



Universidade do Estado do Rio de Janeiro

Centro de Tecnologia e Ciências

Faculdade de Engenharia

Fernando de Oliveira Lima

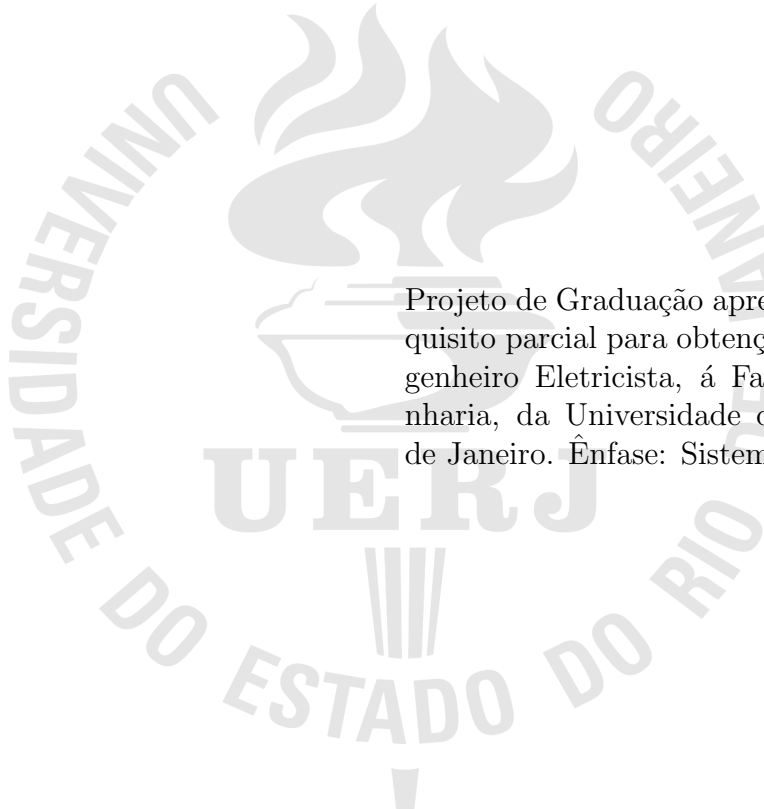
**Sistema Escalável para Aplicações de Internet das Coisas
utilizando MQTT**

Rio de Janeiro

2018

Fernando de Oliveira Lima

Sistema Escalável para Aplicações de Internet das Coisas utilizando MQTT



Projeto de Graduação apresentado, como requisito parcial para obtenção do grau de Engenheiro Eletricista, à Faculdade de Engenharia, da Universidade do Estado do Rio de Janeiro. Ênfase: Sistemas Eletrônicos.

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

Rio de Janeiro

2018

CATALOGAÇÃO NA FONTE
UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

L732 de Oliveira Lima, Fernando
Sistema Escalável para Aplicações de Internet das Coisas utilizando MQTT / Fernando de Oliveira Lima– 2018.
81 f.

Orientador: Michel Pompeu Tcheou.
Orientador: Lisandro Lovisolo.
Projeto de Graduação apresentado à Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia, para obtenção do grau de bacharel em Engenharia Elétrica.
Bibliografia: p.43

1. Internet das Coisas. 2. MQTT. 3. Indústria. I. Tcheou, Michel Pompeu. II. Universidade do Estado do Rio de Janeiro. Faculdade de Engenharia. III. Título.

CDU 621.3

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

Projeto de Graduação apresentado, como requisito parcial para obtenção do grau de Engenheiro Eletricista, à Faculdade de Engenharia, da Universidade do Estado do Rio de Janeiro. Ênfase: Sistemas Eletrônicos.

Aprovado em: 08 de Outubro de 2018

Banca Examinadora:

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

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

Prof. Dr. Téo Cerqueira Revoredo
Departamento de Eletrônica e Telecomunicações da UERJ

Rio de Janeiro

2018

DEDICATÓRIA

A todos que tiveram a paciência para ler esse pequeno texto...

Dedico este trabalho, em primeiro lugar, meus pais e familiares, que vencerem muitas batalhas antes dessa, para que eu esteja aqui, feliz, saudável e com uma condição privilegiada em relação a muitos concluindo mais uma etapa de minha vida. Me ensinaram valores inestimáveis como respeito, educação, compaixão, coragem, determinação entre outros valores não menos importantes. Espero um dia fazer o mesmo por alguém.

Dedico também a todos aqueles que de alguma forma me apoiaram e me ajudaram, seja pelo gesto mais simples, vocês me inspiram, me motivam e sem dúvidas fazem parte de cada palavra deste trabalho.

Não é fácil encontrar sua vocação, o seu lugar. Tive a sorte de definir o que seria de minha vida relativamente cedo e não me arrependo. Mas saiba que existem pessoas que estão tentando de alguma forma se encontrar (ou re-encontrar), seja por problemas pessoais, decepções ou frustrações no passado. Para essas pessoas, fica aqui minha dedicatória, tenha resiliência, sempre vale a pena continuar, uma hora a gente se encontra e logo em seguida já quer outra coisa, faz parte, a insatisfação nos move para frente. Nenhuma decepção é definitiva, tem jeito pra tudo, menos pra morte. Levante a cabeça, ninguém disse que seria fácil, não é ?

AGRADECIMENTO

Agradeço aos amigos e aos não tão amigos, que igualmente fizeram parte da minha evolução como ser humano e cidadão. Todas as pessoas que passaram na minha vida contribuíram um pouco para esse momento, não há como saber onde todas estão, mas se um dia lerem esse texto, quero que saiba que eu agradeço a companhia e o aprendizado.

Não poderia ficar de fora, é claro, todos aqueles que de alguma forma foram meus mentores. Meus professores ao longo da vida, em especial os dois orientadores desse projeto, já os conheço há quase sete anos, fizeram parte da minha formação, me deram oportunidades e ajudaram a concretizar esse trabalho. Agradeço de coração, que mais oportunidades de nos encontrar surjam, e desejo todo o sucesso possível e uma vida feliz e confortável.

Fica meu agradecimento aqui também para essa instituição linda e maravilhosa, que não é fácil de lidar, muitas vezes pareceu que ia nos abandonar, mas trouxemos ela de volta, cada um com sua parte. Obrigado UERJ, por uma parte inesquecível da minha vida e por todas as pessoas que conheci dentro deste universidade.

E se você estiver lendo esse texto, sim, você mesmo leitor! Agradeço seu prestígio, mesmo se você tiver ignorado completamente esta parte. Espero que tenha aprendido um pouco. O texto é para Ciência, para a Tecnologia e para você.

RESUMO

LIMA, Fernando *Sistema Escalável para Aplicações de Internet das Coisas utilizando MQTT*. 81 f. Trabalho de Conclusão de Curso (Engenharia Elétrica ênfase em 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 sistemas de comunicação entre máquinas e sistemas de compartilhamento e visualização de dados sobre dispositivos, seja numa fábrica ou em residências. Este trabalho apresenta um sistema para aplicações de internet das coisas (IoT) utilizando MQTT, um protocolo de aplicação para comunicação entre dispositivos que enviam dados telemétrico, com persistência de dados em banco MongoDB. O sistema engloba todos os setores da aquisição de dados a camada de aplicação, com o objetivo de facilitar a implementação de aplicações eficientes em cada cenário.

O sistema também contempla Data Streams, canais de dados configurável, permitindo definir parâmetros de como os dados serão transmitidos. Com os Data Streams, é possível que o sistema se adapte dinamicamente a cenários onde existem dificuldades na transmissão de dados, como problemas na estabilidade de conexão, congestionamento na rede, entre outros.

O projeto também contempla um caso de uso do sistema, onde são feitas medições de temperaturas em CPUs e na unidade micro-controlada ESP32, realizando três medições. A primeira medição é feita em uma CPU isolada, avaliando seu comportamento, a segunda, um comparativo entre as medições de uma CPU da Intel e o ESP32 e por último, o comparativo das medições de temperatura das CPUs de múltiplos computadores do laboratório PROSAICO.

Palavras-Chave: IoT, MQTT, Data Streams .

ABSTRACT

In the verge of the data revolution, a growing interest in communication systems between machines and the sharing systems of telemetric data on devices rises, whether in a factory or in a residence. This work presents a system for Internet applications of things (IoT) using MQTT, an application protocol for communication between devices that shares telemetric data, with data persistence using MongoDB. The system will encompass all sectors of data acquisition to the application layer, facilitating implementations of applications in each scenario.

The system also contemplates Data Streams, a configurable data channel, allowing you to define parameters how data will be transmitted. With Data Streams, it is possible for the system to dynamically adapt to scenarios where difficulties in data transmission are found, such as problems in connection stability, network congestion, and so on.

The project also contemplates a use case of the system, where temperature measurements are made in CPUs and in the micro-controlled unit ESP32, making three measurements. The first one being the measurement of an isolated CPU, evaluating its behavior, the second, a comparative between measurements of an Intel CPU and the ESP32 and finally, the comparative of Multiple computers CPUs temperatures in PRO-SAICO laboratory.

