UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROJETO DE GRADUAÇÃO

Gabriel Moulin Dan

Desenvolvimento de uma solução M2M de sensoriamento de data center baseado em IoT

Vitória-ES
Junho/2018

Gabriel Moulin Dan

Desenvolvimento de uma solução M2M de sensoriamento de data center baseado em IoT

Parte manuscrita do Projeto de Graduação do aluno Gabriel Moulin Dan, apresentada ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Vitória-ES
Junho/2018

Gabriel Moulin Dan

Desenvolvimento de uma solução M2M de sensoriamento de data center baseado em IoT

Parte manuscrita do Projeto de Graduação do aluno Gabriel Moulin Dan, apresentada ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Aprovado em 26, de Junho de 2018.

COMISSÃO EXAMINADORA:

Prof. Me. Rafael Emerick Zape de Oliveira
Instituto Federal do Espírito Santo
Orientador

Prof. Dr. Hans Jorg Andreas Schneebeli
Universidade Federal do Espírito Santo
Coorientador

Prof. Dra. Eliete Maria de Oliveira Caldeira
Universidade Federal do Espírito Santo
Examinador

Prof. Dr. Rodolfo da Silva Villaça
Universidade Federal do Espírito Santo
Examinador

Vitória-ES
Junho/2018

RESUMO

Data centers são ambientes utilizados para hospedar componentes e recursos computacionais, armazenamento, processamento e transmissão de dados. Tais ambientes possuem usualmente rigoroso controle das variáveis de ambiente e sistemas de segurança com restrição de acesso. Desta forma, com um intuito de verificar o estado deste ambiente de forma automática, o presente trabalho propõe uma solução para monitoramento de sensores em *data centers* focado em interação com sistemas. A solução proposta foi implementada a partir de conceitos de IoT, serviços Web e microsserviços utilizando virtualização via *containers* Docker. Um protótipo da solução foi implantado no *data center* do PoP-ES/RNP. A partir dos testes realizados, a solução adotada mostrou-se satisfatória possibilitando o monitoramento de sensores de forma contínua e fácil integração com outros sistemas via API.

Palavras-chave: Internet das coisas; Sensoriamento; Serviços Web; Microsserviços.

ABSTRACT

Data centers are environments used to host computing components and resources, storage, processing, and data transmission. Such environments usually have strict environment control and security systems with access restriction. Thus, in order to check this environment automatically, the present work proposes a solution for sensor monitoring in data centers focused on system interaction. The proposed solution was implemented from IoT, Web services and microservices concepts, using Docker container virtualization. A prototype of the solution was deployed in PoP-ES/RNP's data center. Through the accomplished tests, the adopted solution was satisfactory allowing sensor monitoring in a continuous way and easy integration with other systems by API.

Keywords: Internet of Things; Sensor monitoring; Web services; Microservices.

LISTA DE FIGURAS

Figura 1 – Esquema de arquitetura a ser utilizada. 14
Figura 2 – Modelo de conexão utilizando protocolo MQTT. 15
Figura 3 – Modelo de serviço Web com arquitetura REST. 19
Figura 4 – Modelo de arquitetura monolítica e microsserviços. 20
Figura 5 – Estrutura de virtualização de *containers* Docker e VMs 21
Figura 6 – Modelo de escalonamento de arquitetura monolítica e microsserviços. . 22
Figura 7 – Modelo de arquitetura proposta. 25
Figura 8 – Fluxo de informação no serviço de coleta de dados. 26
Figura 9 – Atenuação de entrada do conversor A/D. 27
Figura 10 – Esquemático de sensoriamento adotado. 28
Figura 11 – Fluxo de mensagens no *broker* MQTT. 29
Figura 12 – Placa de circuito impresso construída. 31
Figura 13 – Uso de memória do *broker* MQTT em relação ao número de *subscribers*. 34
Figura 14 – Uso de CPU do *broker* MQTT em relação ao número de *subscribers*. . 35
Figura 15 – Uso de CPU do InfluxDB em relação ao número de entradas no banco 36
Figura 16 – Uso de espaço em disco do InfluxDB em relação ao número de entradas no banco 37
Figura 17 – Uso de memória do InfluxDB em relação ao número de entradas no banco 37
Figura 18 – *Dashboard* do Grafana. 39
Figura 19 – Gráfico de de corrente no circuito do Ar Condicionado 4 do PoP-ES/RNP. 39
Figura 20 – Gráfico de de corrente no circuito do Ar Condicionado 1 do PoP-ES/RNP, o qual apresentava defeito. 40

LISTA DE QUADROS

Quadro 1 – Modelo de formato JSON. 19
Quadro 2 – Modelo de resposta em formato JSON. 30

LISTA DE ABREVIATURAS E SIGLAS

A/D	Analógico-Digital
API	*Application Programming Interface*
CoAP	*Constrained Application Protocol*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hyper Text Transfer Protocol Secure*
I2C	*Inter-Integrated Circuit*
IP	*Internet Protocol*
IoT	*Internet of Things*
JSON	*JavaScript Object Notation*
M2M	*Machine to Machine*
MQTT	*Message Queuing Telemetry Transport*
PoP-ES/RNP	Ponto de Presença da RNP no Espírito Santo
QDC	Quadro de Distribuição de Circuitos
REST	*Representational State Transfer*
SNMP	*Simple Network Management Protocol*
SOAP	*Simple Object Access Protocol*
SQL	*Structured Query Language*
TCP	*Transport Control Protocol*
TSDB	*Time-Series Database*
VM	*Virtual Machine*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO . . . 10
1.1 Contexto . . . 10
1.2 Motivação . . . 10
1.3 Objetivo . . . 11
1.3.1 Objetivo Geral . . . 11
1.3.2 Objetivos Específicos . . . 11
1.4 Metodologia . . . 12
1.5 Estrutura da Monografia . . . 12

2 FUNDAMENTAÇÃO TEÓRICA . . . 13
2.1 Internet das coisas . . . 13
2.1.1 Arquitetura . . . 13
2.1.1.1 Camada de Percepção . . . 14
2.1.1.2 Camada de Rede . . . 14
2.1.1.3 Camada de Aplicação . . . 16
2.1.2 Comunicação M2M . . . 16
2.2 *Hardware* . . . 16
2.2.1 Software Embarcado . . . 17
2.3 Serviços Web . . . 17
2.3.1 API REST . . . 17
2.3.2 Servidor Web . . . 19
2.4 Arquitetura de microsserviços . . . 20
2.4.1 Virtualização em *containers* Docker . . . 21
2.4.2 Vantagens . . . 21
2.4.3 Desvantagens . . . 22
2.5 Banco de dados de séries temporais . . . 22

3 SOLUÇÃO PROPOSTA . . . 24
3.1 Arquitetura . . . 24
3.1.1 Sensoriamento . . . 26

3.1.2 Comunicação Microcontroladores x Coletor 28
3.1.3 Coletor 28
3.1.4 Banco de Dados 29
3.1.5 Servidor API 29
3.2 Protótipo 30

4 EXPERIMENTOS E RESULTADOS 32
4.1 Avaliação do serviço de aquisição de dados 32
4.1.1 Tempo de resposta do sensoriamento 32
4.1.2 Serviço *broker* 33
4.1.3 Coletor 34
4.1.4 Serviço de Armazenamento 35
4.2 Servidor API 38
4.3 Prova de conceito 38

5 CONCLUSÕES E TRABALHOS FUTUROS 41
5.1 Trabalhos Futuros 42

REFERÊNCIAS BIBLIOGRÁFICAS 43

1 INTRODUÇÃO

1.1 Contexto

A Internet das Coisas (IoT do inglês, *Internet of Things*) é uma realidade cada vez mais presente em nosso cotidiano. Com dispositivos móveis e microcontroladores poderosos a um baixo custo de aquisição, a expansão do IoT vem atingindo cidades inteligentes, automação residencial, automação industrial e também *data centers*.

A natureza crítica dos serviços de informática tem sensibilizado várias empresas para controle da "saúde"de seus *data centers*. É primordial que o ambiente seja extremamente controlado para manter a integridade de todos os equipamentos ali hospedados. A dificuldade em realizar este controle em *data centers* de pequeno/médio porte é o custo das soluções disponíveis no mercado. Os sistemas existentes atualmente têm em geral preços elevados, encarecendo assim a operação de pequenos *data centers*.

O uso de tecnologia baseada em IoT permite a criação de uma rede onde sensores e atuadores podem ser conectados através de redes sem fio respondendo através de um IP, tornando fácil a integração com softwares de monitoramento. Além disso, o uso de redes sem fio para esta conexão é interessante pois permite um posicionamento de sensores e atuadores de maneira mais flexível no ambiente.

1.2 Motivação

A principal motivação desse trabalho está na automatização do sensoriamento do *data centers* pertencente ao PoP-ES/RNP, visto que o sistema de monitoramento atual não consegue caracterizar bem o ambiente, além de funcionar de maneira apenas semi-automatizada, onde uma das variáveis é monitorada de forma manual.

Um sistema de medição de temperatura funciona de maneira automática, porém tal medição é realizada apenas em um ponto do *data center*, no *rack* onde se localizam os equipamentos mais críticos.

Além da temperatura, a única outra medição realizada é a de corrente nos quadros de distribuição de circuitos elétricos (QDC) do *data centers*, a qual é realizada a cada mês, no conjunto de QDCs que atendem a sala de equipamentos críticos, esta sendo realizada manualmente, onde um técnico se desloca até o local com um multímetro e realiza 3 medições ao longo do dia, pela manhã, tarde e noite. Sendo assim, o recurso

utilizado (hora/homem) para realizar a medição de corrente dos QDCs é além de ineficiente, demasiado caro.

Surge então o interesse em automatizar a medição de corrente dos circuitos dos QDCs que abastecem o *data center*, assim como a distribuição de mais sensores no ambiente, a fim de ter uma melhor percepção da distribuição de calor e umidade no espaço em questão. Apesar de existirem soluções disponíveis no mercado, busca-se uma solução minimalista e de baixo custo, implementada a partir de conceitos de IoT onde a expansão do número de sensores ocorra de maneira a impactar minimamente o ambiente em questão.

1.3 Objetivo

1.3.1 Objetivo Geral

O objetivo geral do trabalho é criar solução de integração de redes de sensores e atuadores baseados em tecnologia IoT que: i) realize o monitoramento e controle das variáveis de ambiente do *data center* do PoP-ES/RNP de modo automático, ii) utilize *hardware* de baixo custo, por meio do direcionamento da maior complexidade computacional para um ponto de maior poder de processamento emulado por *software* e iii) tenha uma Interface de Programação de Aplicativos (API - do inglês *Application Programming Interface*) de fácil integração com outros sistemas já utilizados.

O desenvolvimento desta rede engloba diversas etapas:

- Escolha de componentes, sensores, atuadores e microcontroladores;
- Adoção de uma arquitetura utilizada para o monitoramento;
- Escolha de protocolos de comunicação, escolha de *softwares* para realizar a comunicação, armazenamento e forma de disponibilizar os dados coletados;
- Definição de uma forma de receber comandos externos e realizar uma malha de controle das variáveis de ambiente controláveis.

