é alcançada através da comunicação com esse microsserviço via requisição HTTP.

- **Notification:** Essa operação permite verificar um condicional (>,<,==) sobre um valor específico, e enviar uma notificação via e-mail caso o condicional seja verdadeiro. Essa *feature* foi pensada para usuários que estão interessados em monitorar o comportamento de seu sistema remotamente.

- **Viewer:** Operação que faz a comunicação com o microsserviço *Stage*. As medições dadas como *input* dessa operação serão enviadas a uma de suas *Doors*. Essa interligação será melhor detalhada na subseção posterior. A comunicação com o *Stage* é uma operação manipulável para dar ao programador a liberdade de escolher quais dados e em que momentos do fluxo de processamento esses dados serão mostrados pelo *Stage*. Essa operação conta ainda com a possibilidade de seleção de periodicidade. Definindo esse parâmetro, o desenvolvedor pode selecionar com que periodicidade o *Processor* enviará as informações processadas ao *Stage*. Essa propriedade é limitada pela taxa de atualização / envio de mensagens do dispositivo IoT. Caso essa frequência do dispositivo seja menor que a frequência configurada pelo usuário, o *Processor* manterá o valor da segunda, enviando mensagens repetidas, com diferença apenas no *timestamp*.

As operações são realizadas na ordem especificada e, embora essa versão do IoTalker ainda não conte com uma interface gráfica, as operações podem ser enxergadas como uma conexão de blocos, interligando seus *inputs* e *outputs*. A ordem, variáveis de *input* e *output* são todos especificados pelo arquivo de configuração.

O desenvolvedor pode calibrar cada um de seus medidores de energia multiplicando os valores de transformadores de corrente individuais para cada um deles, agregar seus valores em um consumo total, armazenar o resultado total, programar uma notificação caso esse consumo exceda o limite do mês passado e por fim direcionar as leituras instantâneas para um servidor HTTP através de uma operação *viewer*. Pode também fazer a etapa de

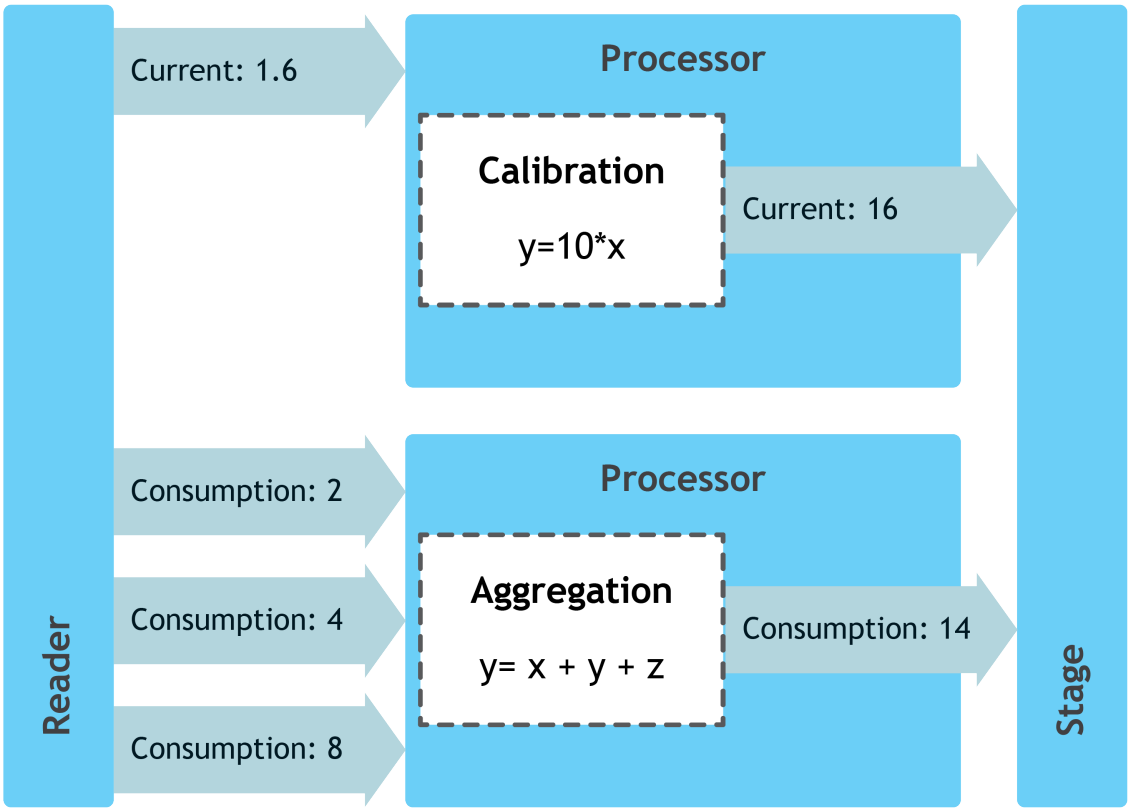

Figura 9 – A) Uma operação *calibration* sobre um valor recebido do *Reader* e B) Aplicação de uma operação *aggregation* sobre 3 valores recebidos do *Reader*.

calibração, em seguida armazenar, para em seguida agregar. O desenvolvedor é livre para manipular as operações da forma que desejar.

A arquitetura de software do microsserviço foi implementada de modo que torne-se fácil adicionar novas operações ao sistema, mantendo sua característica expansível.

4.3 *STAGE*

O microsserviço *Stage* tem a função de expor ou enviar para algum destino / destinatário as informações processadas pelo *Processor* através de algum protocolo. A ideia é que esses dados sejam disponibilizados de maneira estruturada e simplificada para o desenvolvedor ou outras aplicações.

Os microsserviços *Reader* e *Stage* são procedimentos inversos de desencapsulamento e encapsulamento. Enquanto o *Reader* executa vários *Drivers* que interagem com cada um ou mais grupos de dispositivos IoT, o *Stage* executa *Doors* que recebem dados e os disponibilizam através de um protocolo específico.

O *Stage* encapsula uma ou mais *Doors* e expõe uma API RESTful. A primeira implementação possui suporte a duas *Doors*: MQTT e HTTP. Os endereços bem como métodos de acesso à API do *Stage* são mostradas na tabela 5.

URL	GET	POST
iotalker/stage	Todos as Doors em operação	-
iotalker/stage/{door_id}	Detalhes sobre uma Door específica	-
iotalker/stage/{door_id}/measurements	-	Enviar uma ou mais medições para uma Door

Tabela 5 – Urls e seus retornos esperados para o microsserviço *Stage*. Os identificadores são especificados pelo arquivo de configuração e devem ser únicos para cada *Door*.

Sendo *Reader* e *Stage* análogos, uma *Door* é o análogo de um *Driver*. Trata-se de uma abstração da comunicação através de um protocolo em forma de uma classe Python. As *Doors* também possuem obrigatoriamente alguns métodos específicos que devem possuir a mesma assinatura para um funcionamento correto do sistema. São eles:

- **__connect**: É um método responsável por permitir a conexão propriamente dita através do protocolo. No caso de uma *Door* HTTP, esse método inicia a execução de um servidor web HTTP já apto a receber requisições. Tratando-se de uma *Door* MQTT, esse método efetua a conexão com o *broker* MQTT. A função do método _connect é criar uma caminho estável para efetuar envio ou recebimento de mensagens. Tanto no *Driver* quanto na *Door*. Os parâmetros para o estabelecimento da conexão também são obtidos através do arquivo de configuração e são dados como parâmetros do construtor da classe.

- **__send__message**: Dada que a conexão foi criada no método _connect, esse método permite o envio da informação através da conexão criada.

No *Stage* não há necessidade da implementação do método _generate_message pois a informação que chega ao *Processor* já se encontra estruturada, ao ser processada pelo *Reader*.

A configuração de uma *Door* varia de acordo com seu tipo. Se o protocolo for HTTP seus parâmetros de configuração serão url de acesso e porta. Caso MQTT seja o protocolo selecionado, os parâmetros envolverão o endereço IP do *broker* MQTT e tópico onde das mensagens serão publicadas.

Apesar do suporte à apenas dois protocolos, o desenvolvedor pode ainda configurar mais de uma *Door* para o mesmo protocolo (como é feito no *Reader*), utilizando um identificador e parâmetros diferentes. A figura 10 mostra um esquema da arquitetura do *Stage* operando com três *Doors*. Uma para HTTP e duas para MQTT, sendo as últimas especificando *brokers* MQTT diferentes para onde as mensagens serão publicadas.

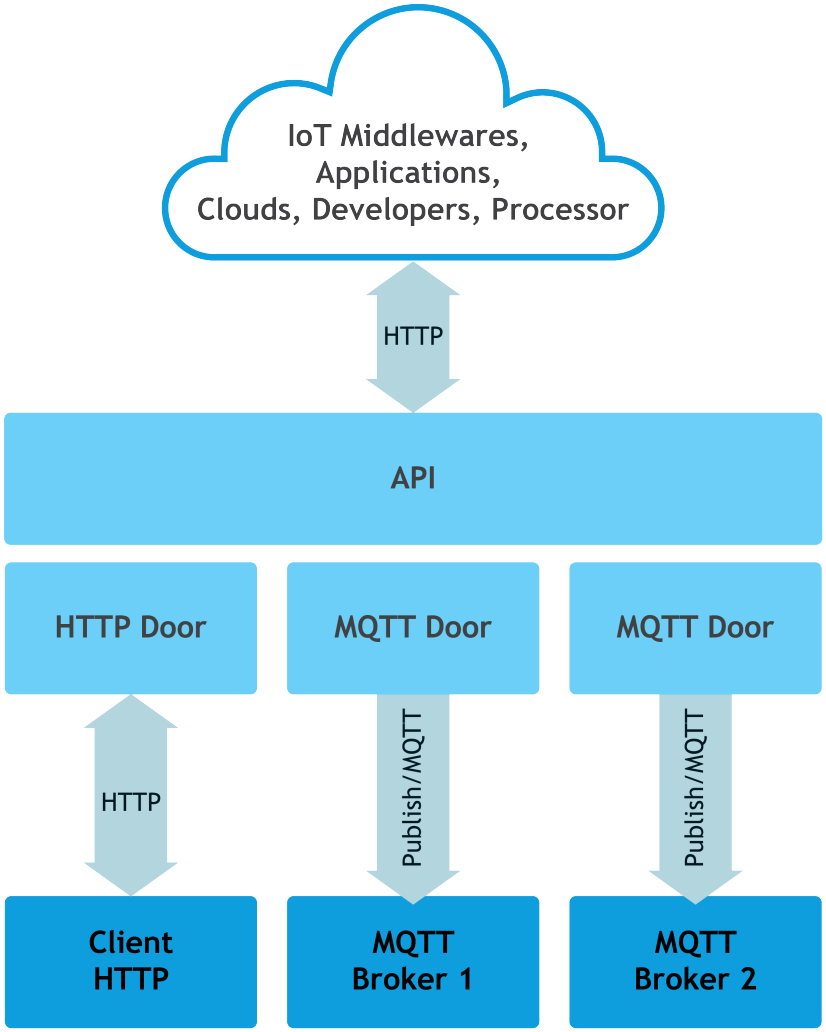

Figura 10 – Exemplo de configuração do *Stage* para utilizar três de suas *Doors*.