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, cada um exercendo uma função em uma aplicação IoT	15
Figura 3	Interface de comunicação. A interface tem sua arquitetura que pode se comunicar com um ou mais protocolos de aplicação	16
Figura 4	As três camadas do IoT, dos sensores ao mundo real.....	17
Figura 5	Fluxo de conexão do HTTP	20
Figura 6	Fluxo da conexão do MQTT	21
Figura 7	O padrão Publish/Subscribe. Retirado de [1]	22
Figura 8	Exemplo de gerenciamento de um broker.....	22
Figura 9	A arquitetura de um banco de dados	24
Figura 10	Comparação de Escrita e Leitura entre LSM e B+, retirado de [2]	25
Figura 11	O conceito de Data Stream para a abstração do transporte de dados.....	28
Figura 12	Um Data Stream é criado a partir do tópico <i>/003/stream:periodic</i>	29
Figura 13	Uma outra representação da comunicação entre Publishers e Subscribers por Data Stream	29
Figura 14	A arquitetura do ESP32, retirado de [3]	30
Figura 15	A arquitetura simplificada de dispositivos com Sistema Operacional.....	31
Figura 16	Diagrama simplificado de uma rotina padrão seguida pela implementação em Microcontroladores	32
Figura 17	Diagrama simplificado de uma rotina assíncrona padrão em Consoles	34
Figura 18	O formato de documento no MongoDB.....	35
Figura 19	Adição de parâmetro de timestamp em milisegundos ao documento	35
Figura 20	Diagrama de fluxos do Publisher e do Subscriber.....	37
Figura 21	Visualização dos dados armazenados em documento em uma coleção do MongoDB	38
Figura 22	Comportamento da temperatura em três momentos.....	38
Figura 23	Comparação das temperaturas com uma CPU core i7 e ESP32	39
Figura 24	Comparação das temperaturas em múltiplos processadores.	40

LISTA DE TABELAS

Tabela 1	Comparativo MQTT X HTTP	21
----------	-------------------------------	----

Lista de Códigos

5.1	Data Stream Header	48
5.2	Data Stream Source	50
5.3	MQTT Publisher Header em C++	50
5.4	MQTT Publisher Source em C++	52
5.5	Data Stream em javascript	58
5.6	Continous Stream	59
5.7	Periodic Stream em Javascript	60
5.8	MQTT Publisher Source em Javascript	62
5.9	Index das implementações	66
5.10	MQTT Subscriber Source em Javascript	66
5.11	Index das implementações	71
5.12	Data Client para MongoDB	71
5.13	Teste do publisher	75
5.14	Teste do subscriber	76
5.15	Teste do publisher no ESP32	77

LISTA DE SIGLAS

IoT	Internet das Coisas
MQTT	Message Queuing Telemetry Transport
TCP	Transmission Control Protocol
IP	Internet Protocol
HTTP	Hyper Text Transfer Protocol
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
CPU	Central Processing Unity

SUMÁRIO

1	MOTIVAÇÃO E VISÃO GERAL DO TRABALHO	13
1.1	Internet das Coisas.....	13
1.2	Visão geral de uma aplicação IoT.....	14
1.3	Interface para Protocolos de Aplicação.....	15
2	CONCEITOS GERAIS DE IOT.....	17
2.1	As Camadas da IoT.....	17
2.2	Tecnologias em IoT	18
2.3	MQTT	19
2.3.1	MQTT X HTTP	20
2.3.2	Publishers e Subscribers	20
2.3.3	Broker.....	21
2.3.4	Tipos de MQTT	22
2.4	Persistência de dados	23
2.5	Bancos para Aplicações IoT	24
3	O PROJETO	27
3.1	Data Streams.....	27
3.2	Implementação em Plataformas.....	29
3.2.1	Sistemas Embarcados	30
3.2.2	Consoles	31
3.3	Arquiteturas e Assíncronismo	32
3.3.1	Arquitetura síncrona em embarcados.....	32
3.3.2	Processos assíncronos em console	33
3.4	Indexação de dados e Timestamp.....	34
4	CASOS DE USO	36
4.1	Medição de temperaturas de CPU	36
	CONCLUSÃO	42
	REFERÊNCIAS.....	43

5	APÊNDICE	46
5.1	Guias de instalação	46
5.1.1	Configurando Broker.....	46
5.1.2	Publishers em C++.....	46
5.1.3	Publishers em Javascript	46
5.1.4	Subscribers em Javascript	47
5.2	Códigos Fonte	47
5.2.1	Publishers em C++.....	48
5.2.2	Publishers em Javascript	58
5.2.3	Subscribers em Javascript	66
5.2.4	Códigos fonte das aplicações em consoles	74
5.2.5	Códigos fonte das aplicações em plataformas embarcadas	77

1 MOTIVAÇÃO E VISÃO GERAL DO TRABALHO

O cenário atual do desenvolvimento tecnológico encontra-se no meio de uma quarta revolução industrial. Nunca se produziu tantos dados e se utilizaram 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 setores de transporte, análise e inteligência de dados.

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. O trabalho propõe um sistema que permite facilitar tal tarefa.

O projeto propõe uma interface para comunicação entre as diferentes tecnologias e protocolos de aplicação em redes, 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 sobre o 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*) [4], um protocolo que opera sobre o TCP/IP, leve e extremamente utilizado para compartilhamento de dados telemétricos e de mensagens. Ele oferece uma API para aquisição, transmissão, recepção e armazenamento de dados telemétricos.

1.1 Internet das Coisas

”A Internet das Coisas tem o potencial de mudar o mundo. Assim como a Internet fez. Talvez até mais” [5]. Uma tradução livre de Rampim [6] 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 [7], há mais objetos conectados que pessoas no mundo.

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, 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 nesse 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.

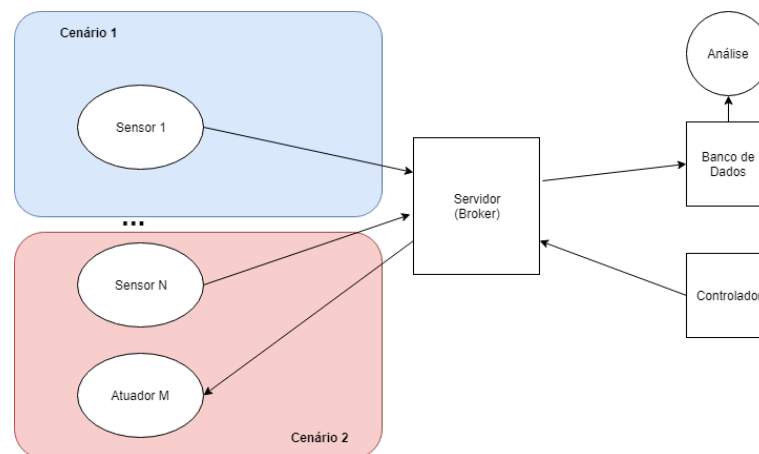


Figura 1 Um exemplo de aplicação IoT que percorre os problemas a solucionar

O sensor 1 está imerso em um ambiente com as 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, é fundamental 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 e criará canais de dados (Data Streams), cuja função será adaptar a transmissão dos dados aos cenários onde apresentam limitações na conexão, seja por problemas de infraestrutura ou congestionamento na rede.

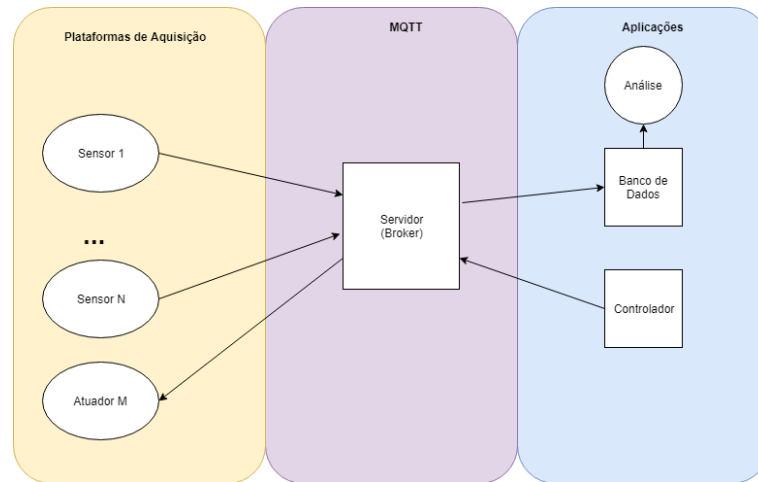


Figura 2 A aplicação dividida em blocos, cada um exercendo uma função em uma aplicação IoT

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.

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 de tecnologias da seção 2.2, 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.

1.3 Interface para Protocolos de Aplicação

Neste projeto será visto a implementação de uma interface entre o desenvolvedor e o protocolo MQTT, implementando o conceito de canais de dados (Data Streams), como mostrado na Figura 3. Algumas destas características podem ser destacadas:

- Multicast. Capaz de enviar mensagens a um ou mais dispositivos simultâneos;
- Envio de mensagens em tempo real;

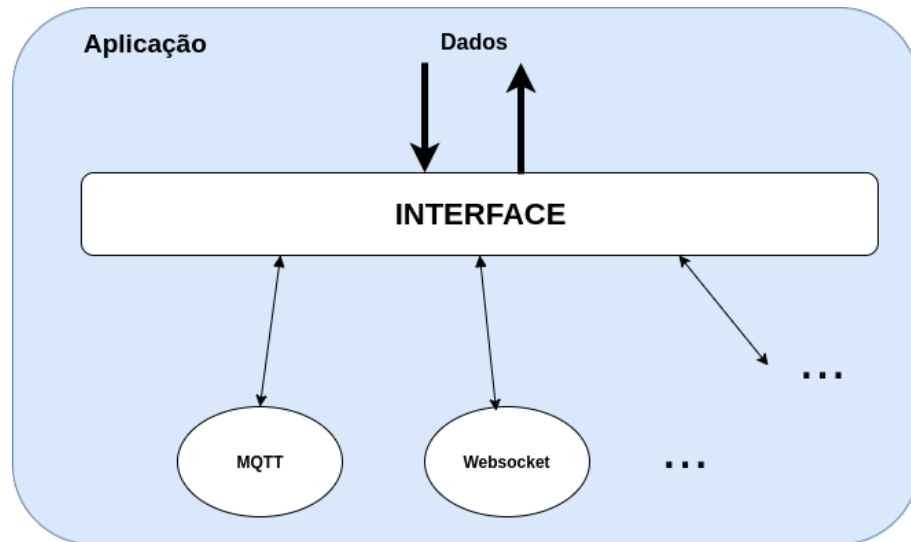


Figura 3 Interface de comunicação. A interface tem sua arquitetura que pode se comunicar com um ou mais protocolos de aplicação

2 CONCEITOS GERAIS DE IOT

No capítulo 1, foram vistos as bases para se implementar um projeto de IoT. A área começou a receber fortes investimentos e atenção por volta de 2009 [6] e desde então foram feitas consideráveis implementações utilizando tecnologias e protocolos diferentes. Neste capítulo serão apresentados conceitos necessários para o projeto.

