UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

EMERSON YAMAMOTO DOS SANTOS
GABRIEL DE SOUZA BUSTAMANTE

SENSE**IN**: UMA PLATAFORMA PARA COLETA E PROCESSAMENTO DE DADOS DE SENSORES

RIO DE JANEIRO
2022

EMERSON YAMAMOTO DOS SANTOS
GABRIEL DE SOUZA BUSTAMANTE

SENSE**IN**: UMA PLATAFORMA PARA COLETA E PROCESSAMENTO DE DADOS DE SENSORES

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Eduardo Freitas Mangeli de Brito, M.Sc.

RIO DE JANEIRO
2022

Santos, Emerson Yamamoto dos
S237s SENSEIN: uma plataforma para coleta e processamento de dados de sensores / Emerson Yamamoto dos Santos e Gabriel de Souza Bustamante. – 2022.

64 f.

Orientador: Eduardo Freitas Mangeli de Brito.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2022.

1. IoT. 2. Sensor. 3. Desenvolvimento de software. I. Bustamante, Gabriel de Souza. II. Brito, Eduardo Freitas Mangeli de (Orient.). III. Universidade Federal do Rio de Janeiro, Instituto de Computação. IV. Título.

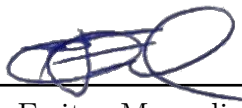
EMERSON YAMAMOTO DOS SANTOS
GABRIEL DE SOUZA BUSTAMANTE

SENSE**IN**: UMA PLATAFORMA PARA COLETA E PROCESSAMENTO DE DADOS DE SENSORES

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 28 de Novembro de 2022

BANCA EXAMINADORA:



Eduardo Freitas Mangeli de Brito
M.Sc. (PESC/COPPE/UFRJ)

Geraldo Bonorio Xexéo
D.Sc. (PESC/COPPE/UFRJ)

Rebeca Campos Motta
D.Sc. (PESC/COPPE/UFRJ)

AGRADECIMENTOS - GABRIEL

Aos meus pais Jina e Ivan, por todo esforço, dedicação, e ensinamentos que ajudaram a moldar a pessoa que sou hoje. Aos meus irmãos Pedro, Joana e Lucas, meus primos e tios – pilares familiar – por todo apoio e influência que eles têm em minha vida.

Ao meu colega de curso, amigo e parceiro nesse trabalho Emerson, que dividiu comigo nesses últimos anos o árduo caminho que nos levou até aqui, entre noites em claro de estudo e outras em claro de diversão.

À minha ex-esposa Adriana que me acompanhou desde o início dessa jornada, sempre mostrando seu apoio, amor e compreensão. O nosso estado civil mudou, mas o carinho e o apoio continuam.

Agradeço também ao meu orientador Eduardo Mangeli, pelos ensinamentos, revisões, orientações e paciência nessa jornada que se mostrou mais longa do que poderia ter sido, e ao professor Geraldo Xexéo por todo o auxílio na elaboração e apresentação desse projeto.

Por fim agradeço a UFRJ e ao corpo docente e discente do DCC. Essa instituição não apenas me deu o conhecimento que, literalmente, me abriu portas para o mundo, como também me permitiu conhecer pessoas e histórias maravilhosas. Em especial aos amigos que levo comigo até hoje.

AGRADECIMENTOS - EMERSON

Agradeço, em primeiro lugar, a Deus, por me capacitar a estar onde muitos não acreditaram que eu pudesse chegar; toda honra a Ti, Senhor.

À minha mãe, Rosemari, por todos os sacrifícios que fez por mim. Por cada gota de suor, cada lágrima derramada, para que eu pudesse vencer. Você é meu tudo. Ao meu pai, Edinaldo (in memorian). Eu consegui, pai. Me perdoa por não ter conseguido te dar essa alegria em vida. Sinto sua falta todos os dias. À minha irmã Michelle e meu cunhado Ismael, por me apoiarem sempre. Vocês estão sempre me ensinando mais sobre o que é fazer parte de uma família.

À minha futura esposa, Bianca, por estar ao meu lado em todos os momentos. Por me oferecer o amor em sua forma mais pura. Por suportar as noites mais difíceis de ansiedade e tristeza, com todo carinho e compreensão. Por me incentivar e acreditar em mim. Eu te amo!

Ao meu grande amigo, Gabriel, por aceitar esse desafio comigo. Pela caminhada que compartilhamos durante todo o curso. Pelos litros de refrigerante e barras *Cookies 'n Cream*, que nos garantiram diversas aprovações. Por não me deixar desistir nos momentos mais difíceis. Nós conseguimos!

Ao senhor Eduardo Mangeli, meu orientador, por ter acreditado nesse trabalho. Sem você nada disso seria possível. Ao professor Luiz Oliveira, por ter compartilhado conosco suas ideias e ter construído bases sólidas para que esse trabalho pudesse acontecer. Ao professor Geraldo Xexéo e à minha amiga Rebeca Motta, por aceitarem participar desse momento único, com toda atenção e carinho. Vocês são referências para mim.

Por fim, agradeço aos meus colegas de trabalho, Gustavo e Elaine, por toda compreensão e incentivo a encerrar esse ciclo.

E a todos os outros amigos que participaram dessa saga, mesmo que indiretamente, o meu muito obrigado; essa vitória também é de vocês.

RESUMO

Com a expansão da Internet das Coisas, o número de dispositivos inteligentes conectados apresenta franco crescimento, criando uma série de desafios referentes aos dados gerados pelos mesmos. Um desses desafios é a produção de aplicações que possam ingerir, armazenar e apresentar informações de forma dinâmica e eficiente. Esse trabalho apresenta uma arquitetura de software, baseada em microsserviços, que atua nessa problemática. Baseado nessa arquitetura, esse trabalho mostra o desenvolvimento de uma plataforma capaz de ingerir dados de telemetria – temperatura, umidade relativa, concentração de CO_2 – e *probe requests*. Para avaliação da plataforma implementada, um estudo de simulação foi realizado.

Palavras-chave: IoT; sensor; desenvolvimento de software; MQTT; Cloud; Banco de dados de séries temporais; microserviços; probe request.

ABSTRACT

With the IoT's industry burst, the number of connected smart devices is growing massively, producing a number of challenges regarding the data generated by them. One of these, is the need for applications that are able to ingest, store and present information in a dynamic and efficient manner. This work presents a microservices-based software architeture. Build upon this architecture, this work implements a platform capable of ingesting telemetry data – temperature, relative humidity, CO_2 concentration, and WiFi probe requests. To evaluate the platform, a simulation study was conducted.

Keywords: IoT; sensor; software development; MQTT; Cloud; Time series database; microservices; probe request.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

AP	Access Point
API	Application Programming Interface
CERP IoT	Cluster of European Research Projects on the Internet of Things
CO2	Dióxido de Carbono
CPU	Central Processing Unit
ETL	Extract, transform, load
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
LSM Tree	Log Structured Merge Tree
M2M	Machine to Machine
MAC	Media Access Control
MQTT	Message Queue Telemetry Transport
MVP	Minimum Viable Product
PC	Personal Computer
ppm	Partes por milhão
QoS	Quality of Service
REST	Representational State Transfer
SDC	StreamSets Data Collector
SGBD	Sistema de Gerenciamento de Banco de Dados
TSDB	Time Series Database
TSM Tree	Time Structured Merge Tree
WSN	Wireless Sensor Network
WS	WebSocket

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Uma das tendências na era atual da computação, em especial no segmento da Internet, reside fora do reino tradicional dos desktops e *notebooks*. No paradigma da Internet das Coisas (do inglês *Internet of Things* - IoT), muitos dos objetos que nos cercam já estão conectados à rede, direta ou indiretamente.

Em 1991, o cientista Mark Weiser usou, em seu artigo *The Computer for the 21st Century*, o termo Computação Ubíqua, referindo-se à computação de forma onipresente e invisível no cotidiano das pessoas. Atualmente, é cada vez mais comum o uso de computação e tecnologia de forma ubíqua em nosso dia a dia, desde relógios, balanças, tênis, lâmpadas, até sensores que, não só captam dados do ambiente, mas também são capazes de interagir com o meio físico. O conceito cunhado por Mark Weiser, trinta anos atrás, hoje é apoiado pelo grande crescimento da Internet das Coisas.

A provedora de dados de mercado *Statista* prevê que teremos 30.9 bilhões de dispositivos IoT conectados no ano de 2025 (STATISTA, 2021). Com o crescimento avassalador do número de dispositivos, surgem problemas relacionados principalmente à escala, nunca antes vista, de aparelhos com acesso direto à Internet. Lidar com a enorme quantidade de dados gerados por esses sensores torna-se então um grande desafio, que nos serviu de motivação para realizar esse trabalho.

O trabalho desenvolvido por (OLIVEIRA et al., 2018) introduziu um sensor, apresentado na Seção 4.1, cujo objetivo é a coleta de dados de telemetria - temperatura, umidade relativa e concentração de dióxido de carbono - e dados de *probe requests*, também serviu de motivação para a criação de uma plataforma capaz de interagir com tais dados.

1.2 OBJETIVO

O objetivo deste trabalho é apresentar uma arquitetura de software, baseada em microsserviços, capaz de processar o fluxo de informações geradas por sensores. Além disso, objetiva na implementação de um produto minimamente viável (MVP) de uma plataforma baseada nessa arquitetura, e a realização de estudos de simulação.

O MVP foca nas etapas de ingestão, armazenamento, processamento e apresentação de informações. Nos capítulos subsequentes, serão apresentados a arquitetura proposta e o conjunto de artefatos desenvolvidos, cuja finalidade é receber dados de sensores, processar esse fluxo de dados e apresentar as informações em uma interface fácil e intuitiva. A fim de avaliar a solução proposta, um estudo de caso evolvendo os processos de coleta e apresentação em tempo real de dados sensoreados por dispositivos fixos foi idealizado.

No entanto, por conta da pandemia do COVID-19 causada pelo coronavírus SARS-CoV-2, não foi possível coletar um conjunto de dados relevante para nosso estudo, devido às políticas de isolamento e a suspensão das atividades presenciais da Universidade. A fim de contornar esse obstáculo, optamos então pela utilização de dados sintéticos, no intuito de simular o processo de coleta de dados de um sensor, através de um estudo de simulação.

1.3 ORGANIZAÇÃO

Este trabalho encontra-se dividido em cinco capítulos. O Capítulo 2 aborda brevemente conceitos que serviram de base para o desenvolvimento deste trabalho. Mais adiante, no Capítulo 3, apresentamos a solução arquitetural para o problema introduzido no capítulo inicial. Em seguida, no Capítulo 4, descrevemos com detalhes a implementação dessa arquitetura, e o processo de desenvolvimento da aplicação MVP *SenseIn*. Por fim, o Capítulo 5 apresenta o estudo de simulação que realizamos utilizando o sistema e os resultados obtidos.

2 FUNDAMENTAÇÃO TEÓRICA

Esse capítulo introduz breves definições para conceitos que serão utilizados no desenvolvimento dos próximos capítulos.

2.1 INTERNET DAS COISAS

Em uma abordagem sem restrições em nível de protocolos de comunicação e centrada no usuário, (GUBBI et al., 2013) define IoT como: dispositivos sencientes e atuantes interconectados, que provêm a habilidade de compartilhar informações entre plataformas através de um *framework* unificado, desenhando um quadro operacional comum que permite o desenvolvimento de aplicações inovadoras. Isso é alcançado através de sensoriamento ubíquo, análise de dados e representação da informação tendo a computação em nuvem como o *framework* unificado. Os autores dividem em três os componentes necessários para alcançar o sensoriamento ubíquo: (a) Hardware - formado por sensores, atuadores e dispositivos embarcados de comunicação, (b) Middleware - ferramentas de armazenamento e computação para análise de dados e (c) Apresentação - ferramentas de visualização e interpretação accessíveis através de diferentes plataformas.

Figura 1 – Diagrama IoT com foco em computação em nuvem

Fonte: Sundmaeker et al. (2010)

O Cluster of European Research Projects on the Internet of Things (CERP IoT) define "coisa" como:

> Uma entidade real/física ou digital/virtual que existe e se move no espaço e no tempo e é capaz de ser identificada. "Coisas" participam ativamente em negócios e processos sociais e de informação, onde são capazes de interagir e se comunicar entre si e com o ambiente, através da troca de dados e informações coletadas sobre o meio, enquanto reagem de forma autônoma aos eventos do mundo real/físico, disparando ações e criando serviços com ou sem direta intervenção humana (SUNDMAEKER et al., 2010, tradução nossa).

2.2 ARQUITETURA DE MICROSSERVIÇOS

A Arquitetura de microsserviços é um padrão arquitetural no qual uma única aplicação é formada por um conjunto de pequenos serviços (microsserviços), cada um sendo executado de forma independente e se comunicando através de mecanismos de troca de mensagem leves, geralmente APIs HTTP. Esses serviços são definidos em torno de regras de negócio concisas, e implantáveis de forma independente. O gerenciamento desses serviços deve ser o mais descentralizado possível, e eles podem ser escritos em linguagem de programação diferentes e se utilizar de diferentes tecnologias para armazenamento de dados (FOWLER; LEWIS, 2014).

Em resumo, DRAGONI et al. define um microsserviço como "um processo independente e coeso, que interage através de troca de mensagens", sendo coeso a palavra empregada para definir um serviço que apenas implementa funcionalidades fortemente ligadas ao domínio e o conjunto de regras de negócio por ele empregado.

Alguns dos principais benefícios da adoção de uma arquitetura baseada em microsserviços são:

Heterogeneidade tecnológica Dado um sistema composto por múltiplos serviços independentes e interoperáveis, cada um deles pode adotar tecnologias diferentes. Isso permite escolher a ferramenta mais adequada para cada problema, em oposição a seleção de uma ferramenta única para toda a aplicação. Dessa forma, se uma parte da aplicação precisa de um aumento de performance, um novo conjunto de ferramentas - por exemplo uma linguagem de programação diferente - pode ser utilizado, sem necessidade de uma transição tecnológica por todas as partes do sistema (NEWMAN, 2015).

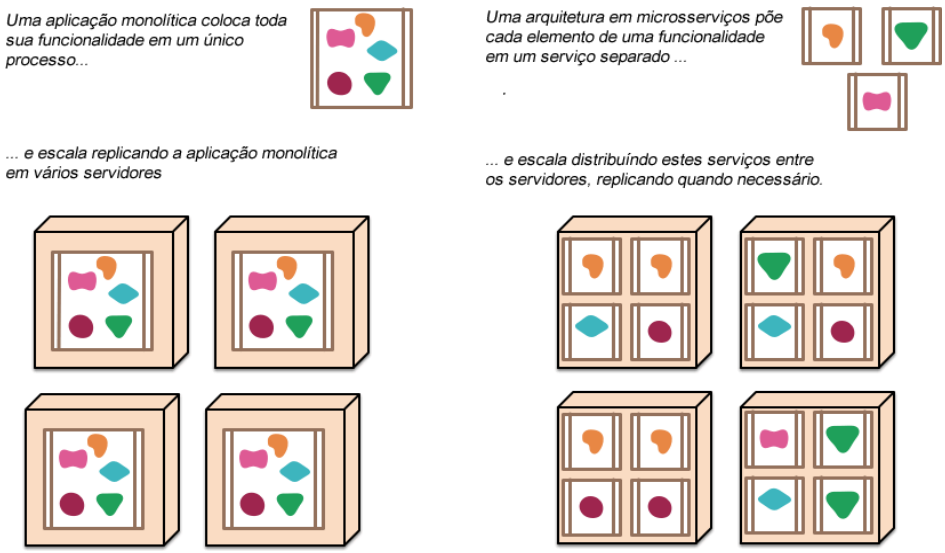
Resiliência Na arquitetura de microsserviços existe o isolamento natural entre cada um dos componentes, diferente da aplicação monolítica onde o sistema é executado como um único processo (NEWMAN, 2015). Por conta desse isolamento, se um componente do sistema falha, essa falha não é propagada em um efeito cascata. Dessa forma é possível isolar o problema permitindo que o resto do sistema continue sendo executado.

Escalabilidade Para serviços monolíticos[1], tudo tem que ser escalado em conjunto. Isso significa que se uma parte da aplicação sofre de problemas de performance, toda a solução precisa ser redimensionada, geralmente escalando de forma horizontal ao adicionar ou remover nós a um sistema distribuído, ou vertical, adicionando ou removendo recursos como CPU e memória a um único nó. Em contraste, numa arquitetura de microsserviços, cada serviço ou componente pode ser escalado de forma independente, permitindo uma gerência mais efetiva dos recursos computacionais (NEWMAN, 2015).