Quando selecionada a *Door* HTTP, o *Stage* iniciará a execução de um servidor web HTTP que disponibilizará apenas a última medição recebida do *Processor* via requisição RESTful do tipo GET. Dessa maneira pode ser acessado por qualquer aplicativo em qualquer linguagem que disponibilize requisições HTTP ou até mesmo via navegador. Se a *Door* MQTT for selecionada, o *Stage* publicará as medições no tópico e *broker* especificados.

Assim como o *Reader*, o *Stage* foi planejado para ser facilmente expansível, podendo ser adicionados novos protocolos em versões posteriores.

4.4 STORAGE

Esse microsserviço é composto por uma implementação baseada em MongoDB combinada com uma API RESTful. O *Storage* permite o armazenamento das medições dos dispositivos IoT, oferecendo as operações básicas de inserção, remoção, edição e consulta. Os

URL	GET	POST	PUT	DELETE
iotalker/storage/devices	Todos os dispositivos que possuem medições armazenadas.	Criação de um novo dispositivo		
iotalker/storage/devices/ {device_id}	Detalhes sobre um dispositivo específico que possui medições armazenadas.		Edição de um dispositivo específico	Remoção de um dispositivo específico e todas suas medições
iotalker/storage/devices/ {device_id}/measurements	Todas as medições de um dispositivo específico.	Adicionar medições a um dispositivo		

Tabela 6 – Urls e seus retornos esperados para o microsserviço *Storage*.

endereços que dão acesso a esses serviços são mostradas na tabela 6.

O *Storage* cria no banco de dados uma *collection* chamada *devices* que armazena documentos de cada um dos dispositivos registrados no sistema.

Quando a primeira medição do dispositivo é enviada através de uma requisição RESTful do tipo POST, o *Processor* verifica a existência do mesmo. Caso o dispositivo não exista, o *Processor* se encarrega de enviar uma requisição para a sua criação, para só então começar a enviar as medições.

Em termos de arquitetura interna, foi utilizada uma estratégia de armazenamento de documentos por dia, chamados *daily docs*. Os valores são armazenados dentro de listas indexadas pelos valores de hora e minuto. Os *daily docs* não são armazenadas na *collection devices* juntamente com os documentos com informações dos dispositivos. As *collections* que os armazenam são individuais para dispositivo e variável medida, sendo nomeadas pela concatenação entre *measurements*, identificador do dispositivo e código da variável, sendo esse último, o mesmo código utilizado pelo *Reader*. Por exemplo, os *daily docs* para o dispositivo *dev1* para medir a variável com código 2 serão armazenados na *collection "measurements_dev1_2"*.

Para exemplificar o passo a passo do processo de armazenamento de medições nos documentos, consideremos uma medição onde um dispositivo IoT mediu uma tensão de 223 Volts no *timestamp "2018-07-16 09:34"*. No momento de armazenar essa medição, o *Storage* consultará a existência de um documento onde o campo *date* possua *"2018-07-16"* como valor. Caso esse documento ainda não exista, ele será criado, com o formato

expresso na figura 11. Se o documento já existir, ele será recuperado e apenas atualizado com a inserção da nova medição.

A hora e minuto da medição criada para o dia *"2018-07-16"* são respectivamente *"09"* e *"34"*. Então o valor da medição, 223, será inserido na lista que é valor para as chaves *"09"* e *"34"*. Efetuada a operação de inserção do valor na lista, os valores das chaves *sum* e *counter* são atualizados. Tanto os aninhados na chave da hora *"09"* quanto os externos ao documento, como mostrado na figura 12. Os valores armazenados nessas chaves são responsáveis por otimizar o tempo de resposta em casos de consulta dos valores agregados de soma e média, para as granularidades temporais de hora e dia.

Considerando a arquitetura implementada, o *Storage* é uma opção pronta para o desenvolvedor que deseja utilizar o sistema de imediato, mas não a única. É possível sua substituição com facilidade por outros sistemas e arquiteturas de banco de dados, desde que respeitadas as APIs e formato de mensagens trocadas.

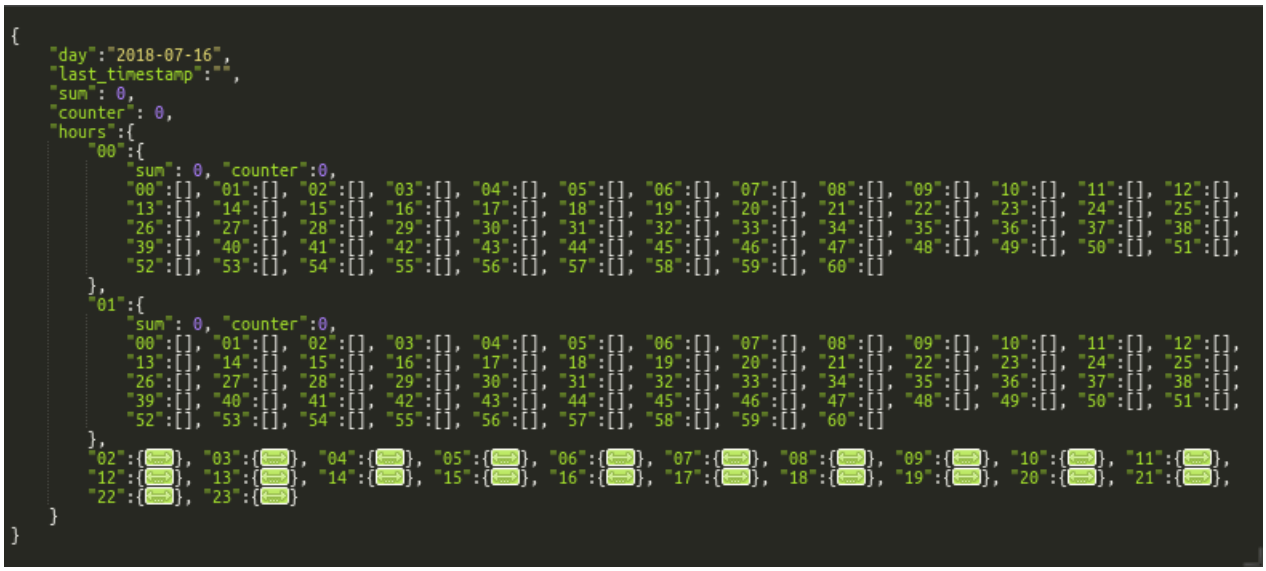

Figura 11 – Formato do documento criado na *collection* para armazenar as medições de um dispositivo. Os valores ocultos para as chaves são idênticos ao conteúdo das chaves "00"e "01".

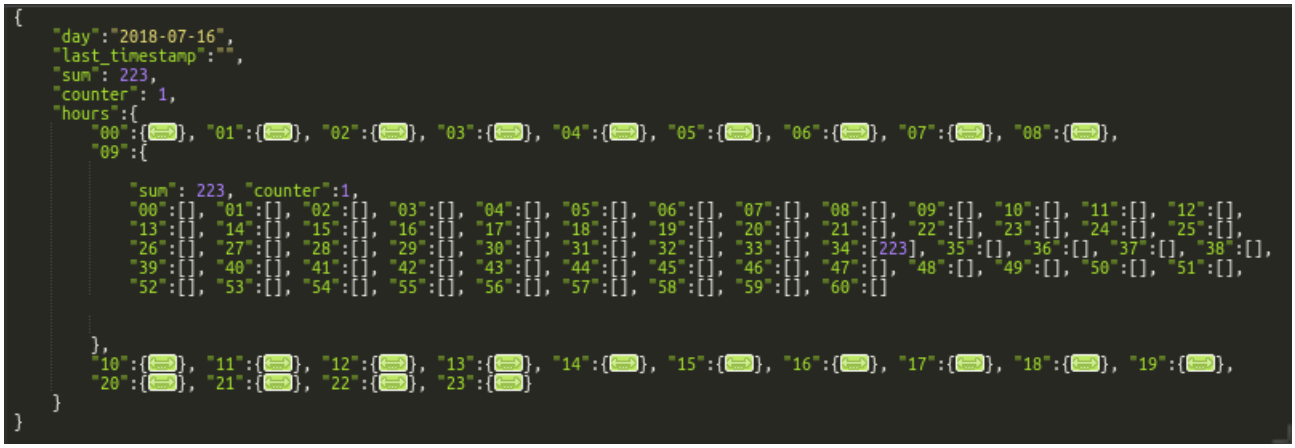

Figura 12 – Estrutura do *daily doc* após a inserção da medição 223 cujo *timestamp* é *"2018-07-16 09:34"*. O valor 223 é armazenado na lista indexada pelas chaves "09"e "34". O valor contido na chave *sum* (interna e externa) é incrementado em 223 e o valor contido na chave *counter* é incrementado em 1.

4.5 MASTER API

O IoTalker é um projeto estruturado em uma arquitetura de microsserviços. Cada um dos microsserviços pode ou não estar executando na mesma máquina e em diferentes portas.

De maneira a facilitar o trabalho do desenvolvedor na utilização da plataforma, foi adicionado um microsserviço com a função de fornecer transparência de localização ao sistema. Esse microsserviço é o *Master API*.

O *Master API* funciona como um roteador ou direcionador de requisições. Esse microsserviço conhece as localizações de cada um dos outros microsserviços bem como suas portas de acesso e os repassa as requisições realizadas, retornando suas respectivas respostas. Ao invés de criar requisições para endereços diferentes, o desenvolvedor precisa enviá-las apenas ao *Master API*, como mostrado na figura 13.

As urls aceitas pelo *Master API* são as mesmas dos outros microsserviços.

O escopo desse trabalho envolve apenas as etapas necessárias para a implementação do IoTalker bem como sua aplicação em alguns cenários com fins de validação. Avaliações de desempenho serão listadas como trabalhos futuros, porém por meio de observação imediata, pode-se cogitar que o *Master API* pode vir a se tornar um gargalo no funcionamento do sistema em um cenário de larga escala.

A proposta da arquitetura em microsserviços contribui para evitar esse tipo de problema. Não apenas no *Master API* mas em todos os outros microsserviços. O desenvolvedor pode executar várias unidades de cada microsserviço, provendo escalabilidade para suportar uma grande carga de requisições de outras aplicações ou mesmo dos próprios microsserviços. Essa é uma das principais vantagens de uma arquitetura seguindo esse padrão.