2.1 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 4.

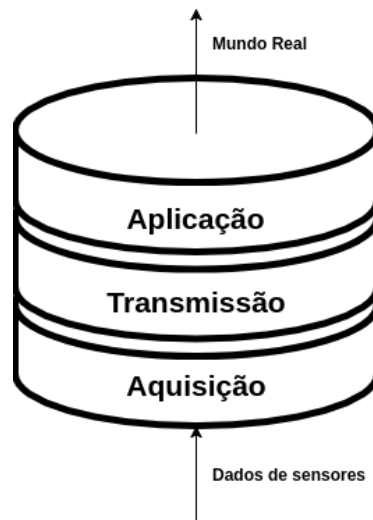


Figura 4 As três camadas do IoT, dos sensores ao mundo real

A primeira camada é a de aquisição de dados, que lida com o mundo físico e captura os dados através de sensores e conversores A/D, também realiza o processamento para entregar em um formato adequado para transmissão e inteligível para aplicações que recebem os dados. 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.

A segunda camada é a camada de transmissão, na qual estão, efetivamente, 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. Esta camada é o coração do IoT. Na transmissão define quais dispositivos eletrônicos e suas especificação técnicas. Também define como os softwares da camada de aplicação e aquisição devem ser implementados baseado na estrutura da pilha de rede.

E por último, a camada de aplicação, a mais abrangente e que envolve maior poder computacional. Ela recebe os dados e lida com os processos de aplicação desses dados, seja análise, visualização, armazenamento ou a estruturação dos mesmos. A camada de aplicação encabeça a pilha do IoT. É ela que de fato trata os dados. Ela disponibiliza os dados para o mundo real, podendo exercer múltiplas funções simultâneas incluindo:

- Armazenamento e Análise, através de Bancos de Dados e Serviços;
- Visualização; Aplicações Web, disponibilizando gráficos e tabelas de dados;
- Inteligência e aprendizado, com Machine Learning e Estatística para decisões e classificação;
- Serviços e servidores . Aplicações em nuvem fornecendo micro-serviços em aplicações Web;

2.2 Tecnologias em IoT

As primeiras aplicações de IoT foram realizadas em laboratórios através de RFID [6], junto com códigos bidimensionais (como o QR Code), para aplicações de identificação de objetos. É uma das soluções mais populares e de baixo custo de IoT utilizando Radio frequência.

Redes que utilizam bandas restritas (NB-IoT) visando baixo consumo e curta distância de transmissão, são utilizadas em redes IoT de celulares e irão se tornar opções dominantes de conectividade com o advento do 5G. As NB-IoT concentram-se especificamente na cobertura, baixo custo, longa duração da bateria e alta densidade de conexão. As mensagens de IoT são geralmente curtas, dados telemétricos, mensagens etc. Já se encontram implementadas algumas redes como SigFox [8] e LoRa [9].

As novas gerações de Bluetooth consomem muito menos energia comparado com as primeiras gerações da tecnologia, o que tornaram a tecnologia viável para aplicações

IoT. Geralmente, módulos Bluetooth são utilizados como beacons [10] que são pontos espalhados por uma região, no qual podem se comunicar com os módulos de dispositivos móveis ao se aproximar, oferecendo links para conteúdo exclusivo a quem está pareado aos pontos (chamados de beacons).

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 (Constrained Application Protocol) [11] protocolo nos mesmos moldes do HTTP com arquitetura REST (Representational State Transfer), que define as regras para criação de uma serviços Web, garantindo interoperabilidade entre sistemas e a Internet. Entretanto o CoAP é mais simples, mais leve, com baixo overhead em relação ao HTTP e utilizado em redes locais.

A interface proposta nesse projeto propõe a arquitetura Publish/Subscribe e se comunicará com Protocolos de aplicações, mesmo que esses não estejam baseados nessa arquitetura . Para critérios de comparação apresenta-se dois protocolos de aplicação construídos sobre o TCP/IP, HTTP e MQTT.

2.3 MQTT

O protocolo MQTT foi 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-os (Subscribe).

Para gerenciar os clientes (responsáveis pela implementação da comunicação MQTT) em cada dispositivo é necessário um servidor chamado Broker. Este foi implementado com o Mosquitto [12], um broker open source, leve capaz e de ser instalado localmente e no servidor do laboratório para testes remotos.

2.3.1 MQTT X HTTP

MQTT e HTTP são protocolos de aplicação construídos sobre TCP/IP. HTTP é o 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, uma conexão não-persistente, como ilustrado na Figura 5, requer mais banda e consome mais energia que protocolos leves e de conexão persistente, na qual o canal de dados permanece aberto e a conexão não é encerrada após o envio de dados, como o MQTT. Logo, com uma breve avaliação, é possível perceber que o HTTP não é o protocolo de aplicação mais eficiente para a aplicação deste projeto.

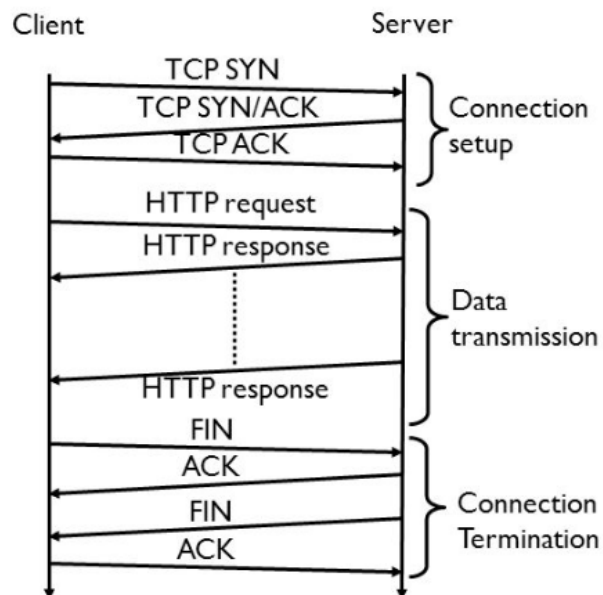


Figura 5 Fluxo de conexão do HTTP

MQTT é um protocolo com cabeçalhos menores, conexão persistente e menos passos para o envio de mensagem, como visto na Figura 6. Por ser um protocolo feito com o objetivo de reduzir a latência, leva vantagem para a transmissão contínua de dados em relação ao HTTP.

Abaixo na Tabela 1 encontra-se o resumo comparativo entre os protocolos MQTT e HTTP:

2.3.2 Publishers e Subscribers

Para enviar e receber dados de uma forma a atender os requisitos da seção 1.3, foi utilizado um padrão de comunicação recorrente em aplicações contemporâneas, o padrão

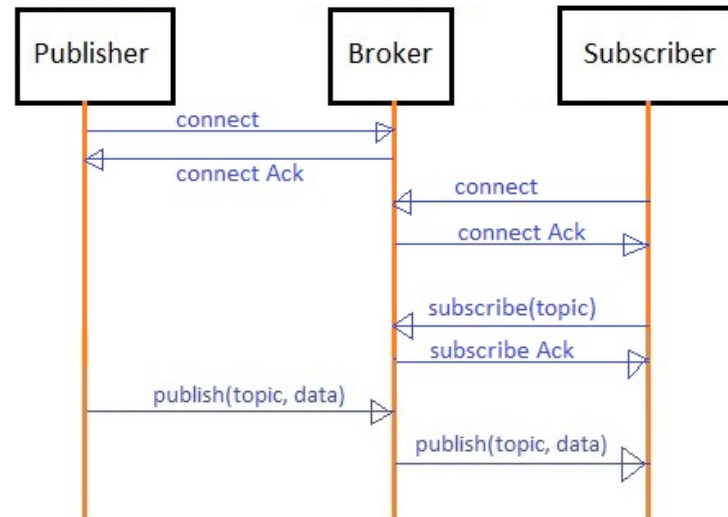


Figura 6 Fluxo da conexão do MQTT

Tabela 1 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

Publish/Subscriber [1].

O padrão Publish/Subscribe permite que as mensagens sejam transmitidas assíncronas e para vários dispositivos simultaneamente. Para transmitir uma mensagem, um cliente pode simplesmente enviar uma mensagem para um determinado tópico. 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 do tópico.

Qualquer mensagem publicada em um tópico é imediatamente recebida por todos os assinantes deste tópico. Com isso, foram criados duas funções possíveis para cada dispositivo dentro deste padrão, os Publishers, quem enviarão as mensagens de dados e os Subscribers, quem receberão. Sua comunicação é ilustrada na Figura 13.

2.3.3 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

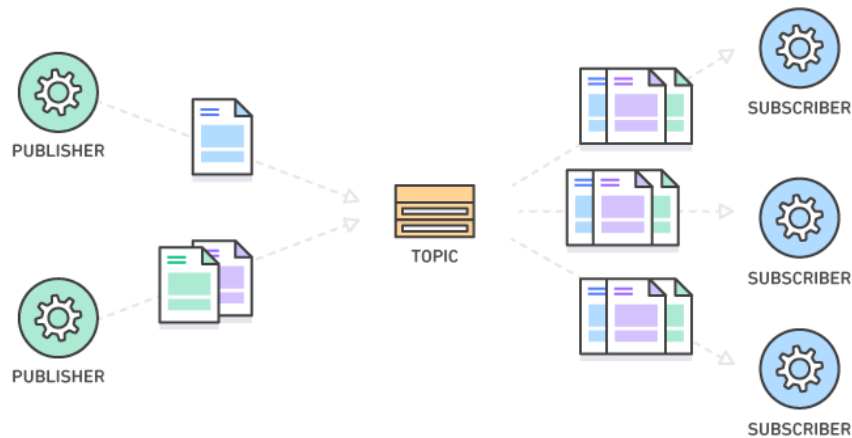


Figura 7 O padrão Publish/Subscribe. Retirado de [1]

não persiste informações dos clientes, ou seja, em caso de queda de conexão, os clientes devem se inscrever novamente nos tópicos. A arquitetura Broker não é exclusividade do MQTT, outros protocolos utilizam esse tipo de implementação em servidores em arquiteturas semelhantes de envio de mensagem, como os Hubs desenvolvidos pela Microsoft na biblioteca SignalR [15].