1.3.2 Objetivos Específicos

Este trabalho tem como objetivo específico, sugerir uma solução a ser utilizada para monitoramento e controle das variáveis de ambiente do *data center* do PoP-ES/RNP,

compreendendo:

- Definição de protocolo de comunicação entre microcontroladores e ponto central da rede de monitoramento/controle;
- Definição do *software* utilizado para comunicação;
- Programação de um protótipo de servidor central de monitoramento da rede;
- Definição de *software* para armazenamento de dados coletados;
- Definição do método de disponibilização de dados coletados de maneira padronizada e integrável com outros sistemas utilizados pelo PoP-ES/RNP;
- Implementação de um protótipo monitorando a corrente de um dos circuitos elétricos no QDC do *data center* do PoP-ES/RNP;
- Análise e estudo de escalabilidade dos componentes de *software* utilizados.

1.4 Metodologia

A metodologia adotada nesse trabalho consiste em i) elaboração de um protótipo como prova de conceito, ii) coletar medidas permitindo avaliar alguns parâmetros relevantes quanto à escolha dos *softwares* utilizados e iii) avaliar a escalabilidade da solução. Os experimentos foram realizados sobre uma infraestrutura isolada, de maneira a não impactar a operação do *data center* do PoP-ES/RNP.

1.5 Estrutura da Monografia

No Capítulo 2 são abordadas informações iniciais, assim como toda a teoria necessária para o entendimento do trabalho realizado, conceitos de IoT utilizados e conceitos relacionados às decisões tomadas durante o andamento do trabalho.

O Capítulo 3 apresenta a solução adotada, seus componentes e funcionamento. É apresentada também a implementação do protótipo da solução.

É apresentada uma série de testes e análises sobre a solução no Capítulo 4. Cada componente é testado e é feita uma análise sobre os resultados obtidos. As conclusões e possíveis trabalhos futuros são apresentados no capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem o objetivo de apresentar alguns conceitos base para o entendimento do trabalho desenvolvido, além de expor os fundamentos utilizados para as tomadas de decisão em seu decorrer.

2.1 Internet das coisas

A primeira proposta de arquitetura que visava integrar o mundo físico com o mundo virtual foi trazida por Engels et al. (2001), o que conhecemos hoje por Internet das Coisas (IoT - do inglês *Internet of Things*). Para cada dispositivo ou coisa seria entregue uma identificação única, que tinha como função identificar as informações sobre o objeto em questão, que poderia ser requisitada através de um serviço. Desta forma, tornou-se possível a integração de objetos ao mundo virtual.

Sethi e Sarangi (2017) definem IoT como um paradigma onde cada objeto é equipado com sensores, atuadores e processadores, comunicando-se entre si a fim de servir um propósito. Isto refere-se a um novo tipo de mundo, onde todos os dispositivos e aparelhos utilizados estarão conectados a uma rede. Pode-se então utilizá-los colaborativamente para executar atividades complexas de alto nível de inteligência.

Aplicações em projetos de cidades inteligentes são um exemplo de aplicação do conceito de IoT. Grandes cidades vêem a necessidade de automação de seus serviços de infraestrutura, onde tem-se como exemplo, o sensoriamento geográfico do serviço de transporte público, onde através de uma interface os usuários, cidadãos, conseguem saber a estimativa de tempo de espera para a chegada de um ônibus.

Existe uma certa incerteza quanto à segurança e privacidade quando se desenvolve um sistema IoT (FAROOQ et al., 2015). No contexto deste trabalho, utilizou-se como condições de contorno apenas a adoção de redes privadas para comunicação, porém para a futura implantação de atuadores no sistema deve-se implantar outros métodos além deste visando maior proteção do sistema.

2.1.1 Arquitetura

A arquitetura de sensoriamento e controle deve ser bem definida, de forma que a mesma satisfaça condições de: i) gerenciamento, ii) escalonamento e iii) segurança.

Para suprir tais requisitos, pretende-se utilizar um modelo de arquitetura como ilustrado na Figura 1.

Figura 1 – Esquema de arquitetura a ser utilizada.

Fonte: Adaptado de (SETHI; SARANGI, 2017).

2.1.1.1 Camada de Percepção

Compreende da camada física, onde os sensores coletam dados relacionados ao ambiente e os atuadores realizam acionamentos. Está diretamente relacionada ao tipo de sensoriamento e ao ambiente a ser instalado. Assim, a camada de percepção tem a função de realizar a coleta de dados do sistema.

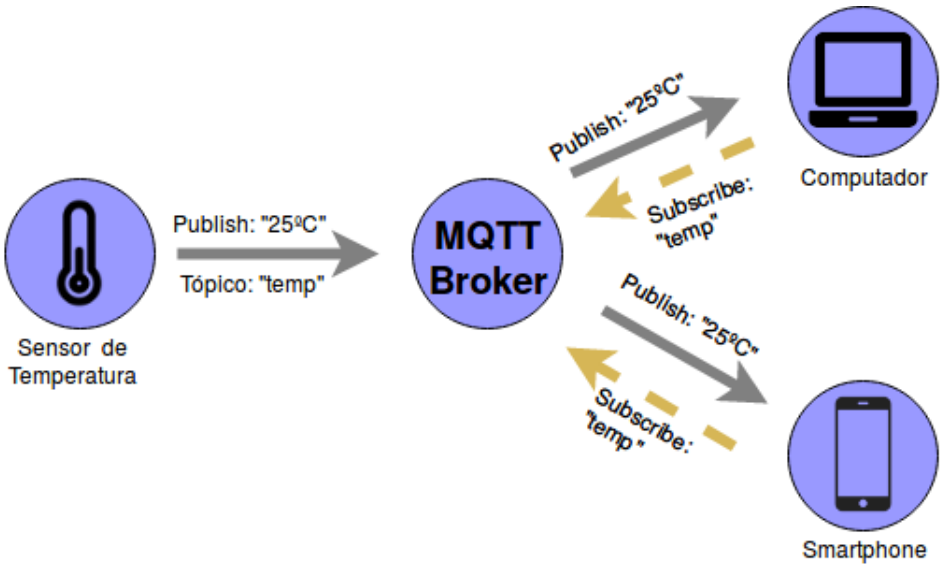
2.1.1.2 Camada de Rede

A camada de rede é responsável por conectar todos os sensores e o ponto central da rede, sendo assim responsável pela transmissão de dados do sistema. Em geral, sistemas de IoT tem como base o protocolo TCP/IP, e utilizam protocolos como: i) HTTP (Protocolo de Transferência de Hipertexto - do inglês *Hypertext Transfer Protocol*), o protocolo do tipo cliente/servidor mais comum de transferência de textos e arquivos na Internet, porém segundo Sethi e Sarangi (2017), por ser um protocolo muito detalhado, o mesmo gera uma sobrecarga na rede e nos microcontroladores, não sendo então adequado para sistemas com recursos limitados. ii) CoAP (do inglês *Constrained Application Protocol*), "um protocolo de transferência de web especializado para uso com nós restritos e redes restritas (por exemplo, de baixa potência e com perdas)" (IETF, 2014) CoAP é um protocolo do tipo cliente/servido, similar ao modelo HTTP, iii) SNMP (do inglês *Simple*

Network Management Protocol), um protocolo do tipo cliente servidor baseado em troca de mensagens UDP (SCHOFFSTALL; DAVIN, 1990) e iv) MQTT (do inglês *Message Queue Telemetry Transport*), um protocolo de mensagens leve para sensores e pequenos dispositivos móveis otimizado para redes TCP/IP não confiáveis ou de alta latência (OASIS, 2015).

O protocolo MQTT tem seu funcionamento baseado no método *publish/subcribe* baseado em tópicos. Portanto no MQTT existem dois tipos de conexão, "*publish*" e "*subscribe*". Para receber mensagens referentes a um tópico, um cliente deve se conectar ao *Broker* MQTT por uma mensagem do tipo *subcribe* com o tópico desejado e então, quando outro cliente enviar uma mensagem do tipo *publish* ao *Broker* neste tópico, o *Broker* se encarrega de encaminhar a todos os clientes inscritos no tópico em questão. Um exemplo de conexão pode ser visto na Figura 2.

Figura 2 – Modelo de conexão utilizando protocolo MQTT.



Fonte: Produção do próprio autor.

O *software* Mosquitto é um *broker* MQTT, o qual é um projeto *open source* implementado em linguagem C. O *software* Mosquitto possui um melhor desempenho quanto ao uso de memória e CPU quando comparado a outros *brokers* MQTT implementados em outras linguagens (TORRES; ROCHA; SOUZA, 2016).

2.1.1.3 Camada de Aplicação

A camada de aplicação é responsável pelo armazenamento e processamento dos dados adquiridos, deste modo, a camada de aplicação fica encarregada então de entregar ao usuário, serviços específicos (SETHI; SARANGI, 2017). Assim, nesta camada, são definidas as aplicações do sistema, como por exemplo, casas inteligentes, cidades inteligentes ou como no caso deste trabalho, o monitoramento de sensores em um *data center*.

2.1.2 Comunicação M2M

Os avanços da tecnologia têm permitido cada vez mais a automatização do monitoramento e controle de ambientes e sistemas, assim, o termo máquina a máquina (M2M - do inglês *Machine to Machine*) refere-se a tecnologias utilizadas para conexão de sistemas tanto com fio como sem fio com outros dispositivos que possuam a mesma habilidade. Em geral, o termo comunicação M2M é utilizado para definir comunicações com pouca ou nenhuma intervenção humana. Em geral ocorre entre vários dispositivos terminais, como computadores, sistemas embarcados, dispositivos móveis, entre outros (CHEN et al., 2014).

2.2 *Hardware*

Um microcontrolador é um computador em um chip ou, se você preferir, um computador de chip único. A palavra "micro" sugere que o dispositivo é pequeno e o "controlador" diz que o dispositivo pode ser usado para controlar objetos, processos ou eventos. Outro termo para descrever um microcontrolador é o controlador embarcado, porque o microcontrolador e seus circuitos de suporte são geralmente incorporados ou embarcados nos dispositivos que controlam (AXELSON, 1994).

O módulo Wemos D1 mini é um microcontrolador que tem como base o ESP8266, contendo um processador de arquitetura RISC de 32bits da fabricante Tensilica modelo L106 que opera a 80MHz. O microcontrolador em questão possui 32KBytes de memória RAM para instruções e 96KBytes de memória RAM para dados, além de 4MBytes de memória *flash*, possui também um módulo WiFi embutido e pinos GPIO , I2C, SPI e saída PWM (Espressif Systems, 2018).

Juntamente com o microcontrolador foi utilizado um módulo conversor analógico/digital (A/D) para realizar leituras de sensores analógicos, utilizou-se o ADS1115, da

fabricante Texas Instruments. O módulo possui 4 entradas analógicas, onde para cada sensor utiliza-se 2 entradas de forma diferencial para evitar interferências, um amplificador de ganho programável e 15 bits de precisão (TEXAS INSTRUMENTS, 2009).

2.2.1 Software Embarcado

O *software* embarcado no microcontrolador foi implementado através da Arduino IDE, a qual o fabricante do *hardware* oferece as bibliotecas utilizadas e permite a programação a partir da linguagem C (Arduino, 2018). Porém, por tratar-se de uma IDE de código fechado, torna o *software* embarcado pouco obscuro, não possibilitando maiores definições de prioridades quanto a requisitos de acesso a *hardware* e performance. Portanto na fase de implementação de atuadores, faz-se necessária a adoção de uma abordagem diferente quanto à programação do *software* embarcado.

2.3 Serviços Web

Segundo Rao e Su (2004), serviços Web são aplicativos modulares independentes e auto-descritivos que podem ser publicados e acessados através da Web. Os serviços Web em geral utilizam protocolo HTTP para comunicação.

