



Universidade do Estado do Rio de Janeiro

Centro de Tecnologia e Ciências

Faculdade de Engenharia

Fernando de Oliveira Lima

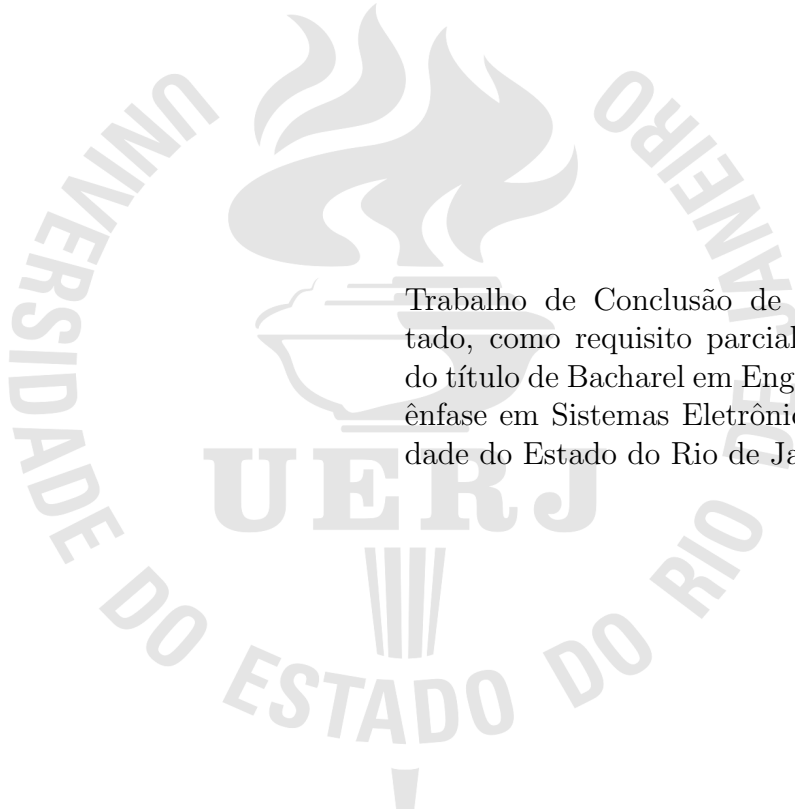
**Sistema Escalável para Aplicações de Internet das Coisas
utilizando MQTT**

Rio de Janeiro

2018

Fernando de Oliveira Lima

Sistema Escalável para Aplicações de Internet das Coisas utilizando MQTT



Trabalho de Conclusão de Curso apresentado, como requisito parcial para obtenção do título de Bacharel em Engenharia Elétrica ênfase em Sistemas Eletrônico, da Universidade do Estado do Rio de Janeiro.

Orientadores: Prof. Michel Tcheou, DSc
Prof. Lisandro Lovisolo, DSc

Rio de Janeiro

2018

CATALOGAÇÃO NA FONTE

S237

UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

de Oliveira Lima, Fernando

Sistema Escalável para Aplicações de Internet das Coisas utilizando MQTT / Fernando de Oliveira Lima– 2018.

105 f.

Orientadores: Michel Tcheou, Lisandro Lovisolo.

Trabalho de Conclusão de Curso – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

Texto a ser informado pela biblioteca

CDU 621:528.8

Autorizo, apenas para fins acadêmicos e científicos, a reprodução total ou parcial desta dissertação, desde que citada a fonte.

Assinatura

Data

Fernando de Oliveira Lima

Sistema Escalável para Aplicações de Internet das Coisas utilizando MQTT

Trabalho de Conclusão de Curso apresentado, como requisito parcial para obtenção do título de Bacharel em Engenharia Elétrica ênfase em Sistemas Eletrônico, da Universidade do Estado do Rio de Janeiro.

Aprovado em: 28 de Agosto 2018

Banca Examinadora:

Prof. Dr. Michel Tcheou (Orientador)
Departamento de Eletrônica e Telecomunicações da UERJ

Prof. Dr. Lisandro Lovisolo (Orientador)
Departamento de Eletrônica e Telecomunicações da UERJ

Prof. Dr. Nome do Professor 3
Universidade Federal do Rio de Janeiro - UFRJ - COPPE

Prof. Dr. Nome do Professor 4
Instituto de Geociências da UFF

Rio de Janeiro

2018

DEDICATÓRIA

Aqui entra sua dedicatória.

AGRADECIMENTO

Aqui entra seu agradecimento.

É importante sempre lembrar do agradecimento à instituição que financiou sua bolsa, se for o caso...

Agradeço à FAPERJ pela bolsa de Mestrado concedida.

RESUMO

LIMA, Fernando *Sistema Escalável para Aplicações de Internet das Coisas utilizando MQTT*. 105 f. Dissertação (Engenharia Elétrica - Sistemas Eletrônicos) - Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, 2018.

No meio da revolução dos dados, cresce o interesse em comunicação entre máquinas e compartilhamento de dados telemétricos sobre dispositivos, seja numa fábrica ou em residências. Esta dissertação trata sobre um sistema para aplicações de internet das coisas(IoT) utilizando MQTT a *lingua franca* para publicação de dados telemétricos via TCP/IP, com persistência de dados em banco MongoDB. Englobando todos os setores de aquisição dos dados a camada de aplicação em consoles, com o objetivo de facilitar a implementação de aplicações eficientes em cada cenário.

Palavras-chave: iot, mqtt, indústria.

ABSTRACT

In the verge of the data revolution, a growing interest in communication between machines and the sharing of telemetric data on devices rises, whether in a factory or in a residence. This dissertation deals with a system for Internet applications of things (IoT) using MQTT the *lingua franca* for publishing telemetric data via TCP / IP, with data persistion using MongoDB. Encompassing all sectors of data acquisition to the application layer in consoles, facilitating implementations of applications in each scenario.

Keywords: iot, mqtt, industry .

LISTA DE FIGURAS

Figura 1	Um exemplo de aplicação IoT que percorre os problemas a solucionar.....	14
Figura 2	A aplicação dividida em blocos exercendo um papel em uma aplicação IoT	15
Figura 3	As três camadas do IoT, dos sensores ao mundo real.....	16
Figura 4	Diferenças entre OSI e TCP/IP em suas camadas	18
Figura 5	Interface de comunicação. A interface tem seu próprio protocolo que direciona e se comunica a um ou mais protocolos de aplicação	22
Figura 6	O padrão Publish/Subscribe. Retirado de [11]	24
Figura 7	Fluxo de conexão do HTTP	25
Figura 8	Fluxo da conexão do MQTT	26
Figura 9	Exemplo de gerenciamento de um broker.....	27
Figura 10	O conceito de Data Stream para a abstração do transporte de dados.....	28
Figura 11	Um Data Stream é criado a partir do tópico <i>/003/stream:periodic</i>	29
Figura 12	Comunicação entre Publishers e Subscribers por Data Stream	30
Figura 13	A arquitetura do ESP32, retirado de [18].....	31
Figura 14	A arquitetura simplificada de dispositivos com Sistema Operacional.....	32
Figura 15	Diagrama simplificado de uma rotina padrão seguida pela implementação em Microcontroladores	33
Figura 16	Diagrama simplificado de uma rotina assíncrona padrão em Consoles	34
Figura 17	A arquitetura de um banco de dados	37
Figura 18	Comparação de Escrita e Leitura entre LSM e B+, retirado de [28]	38
Figura 19	O formato de documento no MongoDB.....	39
Figura 20	Adição de parâmetro de timestamp em milisegundos ao documento	40
Figura 21	Diagrama de fluxos do Publisher e do Subscriber.....	42
Figura 22	Visualização dos dados armazenados em documento em uma coleção do MongoDB	43
Figura 23	Comportamento da temperatura em três momentos.....	43
Figura 24	Comparação das temperaturas com uma CPU core i7 e ESP32	44
Figura 25	Comparação das temperaturas em múltiplos processadores	45
Figura 26	Planta baixa de residência de dois quartos, retirado de [34]	47

LISTA DE TABELAS

Tabela 1	As camadas e e suas funções	19
Tabela 2	Comparativo MQTT X HTTP	26
Tabela 3	Orçamento de um sistema simples para automação da residência da Figura 26.....	47
Tabela 4	Infraestrutura para sistema de controle de iluminação de postos de Com- bustíveis.....	48

Lista de Códigos

5.1	Data Stream Header	56
5.2	Data Stream Source	58
5.3	MQTT Publisher Header em C++	59
5.4	MQTT Publisher Source em C++	61
5.5	Data Stream em javascript	66
5.6	Continuous Stream	68
5.7	Periodic Stream em Javascript	68
5.8	MQTT Publisher Source em Javascript	70
5.9	Index das implementações	74
5.10	MQTT Subscriber Source em Javascript	74
5.11	Index das implementações	79
5.12	Data Client para MongoDB	79
5.13	Teste do publisher	83
5.14	Teste do subscriber	84
5.15	Teste do publisher no ESP32	86

LISTA DE SIGLAS

IoT	Internet das Coisas
MQTT	Message Queuing Telemetry Transport
API	Application Programming Interface
NB	Narrow Band Networks
A/D	Analógico-Digital
DB	Banco de Dados
IaaS	Infrastructure as a Service
MCU	Micro-Controller Unit

SUMÁRIO

	INTRODUÇÃO	13
1	INTRODUÇÃO AO PROJETO	13
1.1	Internet das Coisas.....	13
1.2	Visão geral de uma aplicação IoT.....	14
1.3	As Camadas da IoT.....	15
1.3.1	Aquisição	16
1.3.2	Transmissão	17
1.3.3	Aplicação	17
1.4	Camadas de Rede	18
2	A INTERFACE E SUA LIGAÇÃO COM IOT	20
2.1	Tecnologias em IoT	20
2.2	A Interface.....	21
3	O PROJETO	23
3.1	Camada de Abstração	23
3.2	Publishers e Subscribers	23
3.3	A implementação.....	25
3.3.1	MQTT X HTTP	25
3.3.2	MQTT	26
3.3.2.1	Broker.....	27
3.3.2.2	Tipos de MQTT	27
3.3.3	Data Streams.....	28
3.3.4	Plataformas.....	30
3.3.4.1	Embarcados	30
3.3.4.2	Consoles	31
3.4	Arquiteturas e Assíncronismo	32
3.4.1	Embarcados Síncronos	33
3.4.2	Consoles Assíncronos	34
3.5	Segurança de aplicações	35

3.6	Persistência de dados	36
3.7	Bancos para Aplicações IoT	37
3.8	Indexação de dados e Timestamp	39
4	CASOS DE USO	41
4.1	Medição de temperaturas de CPU	41
4.2	Aplicação: Automação Residencial	46
4.3	Aplicação: Controle de Iluminação de Postos de combustível	47
	CONCLUSÃO	49
	REFERÊNCIAS	51
5	APÊNDICE	54
5.1	Guias de instalação	54
5.1.1	Configurando Broker	54
5.1.2	Publishers em C++	54
5.1.3	Publishers em Javascript	55
5.1.4	Subscribers em Javascript	55
5.2	Códigos Fonte	56
5.2.1	Publishers em C++	56
5.2.2	Publishers em Javascript	66
5.2.3	Subscribers em Javascript	74
5.2.4	Códigos fonte das aplicações em consoles	83
5.2.5	Códigos fonte das aplicações em plataformas embarcadas	86

1 INTRODUÇÃO AO PROJETO

O cenário atual do desenvolvimento tecnológico encontra-se no meio de uma quarta revolução industrial. Nunca se produziu tantos dados e se utilizou redes como a própria internet para propaga-los. É de se esperar que tanto a academia e diferentes mercados demandem inovações para o compartilhamento desses dados em tempo real ou próximo disso. Aquecendo o mercado que engloba transporte, análise e inteligência de dados.

Este projeto propõe uma interface para comunicação entre as diferentes tecnologias e camadas de rede, de forma que o desenvolvedor só se preocupe em implementar e configurar uma interface para mapear a melhor opção de ferramentas para a aplicação. O projeto lida com protocolos baseados na pilha TCP/IP, uma unanimidade em redes conectadas a internet. Podendo se estender para outros protocolos de aplicações de escopo fechado. O foco está no protocolo de aplicação MQTT (*Message Queuing Telemetry Transport*) [1], um protocolo que opera sobre o TCP/IP, leve e extremamente utilizado para compartilhamento dados telemétricos, de estado e de pequenas mensagens. Oferecendo uma API para aquisição, transmissão, recepção e armazenamento de dados telemétricos..

A Internet das Coisas é a rede que permite a conexão e compartilhamento de dados de dispositivos físicos . Ela é derivada de métodos de comunicação entre máquinas e telemetria. Pode ser dissecada em três camadas de aquisição, comunicação e aplicação podendo ser implementada utilizando diversos protocolos de comunicação, dependendo da tecnologia disponível. É importante que sistemas IoT sejam projetados de forma a atender a aplicação eficientemente, porém tal tarefa não é fácil nem simples. Este projeto oferece uma interface que permite facilitar tal tarefa.

1.1 Internet das Coisas

”A Internet das Coisas tem o potencial de mudar o mundo. Assim como a Internet fez. Talvez até mais” [2]. Uma tradução livre de Rampim [3] da frase de Kevin Ashton, cofundador do Auto-ID Center, em 1999. Apesar de ser um nome feito somente para chamar atenção, foi a primeira citação da expressão Internet das Coisas, e de lá vingou.

No contexto da Indústria 4.0, encontra-se a internet das coisas ou IoT, responsável por estruturar as aplicações de aquisição, transmissão e armazenamento de dados a serem analisados. Não é uma surpresa que a Internet das Coisas envolva áreas como eletrônica,

computação e telecomunicações em um pacote só. De fato as camadas de IoT são mundos diferentes interligados a um propósito: transmitir dados sobre um dispositivo e/ou para um dispositivo em tempo real. Segundo a Cisco IBSG, Cisco Internet Business Solutions Group [4], há mais objetos conectados que pessoas no mundo, fazendo com que o ano de 2009 seja considerado o ano de nascimento da IoT.

Pode-se definir IoT como a estrutura que comunica dispositivos em rede, permitindo a transmissão de dados sobre eles em tempo real. Essa estrutura permite a troca de informações sobre um dispositivo, qual seu estado, seu desempenho, suas condições físicas e do ambiente ao seu redor. Mas, para que este ciclo esteja completo são necessárias camadas que desempenham tarefas específicas, para que o dado chegue a quem ou a o que o está esperando.

1.2 Visão geral de uma aplicação IoT

Na Figura 1 ilustrada, temos uma rede de N sensores que enviam dados telemétricos e M atuadores que recebem ordens para executar uma função, todos estão em rede e podem receber e enviar informação em tempo real. O servidor, que pode ser um Broker como será descrito adiante neste trabalho, encaminha os dados (ou mensagens) para o banco de dados. O Banco é utilizado para análise dos dados, o controlador por sua vez envia as mensagens de decisões baseada na análise de dados a serem transmitidas para os atuadores.

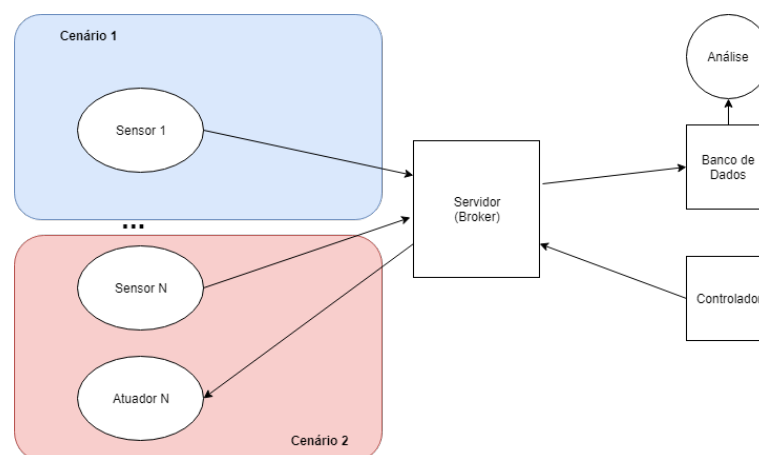


Figura 1 Um exemplo de aplicação IoT que percorre os problemas a solucionar

O sensor 1 está imerso em um ambiente com suas próprias características físicas e de rede, isso ocorre com todos, isto é cada sensor está imerso num cenário próprio,

variando de redes com poucos sensores a redes com grande fluxo de dados, sujeito a congestionamento. Assim, seria de grande ajuda que o sistema se ajustasse aos diferentes cenários.

Este trabalho visa implementar um sistema que utiliza o protocolo de aplicação MQTT, para transmissão de dados em tempo real entre dispositivos). O sistema contempla também persistência de dados utilizando banco de dados MongoDB, com o diferencial de se adaptar a cenários através de canais de dados chamados Data Streams, oferecendo diferentes configurações que podem otimizar o envio de dados em cada cenário.

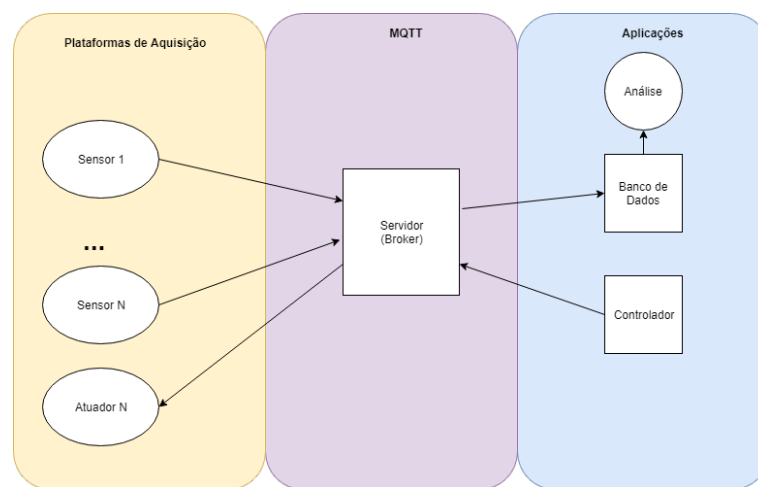


Figura 2 A aplicação dividida em blocos exercendo um papel em uma aplicação IoT

Cada bloco é responsável por uma tarefa no sistema IoT, da aquisição de dados a persistência destes, conforme ilustrado na Figura 2. Para entender melhor cada tarefa, será descrito o projeto e as implementações em cada bloco no sistema.