Facilidade de implantação Atualizações no código de uma aplicação monolítica, por menor que sejam, demandam que toda a aplicação seja reimplantada.

Com microsserviços é possível manter um escopo isolado de mudanças, realizando a implantação em um único serviço de forma independente do resto da aplicação. Isso permite maior velocidade na liberação de uma nova funcionalidade em produção, além de trazer mais segurança a implantações em geral, tendo em vista que caso um problema ocorra, ele pode ser isolado e tratado com mais rapidez (NEWMAN, 2015).

Figura 2 – Aplicações monolíticas e microsserviços



Fonte: Fowler e Lewis (2014)

Temos de considerar também algumas desvantagens nesse tipo de arquitetura. Enquanto um serviço isolado tende a ser mais simples que uma aplicação monolítica complexa, um sistema formado por partes autônomas e especializadas forma um todo mais complexo, distribuído.

[1] Um monolito é uma aplicação de software composta por módulos que não podem ser executados de forma independente

Por isso, seguindo Martin Fowler, especialista em arquitetura de software, "não considere migrar para microsserviços a menos que você já tenha um sistema que seja muito complexo para gerenciar como um monolito".

2.3 PROBE REQUEST

Dispositivos móveis (*smartphones* e *notebooks*, por exemplo), quando têm sua interface WiFi ativa, tentam se conectar periodicamente à pontos de acesso sem fio (APs). Essas redes podem ser conhecidas, ou seja, nas quais o dispositivo já tenha se conectado antes, ou novas.

Para descobrir quais APs se encontram disponíveis nas proximidades, os dispositivos enviam *probe requests*. Eles são um tipo específico de frame de gerenciamento do protocolo 802.11 (ANSI/IEEE, 1999), utilizados durante o processo ativo de descoberta de redes sem fio.

Os *probe requests* podem ser do tipo Broadcast, utilizado ativamente na busca de um AP qualquer no seu raio de alcance, ou direcionado - com envio endereçado a um AP específico. O header desse tipo de frame contém as seguintes informações:

- *Frame Control*: indica o subtipo do frame;
- Endereço 1: endereço de destino do frame (destination address ou DA);
- Endereço 2: endereço de origem do frame (source address ou SA);
- Endereço 3: no caso de um request direcionado, o BSSID (MAC address) do AP alvo;
- *Sequence Control*: número sequencial incremental. Utilizado para distinguir retransmissões.

2.4 PROTOCOLOS

2.4.1 MQTT

Message Queue Telemetry Transport (MQTT)[2] é um protocolo assíncrono baseado no modelo de troca de mensagens *publish/subscribe*, operando sobre a stack TCP. Ele foi desenvolvido pela IBM com foco na comunicação leve entre dispositivos (M2M). O padrão assíncrono *publish/subscribe*, em contraste ao *request/response*, atende melhor as necessidades no contexto IoT já que em geral as mensagens são enviadas de forma unidirecional, como por exemplo na captação de dados de sensores, e não precisam de resposta. Isso significa que a banda de rede utilizada é menor, e com menos mensagens processadas, o

[2] MQTT.org, **MQ Telemetry Transport**. Disponível em: https://mqtt.org Acesso em: 15 jun.2019

consumo energético dos dispositivos consequentemente diminui (KARAGIANNIS et al., 2015).

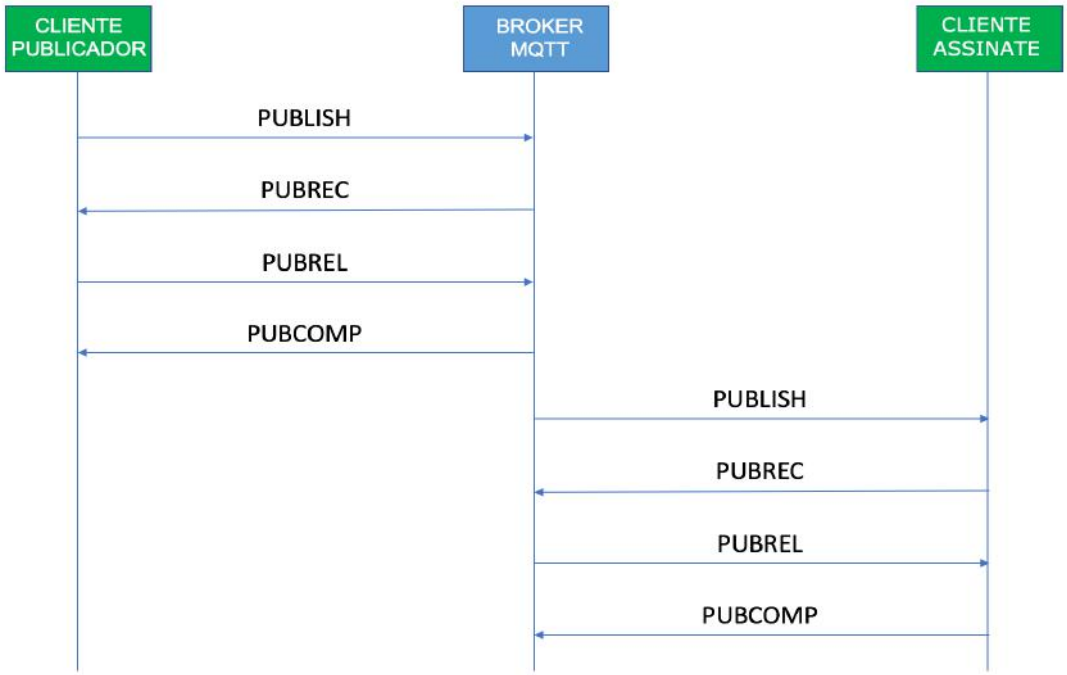
Nesse protocolo, existe o conceito de *broker* (servidor) que contém tópicos. Cada cliente pode publicar mensagens num tópico específico e/ou ser um assinante, recebendo automaticamente mensagens cada vez que determinado tópico receber atualizações. O MQTT foi desenhado para consumir poucos recursos de banda e energia, sendo um ótimo candidato para aplicações IoT. Ainda que operando sobre o protocolo TCP, o MQTT foi desenhado de forma a apresentar um custo operacional baixo se comparado a outros protocolos que também se utilizam do protocolo TCP (THANGAVEL et al., 2014).

Ele conta com mecanismos de confiabilidade na troca de mensagens, implementando três níveis de qualidade de serviço (QoS):

- (0) At most once: a mensagem é enviada sem a necessidade de nenhum reconhecimento de entrega (*ack*);
- (1) At least once: a mensagem é enviada e reconhecida pelo menos uma vez;
- (2) Exactly once: Um mecanismo de *handshake* de quatro vias é utilizado para assegurar que a mensagem será entregue exatamente uma vez.

Quanto maior o nível de QoS, maior será a garantia de entrega, o custo operacional computacional e o atraso entre a produção e a recepção de uma mensagem.

Figura 3 – Troca de mensagem QoS 2 no produtor e consumidor



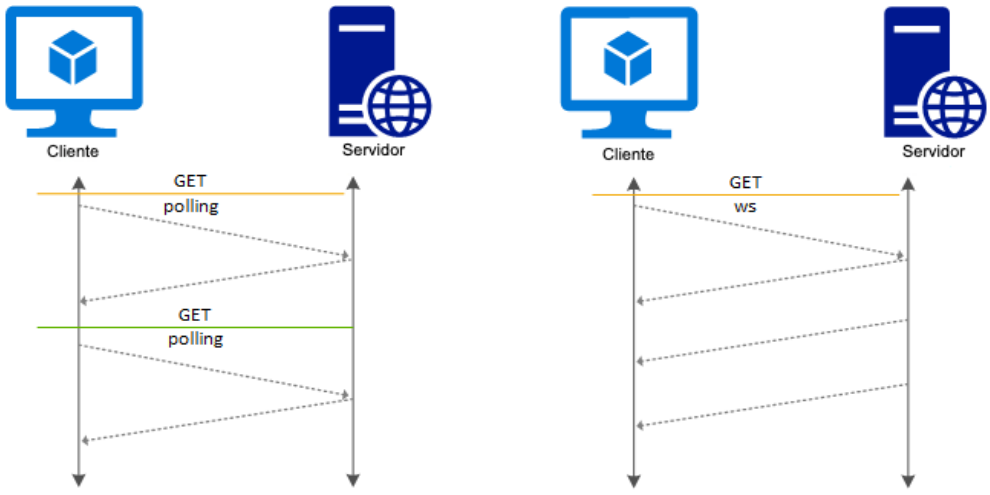
PUBLISH, PUBREC, PUBREL e PUBCOMP são alguns dos possíveis tipos de pacotes do protocolo MQTT

- PUBLISH[3]: Pacote enviado do cliente ao servidor ou do servidor ao cliente para transportar uma mensagem de aplicação. Um dos campos do seu header é o nível de serviço (QoS), que irá determinar o tipo de pacote esperado na resposta.
- PUBREC[4]: Publish received é o pacote de resposta ao pacote PUBLISH no QoS de nível 2.
- PUBREL[5]: Publish released é o pacote de resposta ao pacote PUBREC. É o terceiro pacote na troca de mensagens com QoS 2.
- PUBCOMP[6]: Publish complete é o pacote de resposta ao pacote PUBREL. É o útimo pacote na troca de mensagens com QoS 2.

2.4.2 Websocket

O protocolo Websocket foi desenvolvido como parte de uma iniciativa do HTML 5 para facilitar comunicações em canais TCP (KARAGIANNIS et al., 2015). É um híbrido entre protocolos *request/response* e *publish/subscribe*. Um cliente Websocket inicia uma conexão com um servidor através de um *handshake*, similar ao HTTP, e assim estabelece uma sessão Websocket. Após essa etapa, os headers HTTP são removidos permitindo que durante uma sessão, tanto cliente como servidor possam transmitir mensagens de forma assíncrona numa conexão full-duplex. A sessão pode ser terminada a partir de ambos os lados.

Figura 4 – Comparativo HTTP polling X Websocket



[3] oasis-open.org, **PUBLISH Packet**. Disponível em: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718037 Acesso em: 15 mai.2020

[4] oasis-open.org, **PUBREC Packet**. Disponível em: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718048 Acesso em: 15 mai.2020

[5] oasis-open.org, **PUBREL Packet**. Disponível em: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718053 Acesso em: 15 mai.2020

[6] oasis-open.org, **PUBCOMP Packet**. Disponível em: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718058 Acesso em: 15 mai.2020

Na Figura 4 vemos exemplos de requisições utilizando os modelos HTTP polling e Websocket. Do lado esquerdo vemos duas chamadas ao servidor, incluindo dois *handshakes*. No lado direito da figura, observa-se o modelo Websocket, onde pode-se ver o cliente estabelecendo a conexão apenas uma vez e passando a receber dados do servidor de forma unidirecional.

Esse protocolo foi criado para reduzir o custo operacional na comunicação HTTP padrão e permitir uma comunicação full-duplex em tempo real. Estima-se que a utilização de Websockets represente uma redução de três para um na latência se comparado ao polling HTTP half-duplex, a forma tradicional até então de se obter comunicação "em tempo real" via HTTP. Diferente de protocolos como MQTT, o Websocket não foi desenhado para ser utilizado em dispositivos de recursos limitados, mas sim para permitir comunicação em tempo real com um custo operacional reduzido (KARAGIANNIS et al., 2015).

2.5 INGESTÃO DE DADOS

Ingestão de dados é o processo de transferência de dados de uma ou mais fontes para o sistema de destino. É uma parte do processo ETL (Extract, transform and load).

Ferramentas ETL em geral apresentam as seguintes funcionalidades:

- identificação de informações relevantes nas fontes de dados;
- extração das informações relevantes;
- integração das informações coletadas a partir das múltiplas fontes em um formato único;
- manipulação e limpeza do conjunto de dados resultante com base em regras de negócio;
- propagação do dado para os sistemas de destino (*data warehouses*, *data marts*...)

Tradicionalmente, um ETL é construído como um pipeline de processos em lote, cada um deles recebendo dados de entrada a partir de arquivo e escrevendo em outro arquivo de saída, que pode ser consumido pela próxima etapa do pipeline. Aplicações tradicionais, como *data warehouses*, não são particularmente afetadas pela latência apresentada por esse tipo de processamento, já que elas não precisam necessariamente do dado em seu estado mais recente. Por outro lado, aplicações na arquitetura IoT, utilizadas no suporte a tomada de decisões em tempo real precisam apresentar um modelo mais preciso do mundo real.

Para aplicações modernas e sensíveis a latência na entrada de informações, o processo de ETL deve ser concebido como um problema de *streaming*. Isto é, o dado que chega é entregue para ser processado de forma imediata, passando pelos elementos que preparam

e carregam os dados nos sistemas de gerenciamento de dados de destino (MEEHAN et al., 2017).

2.5.1 Big Data

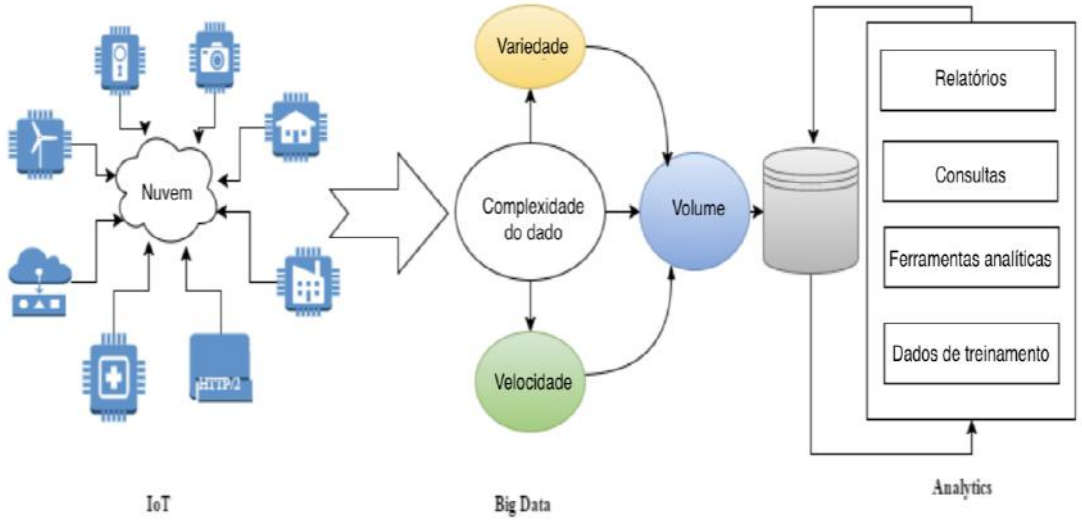
MILOSLAVSKAYA; TOLSTOY define *big data* como conjuntos de dados de tamanho e estrutura que excedem as capacidades de ferramentas de programação tradicionais (bancos de dados, software, etc.) na coleta de dados, armazenamento e processamento em uma janela de tempo razoável, e consequentemente, que estão além da capacidade da percepção humana. Soluções baseadas em TI tradicional são geralmente diferenciadas de soluções baseadas em *big data* pela existência de ao menos um dos seguintes fatores:

- volume - enorme volume de dados;
- velocidade - altíssimas taxas de transferência de dados;
- variedade - dados pouco estruturados, que pode ser entendido como irregularidade na estrutura dos dados e dificuldade na identificação de correlações.

O estudo realizado pelo IDC (2012 apud MARJANI et al., 2017) caracteriza tecnologias *big data* como um conjunto de tecnologias e arquiteturas que objetivam tirar proveito de um volume massivo de dados com formatos heterogêneos, permitindo captura em alta velocidade, descoberta e análise.

2.5.2 Relação entre *big data* e IoT

Figura 5 – Relação entre *big data* e IoT



Fonte: Marjani et al. (2017)

Uma aplicação interessante da Internet das Coisas está na análise de informação acerca das "coisas conectadas". O trabalho realizado por (MARJANI et al., 2017) explica a relação entre *big data* e IoT, exposto na Figura 5. De um lado temos o gerenciamento de fontes de dado IoT, nas quais dispositivos conectados se utilizam de aplicações para interagir entre si. Por exemplo, a iteração de dispositivos como câmeras CCTV, lâmpadas inteligentes e de automatização residencial geram um grande número de fontes de dados de formatos heterogêneos. Informações sobre tais fontes de dado podem ser armazenadas na nuvem em bases de dados tradicionais. Os dados gerado pelos dispositivos são aqui chamados de *big data*, por conta de seu volume, variedade e velocidade. Esse grande volume de dado devem ser armazenado em ferramentas especializadas, com capacidade de armazenamento de grandes arquivos de forma distribuída e tolerante a falhas.