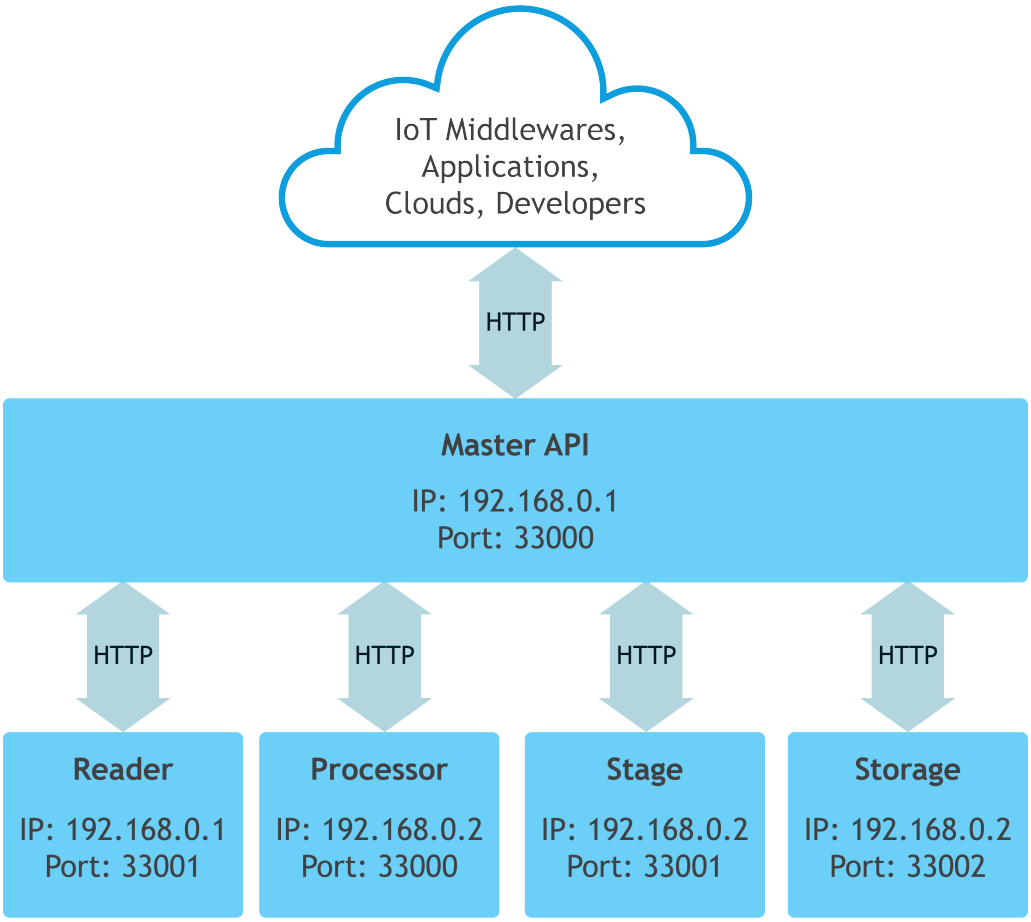

Figura 13 – *Master API* abstraindo a localização física dos microsserviços espalhados em 2 *hosts* diferentes.

4.6 CONFIGURAÇÃO

Como observado ao longo da descrição dos microsserviços e suas arquiteturas, existem muitos aspectos e definições a serem configuradas para seu correto funcionamento. A descrição da estrutura do arquivo de configuração será detalhada nessa subseção.

Na hierarquia de diretórios do IoTalker, os arquivos de configuração estão localizados ou devem ser posicionados na pasta *configuration* e são dois: *config.py* e *index.py*

O arquivo *config.py* especifica:

- *Drivers* que executarão no *Reader*.
- Operações que serão executadas no *Processor*, bem como sua ordem.
- *Doors* que executarão no *Stage*.
- *Hosts* e portas dos microsserviços.

A configuração dos microsserviços se dá por meio da especificação de um JSON chamado *services*, com o formato mostrado na figura 14-A. As chaves para esse JSON são os

nomes dos microsserviços *Reader*, *Processor*, *Stage*, *Storage* e *Master API*. O Valor para cada uma dessas chaves é um JSON aninhado, que possui as chaves: *url* e *port*. O primeiro especifica a localização física do *host* onde o microsserviço executará, e o segundo a porta. O usuário pode não especificar a chave e valor para *url*. Nesse caso o valor *default* será o *localhost* (127.0.0.1). A especificação da porta é obrigatória.

Através dessa configuração em específico, o *Master API* sabe a localização dos outros microsserviços, de modo a rotear as requisições corretamente.

```
services = {
    "Reader":       {
        "url":"http://127.0.0.1",
        "port": 33002
    },
    "Processor":    {
        "url":"http://127.0.0.1",
        "port": 33003,
    },
    "Stage":        {
        "url":"http://127.0.0.1",
        "port": 33004
    },
    "Storage":      {
        "url":"http://127.0.0.1",
        "port": 33005
    },
    "API":          {
        "url":"http://127.0.0.1",
        "port": 33000
    }
}
```

```
drivers = [
    {
        "id": "Medidor_Nansen",
        "type": "opticalAbnt14522",
        "parameters":{
            "usb":"/dev/ttyUSB0",
            "baudrate": 9600
        }
    },
    {
        "id": "Medidor_Eletra",
        "type": "opticalAbnt14522",
        "parameters": {
            "usb": "/dev/ttyUSB1",
            "baudrate": 9600
        }
    }
]
```

Figura 14 – Exemplo de configuração dos microsserviços através da especificação do JSON A) *services* e B) *drivers*.

A configuração dos *Drivers* se dá por meio da especificação de uma lista com o mesmo nome no arquivo de configuração. Os elementos dessa lista são JSONs que descrevem cada um dos *Drivers* em execução. Cada um desses JSONs possuem obrigatoriamente as três chaves:

- *Id*: Identificador único que será atribuído ao *Driver* e utilizado pelo *Processor* para utilizar os valores corretos nas operações. Caso hajam mais de um sensor conectado ao IoTalker pelo mesmo *Driver*, ou hajam sub-sensores, o seu identificador será uma concatenação do identificador original com um específico e particular do sensor ou sub-sensor, como já mencionado anteriormente.

- *Type*: Identifica o tipo do *Driver*. Essa definição ficará mais clara no detalhamento do segundo arquivo de configuração, *index.py*. O valor desse campo define exatamente qual código fonte de *Driver* será utilizado no *Reader*. Se é um *Driver* ABNT14522, um *Driver* SNMP ou mesmo um *Driver* USB.

- *Parameters*: Corresponde a um JSON aninhado que encapsula os parâmetros para uma configuração correta do *Driver*. Esta informação pode envolver endereço IP,

baudrate, porta USB e autenticação entre outros parâmetros. O atributos variam de *Driver* para *Driver*.

A figura 14-B mostra um exemplo de configuração de *Drivers* para leitura das medições de dois medidores de energia. Embora os dois *Drivers* sejam do mesmo tipo, seus identificadores devem ser diferentes. As portas USB conectadas também diferem de um medidor para outro.

A configuração das operações também é uma lista de JSONs. Cada um dos elementos especifica uma operação. A estrutura de dados escolhida é de lista, pois a ordem em que as operações são executadas será determinada por sua ordem na lista. As chaves dos JSONs que especificam uma operação, dentro da lista, são os seguintes:

- *Type*: O valor desse campo determina qual o tipo da operação realizada. A operação pode ser qualquer uma das 5 citadas.

- *Parameters*: Especifica alguns parâmetros para serem utilizados nas operações. Variam de uma operação para a outra. Os parâmetros por operação são descritos na tabela 7.

- *Input*: Quais valores servem de *input* para a operação. Na operação *calibration*, o *input* é utilizado na equação para obter os *outputs*. Já nas operações *aggregation* os *inputs* são somados, por exemplo. O valor para o campo *input* pode ser uma tupla, uma lista de tuplas, uma tripla ou uma lista de triplas. Essas tuplas tem como seu primeiro elemento um identificador e o segundo elemento o código da grandeza mensurada, que serve de chave para uma dos dicionários do arquivo *index.py*. O primeiro pode ser o identificador de um *Driver*, diretamente, ou mesmo um *output* de outra operação. No caso das triplas, os dois primeiros elementos se juntam para formar esse identificador, mas numa versão composta. As triplas são utilizadas principalmente no caso da existência de mais de um sensor conectado ao *Driver*, como no exemplo citado dos medidores de energia.

- *Output*: Determina qual é o valor de *output* da operação. Nas operações de *calibration* e *aggregation*, o *output* corresponde ao valor resultante após a aplicação das funções matemáticas. O formato do *output* também é uma tupla, mantendo a padronização com formato dos *inputs*. Na operação *viewer*, o campo de *output* deve conter o identificador de uma das *Doors* configuradas no *Stage*, especificando que as medições dadas como *input* serão enviadas à uma *Door* em específico. As operações *storage* e *notification* não possuem *output*, logo seu valor deve ser definido como vazio.

Exemplificando, consideremos a criação de operações para a configuração da leitura dos dados de um medidor de energia. A figura 15-A mostra um exemplo de configuração das operações. De acordo com essas definições, o *Processor* agirá da seguinte forma:

Operação	Parâmetros	Descrição
Calibration	equation	Descreve a equação de calibração do sensor. Tendo X como input e Y como output
Aggregation	aggregation_type	Descreve o tipo da agregação de medições
Notification	boolean_expression	Expressão que, caso seja verdadeira, o e-mail será enviado
	e-mail	E-mail destinatário
Viewer	timer	Frequência de atualização da *Door*

Tabela 7 – Parâmetros de configuração por tipo de operação no *Processor*. A operação *storage* não possui parâmetros de configuração.

1. Calibrar os dados da fase A do *Medidor Nansen* através da equação $y = 5*4x$, nomeando o valor calibrado como *"Calibrated PhaseA"*.

2. Agregar os valores da fase A calibrada com os valores lidos das fases B e C do mesmo medidor, nomeando o valor agregado como *"Consumption"*.

3. Armazenar os valores calculados para *Consumption*.

4. Verificar se esse valor excede 40000, em uma operação *Notification*. Caso exceda, um e-mail de notificação será enviado à *admin@iotalker.com*.

5. Encaminhar os valores calculados de *Consumption* para a *Door "webserver http"*, com uma taxa de atualização de 5 segundos.

O último componente do arquivo de configuração é referente à descrição das *Doors*, funcionando de modo semelhante ao processo de descrição dos *Drivers*.

Essa descrição define como cada uma das *Doors* vai operar. Trata-se de uma lista onde cada elemento é um JSON que detalha uma *Door*. Para cada uma são definidos os valores das chaves *id*, *type*, *parameters* e *input*. Os três primeiros possuem o mesmo comportamento e função que têm nos *Drivers*. A diferença vem na existência da chave *input*, que é inexistente nos *Drivers*. Seu comportamento é o mesmo do campo *input* nas operações do *Processor*. A figura 15-B mostra um exemplo da descrição das *Doors*.