1.3 As Camadas da IoT

Uma rede IoT pode ser dividida em camadas que exercem funções específicas no transporte de dados, de uma forma semelhante a redes de computadores, a camada acima não precisa saber como a inferior funciona, formando uma estrutura de pilha como na figura Figura 3.

A primeira camada é a de aquisição de dados, que lida com o mundo físico e captura estes dados através de sensores e conversores A/D, também realiza o processamento para entregar em um formato adequado para transmissão e inteligível do outro lado, dependendo da aplicação. A segunda camada é a camada de transmissão, na qual estão, efetivamente,

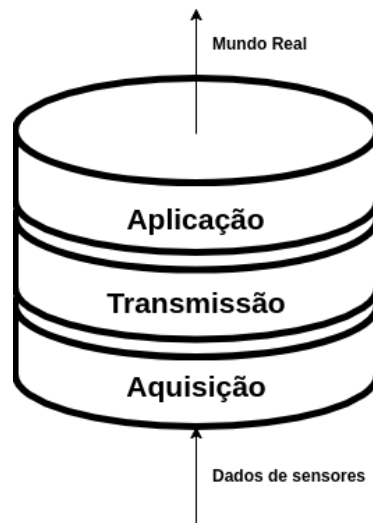


Figura 3 As três camadas do IoT, dos sensores ao mundo real

as camadas de rede embutidas. Como o nome já denuncia, ela lida com os aspectos de rede e comunicação para que o dados cheguem seus destinos. E por último temos a camada de aplicação, a mais abrangente e que envolve maior poder computacional. Ela recebe os dados e lida com os processos de aplicação destes dados, seja análise, visualização, armazenamento ou a estruturação dos mesmos.

1.3.1 Aquisição

A etapa de aquisição está inserida diretamente no contexto de dados físicos, geralmente são hardwares menos complexos, focados em processamento de dados e entrada e saída com conversão analógico-digital. Se comunicam com sensores ou centrais de controle lógico. São responsáveis por:

- Receber dados de sensores;
- Conversão A/D;
- Processamento de valores;
- Envio de dados em tempo real;

Para atender essas tarefas, não é necessário grande poder de processamento, microcontroladores ou microprocessadores são capazes de atender aos requisitos necessários para uma grande gama de variáveis físicas, se acompanhados de módulos de rede e portas I/O, assim como a implementação do software. Serão vistos dois exemplos desses

dispositivos, que utilizam tanto MCU e outro um Consoles com Sistemas Operacionais leves.

1.3.2 Transmissão

Esta camada é o coração do IoT. A forma de transmissão define quais dispositivos eletrônicos e qual sua especificação técnica necessária para os quesitos de transmissão. Também define como os softwares da camada de aplicação e aquisição devem ser implementados baseado na estrutura da pilha de rede que será usada para transmitir.

Na próxima seção, veremos sobre a camada de rede e suas diversas formas de implementação. É importante que esta camada seja definida da melhor forma a atender sua aplicação, atendendo aspectos:

- quantidade de dados transmitido;
- número de acessos;
- distância entre dispositivos;
- segurança;

1.3.3 Aplicação

A camada de aplicação encabeça a pilha do IoT. É ela que de fato trata os dados e realiza as aplicações deste. Ela disponibiliza os dados para o mundo real, podendo exercer múltiplas funções simultâneas incluindo:

- Armazenamento e Análise;
- Visualização;
- Inteligência e aprendizado;
- Serviços e servidores;
- Gerenciamento e configuração;

Nesta camada estão presentes os endpoints apontados pela camada de aquisição, o destino dos dados. Bem assim como os servidores que gerenciam os clientes (geralmente

implementados na camada de aquisição) e serviços e configurações oferecidos pelo sistema em si.

1.4 Camadas de Rede

Como visto anteriormente, a camada de transmissão basicamente define a infraestrutura do sistema. Ela é construído com as camadas de rede como base. Portanto definir as camadas de rede e seus protocolos é definir a camada de transmissão em si.

Redes de computadores são complexas com diferentes aspectos a se preocupar. Dividir em camadas permite modularizar a implementação da rede, de modo que cada camada tenha uma tarefa na estrutura de comunicação dos aspectos físicos ao software. Como a camada de cima não precisa saber sobre os detalhes e especificações da camada de baixo, as mudanças de uma parte do sistema é transparente para o resto do sistema. Existem diversas formas de implementação de camadas, mas todas se baseiam em um modelo de referência, o modelo OSI [5].

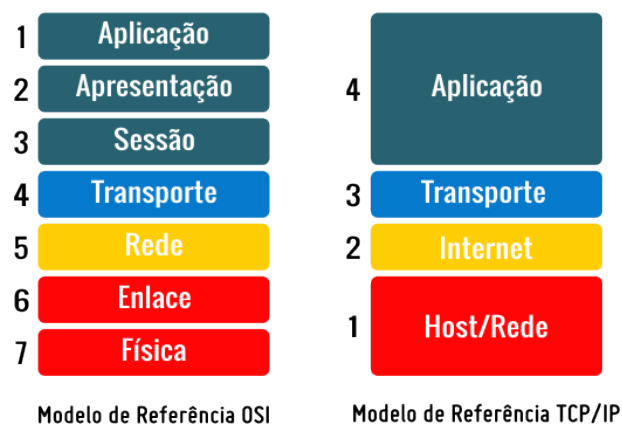


Figura 4 Diferenças entre OSI e TCP/IP em suas camadas

Baseado no modelo OSI. Temos o modelo TCP/IP [6], visualizado em Figura 4 e utilizado na internet e base para protocolos de aplicação muito utilizados como HTTP, WebSocket e MQTT. Em ambos cada camada exerce uma tarefa com seu respectivo protocolo, como resumido na tabela Tabela 1.

O foco desta literatura está na camada de aplicação e suas funcionalidades. Apesar de diferentes tecnologias utilizarem diferentes camadas abaixo, serão específicas características de protocolos construídos em cima do TCP/IP, por seu uso na Internet e redes industriais, e que sejam eficientes em trocas de mensagens em tempo real.

Tabela 1 As camadas e e suas funções

CAMADA	DETALHES
7 - Aplicação	Define instruções específicas da aplicação
6 - Apresentação	Formatação dos dados, conversão dos dados
5 - Sessão	Negociação e conexão com outros nós, analogia
4 - Transporte	Oferece métodos para a entrega de end-to-end
3 - Rede	Roteamento de pacotes em uma ou várias redes
2 - Enlace	Detecção de erros;
1 - Física	Aspectos físicos da transmissão

2 A INTERFACE E SUA LIGAÇÃO COM IOT

No capítulo 1, foi visto as bases para se implementar um projeto de IoT. A área começou a receber fortes investimentos e atenção por volta de 2009 e desde então foram feitas consideráveis implementações utilizando tecnologias e protocolos diferentes. Neste capítulo serão apresentados algumas dessas variações, para fins de comparação e respaldo para importância e objetivo deste projeto.

2.1 Tecnologias em IoT

Estas são algumas tecnologias que satisfazem as condições apresentadas para um sistema IoT, nem todas utilizam o protocolo TCP/IP, mas todas são capazes de fazer seus dispositivos comunicarem-se em tempo real levando em consideração seus alcances e escalabilidade.

As primeiras aplicações de IoT foram em laboratórios de aplicações de RFID [3], junto com códigos bidimensionais, para aplicações de identificação de objetos. Uma das soluções mais populares e de baixo custo de IoT utilizando Rádio frequência.

Redes que utilizam bandas restritas visando baixo consumo e distância de transmissão são a nova fronteira, as mensagens de IoT são geralmente curtas, dados telemétricos, status etc, logo estes protocolos mostram-se úteis para este tipo de aplicação. Já se encontram implementadas algumas redes como SigFox [7] e LoRa [8].

As novas gerações de Bluetooth consomem muito menos energia, o que tornaram a tecnologia viável para aplicações IoT. Geralmente, módulos Bluetooth são utilizados como beacons [9]. Pontos espalhados por uma região, no qual podem se comunicar com os módulos de dispositivos mobile ao se aproximar, oferecendo links para conteúdo e exclusividades.

As tecnologias mais comuns de se encontrar em aplicações IoT, os protocolos construídos com base no TCP/IP são vastamente utilizados e possuem uma rede mundialmente distribuída, o que facilita o uso. Pode-se implementar uma gama de protocolos de aplicações, alguns mais eficientes que outros.

O protocolo mais simples seria o HTTP, altamente usado na internet, porém não é eficiente no consumo de energia por abrir uma conexão a cada envio de dados. Para minimizar estas desvantagens, foi desenvolvido o CoAP [10] protocolo nos mesmos moldes

do HTTP com o modelo REST, porém mais simples, mais leve, com baixo overhead e utilizado em redes locais.

Mas os mais utilizados em aplicações são sem dúvidas os protocolos que mantêm conexão aberta, em especial Websocket e MQTT, sendo o primeiro mais utilizado para chats e mensagens e o segundo domina o mundo do M2M e Telemetria.

2.2 A Interface

Inicialmente, os conceitos e ideias do projeto eram voltados a desenvolver uma interface no qual um desenvolvedor poderia implementar um sistema IoT de ponta a ponta utilizando APIs que direcionariam para um desses protocolos da seção 2.1, porém as diferenças entre os protocolos e as camadas de base, fazem com que esta solução esteja mais distante. Então o foco voltou-se para tecnologias que tenham base na pilha TCP/IP, por sua vasta implementação nas redes industriais e residências e na Internet.

Neste projeto iremos ver a implementação desta interface para o protocolo MQTT, cuja escolha será justificada adiante. Serão descritas as interfaces para as três camadas, que são de baixo custo, open-source e altamente escaláveis para construir outras aplicações com esta como base.

Para abstrair as camadas e aplicação, a interface deve estar dentro dos requisitos do IoT, bem assim como apresentar uma estrutura que se traduza aos protocolos baseados TCP/IP, para isso, algumas características fundamentais podem ser destacadas como base para um sistema IoT descritos.

- Full-Duplex. Capaz de receber e enviar mensagens ao mesmo tempo;
- Multicast. Capaz de enviar mensagens um ou mais dispositivos simultâneos;
- Envio de mensagens em tempo real;

Essas características estão dentro do escopo apresentado no capítulo ??, a estratégia foi direcionada para protocolos que atendessem esse pré-requisito, assim construindo uma camada de abstração da interface por cima dos protocolos que contemplam essas características como mostra a figure Figura 5, o que será mais detalhado no próximo capítulo.

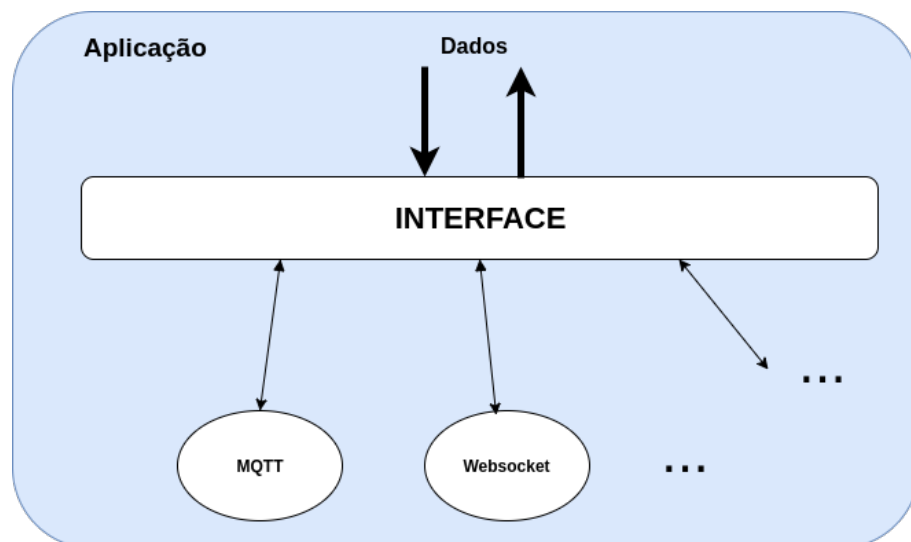


Figura 5 Interface de comunicação. A interface tem seu próprio protocolo que direciona e se comunica a um ou mais protocolos de aplicação

3 O PROJETO

Os dois últimos capítulos descreveram o conceito de Internet das Coisas e as especificações que o projeto deve contemplar, visando sempre ajudar na construção de sistemas IoT que melhor se encaixem na aplicação. Neste capítulo serão descritos as implementações do projeto, apresentado os motivos das escolhas de tecnologias e protocolos especificados. E terminando sobre persistência de dados em aplicações IoT e por quê a escolha de implementação bancos de dados é importante para a aplicação.

3.1 Camada de Abstração

Devido a interação entre dispositivos de aquisição de dados e aplicação e armazenamento de dados, foi necessário uma implementação de um protocolo de comunicação único entre os dispositivos e implementação em cada um destes em suas diferentes linguagens de programação.

O protocolo consiste em uma abstração de um canal de envio de dados chamado Data Stream mostrado em Figura 10, no qual passam dados após realizar um processamento dos dados em uma determinada velocidade podendo conter um limite de pacote de dados. Nas pontas desse canal estão os Publishers e Subscribers, que serão descritos adiante. Este conceito é uma forma de abstrair, unificar e simplificar a forma de transporte de dados, de uma modo que a interface possa ter o controle sobre os aspectos de transmissão. Cada protocolo na camada de aplicação, implementa este conceito de uma certa forma, porém o desenvolvedor não precisará se preocupar com estes detalhes.

3.2 Publishers e Subscribers

Para enviar e receber dados de uma forma a atender os requisitos da seção 2.2, foi utilizado um padrão de comunicação recorrente em aplicações contemporâneas, o padrão Publish/Subscriber [11].

O padrão Publish/Subscribe permite que as mensagens sejam transmitidas assíncronas e para vários dispositivos simultaneamente. Para transmitir uma mensagem, um client pode simplesmente enviar uma mensagem para o tópico que os envia imediatamente para todos os subscribers. Todos os componentes que se inscreverem no tópico receberão todas

as mensagens transmitidas, a menos que uma política de filtragem de mensagens seja definida pelo assinante.

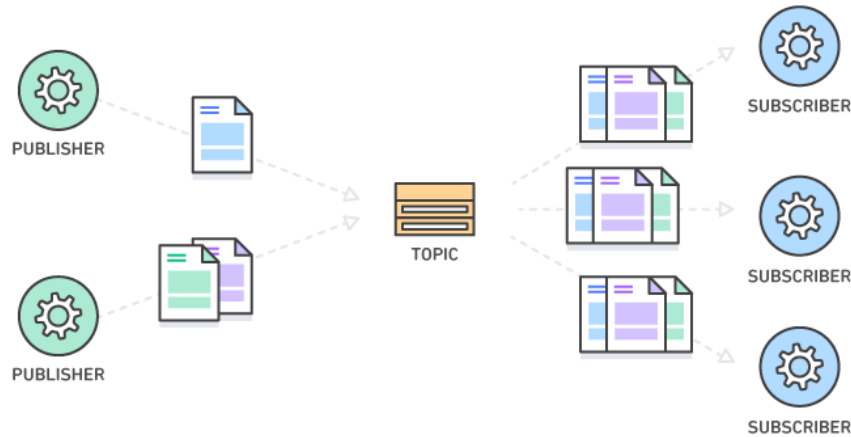


Figura 6 O padrão Publish/Subscribe. Retirado de [11]

Qualquer mensagem publicada em um tópico é imediatamente recebida por todos os subscribers do tópico. As mensagens de podem ser usadas para arquiteturas orientadas a eventos ou para desacoplar aplicativos, aumentando o desempenho, a confiabilidade e a escalabilidade. Com isso, foram criados duas funções possíveis para cada dispositivo dentro deste padrão, os Publishers e os Subscribers, sua comunicação é descrita em Figura 12.

Publishers são dispositivos que criam Data Stream e enviam dados por estes, regulam o processamento dos dados estipulam limites de tamanho de cada pacote de dado e determinam o intervalo de envio de pacotes. O protocolo permite que estes enviem os dados e também permite que outros dispositivos possam passar configurações remotamente para modificar os parâmetros de cada Data Stream, como o intervalo de envio ou outra configuração criada pelo tipo de Data Stream implementado.

Porém este padrão define as configurações gerais do sistema, não contempla as mudanças de cenário possíveis. É necessária a adição de configurações dinâmicas que se adaptem as condições impostas pelos cenários, uma interface que varia com as condições de cada par Publisher/Subscriber formado. Para isso foi criado o conceito Data Stream.

3.3 A implementação

Existem protocolos de aplicação que podem ser facilmente mapeados por esse tipo de interface. Alguns já são construídos no modelo Publish/Subscribe, outros em modelos parecidos. Para critérios de comparação, apresentam-se dois protocolos de aplicação, MQTT e WebSockets [12]. O protocolo MQTT, no qual será implementado nesse projeto, foi moldado no padrão citado, enquanto o WebSockets é orientado a eventos. Ambos enviam mensagens em tempo real através de um servidor que controla o fluxo de mensagens da aplicação.

3.3.1 MQTT X HTTP

Na questão de desempenho, estes protocolos se mostram mais eficientes dentre os construídos sobre TCP/IP, comparado, por exemplo, com o HTTP, protocolo de rede mais recorrente em redes locais e na internet. Como mostrado em [13] e [14], HTTP, por sua natureza de abrir e fechar conexão a cada requisição de dados e seu cabeçalho ilustrado na Figura 7, requer mais banda e consome mais energia que protocolos leves e de conexão persistente como o MQTT. O que faz sua escolha remota para a aplicação deste projeto.