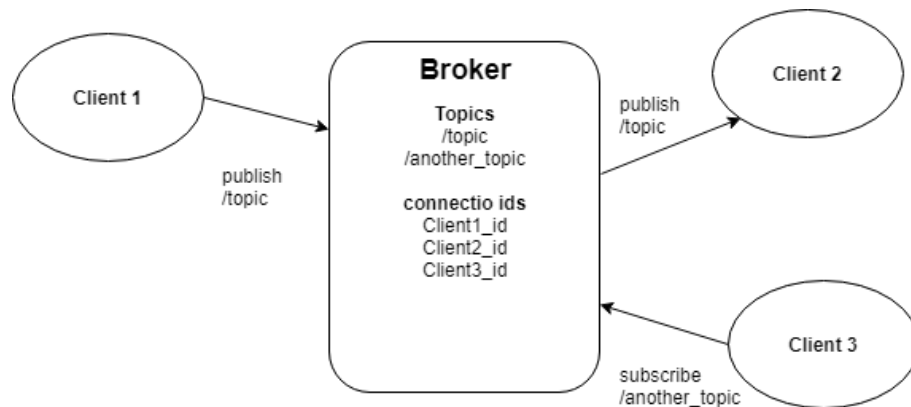


Figura 8 Exemplo de gerenciamento de um broker

Na figura Figura 8, 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* e o cliente 1 envia ordem de publicação para *topic*.

2.3.4 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.

Dentre esses requisitos, tem-se atender dispositivos que não usam a pilha TCP/IP e medidas de segurança da informação nas mensagens enviadas.

Existe uma gama de dispositivos que utilizam protocolos não construídos sobre o TCP/IP, a exemplo do ZigBee [16]. Para isso foi criada uma versão do MQTT para atender esses 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 [17], camada de segurança construídas sobre TCP/IP. Esta camada envolve o processo de encriptação dos cabeçalhos da aplicação e autenticação por certificados. Para poder criar uma conexão é requisitado um certificado SSL, durante o processo de solicitação do certificado SSL, a Autoridade de Certificação validará seus detalhes, permitindo a passagem de dados encriptados.

2.4 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 ou por conta da reutilização do espaço da memória, 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, que são estruturas que organizam o armazenamento de dados persistentes em arquivos. Um Banco de dados é 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.

Um banco é composto por duas ferramentas, o Motor e o Arquivo de dados. O motor é quem realiza as ações sobre o arquivo, ou seja, 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 pelo próprio motor, baseado em algoritmos de estrutura de dados. Um Banco de dados pode possuir vários motores, cada um com um algoritmo de distinta eficiência em termos de tempo de escrita e/ou leitura de acordo com o dado recebido.

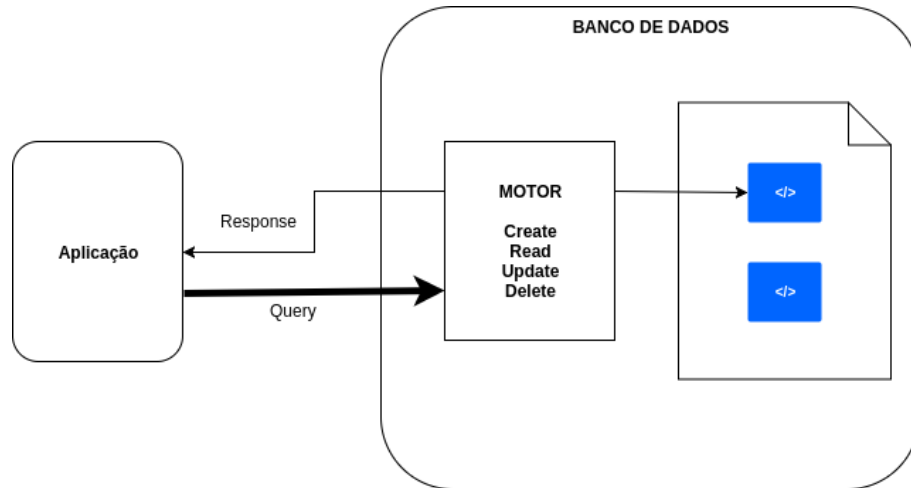


Figura 9 A arquitetura de um banco de dados

O arquivo é o documento onde os dados são armazenados. A forma de armazenamento de dados define a eficiência do motor em buscar e escrever dados, então é necessário a escolha adequada do algoritmo para uma maior eficiência da leitura do arquivo, como em Figura 9, a aplicação envia uma solicitação de criação, leitura, atualização ou remoção (CRUD - Create, Read, Update, Delete) e o motor escreve (ou lê) do arquivo.

2.5 Bancos para Aplicações IoT

Como pode ser observado em [18], uma aplicação eficiente de Bancos de Dados em IoT está ligada ao tempo de inserção de dados no banco, ou seja, o tempo total em que a aplicação leva para enviar e inserir o dado 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 tempo ele leva para escrever o dado no documento. A maioria dos dados são armazenados em estruturas de árvore B+ [19], porém pode-se observar em [18] que a estrutura de árvore LSM [20] possui maior eficiência na escrita.

Comparando as duas estruturas, como mostrado em Figura 10, percebe-se que a estrutura LSM sustenta até 2x mais inserções da estrutura B+, hoje os Bancos de Dados modernos utilizam desta vantagem. Aplicações modernas coletam dados constantemente, então a eficiência da operação de escrita feita por um motor de um banco torna-se extremamente relevante, a estrutura LSM leva vantagem neste quesito.

Existem várias abstrações de Bancos de dados. Em [21] compara-se e conclui-se

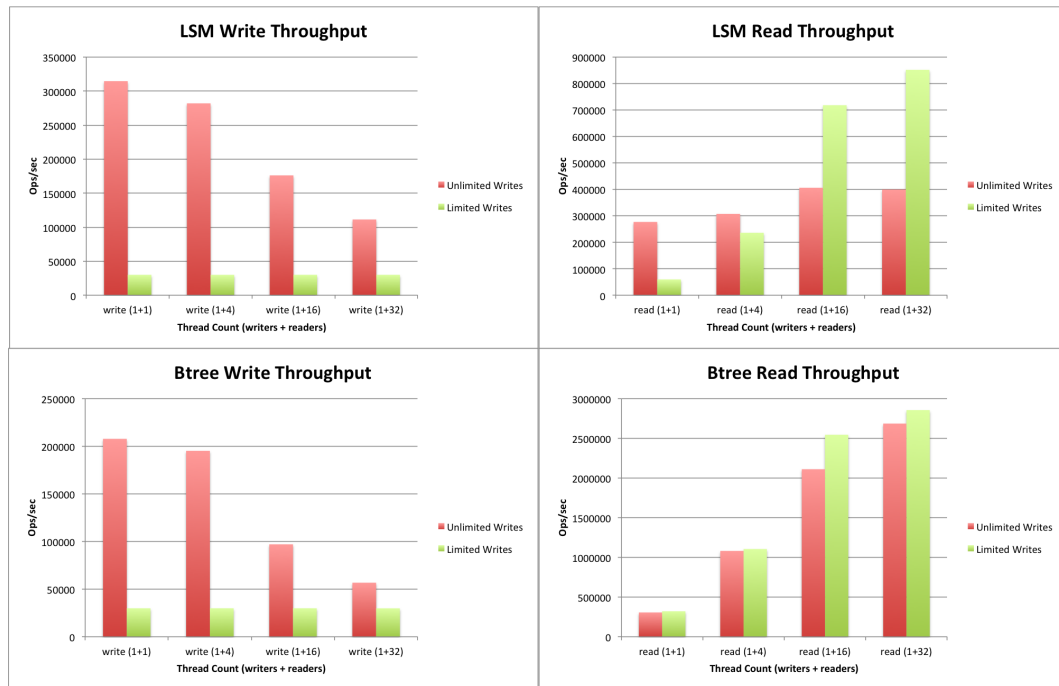


Figura 10 Comparação de Escrita e Leitura entre LSM e B+, retirado de [2]

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 (bancos não-relacionais) são, em geral, mais leves e possuem uma flexibilidade maior para lidar e estruturar dados, o que faz esses tipos de Banco mais favoráveis a aplicações de IoT, porém outras estruturas como um banco dividido em séries de tempo [22], onde a indexação dos dados é feita pelo tempo de quando a amostra foi coletada (isso será debatido em 3.4), mostram-se eficientes com bancos relacionais 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 [23] e são usados, na maioria dos casos, para armazenar os dados quando não há conexão. 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 em série de tempo, feito para aplicações em tempo real. Mas para o projeto, foi utilizado o MongoDB [24], um banco NoSQL, leve e 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 O PROJETO

Devido a interação entre dispositivos de aquisição de dados e os de aplicação e armazenamento de dados em diferentes cenários (como descrito em 1.2), foi necessário uma implementação de um protocolo de comunicação entre os dispositivos em diferentes linguagens de programação.

A arquitetura Publish/Subscribe define as configurações gerais do sistema, não contempla as mudanças de cenário possíveis, onde, por exemplo, a rede esteja congestionada e é necessário tomar medidas como reduzir a taxa de dados que são enviados em tempo real. 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 de Data Stream.

O protocolo consiste em configurações de uma abstração de um canal de envio de dados chamado Data Stream mostrado em Figura 11, no qual transitam dados em uma determinada velocidade, podendo conter um limite de pacote de dados. Nas pontas desse canal estão os Publishers e Subscribers, descritos na seção 2.3.2. Este conceito permite que a interface possa reagir a limitações de transmissão, como congestionamento na rede. O desenvolvedor irá se preocupar somente com as configurações do canal.