Serviços Web em geral possuem uma API, que é um conjunto de padrões, rotinas e programações para acesso a plataformas ou serviços Web. Os principais tipos de API são Protocolo Simples de Acesso a Objetos (SOAP - do inglês *Simple Object Access Protocol*) e Transferência de Estado Representacional (REST - do inglês *Representational State Transfer*). Segundo (MUMBAIKAR; PADIYA, 2013) a estrutura REST quando comparada à SOAP possui um menor consumo de recursos e maior flexibilidade quanto a estrutura dos dados. A estrutura REST pode utilizar diferentes formatos de dados, incluindo Notação de Objetos JavaScript (JSON - do inglês *JavaScript Object Notation*). Já a estrutura SOAP troca dados apenas em formato Linguagem de Marcação Extensível (XML - do inglês *Extensible Markup Language*).

2.3.1 API REST

Fielding e Taylor (2000) descreveram e nomearam um estilo de arquitetura que evitaria uma crise de escalabilidade nos serviços Web, este estilo de arquitetura foi nomeado

REST. Para a compreensão da arquitetura REST é necessário conhecer a definição de três termos:

- Recurso: qualquer coisa pode ser definida como um recurso, um objeto ou um conceito abstrato. Se o objeto tratado refere-se a algo que deve ser definido, então trata-se de um recurso;
- Estado: REST utiliza dois estados, estado de recurso, que refere-se a uma informação sobre o recurso e estado de aplicação que refere-se ao caminho que o cliente utilizou para chegar na aplicação;
- Representação: alguma informação útil sobre o estado do recurso.

Estados de recursos ficam no servidor, já os estados de aplicação são de responsabilidade do cliente. Desta forma, segundo Feng, Shen e Fan (2009), a arquitetura REST fornece um conjunto de restrições que, quando aplicadas, enfatizam a escalabilidade das interações dos componentes, a generalidade das interfaces, a implantação independente de componentes e os componentes intermediários para reduzir a latência da interação, reforçar a segurança e encapsular os sistemas legados.

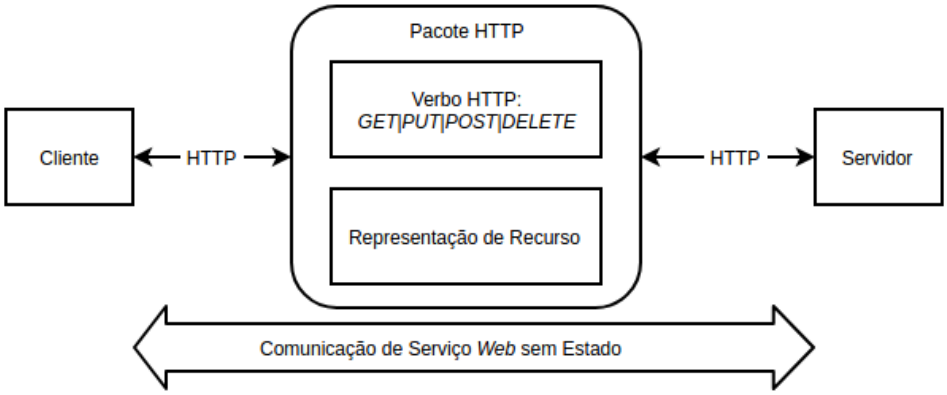
API REST é um conjunto de rotinas e padrões definidos que utilizam o padrão REST. Sua implementação é estruturada em:

- URI: Identificador Uniforme de Recursos (URI - do inglês *Uniform Resource Identifier* é um conjunto de caracteres usado para identificar ou denominar recursos na Internet. O tipo mais comum é o Localizador-Padrão de Recursos (URL - do inglês *Uniform Resource Locator*) que tem a seguinte estrutura:

  `protocolo://domínio:porta/caminho/recurso`;
- Protocolo HTTP: Utiliza requisições HTTP com os métodos GET, PUT, POST e DELETE para, respectivamente, ler, editar, criar e apagar dados no servidor.
- Formato de dados: Compreende ao formato de dados utilizado pela API.

A Figura 3 mostra uma ilustração de um serviço Web de arquitetura REST. O cliente interage com o servidor através de uma interface uniforme. Durante a interação, o servidor e o cliente trocam representações de recursos.

Figura 3 – Modelo de serviço Web com arquitetura REST.



Fonte: Adaptado de Feng, Shen e Fan (2009)

Segundo Nurseitov et al. (2009), os formatos de dados mais utilizados na indústria são JSON e XML. Ambas as estruturas possuem estrutura de fácil compreensão tanto para máquinas quanto para humanos. Porém, ainda segundo Nurseitov et al. (2009), o desempenho quanto a velocidade de tradução de dados e uso de recursos é melhor utilizando-se o formato JSON, quando comparado ao XML.

O formato de texto JSON tem a estrutura demonstrada no Quadro 1. Onde os carateres "{ }" delimitam objetos e os caracteres "[ ]" delimitam vetores.

Quadro 1 – Modelo de formato JSON.

```
1 {
2     "<nome_1>": {
3         "<nome_2>": valor_2,
4         "<nome_3>":[valor_3_1, valor_3_2,...,valor3_n]
5     }
6 }
```

Fonte: Produção do próprio autor.

2.3.2 Servidor Web

O termo servidor Web pode referir-se tanto a *hardware* quanto a *software*. Neste trabalho o termo será utilizado exclusivamente para *software* com clientes utilizando protocolo HTTP.

A função primária de um servidor Web é armazenar, processar e entregar páginas para clientes. As requisições HTTP enviadas pelo cliente são processadas e então uma

resposta é entregue ao cliente. As páginas mais frequentes são páginas de Linguagem de Marcação de Hipertexto(HTML - do inglês *HyperText Markup Language*), porém podem ser imagens, planilhas e *scripts* (DALZIEL, 2014).

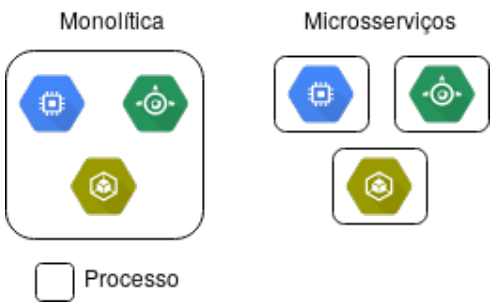
Os servidores Web mais utilizados atualmente são Apache HTTP Server e Nginx. Quando comparados somente os sites mais acessados, o uso do Nginx é superior ao uso do Apache (W3techs, 2018). Segundo Cisneiros e Ramos (2015) o Nginx tem um melhor desempenho quando comparado ao Apache em relação ao número de conexões simultâneas.

2.4 Arquitetura de microsserviços

Microsserviços é um estilo de arquitetura baseada na divisão de sistemas em partes pequenas e leves que servem a um propósito específico e coeso no escopo do sistema. É considerada uma evolução do estilo de arquitetura orientada a serviços (ALSHUQAYRAN; ALI; EVANS, 2016). Cada serviço funciona em seu próprio processo e se comunica muitas vezes através de uma API com recursos HTTP. Desta forma a programação de cada serviço é independente, podendo-se utilizar linguagens e estruturas de programação diferentes para cada serviço. Devido à separação provida pela arquitetura, o gerenciamento é realizado de maneira separada.

A arquitetura monolítica é uma abordagem contrária à abordagem por microsserviços, como pode ser notado na Figura 4. Um sistema de arquitetura monolítico implica em um único código base/repositório que oferece dezenas ou centenas de serviços usando interfaces como HTML e *Web services* (VILLAMIZAR et al., 2015).

Figura 4 – Modelo de arquitetura monolítica e microsserviços.



Fonte: Adaptado de (TEIXEIRA, 2016)

2.4.1 Virtualização em *containers* Docker

Containers são uma abstração na camada de aplicações que agrupam um conjunto de códigos e dependências. Em geral, os *containers* ocupam menos espaços, têm um tempo de inicialização menor quando comparados a máquinas virtuais (VM - do inglês *Virtual Machines*) (Docker Inc., 2018). Docker é um projeto de código aberto que possui uma plataforma para *containers*, fornecendo uma maneira de executar praticamente qualquer aplicação de forma padronizada e isolada. Este isolamento permite que múltiplas aplicações estejam em execução simultaneamente em uma mesma máquina. A Figura 5 faz uma comparação da execução de *containers* Docker e VMs.

Figura 5 – Estrutura de virtualização de *containers* Docker e VMs

(a) Máquinas Virtuais. (b) *Containers.*

Fonte: Adaptado de (Docker Inc., 2018).

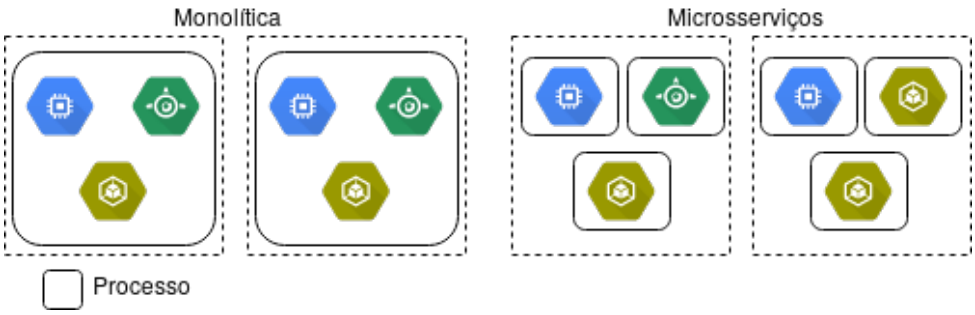
Segundo Celesti et al. (2016), o uso de *containers* é uma alternativa mais leve quando comparado a virtualização com VMs, tendo como maiores benefícios a flexibilidade e escalabilidade, tornando-se assim interessante para utilização em implementações baseadas em IoT. Os *containers* podem ser instanciados de acordo com a demanda, o que permite um melhor gerenciamento de recursos. Outra vantagem é a possibilidade de instanciação de *containers* em *hardware* de baixo custo como Raspberry Pi, o que se torna inviável no caso de VMs.

2.4.2 Vantagens

Quanto à operação, a principal vantagem da arquitetura de microsserviços é a escalabilidade. Como visto na Figura 6, para escalonar uma arquitetura monolítica temos que escalonar todos os seus componentes, mesmo os que estão subutilizados, aumentando o uso de recursos de maneira desnecessária. Já na arquitetura de microsserviços, devido

seu aspecto de independência de serviços, é possível replicar apenas os serviços necessários, deixando outros intocados.

Figura 6 – Modelo de escalonamento de arquitetura monolítica e microsserviços.



Fonte: Adaptado de (TEIXEIRA, 2016)

Há também, na operação, a vantagem quanto a falhas. Em caso de falha de um serviço, apenas aquele serviço é afetado. Já em arquiteturas monolíticas o funcionamento de um serviço pode afetar toda a aplicação.

Com relação ao desenvolvimento, existe a vantagem que cada serviço pode ser desenvolvido independentemente tornando mais fácil a compreensão do código por se tratar de blocos menores de código.

2.4.3 Desvantagens

A principal desvantagem oferecida pela arquitetura de microsserviços está na comunicação entre serviços. Devido ao fato destes serviços executarem de maneira isolada, a comunicação entre os mesmos deve ser feita através da rede, em geral via protocolo HTTP, há então uma perda de desempenho devido à serialização de dados além de um uso de rede elevado. Outro aspecto a ser considerado é a segurança, pois ao realizar transmissão de dados pela rede insere-se pontos de vulnerabilidade no sistema (Philipp Hauer, 2015).