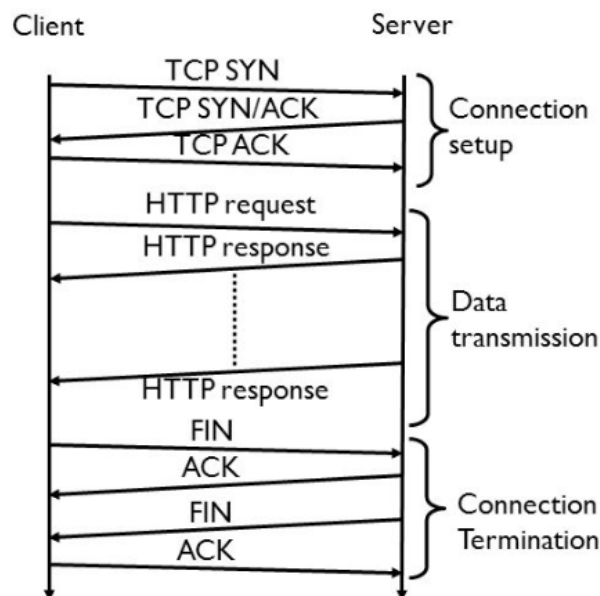


Figura 7 Fluxo de conexão do HTTP

Já o MQTT é um protocolo de cabeçalhos menores, conexão persistente e menos passos para o envio de mensagem, como visto na Figura 8, por ser um protocolo feito com

o objetivo de reduzir a latência, levando vantagem para a transmissão contínua de dados, em relação ao HTTP.

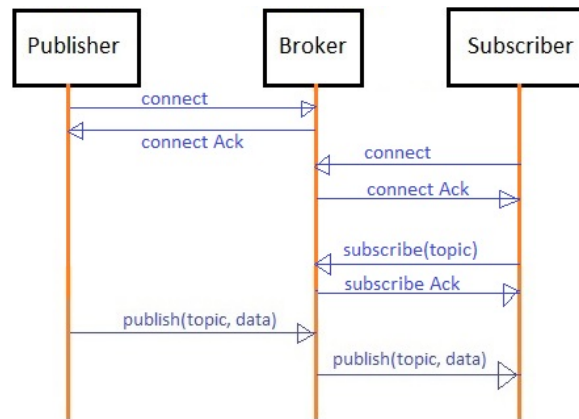


Figura 8 Fluxo da conexão do MQTT

Abaixo na Tabela 2 encontra-se o resumo comparativo entre os protocolos MQTT e HTTP:

Tabela 2 Comparativo MQTT X HTTP

Característica-Protocolo	MQTT	HTTP
Arquitetura	Publish/Subscribe	Request/Response
Complexidade	simples	complexa
Segurança	TSL/SSL	TSL/SSL
Camada de Transporte	TCP	TCP ou UDP
Tamanho/Formato de Mensagens	curtas, binário com cabeçalho de 2Bytes	Grande, Formato ASCII
Porta Padrão	1883	80 ou 8080
Distribuição de dados	1 um para 0/1/N	um para um

3.3.2 MQTT

O protocolo MQTT foi utilizado escolhido por ser leve e ideal para aplicações em tempo real com vários dispositivos simultaneamente. É um protocolo no padrão Publish/Subscribe ideal para definir a função de cada dispositivo seja enviando dados (Publish) ou recebendo estes (Subscribe).

Para gerenciar os clients (responsáveis pela implementação da comunicação MQTT) em cada dispositivo é necessário um servidor chamado Broker. Este foi implementado com o Mosquitto [15], um broker open source e leve capaz de ser instalado localmente e no servidor do laboratório para testes remotos.

3.3.2.1 Broker

O Broker é o servidor do padrão Publish/Subscribe, ele efetivamente executa as ordens de publicação (publish) feita por algum cliente para os tópicos que outros clientes estão inscritos (subscribed), possui todas as listas de tópicos, é orientado a conexão e não persiste informações dos clientes, ou seja, em caso de queda de conexão, estes devem se inscrever novamente nos tópicos.

Nota: A arquitetura Broker não é exclusividade do MQTT, outros protocolos utilizam esse tipo de implementação em servidores.

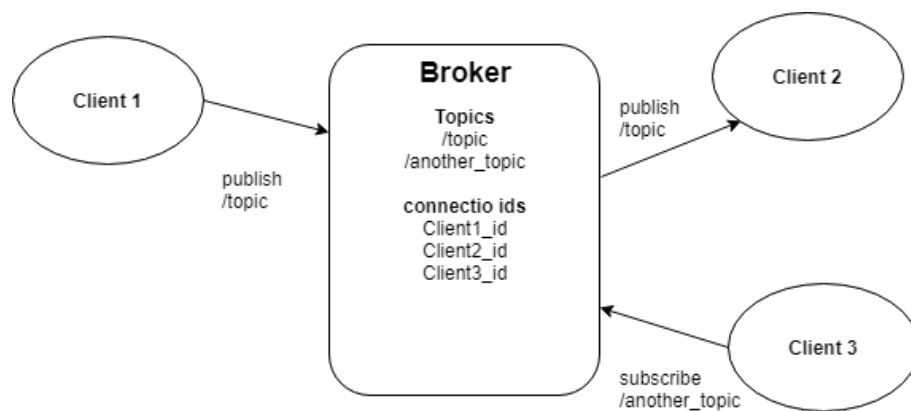


Figura 9 Exemplo de gerenciamento de um broker

Na figura Figura 9, o Broker armazena os tópicos e os ids de conexão dos clientes, 2 estava inscrito para ouvir as mensagens do tópico *topic*, enquanto 3 enviava uma ordem de inscrição em *another_topic*, 1 envia ordem de publicação para *topic*.

3.3.2.2 Tipos de MQTT

Com a evolução e o uso do protocolo, foram necessárias atualizações que contemplam funcionalidades que atendem requisitos essenciais para aplicações da indústria. Como atender dispositivos que não usam a pilha TCP/IP e medidas de segurança (que serão melhor debatidas á frente).

Existe uma gama de dispositivos que utilizam protocolos específicos, geralmente leves, para redes locais e leves para o transporte de dado, a exemplo do ZigBee [16]. Para isso foi criada uma versão do MQTT para atender estes tipos de protocolos, substituindo a base TCP/IP por outros protocolos destas camadas, mantendo a camada de aplicação e o padrão Publish/Subscribe.

Para resolver questões de segurança, foi criada uma variação do MQTT que adiciona camadas deste quesito ao protocolo de aplicação. Assim como o HTTPS o protocolo MQTTS é construído em cima do protocolo SSL/TLS (explicado em ??), camada de segurança que também usa como base TCP/IP. Esta camada envolve o processo de encriptação dos cabeçalhos da aplicação e autenticação por passagem de certificados.

3.3.3 Data Streams

Um Data Stream é uma interface que permite adicionar configurações de como o dado será enviado pelo Publisher, permitindo que este lide com os problemas causados pelo cenário, como congestionamentos, limitando o tamanho de mensagens, conversões de dados ou problemas de processamento. As configurações podem ser enviadas pelo Subscriber, permitindo um dinamismo caso aconteça mudanças de cenários em algum Publisher. A configuração pode lidar com qualquer aspecto do envio de dados, como o tamanho das mensagens enviadas, ou a taxa de envio ou no próprio formato de mensagem enviado.

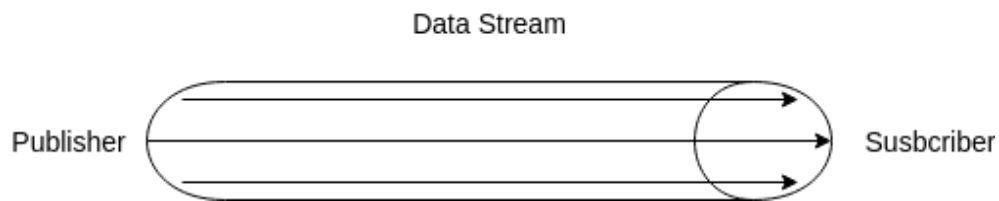


Figura 10 O conceito de Data Stream para a abstração do transporte de dados

Para criar um Data Stream, basta um Subscriber estar ouvindo um tópico no formato abaixo. E um Publisher publicar neste tópico. O ID corresponde a uma identificação única que pode ser definida pelo desenvolvedor. O *stream_nome* corresponde ao tipo de Data Stream utilizado.

$$/{data_stream_id}/stream : \{stream_nome\}$$

Quando um Data Stream é criado, um conjunto de configurações determina como o dado será enviado, essas configurações estão contidas nos publishers dependendo da lista de Data Streams que este possui. Os Subscribers podem alterar estas configurações, baseada nas necessidades da aplicação, como problemas de processamento ou congestionamento etc.

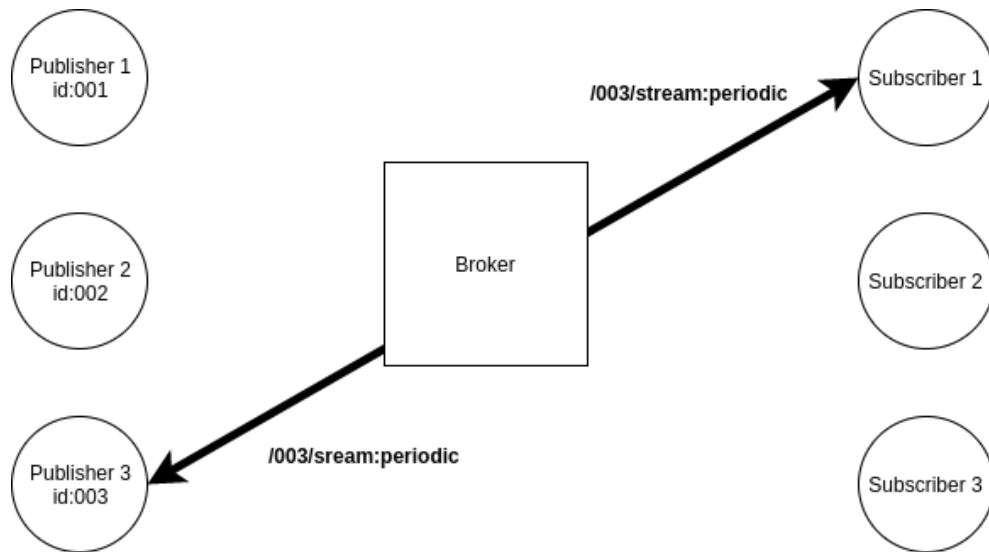


Figura 11 Um Data Stream é criado a partir do tópicos */003/stream:periodic*

Para um Subscriber enviar as alterações nas configurações basta publicar no tópicos abaixo quase idêntico ao anterior. As configurações são feitas por uma string JSON [17], um conjunto de chaves-valor universalmente interpretada por várias linguagens de programação como forma de transporte de objetos de uma classe.

$$/{data_stream_id}/configure/stream : \{stream_nome\}$$

Existem dois tipos de Data Stream já implementados em qualquer publisher. Porém o desenvolvedor pode implementar seus próprios Data Stream dependendo da linguagem de programação utilizada:

- Contínuo: Data Stream padrão sem configurações definidas que publica continuamente dados;
- Periódico: Publica dados esperando um período T antes de publicar, este período pode ser alterado;
- Customizáveis: Criados pelo desenvolvedor, com suas próprias configurações.

Subscribers estão na outra ponta recebendo os dados, são capazes de enviar as configurações do Data Stream para os Publishers a chegada destes dados como um driver para a aplicação. Essas funcionalidades foram implementadas Orientadas a Objeto e são escaláveis para aplicações mais complexas que serão implementadas para o uso dos sistemas em aplicações de sensoriamento e visualização dos dados.

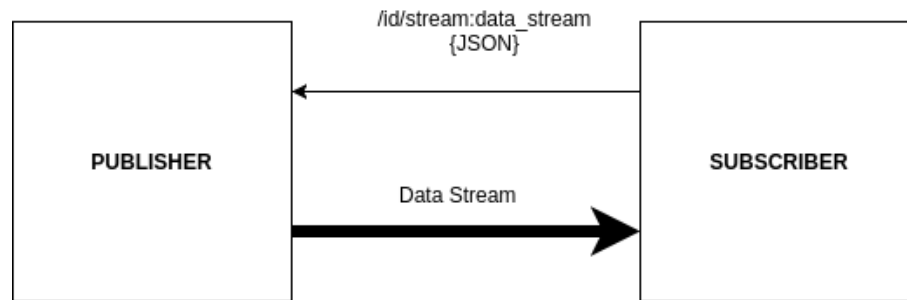


Figura 12 Comunicação entre Publishers e Subscribers por Data Stream

3.3.4 Plataformas

Para que o sistema esteja completo, é necessário estender a interface a camada de aquisição e aplicação da pilha do IoT, para isso, foi necessário criar SDKs (Software Development Kits) para plataformas que suportem o protocolo TCP/IP. E que possam ser utilizadas em sistemas IoT. Para a camada de aplicação, foram focados em plataformas embarcadas com acesso a rede e para a aplicação, plataformas com sistemas operacionais, podendo suportar aplicações mais complexas.

A camada de aquisição apresenta implementação dos Publishers, pois são que enviam os dados, as plataformas possuem unidades de processamento e módulos de rede o que as torna ideais para publishers, porém não tão eficientes para serem Subscribers. Estes últimos são implementados na camada de aplicação, onde estão dispositivos com maiores recursos e com a função de receber dados e criar aplicações em cima desta função básica.

3.3.4.1 Embarcados

Embarcados, são sistemas alimentados por baterias, sem alimentação de rede elétrica, portáteis, econômicos, com sistemas de controles geralmente feitos por microcontroladores ou microprocessadores, podendo contemplar sistemas operacionais leves. Com essa descrição, pode-se imaginar que estes dispositivos possuem processamento, energia e desempenho limitados. Para isso foi necessário a criação de uma implementação de interface leve e eficiente

Foi escolhida as plataformas microcontroladas pela arquitetura Espressif [18]. MCUs (Micro-Controller Units) que contemplam processadores e módulos WiFi e até Bluetooth (não utilizado na interface atual), mostrados em Figura 13 na arquitetura do esp32, pela des-

criação técnica pode-se ver um poder de processamento maior que um Arduino [19], muito utilizado nessas aplicações e que também é compatível com a interface se adicionado shields WiFi. O ESP utiliza linguagem C++ [20] para desenvolvimento do Publisher e Data Stream, com framework Arduino que permite implementação em outras plataformas, além de uma firmware escalável, circuito open hardware e open source.

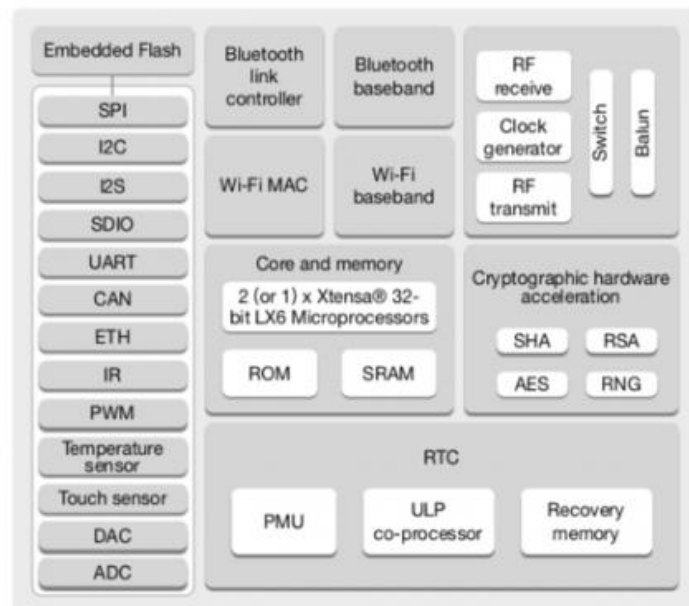


Figura 13 A arquitetura do ESP32, retirado de [18]

Também foi implementada em Node.js (que será descrito abaixo) aplicações do Publisher para embarcados com sistemas operacionais, [21], como Raspberry Pi [22] e sua arquitetura mais robusta, e Intel Galileo. Permitindo multi-uso entre as funções de Publisher e Subscriber, podendo ser utilizados como Hubs ou bridges de dados. Esses consoles possuem, processadores mais potentes, periféricos, Sistemas Operacionais, assim como entradas e saídas digitais.

3.3.4.2 Consoles

Consoles são sistemas que contemplam sistemas operacionais, o que permitem mais liberdade para a implementação da interface. Foi escolhida então, realizar a implementação com Node.js [23], um ambiente de Javascript que permite criar aplicações fora do browser, além de outras aplicações, como mobile e desktop. Possui extensas bibliotecas para HTTP e MQTT além de pipelines que permitem fácil comunicação de protocolos no mesmo processo, o que é fundamental para o conceito de escalabilidade

deste projeto.

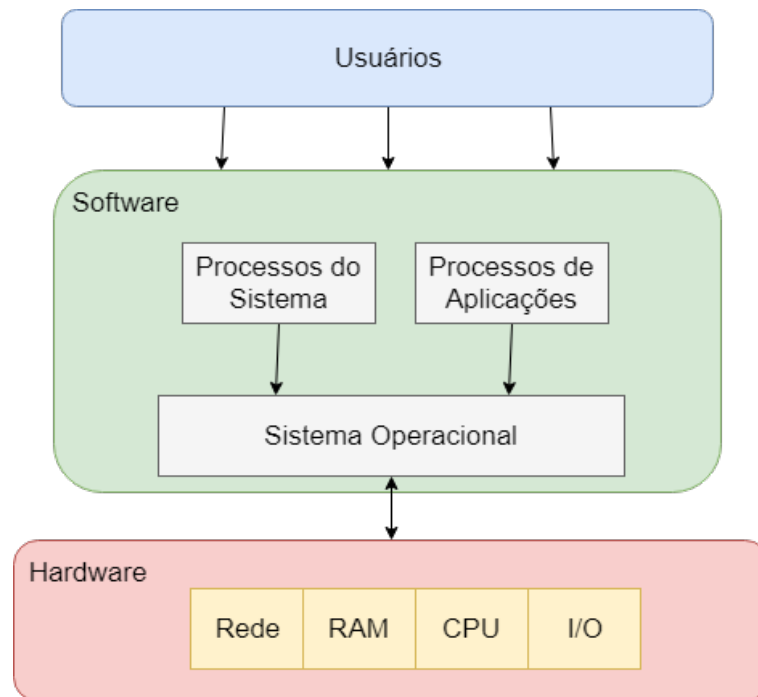


Figura 14 A arquitetura simplificada de dispositivos com Sistema Operacional

Além disso Node.js é uma ferramenta multiplataforma, com distribuições para Windows, Linux e MAC, além de versões para embarcados de arquitetura ARM, como o próprio Raspberry Pi. Possui Módulos que permitem acessar processos do sistema operacional como ilustrado na Figura 14 permitindo acesso a Rede além de informações do próprio sistema. O ambiente permite a implementação com programação orientada a objeto, Publishers, Data Streams, Subscribers, são instancias de classes mostradas no apêndice em 5.2, permitindo que a aplicação seja feita em um processo (obs: este processo pode executar outros processos, com algum overhead). Com isso foram implementadas bibliotecas que constroem e interface sobre o MQTT, no lado Subscriber do sistema.