Essas definições fecham e cobrem todas as configurações possíveis para o arquivo *config.py*. Na mesma pasta, existe um segundo arquivo de configuração: o *index.py*.

```
operations = [
    {
        "type":"calibration",
        "parameters": {"equation": "y = 5*4x"},
        "input": ("Medidor_Nansen", "phaseA", 1),
        "output": ("Calibrated phaseA", 1),
    },
    {
        "type": "aggregation",
        "parameters": {"aggregation_type": "sum"},
        "input": [("Medidor_Nansen", "phaseB", 1),
                  ("Medidor_Nansen", "phaseC", 1),
                  ("Calibrated phaseA", 1)],
        "output": ("Consumption", 1)
    },
    {
        "type": "storage",
        "parameters": {},
        "input": [("Consumption", 1)],
        "output": ""
    },
    {
        "type": "email-notifier",
        "parameters": {
            "boolean_expression": "x>40000",
            "email": "admin@iotalker.com"
        },
        "input": [("Consumption", 1)],
        "output": ""
    },
    {
        "type": "viewer",
        "parameters":{"timer":5},
        "input":[("Consumption",1)],
        "output": "webserver_http"
    }
]
```

```
doors = [
    {
        "id": "webserver http",
        "type": "http",
        "parameters": {
            "url": '/iotalker/measurements',
            "port": 33010
        },
        "input": [('Consumption',1)]
    },
    {
        "id": "broker mqtt",
        "type": "mqtt",
        "parameters": {
            "topic": '/iotalker/measurements',
            "broker_ip": "192.168.0.193",
            "port": 33011
        },
        "input": [('Consumption', 1)]
    }
]
```

Figura 15 – A) Exemplo de configuração das operações a serem realizadas no *Processor* e B) Exemplo de configuração das *Doors* no *Stage*.

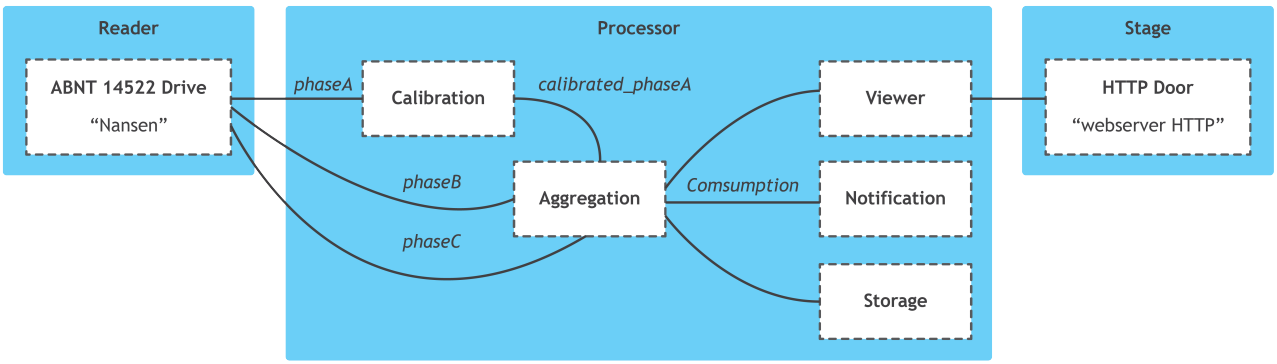

Figura 16 – Visualização em diagrama de blocos do funcionamento do IoTalker para um cenário genérico.

Não há a necessidade de alterar seu conteúdo caso o desenvolvedor deseje utilizar apenas as *features* já oferecidas pelo IoTalker. O *index.py* se divide em três estruturas de dados:

- *drivers_index* e *doors_index*: Estrutura de dados que indexa chaves aos seus respec-

tivos arquivos. Os valores inseridos no campo *type* durante a definição dos *Drivers* e *Doors* são colocados como entrada nessas estruturas para mapear o arquivo e classe corretos. Os arquivos com a real implementação dos *Drivers* e *Doors* encontram-se dentro da pasta do *Reader* e *Stage* no projeto, respectivamente.

- *variables_index*: JSON de tuplas onde vários microsserviços mapeiam o *metric code* tendo acesso à descrição e unidade da métrica em questão, mantendo sua padronização.

Embora ainda não conte com uma interface gráfica, os *Drivers*, operações e *Doors* podem ser enxergados como blocos que se interconectam para compor o funcionamento do IoTalker, como mostrado na figura 16. De acordo com a configuração definida pelos blocos na figura 16, o IoTalker irá operar da seguinte forma:

1. O *Reader* executará com apenas um *Driver*, com identificador *"Nansen"*, do tipo ABNT14522. Visto que os medidores de energia desse padrão são trifásicos, cada uma das fases (A,B e C) podem ser visualizadas como dispositivos individuais.

2. A fase A do medidor passa por uma operação de calibração. A operação tem como *output* o valor calibrado da fase A.

3. As fases B e C e o valor calibrado da fase A são dados como *inputs* para uma operação de agregação, que condensa os valores em apenas um *output*: *"Consumption"*.

4. *"Consumption"* é dado como entrada em uma operação *storage*, permitindo seu armazenamento no banco de dados do IoTalker.

5. *"Consumption"* é dado como entrada em uma operação *notification*, permitindo que o usuário receba uma notificação por e-mail caso o valor exceda um limite estimado.

6. *"Consumption"* é dado como entrada em uma operação *viewer*, permitindo que seu valor seja enviado para a *Door "webserver HTTP"*.

7. O *Stage* é configurado para operar com apenas uma *Door*, com identificador *"webserver HTTP"*. Concluindo a configuração da plataforma.

A utilização dos dois arquivos de configuração já permitem o funcionamento correto de todos os microsserviços que compõem o IoTalker, mas o desenvolvedor é livre para modificá-los à sua necessidade visto que trata-se de uma plataforma de desenvolvimento. O projeto conta com alguns exemplos de *config.py*, envolvendo medidores fictícios.

A combinação entre os microsserviços operam para fornecer ao desenvolvedor o poder de gerenciar o monitoramento de seus dispositivos IoT apenas escrevendo um arquivo de configuração simples, tendo que lançar mão de programação apenas no caso do protocolo não ser suportado.

Cada um dos microsserviços possui uma API RESTful que permite sua utilização individual e integrada a outras aplicações e projetos.

Tendo sido explanada a arquitetura e funcionamento dos microsserviços, o IoTalker será aplicado em dois cenários de teste no capítulo seguinte.

5 RESULTADOS

De maneira a validar o funcionamento e proposta do IoTalker, o mesmo foi utilizado em dois cenários, envolvendo situações reais dentro da Universidade Federal de Pernambuco (UFPE).

O primeiro caso engloba a implementação de um *gateway* capaz de mensurar a potência ativa total de todos os sistemas elétricos do prédio DINE, localizado próximo a entrada da UFPE. O segundo caso consiste na configuração do IoTalker para monitorar um aparelho *nobreak*, informando ao seu administrador a carga total do equipamento, bem como o status da bateria, provendo a possibilidade de detectar faltas de energia.

Cada uma das duas subseções seguintes descreverá em detalhes as etapas de configuração e execução do IoTalker, bem como os resultados obtidos ao longo de sua utilização. Em seguida é apresentada uma comparação com os trabalhos relacionados encontrados na literatura.

5.1 CENÁRIO 1

Foram instalados no prédio do DINE medidores de energia com entrada óptica, como mostrado na figura 17-A. Os medidores possuem os seguintes identificadores de fábrica, e estão conectados aos seguintes sistemas elétricos do edifício:

- Medidor 1406089: Realizando medições do sistema elétrico emergencial do edifício.
- Medidor 1406396: Realizando medições do sistema elétrico não emergencial do edifício.
- Medidor 1406394: Realizando medições do sistema foto-voltaico do edifício.

O objetivo desse cenário é configurar o IoTalker como um *gateway* que colete apenas o valor total de potência ativa correspondente ao prédio inteiro. Convertendo os 3 medidores físicos em um único dispositivo virtual, que mensure o valor total dessa grandeza física.

Esse valor deverá ser enviado para o sistema IMMS (MONTE et al., 2018). O IMMS é uma plataforma de monitoramento de medidores de múltiplos contextos e variáveis. Recebe as medições dos dispositivos, processadas por seus *gateways* via MQTT. O IoTalker se comportará como um desses *gateways*, publicando os valores lidos dos medidores no tópico onde o IMMS recebe as mensagens dos outros *gateways*.

De acordo com a especificação dos medidores, os mesmos obedecem as normas ABNT14522. Desse modo, o IoTalker já disponibiliza um *Driver* para operar com esse tipo de dispositivo. Sendo necessária a definição do arquivo de configuração.

Figura 17 – A) Medidores instalados na sala de força do Dine B)Raspberry PI configurado para o uso do IoTalker, posicionado próximo aos medidores.

De acordo com os engenheiros projetistas do circuito elétrico, a potência ativa total é o somatório da potência ativa das três fases dos medidores 1406089 e 1406396.

Os dois medidores estão conectados utilizando módulos transformadores de corrente, de valores 15 e 12 para os medidores 1406089 e 1406396, respectivamente. Esses valores devem ser levados em consideração para obtenção das medições corretas.

O IoTalker deverá publicar as medições realizadas utilizando MQTT em um *broker* específico, onde as mesmas poderão ser lidas pelo sistema de monitoramento IMMS .

A máquina que executa o IoTalker é um Raspberry PI modelo 3 instalado nas proximidades do quadro de força onde os medidores estão posicionados, como mostrado na figura 17-B. Foi instalada a distro Raspbian (RASPBIAN...,) no Raspberry Pi.

Com base nessas informações, pode-se construir o arquivo de configuração especificando as estruturas de dados da seguinte forma:

- *Services*: Permanece inalterado. Todos os serviços executarão no *localhost*.
- *Drivers*: Considerando que existem dois medidores, deve-se adicionar duas entradas de *Drivers*, as quais chamemos *medidor_1406089* e *medidor_1406396* (valores inseridos no campo *id*. O valor dos campos *type* serão *"opticalAbnt14522"*, visto que os medidores são idênticos. Na definição dos parâmetros, tem-se os campos *baudrate* e USB. Para o *baudrate*, o valor em ambos os *Drivers* serão 9600, que é o padrão de operação normal dos medidores. Para o parâmetro USB, deve-se especificar em qual porta USB do Raspberry o medidor encontra-se conectado. As portas para os medidores 1406089 e 1406396 são */dev/ttyUSB1* e */dev/ttyUSB2*, respectivamente. Feitas essas considerações, a estrutura de dados que configura os *Drivers* no arquivo pode ser escrita como mostrada na figura 18-A.

- Operações: Dado o cenário em questão e tendo-se como base as operações permitidas no IoTalker, serão necessárias 6 operações. A figura 18-B apresenta a estrutura. As operações, expressas em ordem, são:

    1. *Aggregation*: Agregação das potências ativas das 3 fases do medidor 1406089.
    2. *Aggregation*: Agregação das potências ativas das 3 fases do medidor 1406396.
    3. *Calibration*: Calibração da potência ativa total do medidor 1406089, multiplicando seu valor por 15.
    4. *Calibration*: Calibração da potência ativa total do medidor 1406396, multiplicando seu valor por 12.
    5. *Aggregation*: Agregação das potências ativas totais calibradas dos dois medidores.
    6. *Viewer*: Visualização das medições na *Door* que publica as mensagens no *broker* MQTT correto.