2.5 Banco de dados de séries temporais

Dados temporais são simplesmente medidas ou eventos rastreados, monitorados, amostrados e agregados através do tempo. A chave para diferenciar dados temporais e outros tipos de dados é que quando se trata de dados temporais, se está sempre fazendo perguntas com relação ao tempo (DIX, 2016). Assim, Bancos de Dados de Séries Temporais

(TSDB - do inglês *Time-Series Database*) são otimizados para consultas relacionadas a espaços de tempo (DUNNING; FRIEDMAN, 2014).

Segundo Boncea e Bacivarov (2016), um TSDB para aplicação em IoT deve satisfazer os seguintes tópicos: i) rotular e marcar dados, ii) rótulos devem ser indexáveis permitindo a aplicação de filtros, iii) otimizado para escrita intensa e poucas mudanças, iv) armazenamento comprimido, v) integração via API HTTP.

O InfluxDB é um TSDB que além de contemplar os pontos citados acima, possui uma linguagem de consulta parecida com a linguagem SQL e uma otimização para respostas de consultas frequentes (BONCEA; BACIVAROV, 2016). Assim, para uma integração M2M, onde em geral as consultas são realizadas de maneira padronizada, o InfluxDB torna-se um atrativo.

3 SOLUÇÃO PROPOSTA

A solução proposta foi implementada de maneira a ser dividida em dois serviços principais, o serviço de leitura de dados e o serviço de interface API. Desta forma, em uma manobra do sistema, é possível parar apenas uma parte sem afetar diretamente a outra. Para realizar tal separação, optou-se por uma arquitetura então baseada em microsserviços, assim cada microsserviço consegue trabalhar de maneira isolada, a fim de se conseguir um escalonamento horizontal, ou microsserviços de coleta e armazenamento de dados distribuídos onde os dados são acessados apenas através de um servidor central.

Pela implantação da solução com a utilização de *containers* Docker, fica a critério do administrador implementar a arquitetura de maneira distribuída em várias máquinas ou apenas uma com todos os microsserviços. Tal implementação garante a escalabilidade da arquitetura.

A adoção de um método de comunicação do tipo *publish/subscribe* permite que o sentido de comunicação do servidor de coleta de dados com os microcontroladores ocorra de maneira genérica, possibilitando até que esta comunicação ocorra quando os microcontroladores estiverem conectados a redes NAT[1], simplificando a configuração de rede do sistema.

O esqueleto da arquitetura tem como preceito exigir um mínimo de recursos de *hardware* do *data center* e permitir que a escolha de microcontroladores e sensores fique a critério do usuário, desde que obedeçam a requisitos mínimos. A implementação compreende também uma otimização da interação M2M onde os dados sejam disponibilizados aos usuários através de uma API.

3.1 Arquitetura

Seguindo os preceitos citados anteriormente, o serviço de monitoramento ficou subdividido em cinco microsserviços, sendo estes:

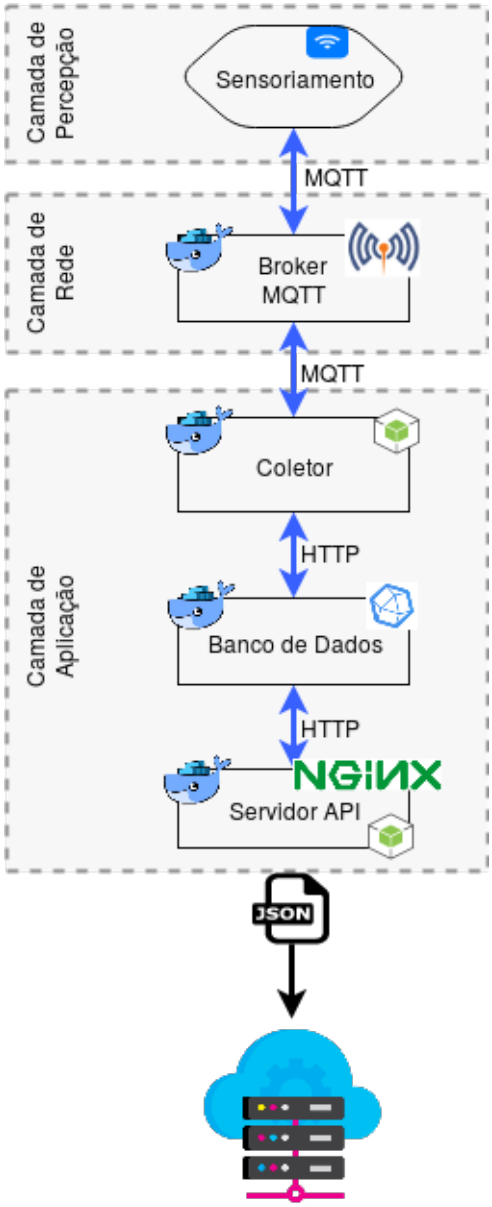
1. Sensoriamento;
2. Comunicação Microcontroladores x Coletor;
3. Coletor;

[1] NAT (do inglês - *Network Address Translation*) é uma técnica que consiste em reescrever os endereços de origem de um pacote permitindo que redes internas tenham acesso a redes externas (EGEVANG; FRANCIS, 1994)

4. Banco de Dados;

5. Servidor API.

Figura 7 – Modelo de arquitetura proposta.



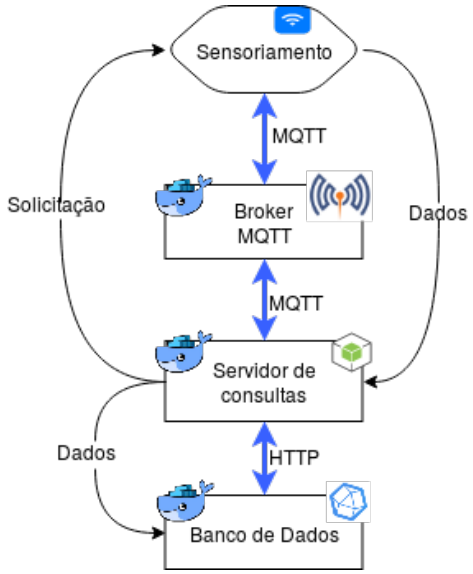
Fonte: Produção do próprio autor.

A Figura 7 ilustra a arquitetura adotada. O coletor, responsável por gerenciar os intervalos de leitura dos sensores, comunica-se através de um tópico de controle no *broker* MQTT com o serviço de sensoriamento, fazendo a solicitação de leitura de sensores e recebendo os dados referentes às leituras solicitadas em diversos tópicos, onde cada tópico corresponde a um sensor. Após receber o dado solicitado, o coletor envia o dado ao banco

de dados para ser armazenado. A API, ao receber uma requisição HTTP comunica-se com o banco de dados buscando os dados solicitados na requisição, estrutura os dados em um JSON e responde a requisição inicial com o JSON montado.

O fluxo de informação na coleta de dados pode ser visto na Figura 8, onde o servidor de consultas envia solicitações aos microcontroladores e recebe dados dos mesmo, enviando-os antão ao banco de dados para serem armazenados.

Figura 8 – Fluxo de informação no serviço de coleta de dados.



Fonte: Produção do próprio autor.

3.1.1 Sensoriamento

O protótipo sugerido foi testado com a medição de corrente dos circuitos de um dos quadros elétricos que abastece o *data center* do PoP-ES/RNP. Para a realização das medidas, foi adotado o seguinte *hardware*:

- Sensor: SCT013-30A da fabricante YHDC é do tipo transformador de corrente portanto capaz de medir apenas em regime de corrente alternada (CA). Para a determinação do sensor foram avaliados: i) corrente de pico no circuito, ii) custo de aquisição.

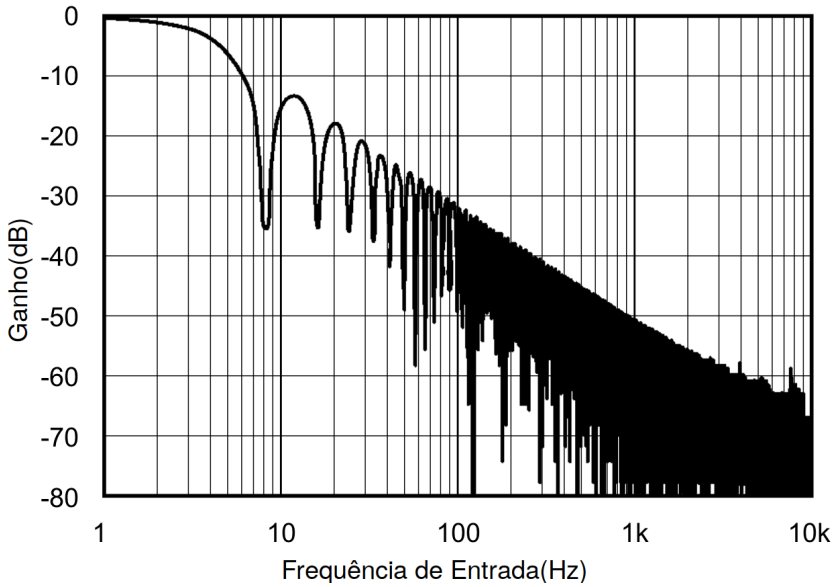
- Microcontrolador: D1 mini da fabricante Wemos possui um chip ESP8266, módulo WiFi embutido e suporte a comunicação via barramento I2C. Para o microcontrolador,

foram avaliados: i) comunicação via WiFi, ii) suporte a comunicação I2C, iii) custo de aquisição.

- Conversor Analógico-Digital (A/D): ADS1115 da fabricante Adafruit possui 15 bits de precisão e comunicação via barramento I2C. Para a escolha do conversor A/D, foram avaliadas: i)precisão, ii)comunicação via barramento I2C.

O conversor A/D utilizado possui uma atenuação de entrada como pode ser visto pela Figura 9. Para ter uma melhor estimativa desta atenuação o conversor A/D foi testado com o auxílio de um gerador de função e um osciloscópio na frequência de 60Hz, correspondente à frequência fundamental da rede elétrica, e obteve-se então a atenuação inserida na entrada do conversor A/D. A partir da determinação desta atenuação, foi inserido um ganho via *software* no microcontrolador, corrigindo então o ganho negativo inserido pelo conversor A/D.

Figura 9 – Atenuação de entrada do conversor A/D.

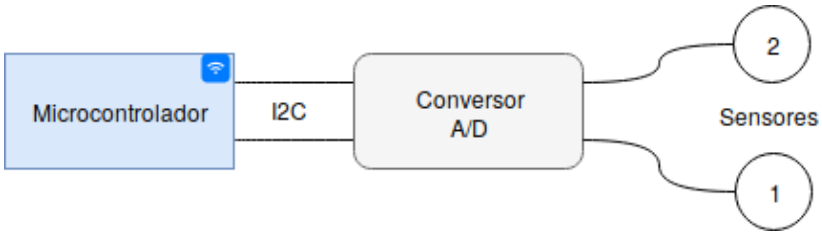


Fonte: Adaptado de Texas Instruments (2009).

A Figura 10 mostra um modelo de *hardware* para sensoriamento, onde um microcontrolador com conexão WiFi conecta-se via barramento I2C com um conversor A/D ligado a sensores analógicos.

É importante ressaltar neste ponto que, a escolha dos sensores e microcontroladores pode ser feita de acordo com a preferência do administrador desde que satisfeitas as condições citadas acima.

Figura 10 – Esquemático de sensoriamento adotado.



Fonte: Produção do próprio autor.