3.4 Arquiteturas e Assíncronismo

Na seção anterior foi discutido a implementação em Hardware do sistema e as diferenças das tecnologias contempladas. Sistemas embarcados possuem muito menos opções de implementação que um console gerido por um sistema operacional, sendo uma das principais o paralelismo de processos, a capacidade de ser multi-tarefas, por isso exigiu-se duas filosofias diferentes de implementação do sistema para os dois tipos de plataformas.

3.4.1 Embarcados Síncronos

Sistemas embarcados possuem um poder de processamento limitado, apesar da tendência de elevar este poder na ponta, estes sistemas são geralmente Microcontrolados. Microcontroladores possuem arquiteturas mais simples, geralmente executando instruções de um programa compilado para linguagem do MCU e gravado neste. Não há paralelismo, cada instrução é síncrona, ou seja são executadas uma a uma, a próxima deve esperar a anterior acabar de ser executada pela unidade de processamento.

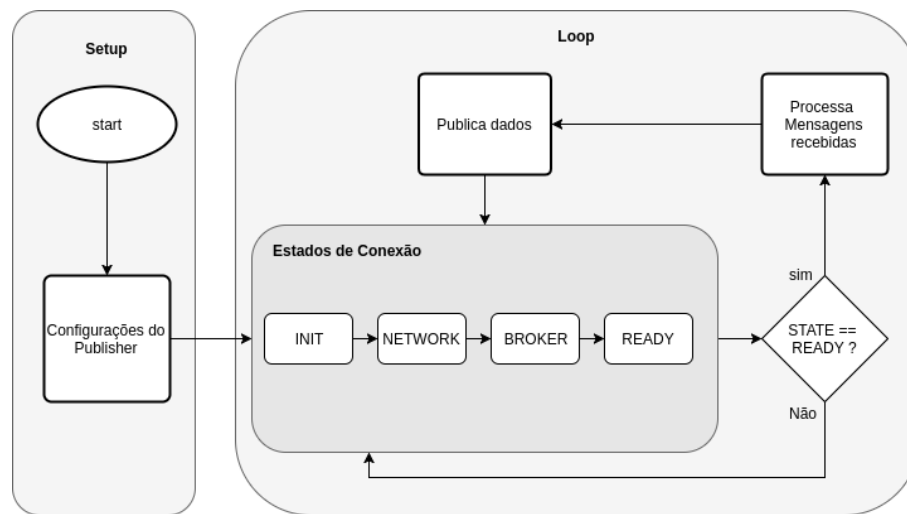


Figura 15 Diagrama simplificado de uma rotina padrão seguida pela implementação em Microcontroladores

A Figura 15 simplifica a lógica da implementação em dispositivos síncronos como microcontroladores. No Apêndice deste documento pode-se encontrar as implementações em C++. O programa começa configurando o objeto Publisher (MQTT) com configurações de conexão a rede e o broker, além de outras definições do desenvolvedor. O bloco de loop repete-se indeterminadamente, verificando o estado de conexão da aplicação que passa pelos seguintes estados:

- INIT: Estado inicial, verifica conexão com rede;
- NETWORK: Tem conexão com rede, verifica conexão com Broker;
- BROKER: Tem conexão com Broker, inicia configurações dos Data Streams;
- READY: Pronto para enviar e receber mensagens !

Quando o estado está em READY, a aplicação está pronto para processar mensagens recebidas e publicar, lembrando que está é uma lógica básica, o desenvolvedor pode adicionar outros passos, mas a verificações de estado e as configurações do Publisher são obrigatórias para o funcionamento do Sistema. Repare que toda a lógica é sequencial, síncrona, não há paralelismo na rotina.

3.4.2 Consoles Assíncronos

Consoles são dispositivos que possuem uma Arquitetura mais complexa, consequentemente mais poder de processamento. Ilustrado na Figura 14, a presença de um sistema operacional gerenciando processos do sistema e de aplicações e informações do Hardware permitem a execução de múltiplas tarefas em paralelo, o que caracteriza processos Assíncronos. A linguagem utilizada para a implementação foi Javascript com o ambiente Node.js, um interpretador da linguagem que pode ser usado fora de um Browser. Node possui uma gama de suportes para a implementação de funções assíncronas, permitindo o sistema a explorar essa característica e, diferentemente dos embarcados microcontrolados, executar Threads em paralelo, como receber e enviar mensagens em paralelo simultaneamente.

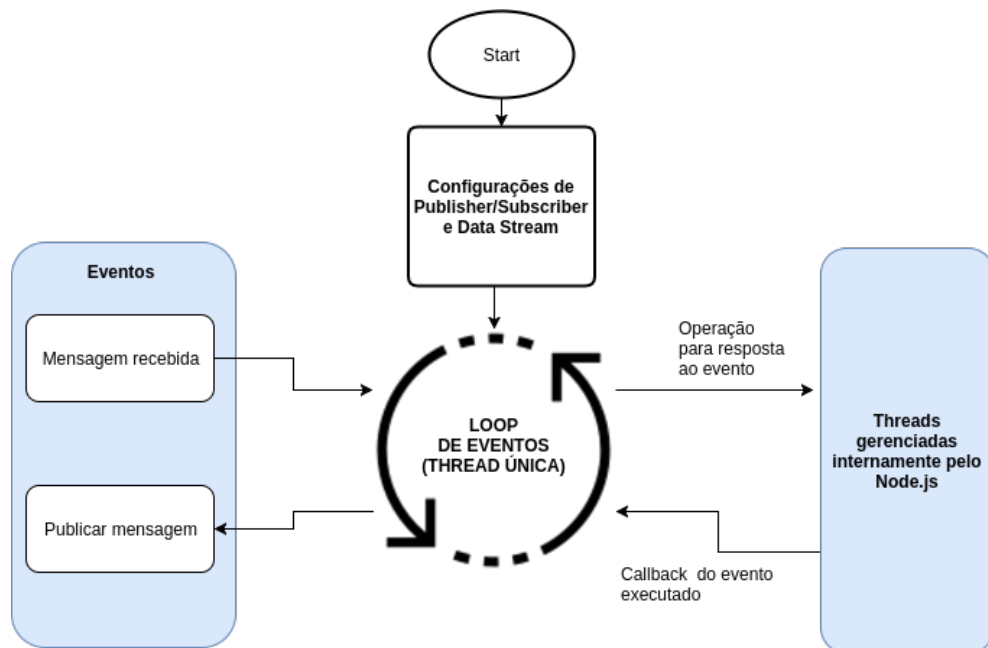


Figura 16 Diagrama simplificado de uma rotina assíncrona padrão em Consoles

A Figura 16 mostra o funcionamento simplificado das aplicações de Publisher e Subscriber em Node.js. A aplicação faz as configurações específicas de cada objeto Pu-

blisher ou Subscriber e logo após entra em um Loop de eventos, onde existem eventos de Conexão com rede, com o Broker e de envio/recebimento de mensagens. Estes eventos quando são acionados pelo sistema, executam callbacks, funções assíncronas registradas previamente nas configurações. Essas funções assíncronas são gerenciadas internamente pelo Node.js e direcionadas ao loop de eventos.

3.5 Segurança de aplicações

SSL [24] (Secure Sockets Layer) é a tecnologia de segurança padrão para estabelecer um link criptografado entre um servidor da Web e um cliente. Um certificado SSL em seu servidor e um navegador se conecta a ele, a presença do certificado SSL aciona o protocolo SSL (ou TLS), que criptografa as informações enviadas entre o servidor e cliente.

O SSL opera diretamente no topo do protocolo de controle de transmissão (TCP), além de permitir que camadas de protocolo de aplicações sejam construídas por cima, agora com sob uma camada de segurança. Portanto, sob a camada SSL, as outras camadas de protocolo podem funcionar normalmente, como o HTTP e o MQTT.

Com um certificado SSL, todos os invasores poderão saber qual IP e porta e quantos dados estão sendo enviados. Eles podem terminar a conexão, mas tanto o servidor quanto o usuário poderão dizer que isso foi feito por terceiros. No entanto, eles não serão capazes de interceptar qualquer informação, o que a torna essencialmente um passo ineficaz. O invasor pode descobrir qual nome de host, mas como a conexão é criptografada, as informações importantes permanecem seguras.

Para poder criar uma conexão é requisitado um Certificado SSL. Quando você optar por ativar o SSL em seu servidor da Web, será solicitado que você responda a várias perguntas sobre a identidade do seu site e da sua empresa. Seu servidor da Web cria duas chaves criptográficas - uma chave privada e uma chave pública.

A chave pública não precisa ser secreta e é colocada em uma solicitação de assinatura de certificado (CSR) - um arquivo de dados que também contém seus detalhes, então deve-se enviar o CSR. Durante o processo de solicitação do Certificado SSL, a Autoridade de Certificação validará seus detalhes e emitirá um Certificado SSL contendo seus detalhes e permitindo que você use SSL.

Além do SSL existem outras formas de segurança. A maioria envolve encriptação. Como o uso de algoritmos de hash para encriptar as mensagens enviadas, ficando a cargo

da aplicação descriptar.

A Interface aqui construída é transparente ao SSL, ou seja, pode ser implementada em cima ou não deste protocolo, os exemplos são feitos sem esta camada, porém basta criar ou adquirir um certificado em uma autoridade de certificação e gerar as chaves. As APIs usadas contemplam da opção de usarem clients SSL.

3.6 Persistência de dados

Os dados adquiridos pela plataforma e suas camadas, são armazenados em memórias e enviados. Memórias voláteis que podem facilmente perder dados com quedas de energia o reaproveitamento do sobre-inscrição do próprio gerenciamento do sistemas, para garantir que os dados não sejam perdidos, é necessário que o sistema possua persistência, uma forma de memória não-volátil que armazene os dados sem energia.

Essa persistência é implementada com Banco de Dados, estruturas que organizam o armazenamento de dados persistentes em arquivos. Um Banco de dados é uma basicamente uma aplicação, um serviço do sistema que recebe requisições de rede e escreve ou lê dados em um arquivo. Existem inúmeras formas de implementação e protocolos de comunicação para Bancos de Dados. Porém todos eles seguem abstrações em comum.

Um banco é composto por duas ferramentas. O Motor e o Arquivo de dados. O motor é quem realiza as ações sobre o arquivo, é o sistema de gerenciamento. Recebe as requisições e aplica algoritmos de escrita de dados eficientes no arquivo para armazenar os dados em uma estrutura definida. Um Banco de dados pode possuir vários motores, cada um com algum algoritmo que varia a eficiência e o tempo de escrita e/ou leitura dependendo do dado recebido.

O Arquivo é o documento onde os dados são armazenados, na estrutura definida. Possuem formatações de dados específicas de cada tipo de banco, seguindo uma abstração. O formato do armazenamento de dados, define e limita eficiência do motor, então é necessário a escolha adequada de algoritmo para uma maior eficiência da leitura do arquivo. Como em Figura 17 a aplicação envia uma solicitação de criação ou leitura ou atualização ou remoção (CRUD - Create, Read, Update, Delete) e o motor lida com os dados, armazenando-os em uma determinada estrutura no Arquivo.

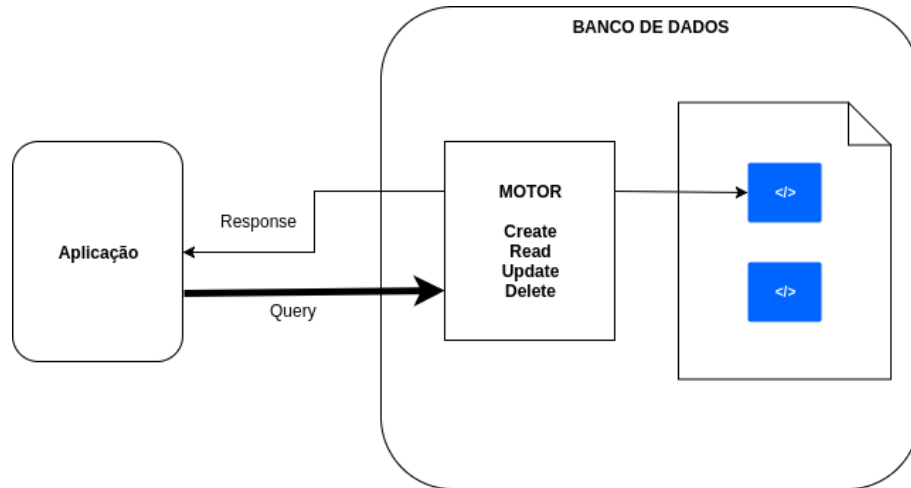


Figura 17 A arquitetura de um banco de dados

3.7 Bancos para Aplicações IoT

Na área de Bancos de Dados (ou DBs), com escopo focado em IoT, há um porém. Para a estruturação de uma aplicação eficiente, não se deve usar qualquer tipo de banco. Devem-se escolher motores e estruturas de bancos adequadas para a aplicação. Não existe um critério definitivo que guia o desenvolvedor para melhor escolha. Mas algumas características de bancos de dados podem ser exploradas em aplicações IoT, formando um DB eficiente para tal.

Como pode ser observado em [25], uma aplicação eficiente de Bancos e IoT está ligada ao tempo de inserção de dados no banco, o tempo total em que a aplicação leva para enviar, aplicar a busca de onde o dado deve ser inserido no documento e de fato armazenar, podem ser feitas várias inserções de pequenos pacotes de dados, dependendo do tamanho da mensagem. Esta característica está ligada ao motor do banco, que determina como o dado será armazenado, e quanto ele leva para escrever o dado no documento. A maioria dos dados são armazenados em estruturas B+Trees [26], porém pode-se observar em [25] que a estrutura LSM Tree [27] possui maior eficiência na escrita.

Comparando as duas estruturas, como mostrado em Figura 18, percebe-se que a estrutura LSM sustenta até 2x mais inserções da estrutura B+, hoje os Bancos de Dados modernos utilizam desta vantagem, devida a nova tendência de aplicações de coletar dados constantemente, o que leva a operação de escrita no banco ter mais importância e ocorrências que a leitura e explica a migração para a estrutura LSM.

Existem várias abstrações de Bancos de dados. Em [29] compara-se e conclui-

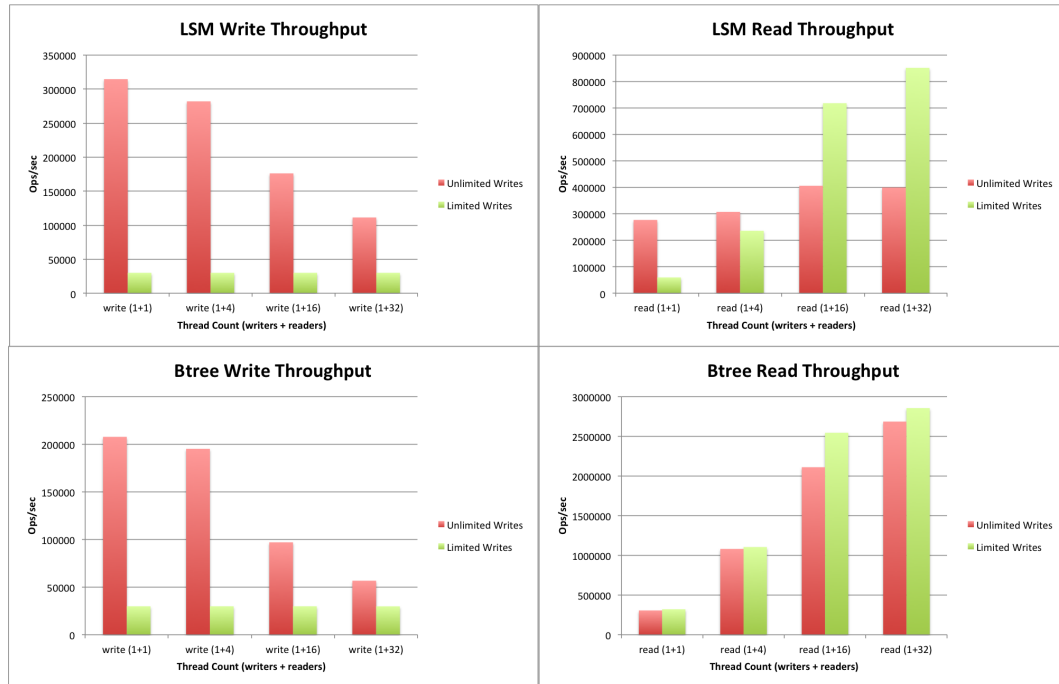


Figura 18 Comparação de Escrita e Leitura entre LSM e B+, retirado de [28]

se que o banco MongoDB, um banco NoSQL possui um tempo de resposta de inserção menor que o MySQL (bancos relacionais), porém o último é mais estável. De fato bancos NoSQL são, em geral, mais leves, possui uma flexibilidade maior para lidar e estruturar dados, o que fazem estes tipos de Banco mais favoráveis a aplicações de IoT. Porém outras estruturas como um banco dividido em timescale mostram-se eficientes SQL ou não.