- *Doors*: Conforme os requisitos do cenário, as medições obtidas dos medidores deverão ser enviadas ao sistema de monitoramento IMMS. Para tal, devem ser publicadas em um *broker* MQTT, no tópico *"/imms/dine_meters/measurements"*. Para atingir esse fim, deverá ser configurada uma *Door* do tipo MQTT. Seu identificador deverá ser o mesmo referenciado na operação *viewer*, tendo seu valor, portanto, *broker_mqtt*. Seu tipo é MQTT. Os parâmetros de configuração das *Doors* do tipo MQTT, conforme citado anteriormente, são: *topic*, *broker_ip* e *port*. O primeiro deverá ter seu valor como */imms/dine_meters/measurements*, que é o tópico pela qual o IMMS terá acesso as medições recebidas e tratadas pelo IoTalker. O segundo e terceiro parâmetro deverão ser relativos ao *broker* onde o IMMS buscará as mensagens publicadas. A figura 18-C mostra a estrutura para a configuração da *Door* em questão.

Construído o arquivo *config.py*, as preparações para execução do IoTalker estão finalizadas. Visto que foram utilizadas apenas variáveis, *Doors* e *Drivers* existentes, não há necessidade de alteração do arquivo *index.py*.

De acordo com as especificações definidas, o IoTalker irá operar da forma mostrada no diagrama de blocos da figura 19.

Para iniciar a execução do IoTalker, o usuário deverá inicializar os microsserviços individualmente. É fornecido um *script* para a execução simultânea de todos os microsserviços no mesmo *host*, que é o caso base. Se o usuário desejar customizar tal execução, pode alterar o *script* da forma que desejar.

Todos os microsserviços executarão no mesmo *host*, que nesse cenário é o Raspberry Pi, aplicando-se ao caso mais simples e bastando apenas a execução do *script*. Removendo

```
drivers = [
    {
        "id": "medidor_1406089",
        "type": "opticalAbnt14522",
        "parameters":{
            "usb":"/dev/ttyUSB1",
            "baudrate": 9600
        }
    },
    {
        "id": "medidor_1406396",
        "type": "opticalAbnt14522",
        "parameters": {
            "usb": "/dev/ttyUSB2",
            "baudrate": 9600
        }
    }
]
```

```
doors = [
    {
        "id": "broker_mqtt",
        "type": "mqtt",
        "parameters": {
            "topic": '/imms/dine_meters/measurements',
            "broker_ip": "192.168.1.15",
            "port": 1887
        },
        "input": [('total_consumption', 8)]
    }
]
```

```
operations = [

    {
        "type": "aggregation", "parameters": {"aggregation_type": "sum"},
        "input": [("medidor_1406089", "phaseA", 8), ("medidor_1406089", "phaseB", 8),
                  ("medidor_1406089", "phaseC", 8)],
        "output": ("total_medidor_1406089", 8)
    },
    {
        "type": "aggregation", "parameters": {"aggregation_type": "sum"},
        "input": [("medidor_1406396", "phaseA", 8), ("medidor_1406389", "phaseB", 8),
                  ("medidor_1406396", "phaseC", 8)],
        "output": ("total_medidor_1406396", 8)
    },
    {
        "type":"calibration", "parameters": {"equation": "y = 15*x"},
        "input": ("total_medidor_1406089", 8), "output": ("calibrated_medidor_1406089", 8)
    },
    {
        "type":"calibration", "parameters": {"equation": "y = 12*x"},
        "input": ("total_medidor_1406396", 8), "output": ("calibrated_medidor_1406396", 8)
    },
    {
        "type": "aggregation", "parameters": {"aggregation_type": "sum"},
        "input": [("calibrated_medidor_1406396", 8), ("calibrated_medidor_1406089", 8)],
        "output": ("total_consumption", 8)
    },
    {
        "type": "viewer", "parameters": { "timer": 10}, "input": [("Consumption", 1)],
        "output": "broker_mqtt"
    }
]
```

Figura 18 – Estruturas do arquivo de configuração para o caso 1: A) *Drivers*, B) Operações e C) *Doors*.

entradas de serviço do *script* o desenvolvedor poderia executar apenas um microsserviço em uma máquina e os restantes em outra.

Iniciada sua execução, as medições deverão estar sendo publicadas no *broker* MQTT e sendo acessadas pelo IMMS. Existem alguns pontos que podem ser observados ao longo de sua execução, analisando o funcionamento individual dos microsserviços envolvidos.

De acordo com a API do *Reader*, pode-se observar os valores lidos das medições em seu estado bruto, antes de qualquer manipulação matemática. As figuras 20-A e 20-B mostram a resposta para uma requisição HTTP GET ao *Reader*, requisitando as últimas medições dos medidores 1406089 e 1406396. Essa mesma requisição é realizada pelo

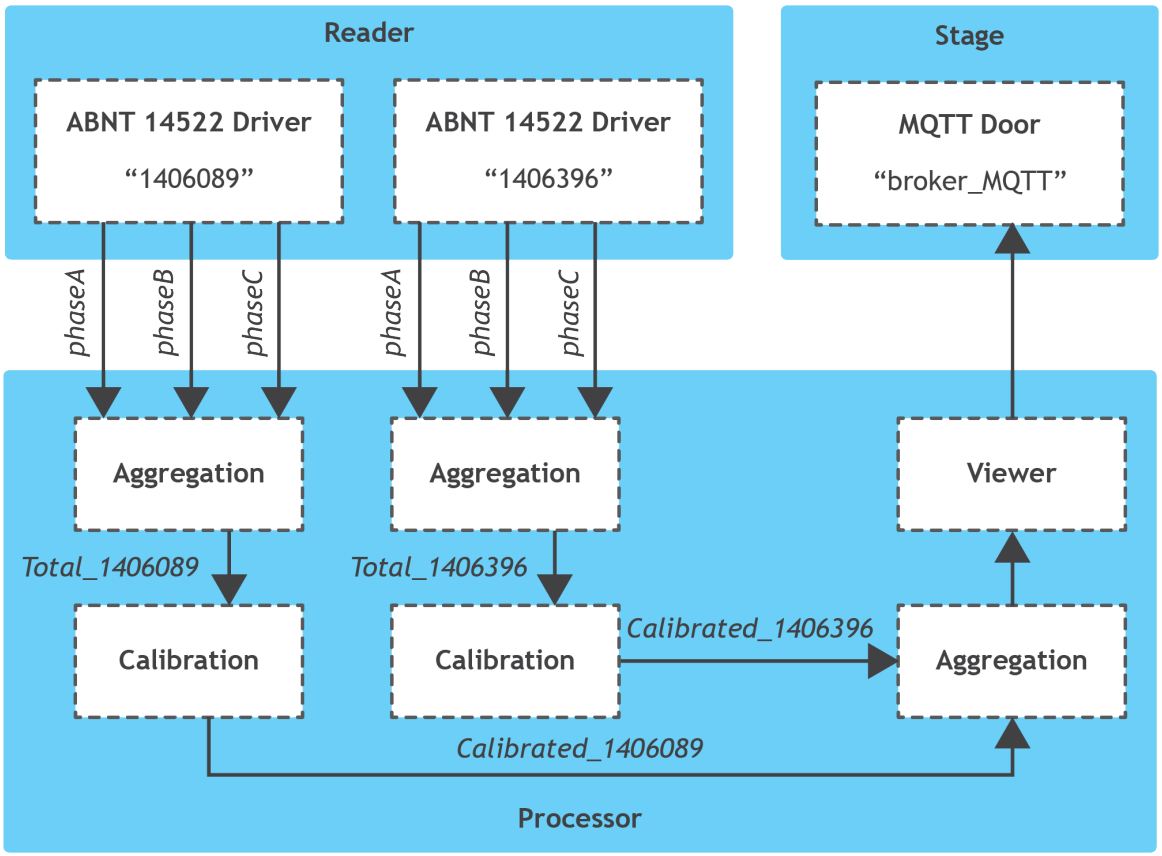

Figura 19 – Visualização em blocos da configuração definida para caso de uso 1.

Processor periodicamente para a obtenção e tratamento desses valores, de acordo com as operações estabelecidas.

Para esse caso 1, também é possível observar respostas intermediárias ao funcionamento do IoTalker no microsserviço *Stage*. Esse microsserviço oferece operações de GET para requisitar detalhes dos *Drivers* em operação e POST para receber medições, sendo essa segunda utilizada pelo *Processor* ao enviar medições para uma *Door*. Pode-se observar seu funcionamento consultando as informações dos *Drivers* em atuação, que na primeira versão do IoTalker, são os dados listados no arquivo de configuração. A figura 20-C ilustra a resposta para essa requisição.

Como resultado final do cenário, as mensagens deverão estar sendo publicadas pelo *Stage* no tópico *"/imms/dine_meters/measurements"*. As medições já terão passado por todas as etapas de processamento e estarão em sua forma final. A figura 21 mostra algumas dessas medições. Foi utilizado o cliente MQTT Mosquitto para efetuar subscrição no mesmo tópico o qual o IMMS se sobrescreve, garantindo assim que as mensagens estão chegando a seu destinatário.