A implementação da leitura dos sensores de corrente foi realizada de modo que o microcontrolador seja capaz de informar sempre a corrente RMS. Por se tratar de correntes AC, a estratégia adotada foi: a cada solicitação feita pelo serviço de coleta de dados, o microcontrolador realiza 1000 leituras do sensor de corrente correspondente e assim calcula o valor RMS da corrente no circuito.

3.1.2 Comunicação Microcontroladores x Coletor

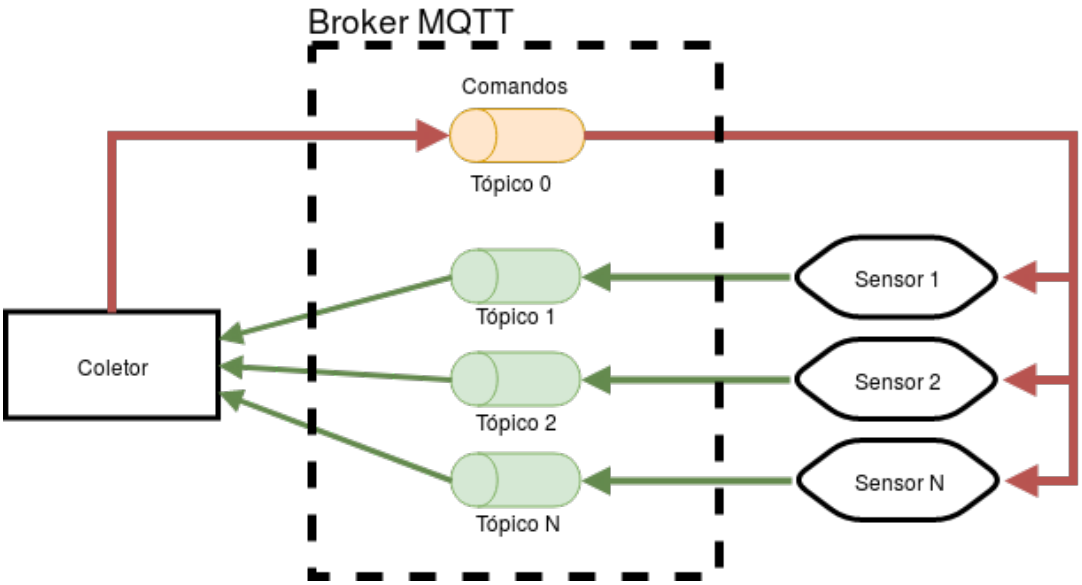
A comunicação dos microcontroladores e o coletor é realizado através de uma rede TCP/IP, onde se utiliza do protocolo de comunicação MQTT pois segundo Yokotani e Sasaki (2017) tem um menor *overhead* de mensagens e facilidade de comunicação um para muitos e muitos para muitos, além de enfileirar as mensagens para que sejam consumidas pelo microcontrolador.

A Figura 11 demonstra como é realizada a comunicação entre o coletor e o sensoriamento, onde um tópico é utilizado para enviar mensagens de comando, informando qual sensor deve ser lido, ao serviço de sensoriamento, por sua vez, o serviço de sensoriamento publica em tópicos relacionados a cada sensor pelos os quais o coletor recebe os dados. Desta forma, o número de tópicos no *broker* está diretamente relacionado ao número de sensores conectados ao sistema.

3.1.3 Coletor

O coletor é responsável por solicitar aos microcontroladores a leitura das medidas dos sensores, receber os dados referentes à leitura dos sensores e armazenar no banco de dados. Ao tornar o coletor responsável por solicitar os dados aos microcontroladores, é possível manter as configurações de intervalo de consultas centralizadas, facilitando ao administrador a configuração do sistema. A comunicação com os microcontroladores é

Figura 11 – Fluxo de mensagens no *broker* MQTT.



Fonte: Produção do próprio autor.

feita via mensagens MQTT. Já em relação ao banco de dados a comunicação é realizada através do protocolo HTTP/HTTPS.

Para implementar o coletor foi utilizada a linguagem JavaScript com o interpretado Node.js. O Node.js é um interpretador JavaScript assíncrono, orientado a eventos projetado para construir aplicações Web de maneira escalável (Node.js Foundation, 2018). com Node.js várias conexões podem ser tratadas ao mesmo tempo, podendo tratar as respostas dos microcontroladores de maneira imediata.

3.1.4 Banco de Dados

O banco de dados utilizado no protótipo foi o InfluxDB, um banco de dados de séries temporais onde a linguagem de consulta é muito similar a linguagens SQL, além de possuir uma API de comunicação simplificada com comunicação via protocolo HTTP/HTTPS.

3.1.5 Servidor API

O servidor API foi implementado em JavaScript, disponível via HTTP/HTTPS. O servidor é acessado através de um *proxy* criado por um servidor Web Nginx, adicionado ao sistema para obter-se uma camada a mais de segurança e visando facilitar a configuração do serviço ao usuário.

As consultas realizadas na API devem seguir um padrão previamente definido. As respostas referentes a estas consultas seguem também um padrão estruturado em forma de JSON, pois além de ser amplamente utilizado e facilmente interpretado como dito na Seção 2.3.1, é o formato nativo da linguagem utilizada na implementação da API.

O servidor API opera de maneira simples, com os tipos de consulta de dados padronizados via método GET do protocolo HTTP/HTTPS:

- Última leitura de todos os sensores do sistema, acessando a página raiz (/);
- Última leitura de todos os sensores ligados a um microcontrolador específico (/microcontroladorID);
- Última leitura de um sensor específico (/microcontroladorID/sensorID);
- Todas as leituras de um sensor específico após um lugar no tempo (/microcontroladorID/sensorID/inicioIntervalo)
- Todas as leituras de um sensor específico em um intervalo específico (/microcontroladorID/sensorID/inicioIntervalo/fimIntervalo).

Ao receber uma das solicitações acima o servidor realiza a pesquisa no banco de dados e retorna um JSON estruturado de acordo com o Quadro 2.

Quadro 2 – Modelo de resposta em formato JSON.

```
1 {
2     "<microcontroladorID>": {
3         "<sensorID>": [{
4             "time": "dd-mm-yyyy hh:mm:ss",
5             "measure": <medida>
6         }]
7     }
8 }
```

Fonte: Produção do próprio autor.

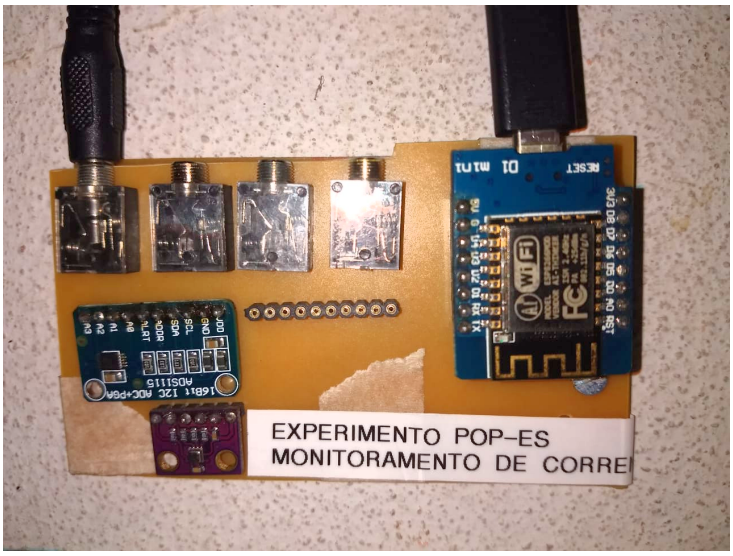
3.2 Protótipo

Como prova de conceito, foi implementado um protótipo com o sensoriamento de corrente em um dos circuitos do QDC onde ficam os aparelhos de refrigeração do

data center do PoP-ES/RNP. Foi construída uma placa de circuito impresso utilizando o *hardware* especificado na Seção 3.1.1 e uma VM com 500MB de memória RAM, 20GB de disco e 1vCPU.

A placa de circuito impresso construída foi elaborada para até 4 sensores analógicos a serem conectados via conector modelo P2. A Figura 12 mostra a placa construída. Ela é composta por um microcontrolador, dois conversores A/D de 4 canais cada e um *shield* com sensor de temperatura umidade e pressão. Por se tratar de sensores de corrente, cada conversor A/D consegue conectar apenas 2 sensores, pois é necessário que a leitura seja feita de forma diferencial, assim cada sensor utiliza de 2 canais do conversor A/D.

Figura 12 – Placa de circuito impresso construída.



Fonte: Produção do próprio autor.

4 EXPERIMENTOS E RESULTADOS

Neste capítulo será realizada uma avaliação da solução implementada. Para avaliar a solução foram realizados experimentos nos principais serviços da arquitetura, assim como em cada um de seus microsserviços. A partir do *hardware* e *softwares* definidos, os testes referentes a microsserviços embarcados em *containers* Docker foram realizados em uma VM minimalista como especificado no Capítulo 3.

4.1 Avaliação do serviço de aquisição de dados

Para avaliarmos o serviço de aquisição de dados, foram realizados testes em cada um dos microsserviços que compõem este serviço. Visto que cada microsserviço trabalha de maneira isolada e desta forma podemos ter uma boa percepção do sistema como um todo.

4.1.1 Tempo de resposta do sensoriamento

A avaliação deste serviço é medida pelo tempo de resposta do microcontrolador para cada solicitação feita. Para obtenção dos dados foram realizadas mil solicitações ao sistema de sensoriamento através de uma máquina externa, o tempo de resposta considerado compreende do momento quando é feita a solicitação até o momento onde há a resposta do microcontrolador. O tempo médio de resposta para leitura da corrente de um circuito, como pode ser observado na Tabela 1 foi de 9,10s com um desvio padrão de 0,11s. Este tempo de resposta é justificado pois como dito no capítulo anterior, o microcontrolador precisa coletar várias amostras para calcular o valor RMS da corrente antes de enviar a resposta.

Tabela 1 – Tempo de resposta do sensoriamento de corrente.

Tempo de resposta(s)	Desvio padrão(s)
9,1	0,11

Fonte: Produção do próprio autor.

As amostras são coletadas em uma frequência de 128 amostras por segundo devido às frequência de amostragem do conversor A/D. O conversor A/D utilizado permite uma maior frequência de amostragem, porém, a partir de 128 amostras por segundo é inserido

um erro de amostragem no sistema. Como ideia da solução é intervalos de consulta entre 1 e 5 minutos, o tempo de resposta obtido na Tabela 1 atende aos requisitos da solução.

Visto que a arquitetura proposta permite flexibilidade quanto aos sensores e microcontroladores utilizados, não é escopo deste trabalho realizar maiores testes quanto à parte do microcontrolador.

4.1.2 Serviço *broker*

O protocolo MQTT mostrou-se de grande importância na solução proposta, visto que o mesmo enfileira as mensagens para serem consumidas pelo microcontrolador. Caso as requisições enviadas ao microcontrolador tivessem que ser consumidas de maneira instantânea, algumas solicitações poderiam não ser processadas, devido a uma demora de alguns segundos para certos tipos de leitura realizados pelo microcontrolador.