Outros aspectos podem contribuir para a eficiência de persistência de dados. Criar bancos locais diminuem a latência e a necessidade de conexão, aumentando a capacidade de inserção de dados, além de ser uma forma de backup de dados. Um banco local, geralmente fica em uma plataforma como em [30], são bancos leves em aplicações de baixo consumo, devido a capacidade de processamento limitada. Seu papel é geralmente para armazenar os dados quando não há conexão, e quando esta é restabelecida os dados são enviados para um banco remoto com mais capacidade de processamento e aplicações.

Dentre os bancos estudados, alguns se destacam como o Cassandra, usado pela Netflix para coletar dados sobre o comportamento do usuário na plataforma ou o InfluxDB, um banco de arquitetura TimeScale, feito para aplicações em tempo real. Mas para esse projeto, foi utilizado o MongoDB [31], um banco NoSQL, leve, de fácil integração com as plataformas utilizadas e que possui implementações de motores que priorizam a eficiência na escrita de dados, como a LSM tree.

3.8 Indexação de dados e Timestamp

Na seção anterior foram discutidos estruturas da organização de dados e o formato de armazenamento de dados como o formato de Documento e TimeScale. É importante que estes formatos tenham formas eficientes de indexação, de modo a facilitar a busca e a análise de dados.

Em uma aplicação de IoT, que envolve coleta de dados em tempo real, é fundamental, independentemente do formato escolhido, a informação de quando este dado foi colhido (data e hora). Isto permite a análise dos dados ao longo do , conforme forem armazenados. Aplicações de decisão e análise utilizam ferramentas estatística com base nas ocorrências temporais, projetando previsões e classificação. No projeto atual, foi implementado o MongoDB um banco de dados de documentos. Cada documento é indexado por uma identificação única como mostrado na Figura 19, permitindo também criar relações entre documentos.

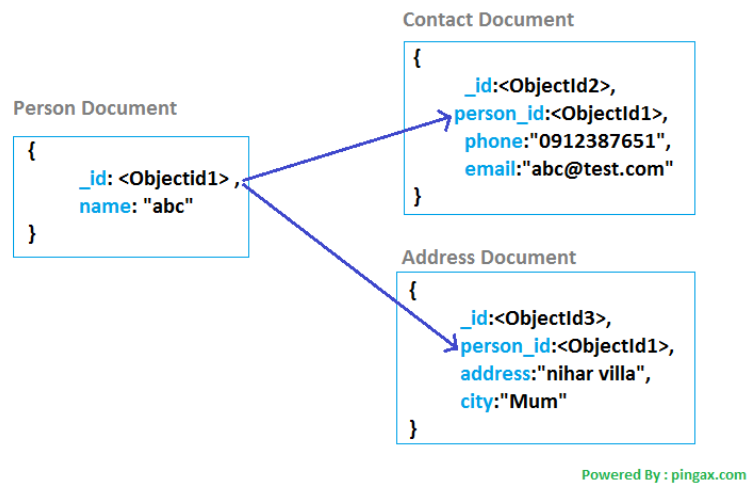


Figura 19 O formato de documento no MongoDB

O banco também permite adicionar outros parâmetros de indexação definidos pelo desenvolvedor, foi utilizado desta funcionalidade para adicionar um campo de Timestamp, uma informação de data e hora da inserção no banco. Na implementação foi utilizado o formato Unix Timestamp, o número de milissegundos que se passaram desde 1 de Janeiro de 1970 às 00:00:00 (UTC), a Figura 20 ilustra o formato de documento completo usado no sistema, o campo de dados é definido pelo desenvolvedor.

Vale ressaltar a existência de uma nova geração de Bancos de Dados TimeScale, um formato parecido com o de Documento, porém com indexação feita por Timestamp

```
  _id: ObjectId("5b91eec374a9243d4e929513")  
  timestamp: 1536290499833  
  ✓ data: Object  
    device: "esp32"  
    temperature: 38.333  
    main: 38.333
```

Figura 20 Adição de parâmetro de timestamp em milisegundos ao documento

ao invés de uma chave única. Em destaque o Banco InfluxDB, que possui estrutura LSM-Tree além de ser um banco TimeScale, o que o faz ser utilizado cada vez mais em aplicações IoT.

4 CASOS DE USO

No capítulo 3 descrevemos o funcionamento da Interface e sua comunicação com a linguagem MQTT. Este capítulo busca demonstrar o funcionamento do sistema em hardwares com a interface implementada pelos softwares descritos na seção 5.2 disponível no Apêndice. São aplicações simples que mostram a facilidade e a escalabilidade do sistema, além de demonstrar como o sistema pode ser implementado em plataformas.

4.1 Medição de temperaturas de CPU

obs: *Para reproduzir este teste, é necessário seguir as instruções encontradas no Apêndice.*

Este exemplo tem como objetivo medir a temperatura da CPU de um console com baseado em suas atividades, serviços e processos em execução. A aplicação pode ser escalada para a obtenção de outras informações da CPU e do sistema, podendo assim disponibilizar análises de desempenho da plataforma, além de montar perfis de uso do sistema e administrar seu uso.

A escalabilidade do sistema será testada em partes, a primeira será analisar medições de temperatura de uma CPU de um console. Em seguida comparar medições de temperatura da CPU com um ESP32 e por último analisar temperatura de múltiplas CPUs.

Para isso precisaremos utilizar uma instância da classe Publisher disponível 5.2.2 como e no console a ter informações de temperatura a ser coletadas e uma instância do Subscriber 5.2.3 para receber estas temperaturas via MQTT e persisti-las em banco de dados. Ambas as aplicações utilizarão as APIs em Javascript, utilizando Node.js para coletar as informações do sistema, implementar o Publisher, o Subscriber, o driver para MongoDB (também disponível em anexo) e a geração de um gráfico utilizando a plataforma plotly [32]. Isso foi implementado no código 5.2.4.

CPUs são tecnologias feitas por transistores. Milhões de aglomerações de MOSFETS que começam a ter perda de performance no processador conforme o aumento de temperatura, em [33], pode-se observar os efeitos do aumento de temperatura nos parâmetros do MOSFET especialmente na queda de mobilidade e na velocidade de saturação que provocam perda de performance. CPUs modernas são capazes de ajustar suas frequências operacionais, a fim de reduzir seu consumo de energia ou fornecer a máxima

potência, conforme necessário e possuem também proteção térmica extremamente robusta. Se a unidade começar a operar acima do limite térmico, ela começará a reduzir a frequência para evitar uma falha catastrófica.

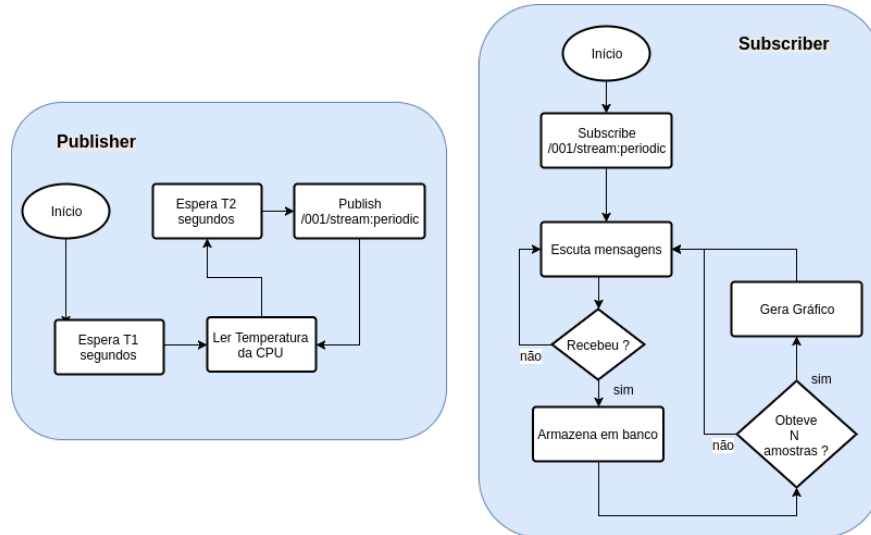


Figura 21 Diagrama de fluxos do Publisher e do Subscriber

A Figura 21 mostra todo o fluxo das duas aplicações, o Publisher publica em no tópico `/001/stream:periodic`, a informação coletada a cada $T1=3$ segundos e espera $T2=1$ antes de enviar. O Subscriber escuta este tópico e persiste ao chegar uma mensagem de dados pelo Data Stream, ao atingir $N=100$ amostras, um gráfico de Temperatura da CPU principal pela Data-Hora de inserção é gerado com as últimas 100 inserções no banco.

Repare que a medição depende da data e da hora de inserção no banco, o tempo de chegada até a persistência varia muito com a latência e com o processamento da aplicação, desta forma temos uma medida mais constante. A Figura 22 mostra o formato de dado armazenado, com a ferramenta Compass para visualização de dados do MongoDB. Repare que lidamos com a estrutura de dados em documento porém obrigatoriamente todo documento da Interface possui o timestamp da inserção no banco, o campo `data` é o objeto de dados em medição. A comunicação com o banco pode ser feita por uma instância da classe `MongoDataClient` em 5.2.3 que cuida da inserção com o *timestamp* e o objeto de dados em *data*.

A visualização de dados é feita pela ferramenta Plotly, um serviço que fornece uma interface para criar, editar e analisar gráficos, basta criar uma conta, gratuita ou paga, e o usuário poderá criar gráficos na plataforma web ou através de APIs implementadas em múltiplas linguagens de programação conhecidas. A aplicação do Subscriber utiliza

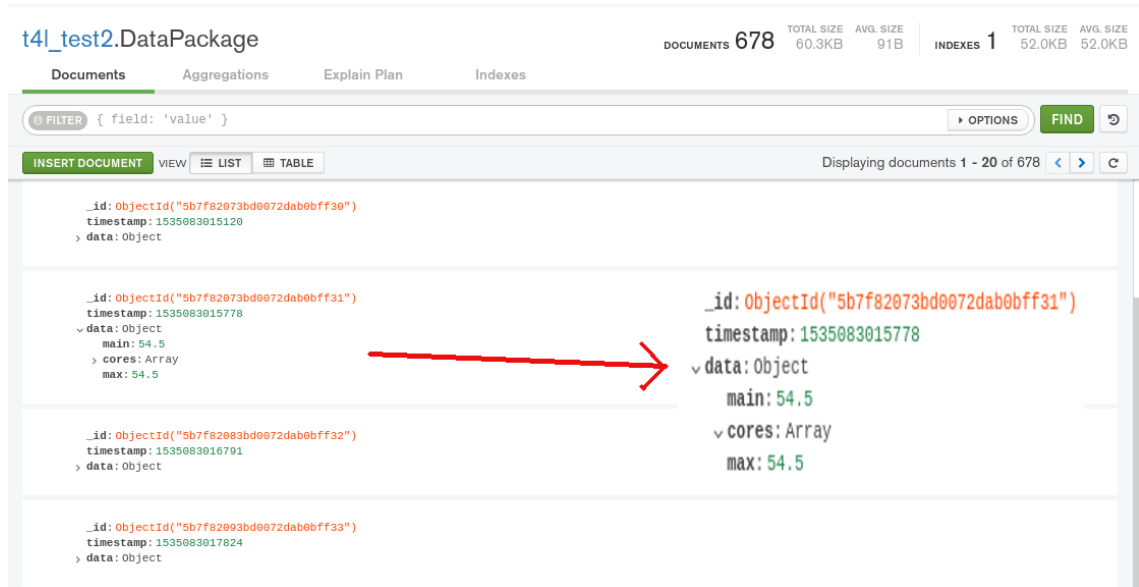


Figura 22 Visualização dos dados armazenados em documento em uma coleção do MongoDB

da segunda opção com o módulo plotly.js, a implementação em Javascript da plataforma. A cada $N=100$ amostras são produzidos gráficos como o da Figura 23.

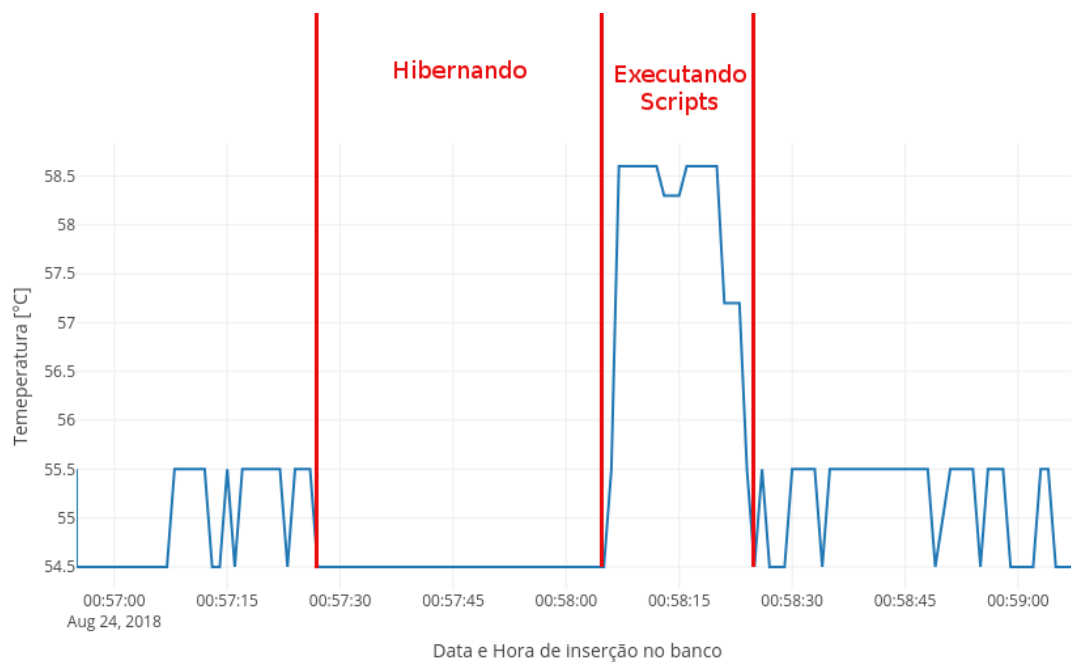


Figura 23 Comportamento da temperatura em três momentos

A Figura 23 mostra a variação da temperatura de uma CPU da Intel Core I7 em três momentos. O gráfico começa no primeiro momento onde só os processos básicos do computador estão em execução, mantendo a temperatura constante, logo em seguida

entra o momento onde o computador está hibernando o que leva a uma pequena baixa na temperatura. O terceiro momento descreve o comportamento quando o computador executa o MATLAB em um script que exige capacidade de processamento, causando uma leve alta de temperatura, mas não tão alta devido a capacidade do processador e depois estabilizando e voltando ao primeiro momento. Com isso fechando o ciclo das aplicações.

Finalizada a medição de uma CPU, pode-se escalar para uma Análise mais elaborada, junto as medições desta CPU core i7, foi acrescentada uma aplicação contendo um Publisher em um ESP32. Conforme descrito no capítulo anterior na Figura 13, o ESP32 é um MCU que possui módulos WiFi e sensor de temperatura interno, facilitando a medição, como trata-se de um Microcontrolador foi utilizado a implementação em C++ 5.2.5, uma implementação síncrona.

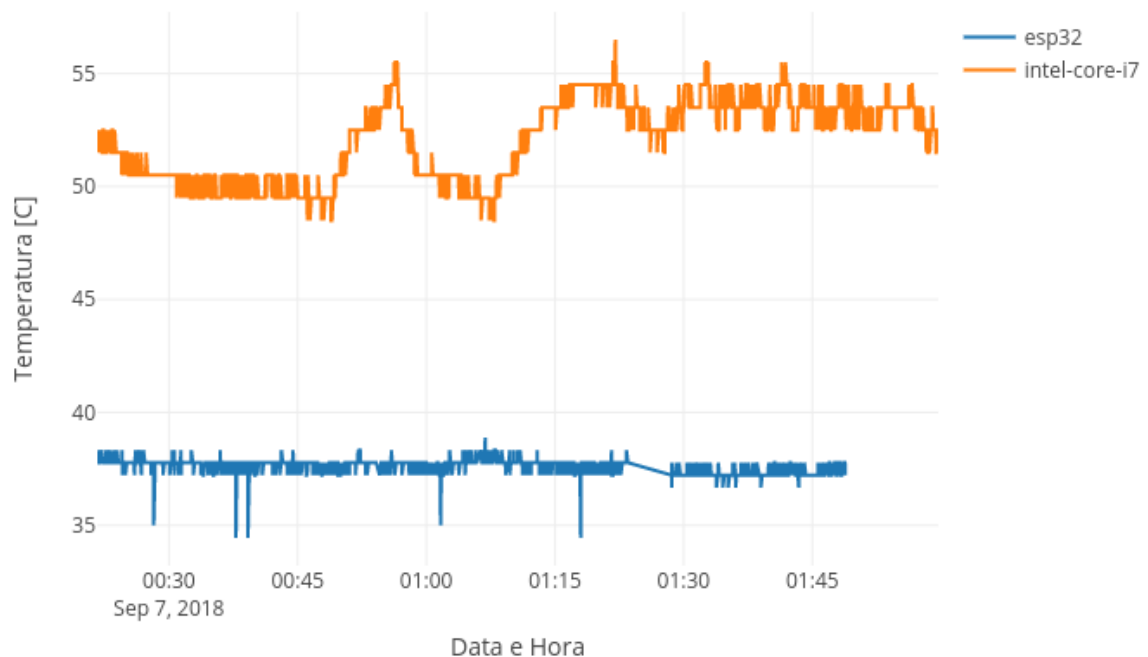


Figura 24 Comparação das temperaturas com uma CPU core i7 e ESP32

Ambas as plataformas foram configuradas para enviar informações de temperatura com intervalos de um segundo, durante um período de cerca de uma hora e meia, ao completar 10 mil amostras totais, um gráfico é gerado. Cada plataforma contribui com cerca de metade das amostras, porém cada uma possui uma rotina e capacidade de pro-

cessamento, fora o tempo de envio para Broker, é de se esperar que o número de amostras de cada um não seja igual. Pela Figura 24 pode-se observar as diferenças de temperatura e o comportamento das duas CPUs. O ESP32 com sua arquitetura mais simples, mantém-se relativamente constante a 30 graus Celsius, isso pode ser explicado devido ao fato do ESP estar rodando somente um programa devido suas limitações. A CPU i7 varia sua temperatura em torno de 50 graus Celsius, possuindo variações mais bruscas, devido ao fato do processador está executando múltiplos processos. Durante este experimento o computador da CPU foi usado normalmente, passando por cenários de hibernação até o uso do Browser, recebendo requisições de páginas da web, assistindo vídeos e áudios, provocando as variações de temperatura semelhantes ao primeiro teste.