2.6 BANCOS DE DADOS NOSQL

NoSQL é uma abordagem arquitetural para bancos de dados que podem acomodar uma variedade de modelos de dados, como chave-valor, documentos, colunar e grafos. O termo vem do inglês *"not only SQL"* ou *"no-SQL"* como uma alternativa ao tradicional modelo relacional, onde os dados são armazenados em tabelas e possuem esquemas bem definidos. Além de serem não relacionais, algumas das características de um banco de dados NoSQL são:

- Distribuídos: os dados podem ser armazenados e gerenciados em diferentes máquinas.
- Escalabilidade horizontal: o número de nós pode ser alterado a fim de ajustar a performance de forma quase linear.
- Ausência de esquema ou esquema flexível: ausência completa ou quase total do esquema que define a estrutura dos dados modelados. Esta ausência de esquema torna mais fácil a tarefa de armazenar registros com estruturas diferentes e também proporciona ganhos de performance, ao se reduzir o número de junções necessárias para obtenção de registros.
- Consistência eventual: a consistência nem sempre é mantida entre os diversos pontos de distribuição de dados. Esta característica tem como princípio o teorema CAP (Consistency, Availability e Partition tolerence), que diz que, em um dado momento, só é possível garantir duas de três propriedades entre consistência, disponibilidade e tolerância à partição. No contexto de bancos NoSQL geralmente são privilegiadas a disponibilidade e a tolerância à partição. Como consequência, as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) não podem ser obedecidas simultaneamente. (LÓSCIO; OLIVEIRA; PONTES, 2011)

2.6.1 Chave-valor

O modelo chave-valor, como em uma tabela *hash*, é um banco de dados composto por um conjunto de chaves, cada uma delas associadas a um único valor. Dependendo da implementação do banco, os valores podem ser de diferentes tipos, como strings, binários ou estruturas de dados conhecidas, como vetores, listas e conjuntos. Por conta de suas características, apresenta baixa latência em consultas diretas.

2.6.2 Temporal

Séries temporais são conjuntos de observações sequenciais ao longo do tempo. Exemplos de séries temporais são medições de temperatura em uma estação climática coletadas a cada hora, ou então, a cotação de fechamento diária de ações na bolsa de valores. Bancos de dados de séries temporais (do inglês *Time Series Database* - TSDB) são otimizados para trabalhar com dados desse tipo. Um TSDB é construído e desenhado para lidar com métricas, eventos ou medições de registros temporais, permitindo ao seu usuário criar, enumerar, atualizar, destruir e organizar séries temporais de forma eficiente (NAQVI; YFANTIDOU; ZIMÁNYI, 2017). Uma característica fundamental dos dados temporais é que na maior parte das vezes a sua análise é feita em função do tempo, por exemplo: "Qual foi a temperatura máxima nos últimos 7 dias de uma série de observações?".

2.7 SERVIÇOS RESTFUL

REST (Representational State Transfer) é um conjunto de princípios arquiteturais utilizados na criação de serviços WEB que focam em **recursos** de um sistema, como a forma de endereçar e transferir um recurso através do protocolo HTTP. É o padrão mais utilizado na implementação de serviços WEB (RODRIGUEZ, 2008). Um serviço é dito RESTful se ele implementa o conjunto de princípios REST.

Uma implementação de um serviço REST segue os seguintes princípios básicos:

- Utiliza métodos HTTP explicitamente.
- Não armazena estado.
- Expõe URIs estruturadas em diretórios.
- Transfere XML, JSON, ou os dois.

Uma das características de um serviço RESTful é o uso explícito de métodos HTTP seguindo o protocolo como definido no RFC 2616.

De forma resumida, são eles:

- GET: Para recuperar um recurso.

- POST: Para criar um recurso no servidor.
- PUT: Para atualizar ou alterar o estado de um recurso.
- DELETE: Para remover ou excluir um recurso.
- PATCH: Para atualizar ou alterar parcialmente o estado de um recurso.

2.8 APLICATIVO DE PÁGINA ÚNICA

Um aplicativo de página única (*single page application* ou SPA) é uma aplicação web ou site que consiste de uma única página web, onde todos os recursos necessários para navegação são obtidos no primeiro carregamento (FLANAGAN, 2006). Conforme o usuário interage com a página, o conteúdo é carregado dinamicamente. A URL é atualizada conforme a navegação, de maneira a emular a navegação tradicional; ainda assim, uma nova solicitação de página completa nunca é realizada.

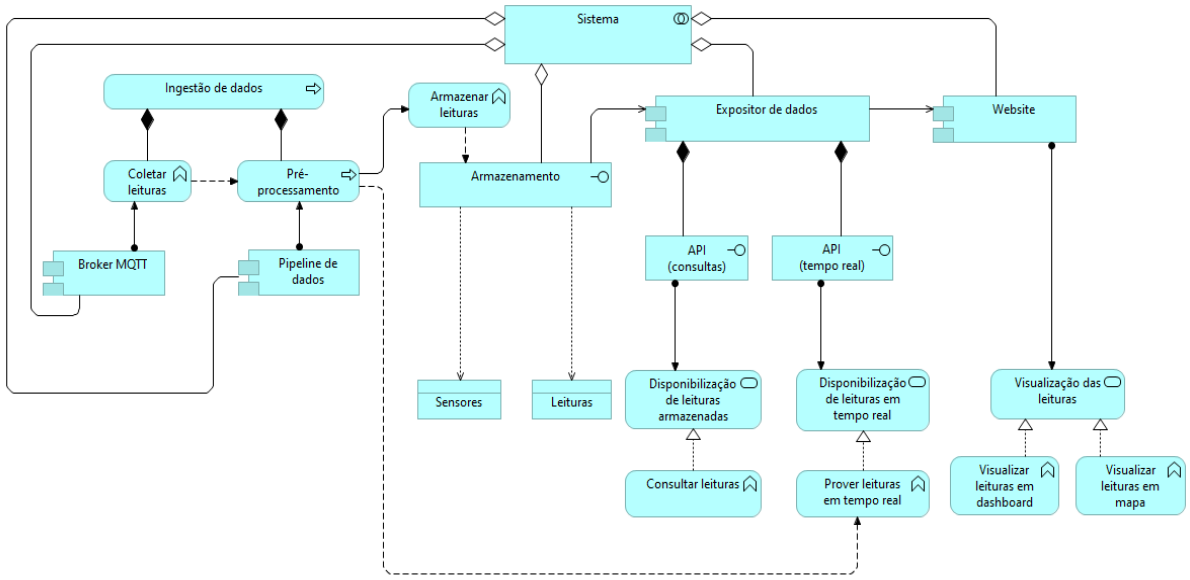
Essa abordagem evita a interrupção da experiência do usuário entre páginas sucessivas, fazendo com que o aplicativo se comporte como um aplicativo *desktop*.

Com os recursos carregados por completo no primeiro carregamento, as requisições ao servidor web ganham em desempenho, pois passam a trazer apenas os dados a serem apresentados; esses dados trafegam, em geral, no formato JSON. Além disso, com a interface desacoplada dos dados, é possível realizar alterações na interface do usuário sem afetar o *back-end*.

3 ARQUITETURA PROPOSTA

Esse capítulo apresenta o modelo da arquitetura proposta. O principal objetivo dessa arquitetura é a ingestão de dados gerados por sensores, problema introduzido no capítulo inicial desse projeto. O modelo foi desenvolvido utilizando a linguagem de modelo ArchiMate opengroup.org (2022), com a ferramenta Archi[1].

Figura 6 – Modelo da arquitetura na linguagem ArchiMate



A Figura 6 apresenta camada de aplicação da solução arquitetural proposta, seus componentes e como eles se comunicam. A arquitetura (estruturada) tem como base quatro componentes: Broker MQTT, Pipeline de dados, Expositor de dados e Website; e três interfaces: Armazenamento, API de consultas e API de tempo real.

Ao broker MQTT está atribuída a função de coletar leituras externas. Essas leituras podem ser providas por qualquer origem; em nosso sistema, elas são obtidas por sensores que leem dados de telemetria e probe request. Após a coleta, o broker disponibiliza as leituras para o processo de pré-processamento, realizado pelo pipeline de dados. Esse processo realiza diversas correções e adaptações necessárias ao dado, que é coletado em estado bruto. Esses passos compõem o processo de ingestão de dados do sistema.

Após ser processada, a leitura percorre dois caminhos: de um lado, o pipeline dispara a função de armazenar leituras, armazenando a leitura processada no banco de dados; do outro lado, essa mesma leitura processada flui para a função que provê leituras em tempo real, que tem como objetivo alimentar o serviço de disponibilização de leituras em tempo

[1] Archi. **Open Source ArchiMate modelling**. Disponível em: https://www.archimatetool.com/. Acesso em: 21 fev.2021

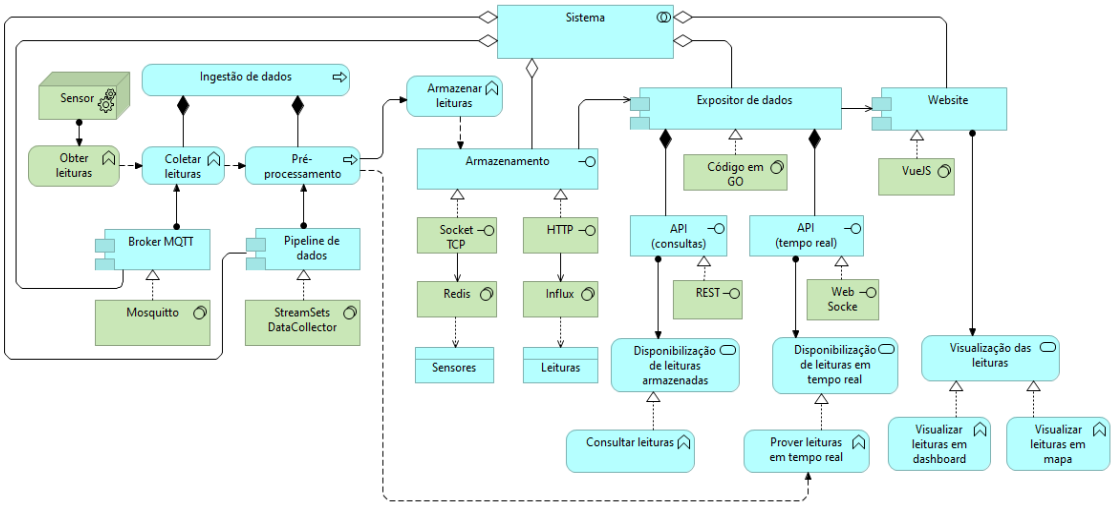
real. Esse serviço está associado diretamente à interface de API de tempo real, que é uma das interfaces que compõem o módulo expositor de dados. A outra interface que compõe o módulo expositor de dados é a API de consultas. Essa interface está associada ao serviço de disponibilização de leituras armazenadas. Através do método de consultar leituras, o serviço consulta o banco de leituras do sistema, obtendo qualquer leitura já processada pelo sistema, em qualquer intervalo de tempo.

O módulo expositor de dados tem, então, duas origens de leituras: em tempo real, que são enviadas diretamente pelo pipeline de dados, após pré-processamento; e armazenadas, que são consultadas sob demanda. Essas leituras são servidas, através do expositor, ao módulo de website, que está associado ao serviço de visualização das leituras. Esse serviço possui duas funções principais: a função de visualizar leituras em dashboard, mais apropriada para uma visualização completa das leituras, com gráficos e tabelas, para qualquer intervalo de tempo (inclusive tempo real); e a função de visualizar leituras em mapa, utilizada para realizar a visualização, em tempo real, dos dados oriundos dos N sensores dispostos geograficamente.

4 IMPLEMENTAÇÃO

Esse capítulo apresenta de forma detalhada o conjunto de artefatos de software desenvolvidos nesse trabalho, que se encarregam de ingerir, processar, armazenar e apresentar dados coletados a partir de sensores. Todo o código-fonte da aplicação encontra-se hospedado na plataforma GitHub[1]. A aplicação desenvolvida pode ser acessada através do endereço https://sensein.tech[2]. Essa aplicação permite que atores possam visualizar a informação processada de forma intuitiva, diminuindo o tempo entre a ocorrência de um evento e uma possível atuação. Por exemplo, coletando continuamente a concentração de CO2 em um ambiente, a aplicação pode disparar um alarme quando esta concentração for superior ao recomendado. Esse é um problema mais comum do que se pensa, pois os aparelhos de ar-condicionado do tipo split não promovem renovação de ar. Cabe salientar que ares-condicionados do tipo split são amplamente utilizados em ambientes residenciais e comerciais, e que a ausência de um mecanismo de renovação de ar pode causar sonolência, perda de produtividade e até mesmo danos à saúde de seus usuários. O sistema proposto neste trabalho, permite que usuário monitorem a concentração de CO2 no ambiente, dentre muitas outras medidas, permitindo que seja acionado de forma manual ou automática algum mecanismo de renovação de ar do ambiente. A escolha dos componentes foi dada seguindo os seguintes critérios: licença de código livre, facilidade de implementação e escalabilidade.

Figura 7 – Modelo da arquitetura implementada, na linguagem ArchiMate



A Figura 7 mostra a camada de tecnologia da arquitetura proposta e seus componentes.

[1] github.com, **github.com: Trabalho de conclusão de curso**. Disponível em: https://github.com/gsouzab/tcc. Acesso em: 12 dez. 2022

[2] sensein.tech, **SenseIN**. Disponível em: https://sensein.tech. Acesso em: 12 dez. 2022

4.1 SENSORES

Os sensores físicos utilizados nesse trabalho foram desenvolvidos pela equipe do projeto Sherlock, associado ao Laboratório do Futuro PESC/COPPE/UFRJ. Os sensores foram desenvolvidos em uma plataforma aberta de prototipação de hardware chamada Raspberry Pi - um mini computador de baixo custo e recursos limitados. Um sensor é identificado pelo seu endereço físico, conhecido como MAC address, que é um endereço único que identifica cada interface de rede sem fio que implementa o protocolo 802.11. Cada sensor coleta continuamente dados de telemetria - temperatura, umidade relativa, concentração de CO_2 e probe requests WiFi. O sensor envia as medições coletadas, estruturadas no formato JSON, a um broker MQTT. Os metadados dos sensors: nome, latitude, longitude, são armazenados num banco chave-valor para posterior utilização. Esses dados são ingeridos pelo pipeline de dados que será apresentado nas próximas seções.

Figura 8 – Dispositivo sensor Raspberry

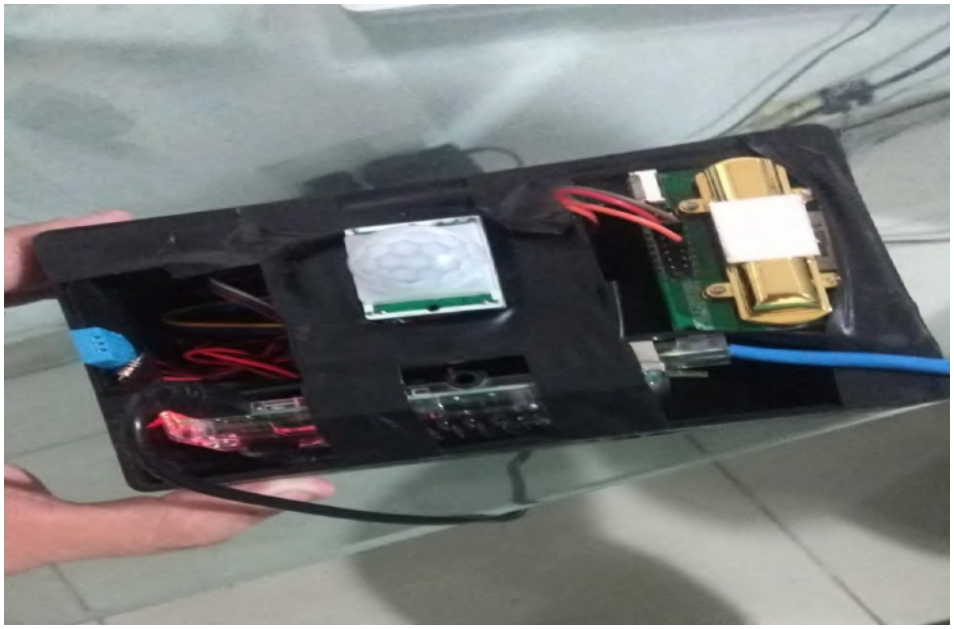


Tabela 1 – Dados de telemetria

Campo	Descrição	Exemplo
sensor	MAC address do sensor coletor	B8:27:EB:F2:64:A9
temp	temperatura em Celsius	26.4
co2	concetração de Dióxido de Carbono em partes por milhão (ppm)	599
hum	umidade relativa do ar, de 0 a 100	46
timestamp	data e hora da coleta do dado	1560798283

Tabela 2 – Dados de probe request

Campo	Descrição	Exemplo
sensor	MAC address do sensor coletor	B8:27:EB:F2:64:A9
SSID	nome da rede alvo do probe request	“LabLegal”
srcMac	MAC do disposito que envia o	00:19:B9:FB:E2:58
dstMac	MAC alvo do probe request	ff:ff:ff:ff:ff:ff
RSSI	intensidade do sinal de retorno em dBm	-55
Channel	canal em que o probe foi enviado	8
timestamp	data e hora da coleta do probe	1560798283

4.2 *BROKER* MQTT

O broker MQTT é o componente que recebe as informações enviadas pelos sensores. Além de receber esses dados, o broker os disponibiliza em uma estrutura de tópicos atualizada em tempo real. Nesse trabalho, foi utilizado o broker Mosquitto, um projeto open source da fundação Eclipse, que implementa o protocolo MQTT (MOSQUITTO.ORG, 2019). No entanto, a arquitetura proposta nesse trabalho é compatível com qualquer outro broker que implemente o protocolo MQTT.