3.1 Data Streams

Um Data Stream é uma parte da interface que permite adicionar configurações de como o dado será enviado pelo Publisher para lidar com problemas na transmissão, como o próprio cenário de congestionamento da rede já citado. As configurações podem ser enviadas (ou alteradas) pelo Subscriber, permitindo configurações dinâmicas para o Data Stream utilizado na transmissão. 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.

Publishers são dispositivos que criam Data Stream e enviam dados por esses, 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 os Publishers 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 a adição de um novo parâmetro (como um limite de dados por mensagem, por exemplo) a ser adicionado a configuração.

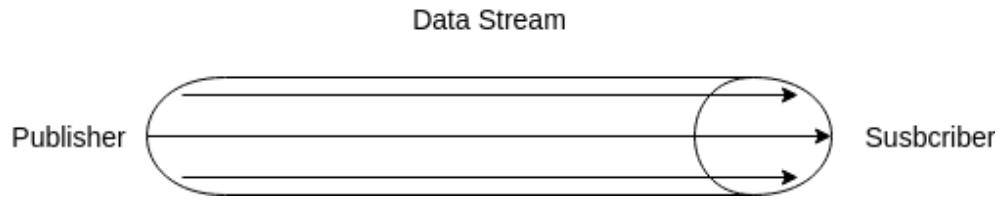


Figura 11 O conceito de Data Stream para a abstração do transporte de dados

Para criar um Data Stream, basta um Subscriber estar assinando um tópico no formato abaixo. E um Publisher publicar neste tópico, como na Figura 12. O Id corresponde a uma identificação única que pode ser definida pelo desenvolvedor. O *stream_nome* corresponde ao tipo de Data Stream utilizado, que serão detalhados a seguir.

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

Existem dois tipos de Data Stream já implementados nos Publishers. Porém o desenvolvedor pode implementar seus próprios Data Streams dependendo da linguagem de programação utilizada, logo existem três tipos de Data Stream definidos:

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

Os Subscribers podem alterar as configurações específicas de cada tipo de Data Stream, baseada nas necessidades da aplicação, como problemas de processamento ou congestionamento na rede etc.

Para um Subscriber enviar configurações basta publicar no tópico abaixo. As configurações são feitas por uma string JSON [25], um conjunto de chaves-valor universalmente interpretada por várias linguagens de programação como forma de transporte de objetos de uma classe.

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

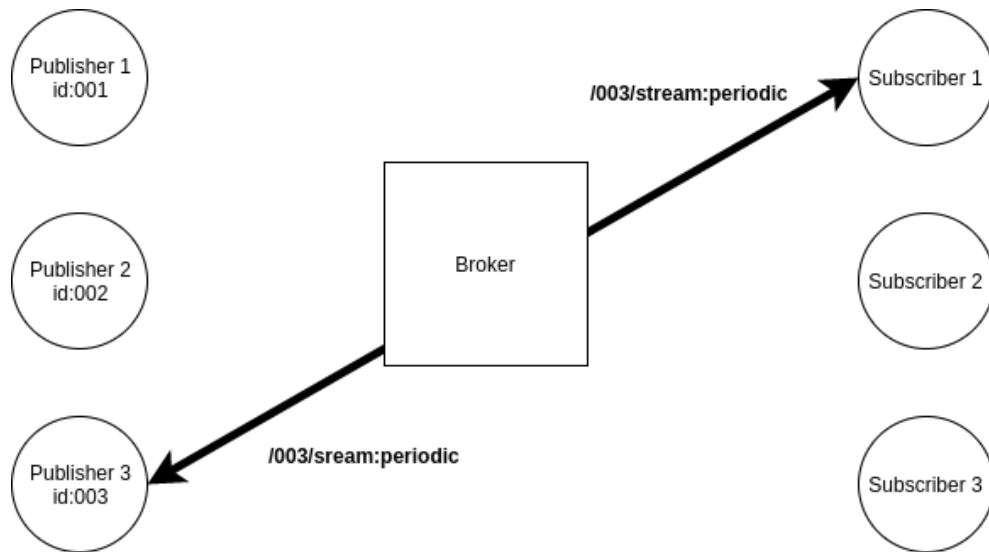


Figura 12 Um Data Stream é criado a partir do tópico */003/stream:periodic*

A Figura 13 descreve um cenário onde um Publisher e um Subscriber estão transmitindo dados por um Data Stream pelo tópico *textit/003/stream:periodic*, as setas indicam que o Publisher pode enviar dados para um Subscriber por um Data Stream do tipo *Periódico* com Id *003*, assim como o Subscriber pode enviar alterações nos parâmetros de configuração deste tipo de Data Stream, como o próprio período.

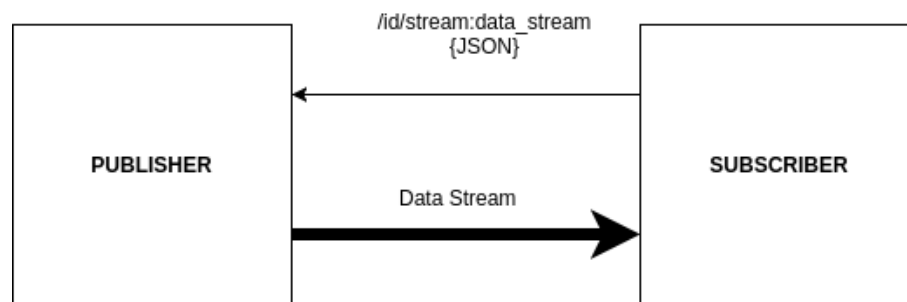


Figura 13 Uma outra representação da comunicação entre Publishers e Subscribers por Data Stream

3.2 Implementação em Plataformas

A camada de aquisição apresenta a implementação dos Publishers, que enviam os dados. As plataformas possuem unidades de processamento e módulos de rede e se comunicam com sensores, para coletar dados físicos, e atuadores que recebem instruções para a execução de alguma função, como abrir/fechar alguma porta ou compartimento, ou ligar alguma iluminação e acionamentos em geral.

3.2.1 Sistemas Embarcados

Sistemas Embarcados, são sistemas geralmente alimentados por baterias, sem alimentação de rede elétrica, portáteis, econômicos, com sistemas de controles, na maioria dos casos, feitos por micro-controladores ou microprocessadores, podendo contemplar sistemas operacionais simples. Com essa descrição, pode-se imaginar que esses dispositivos possuem processamento, energia e desempenho limitados. Neste caso, foi necessário a criação de uma implementação de interface leve, que não consuma muito armazenamento e processamento do sistema embarcado, e eficiente no consumo de energia.

Foram escolhidas para a implementação em embarcados as plataformas micro-controladas com arquitetura Espressif [3], como a arquitetura ESP32 que contempla MCUs (Micro-Controller Units), módulos WiFi, sensores internos, entradas e saídas analógicas e digitais e até Bluetooth (não utilizado na versão atual), mostrados na Figura 14. Pela descrição técnica pode-se ver um poder de processamento maior que um Arduino [26], muito utilizado nessas aplicações e que também é compatível com a interface se adicionado shields WiFi. O ESP utiliza linguagem C++ [27] com framework Arduino para implementar os Publisher e os Data Stream contidos neles no ESP32.

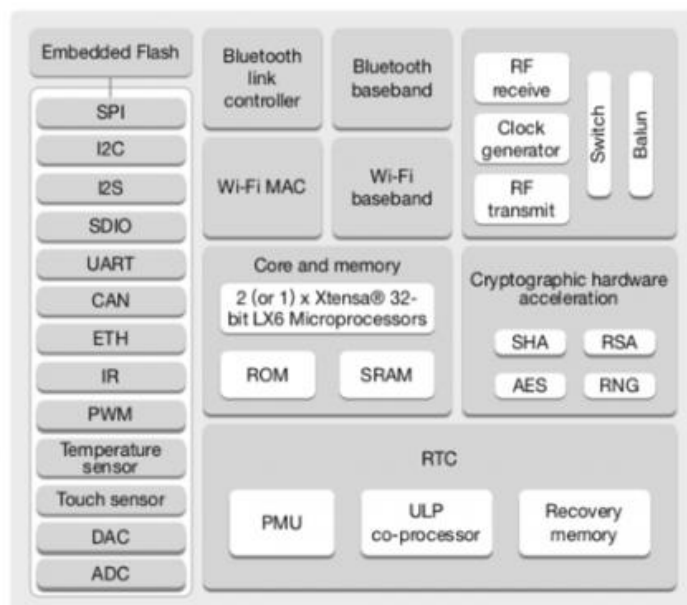


Figura 14 A arquitetura do ESP32, retirado de [3]

Também foram implementadas, em Javascript utilizando Node.js [28], aplicações do Publisher para embarcados com sistemas operacionais, como Raspberry Pi [29]. Permitindo multi-uso entre as funções de Publisher e Subscriber. Esses embarcados mais

robustos possuem processadores mais potentes, módulos de rede, sistemas operacionais, entradas e saídas digitais entre outros.

3.2.2 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 um interpretador de Javascript que permite criar aplicações sem a necessidade de usar um browser para interpretar Javascript, como interfaces de linha de comando, aplicações Desktop, entre outros processos permitidos no Sistema operacional. Node.js possui extensas bibliotecas para HTTP e MQTT além de pipelines que permitem fácil comunicação de protocolos e transição de dados no mesmo processo.