Por último o sistema foi utilizado em um teste em múltiplos computadores com processadores distintos no Laboratório PROSAICO. A finalidade foi observar o comportamento e a robustez da aplicação ao receber dados de múltiplas fontes em larga escala (cerca de 15 mil amostras totais no banco).

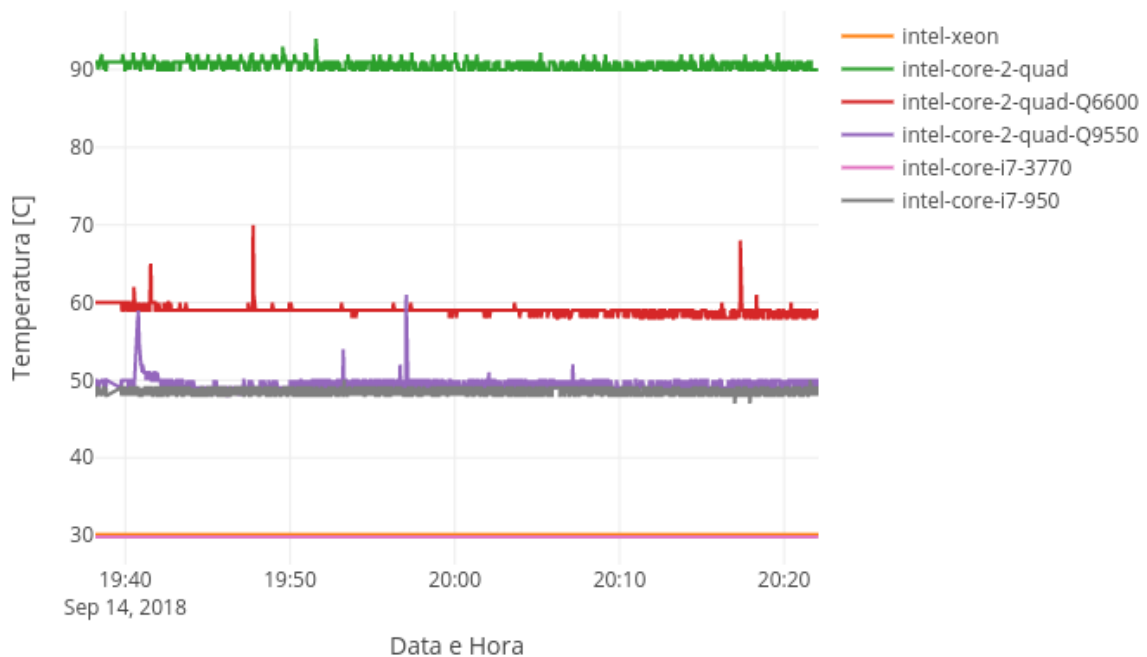


Figura 25 Comparação das temperaturas em múltiplos processadores

Na Figura 25, observa-se vários níveis de temperatura média entre os processado-

res, todos executando processos comuns dos sistemas operacionais em máquinas Windows e Linux. Esta diferença está atrelada aos fatores de resfriagem térmica e a capacidade de processamento. Vale ressaltar alguns comportamentos. Os dois processadores de temperatura mais baixa, por volta de 30 graus célsius possuem gabinetes mais novos e apresentam um sistema de resfriamento melhor que os mais antigos. Já o processador que registrou uma maior temperatura média, fora o processador do computador usado como servidor Web e IoT do laboratório, hospedando o Broker Mosquitto do sistema, inclusive. Dois fatores contribuíram para essa mudança abrupta de tecnologia, os processos executados dos serviços, porém majoritariamente pela manutenção do resfriamento do processador, foi detectado que a CPU estava sem pasta térmica, grande responsável pela troca de calor na própria.

4.2 Aplicação: Automação Residencial

Para validar o baixo custo do sistema, as próximas seções tem o foco em simular o orçamento de dois projetos. Um feito para a indústria e outro em redes domésticas para automações residenciais. Serão propostos cenários de aplicações reais, capazes de confirmar a premissa do projeto. Como vimos no capítulo de projetos, o sistema pode ser instalado em plataformas mais sofisticadas, contendo sistemas operacionais e hardware dedicado. Mas nestes cenários utilizaremos o hardware mínimo que é compatível com o sistema e atua satisfatoriamente na aplicação.

O caso em estudo é a automação parcial de uma residência. Como pode ser visto em Figura 26, a casa possui dois quartos, sala de estar, dois banheiros, cozinha, área de serviço e varanda. O objetivo é monitorar a temperatura local, o consumo de energia, detectar aberturas de portas e janelas de entrada da residência para fins de segurança e acionamento de luzes.

Para isso é necessário acionadores para a sala e cozinha, mais os quartos, totalizando cerca de 4 pontos de luz para acionar, o mesmo vale para os sensores de temperatura, no qual farão a média de temperatura da casa. Para controle de consumo de energia, nos limitaremos as tomadas de eletrodomésticos e eletrônicos, que representam a maior parte do consumo, para um ponto na sala, nos quartos, na tomada da geladeira, micro-ondas e lavadora, cerca de 6 tomadas a colocar sensores de tensão e corrente AC para cálculo da potência. Serão colocados sensores magnéticos para detectar abertura de portas e janelas,

na porta de entrada, na porta da varanda, nas janelas do quarto e na área de serviço.



Figura 26 Planta baixa de residência de dois quartos, retirado de [34]

Nota-se pela Tabela 3, com menos de R\$ 2000,00 pode-se instalar um sistema robusto para automatizar uma casa. O projeto já conta que a residência possui uma rede WiFi e se a casa tiver um PC, pode-se abater do custo do servidor.

Tabela 3 Orçamento de um sistema simples para automação da residência da Figura 26

Item	Descrição	Qtde	Unidade (R\$)	Total (R\$)
esp32	Módulo de aquisição	4	\$30.00	\$120.00
DHT11	Sensor de temperatura	4	\$10.00	\$40.00
P8	Módulo sensor de tensão	6	\$20.00	\$120.00
Acs712 - 5a	Módulo sensor de corrente	6	\$15.00	\$90.00
Ssr-25	Relé Estado Sólido	4	\$30.00	\$120.00
Desktop	Servidor Local	1	\$600.00	\$600.00
Sensores Magnéticos	Sensores de abertura	6	\$40.00	\$240.00
Infraestrutura	Caixas de proteção, fios etc	1	\$500.00	\$500.00
TOTAL:				\$1,830.00

4.3 Aplicação: Controle de Iluminação de Postos de combustível

Essa é uma aplicação que pode ser usado no comércio assim como na indústria. Trata-se de uma central que controla via rede local as iluminações de postos de com-

bustível. Estes possuem uma fonte de luz bastante potente na testeira(Parte de cima do posto) e no Totem (presente na entrada do posto, onde são mostrados os preços dos combustíveis). A infraestrutura do projeto não é complexa, Necessitará de um módulo ESP32 para processamento e envio dos acionamentos, relés de estado sólido, disjuntores de curva C, uma bateria que converte CA em CC, geralmente baterias similares as de celulares satisfazem, isso para o interfaciamento entre a rede elétrica e o sistema digital de controle. Por fim tudo isso ficará presente em uma caixa com vedação perto da caixa de energia do posto.

Para montar a rede Wifi local, será utilizado um roteador dedicado, configurando uma rede local, não é necessário a conexão a internet neste ponto do projeto. O ESP32 irá se conectar a rede por wps [35], O broker irá ser instalado em um desktop na parte administrativa do posto. O diferencial da aplicação está na aplicação. O frentista responsável poderá acessar a rede e um servidor HTTP irá fornecer uma página web no qual ele pode enviar comandos de acionamento das luzes e até agendar o acionamento destas para um horário específico.

Tabela 4 Infraestrutura para sistema de controle de iluminação de postos de Combustíveis

Item	Descrição	Qtde	Unidade (R\$)	Total (R\$)
esp32	Módulo de aquisição	1	\$30.00	\$30.00
Disjuntor	Curva C	4	\$40.00	\$160.00
P8	Módulo sensor de tensão	6	\$20.00	\$120.00
Acs712 - 5a	Módulo sensor de corrente	6	\$15.00	\$90.00
Ssr-25	Relé Estado Sólido	4	\$30.00	\$120.00
Desktop	Servidor Local	1	\$600.00	\$600.00
TP-Link AC1350	Roteador Wireless	1	\$280.00	\$280.00
Infraestrutura	Caixa de proteção, fios etc	1	\$700.00	\$700.00
TOTAL				\$2,100.00

Na Tabela 4, como temos um sistema trifásico é necessário mais acionadores, como as ferramentas utilizadas para o projeto da aplicação são de escopo aberto e gratuitas, não possuímos despesas neste aspecto. Deve-se planejar um local para que a caixa de controle consiga captar um sinal estável do roteador, e é recomendável configura-lo para enviar dados em um canal menos congestionado. O sensores de tensão e corrente foram adicionados caso seja adicionado ao projeto um monitoramento de consumo de energia.

CONCLUSÃO

Nesta dissertação, foram apresentados todos os aspectos de hardware, software e econômicos em seus capítulos. Cada um explicando o planejamento a a forma de implementação da ideia, de modo a deixar claro ao leitor como construir e utilizar o sistema como um todo.

O sistema segue sua proposta de escopo aberto, escalável e de baixo custo. Foram apresentadas plataformas de hardware aberto, oferecendo a possibilidade de uma organização ou empresa distribuírem suas próprias versões e baratear ainda mais os custos em larga escala. Os softwares são de escopo aberto e licenças altamente permissíveis, o que significa que possuem versões gratuitas e escaláveis, de modo a também oferecerem a possibilidade de criar versões personalizadas. E por último as versões prontas para uso são de baixo custo, como comprovado em cotações feitas em 4.2 e 4.3.

A escolha de começar com aplicações feitas com base no TCP/IP, tornou o software flexível para a implementação de novos tipo de protocolos desta camada. Unido a grande quantidade de bibliotecas feitas em diversas linguagens de programação, o sistema pode ser escalado facilmente para aplicações específicas. O modelo Publish/Subscribe é um modelo que reúne características ideais para protocolos full-duplex, em transferências em tempo real, oferecendo a funcionalidade de cada dispositivo escolher quais dados desejam receber através do sistema de tópicos, oferecendo economia energética e de dados.

No capítulo 4, foram simulados dois casos de uso, para aplicações residenciais e na indústria/comércio, baseado em preços de mercado, não considerando os preços em atacado que podem baratear o custo para um empresa que deseja vender uma versão desse sistema. O custo total mostrou-se parecido nos dois casos, o que mostra que o sistema é flexível para implementações nos dois tipos de clientes, por outro lado, como a indústria possui uma movimentação monetária e mais recursos para investimento, pode ser mais vantajoso aplicações industrias o do que domésticas, já que o preço é razoavelmente constante.

Vale ressaltar sobre plataformas em nuvem que fornecem serviços IaaS, que podem ser inseridos nas camadas de IoT. Serviços como Brokers, Bancos de Dado, Autenticação, segurança, Inteligência Artificial, Análise e Visualização de dados, estão cada vez mais frequentes no universo do IoT. Isso facilita a implementação do sistema e podem oferecer

soluções mais confiáveis e estáveis, porém isso aumenta o custo do sistema, pois esses serviços são pagos.

Por fim, este documento contempla e instrui o usuário sobre a Indústria 4.0, sobre Internet das Coisas e uma forma flexível de implementação do sistema. Para futuras funcionalidade, pode-se incluir ferramentas de segurança, integração com a cloud, outras implementações para bancos de dados e implementações da API em outras linguagens de programação.

REFERÊNCIAS

- [1] IBM. *MQTT*. Disponível em: <<http://mqtt.org/>>. Acesso em: 21/07/2018.
- [2] ASHTON, K. That 'internet of things' thing. v. 22, p. 97–114, 01 2009.
- [3] DIAS, R. R. de F. *Internet das Coisas sem Mistérios*. [S.l.]: Netpress Books, 2016.
- [4] EVANS, D. The internet of things how the next evolution of the internet is changing everything. Cisco Internet Business Solutions Group (IBSG), 04 2011.
- [5] ZIMMERMANN, H. Innovations in internetworking. In: PARTRIDGE, C. (Ed.). Norwood, MA, USA: Artech House, Inc., 1988. cap. OSI Reference Model&Mdash;The ISO Model of Architecture for Open Systems Interconnection, p. 2–9. ISBN 0-89006-337-0. Disponível em: <<http://dl.acm.org/citation.cfm?id=59309.59310>>.
- [6] E., C. D. *Internetworking with TCP/IP - Volume I: Principles, Protocols and Architecture*. 3. ed. [S.l.]: Prentice Hall, 1995.
- [7] Sigfox. *Sigfox Technology Overview*. Disponível em: <<https://www.sigfox.com/en/sigfox-iot-technology-overview>>. Acesso em: 21/07/2018.
- [8] LoRa Alliance, Inc. LoRaWAN™ 1.0.3 Specification. 2017.
- [9] Endeavor Brasil - Time De Conteúdo. *Beacon: o GPS que ajuda sua marca a localizar as melhores oportunidades*. Disponível em: <<https://endeavor.org.br/estrategia-e-gestao/beacon/>>. Acesso em: 21/07/2018.
- [10] CoAP. *Constrained Application Protocol (CoAP)*. Disponível em: <<http://coap.technology/>>. Acesso em: 21/07/2018.
- [11] AWS. *Pub/Sub Messaging Asynchronous event notifications*. Disponível em: <<https://aws.amazon.com/pt/pub-sub-messaging/>>. Acesso em: 21/07/2018.
- [12] Mozilla Developer Network. *WebSockets*. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API>. Acesso em: 21/07/2018.

- [13] TETSUYA, Y.; YUYA, S. Comparison with http and mqtt on required network resources for iot. 01 2017.
- [14] NITIN, N. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. 10 2017.
- [15] Eclipse. *Mosquitto*. Disponível em: <<https://mosquitto.org/>>. Acesso em: 21/07/2018.
- [16] Zigbee Alliance. *Zigbee Alliance*. Disponível em: <<https://www.zigbee.org/>>. Acesso em: 21/07/2018.
- [17] 2017, E. I. *The JSON Data Interchange Syntax*. Disponível em: <<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>>. Acesso em: 21/07/2018.
- [18] Espressif. *Hardware*. Disponível em: <<https://www.espressif.com/en/products/hardware>>. Acesso em: 21/07/2018.
- [19] Arduino. *What is Arduino ?* Disponível em: <<https://www.arduino.cc/en/Guide/Introduction>>. Acesso em: 21/07/2018.
- [20] cplusplus.com. *A Brief Description of C++*. Disponível em: <<http://www.cplusplus.com/info/description/>>. Acesso em: 21/07/2018.
- [21] ARM. *arm developer*. Disponível em: <<http://infocenter.arm.com/help/index.jsp>>. Acesso em: 21/07/2018.
- [22] Raspberry Pi Foundation. *Getting started with the Raspberry Pi*. Disponível em: <<https://projects.raspberrypi.org/en/projects/raspberry-pi-getting-started>>. Acesso em: 21/07/2018.
- [23] Node.js. *About Docs*. Disponível em: <<https://nodejs.org/en/docs/>>. Acesso em: 21/07/2018.
- [24] Digicert. *What is SSL, TLS and HTTPS?* Disponível em: <<https://www.websecurity.symantec.com/security-topics/what-is-ssl-tls-https>>. Acesso em: 21/07/2018.

- [25] Prithiviraj, Damodaran. *Art of choosing a datastore*. Disponível em: <<http://bytecontinnum.com/2016/02/choosingtorechoice/>>. Acesso em: 21/07/2018.
- [26] ANDREAS, K.; LEFTERIS, K.; DANI, M. <http://www.di.ufpb.br/lucidio/Btrees.pdf>. Disponível em: <B-trees>. Acesso em: 21/07/2018.
- [27] PATRICK, O. et al. The log-structured merge-tree (lsm-tree). Acta Informatica, 2009.
- [28] WIREDTIGER. *B-Trees vs LSM*. Disponível em: <<https://github.com/wiredtiger/wiredtiger/wiki/Btree-vs-LSM>>. Acesso em: 21/07/2018.
- [29] SHARVARI, R.; M., B. D. D. Mysql and nosql database comparison for iot application. 2016.
- [30] PORNPAT, P.; MIKIKO, S.; MITARO, N. Low-power distributed nosql database for lot middleware. 2016.
- [31] INC, M. *What is MongoDB ?* Disponível em: <<https://www.mongodb.com/what-is-mongodb>>. Acesso em: 21/07/2018.
- [32] 2018, P. *Plotly Community Feed*. Disponível em: <<https://plot.ly/#/>>. Acesso em: 21/07/2018.
- [33] JITTY, J.; KEERTHI, K. N.; AJITH, R. Analysis of temperature effect on mosfet parameter using matlab. IJEDR, v. 4, 2016.
- [34] CASAS, D. *Plantas de apartamentos com 2 quartos*. Disponível em: <<http://decorandocasas.com.br/2014/10/22/plantas-de-apartamentos-com-2-quartos/>>. Acesso em: 21/07/2018.
- [35] INTERNATIONAL, I. B. *Connecting devices using Wi-Fi Protected Setup™ (WPS) on your Linksys router*. Disponível em: <<https://www.linksys.com/us/support-article?articleNum=143300>>. Acesso em: 21/07/2018.