Foram considerados dois tópicos, um para dados de telemetria e outro para dados de probe requests. Desta forma, os diferentes pipelines criados processam exclusivamente os dados pertinentes.

4.3 INGESTÃO DE DADOS

A etapa de processamento do stream de dados é uma das mais importantes. Ela deve ser executada de forma eficiente a fim de mitigar os atrasos entre a coleta do dado e sua exibição. Para essa finalidade, foi utilizado o software open source StreamSets Data Collector (SDC) (STREAMSETS, 2019). Ele provê uma interface gráfica que permite a criação de pipelines de dados de forma rápida e eficiente, contando com diversas fontes de dados, etapas de transformação e outputs. Além de diminuir substancialmente o tempo de desenvolvimento, ele conta com mecanismos de tratamento de erro e recuperação de falhas. Todo pipeline criado nessa ferramenta deve ter pelo menos uma origem e um destino, além de etapas intermediárias opcionais de transformação de dados.

Neste trabalho, a origem de todos os pipelines é um assinante MQTT. Abaixo detalharemos os passos do *pipeline* de dados de *probe requests*:

1. Assinar o tópico de *probes*. Os passos seguintes são executados a cada mensagem recebida;

2. Descartar registros duplicados;

a) Em uma janela de dez segundos, registros que têm mesmo **sensor** e **srcMac**, ou seja, *probe requests* duplicados enviados por um mesmo dispositivo são descartados.

3. Consultar de dados do sensor;

   a) Na mensagem recebida, o sensor é identificado apenas pelo seu MAC address;

   b) A partir desse identificador, é realizada uma consulta na base de metadados de sensores;

   c) A mensagem é enriquecida com metadados do sensor (latitude e longitude).

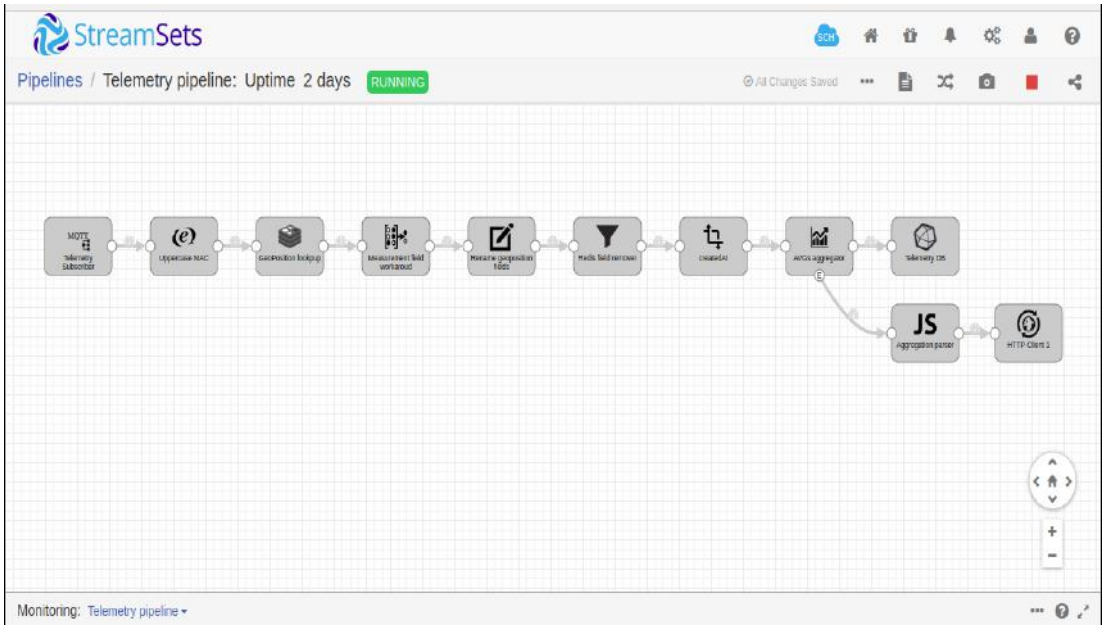
4. Agregar mensagens. Essa etapa tem duas saídas, o fluxo padrão e o fluxo do evento de agregação gerado. Essa agregação tem como finalidade calcular o número estimado de dispositivos por sensor. Para isso, é utilizado uma função de agregação contadora que considera registros numa janela de tempo de um minuto. A cada 15 segundos é gerado um evento com o resultado da agregação;

5. Processar mensagens recebidas da etapa anterior.

   a) Persiste a mensagem recebida do fluxo padrão no banco de dados temporais, InfluxDB;

   b) Envia para a API uma requisição HTTP que transmite para todos os clientes o valor da agregação obtida da etapa anterior.

Figura 9 – Pipeline de dados de telemetria



4.4 ARMAZENAMENTO

Duas estruturas distintas foram utilizadas para armazenar os dados da aplicação desenvolvida. A primeira, o banco de dados em memória Redis[3] foi utilizado para armazenar os metadados de um sensor, a "coisa" no contexto de IoT que foi aplicado nesse estudo.

A estrutura utilizada no armazenamento do conjunto de metadados foi a de *hashes* - uma estrutura chave-valor cuja complexidade computacional de acesso é constante. A chave é o indentificador de um sensor, seu MAC address, e o valor são seus respectivos metadados. Essa estrutura foi escolhida principalmente para possibilitar uma melhor performance nas consultas realizados na etapa de processamento do pipeline de dados, apresentada na Seção 4.3. A Tabela 3 descreve os dados armazenados neste banco. Esses metadados são manipulados pelo usuário, a partir da interface web apresentada na Seção 4.6. Nesta interface é possível criar, editar e excluir sensores, utilizando os componentes da interface que acionam os métodos da API que sincronizam os dados com o Redis.

Tabela 3 – Metadados armazenados no Redis

Campo	Descrição	Exemplo
MAC	MAC address identificador do sensor	B8:27:EB:F2:64:A9
Name	nome do sensor	Sensor sala
Description	breve descrição do sensor	Sensor de temperatura e umidade da sala de estar
Latitude	geoposição em graus decimais	-22.48795
Longitude	geoposição em graus decimais	-43.56623

A segunda estrutura, utilizada para o armazenamento das informações coletadas foi o InfluxDB[4], um banco de dados de séries temporais já apresentado na Seção 2.6.2. Por ser um banco otimizado para armazenamento de dados temporais, ele se torna uma ótima escolha para manipular os dados gerados pelos sensores apresentados. No InfluxDB foram criadas duas *measurements*, o análogo a tabelas no contexto de bancos relacionais. A primeira armazena dados relativos às condições ambientais - temperatura, umidade e concentração de dióxido de carbono, chamada *telemetry*. A segunda, chamada *probe*, dados associados aos probe requests, como os endereços MAC de origem e destino do probe, o RSSI, dentre outros. As Tabelas 4 e 5 descrevem os campos armazenados.

Os dados de leitura dos sensores são ingeridos automaticamente, a partir do fluxo apresentado nas seções anteriores. Eles são lidos pelo sensor, descrito na Seção 4.1, que por sua vez enviam os dados ao *broker* MQTT (Seção 4.2). Depois disso, os dados são

[3] redis.io. **Redis**. Disponível em: https://redis.io/. Acesso em: 30 jul.2019

[4] InfluxData. **InfluxDB**. Disponível em: https://www.influxdata.com/products/influxdb-overview/. Acesso em: 30 jan.2020

Tabela 4 – Séries de dados de probe requests armazenadas no InfluxDB

Campo	Descrição	Exemplo
sensor	MAC address identificador do sensor coletor	B8:27:EB:F2:64:A9
srcMac	MAC address identificador do dispositivo de origem	63:AA:1D:C9:EC:39
dstMac	MAC address identificador do dispositivo de destino	FF:FF:FF:FF:FF:FF
SSID	identificador da rede sem fio de destino	LabLegal
RSSI	intensidade do sinal de retorno em dBm	-60
Channel	canal Wi-Fi no qual o probe foi coletado	11
Latitude	geoposição do sensor em graus decimais	-22.48795
Longitude	geoposição do sensor em graus decimais	-43.56623
timestamp	timestamp UNIX da coleta do dado	1561465254

Tabela 5 – Séries de dados de condições climáticas de armazenadas no InfluxDB

Campo	Descrição	Exemplo
sensor	MAC address identificador do sensor	B8:27:EB:F2:64:A9
temp	temperatura em graus Célcius	22
co2	concentração de dióxido de carbono	699
hum	umidade relativa do ar	10
Latitude	geoposição do sensor em graus decimais	-22.48795
Longitude	geoposição do sensor em graus decimais	-43.56623
timestamp	timestamp UNIX da coleta do dado	1561465254

ingeridos pelo *pipeline* de tratamento de dados (Seção 4.3), que processa os registros recebidos e os armazena no banco de dados de série temporal 2.6.2.

4.5 A API

A API foi desenvolvida na linguagem de programação Go, utilizando a arquitetura REST na construção de suas interfaces, disponibilizadas via protocolo HTTP. As informações trafegam pela API no formato JSON[5]. Ela foi criada de forma desacoplada, permitindo que os outros componentes acessem de maneira única e padronizada os dados armazenados. Essas características também facilitam a inserção de novos componentes na solução. Imagine que uma nova aplicação, um aplicativo móvel Android, seja imple-

[5] json.org. **JSON**. Disponível em: https://www.json.org/. Acesso em: 15 jul.2019

mentado. Todas as interfaces da API disponíveis poderão ser acessadas, sendo necessário apenas o desenvolvimento dos recursos específicos da aplicação, como as interfaces gráficas e os mecanismos de comunicação, por exemplo. O Quadro 1 apresenta os métodos que foram implementados para cada recurso.

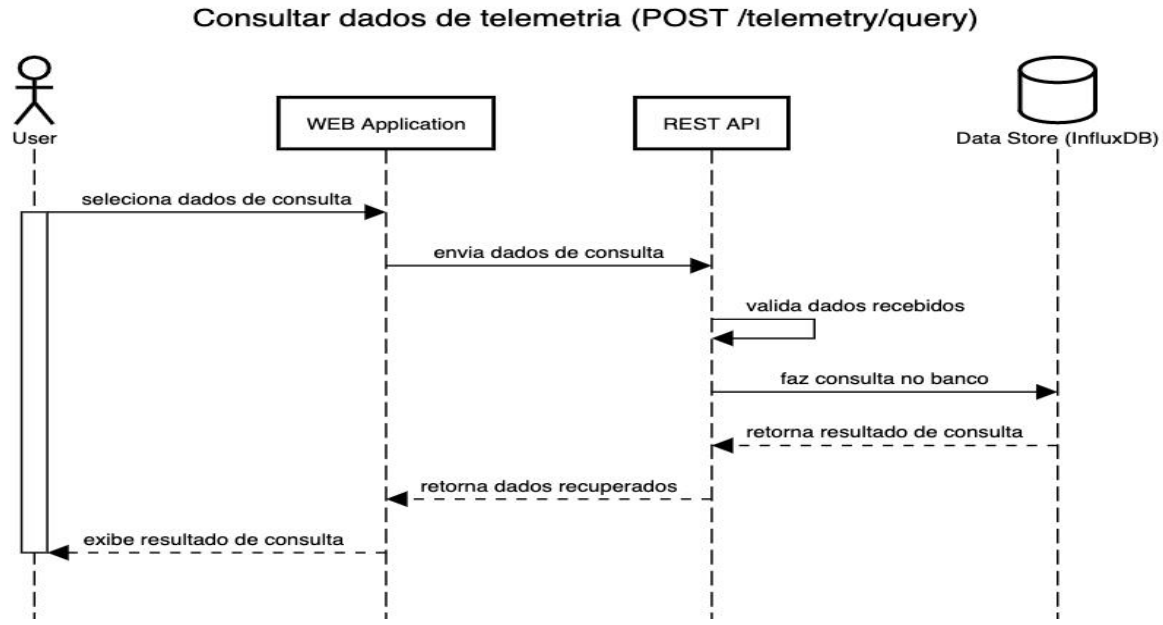
Quadro 1 – Métodos da API

	VERBO	RECURSO	DESCRIÇÃO
/sensors	GET	Sensor	Recupera todos os sensores
/sensors	POST	Sensor	Cria um novo sensor
/sensors/:mac	PUT	Sensor	Atualiza um sensor
/sensors/:mac	DELETE	Sensor	Remove um sensor
/probes	POST	Probe	Adiciona uma coleta de probe
/probes/query	POST	Probe	Recupera dados de probe
/telemetry	POST	Telemetria	Adiciona uma coleta de telemetria
/telemetry/query	POST	Telemetria	Recupera dados de telemetria
/broadcast/telemetry	POST	Telemetria	Notifica todos os clientes WS
/broadcast/probe	POST	Probe	Notifica todos os clientes WS
/ws/telemetry	POST	Telemetria	Cria uma nova conexão WS
/ws/probe	POST	Probe	Cria uma nova conexão WS

Na solução proposta, a API é acessada a partir do website, apresentado na Seção 4.6. Isso é exemplificado na Figura 6. O website utiliza principalmente os métodos conhecidos como CRUD (acrônimo em inglês para criação, leitura, atualização e remoção) aplicados sobre sensores. Além disso, a aplicação web mantém ainda uma conexão permanente com a API através de um websocket para recepção dos dados coletados pelos sensores em tempo real.

O diagrama da Figura 10 exemplifica uma chamada à consulta de dados de telemetria. Essa operação de consulta é disparada pelo usuário a partir da seleção de um intervalo de tempo através da interface Web. Os dados são então enviados para a API em uma requisição HTTP. A API por sua vez processa os dados recebidos, validando-os e os utilizando na consulta realizada no banco de dados de séries temporais, onde os dados de telemetria são armazenados. O resultado dessa consulta é então retornado à interface Web, apresentado na forma de gráficos para o usuário.

Figura 10 – Diagrama de sequência de chamada da API



4.6 O WEBSITE - SENSEIN

As etapas de coleta, processamento e armazenamento dos dados que foram introduzidas nas seções anteriores, são executadas de forma autônoma e contínua em segundo plano. Já a apresentação dos dados coletados, manipulação dos metadados de sensores e consultas às medições armazenadas são feitas de forma ativa através de um interface gráfica. Para isso, desenvolvemos a SenseIn, uma aplicação de página única (conceito introduzido na Seção 2.8), utilizando a linguagem de programação JavaScript e o *framework* Vue.JS, por ser um *framework* leve, flexível e de fácil aprendizado. Além do *framework* citado, foram utilizados ainda componentes da biblioteca Vuetify[6]. Essa biblioteca implementa o estilo de design Material[7], proposto pelo Google e utilizado como guia na construção de experiências digitais de alta qualidade. Esta biblioteca foi utilizada para facilitar o desenvolvimento da interface web, pois ela já conta com diversos componentes reutilizáveis, como botões, menus e formulários.