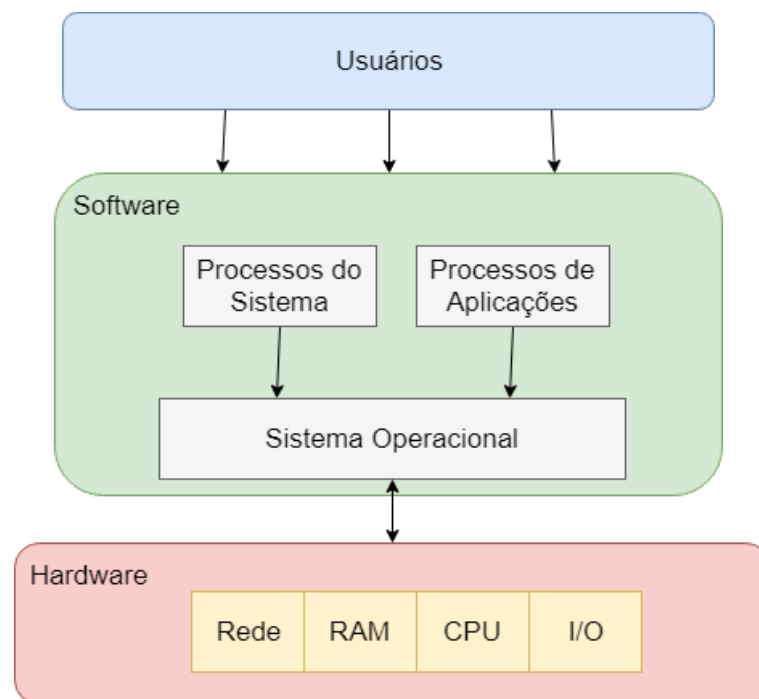


Figura 15 A arquitetura simplificada de dispositivos com Sistema Operacional

Além disso, Node.js é uma ferramenta multi-plataforma, 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 15 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 instâncias 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), diminuindo a complexibilidade do sistema, evitando a necessidade de comunicação inter-processos (IPC). Com isso foram implementadas bibliotecas que constroem a interface sobre o MQTT.

3.3 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 funcionalidades que um console gerido por um sistema operacional, sendo uma das principais o paralelismo de processos, a capacidade de ser multi-tarefas. Por isso exigiram-se duas filosofias diferentes de implementação do sistema para os dois tipos de plataformas.

3.3.1 Arquitetura síncrona em embarcados

Sistemas embarcados possuem um poder de processamento limitado, apesar da tendência de desenvolver embarcados mais potentes. Esses sistemas são geralmente Micro-controlados, ou seja, com arquiteturas mais simples, geralmente executando instruções de somente um programa compilado para linguagem do MCU e gravado neste. Não há paralelismo, cada instrução é síncrona, ou seja, são executadas uma a uma, pela unidade de processamento.

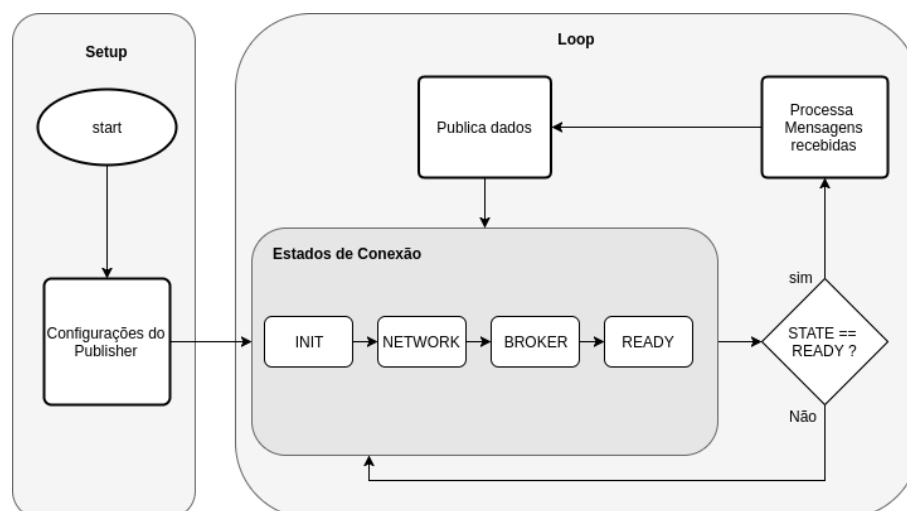


Figura 16 Diagrama simplificado de uma rotina padrão seguida pela implementação em Microcontroladores

A Figura 16 simplifica a lógica da implementação em dispositivos síncronos como micro-controladores. 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.

3.3.2 Processos assíncronos em console

Consoles são dispositivos que possuem uma Arquitetura mais complexa, consequentemente mais poder de processamento. Na Figura 15, 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. O interpretador Node.js possui uma gama de bibliotecas para a implementação de funções assíncronas, diferentemente dos Sistemas Embarcados microcontrolados, que executa Threads em paralelo, para receber e enviar mensagens simultaneamente.

A Figura 17 mostra o funcionamento simplificado das aplicações de Publisher e Subscriber em Node.js. A aplicação faz as configurações específicas de cada objeto Publisher ou Subscriber e logo após entra em um Loop de eventos, onde existem eventos de Conexão com a rede e com o Broker, e de envio/recebimento de mensagens, entre outros. Esses eventos quando são acionados pelo sistema, executam callbacks, funções assíncronas definidas e associadas a um evento. Esses callbacks são gerenciados internamente pelo Node.js e direcionadas ao loop de eventos.

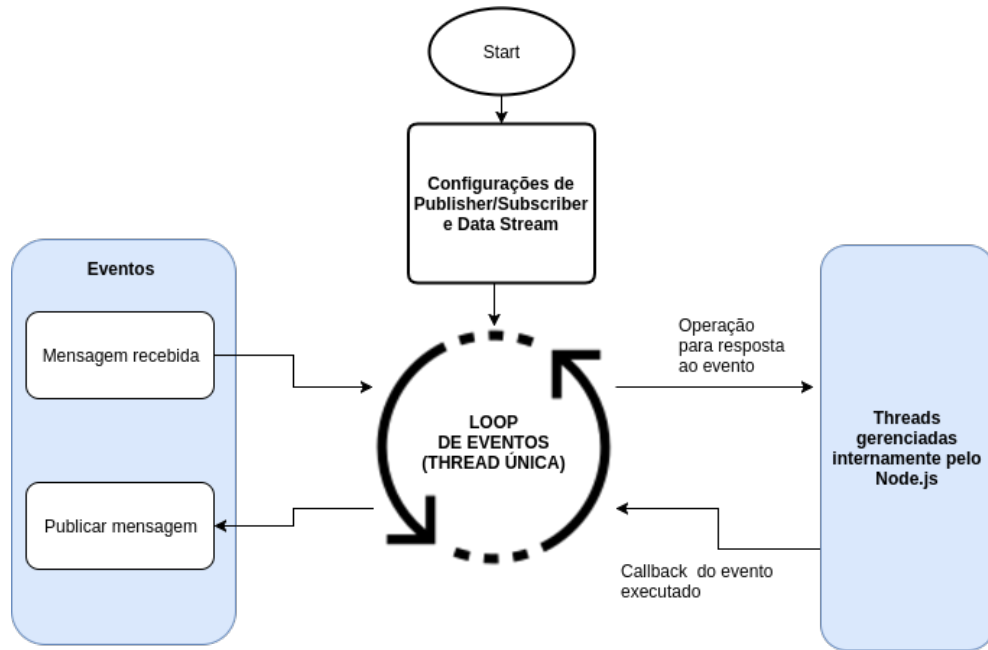


Figura 17 Diagrama simplificado de uma rotina assíncrona padrão em Consoles

3.4 Indexação de dados e Timestamp

Na seção, 2.5 foram discutidos estruturas da organização de dados e o formato de armazenamento de dados como o formato de documento e séries de tempo. É importante que esses 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 data e hora da coleta do dado. Isto permite a análise dos dados ao longo do tempo. Aplicações de decisão e análise utilizam ferramentas estatística com base nas ocorrências temporais, projetando previsões e classificação. No projeto atual, foi implementado o MongoDB, que é um banco de dados de documentos. Cada documento é indexado por uma identificação única como mostrado na Figura 18, permitindo também criar relações entre documentos.

O banco também permite adicionar outros parâmetros de indexação definidos pelo desenvolvedor, como o Timestamp, uma informação de data e hora da inserção no banco. Na implementação foi utilizado o formato Unix Timestamp, o número de milisegundos que se passaram desde 1 de Janeiro de 1970 às 00:00:00 (UTC). A Figura 19 ilustra o formato de documento completo usado no sistema, o campo de dados é definido pelo desenvolvedor.

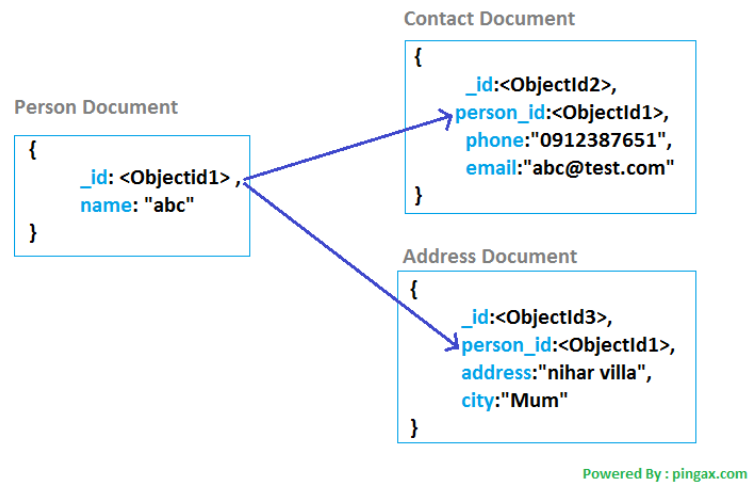


Figura 18 O formato de documento no MongoDB

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

Figura 19 Adição de parâmetro de timestamp em milisegundos ao documento

Vale ressaltar a existência de uma nova geração de Bancos de Dados de séries de tempo, um formato parecido com o de Documento, porém com indexação feita por Timestamp ao invés de uma chave única. Em destaque o Banco InfluxDB, que possui estrutura LSM-Tree além de ser um banco de séries de tempo, o que o faz ser utilizado cada vez mais em aplicações IoT.

4 CASOS DE USO

Este capítulo busca demonstrar o funcionamento do sistema em dispositivos 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

¹ 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 e uma instância do Subscriber para receber estas temperaturas via MQTT e persisti-las em banco de dados. Ambas as aplicações utilizarão as APIs em Javascript, através do 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 [30]. Isso foi implementado no código 5.2.4.