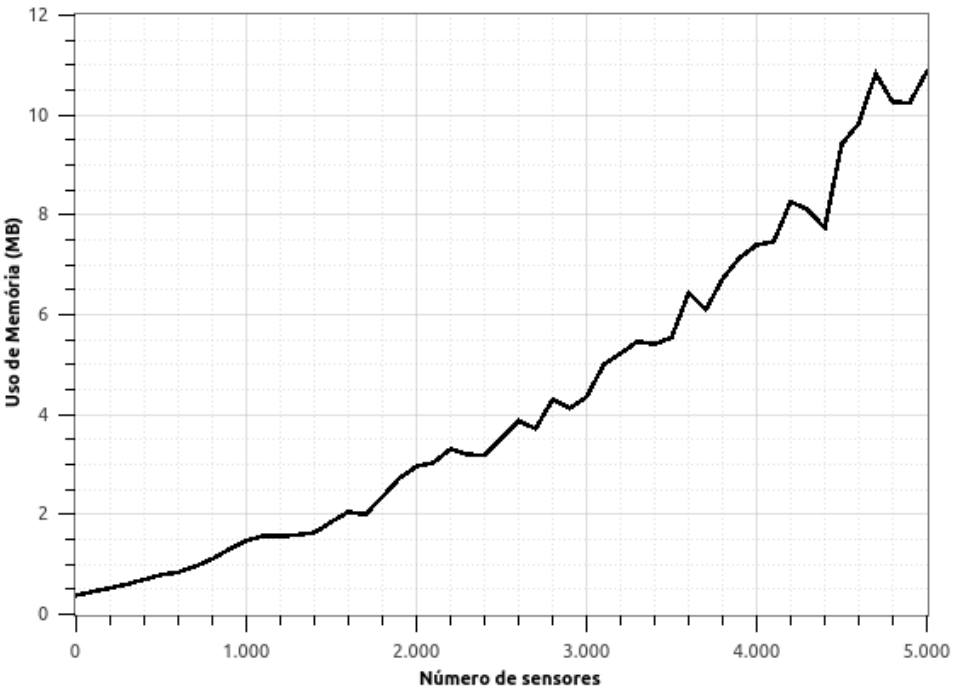
Com relação ao *broker* MQTT, foram levantados dados quanto a sua escalabilidade, visto que cada sensor ou microcontrolador adicionado ao sistema corresponde a um *subscribe* em um tópico no *broker*. Assim, a escalabilidade do mesmo foi testada comparando o uso de CPU e memória RAM em relação ao crescimento de *subscribers* em tópicos distintos no MQTT.

Para a identificação de requisitos mínimos de *hardware* para a adequação de recursos, foi realizado teste constituindo de uma avaliação do *broker* MQTT com relação ao número de *subscribers*, ou seja, diretamente relacionado ao número de sensores agregados à rede. Para esta avaliação, foi levantada uma instância do *broker* MQTT na VM de testes e a partir de um máquina terceira conectada à mesma rede, foram realizados os testes.

O uso de memória do *broker* em relação ao número de sensores no sistema é mostrado pela Figura 13. Pode-se observar que o uso de memória pelo *broker* de uma forma exponencial em relação ao número de sensores conectados. Apesar de o uso de memória crescer em relação ao número de sensores, para até 5000 sensores o uso de memória pelo *broker* ainda é menor que 15MB, condizendo com a aplicação em uma arquitetura minimalista.

Já o uso de CPU, como demonstrado na Figura 14, ocorre um salto com aproximadamente 3400 sensores conectados. Este comportamento não é esperado em nos microsserviços que compõem a solução, interferindo assim na escalabilidade do sistema. Porém, como o *broker* apresentou baixo uso de CPU para até 3300 sensores conectados,

Figura 13 – Uso de memória do *broker* MQTT em relação ao número de *subscribers*.



Fonte: Produção do próprio autor.

com menos de 5% de uso de CPU, pode-se então utilizar da escalabilidade da arquitetura de microsserviços para instanciação de maior número de sensores. Recomenda-se então, conectar até 2000 sensores a cada instância de *broker* MQTT, tendo assim uma margem maior até o ponto de salto no uso de CPU.

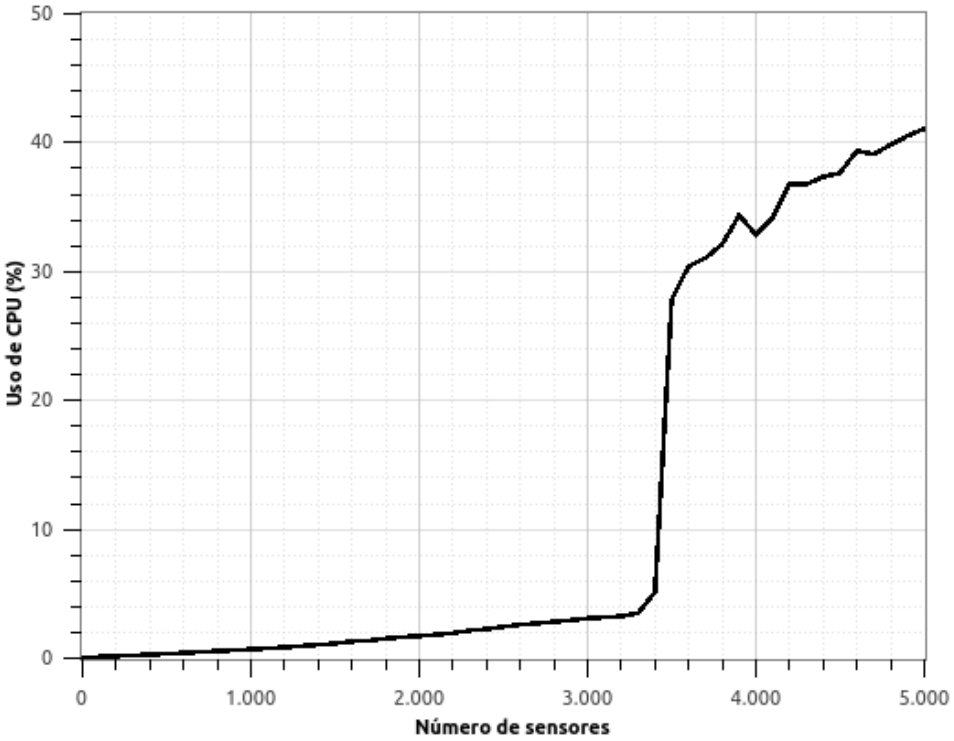
Não há uma certeza quanto ao motivo do salto observado na Figura 14, porém tal característica pode estar vinculada à virtualização de CPU ou algum aspecto de implementação de *software*.

4.1.3 Coletor

Para uma análise dos requisitos de *hardware* necessários para o coleotr é necessária a implementação de um método de configuração/adição de sensores, o qual não é escopo deste projeto. Hoje, a adição de novos sensores deve ser feita com modificação direta no código do coletor. Assim, não foi possível uma avaliação deste serviço com relação aos requisitos de *hardware* necessários em relação ao número de sensores conectados.

Por outro lado, com relação à funcionalidade do serviço, foi comprovada sua eficácia ao gerenciar o intervalo de consultas, enviar requisições e receber dados dos

Figura 14 – Uso de CPU do *broker* MQTT em relação ao número de *subscribers*.



Fonte: Produção do próprio autor.

microcontroladores e inserir os dados no banco de dados.

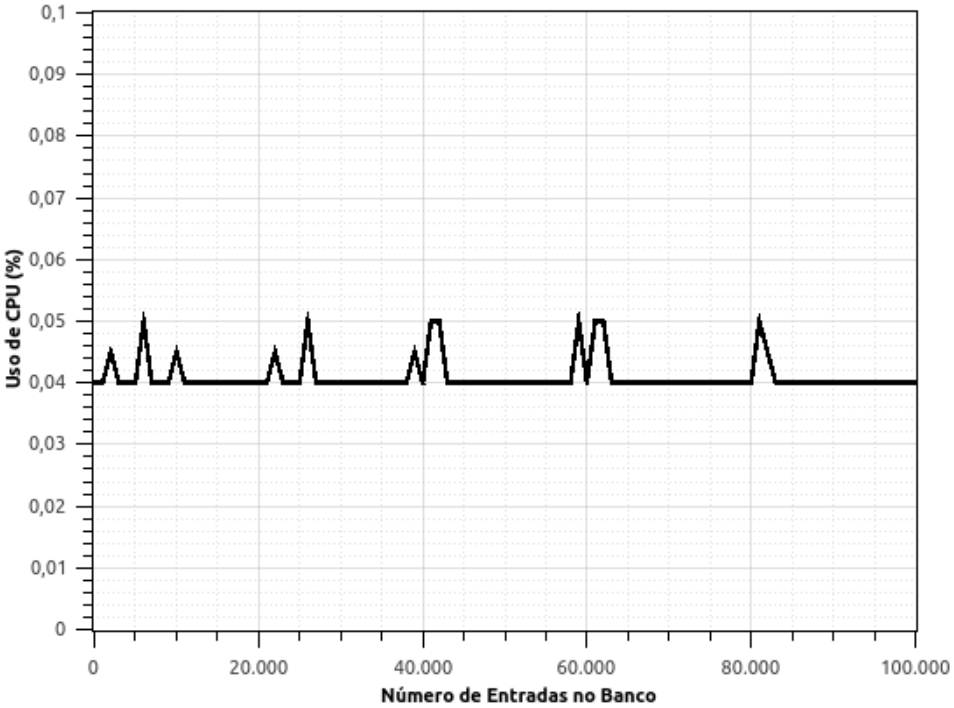
4.1.4 Serviço de Armazenamento

Para o serviço de armazenamento, foram analisados aspectos do uso de recursos (memória, CPU e ocupação de disco) em relação à quantidade de entradas no banco. Os testes foram realizados através de máquinas externas inserindo os dados no banco de dados.

Na análise, foram executados testes comparando o uso de recursos em relação à quantidade de medidas armazenadas no banco de dados. A variação de entradas no banco de dados utilizada corresponde a aproximadamente o número de entradas de um sensor em uma janela de um ano com intervalo de coleta de dados de 5 minutos.

O uso de CPU quando comparado ao número de entradas no banco permanece constante e em níveis abaixo de 0,1% como pode ser observado na Figura 15. Este comportamento é ideal para a implantação em uma VM minimalista onde os recursos são limitados.

Figura 15 – Uso de CPU do InfluxDB em relação ao número de entradas no banco



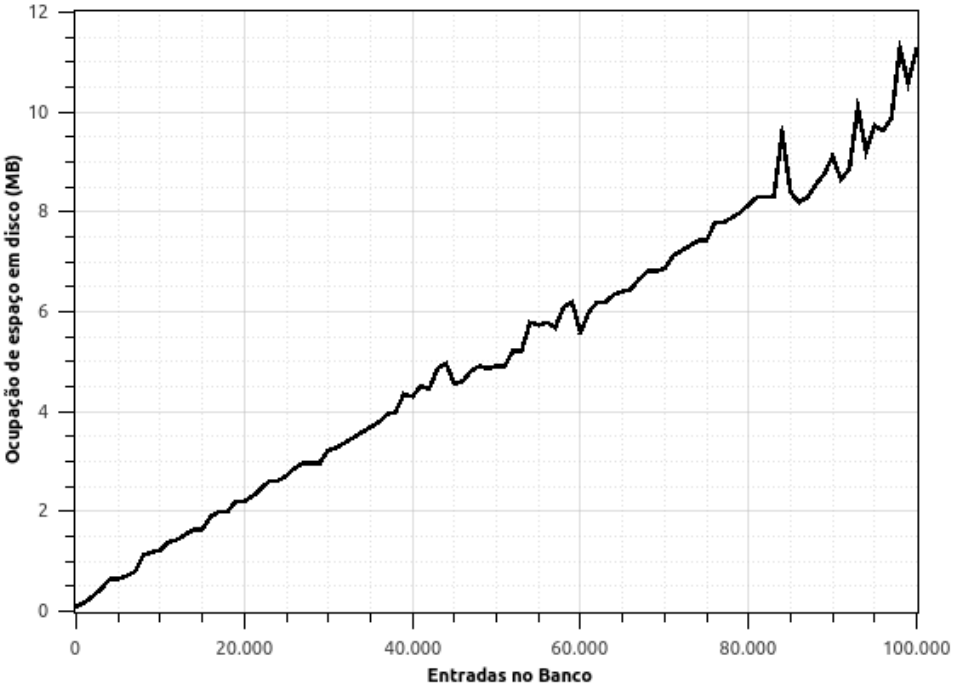
Fonte: Produção do próprio autor.