Para exibição de mapas, foi utilizada a API do Google Maps[8]. Essa aplicação se comunica ativamente com a API apresentada na Seção 4.5 através de chamadas HTTP assíncronas e conexões WebSocket. A interface Web foi dividida em três principais módulos que serão descritos a seguir:

[6] vuetifyjs.com. **Vuetify**. Disponível em: https://vuetifyjs.com/en/getting-started/quick-start. Acesso em: 30 jul.2019

[7] vuetifyjs.com. **Vuetify**. Disponível em: https://material.io/design. Acesso em: 05 dez.2020

[8] Google. **Google Maps JavaScript API**. Disponível em: https://developers.google.com/maps/documentation/javascript/tutorial. Acesso em: 30 jul.2019

4.6.1 Mapa

Nesse componente, foram implementadas as funcionalidades de manipulação dos sensores e recebimento de atualizações em tempo real das medições de dados de telemetria e presença, diretamente no mapa. Essa interface foi criada para simplificar a manipulação dos sensores, já que não é necessário informar dados georreferenciados - como latitude e longitude - mas somente clicar na posição desejada do mapa. Entrando nesse modo, a aplicação WEB abre duas conexões *WebSocket*, uma para dados de telemetria e outra para dados de presença, permitindo assim que as informações que chegam nos componentes expostos anteriormente sejam apresentadas quase que instantaneamente. Além disso, é a partir desse modo de visualização que se dão as chamadas HTTP de criação, edição e remoção dos sensores para a API.

Figura 11 – Interface de criação de sensor

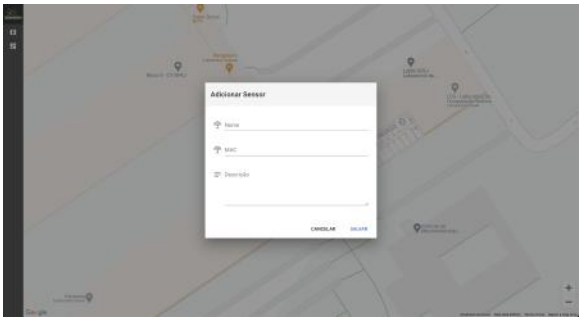
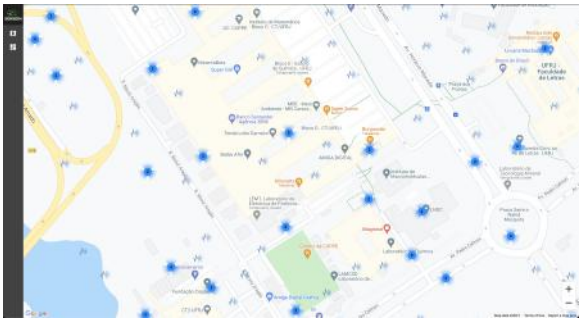


Figura 12 – Visualização no mapa

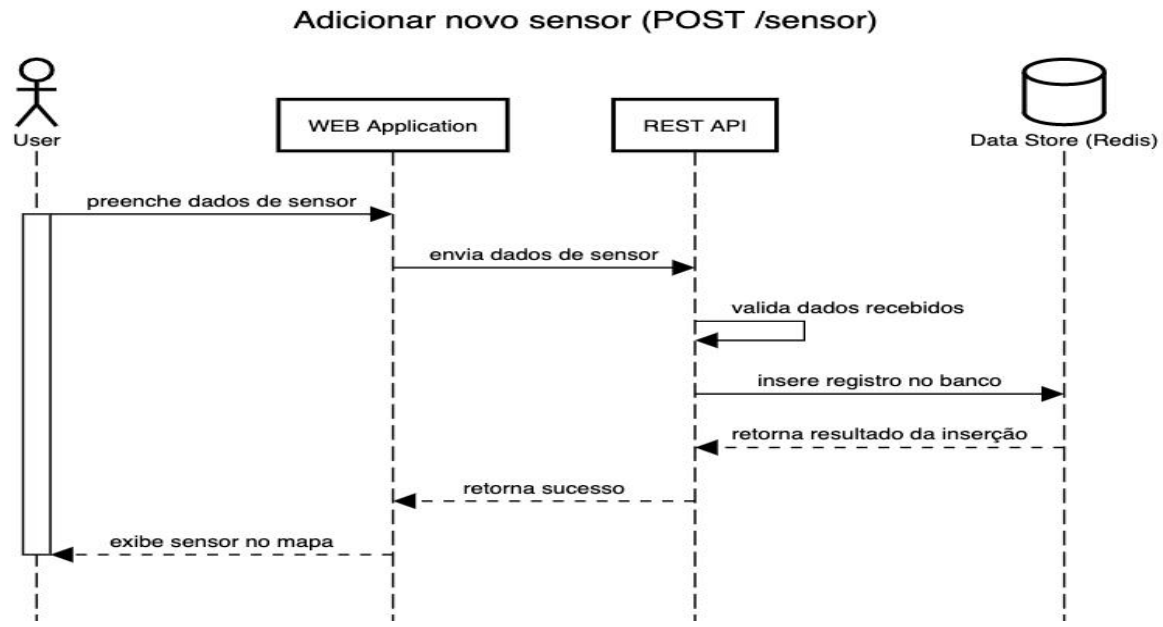


Ao clicar com o botão direito sobre uma posição qualquer do mapa, um menu se abre. Caso a posição seja sobre um sensor já existente, as possibilidades do menu são de editar ou excluir o sensor em questão. Caso contrário, um botão de criação de um novo sensor é apresentado. Ao selecionar a opção de adicionar um sensor, por exemplo, um modal é apresentado para que sejam preenchidos os dados de um sensor - nome, endereço MAC e descrição. Além disso, os dados de geo-posição são capturados a partir da posição no mapa.

A Figura 11 demonstra o modal de adição de um sensor, descrito anteriormente. Já na Figura 12 é possível ver os detalhes de um sensor, como seu nome e descrição cadastrado, além dos dados de número de dispositivos na proximidade, temperatura, umidade relativa e concentração de CO2. Os dados preenchidos são então enviados para a API numa requisição HTTP assíncrona, como exemplificado no diagrama da Figura 13.

Uma importante funcionalidade provida por este componente é a inserção de plantas baixas no mapa. Essa funcionalidade permite que a contextualização espacial do sensor frente ao ambiente físico que ele está inserido. Dessa forma, é possível observar a distribuição dos sensores em salas de um prédio. O sistema é capaz de receber os metadados de posição desta planta, bem como o arquivo de imagem que a define. A interface desenvolvida suporta o envio de inúmeras plantas, tantas quantas forem necessárias. Os metadados destas plantas são armazenadas também no banco de dados chave-valor Redis.

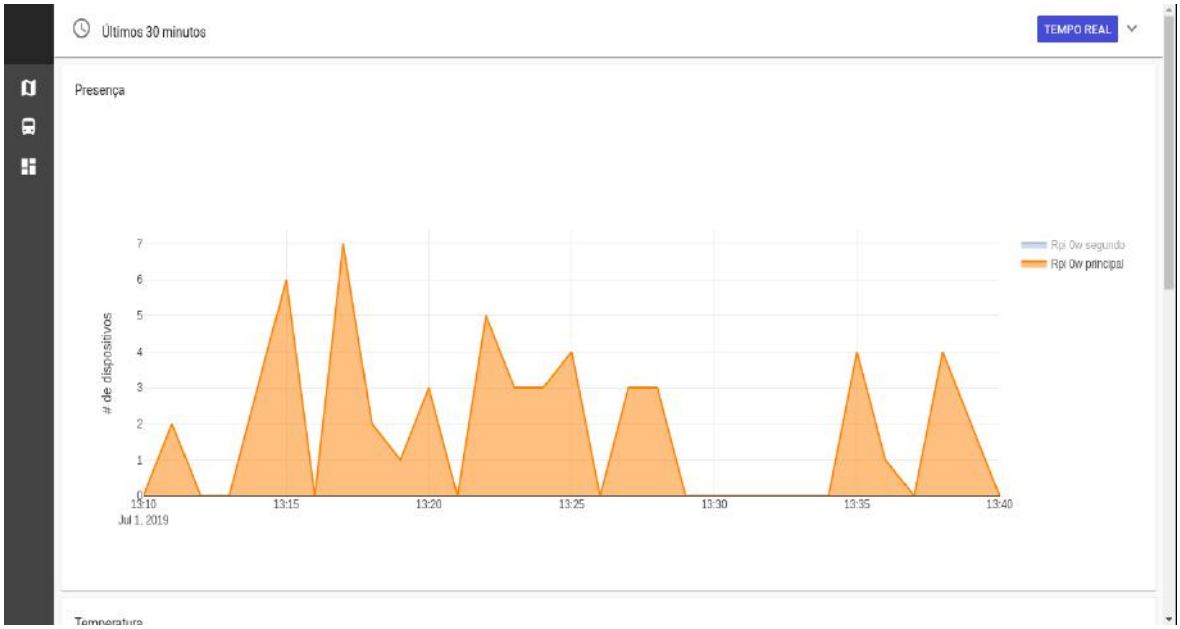
Figura 13 – Diagrama de sequência de chamada da API



4.6.2 Gráficos

No segundo módulo de exibição, as principais ações contempladas são as de consultas a dados em tempo real ou a partir de um período de tempo selecionado. Os gráficos foram criados utilizando a biblioteca Plotly[9]. Assim como introduzido na Seção 4.6.1 nessa tela também são abertas duas conexões WebSocket, se diferenciando da visualização no mapa apenas na forma de exibição da informação recebida, já que nesse modo, a cada novo registro recebido na conexão WS ativa o gráfico é atualizado.

Figura 14 – Interface de visualização em gráfico



[9] plot.ly. **Plotly**. Disponível em: https://plot.ly/javascript/. Acesso em: 30 jul.2019

Foram criados quatro gráficos para exibição das informações coletadas, sendo eles:

1. Gráfico de presença;
2. Gráfico de temperatura;
3. Gráfico de umidade relativa do ar;
4. Gráfico de concentração de CO2.

Para o gráfico de presença, o número de dispositivos é calculado a partir da média móvel com 5 janelas de um minuto. Esse cálculo é feito diretamente na consulta ao banco de dados banco de dados de série temporal, a partir da consulta apresentada no Código 1.

Código 1 – Consulta para cálculo de presença

```
1 SELECT CEIL(MOVING_AVERAGE(COUNT("srcMac"),5))
2 FROM "probe_data"
3 WHERE time >= whereStartTime AND time <= whereEndTime
4 GROUP BY sensor, time(1m)
```

Essa consulta faz uma contagem aplicando a função COUNT no *campo* **srcMac**, que representa o endereço MAC do dispositivo cujo probe request foi capturado. O resultado da contagem é então utilizado no cálculo da média móvel com 5 intervalos, utilizando a função MOVING_AVERAGE, e finalmente, a função CEIL é aplicada para aproximar o valor da média para o próximo inteiro. Os dados são selecionados da *measurement* **probe_data**, filtrados a partir da data de leitura dos *probes* e agrupados pela *tag* **sensor** em janelas de um minuto. A Figura 14 mostra o resultado dessa consulta em um intervalo de trinta minutos.

Vale lembrar que o mecanismo de cálculo de presença utilizado nesse trabalho é bastante simplista, não contemplando as especificidades que tal métrica exige (OLIVEIRA et al., 2019). Considerando também que não é objetivo deste trabalho oferecer uma métrica de estimativa de presença, mas somente uma arquitetura computacional robusta para tal, optamos por utilizar esse mecanismo simplista apenas a fim de expor uma das possíveis aplicação da arquitetura apresentada.

Código 2 – Consulta para cálculo de temperatura

```
1 SELECT MEAN("temp")
2 FROM "telemetry_data"
3 WHERE time >= whereStartTime AND time <= whereEndTime
4 GROUP BY sensor, time(1m)
```

Os gráficos de dados do ambiente (temperatura, umidade e CO2) seguem a mesma lógica, porém tem suas consultas simplificadas. Os dados são selecionados da *measurement*

telemetry_data, filtrados por data e agrupados pela *tag* **sensor** em janelas de um minuto. A grande diferença com a abordagem anterior, da consulta de quantidade de dispositivos presentes por sensor, está nas funções aplicadas. Nesse caso, apenas a função de média é utilizada a partir do campo a ser selecionado. O Código 2 apresenta a consulta no campo de temperatura, representado pelo *field* **temp** na modelagem utilizada.

5 ESTUDO DE SIMULAÇÃO

Neste capítulo apresentaremos um estudo de simulação utilizando a aplicação MVP apresentada no capítulo anterior. Utilizaremos o *SenseIn*, sistema desenvolvido nesse trabalho, em dois estudos de simulação: a simulação de um uso real da aplicação, em pequena escala, para que possamos avaliar a funcionalidade da solução; e um teste de carga, onde simularemos cenários de maior fluxo de dados, a fim de avaliar o comportamento da solução sob estresse.

Por conta da política de isolamento e da suspensão das atividades presenciais na universidade, em decorrência da pandemia de COVID-19, a coleta de um conjunto de dados relevante para nosso estudo utilizando um sensor real foi impossibilitada. Com o objetivo de contornar esse obstáculo, optamos então pela utilização de dados sintéticos, no intuito de simular o processo de coleta de dados de um sensor real.

5.1 SIMULAÇÃO DE UM USO REAL

Esta seção tem como foco simular a aplicação da arquitetura proposta num contexto real. Como destacamos na seção anterior, utilizaremos dados sintéticos. É importante salientar que apenas a geração das leituras será artificial; após geradas, as leituras serão publicadas no broker MQTT, apresentado na Seção 4.2, da mesma maneira que leituras coletadas por sensores reais seriam. Todo o resto do processo acontecerá exatamente como numa utilização real.

Trabalharemos com dois tipos de leituras: telemetria — que agrupa dados de temperatura, umidade e CO2 — e *probe request*. A criação dos sensores, assim como os algoritmos utilizados para a geração das leituras serão melhor descritos a seguir.

5.1.1 A criação dos sensores

Para que possamos nos aproximar de um uso real, optamos por criar os sensores de forma manual, utilizando o sistema *SenseIn*.

Ao todo, para essa simulação, foram criados doze sensores. Utilizamos como base a planta baixa dos nossos laboratórios, localizados no Centro de Tecnologia da Universidade Federal do Rio de Janeiro; posicionamos, ao total, oito sensores nessa região, um em cada laboratório. Dos quatro sensores restantes, dois foram posicionados em restaurantes próximos, e os outros dois em pontos de ônibus.

A Figura 15 mostra os oito sensores posicionados nos laboratórios. A Figura 16 mostra, no mapa, todos os doze sensores que foram criados para esta simulação. Note que, por conta da proximidade dos sensores e da diminuição do zoom, alguns sensores aparecem agrupados.