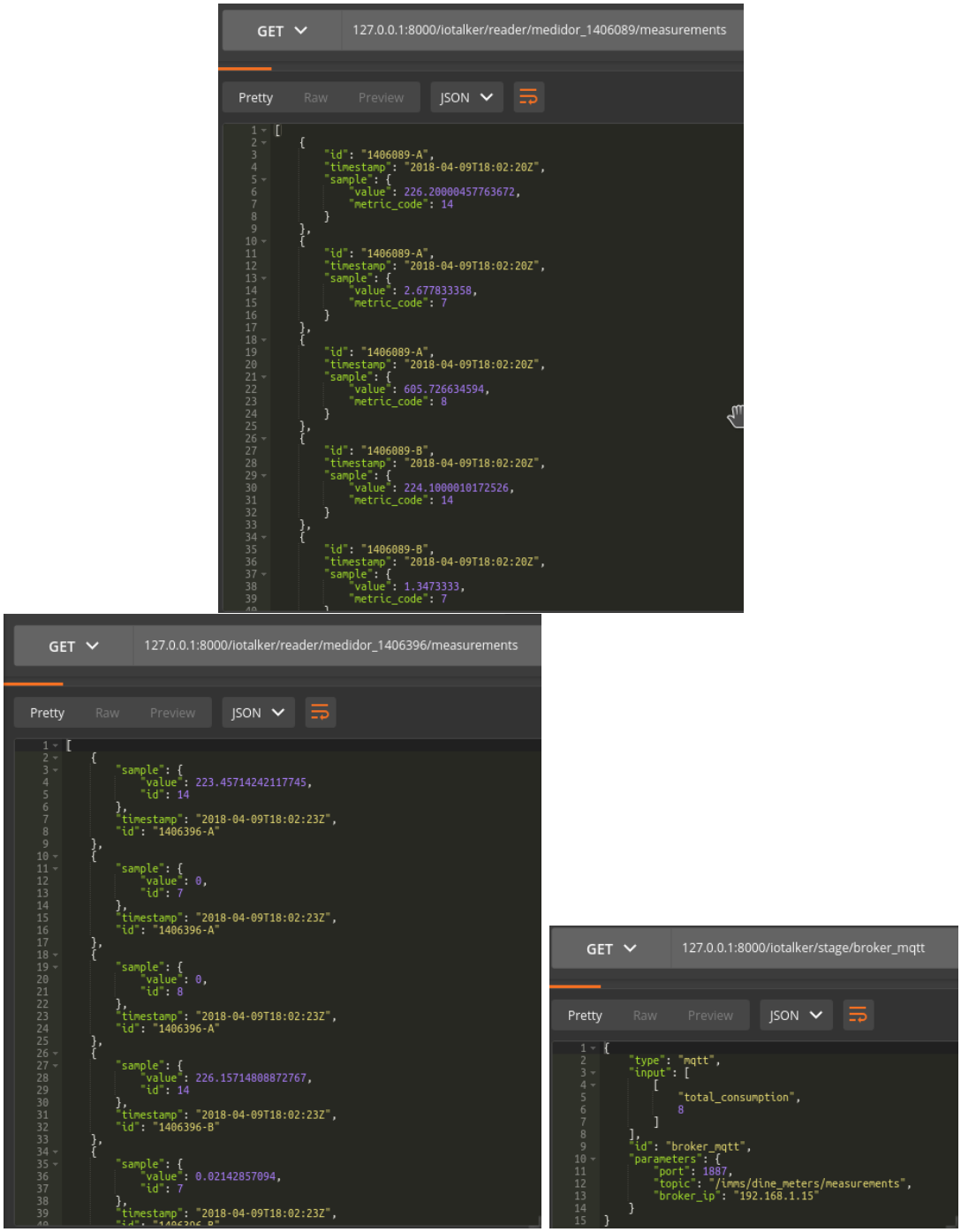

Figura 20 – A) Requisição GET no microsserviço *Reader* retornando as medições do medidor 1406089 B) Requisição GET no microsserviço *Reader* retornando as medições do medidor 1406396 C) Requisição GET no microsserviço *Stage* das informações da *Door "broker_mqtt"*.

5.1.1 Análise

Ao término da realização do experimento, o mesmo cenário foi implementado sem a utilização do IoTalker, utilizando diretamente as bibliotecas já mencionadas, para fins comparativos.

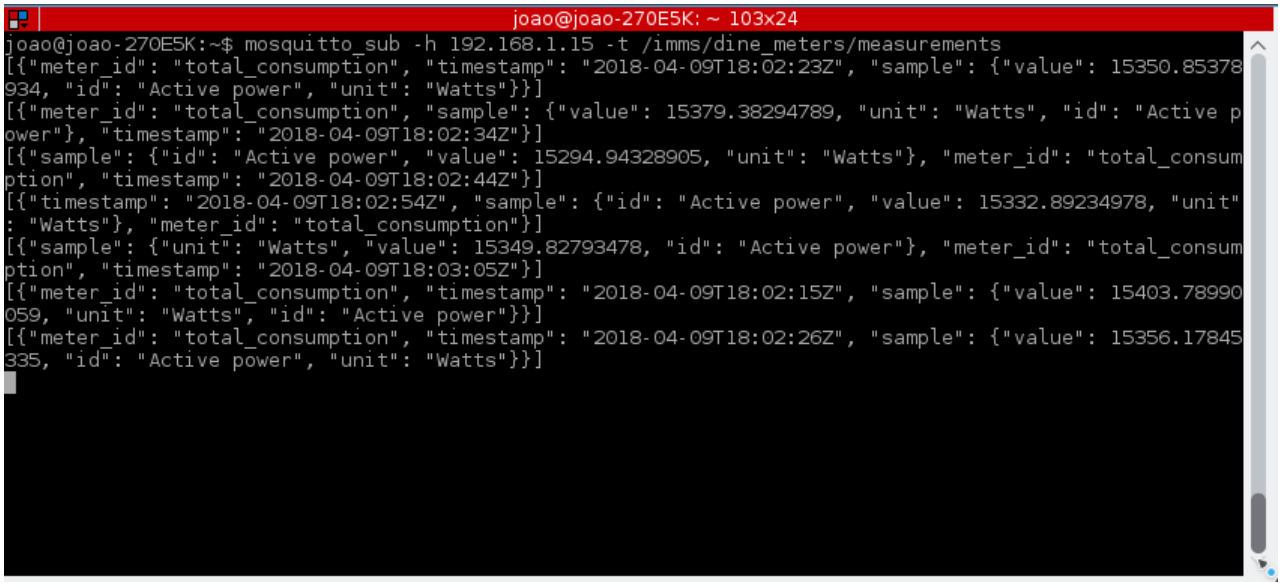

Figura 21 – Mensagens publicadas pelo IoTalker através da *Door broker_mqtt*.

A implementação em código do cenário 1 pode ser subdividida nas seguintes etapas:

1. Leitura das medições dos medidores de energia.
2. Criação de um servidor web com as mesmas urls do *Reader*.
3. Manipulações matemáticas.
4. Conexão com *broker* MQTT.
5. Encapsulamento da conexão com *broker* MQTT em forma de uma API, trabalho esse realizado pelo *Stage*.

As linhas de código necessárias para a implementação de cada uma dessas etapas foram analisadas individualmente, em forma de funções e classes necessárias, apenas para fins comparativos. A conexão entre as mesmas pode ser realizada de diferentes formas, pelo programador. O número de linhas, aproximado, para cada passo pode ser observado na tabela 8. Esses resultados são arredondados para baixo, pois o desenvolvedor ainda precisará integrar os *snippets* de código.

Embora o arquivo de configuração para esse cenário possua cerca de 90 linhas, as mesmas foram espaçadas para melhor visualização de seu conteúdo.

Se atribuídas uma linha de código por elemento nas estruturas de dados do arquivo de configuração, a quantidade de linhas torna-se o mostrado na tabela 9.

Comparando-se a quantidade linhas de código escritas com e sem o IoTalker, verifica-se uma diferença significativa de mais de 200 linhas de código.

Ainda podem ser considerados outros fatores, como os tempos de aprendizagem necessários para tratar a comunicação com os medidores de energia e utilização das bibliotecas

Etapa	Linhas de código
1	180
2	15
3	5
4	15
5	15
Total	**230**

Tabela 8 – Linhas de código necessárias para a implementação pura do cenário 1 sem a utilização do IoTalker.

Estrutura de dados	Linhas de código
Services	5
Drivers	2
Operações	6
Doors	1
Total	**14**

Tabela 9 – Estimativa de linhas contidas no arquivo de configuração para a execução do IoTalker no cenário 1.

para lidar com comunicação MQTT e servidores web. O IoTalker também se destaca nesse aspecto, devido a sua configuração simplificada.

5.2 CENÁRIO 2

O GPRT conta com um *nobreak*, mostrado na figura 22, para manter as máquinas e equipamentos essenciais funcionando em casos de falta de energia.

A comunicação com esse dispositivo acontece via SNMP e como já mencionado, o IoTalker possui um *Driver* para gerenciar e abstrair essa comunicação. O *nobreak* não conta com suporte à SNMP. Esse suporte é atingido através de um módulo adicional, que é conectado ao *nobreak* via porta serial. Esse módulo está posicionado encima do aparelho e também é mostrado na figura 22.

Dentre as medições que podem ser lidas, o administrador do laboratório tem interesse na carga da saída do *nobreak* e no estado da bateria. A primeira informa o quanto o equipamento encontra-se sobrecarregado em relação ao valor da carga de entrada. O segundo informa o *status* em porcentagem da bateria do *nobreak*, que permanece constante em 100% e tendo seu valor reduzido apenas em situações de falta de energia. Situações essas em que o administrador deseja ser notificado.

Figura 22 – *Nobreak* instalado no GPRT. As medições são requisitadas via SNMP.

Para atender às demandas desse cenário, a configuração do IoTalker deve atingir dois objetivos: Permitir ao administrador a visualização da medição mais recente da carga de

saída do *nobreak* e a emissão de uma notificação quando a bateria do *nobreak* começar a ser utilizada, indicando uma falta de energia.

Bem como os medidores de energia do cenário 1, o *nobreak* possui três classes de subsensores com medições e valores distintos. Enquanto os medidores de energia mensuravam valores para cada uma de suas 3 fases, o *nobreak* mensura valores para bateria, carga de *input* e carga de *output*.

De modo a cumprir estes requisitos, pode-se construir o arquivo de configuração especificando as estruturas de dados da seguinte forma:

- *Services*: Permanece inalterado. Todos os serviços executarão no *localhost*. Nesse cenário o *host* é um computador comum.
- *Drivers*: Para esse cenário, o IoTalker precisará lidar com apenas com o *nobreak*, caso que demanda apenas um *Driver*. Os parâmetros para esse *Driver* são o endereço IP e porta de acesso do servidor SNMP, respectivamente *192.168.0.35* e 161.
- Operações: O conjunto de operações para esse cenário é bem mais simples que o conjunto do cenário 1. Não são necessárias manipulações matemáticas ou agregação de sensores. Dado o cenário em questão e tendo-se como base as operações permitidas no IoTalker, serão necessárias apenas 2 operações:
  1. *Notification*: Notificação via e-mail caso a bateria esteja menor que 100%.
  2. *Viewer*: Visualização das medições relativas a carga na entrada e na saída do *nobreak* na *Door* que cria um servidor web, acessível por qualquer navegador.
- *Doors*: As medições obtidas do *nobreak* deverão estar disponíveis em um servidor web. Deverá ser utilizada a *Door* do tipo HTTP, passando-se como *input* os valores solicitados. Os parâmetros para configuração da *Door* HTTP são a url e a porta na qual o servidor web disponibilizará os dados.

As figuras 23-A, 23-B e 23-C mostram respectivamente as estruturas de dados que compõem o arquivo de configuração desse cenário.

De acordo com a configuração definida para o caso 2, pode-se visualizar as operações e funcionamento dos *Drivers* e *Doors* como o diagrama de blocos mostrado na figura 24.

Repetindo a análise realizada no cenário 1, pode-se analisar o funcionamento intermediário dos microsserviços envolvidos. Efetuando-se uma requisição GET aos *Drivers* do *Reader*, pode-se obter as últimas medições coletadas do *nobreak* em seu estado bruto, como mostrado na figura 25-A. Uma requisição GET ao *Stage* também retorna os dados da única *Door* em funcionamento no sistema, como mostrado na figura 25-B.