O uso de disco, apresentado pela Figura 16, também apresenta o comportamento desejado com um crescimento linear do espaço em disco ocupado. Pode-se observar alguns picos no gráfico apresentado, porém é um comportamento normal do banco de dados, onde após certo número de dados inseridos os dados são comprimidos para uma menor ocupação de disco.

Ao analisar o uso de memória, apresentado pela Figura 17, é possível notar um um aumento com relação ao número de entradas no banco. Uma possibilidade do alto uso de memória por parte do banco de dados é a utilização de memória RAM como *cache*, aumentando a velocidade de acesso aos dados. Porém tal comportamento não é esperado para uma aplicação deste tipo, o número de entradas armazenadas no banco não deve interferir no uso de memória da aplicação. É necessário então, para cumprir o objetivo de uma infraestrutura minimalista, estudar maneira de limitar o uso de memória pelo banco de dados, ou até uma outra alternativa para armazenamento dos dados.

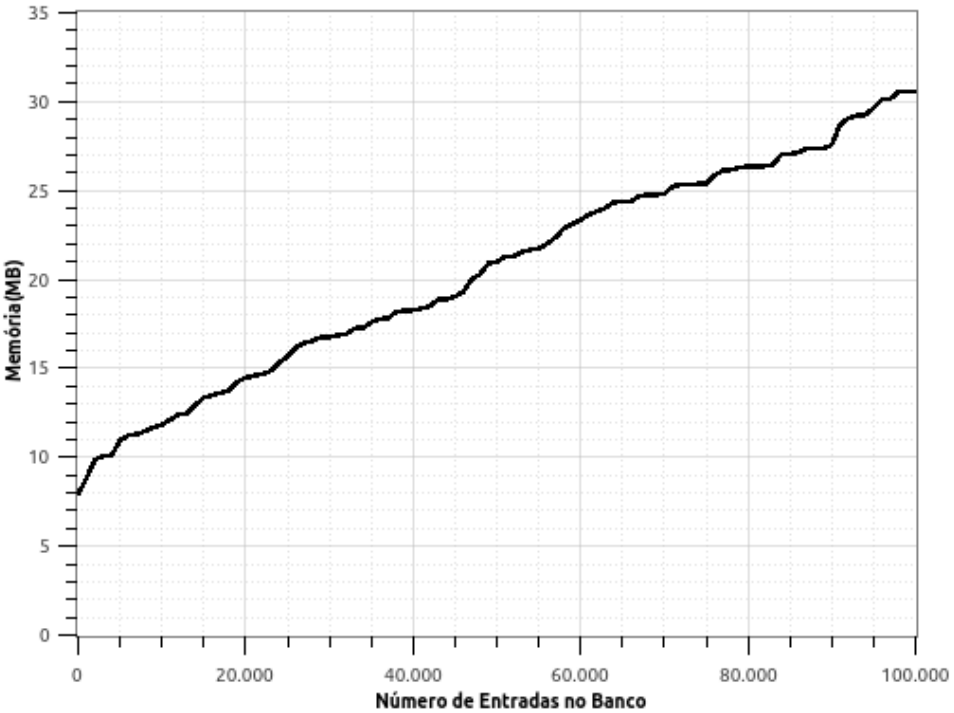
Desta forma, o InfluxDB não preenche os requisitos para um sistema de armazenamento ideal em uma arquitetura minimalista. Sendo necessário o estudo de uma solução que apresente uso de recursos de CPU e espaço em disco similares ao InfluxDB e um uso

Figura 16 – Uso de espaço em disco do InfluxDB em relação ao número de entradas no banco



Fonte: Produção do próprio autor.

Figura 17 – Uso de memória do InfluxDB em relação ao número de entradas no banco



Fonte: Produção do próprio autor.

de memória limitada, diferente do apresentado nesta solução.

4.2 Servidor API

A API foi avaliada quanto ao tempo de resposta da mesma, ou seja, a percepção de tempo que o usuário tem ao realizar uma consulta a um determinado sensor ou conjunto de sensores. O tempo de resposta, como observado na Tabela 2 foi de 12ms com um desvio padrão de 4 ms. Considerando uma requisição HTTP pode-se considerar tal tempo de resposta aceitável visto que está em mesma escala quando comparado com a latência adicionada pela rede IP.

Tabela 2 – Tempo de resposta da API

Tempo de resposta (ms)	Desvio padrão (ms)
12	4

Fonte: Produção do próprio autor.

4.3 Prova de conceito

A partir da prova de conceito citada no capítulo anterior foi possível monitorar a corrente de um dos circuitos do QDC onde ficam ligados os aparelhos de refrigeração do *data center* do PoP-ES/RNP. Utilizou-se da ferramenta Grafana[1] para visualização dos dados em forma gráfica. A interface do Grafana pode ser vista na Figura 18.

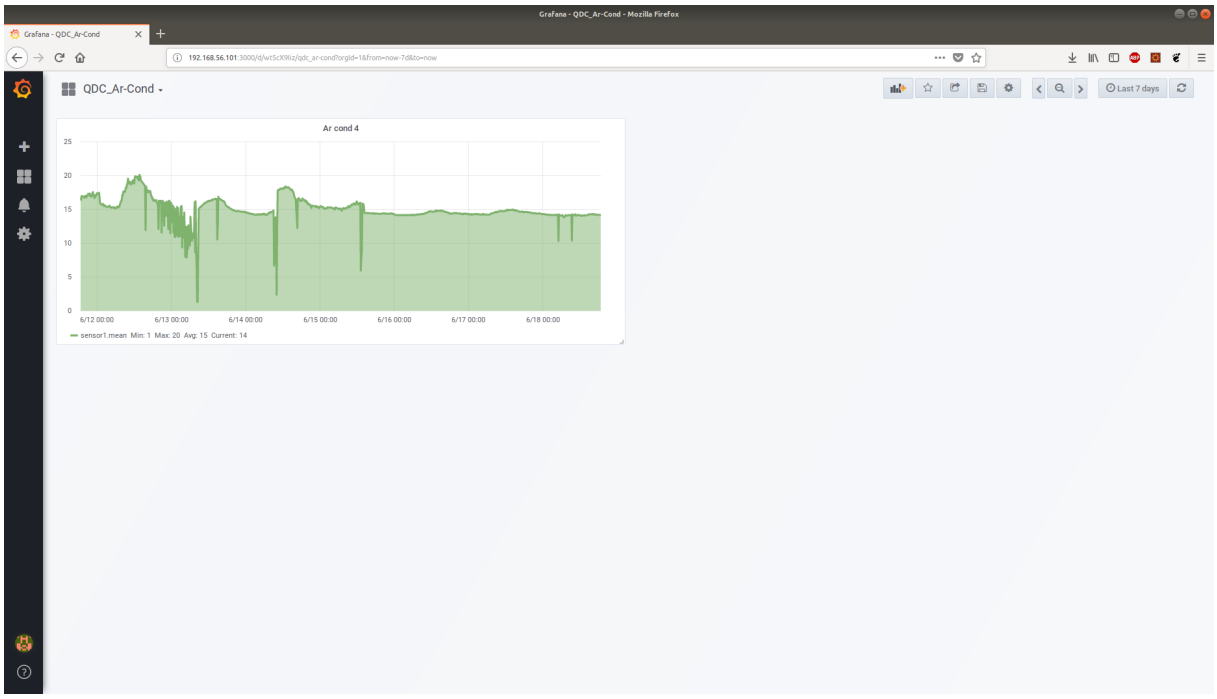
É importante salientar que a integração desta ferramenta de visualização não é escopo deste trabalho, porém quando realizada pelo PoP-ES/RNP demonstra a adaptação dos microsserviços com outras ferramentas e permite que humanos tenha uma melhor interpretação dos dados coletados.

Um exemplo de gráfico gerado a partir dos dados coletados pelo sistema pode ser visto na Figura 19, referente ao circuito do aparelho de ar condicionado 04 no *data center* do PoP-ES/RNP.

Outro caso interessante de uso aconteceu na semana de implantação do protótipo, quando os aparelhos de refrigeração do *data center* estavam apresentando defeitos e

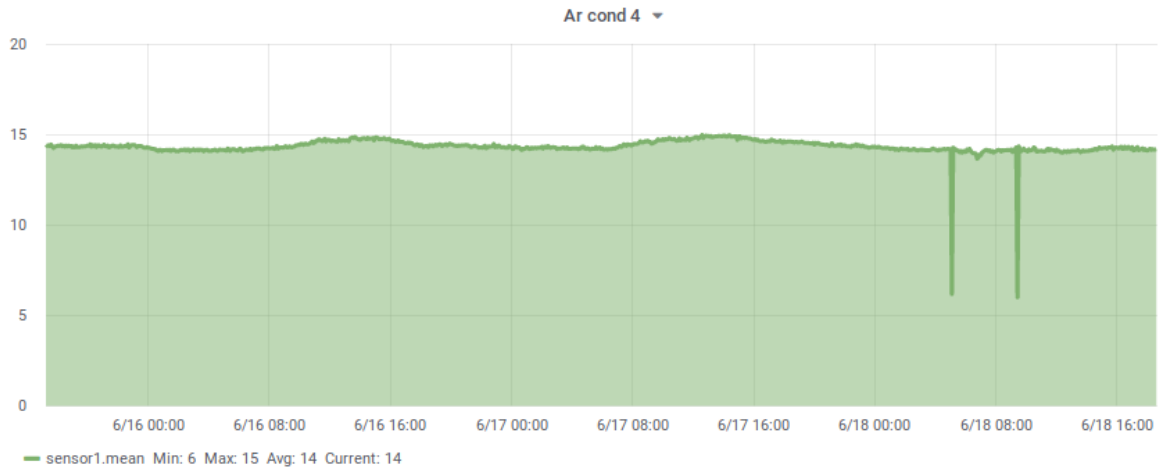
[1] Grafana é uma ferramenta de visualização de dados integrável com diversos serviços de armazenamento de dados, dentre eles o InfluxDB.

Figura 18 – *Dashboard* do Grafana.



Fonte: Produção do próprio autor.