5 APÊNDICE

5.1 Guias de instalação

Atenção: Essas instruções são uma versão de quando este trabalho foi publicado, para obter a versão mais atualizada e outras versões, acesse <https://github.com/fol21>. Este guia tem como reproduzir os testes feitos no sistema, que foi testado em distribuições de Windows 10 e Linux.

5.1.1 Configurando Broker

1. Acesse <https://mosquitto.org/download/> e siga as instruções para baixar e instalar o Mosquitto;
2. No terminal execute o comando *mosquitto -d*
3. Use o IP e a porta configurada nos exemplos de Publisher e Subscriber;
4. Em C++ as configurações são encontradas no próprio código;
5. Em Javascript acesse o arquivo *config.json* no diretório *example/*/resources* e mude as configurações de IP e Porta;

5.1.2 Publishers em C++

1. Siga as instruções de instalação em <http://docs.platformio.org/en/latest/installation.html>;
2. Acesse <https://docs.espressif.com/projects/esp-idf/en/latest/get-started/establish-serial-connection.html> e instale os drivers adequados para o ESP32;
3. Acesse <https://github.com/fol21/things-4-labs-acquirer-platform> e baixe o projeto;
4. No diretório do projeto digite os comandos *pio run -t upload -e esp32*;

5.1.3 Publishers em Javascript

1. Acesse <https://nodejs.org/en/> e baixe sua versão do Node.js na versão 8 ou maior;
2. Acesse <https://github.com/fol21/things-4-labs-acquirer-raspberrypi> e baixe o projeto;
3. Entre no diretório `examples/gpio`, presente no projeto;
4. Coloque as informações de IP e Porta no arquivo `resources/config.json`;
5. Execute o comando como administrador `npm install`;
6. Execute o comando como administrador `node index.js`;

5.1.4 Subscribers em Javascript

1. Acesse <https://nodejs.org/en/> e baixe sua versão do Node.js na versão 8 ou maior;
2. Acesse <https://docs.mongodb.com/manual/installation/> e siga as instruções para instalar e iniciar o mongodb conforme seu Sistema Operacional;
3. Crie uma conta gratuita em <https://plot.ly/>;
4. Acesse <https://github.com/fol21/things-4-labs-console-subscriber> e baixe o projeto;
5. Entre no diretório `examples/simple-subscriber`, presente no projeto;
6. Coloque as informações de IP e Porta no arquivo `resources/config.json`;
7. Coloque seu Id e chave da sua conta do Plotly em `index.js`;
8. Execute o comando como administrador `npm install`;
9. Execute o comando como administrador `node index.js`;

5.2 Códigos Fonte

Atenção: Estes códigos são uma versão de quando este trabalho foi publicado, para obter a versão mais atualizada e outras versões, acesse <https://github.com/fol21>.

5.2.1 Publishers em C++

Códigos 5.1 Data Stream Header

```
#ifndef DATA_STREAM_H
#define DATA_STREAM_H

#include <Arduino.h>
#include <string.h>
#include <ArduinoJson.h>

#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))

#define CONTINUOUS_STREAM "continuous"
#define PERIODIC_STREAM "periodic"
#define CONTINUOUS_STREAM_STRING String("continuous")
#define PERIODIC_STREAM_STRING String("periodic")

class data_stream
{
public:

    int Threshold(){ return this->threshold; }
    char* Name(){ return this ->name; };

    //overridable stream process
    virtual void process() = 0;

    virtual void onMessage(char*, const char*, unsigned int) = 0;

    const char* send(const char*); // send message after process is done
```

```

    bool operator==(const data_stream& o )
    {
        return (strcmp(this->name, o.name) == 0) ? true:false;
    };

protected:

    char* name;
    int threshold = 0;
    char* payload = NULL;
    bool lock = false;
};

class continous_stream : public data_stream
{
public:
    continous_stream() {
        this->name = CONTINUOUS_STREAM;

    };
    continous_stream(int size)
    {
        this->name = CONTINUOUS_STREAM;
        this->threshold = size;
    };

    void process(){};

    void onMessage(char* topic, const char* payload,unsigned int
        length){};
};

class periodic_stream : public data_stream
{
public:
    periodic_stream()
    {
        this->name = PERIODIC_STREAM;
    }
};

```

```

    periodic_stream(int size)
    {
        this->name = PERIODIC_STREAM;
        this->threshold = size;
    }

    void onMessage(char* topic, const char* payload,unsigned int
        length)
    {
        StaticJsonBuffer<200> jsonBuffer;
        this->payload = (char*) payload;
        JsonObject& params = jsonBuffer.parseObject(this->payload);
        this->millis = params["millis"];
    }

    void process()
    {
        delay(this->millis);
    }
protected:
    int millis = 0;
};

#endif // DATA_STREAM_H

```

Códigos 5.2 Data Stream Source

```

#include <data_stream.h>

const char* data_stream::send(const char* message)
{
    if(payload != NULL && !(this->lock))
        this->process();

    if(this->threshold != 0)
    {
        if(ARRAY_SIZE(message) > this->threshold) return "Message_size_
            is_above_allowed!";
    }
}

```

```

        else return message;
    }
    else return message;
}

```

Códigos 5.3 MQTT Publisher Header em C++

```

#ifndef MQTTPUBLISHER_H
#define MQTTPUBLISHER_H

#define DEVICE_ID_PATTERN "/id:"
#define DEVICE_ID_PATTERN_STRING String("/id:")
#define STREAM_PATTERN "/stream:"
#define STREAM_PATTERN_STRING String("/stream:")

#include <string>
#include <list>
#include <algorithm>

#include <PubSubClient.h>
#include <data_stream.h>

struct MqttConfiguration
{
    const char* ssid;
    const char* password;
    const char* client_id;
    const char* host;
    unsigned int port;
};

enum publisher_state {INIT, NETWORK, BROKER, READY};

class MqttPublisher
{
public:

    MqttPublisher(Client&, MqttConfiguration& config);

```

```

MqttPublisher(Client&, const char*, const char*, unsigned int);
const char* publish_stream(const char*, const char*, const char*);
void check_network(bool (*)(void));
void init_network(bool (*)(void));
bool reconnect(void (*handler)(void));
bool broker_connected();
int Client_state();
int Publisher_state();

void onMessage(void (*)(char*, uint8_t*, unsigned int));

void add_stream(data_stream*);
void remove_stream(const char*);
data_stream* find_stream(const char*);
void middlewares(char*, uint8_t*, unsigned int);

PubSubClient* PubSub_Client(){return pubSubClient;}

protected:

const char* client_id;
const char* host = NULL;
unsigned int port = 0;
PubSubClient* pubSubClient;

void(*message_callback)(char*, uint8_t*, unsigned int);

bool (*has_network)(void);
bool (*network_start)(void);

continous_stream c_stream;
periodic_stream p_stream;
std::list<data_stream*> streamList;

publisher_state state = INIT;

};

```

```
#endif // !MQTTPUBLISHER_H
```

Códigos 5.4 MQTT Publisher Source em C++

```
#include <MqttPublisher.h>
```

```
struct is_name
```

```
{
    is_name(const char*& a_wanted) : wanted(a_wanted) {}
    const char* wanted;
    bool operator()(data_stream*& stream)
    {
        return strcmp(wanted, stream->Name()) == 0;
    }
};
```

```
const char* StringToCharArray(String str)
```

```
{
    return str.c_str();
}
```

```
MqttPublisher::MqttPublisher(Client& client, MqttConfiguration& config)
```

```
{
    this->client_id = config.client_id;
    this->host = config.host;
    this->port = config.port;
    this->pubSubClient = new PubSubClient(client);
    this->pubSubClient->setServer(this->host, this->port);

    this->c_stream = continous_stream();
    this->p_stream = periodic_stream();
}
```

```
MqttPublisher::MqttPublisher(Client& client, const char* client_id,
    const char* host,
    unsigned int port)
```

```

{
    this->client_id = client_id;
    this->host = host;
    this->port = port;
    this->pubSubClient = new PubSubClient(client);
    this->pubSubClient->setServer(this->host, this->port);

    this->c_stream = continous_stream();
    this->p_stream = periodic_stream();
}

void MqttPublisher::onMessage(void (*callback)(char*, uint8_t*, unsigned
    int))
{
    this->message_callback = callback;
    this->pubSubClient->setCallback(callback);
}

void MqttPublisher::middlewares(char* topic, uint8_t* payload, unsigned
    int length)
{
    if(strcmp(topic, (String(this->client_id) + STREAM_PATTERN_STRING+
        CONTINUOUS_STREAM_STRING).c_str()) == 0)
    {
        this->c_stream.onMessage(topic, (const char*) payload,
            length);
    }
    else if(strcmp(topic, (String(this->client_id) +
        STREAM_PATTERN_STRING+PERIODIC_STREAM_STRING).c_str()) == 0)
    {
        this->p_stream.onMessage(topic, (const char*) payload,
            length);
    }
    else
    {
        String str_topic = String(topic);
        int index = str_topic.indexOf(":") + 1;
        String s = str_topic.substring(index);
    }
}

```



```

        const char* c = s.c_str();
        this->find_stream(c)->onMessage(topic, (const char*) payload,
            length);
    }
}

void MqttPublisher::add_stream(data_stream* stream)
{
    this->streamList.push_back(stream);
}

void MqttPublisher::remove_stream(const char* stream_name)
{
    if(!this->streamList.empty()) this->streamList.remove_if(is_name(
        stream_name));
}

data_stream* MqttPublisher::find_stream(const char* stream_name)
{
    if(stream_name == CONTINUOUS_STREAM)
        return &(this->c_stream);
    else if(stream_name == PERIODIC_STREAM)
        return &(this->p_stream);
    if(!this->streamList.empty())
    {
        return *(std::find_if(this->streamList.begin(), this->streamList
            .end(), is_name(stream_name)));
    }
    else
        return &(this->c_stream);
}

const char* MqttPublisher::publish_stream(const char* topic, const char*
    stream_name, const char* message)
{
    if(this->state == READY)
    {

```

```

        if(strcmp(stream_name,CONTINUOUS_STREAM) == 0)
            this->pubSubClient->publish(topic, this->c_stream.send(
                message));
        else if(strcmp(stream_name,PERIODIC_STREAM) == 0)
            this->pubSubClient->publish(topic, this->p_stream.send(
                message));
        else
            this->pubSubClient->publish(topic, (char*) this->find_stream
                (stream_name)->send(message));

        return message;
    }
    else return "" + pubSubClient->state();
}

void MqttPublisher::check_network(bool (*check)(void))
{
    this->has_network = check;
}

void MqttPublisher::init_network(bool (*connectionHandler)(void))
{
    this->network_start = connectionHandler;
}

bool MqttPublisher::reconnect(void(*handler)(void))
{
    if(this->state == INIT)
    {
        if(this->network_start())
            this->state = NETWORK;
    }

    if(this->state == NETWORK)
    {
        if(this->pubSubClient->connect(this->client_id))
        {

```

```

        this->pubSubClient->subscribe(StringToCharArray(String(this
            ->client_id) + STREAM_PATTERN_STRING +
                CONTINUOUS_STREAM_STRING));

        this->pubSubClient->subscribe(StringToCharArray(String(this
            ->client_id) + STREAM_PATTERN_STRING +
                PERIODIC_STREAM_STRING));

        if(!this->streamList.empty())
        {
            for (std::list<data_stream*>::iterator it=this->
                streamList.begin();
                it!=this->streamList.end(); ++it)
            {
                this->pubSubClient->subscribe((String(this->
                    client_id) +
                        STREAM_PATTERN_STRING + String
                            ((*it)->Name()).c_str()));
            }
        }

        this->state = BROKER;
    }
}

if(this->state == BROKER)
{
    if(this->broker_connected())
        this->state = READY;
    else
    {
        if(this->has_network())
            this->state = NETWORK;
        else
            this->state = INIT;
    }
}

if(this->state == READY)
{

```

```

        if(!this->has_network())
            this->state = INIT;
        if(!this->broker_connected())
            this->state = NETWORK;
    }
    delay(100);
    this->pubSubClient->loop();
    (*handler)();
}

bool MqttPublisher::broker_connected()
{
    return this->pubSubClient->connected();
}

int MqttPublisher::Client_state()
{
    return this->pubSubClient->state();
}

int MqttPublisher::Publisher_state()
{
    return this->state;
}

```

5.2.2 Publishers em Javascript

Códigos 5.5 Data Stream em javascript

```

class DataStream {

    /**
     * Creates an instance of DataStream.
     * @param {string} name
     * @memberof DataStream
     */
    constructor(name) {
        this.name = name;
        this.threshold = 0;
    }
}

```

```

/**
 *
 * @type {object}
 * @param {Object.<string, any>} [configuration=null]
 * @memberof DataStream
 */
onMessage(configuration = null) {
}

/**
 *
 *
 * @param {string} message
 * @param {streamProcess} [process=null]
 * @returns {string}
 * @memberof DataStream
 */
validate(message) {

    if (this.threshold !== 0) {
        if (message.length > this.threshold) return "Message_size_is
            _above_allowed!";
        else return message;
    } else return message;
}

/**
 * Async method to be implemented by a child of DataStream
 *
 * @memberof DataStream
 */
async send(message) {
    console.log("Please_Implement_an_async_send_method_in_your_child
        _of_DataStream")
    return message;
}

```

```

}

module.exports = DataStream

```

Códigos 5.6 Continous Stream

```

const DataStream = require('./DataStream');

const continuousStream = new DataStream('continuous');

continuousStream.onMessage();

continuousStream.send = async function(message) {
  try {
    return continuousStream.validate(message);
  } catch (error) {
    console.log(error);
  }
}

module.exports = continuousStream;

```

Códigos 5.7 Periodic Stream em Javascript

```

const DataStream = require('./DataStream');

class PeriodicStream extends DataStream {

  constructor(delay = 100) {
    super('periodic')
    // Sets default delay to 100ms
    this.configuration = {delay};
  }

  /**
   * Sets the threshold
   *
   * @param {number} threshold
   * @memberof PeriodicStream
   */
}

```

```

Threshold(threshold) {
    super.threshold = threshold;
}

/**
 * Set the Stream delay for delivering messages
 *
 * @param {number} delay
 * @memberof PeriodicStream
 */
Delay(delay)
{
    this.configuration.delay = delay;
}

/**
 * Sets timeout delay
 *
 * @param {number} delay
 * @memberof PeriodicStream
 */
onMessage(configuration) {
    this.configuration = configuration;
}

/**
 * Sends async message in timed delays
 *
 * @param {string} message
 * @memberof PeriodicStream
 */
async send(message) {
    let self = this;
    let sender = new Promise((resolve) =>
    {
        setTimeout(() =>
        {
            resolve(self.validate(message));
        }, self.configuration.delay)
    }
    );
}

```

```

    });
    try {
        let message = await sender;
        return message;
    } catch (error) {
        console.log(error);
    }
}
}

module.exports = PeriodicStream;

```

Códigos 5.8 MQTT Publisher Source em Javascript

```

const mqtt = require('mqtt');
const program = require('commander');
const _ = require('lodash/array');

const {
    DataStream,
    ContinousStream,
    PeriodicStream
} = require('./data-stream/index')
/**
 * A MQTT Based Publisher with Data Stream transactions avaiable
 *
 * @class MqttPublisher
 */
class MqttPublisher {
    constructor(config = {}) {

        this.program = program
            .version('0.1.0')
            .option('-t, --topic <n>', 'Choose topic to be subscribed')
            .option('-h, --host <n>', 'Overrides pre-configure host')
            .option('-p, --port <n>', 'Overrides pre-configure port',
                parseInt)

        this.host = program.host || config.host;
        this.port = parseInt(program.port || config.port);
    }
}

```



```

        if(config.topic)
        {
            this.topic = config.topic;
        }
        else this.topic = null;
    }

    /**
     *
     * Publishes a message by a Data Stream
     *
     * @param {string} topic
     * @param {string} message
     * @param {string} [streamName='continous']
     * @memberof MqttPublisher
     */
    publish(topic, message, streamName = 'continous') {
        try {
            if (this.client.connected) {

                let stream = this.findDataStream(streamName);
                if(stream)
                {
                    stream.send(message).then((message) =>
                    {
                        this.client.publish(topic,message);
                    })
                } else console.log("Stream_not_found_in_stream_list")
            } else console.log('Unable_to_publish.No_connection_avaiable
                .')
        } catch (err) {
            console.log(err);
        }
    }

    /**
     *
     *

```

```

    * @param {DataStream} stream
    * @memberof MqttPublisher
    */
    addDataStream(stream) {
        this.streams.push(stream);
        this.client.subscribe(`${this.id}/stream:${stream.name}`);
    }

    /**
     *
     *
     * @param {string} streamName
     * @memberof MqttPublisher
     */
    removeDataStream(streamName) {
        _.remove(this.streams, (s) => {
            return s.name == streamName;
        });
    }

    /**
     * Finds an implementation of DataStream in the streams list
     * Returns null if not listed
     *
     * @param {string} streamName
     * @returns {DataStream}
     * @memberof MqttPublisher
     */
    findDataStream(streamName) {
        let index = _.findIndex(this.streams, (s) => {
            return s.name == streamName;
        });
        if(index || index == 0 )
            return this.streams[index];
        else return null;
    }

    /**
     *
     *
     * Starts a publisher with an id, with Continuous and Periodic
     * Streams by default

```

```

*
* @param {Function} callback callback type of (void) : void
* @param {string} [id]
* @memberof MqttPublisher
*/
init(id) {
  this.id = id;
  this.program.parse(process.argv);

  this.streams = [];

  if (program.topic && !this.topic) {
    this.topic = program.topic;
  }
  else if (!program.topic && !this.topic){
    console.log("Topic is required (run program with -t <topic> flag)")
    process.exit();
  }

  this.client = mqtt.connect({
    host: this.host,
    port: this.port
  });

  this.client.on('message', (topic, message) => {
    if (topic.match(/(\w+)\//configure\//stream:(\w+)/g))
    {
      message = message.toString()
      console.log('Received Configuration ${message}');
      let streamName = topic.match(/(?!stream:)\w+$/g);
      let stream = this.findDataStream(streamName[0])
      if(stream)
      {
        console.log('Sent configurations to ${streamName} Data')
        stream.onMessage(JSON.parse(message));
      }
    }
  })
}