Os resultados finais desse cenário devem ser: O e-mail enviado quando carga da bateria for menor que 100% e um servidor web executando e respondendo a solicitações na url *"/iotalker/measurements"*. A figura 26 mostra o servidor web, disponível para acesso via navegador, concluindo o cenário 2.

```
drivers = [
    {
        "id": "nobreak",
        "type": "snmp",
        "parameters":{
            "ip":"192.168.0.35",
            "port": 161
        }
    }
]
```

```
operations = [
    {
        "type": "email-notifier",
        "parameters": {
            "boolean_expression": "x<100",
            "email": "suporte@gprt.ufpe.br"
        },
        "input": [("nobreak", "battery", 16)],
        "output": ""
    },
    {
        "type": "viewer", "parameters": { "timer": 10},
        "input": [("nobreak", "input", 8), ("nobreak", "output", 8)],
        "output": "webserver"
    }
]
```

```
doors = [
    {
        "id": "webserver",
        "type": "http",
        "parameters": {
            "url": '/iotalker/measurements',
            "port": 33010
        },
        "input": [("nobreak", "input", 8),
                  ("nobreak", "output", 8)]
    },
]
```

Figura 23 – Estruturas de dados que compõem o arquivo de configuração para o caso 2: A) *Drivers* B) Operações C) *Doors*.

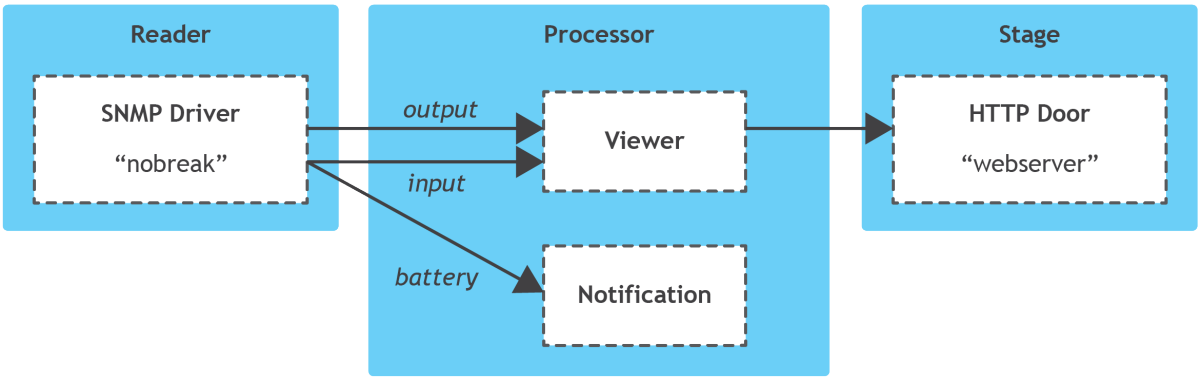

Figura 24 – Visualização em blocos da configuração definida para o caso de uso 2.

5.2.1 Análise

O cenário 2, assim como o primeiro cenário, foi implementado utilizando as mesmas bibliotecas, sem a utilização do IoTalker, para fins de comparação da quantidade de linhas de código escritas.

A implementação em código do cenário 2 pode ser subdividida nas seguintes etapas:

1. Leitura das medições do *nobreak*.
2. Criação de um servidor web com as mesmas urls do *Reader*.
3. Envio de e-mail em caso de uma condição ser verdadeira.

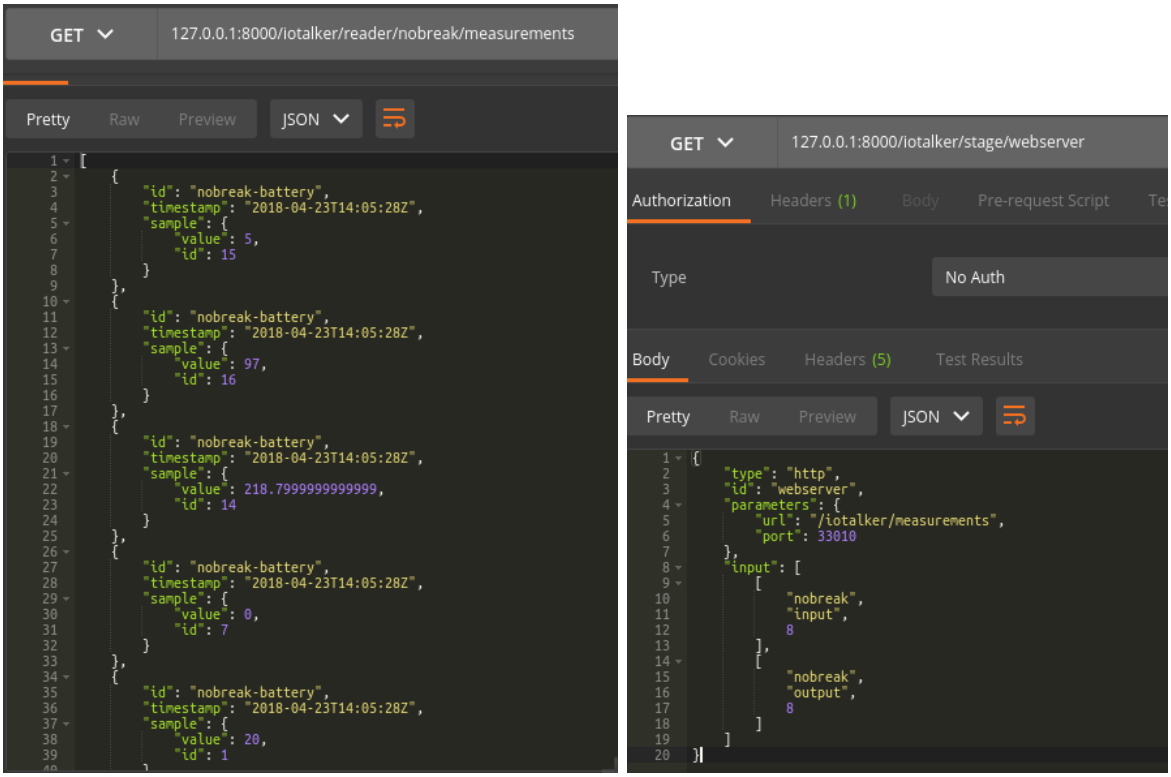

Figura 25 – A) Requisição GET ao *Reader*, retornando as últimas medições do *nobreak* B) Requisição GET ao *Stage* solicitando os dados das *Doors* em atuação.

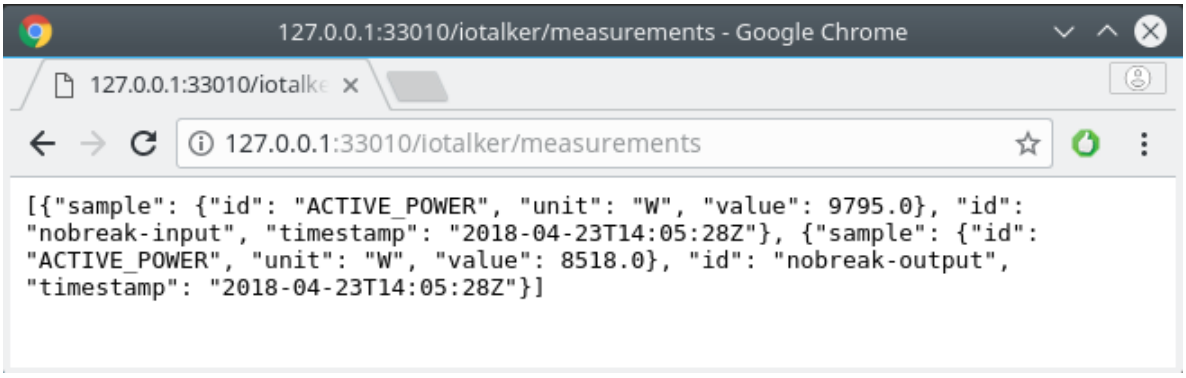

Figura 26 – Servidor *web* respondendo requisições com as últimas medições das variáveis de interesse do *nobreak*.

4. Criação de um servidor web onde as medições ficam disponíveis, simulando o comportamento do *Stage*.

Seguindo o padrão da análise do primeiro cenário, foram considerados *snippets* de código necessários para a implementação das funcionalidades descritas, cabendo ao desenvolvedor a conexão entre as mesmas.

Sob estas condições, a quantidade de linhas de código para cada uma das etapas descritas são, aproximadamente, mostradas na tabela 10.

A tabela 11 mostra a quantidade de linhas necessárias no arquivo de configuração

Etapa	Linhas de código
1	90
2	15
3	10
4	20
Total	**135**

Tabela 10 – Linhas de código para implementação de cada uma das funcionalidades necessárias ao cenário 2 sem a utilização do IoTalker.

Estrutura de dados	Linhas de código
Services	5
Drivers	1
Operações	2
Doors	1
Total	**9**

Tabela 11 – Linhas de código por estrutura de dados do arquivo de configuração para o cenário 2.

para esse cenário. Foram consideradas uma linha por item em cada uma das estruturas de dados que compõem o arquivo.

Semelhante aos resultados obtidos no cenário 1, houve uma diferença significativa na quantidade de linhas necessárias para os dois casos. Desconsiderando ainda o tempo necessário para aprendizagem sobre a biblioteca que trata as comunicações SNMP.

6 CONCLUSÃO

Ao passo que novos dispositivos surgem todos os dias, facilitando a vida das pessoas, a heterogeneidade traz dificuldades aos desenvolvedores que desejam utilizar esses dispositivos em suas aplicações. Cada dispositivo possui seus protocolos, sua API e suas particularidades.

Este trabalho traz a proposta do IoTalker, um *gateway* IoT expansível. O IoTalker busca homogeneizar essa irregularidade de dispositivos, trazendo-os a uma linguagem comum e um fluxo de operações totalmente definido pelo desenvolvedor, através da agregação de *features* poderosas e trazendo mais versatilidade a um *gateway* IoT.

A arquitetura de software do IoTalker é composta por microsserviços responsáveis por tarefas específicas:

- *Reader*: desencapsula os dados em um determinado protocolo e fornece uma API onde esses dados tornam-se acessíveis.
- *Processor*: aplica uma sequência de processamentos, totalmente definidas pelo desenvolvedor, nos dados obtidos do *Reader*.
- *Storage*: Fornece armazenamento persistente, quando requisitado. Os dados armazenados são acessíveis via uma API, também oferecida pelo microsserviço.
- *Stage*: encapsula os dados resultantes das operações realizadas no *Processor* em um novo protocolo, tornando seu acesso disponível à outras aplicações, desenvolvedores e *middlewares* IoT.
- *Master API*: Fornece transparência de localização, roteando requisições para os outros microsserviços, visto que os mesmos podem estar sendo executados em outras máquinas.

Após a etapa de planejamento, implementação e testes, o IoTalker foi aplicado com sucesso em dois cenários reais, atingindo seus requisitos.

No primeiro cenário, de 3 medidores de energia instalados em um prédio da UFPE, 2 deles foram utilizados para criar um único sensor virtual, que mensura o consumo de energia do edifício.

As medições foram obtidas através de uma conexão óptica serial, calibradas, agregadas, processadas e, por fim, publicadas em um *broker* MQTT, num tópico onde são recebidas por outra plataforma de software.

No segundo cenário, o objetivo foi o monitoramento de um *nobreak*, visualizando sua carga de entrada e de saída. As medições foram acessadas via SNMP, processadas e disponibilizadas através de um servidor web, acessível de qualquer navegador de Internet.