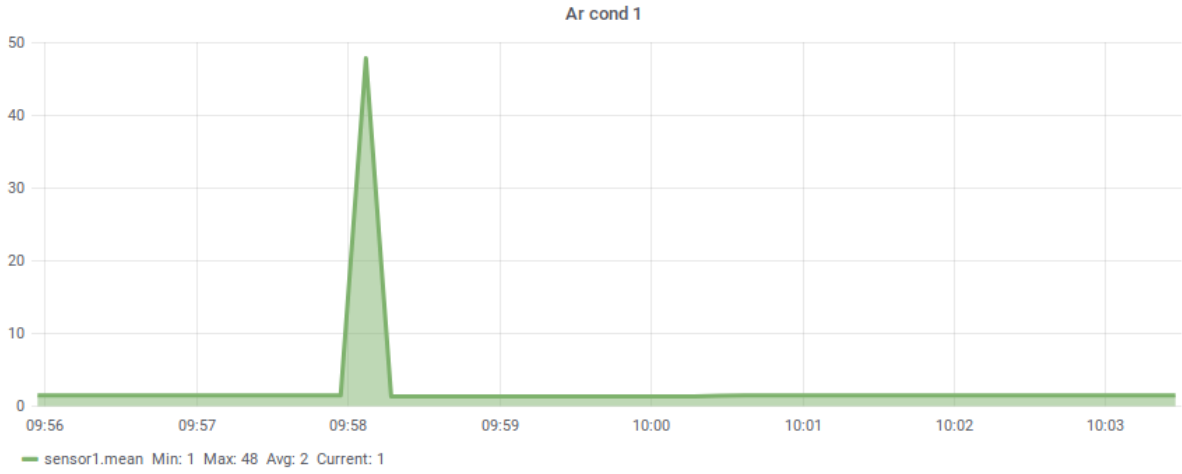
Figura 19 – Gráfico de de corrente no circuito do Ar Condicionado 4 do PoP-ES/RNP.



Fonte: Produção do próprio autor.

passaram por processo de manutenção. Nesta semana então, o sensor foi mudado entre os circuitos e o intervalo de coleta foi configurado para 20s, a fim de detectar alguma anomalia com relação à corrente de alimentação das máquinas. A Figura 20 mostra um pico de corrente no circuito que alimenta uma máquina, esta desarmava seu disjuntor sempre que seu compressor era acionado. Ao fim deste pico observado no gráfico o disjuntor do circuito desarmou. O procedimento de manutenção adotado foi a troca do compressor.

Figura 20 – Gráfico de de corrente no circuito do Ar Condicionado 1 do PoP-ES/RNP, o qual apresentava defeito.



Fonte: Produção do próprio autor.

Portanto é possível dizer que além de permitir uma medição de corrente contínua do circuito, foi possível a identificação de comportamento estranho das máquinas ligadas ao mesmo.

5 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho implementou-se um protótipo de arquitetura para monitoramento de sensores em *data centers* com base em IoT. Foram testados também, aspectos de desempenho de componentes da solução implementada.

Os resultados obtidos demonstram o bom funcionamento da arquitetura. Com o protótipo foi possível capturar dados de sensores, armazená-los e disponibilizá-los de forma estruturada permitindo a integração com sistemas de monitoramento e supervisão.

O modo de implementação utilizando *containers* Docker mostrou-se de grande auxílio, dando ao sistema um escalonamento horizontal de maneira rápida e fácil. O provisionamento de novas instâncias dos serviços pode ser realizado executando-se um simples comando através de imagens Docker padronizadas.

O software Mosquitto, utilizado como *broker* MQTT, apesar de possuir uma não linearidade de uso de recursos quanto ao número de *subscribers* este consegue atender em uma instância até dois mil sensores. Caso o número de sensores seja maior que dois mil, pode-se utilizar de várias instâncias tendo sempre uma capacidade de sensores múltipla de 2 mil.

O InfluxDB mostrou um comportamento não esperado quanto ao uso de memória com relação ao crescimento do número de dados armazenados. Assim, para a solução minimalista desejada, deve-se estudar uma maneira de limitar o uso de memória do serviço, ou encontrar uma alternativa que tenha uma demanda de recursos menor comparada ao InfluxDB.

A utilização de um tópico de controle para comunicação com todos os microcontroladores, apesar de ser um sucesso no caso teste, pode gerar problemas quanto ao uso de processador dos microcontroladores por um *overhead* de mensagens quando a solução é escalada, necessitando-se ainda de maiores estudos quanto a este tipo de comunicação.

Os demais componentes da arquitetura são adequados ao propósito da mesma. O serviço de coleta de dados consegue gerenciar as requisições de leitura e armazenar as respostas dos microcontroladores no banco de dados. A API cumpre seu propósito de entregar dados de maneira estruturada via protocolo HTTP/HTTPS permitindo assim a integração com sistemas de monitoramento que tenham suporte a API REST.

Pode-se considerar então que a arquitetura implantada neste trabalho pode ser empregada no *data center* do PoP-ES/RNP e em outros *data centers* de pequeno/médio porte. Os resultados obtidos comprovam o funcionamento da arquitetura e o benefício

a ser obtido pela implantação do sistema é compensador. As medições são realizadas de maneira contínua, tendo um histórico do ambiente ao longo do tempo auxiliando no diagnóstico tanto da saúde do *data center* quanto de eventos que possam comprometer seu funcionamento.

5.1 Trabalhos Futuros

O presente trabalho realizou a implantação de um protótipo de arquitetura de gerenciamento de sensores e atuadores cujos testes serviram como avaliação para a implantação do sistema. Porém, para a implantação completa do sistema, devem ser investigados e aprimorados os seguintes pontos:

- Criação de uma estrutura de configuração da arquitetura através de arquivo de configuração, incluindo servidor de coleta de dados e API;
- Estudo do uso de CPU dos microcontroladores quando grande número de mensagens é enviado através do tópico de comandos;
- Avaliação do uso de recursos de *hardware* do serviço de coleta de dados;
- Implantação e gerenciamento de atuadores com estudo do impacto no ambiente;
- Implementação de camada de segurança na comunicação via protocolo MQTT;
- Avaliação da implementação de uma camada de comunicação diferente do protocolo WiFi evitando o uso de frequências com alto índice de interferência e ampla utilização;
- Avaliação de alternativas para o uso de memória do banco de dados.

REFERÊNCIAS BIBLIOGRÁFICAS

ALSHUQAYRAN, N.; ALI, N.; EVANS, R. A systematic mapping study in microservice architecture. *Proceedings - 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications, SOCA 2016*, p. 44–51, 2016.

Arduino. *Arduino Software (IDE)*. 2018. Disponível em: <https://www.arduino.cc/en/Guide/Environment>. Acesso em: 2018-07-03.

AXELSON, J. The microcontroller idea book. *USA: Lakeview Research*, 1994.

BONCEA, R.; BACIVAROV, I. A System Architecture for Monitoring the Reliability of IoT. *Proceedings of the 15th International Conference on Quality and Dependability*, n. October, p. 143–149, 2016. Disponível em: <http://ccf-quality-conference.com/wp-content/uploads/ccf2016-143-149.pdf>.

CELESTI, A. et al. Exploring Container Virtualization in IoT Clouds. *2016 IEEE International Conference on Smart Computing, SMARTCOMP 2016*, n. Lxc, 2016.

CHEN, M. et al. A survey of recent developments in home M2M networks. *IEEE Communications Surveys and Tutorials*, v. 16, n. 1, p. 98–114, 2014. ISSN 1553877X.

CISNEIROS, V. C. d. A.; RAMOS, H. Comparação Entre Servidores Web Apache Http Server E Nginx. 2015. Disponível em: <https://victorcisneiros.com/ArtigoFinal.pdf>.

DALZIEL, H. *How to attack and defend your website*. [S.l.]: Syngress, 2014.

DIX, P. Why Time Series Metters for Metrics Real Time and Sensor Data. *InfluxData Whitepaper*, 2016.

Docker Inc. *WHAT IS A CONTAINER*. 2018. Disponível em: <https://www.docker.com/what-container>. Acesso em: 2018-06-14.

DUNNING, T.; FRIEDMAN, E. *Time Series Databases New Ways to Store and Access Data*. [s.n.], 2014. 71 p. ISSN 9781491917022. ISBN 9781491917022. Disponível em: <http://info.mapr.com/rs/mapr/images/Time{_}Series{_}Databases.pdf{\%}0Ahttp://oreilly.com/catalog/e>.

EGEVANG, K.; FRANCIS, P. *The IP network address translator (NAT)*. [S.l.], 1994.

ENGELS, D. W. et al. The networked physical world: An automated identification architecture. In: *In Proceedings of of the Second IEEE Workshop on Internet Applications*. [S.l.: s.n.], 2001.

Espressif Systems. ESP32 Datasheet. p. 58, 2018. Disponível em: <http://espressif.com/sites/default/files/documentation/esp32{_}datasheet{_}>.

FAROOQ, M. U. et al. A critical analysis on the security concerns of internet of things (iot). *International Journal of Computer Applications*, Foundation of Computer Science, v. 111, n. 7, 2015.

FENG, X.; SHEN, J.; FAN, Y. Rest an alternative to rpc for web services architecture. *First International Conference on Future Information Networks*, 2009.

FIELDING, R. T.; TAYLOR, R. N. *Architectural styles and the design of network-based software architectures*. [S.l.]: University of California, Irvine Doctoral dissertation, 2000. v. 7.

IETF. *The Constrained Application Protocol (CoAP)*. [S.l.], 2014.

MUMBAIKAR, S.; PADIYA, P. Web Services Based On SOAP and REST Principles. *International Journal of Scientific and Research Publications*, v. 3, n. 1, p. 2250–3153, 2013. Disponível em: <www.ijsrp.org>.

Node.js Foundation. *About Node.js*. 2018. Disponível em: <https://nodejs.org/en/about/>. Acesso em: 2018-06-18.

NURSEITOV, N. et al. Comparison of JSON and XML Data Interchange Formats: A Case Study. *Scenario*, v. 59715, p. 1–3, 2009. ISSN 09505849. Disponível em: <http://www.cs.montana.edu/izurieta/pubs/caine2009.pdf>.

OASIS. *MQTT Version 3.1.1 Plus Errata 01*. [S.l.], 2015.

Philipp Hauer. *Microservices in a Nutshell. Pros and Cons*. 2015. Disponível em: <https://blog.philipphauer.de/microservices-nutshell-pros-cons/>. Acesso em: 2018-06-17.

RAO, J.; SU, X. A survey of automated web service composition methods. In: SPRINGER. *International Workshop on Semantic Web Services and Web Process Composition*. [S.l.], 2004. p. 43–54.

SCHOFFSTALL, J. C. M. F. M.; DAVIN, C. Rfc 1157: Simple network management protocol (snmp). *IETF, April*, 1990.

SETHI, P.; SARANGI, S. R. Internet of things: architectures, protocols, and applications. *Journal of Electrical and Computer Engineering*, Hindawi, v. 2017, 2017.

TEIXEIRA, M. D. V. T. *Uma proposta de validação ágil de circuitos virtuais baseada em microsserviços*. 69 p. Monografia (graduação) — Universidade Federal do Espírito Santo, Vitoria, 2016.

TEXAS INSTRUMENTS. *Ultra-Small, Low-Power, 16-Bit Analog-to-Digital Converter with Internal Reference*. [S.l.], 2009.

TORRES, A. B.; ROCHA, A. R.; SOUZA, J. N. de. Análise de desempenho de brokers mqtt em sistema de baixo custo. In: *Anais do XXXVI congresso da sociedade brasileira de computaç ao. Sociedade Brasileira de Computaçao*. [S.l.: s.n.], 2016.

VILLAMIZAR, M. et al. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: IEEE. *Computing Colombian Conference (10CCC), 2015 10th*. [S.l.], 2015. p. 583–590.

W3techs. *Usage of web servers broken down by ranking*. 2018. Disponível em: <https://w3techs.com/technologies/cross/web_server/ranking>. Acesso em: 2018-06-18.

YOKOTANI, T.; SASAKI, Y. Comparison with http and mqtt on required network resources for iot. *ICCEREC 2016 - International Conference on Control, Electronics, Renewable Energy, and Communications 2016, Conference Proceedings*, p. 1–6, 2017.