Figura 15 – Sensores posicionados nos laboratórios, utilizando a planta baixa

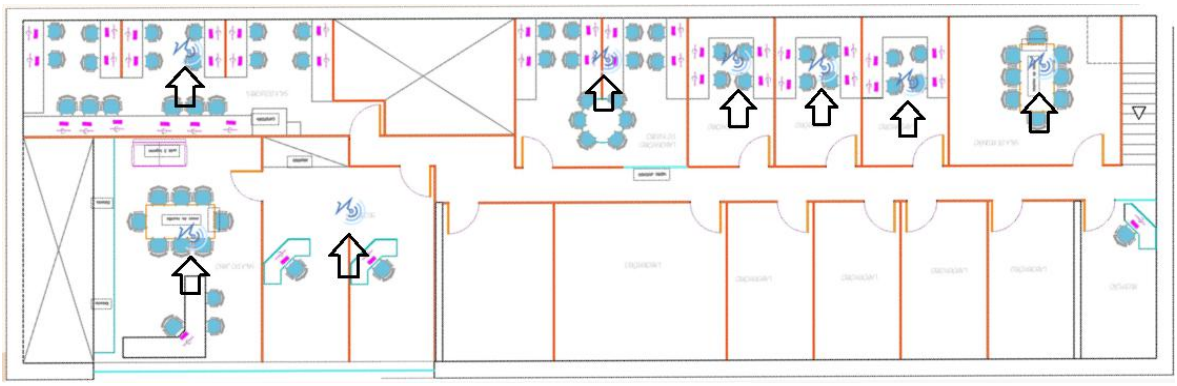
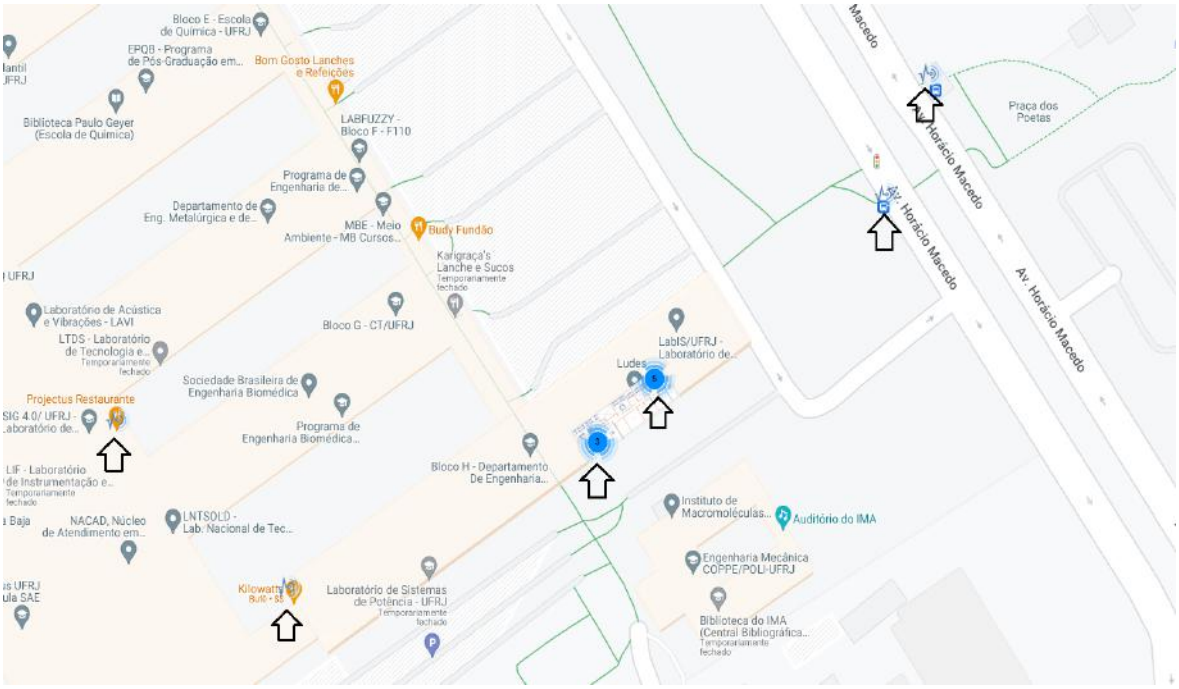


Figura 16 – Os doze sensores criados para a simulação



5.1.2 Geração de leituras de telemetria

O Algoritmo 1 detalha a geração de leituras de telemetria. A implementação do algoritmo em python está disponível no Apêndice B.

Algoritmo 1 Geração de leituras de telemetria

```
1: Ler lista de sensores do arquivo de entrada
2: while true do
3:     for cada sensor s do
4:         Gerar e publicar leitura aleatória de temperatura, umidade e CO2
5:     end for
6:     Aguardar T segundos
7: end while
```

A constante T representa o tempo que o algoritmo espera, em segundos, até executar

a próxima iteração.

A simulação das leituras de telemetria é simples, pois em uma condição real, o sensor poderia realizar a leitura a qualquer momento, visto que depende apenas do ambiente em que está colocado para tal. Dessa maneira, não é necessária nenhuma lógica extra para a geração dessas leituras.

Veremos, a seguir, que a simulação de leituras de *probe request* tem uma complexidade maior, pois em uma situação real, depende da presença de dispositivos nas proximidades do sensor, o que a torna mais dinâmica e imprevisível.

5.1.3 Geração de leituras de *probe request*

O Algoritmo 2 detalha a geração de leituras de *probe request*. A implementação do algoritmo em python está disponível no Apêndice C.

Algoritmo 2 Geração de leituras de *probe request*

```
1: Ler lista de sensores do arquivo de entrada
2: while true do
3:     for cada sensor s do
4:         Gerar aleatoriamente n_s novos dispositivos próximos
5:         for cada dispositivo próximo gerado d do
6:             Gerar p_ini e atribuir à presença de d
7:             Incluir d na lista de dispositivos próximos do sensor s
8:         end for
9:         for cada dispositivo próximo d do
10:            Gerar e publicar leitura aleatória de probe request
11:            Subtrair aleatoriamente 0 ou 1 da presença de d
12:            if presença de d é igual a 0 then
13:                Remover d da lista de dispositivos próximos do sensor s
14:            end if
15:        end for
16:    end for
17:    Aguardar T segundos
18: end while
```

O algoritmo implementa uma lógica para simular a presença de cada dispositivo próximo ao sensor, além da possibilidade de chegada ou não de novos dispositivos nas proximidades do sensor, a cada iteração.

Nós definimos, através da Fórmula 5.1, o valor p_{ini}, que é o valor de presença inicial de um dispositivo próximo gerado, para um determinado sensor s.

$$p_{ini} = rand_{int}(1, P_{max}) \tag{5.1}$$

onde:

$rand_{int}$ = função que retorna aleatoriamente um valor inteiro entre dois números
P_{max} = presença inicial máxima de um dispositivo gerado

Por outro lado, nós definimos, através da Fórmula 5.2, o valor n_s, que é a quantidade de novos dispositivos próximos gerados para um determinado sensor, em uma iteração.

$$n_s = max(0, rand_{int}(-D_{max}, D_{max})) \tag{5.2}$$

onde:

max = função que retorna o valor inteiro máximo entre dois números
$rand_{int}$ = função que retorna aleatoriamente um valor inteiro entre dois números
D_{max} = quantidade máxima de novos dispositivos gerados por sensor/iteração

Note que, apesar de P_{ini} representar o valor máximo de presença inicial que pode ser associada a um novo dispositivo próximo gerado d, essa presença pode ser prorrogada ou não, aleatoriamente, de acordo com a linha 11 do Algoritmo 2.

Por fim, como na geração de dados de telemetria, temos também a constante T, que representa o tempo que o algoritmo espera, em segundos, até executar a próxima iteração.

5.1.4 Execução e resultados

Esta seção apresentará a execução da simulação e seus resultados. Realizamos uma execução contínua dos algoritmos de geração de leituras de telemetria e *probe request*, durante aproximadamente cinco horas. As leituras foram enviadas ao *broker* MQTT, como seriam enviadas leituras realizadas por sensores reais. Todo o processamento e visualização dos dados foi realizado pelo sistema *SenseIn*, a nossa implementação da arquitetura proposta nesse trabalho.

A seguir apresentaremos a avaliação do comportamento do sistema, separada em dois tipos de visualização: mapa e gráficos. Além disso, avaliaremos também o comportamento do sistema realizando leituras em tempo real, bem como leituras passadas, armazenadas em nosso banco de dados.

5.1.4.1 A visualização em mapa

A visualização dos sensores em mapa é a tela inicial do sistema *SenseIn*. Nessa visualização, é possível ver os sensores dispostos geograficamente. Todas as leituras apresentadas nessa tela são realizadas em tempo real, utilizando o protocolo *WebSocket*. A cada cinco segundos, o sistema recebe leituras dos sensores e monta, relativamente, um mapa de calor, para cada sensor ou agrupamento de sensores[1], baseado apenas na quantidade de dispositivos próximos; ou seja, em nosso sistema, dados de telemetria não afetam o mapa de calor.

[1] Agrupamento de sensores são formados dependendo da distância de visualização/proximidade dos sensores. Nesses casos, um número dentro de um círculo azul indica quantos sensores estão agrupados.

Ao clicar em um sensor, podemos observar algumas informações importantes, como o nome do sensor, seu endereço MAC, dados de telemetria e de dispositivos próximos, assim como a data e hora em que esse sensor foi atualizado.

Figura 17 – Visualização dos sensores em mapa, recebendo leituras em tempo real

A Figura 17 mostra uma visualização aberta dos sensores em mapa, recebendo leituras em tempo real, durante a simulação. Note que, no centro da imagem, é possível ver dois agrupamentos de sensores, com cinco e três sensores agrupados, respectivamente. Esses agrupamentos acontecem onde está localizada a planta baixa dos nossos laboratórios, como mostrado com maior detalhe na Figura 15. É possível perceber também que nesses agrupamentos forma-se uma cor vermelha bastante acentuada, em relação aos outros sensores. Isso acontece pois, por se tratar de um agrupamento, há uma maior quantidade de dispositivos próximos, quando comparados com sensores não agrupados.

Na Figura 18, podemos observar, mais de perto, os sensores posicionados na planta baixa dos laboratórios. É possível identificar que o sensor chamado Sala 01 está mais vermelho que os outros sensores, o que indica que, naquele momento, possuía maior número de dispositivos próximos que os sensores posicionados nos outros laboratórios. Podemos observar também os detalhes do sensor Sala 01, já citados anteriormente: seu endereço MAC, dados de telemetria, dispositivos próximos e última atualização.

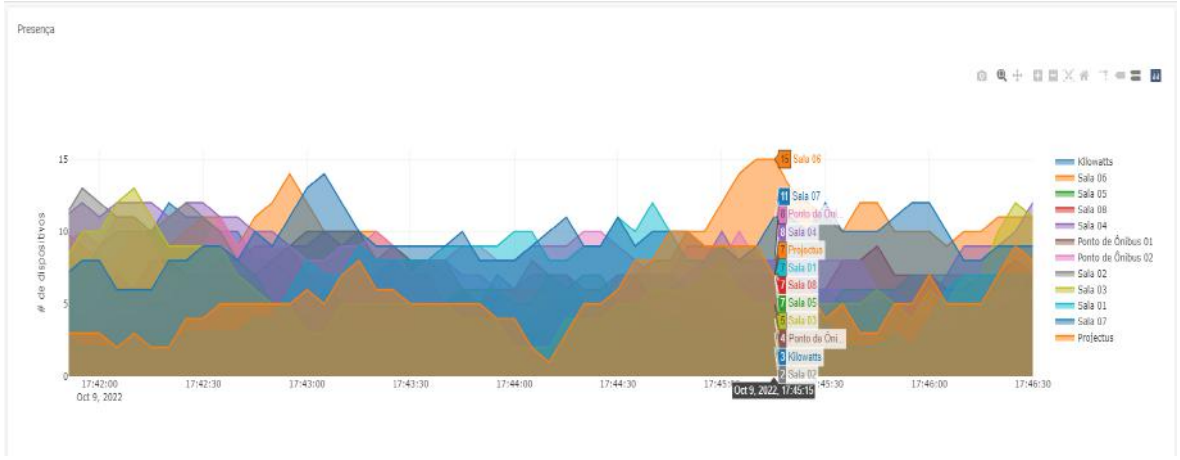
Podemos concluir, então, que na visualização em mapa, o sistema comportou-se corretamente durante a simulação. Foi possível detalhar sensores e visualizar seus dados de telemetria e *probe request*, como apresentado na Figura 18, além de realizar leituras simultâneas em tempo real, em todos os doze sensores criados, como observado na Figura 17.

Figura 18 – Sensores posicionados na planta baixa, recebendo leituras em tempo real

5.1.4.2 A visualização em gráficos

A visualização em gráficos facilita a comparação das leituras entre os sensores. Para a visualização gráfica das leituras em tempo real, os gráficos são atualizados a cada cinco segundos, como na visualização em mapa. É possível também, consultar períodos passados, armazenados em nosso banco de dados de série temporal, apresentado na seção 4.4.

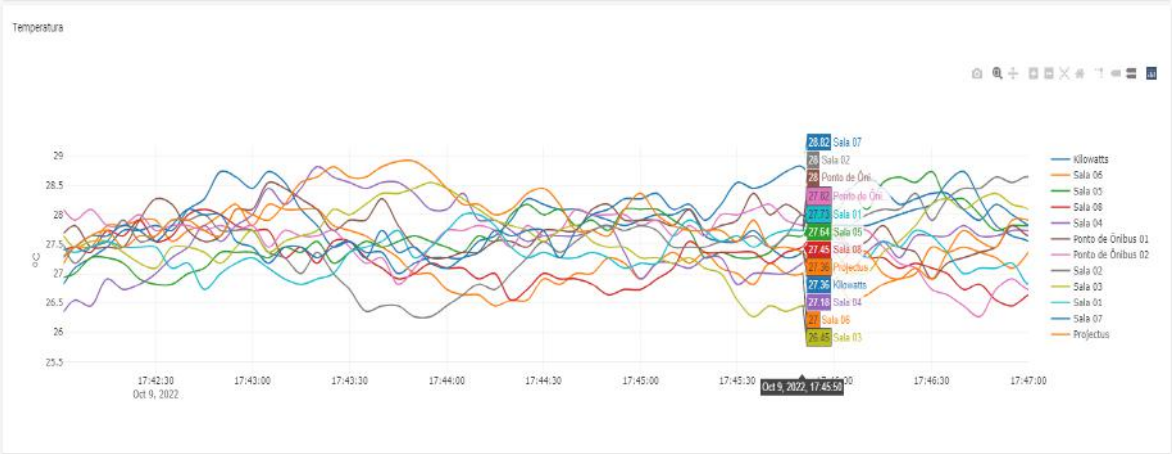
Figura 19 – Gráfico de dispositivos próximos recebendo leituras em tempo real



A Figura 19 mostra o gráfico de dispositivos próximos recebendo leituras em tempo real. É possível identificar que quase sempre há variação a cada intervalo de tempo, característica do algoritmo de geração de leituras, que executa uma nova iteração a cada cinco segundos. Note que, ao passar o mouse em cima do mapa, é possível detalhar um

intervalo de tempo específico. Na Figura 19, detalhamos um intervalo de tempo onde o sensor Sala 06 possuía quinze dispositivos próximos, enquanto o sensor Sala 02 possuía apenas dois.

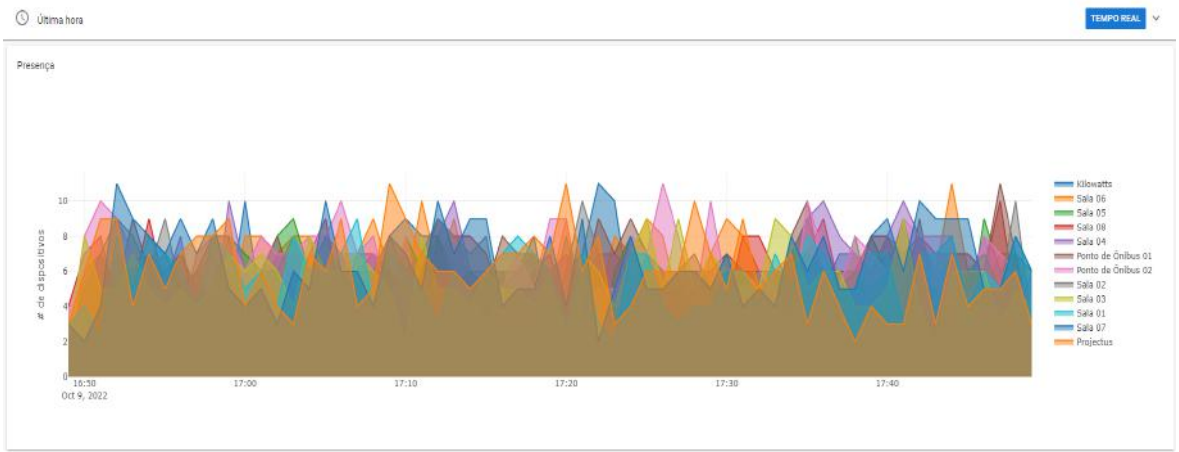
Figura 20 – Gráfico de temperatura recebendo leituras em tempo real



A Figura 20 mostra o gráfico de temperatura – um dos dados de telemetria – recebendo leituras em tempo real. Observe que, apesar de as linhas dos gráficos serem contínuas, temos informações apenas dos instantes de tempo. A ligação entre os pontos é feita apenas para facilitar a leitura dos gráficos e não representa a realidade.

A Figura 21 mostra o gráfico de sensores próximos num intervalo, passado, de uma hora. Essas leituras, armazenadas em nosso banco de dados, são recuperadas utilizando nossa API, apresentada na seção 4.5.