Nesse cenário o IoTalker também foi configurado para enviar um e-mail ao administrador de redes do GPRT, quando a carga da bateria for menor que 100%, indicando uma queda de energia.

Em ambos os cenários, o IoTalker foi capaz de desempenhar, com poucas linhas de código, muitas funções no âmbito de um fluxo de processamento completo ao extrair as informações de dispositivos IoT e disponibilizá-las de diferentes formas.

6.1 TRABALHOS FUTUROS

Embora a plataforma desenvolvida esteja estável, funcionando corretamente para os cenários supracitados, ainda restam tarefas a serem realizadas.

Como trabalhos futuros, pode-se destacar:

- Implementação de uma interface gráfica, facilitando a configuração de *Doors*, *Drivers* e operações.
- Suporte a novos protocolos.
- Implementação de uma base de *Doors* e *Drivers*, acessíveis e disponíveis para o usuário via *download*.
- Escrita de testes automatizados para o projeto.
- Realização de avaliações de desempenho na plataforma.

6.2 CONTRIBUIÇÕES

Em comparação com os trabalhos relacionados, o IoTalker apresenta uma combinação entre *features* encontradas nos *gateways* IoT e *middlewares* IoT.

O IoTalker não é um *middleware* IoT, equivalendo a apenas um dos módulos que os compõem: o *gateway*. Embora não seja um *middleware* IoT, sua arquitetura se enquadra nos requisitos do tipo *actor-based* (NGU et al., 2017b). É subdividido em microsserviços que podem ser executados em diferentes máquinas ou camadas. Seus *Actors* são os *Drivers*, Operações e *Doors*, que o usuário pode manipular da forma que desejar para atingir seus objetivos.

Se o desenvolvedor deseja executa-lo em um dispositivo com poucos recursos computacionais, pode configurar apenas a execução do *Reader*, mantendo os outros microsserviços numa máquina mais potente, por exemplo. Seus microsserviços possuem APIs individuais, permitindo que os usuários usem apenas os que desejarem ou todos. Essa característica se assemelha aos *middlewares* IoT do tipo *cloud-based*.

Essa questão foi levantada no cenário 1, principalmente, visto que o *host* era um Raspberry PI. A princípio o *Storage* poderia ter dificuldades em ser executado, por tratar-

se de um banco de dados. Mas não houve necessidade, pois havia uma versão MongoDB nos repositórios do Raspbian.

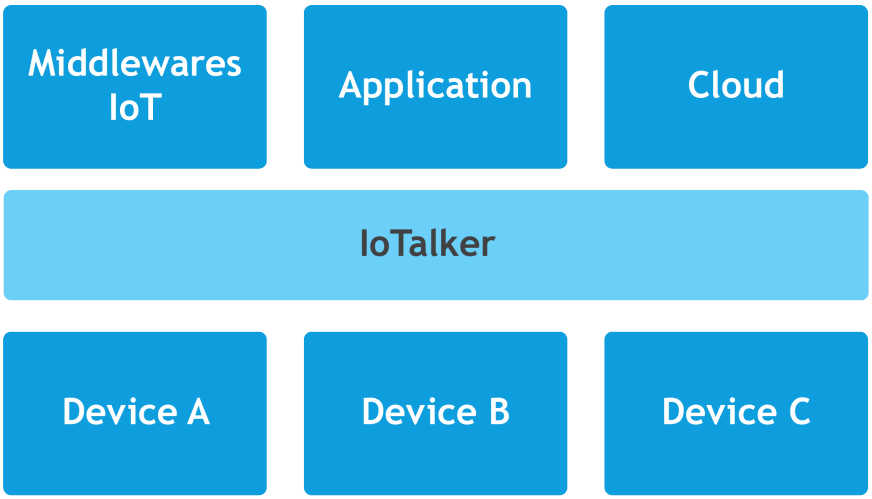

Figura 27 – Posicionamento do IoTalker em uma arquitetura IoT.

O IoTalker traz algumas das vantagens e versatilidade de um *middleware* IoT dando mais poder a uma camada inferior da arquitetura IoT, acima dos dispositivos e abaixo da camada de aplicação, como mostrado na figura 27.

Embora uma biblioteca de *Drivers*, *Doors* e Operações ainda seja um trabalho futuro, sua implementação de protocolos compatíveis foi realizada de modo a ser expansível. É permitido ao usuário adicionar a compatibilidade a novos protocolos e aumentar a abrangência de dispositivos suportados pela plataforma, fazendo com que a mesma não esteja limitada a um conjunto inicial de dispositivos.

Nos dois cenários, o IoTalker obteve diferença significativa em relação a implementação sem sua utilização. Em poucas linhas de código, é capaz de desempenhar funções poderosas na comunicação com diferentes protocolos e dispositivos IoT, minimizando o trabalho do desenvolvedor.

Com base nessas observações e no trabalho apresentado, pode-se elencar as principais contribuições do IoTalker com a literatura:

1. Uma plataforma *IoT* onde o desenvolvedor pode trabalhar utilizando Python3, ao invés de apenas Java, Javascript e Node.js, que são mais comuns.

2. *Gateway* com *features* e versatilidade de um *middleware* IoT, dentre as quais destaca-se a implementação de uma arquitetura expansível de suporte a novos protocolos.

3. O desenvolvedor consegue manipular o funcionamento do *gateway* apenas alterando e escrevendo o arquivo de configuração, economizando escrita de código.

4. Suporte a medidores de energia padronizados com a norma ABNT14522. Por tratar-se de um padrão brasileiro, foi encontrado pouco material a respeito na literatura.

Alguns dos conceitos e módulos de software do IoTalker foram utilizados na implementação do IMMS: um middleware IoT genérico adaptável à diferentes cenários. Um short paper sobre o IMMS foi publicado na conferência ISCC (IEEE...,) (qualis A2), com o título *IMMS: IoT Management and Monitoring System* (MONTE et al., 2018).

REFERÊNCIAS

AAZAM, M.; HUNG, P. P.; HUH, E. Smart gateway based communication for cloud of things. In: *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. [S.l.: s.n.], 2014. p. 1–6.

ABERER, K.; HAUSWIRTH, M.; SALEHI, A. A middleware for fast and flexible sensor network deployment. In: *Proceedings of the 32Nd International Conference on Very Large Data Bases*. VLDB Endowment, 2006. (VLDB '06), p. 1199–1202. Disponível em: <http://dl.acm.org/citation.cfm?id=1182635.1164243>.

ABNT. *Página contendo os dados sobre a norma ABNT14522*. Disponível em: <http://www.abntcatalogo.com.br/norma.aspx?ID=822>.

AMAZON. *Descrição de bancos de dados não-relacionais*. Disponível em: <https://aws.amazon.com/pt/nosql>.

Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, v. 33, n. 3, p. 42–52, May 2016. ISSN 0740-7459.

DESAI, P.; SHETH, A.; ANANTHARAM, P. Semantic gateway as a service architecture for iot interoperability. In: *2015 IEEE International Conference on Mobile Services*. [S.l.: s.n.], 2015. p. 313–319. ISSN 2329-6429.

DIGI. *Página oficial do protocolo zigbee*. Disponível em: <https://www.zigbee.org>.

EISENHAUER, M.; ROSENGREN, P.; ANTOLIN, P. A development platform for integrating wireless devices and sensors into ambient intelligence systems. In: *2009 6th IEEE Annual Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks Workshops*. [S.l.: s.n.], 2009. p. 1–3. ISSN 2155-5486.

GARTNER. *Consultoria Gartner*. Disponível em: <https://www.gartner.com/en>.

GOOGLE Fit. Disponível em: <https://www.google.com/fit/>.

GUOQIANG, S.; YANMING, C.; CHAO, Z.; YANXU, Z. Design and implementation of a smart iot gateway. In: *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. [S.l.: s.n.], 2013. p. 720–723.

IEEE Symposium on Computers and Communications. Disponível em: <http://iscc2018.ieee-iscc.org/>.

KIM, S.; CHOI, H.; RHEE, W. Iot home gateway for auto-configuration and management of mqtt devices. In: *2015 IEEE Conference on Wireless Sensors (ICWiSe)*. [S.l.: s.n.], 2015. p. 12–17.

KUROSE, J. F.; ROSS, K. W. *Computer Networking: A Top-Down Approach (6th Edition)*. 6th. ed. [S.l.]: Pearson, 2012. ISBN 0132856204, 9780132856201.

MINIMALMODBUS. Disponível em: <https://minimalmodbus.readthedocs.io/en/master/installation.html>.

MONGODB. *Página oficial do banco de dados MongoDB*. Disponível em: <https://www.mongodb.com>.

MONTE, J. V. L. d.; FRAGA, V. M. d. S.; RIBEIRO, A. M. N. C.; SADOK, D.; KELNER, J. Imms: Iot management and monitoring system. In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. [S.l.: s.n.], 2018. p. 00422–00425. ISSN 1530-1346.

MOSQUITTO broker. Disponível em: <https://mosquitto.org/>.

MQTT. Disponível em: <http://mqtt.org>.

NGU, A. H.; GUTIERREZ, M.; METSIS, V.; NEPAL, S.; SHENG, Q. Z. Iot middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal*, v. 4, n. 1, p. 1–20, Feb 2017. ISSN 2327-4662.

NGU, A. H.; GUTIERREZ, M.; METSIS, V.; NEPAL, S.; SHENG, Q. Z. Iot middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal*, v. 4, n. 1, p. 1–20, Feb 2017. ISSN 2327-4662.

NODERED. Disponível em: <https://nodered.org/>.

ORIENTDB. *Página oficial do banco de dados OrientDB*. Disponível em: <https://orientdb.com/>.

PERSSON, P.; ANGELSMARK, O. Calvin – merging cloud and iot. *Procedia Computer Science*, v. 52, p. 210 – 217, 2015. ISSN 1877-0509. The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015). Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050915008595>.

PINTUS, A.; CARBONI, D.; PIRAS, A. Paraimpu: A platform for a social web of things. In: *Proceedings of the 21st International Conference on World Wide Web*. New York, NY, USA: ACM, 2012. (WWW '12 Companion), p. 401–404. ISBN 978-1-4503-1230-1. Disponível em: <http://doi.acm.org/10.1145/2187980.2188059>.

PTOLEMY. Disponível em: <http://ptolemy.berkeley.edu/ptolemyII/>.

PUDER, A.; RÖMER, K.; PILHOFER, F. *Distributed systems architecture: a middleware approach*. [S.l.]: Elsevier, 2011.

PYMONGO. Disponível em: <https://api.mongodb.com/python/current/>.

PYSNMP. Disponível em: <http://snmplabs.com/pysnmp/index.html>.

RASPBIAN page. Disponível em: <https://www.raspberrypi.org/downloads/raspbian/>.

XBEE Python3 library. Disponível em: <https://xbplib.readthedocs.io/en/latest/getting_started_with_xbee_python_library.html>.

XIVELY. Disponível em: <https://xively.com/>.

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.