As CPUs são tecnologias feitas por transistores. Milhões de aglomerações de MOSFETS que começam a ter perda de desempenho de processamento conforme o aumento de temperatura, em [31]. 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 essa perda de desempenho. CPUs modernas são capazes de ajustar suas frequências operacionais, a fim de reduzir seu consumo de energia ou fornecer a máxima potência. 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.

¹**obs:** Para reproduzir este teste, é necessário seguir as instruções encontradas no Apêndice.

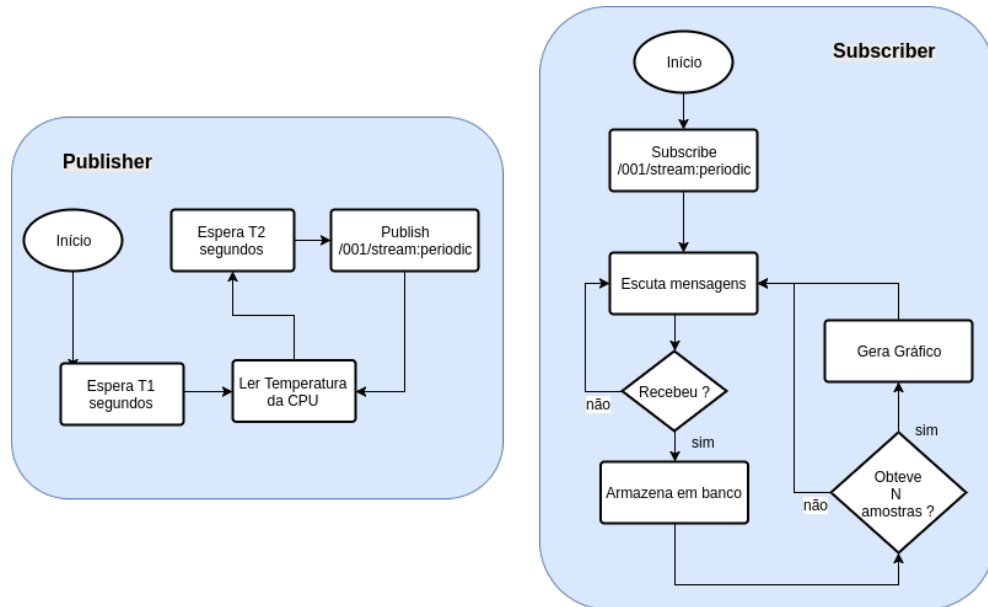


Figura 20 Diagrama de fluxos do Publisher e do Subscriber

A Figura 20 mostra 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 amostras (definidas pelo desenvolvedor), um gráfico de Temperatura da CPU principal pela Data-Hora de inserção é gerado.

A Figura 21 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 possui o timestamp da inserção no banco, o campo *data* são os 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 do campo *timestamp* e do 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, além de disponibilizar links públicos para acesso aos gráficos na Internet. 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 a segunda opção com o módulo *plotly.js*, que é a implementação em Javascript da plataforma. A cada N amostras são produzidos gráficos como o da Figura 22.

A Figura 22 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

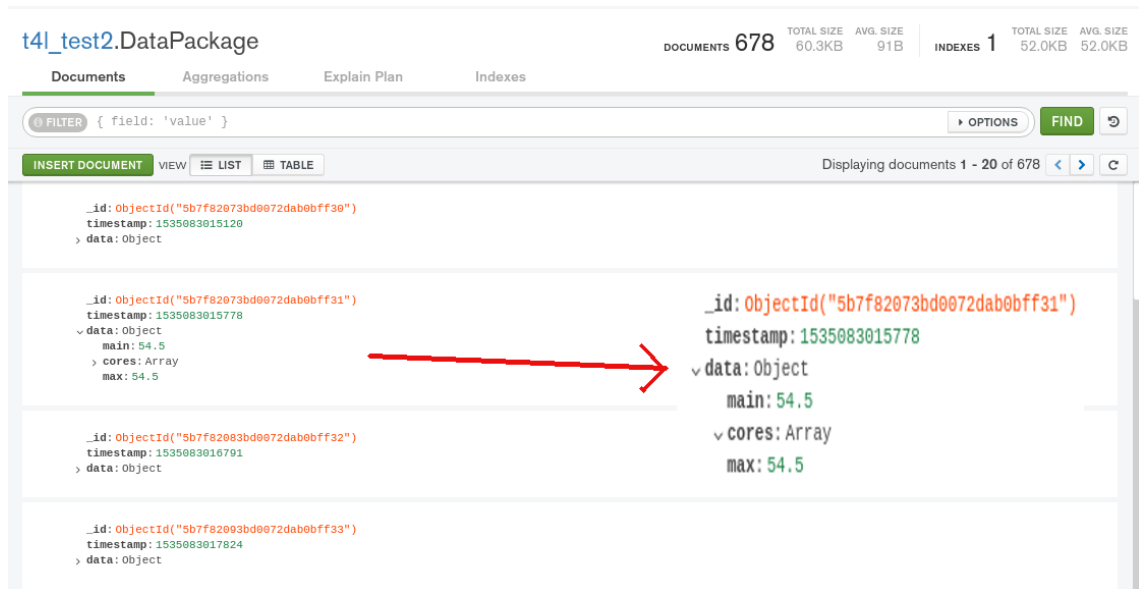


Figura 21 Visualização dos dados armazenados em documento em uma coleção do MongoDB

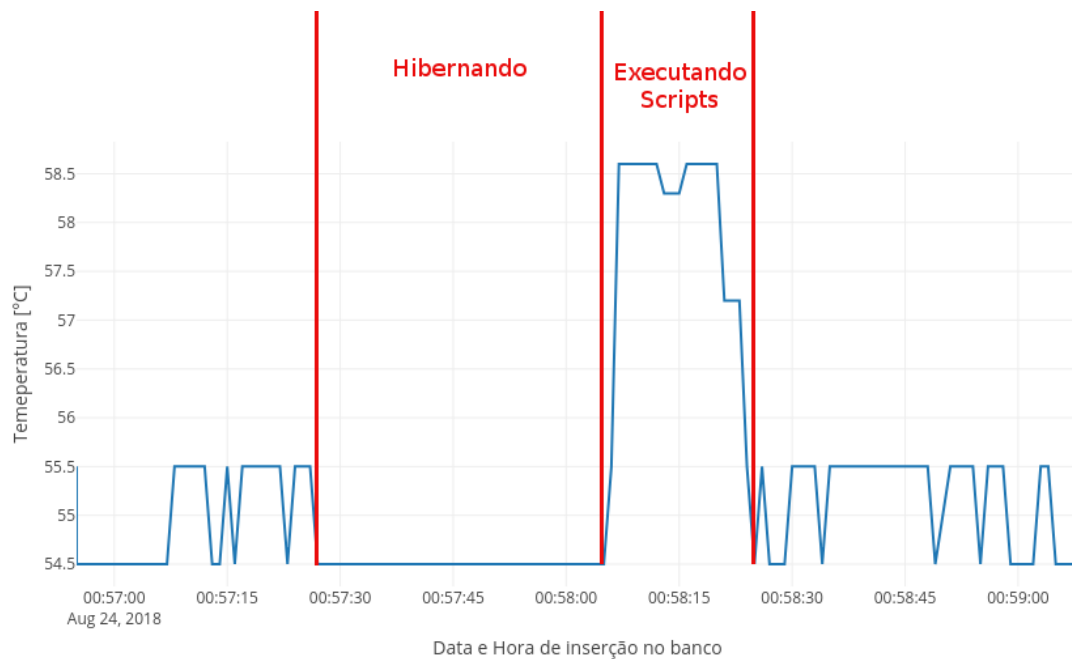


Figura 22 Comportamento da temperatura em três momentos

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 redução 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 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. Além das medições desta CPU core i7, foi acrescentada uma aplicação contendo um Publisher em um ESP32. Conforme descrito no capítulo anterior na Figura 14, o ESP32 é um MCU que possui módulos WiFi e sensor de temperatura interno, facilitando a medição, como trata-se de um Microcontrolador uma implementação síncrona.

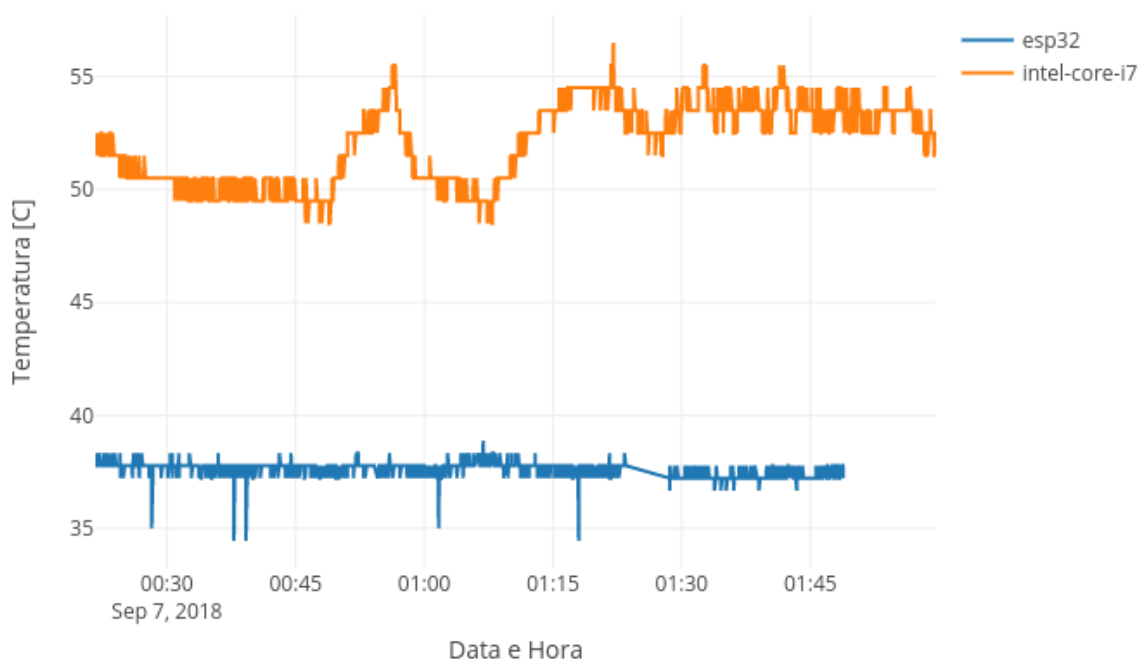


Figura 23 Comparação das temperaturas com uma CPU core i7 e ESP32

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 processamento, fora o tempo de envio para Broker. É de se esperar que o número de amostras de cada um sejam diferentes entre si. Pela Figura 23, 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, por estar rodando somente um programa devido suas limitações. A CPU i7 apresenta temperatura em torno de 50 graus Celsius, possuindo variações mais bruscas, decorrente do proces-

sador 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, reproduzindo 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).