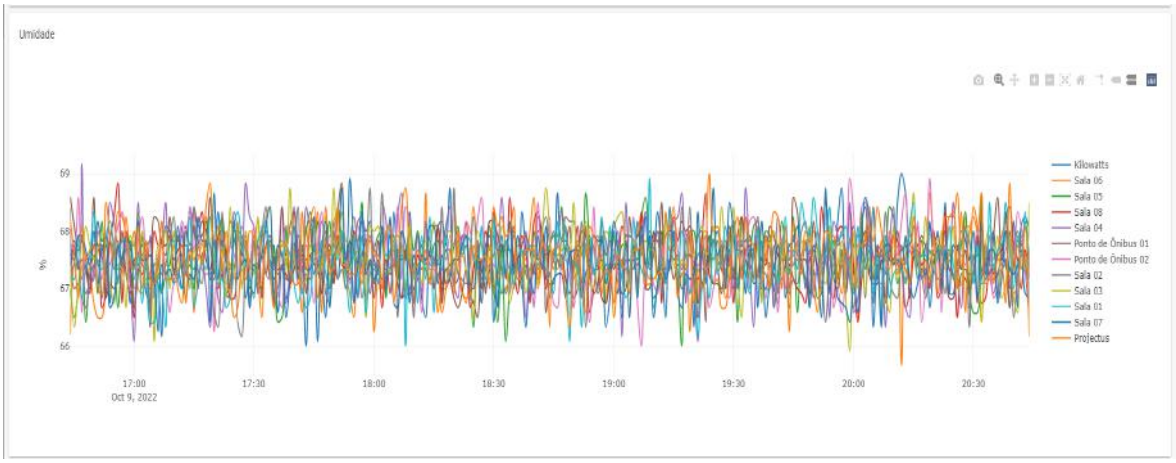
Figura 21 – Gráfico de dispositivos próximos num intervalo, passado, de uma hora



Uma grande diferença presente na visualização de intervalos passados é que, para essa visualização, os intervalos entre os pontos são de um minuto, enquanto que na visualização em tempo real, os intervalos são de cinco segundos. Tomamos essa decisão pois um intervalo de cinco segundos geraria um número muito grande de pontos para intervalos de tempo maiores do que uma hora, prejudicando a visualização das leituras.

Por fim, a Figura 22 mostra o gráfico de leituras de umidade num intervalo passado maior: quatro horas.

Figura 22 – Gráfico de umidade num intervalo, passado, de quatro horas



É correto afirmar que, quanto maior o intervalo de tempo e/ou a variação das leituras, pior fica a visualização dos gráficos, por conta da maior quantidade de pontos a serem distribuídos no espaço. Em vez de simplesmente aumentar o intervalo entre os pontos para um minuto, poderíamos ter diminuído a periodicidade dos pontos proporcionalmente ao intervalo de tempo selecionado, a fim de termos uma melhor visualização das informações.

Podemos concluir, então, que na visualização em gráficos, o sistema comportou-se corretamente durante a simulação. Foi possível visualizar dados de telemetria e *probe request* em tempo real, como apresentado nas Figuras 19 e 20, além de realizar leituras de intervalos passados, como mostrado na Figura 21, para um intervalo de uma hora, e na Figura 22, para um intervalo de quatro horas.

5.2 TESTE DE CARGA

Esta seção apresentará simulações focadas no teste da aplicação MVP SenseIn sob carga, com a intenção de avaliar nossas premissas arquiteturais, e identificar possíveis limitações do sistema. Para realização das mesmas, foi criado um pequeno cluster com dois nós computacionais, utilizando o serviço de computação em nuvem da plataforma DigitalOcean[2]. As características do cluster podem ser consultadas no Quadro 2. Os componentes apresentados no Capítulo 4 foram distribuídos utilizando tecnologias de conteinerização[3] e orquestração de contêineres[4]. Eles operam de forma isolada e se comunicam através de chamadas de rede.

[2] DigitalOcean. **DigitalOcean**. Disponível em: https://www.digitalocean.com/about/. Acesso em: 01 out.2022

[3] RedHat. **O que é um container Linux?**. Disponível em: https://www.redhat.com/pt-br/topics/containers/whats-a-linux-container. Acesso em: 01 out.2022

[4] RedHat. **Kubernetes e a tecnologia de containers**. Disponível em: https://www.redhat.com/pt-br/topics/containers/what-is-kubernetes. Acesso em: 01 out.2022

As simulações concentram-se em duas das funcionalidades do sistema; manipulação de sensores e leitura de dados de telemetria. Serão avaliadas métricas como latência e taxa de processamento, além do funcionamento adequado do sistema sob as condições estudadas.

Quadro 2 – Cluster Kubernetes

NOME	CPU	MEMÓRIA	DISCO
tcc-1-3865w	2 vCPU[5]	4 GB	80 GB
tcc-1-3865c	2 vCPU	4 GB	80 GB

5.2.1 Manipulação de sensores

Essa simulação consiste na criação e apresentação dos sensores virtuais. Para criação, utilizamos uma pequena rotina desenvolvida em Python, que interage com a API apresentada na Seção 4.5, através do *endpoint* de criação de sensores. O código referente à operação descrita pode ser encontrado no Apêndice A. Foram criados até 5000 sensores, a fim de avaliar a solução de armazenamento de metadados de sensores. Introduzidos nos Capítulos 4.1 e 4.4, tais metadados são armazenados no banco chave-valor Redis. Para a recuperação e visualização, utilizamos o *endpoint* de listagem de sensores e realizamos medições do tempo de resposta.

5.2.1.1 Resultados

O espaço em memória utilizado por 1000 sensores foi de aproximadamente 199 Kbytes. A Figura 24 apresenta o gráfico do consumo de armazenamento dos sensores. Nela é possível identificar o padrão linear de crescimento, com um consumo médio de 200 bytes por sensor. Com isso, é possível concluir que a solução para armazenamento de metadados de sensor, utilizando o banco Redis, pode facilmente armazenar milhões de sensores. Um banco redis com capacidade de 4Gb é capaz de armazenar em torno de 21,5 milhões de sensores. Considerando uma escala de bilhões de sensores, é preferível a utilização de um outro tipo de armazenamento, possivelmente um banco de dados não relacional orientado a documentos.

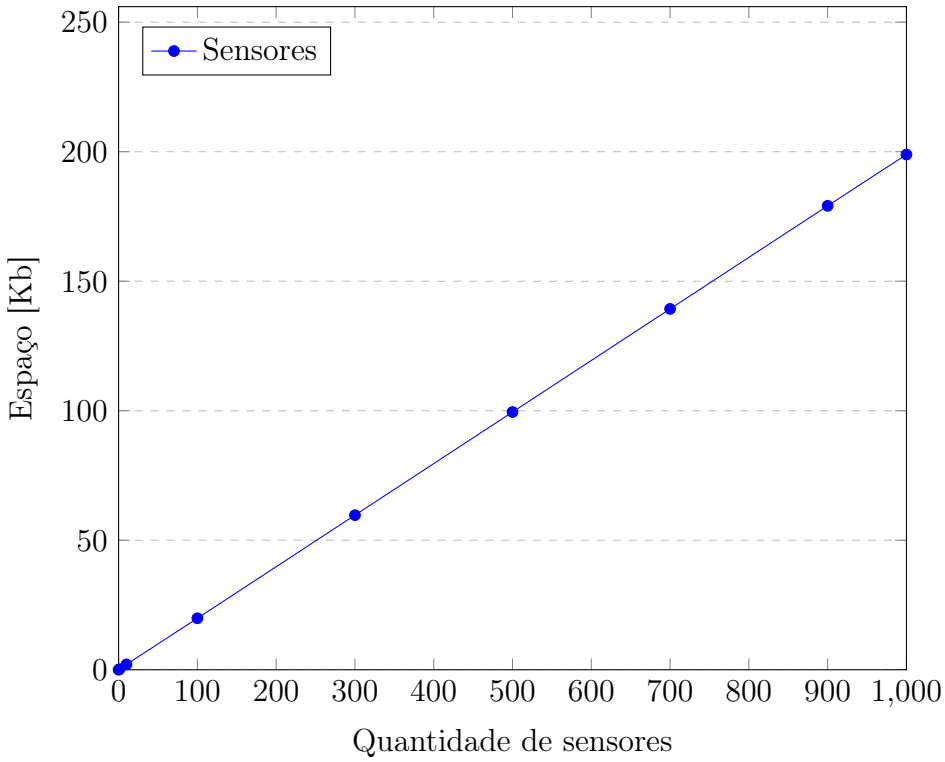
A consulta aos metadados de sensores é realizada através da requisição à API de listagem de sensores, e retorna os metadados dos sensores criados. O Gráfico 25 mostra a duração média de resposta da listagem por quantidade de sensores. Nele é possível verificar a variação do crescimento da curva, com tempos de resposta elevando-se rapidamente à medida que o número de sensores cresce. A Figura 23 traz a distribuição do tempo de resposta na requisição de listagem de 4000 sensores. Analisando o tempo para primeiro

[5] Na plataforma DigitalOcean, uma vCPU é a unidade de processamento correspondente a um hyperthread de um processador.

Figura 23 – Decomposição da requisição de listagem de sensores

byte de 495.49 ms e 96.21 ms de download do conteúdo com o tamanho da resposta de 560kb, percebemos que a aplicação utiliza um tempo considerável recuperando os registros do bando de dados. Isso se deve ao fato do Redis ser um banco otimizado para o acesso direto aos seus registros, e a operação de listar todos os registros é computacionalmente custosa.

Figura 24 – Armazenamento utilizado

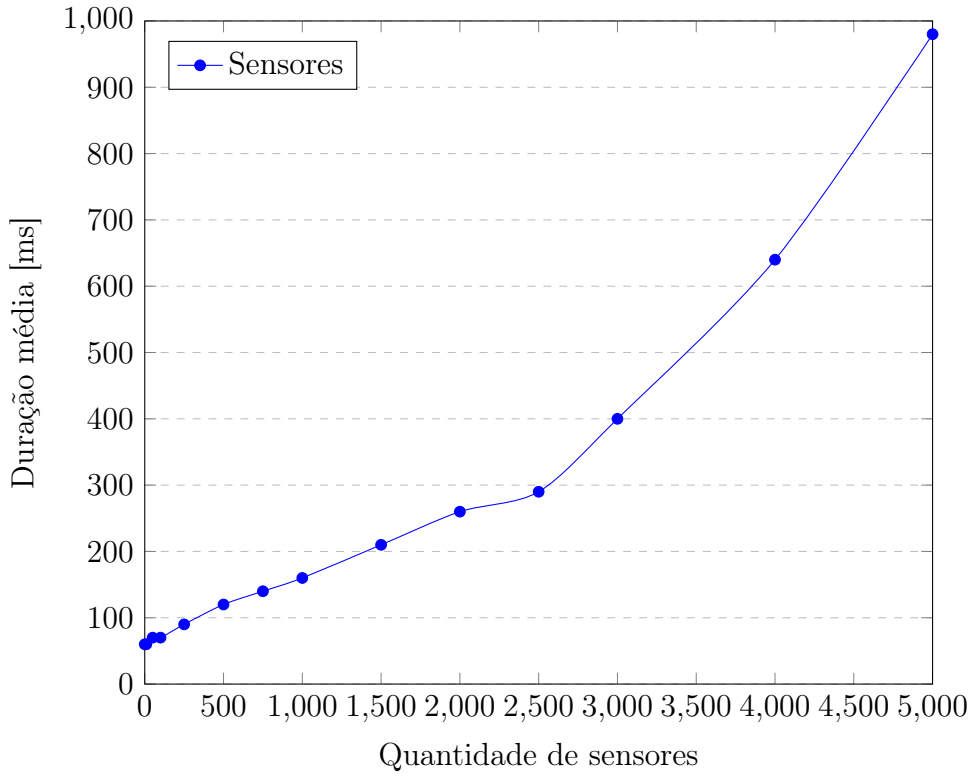


A partir das simulações realizadas, foram detectados problemas na exibição dos sensores; a visualização no mapa e a listagem dos metadados. Para ambos os casos, a utilização de técnicas de paginação se mostra necessária. No caso da visualização em mapa, apresentada na Seção 4.6.1, um número grande de sensores faz com que a renderização se torne lenta, afetando a usabilidade da ferramenta.

Além disso, foi possível perceber que a escolha por um banco chave-valor impacta negativamente na listagem dos sensores, pois um banco desse tipo é otimizado para o acesso direto à um elemento, e não para a listagem dos seus registros, além de não contar

com funcionalidades de busca e índices.

Figura 25 – Tempo de resposta



5.2.2 Geração de dados de telemetria

A simulação apresentada nessa sessão utiliza o Algoritmo 1, descrito inicialmente no Capítulo 5.1.2, que implementa a geração de dados sintéticos de telemetria. Também foi utilizada a ferramenta *mqtt-benchmark*[6]. A análise dos resultados foca em três métricas; taxa de ingestão de dados, armazenamento utilizado e latência na exibição das leituras. Iremos também monitorar a utilização de recursos de cada componente. O Quadro 3 apresenta a utilização de CPU e memória de cada componente em inatividade.

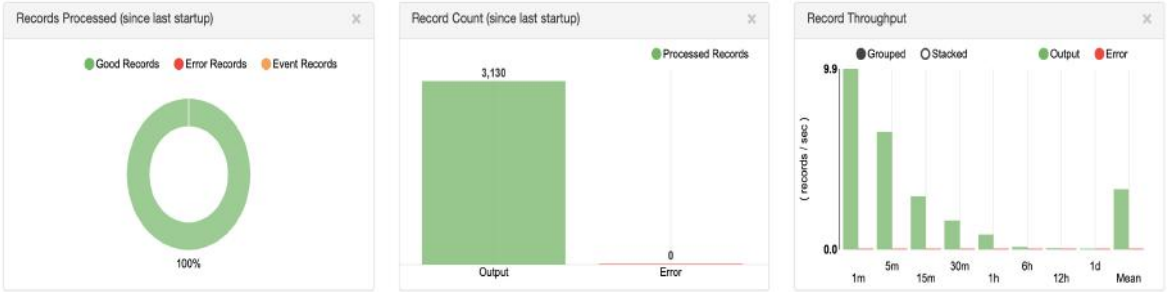
Quadro 3 – Utilização de recursos por componente com sistema em inatividade

Componente	CPU(cores)	Memória(bytes)
api	0m	1Mi
cliente	0m	2Mi
influx	2m	78Mi
mosquitto	1m	17Mi
redis	2m	7Mi
sdc	20m	407Mi

5.2.2.1 Resultados

Executando o Algoritmo 1, foram realizadas simulações com taxa de 10 e 100 leituras por segundo. As Figuras 26 e 27 apresentam métricas coletadas pelo SDC com 10 e 100 mensagens enviadas por segundo, respectivamente. É possível notar que todas as mensagens foram processadas corretamente, e a taxa de publicação de mensagens é a mesma que a taxa de processamento das mesmas. A latência de inserção – tempo entre a geração da leitura, processamento e armazenamento – foi menor que um segundo nesses casos.