```

```

        //console.log(topic + ' : ' + message)
    });

    return new Promise(resolve =>
    {
        this.client.on('connect',
        () => {
            console.log('Connected, Listening to:
            host: ${this.host}
            port: ${this.port}');

            this.addDataStream(ContinuousStream);
            this.addDataStream(new PeriodicStream());

            resolve(this.client.connected);
        });
    });

}
}

module.exports = MqttPublisher;

```

Códigos 5.9 Index das implementações

```
module.exports = require("./src/MqttPublisher");
```

5.2.3 Subscribers em Javascript

Códigos 5.10 MQTT Subscriber Source em Javascript

```

const readline = require('readline');
const mqtt = require('mqtt');
const program = require('commander');

/**
 *
 *
 * @class MqttSubscriber

```

```

*/
class MqttSubscriber {

    constructor(config = {}) {

        this.program = program
        .version('0.1.0')
        .option('-t, --topic <n>', 'Choose topic to be subscribed',
            (val) => {
                return val
            })
        .option('-c, --context <n>', 'Add context to incoming
            messages')
        .option('-h, --host <n>', 'Overrides pre-configure host')
        .option('-p, --port <n>', 'Overrides pre-configure port',
            parseInt)
        .option('-C, --configure <items>', 'Name of configuration
            topic and json', (val) => {
                return val.split(',');
            })
        .parse(process.argv);

        this.host = program.host || config.host;
        this.port = program.port || config.port;
        this.configure = (config.configure) ? config.configure : null;

        this.messageCallback = null;

        this.topic = null;
        if (program.topic || config.topic || program.configure) {
            if (program.topic) this.topic = program.topic;
            else if (config.topic) this.topic = config.topic;
            if (program.configure) this.configure = program.configure;
        } else {
            console.log("Topic or Configuration is required (run program
                with -t <topic> or -C <configuration> flag)")
            process.exit();
        }
    }
}

```

```

}

_defaultMessageCallback(topic, message) {
    // message is Buffer
    if (program.context) console.log(`${program.context} ${message
        }`);
    else console.log(message.toString());
}

/**
 *
 *
 * @param {function (topic,message)} callback
 * @memberof MqttSubscriber
 */
onMessage(callback) {
    this.messageCallback = callback;
}

//Parses Configuration
_parseConfigure() {
    let body = null;
    if (this.configure.constructor === Array) {
        body = {
            topic: this.configure[0],
            json: this.configure[1],
            configuration: JSON.parse(this.configure[1])
        }
    } else {
        body = {
            topic: this.configure.topic,
            json: this.configure.json,
            configuration: JSON.parse(this.configure.json)
        }
    }
    return body
}

```

```

// /**
//  *
//  *
//  * @param {string} topic
//  * @param {string} json
//  * @memberof MqttSubscriber
//  */
// sendConfiguration(topic, json) {
//     this.client.end(
//         () => console.log("Reconfiguring Stream...")
//     );
//     this.configure = {
//         topic: topic,
//         json: json,
//         configuration: JSON.parse(json)
//     }
// }
// }

/**
 *
 * Starts the transaction of the Data Stream defining the
 * configurations
 *
 * @param {*} reconnect
 * @param {*} callback
 */
_connectionStarter(reconnect, callback)
{
    return () => {
        if (this.configure) {
            let conf ;

            if(reconnect) conf = this.configure;
            else conf = this._parseConfigure()
            if (conf.topic.match(/(\w+)\.configure\/stream:(\w+)/g))
            {

```

```

        this.client.publish(conf.topic, conf.json);
        console.log('Sent Configuration ${conf.json} to ${
            conf.topic}')
    }
}

if (this.topic) {
    console.log('Connected, Listening to:
host: ${this.host}
port: ${this.port}
topic: ${this.topic}');
    this.client.subscribe(this.topic);
    if(callback) callback()
} else {
    process.exit();
}
}

_clientInit()
{
    this.client = mqtt.connect({
        host: this.host,
        port: this.port
    });
}

/**
 * Initialize console application
 * Use after setup every callback
 * Must be called only once
 * @memberofof MqttSubscriber
 */
init(callback = null, reconnect = false) {

    //program.parse(process.argv);
    this._clientInit();

    this.client.on('connect', this._connectionStarter(reconnect,
        callback));

```



```

        //this.client.on('reconnect', this._connectionStarter)
        this.client.on('message', this.messageCallback || this._defaultMessageCallback)
    }

}

module.exports = MqttSubscriber;

```

Códigos 5.11 Index das implementações

```

module.exports = {
    MqttSubscriber : require('./src/MqttSubscriber'),
    MongoClient : require('./src/db/MongoDataClient')
};

```

Códigos 5.12 Data Client para MongoDB

```

const mongo = require('mongodb');
const MongoClient = require('mongodb').MongoClient;

module.exports = class MongoDataClient {

    /**
     * Creates an instance of MongoDataClient.
     * @param {string} url
     * @param {string} [database=null]
     */
    constructor(url, database = null) {
        this.url = url;
        this.database = database;
        this.SINGLE_DATA_PACKAGE_COLLECTION = "DataPackage"
        //this._start(this.url);
    }

    /**
     *
     *
     * @returns {Promise<MongoDataClient>}
     * Create DataStore
     */
}

```

```

    */
    async start() {
        let self = this;
        const client = await MongoClient.connect(self.url)

        let dbo = client.db(self.database || 't4ldb');
        let collection = await dbo.createCollection(self.
            SINGLE_DATA_PACKAGE_COLLECTION);

        client.close();
        return this;
    }

    /**
     *
     * @param {any} values
     * @returns {Promise<MongoDataClient>}}
     * Insert Single Data Packet
     */
    async insertOne(value) {
        let self = this;
        let client = await MongoClient.connect(this.url);
        let dbo = client.db(self.database || 't4ldb');
        let myobj = {
            timestamp: Date.now(),
            data: value
        };
        let collection = await dbo.collection(self.
            SINGLE_DATA_PACKAGE_COLLECTION).insertOne(myobj)
        client.close();

        return this;
    }

    /**
     *
     * @param {any[]} values
     * @returns {Promise<MongoDataClient>}}

```

```

* Insert Collection of Single Data Packet
*/
async insertMany(values) {
  let self = this;
  let client = await MongoClient.connect(self.url)
  let dbo = client.db(self.database || 't4ldb');

  let chunks = [];
  values.forEach(element => {
    chunks.push({
      timestamp: Date.now(),
      data: element
    });
  });
  let collection = await dbo.collection(self.
    SINGLE_DATA_PACKAGE_COLLECTION).insertMany(chunks)
  client.close();
  return this;
}

/**
 *
 * @param {number} quantity
 * @returns {any[]}
 *
 * Returns Promise with the latests Data Documents
 */
async collectData(quantity) {
  let chunk = [];
  let self = this;
  let client = await MongoClient.connect(self.url)
  let dbo = client.db(self.database || 't4ldb');

  let res = await dbo.collection(self.
    SINGLE_DATA_PACKAGE_COLLECTION)
    .find({})
    .toArray()
  for (let i = 0; i < quantity; i++) {

```

```

        chunk.push(res[i]);
    }
    client.close();
    return chunk;
}

/**
 *
 * @param {number} quantity
 * @returns {Promise<MongoDataClient>}
 * Delete latests ocurrence top to botton of Collection
 */
async deleteLatests(quantity) {
    let self = this;
    try {
        const client = await MongoClient.connect(self.url);
        let dbo = client.db(self.database || 't4ldb');

        let res = await dbo.collection(self.
            SINGLE_DATA_PACKAGE_COLLECTION)
            .find({})
            .toArray()

        let collection;
        for (let i = 0; i < quantity; i++) {
            collection = await dbo.collection(self.
                SINGLE_DATA_PACKAGE_COLLECTION).deleteOne(res[i])
        }
        client.close();
    } catch (err) {
        console.error(err);
    }
    return this;
}

// Create Custom Data Collection

// Insert Custom Data Packet

```

```
}
```

5.2.4 Códigos fonte das aplicações em consoles

Códigos 5.13 Teste do publisher

```
const MqttPublisher = require('t4l-raspberrypi-publisher')
const si = require('systeminformation');
const moment = require('moment');;
const conf = require('./resources/config.json');

const client = new MqttPublisher({
  host: conf.mqtt.host,
  port: conf.mqtt.port,
  topic: conf.mqtt.ds_topic
});

let duration = moment().add(1, 'm').diff(moment());

client.init('001').then(() => {
  client.findDataStream('periodic').Delay(1000);
  collectSysInfo(client, 5000, duration);
});

async function sleep(millis)
{
  let timeout = new Promise(resolve => setTimeout(resolve, millis));
  try {
    return await timeout;
  } catch (error) {
    console.log(error);
  }
}

async function collectSysInfo(client, millis, steps) {
  try {
    for (let index = 0; index < steps ; index++) {
```

```

        await sleep(millis).then(()=>
        {
            console.log(index);
        });
        client.publish('/001/stream:periodic',
            JSON.stringify(await si.cpuTemperature()),
            'periodic');
    }
} catch (error) {
    console.log(error);
}
}

```

Códigos 5.14 Teste do subscriber

```

const moment = require('moment');
const _ = require('lodash/collection');
const plotly = require('plotly')('fol21', 'y32TAWwBymoF0nT0vkvE');
const {
    MqttSubscriber,
    MongoDataClient
} = require('t4l-console-susbcriber');

const conf = require('./resources/config.json');
const monitor = new MqttSubscriber({
    host: conf.mqtt.host,
    port: conf.mqtt.port,
    topic: conf.mqtt.ds_topic
});

let dbName = 't4l_test2'
let url = 'mongodb://localhost:27017/' + dbName;
const client = new MongoDataClient(url, dbName);

let count = 0;
let graphIndex = 1;

monitor.onMessage((topic, message) => {
    console.log(message.toString())
    message = JSON.parse(message.toString());

```

```

client.start().then((client) => {
  client.insertOne(message);
  if (count >= 100) {
    count = 0;
    client.collectData(100).then((chunk) => {
      let timeseries = _.map(_.map(chunk, 'timestamp'), (ts)
        => {
          return moment(ts).format("YYYY-MM-DD□HH:mm:ss");
        });
      let temps = _.map(chunk, 'data.main');

      var data = [{
        x: timeseries,
        y: temps,
        type: "scatter"
      }];

      var graphOptions = {
        filename: "cpu-temp_" + graphIndex,
        fileopt: "overwrite"
      };
      plotly.plot(data, graphOptions, function (err, msg) {
        if(err)
          console.log(err);
        else
        {
          console.log(msg);
          graphIndex++;
        }
      });
    });
  }
})
count++;
})

monitor.init();

// const plotly = require('plotly')('fol21', 'y32TAWwBymoF0nT0vkvE');

```

```
// var data = [
//   {
//     x: ["2013-10-04 22:23:00", "2013-11-04 22:23:00", "2013-12-04
//       22:23:00"],
//     y: [1, 3, 6],
//     type: "scatter"
//   }
// ];
// var graphOptions = {filename: "date-axes", fileopt: "overwrite"};
// plotly.plot(data, graphOptions, function (err, msg) {
//   console.log(msg);
// });
```

5.2.5 Códigos fonte das aplicações em plataformas embarcadas

Códigos 5.15 Teste do publisher no ESP32

```
#include <MqttPublisher.h>
#include <WiFi.h>
#include <ArduinoJson.h>

#ifdef __cplusplus
extern "C" {
#endif
uint8_t temprature_sens_read();
#ifdef __cplusplus
}
#endif
uint8_t temprature_sens_read();

WiFiClient espClient;
//struct MqttConfiguration config = {"Prosaico01", "semfiopros@ico01",
//  "001", "152.92.155.5", 1883};
struct MqttConfiguration config = {"LUFER", "21061992", "001", "
  192.168.15.7", 1883};
MqttPublisher publisher = MqttPublisher(espClient, config);

void callback( char* topic, uint8_t* payload, unsigned int length) {
```



```

    Serial.print("Message_ arrived_["");
    Serial.print(topic);
    Serial.print("]_");
    Serial.println((char*) payload);
}

class const_stream : public data_stream
{
public:
    const char* constant = "";
    const_stream(const char* consta)
    {
        this->name = "constant";
        this->payload = "Hallo!";
        this-> constant = consta;
        Serial.println(this->constant);
    }

    void onMessage(char* topic, const char* payload,unsigned int length
        )
    {
        StaticJsonBuffer<200> jsonBuffer;
        this->payload = (char*) payload;
        JsonObject& params = jsonBuffer.parseObject(this->payload);
        this->constant = params["constant"];
    }

    void process()
    {
        Serial.println(this->constant);
    }
};

const_stream* cs;

void setup()
{
    Serial.begin(115200);
    delay(3000);

```

```

publisher.onMessage(
  [](char* topic, uint8_t* payload, unsigned int length)
  {
    publisher.middlewares(topic, payload, length);
    callback(topic, payload, length);
  });
cs = new const_stream("initial");
Serial.println(cs->constant);
publisher.add_stream(cs);

publisher.check_network(
  []() -> bool
  {
    if(WiFi.status() == WL_CONNECTED)
      return true;
    else
      return false;
  });

publisher.init_network(
  []() -> bool
  {
    delay(10);
    // We start by connecting to a WiFi network
    Serial.println();
    Serial.print("Connecting_ to_");
    Serial.println(config.ssid);

    WiFi.begin(config.ssid, config.password);

    while (WiFi.status() != WL_CONNECTED) {
      delay(500);
      Serial.print(".");
    }
    Serial.println("");
    Serial.println("[Network]_ :_ Connected");
    return true;
  });

```

```

}

//temporarily holds data from vals
char charVal[10];
String json;
bool lock = true;
void loop()
{

    publisher.reconnect(
        [=]()
        {
            if(lock){
                lock = false;
                publisher.publish_stream("/001/configure/stream:periodic", "
                    continous", "{\"millis\":1000}");
            }
            Serial.println("Publisher_state:_" + String(publisher.
                Publisher_state()));
        });

    //4 is mininum width, 3 is precision; float value is copied onto buff
    dtostrf((temprature_sens_read() - 32) / 1.8, 4, 3, charVal);
    json = "{\"device\":\"esp32\",\"temperature\":" + String(charVal) + "}
        ";
    Serial.println(json);
    publisher.publish_stream("/001/stream:periodic", "periodic", json.c_str
        ());
}

// #ifdef __cplusplus
// extern "C" {
// #endif
// uint8_t temprature_sens_read();
// #ifdef __cplusplus
// }

```

```

// #endif
// uint8_t temprature_sens_read();

// void setup() {
//   Serial.begin(115200);
// }

// void loop() {
//   Serial.print("Temperature: ");

//   // Convert raw temperature in F to Celsius degrees
//   Serial.print((temprature_sens_read() - 32) / 1.8);
//   Serial.println(" C");
//   delay(5000);
// }

// #include <ESP8266WiFi.h>
// #include <PubSubClient.h>

// // Update these with values suitable for your network.

// const char* ssid = "FOL";
// const char* password = "21061992";
// const char* mqtt_server = "192.168.15.5";

// WiFiClient espClient;
// PubSubClient client(espClient);
// long lastMsg = 0;
// char msg[50];
// int value = 0;

// void setup_wifi() {

//   delay(10);
//   // We start by connecting to a WiFi network
//   Serial.println();

```

```

// Serial.print("Connecting to ");
// Serial.println(ssid);

// WiFi.begin(ssid, password);

// while (WiFi.status() != WL_CONNECTED) {
//     delay(500);
//     Serial.print(".");
// }

// randomSeed(micros());

// Serial.println("");
// Serial.println("WiFi connected");
// Serial.println("IP address: ");
// Serial.println(WiFi.localIP());
// }

// void callback(char* topic, byte* payload, unsigned int length) {
//     Serial.print("Message arrived [");
//     Serial.print(topic);
//     Serial.print("] ");
//     for (int i = 0; i < length; i++) {
//         Serial.print((char)payload[i]);
//     }
//     Serial.println();

//     // Switch on the LED if an 1 was received as first character
//     if ((char)payload[0] == '1') {
//         digitalWrite(BUILTIN_LED, LOW); // Turn the LED on (Note that
//         LOW is the voltage level
//         // but actually the LED is on; this is because
//         // it is active low on the ESP-01)
//     } else {
//         digitalWrite(BUILTIN_LED, HIGH); // Turn the LED off by making
//         the voltage HIGH
//     }
// }

```

```

// void reconnect() {
//   // Loop until we're reconnected
//   while (!client.connected()) {
//     Serial.print("Attempting MQTT connection...");
//     // Create a random client ID
//     String clientId = "ESP8266Client-";
//     clientId += String(random(0xffff), HEX);
//     // Attempt to connect
//     if (client.connect(clientId.c_str())) {
//       Serial.println("connected");
//       // Once connected, publish an announcement...
//       client.publish("outTopic", "hello world");
//       // ... and resubscribe
//       client.subscribe("inTopic");
//     } else {
//       Serial.print("failed, rc=");
//       Serial.print(client.state());
//       Serial.println(" try again in 5 seconds");
//       // Wait 5 seconds before retrying
//       delay(5000);
//     }
//   }
// }

// void setup() {
//   pinMode(BUILTIN_LED, OUTPUT);      // Initialize the BUILTIN_LED pin
//   as an output
//   Serial.begin(115200);
//   setup_wifi();
//   client.setServer(mqtt_server, 1883);
//   client.setCallback(callback);
// }

// void loop() {

//   if (!client.connected()) {
//     reconnect();
//   }

```

```
//  client.loop();

//  long now = millis();
//  if (now - lastMsg > 2000) {
//      lastMsg = now;
//      ++value;
//      snprintf (msg, 75, "hello world #%ld", value);
//      Serial.print("Publish message: ");
//      Serial.println(msg);
//      client.publish("outTopic", msg);
//  }
// }
```