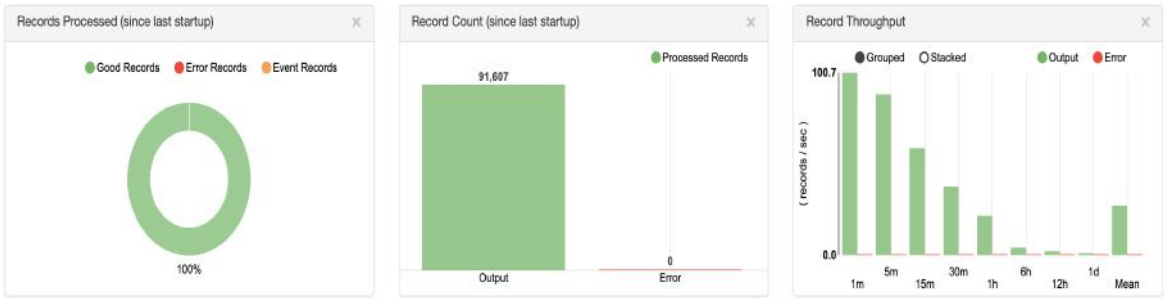
Figura 26 – Métricas SDC com taxa de 10 msg/s



Ao analisar a latência total – tempo entre a geração da leitura e sua visualização – foi também possível registrar tempos menores que um segundo, para simulações com taxa de geração de 100 msg/s. A Figura 28 mostra o gráfico gerado em tempo real dessas

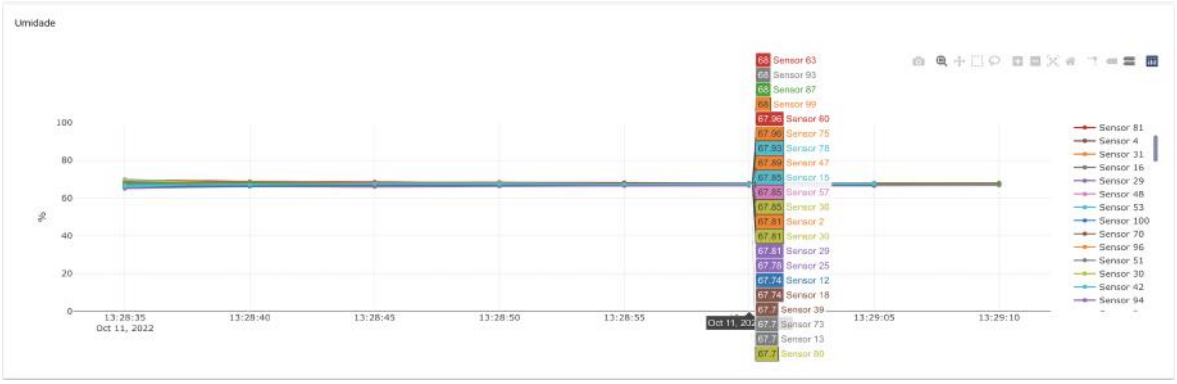
[6] **mqtt-benchmark** para simulação de envio de múltiplos sensores concorrentemente. Disponível em: https://github.com/krylovsk/mqtt-benchmark. Acesso em: 11 out.2022

Figura 27 – Métricas SDC com taxa de 100 msg/s



leituras. Apesar da baixa latência, já é possível notar uma diminuição na responsividade da tela. Além disso, o número elevado de sensores dificulta a usabilidade e legibilidade dos gráficos. Como dito anteriormente na seção 5.2.1, telas desse tipo precisariam de filtros e agregações para melhorar a usabilidade e responsividade.

Figura 28 – Gráfico de umidade de 100 sensores



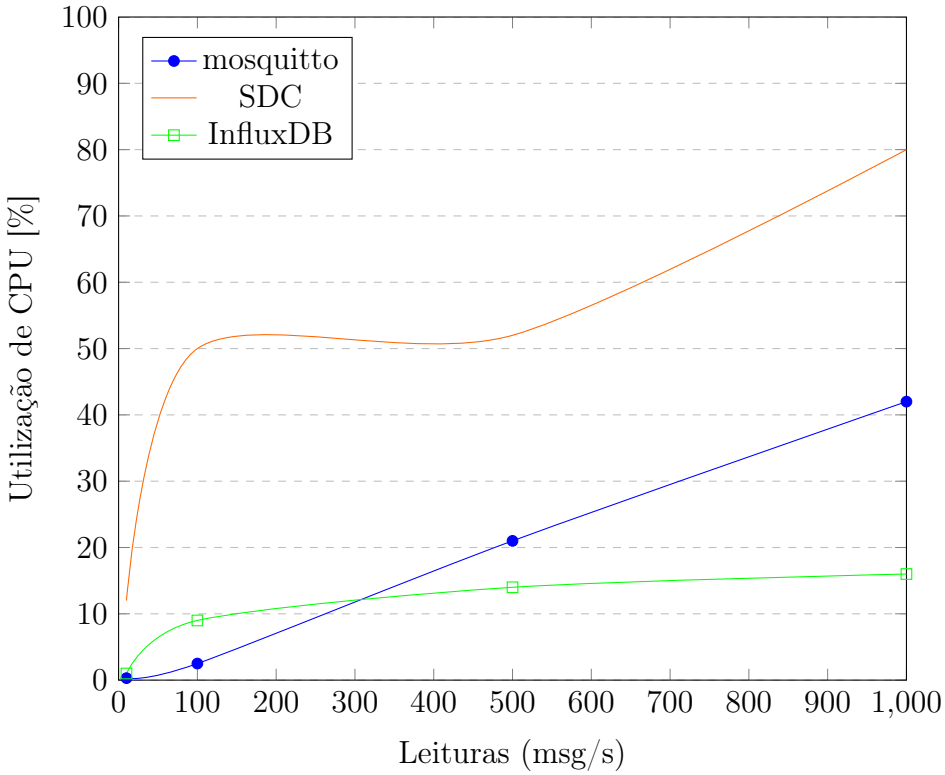
Ao tentar simular um número de leituras mais elevado, uma limitação no Algoritmo 1 foi detectada. Para a publicação de mensagens no *broker* MQTT, uma única conexão é criada, e leituras de múltiplos sensores são enviadas de forma sequencial. Para contornar essa limitação, optamos pela utilização da ferramenta *mqtt-benchmark*. Com ela foi possível criar múltiplos clientes e gerar leituras de forma concorrente. O Quadro 4 apresenta os resultados das simulações com variadas taxas de leitura. Ele contém as taxas de leitura, armazenamento e o número total de mensagens enviadas. Com até 100 mensagens enviadas por segundo, a taxa de leitura e armazenamento são similares, como já foi apontado.

Quadro 4 – Benchmark leitura e processamento de dados de telemetria

Taxa de leitura (msg/s)	Taxa armazenamento (msg/s)	Total mensagens
9.957	9.91	1000
95.530	94.3	10000
480.408	171.8.	50000
981.140	211	300000

Foi possível chegar a taxas de aproximadamente 980 mensagens/segundo publicadas, porém o limite do processamento registrado foi de 211 mensagens/segundo. A publicação de 300000 mensagens por 1000 clientes simulados levou em média 305 segundos (aproximadamente 5 minutos). Isso mostra que o *broker* conseguiu receber as mensagens com sucesso. A ingestão das mensagens, numa taxa máxima de 211 mensagens/segundo, teve duração de 26 minutos. As mensagens não processadas foram enfileiradas e entregues ao SDC para posterior armazenamento.

Figura 29 – Utilização de CPU



Para visualizar a limitação no processamento das mensagens, analisamos também a utilização de CPU dos componentes diretamente relacionados à coleta e inserção das medições. Na Figura 29 podemos ver que com 100 mensagens/segundo, a utilização de CPU do SDC já é de 50%. Com 500 e 1000 leituras, a utilização do *broker* aumenta consideravelmente, pois as mensagens começam a enfileirar devido a baixa taxa de processamento do SDC. A partir dessas informações, concluímos que o componente que processa as mensagens, o SDC, chegou próximo ao seu limite e precisaria ser escalado horizontalmente.

A partir da análise dos resultados das simulações foi possível perceber a importância da comunicação assíncrona e de uma arquitetura com componentes desacoplados. No caso da utilização de uma comunicação síncrona para o envio das leituras, a taxa máxima de recebimento dos dados seria o limite de processamento do SDC. Com a utilização de um *broker* é possível receber mensagens em uma determinada taxa e processá-las em outra, sem bloquear a leitura de novas medições. Além disso, com esse tipo de arquitetura é

possível dimensionar cada componente de acordo com suas limitações. Nesse caso, uma das possibilidades seria aumentar o número de réplicas do SDC (escalar horizontalmente) e/ou disponibilizar mais recursos computacionais para esse componente (escalar verticalmente).

6 CONCLUSÃO

Sensores IoT e dispositivos inteligentes já estão presentes no nosso cotidiano, e sabemos que a previsão de crescimento no números desses dispositivos conectados é avassaladora. Para interagir com os dados gerados por tais dispositivos, foi apresentada uma arquitetura de software voltada para o processo de captura, processamento e armazenamento de dados de sensores. Esse tipo de arquitetura se mostra cada vez mais necessária.

Parte desse trabalho, consistiu no processo de implementação de uma aplicação baseada na arquitetura proposta chamada SenseIn. Além de desenvolver academicamente, esse trabalho influenciou positivamente também no desenvolvimento profissional dos autores, que hoje aplicam o conhecimento adquirido diretamente nos seus respectivos empregos.

Com a realização dos estudos de simulação, apresentado no Capítulo 5, foi possível verificar algumas das premissas apresentadas nos capítulos introdutórios. Além disso, detectou-se a necessidade de filtros e agregações ao se trabalhar com visualização de dados de múltiplos sensores. Também foram identificadas algumas limitações em escolhas como o banco de dados chave-valor Redis para armazenamento dos metadados dos sensores.

Os resultados apresentados na Seção 5.2, na parte de geração de dados de telemetria, mostram a importância de uma arquitetura distribuída e da utilização de comunicação assíncrona na ingestão dos dados de telemetria. A utilização de uma arquitetura distribuída e desacoplada e comunicação assíncrona foi uma escolha se mostrou de extrema importância. Uma arquitetura desse tipo permite escalar componentes de forma independente.

Um dos possíveis trabalhos futuros consiste na implementação de técnicas de fusão de dados na borda da rede de sensores, ou seja, antes do envio para o *broker* MQTT. Essas técnicas tem potencial de melhorar a qualidade da informação gerada por eles (MITCHELL, 2007), e também de reduzir a quantidade de informação a ser armazenada.

REFERÊNCIAS

ANSI/IEEE. Wireless lan medium access control (mac) and physical layer (phy) specifications. 1999.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. In: **Present and ulterior software engineering**. [S.l.]: Springer, 2017. p. 195–216.

FLANAGAN, D. **JavaScript - The Definitive Guide**. 5th ed.. ed. [S.l.]: O'Reilly, 2006.

FOWLER, M.; LEWIS, J. Microservices a definition of this new architectural term. **URL: http://martinfowler.com/articles/microservices.html**, 2014.

GUBBI, J. et al. Internet of things (iot): A vision, architectural elements, and future directions. **Future generation computer systems**, Elsevier, v. 29, n. 7, p. 1645–1660, 2013.

KARAGIANNIS, V. et al. A survey on application layer protocols for the internet of things. **Transaction on IoT and Cloud computing**, v. 3, n. 1, p. 11–17, 2015.

LÓSCIO, B. F.; OLIVEIRA, H. d.; PONTES, J. d. S. Nosql no desenvolvimento de aplicações web colaborativas. **VIII Simpósio Brasileiro de Sistemas Colaborativos**, v. 10, n. 1, p. 11, 2011.

MARJANI, M. et al. Big iot data analytics: architecture, opportunities, and open research challenges. **ieee access**, IEEE, v. 5, p. 5247–5261, 2017.

MEEHAN, J. et al. Data ingestion for the connected world. In: **CIDR**. [S.l.: s.n.], 2017.

MILOSLAVSKAYA, N.; TOLSTOY, A. Big data, fast data and data lake concepts. **Procedia Computer Science**, Elsevier, v. 88, p. 300–305, 2016.

MITCHELL, H. B. **Multi-sensor data fusion: an introduction**. [S.l.]: Springer Science & Business Media, 2007.

MOSQUITTO.ORG. **Eclipse Mosquitto**. 2019. Disponível em: https://mosquitto.org. Acesso em: 15 jun.2019.

NAQVI, S. N. Z.; YFANTIDOU, S.; ZIMÁNYI, E. Time series databases and influxdb. **Studienarbeit, Université Libre de Bruxelles**, 2017.

NEWMAN, S. **Building microservices: designing fine-grained systems**. [S.l.]: "O'Reilly Media, Inc.", 2015.

OLIVEIRA, L. et al. Sherlock: Capturing probe requests for automatic presence detection. In: IEEE. **2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design ((CSCWD))**. [S.l.], 2018. p. 848–853.

OLIVEIRA, L. et al. Mobile device detection through wifi probe request analysis. **IEEE Access**, IEEE, 2019.

OPENGROUP.ORG. **ArchiMate**. 2022. Disponível em: https://www.opengroup.org/archimate-forum/archimate-overview. Acesso em: 13 abr.2022.

RODRIGUEZ, A. Restful web services: The basics. **IBM developerWorks**, v. 33, p. 18, 2008.

STATISTA. **Number of Connected Devices Worldwide**. 2021. Disponível em: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/. Acesso em: 07 jan.2021.

STREAMSETS. **StreamSets Data Collector ™**. 2019. Disponível em: https://streamsets.com/products/sdc. Acesso em: 15 jun.2019.

SUNDMAEKER, H. et al. Vision and challenges for realising the internet of things. **Cluster of European Research Projects on the Internet of Things, European Commision**, v. 3, n. 3, p. 34–36, 2010.

THANGAVEL, D. et al. Performance evaluation of mqtt and coap via a common middleware. In: IEEE. **2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)**. [S.l.], 2014. p. 1–6.

APÊNDICES

APÊNDICE A – CÓDIGO EM PYTHON RESPONSÁVEL PELA GERAÇÃO DE SENSORES.

Código 3 – Geração de sensores

```
import random
import requests
import json

N = 1000
CENTER_LAT = -22.861592962437157
CENTER_LON = -43.22808397926099

def create_sensor(i):
    mac = "02:00:00:%02X:%02X:%02X" % (random.randint(0, 255),
                                       random.randint(0, 255),
                                       random.randint(0, 255))

    return {
        "mac": mac,
        "name": "Sensor %d" % i,
        "description": "Sensor #%d" % i,
        "latitude": "%s" % (CENTER_LAT +
                            (random.uniform(-1, 1) / 100)),
        "longitude": "%s" % (CENTER_LON +
                             (random.uniform(-1, 1) / 100))
    }

for i in range(1, N + 1):
    sensor = create_sensor(i)
    requests.post("http://localhost:8000/sensors",
                  data=json.dumps(sensor))
```

APÊNDICE B – CÓDIGO EM PYTHON RESPONSÁVEL PELA GERAÇÃO DE LEITURAS SINTÉTICAS DE TELEMETRIA.

Código 4 – Geração de leituras sintéticas de telemetria

```
from random import randint
import paho.mqtt.client as mqtt
import time
import json

sensors_file = open("sensors.txt", "r")

N = 12
T = 5
QOS = 1
SENSORS = sensors_file.read().splitlines()

def create_json_message(sensor):
    message = {}
    message['temp'] = randint(25,30)
    message['hum'] = randint(65,70)
    message['co2'] = randint(1100,1110)
    message['sensor'] = sensor
    message['createdAt'] = int(time.time() * 1000)
    return json.dumps(message)

def publish(client):
    for sensor in SENSORS[0:N]:
        jsonMessage = create_json_message(sensor)
        client.publish('telemetry', jsonMessage, QOS)

client = mqtt.Client()
client.connect("localhost", 1883, 60)
client.loop_start()

while True:
    publish(client)
    time.sleep(T)
```

APÊNDICE C – CÓDIGO EM PYTHON RESPONSÁVEL PELA GERAÇÃO DE LEITURAS SINTÉTICAS DE *PROBE REQUESTS*.

Código 5 – Geração de leituras sintéticas de *probe requests*

```
from random import randint
import paho.mqtt.client as mqtt
import json
import time

sensors_file = open("sensors.txt", "r")
T = 5
D_MAX = 2
P_MAX = 3
QOS = 1
SENSORS = sensors_file.read().splitlines()

def random_new_nearby():
    new_nearby_count = max(0, randint(-D_MAX,
                                      D_MAX))
    return generate_devices(new_nearby_count)

def generate_mac_address():
    return "02:00:00:%02X:%02X:%02X" % (randint(0, 255),
                                        randint(0, 255),
                                        randint(0, 255))


def generate_devices(n):
    devices = list()

    for _ in range(n):
        device = {}
        device['mac'] = generate_mac_address()
        device['presence'] = randint(1, P_MAX)
        devices.append(device)

    return devices
```

```
def create_json_message(srcMac, sensor):
    message = {}
    message['rssi'] = randint(-75,-20)
    message['sensor'] = sensor
    message['srcMac'] = srcMac
    message['createdAt'] = int(time.time() * 1000)
    return json.dumps(message)

def publish(client, sensors):
    for sensor in sensors:
        for device in sensors[sensor]:
            jsonMessage = create_json_message(device['mac'],
                                              sensor)
            client.publish('probes', jsonMessage, QOS)

def refresh_sensors(sensors):
    for sensor in sensors:
        devices_refreshed = list()

        for device in sensors[sensor]:
            device['presence'] -= randint(0, 1)

            if(device['presence'] > 0):
                devices_refreshed.append(device)

        sensors[sensor] = devices_refreshed




client = mqtt.Client()
client.connect("localhost", 1883, 60)
client.loop_start()

sensors = {}
for sensor in SENSORS:
    sensors[sensor] = random_new_nearby()
```

```
while True:
    publish(client, sensors)
    refresh_sensors(sensors)

    for sensor in sensors:
        sensors[sensor] += random_new_nearby()

    time.sleep(T)
```