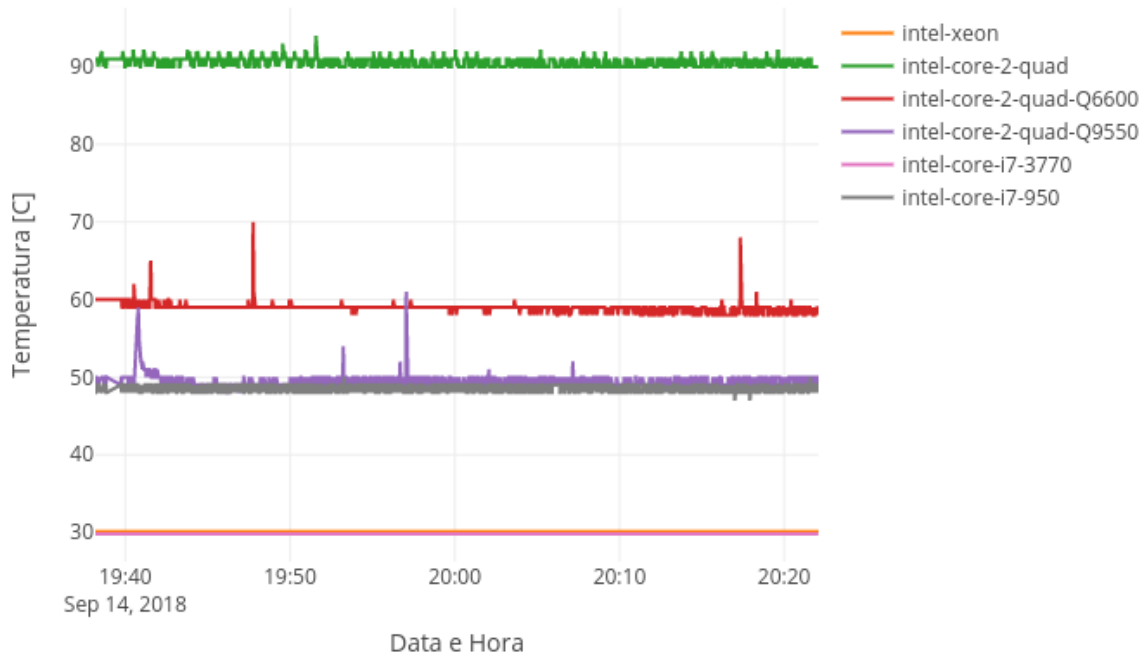


Figura 24 Comparação das temperaturas em múltiplos processadores.

Na Figura 24, observam-se vários níveis de temperatura média entre os processadores, todos eles executando processos comuns dos sistemas operacionais em máquinas Windows e Linux. Esta diferença está atrelada aos fatores de resfriamento térmico e a capacidade de processamento. Vale ressaltar alguns comportamentos. Os dois processadores de temperatura mais baixa, por volta de 30 graus Celsius 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 e, majoritariamente, pela manutenção do resfriamento do processador, foi detectado que a CPU estava sem pasta térmica, grande responsável pela troca de calor na própria.

CONCLUSÃO

Neste projeto, foram apresentados todos os aspectos de hardware e software em seus capítulos. Encontrou-se uma certa dificuldade em implementar o Data Stream nas duas arquiteturas apresentadas, de modo a ter o mesmo funcionamento em ambas. As linguagens e plataformas usadas possuem uma grande quantidade de bibliotecas que facilitaram a realização dos códigos que implementam Publishers, Subscribers e Data Streams.

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 permissíveis, o que significa que possuem versões gratuitas e escaláveis, de modo a também oferecerem a possibilidade de criar versões personalizadas.

O projeto englobou conceitos nas áreas de Eletrônica, Telecomunicação e Computação espalhadas pelas camadas de IoT. O aprendizado e a integração de ferramentas nessas áreas, permitiu criar um caso de uso que validou o funcionamento do sistema. A medição de temperatura das CPUs foi uma forma de observar o comportamento do sistema, conforme o aumento do número de dispositivos utilizados, além de gerar conclusões a partir das medições de temperatura. Porém, o diferencial são os Data Streams. Sua implementação permitiu diferentes formas de transmitir dados em uma mesma aplicação de uma forma que a arquitetura Publish/Subscribe não permite, além de mudanças dinâmicas destas formas de transmissão.

Planeja-se dar um maior enfoque na implementação dos Data Streams em versões futuras, criando novos tipos de Data Streams e expandindo o suporte para outras linguagens de programação. O sistema já é compatível com o protocolo de segurança SSL/TLS, porém outras formas de segurança como encriptação das mensagens e suporte para Blockchain estão no planejamento para novas versões do sistema. Por fim, planeje-se implementar futuramente as funcionalidades de adicionar mais de um broker, para aplicações que exigem um grande fluxo de dados e estender a interface para outros protocolos de aplicação.

REFERÊNCIAS

- [1] AWS. *Pub/Sub Messaging Asynchronous event notifications*. Disponível em: <<https://aws.amazon.com/pt/pub-sub-messaging/>>. Acesso em: 21/07/2018.
- [2] WIREDTIGER. *B-Trees vs LSM*. Disponível em: <<https://github.com/wiredtiger/wiredtiger/wiki/Btree-vs-LSM>>. Acesso em: 21/07/2018.
- [3] Espressif. *Hardware*. Disponível em: <<https://www.espressif.com/en/products/hardware>>. Acesso em: 21/07/2018.
- [4] IBM. *MQTT*. Disponível em: <<http://mqtt.org/>>. Acesso em: 21/07/2018.
- [5] ASHTON, K. That 'internet of things' thing. v. 22, p. 97–114, 01 2009.
- [6] DIAS, R. R. de F. *Internet das Coisas sem Mistérios*. [S.l.]: Netpress Books, 2016.
- [7] EVANS, D. The internet of things how the next evolution of the internet is changing everything. Cisco Internet Business Solutions Group (IBSG), 04 2011.
- [8] Sigfox. *Sigfox Technology Overview*. Disponível em: <<https://www.sigfox.com/en/sigfox-iot-technology-overview>>. Acesso em: 21/07/2018.
- [9] LoRa Alliance, Inc. LoRaWAN™ 1.0.3 Specification. 2017.
- [10] 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.
- [11] CoAP. *Constrained Application Protocol (CoAP)*. Disponível em: <<http://coap.technology/>>. Acesso em: 21/07/2018.
- [12] Eclipse. *Mosquitto*. Disponível em: <<https://mosquitto.org/>>. 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] Microsoft. *Introduction to SignalR*. Disponível em: <<https://docs.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr>>. Acesso em: 21/07/2018.
- [16] Zigbee Alliance. *Zigbee Alliance*. Disponível em: <<https://www.zigbee.org/>>. Acesso em: 21/07/2018.
- [17] 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.
- [18] Prithiviraj, Damodaran. *Art of choosing a datastore*. Disponível em: <<http://bytecontinnum.com/2016/02/choosingtorechoice/>>. Acesso em: 21/07/2018.
- [19] ANDREAS, K.; LEFTERIS, K.; DANI, M. *<http://www.di.ufpb.br/lucidio/Btrees.pdf>*. Disponível em: <B-trees>. Acesso em: 21/07/2018.
- [20] PATRICK, O. et al. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 2009.
- [21] SHARVARI, R.; M., B. D. D. Mysql and nosql database comparison for iot application. 2016.
- [22] INC, I. *Time Series Database (TSDB) Explained*. Disponível em: <<https://www.influxdata.com/time-series-database/>>. Acesso em: 21/07/2018.
- [23] PORNPAT, P.; MIKIKO, S.; MITARO, N. Low-power distributed nosql database for lot middleware. 2016.
- [24] INC, M. *What is MongoDB ?* Disponível em: <<https://www.mongodb.com/what-is-mongodb>>. Acesso em: 21/07/2018.

- [25] 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.
- [26] Arduino. *What is Arduino ?* Disponível em: <<https://www.arduino.cc/en/Guide/Introduction>>. Acesso em: 21/07/2018.
- [27] cplusplus.com. *A Brief Description of C++*. Disponível em: <<http://www.cplusplus.com/info/description/>>. Acesso em: 21/07/2018.
- [28] Node.js. *About Docs*. Disponível em: <<https://nodejs.org/en/docs/>>. Acesso em: 21/07/2018.
- [29] 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.
- [30] 2018, P. *Plotly Community Feed*. Disponível em: <<https://plot.ly/#/>>. Acesso em: 21/07/2018.
- [31] JITTY, J.; KEERTHI, K. N.; AJITH, R. Analysis of temperature effect on mosfet parameter using matlab. IJEDR, v. 4, 2016.

5 APÊNDICE

5.1 Guias de instalação

Atenção: Para obter a versão mais atualizada e outras versões, acesse <https://github.com/fol21>. Este guia tem como objetivo reproduzir os testes feitos no sistema, 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: 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);

    continuous_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 continousStream = new DataStream('continous');

continousStream.onMessage();

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

module.exports = continousStream;

```

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 available.')
        } 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+)\backslashconfigurebackslashstream:(\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
 * @memberof 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'),
  MongoDataClient : 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();

```

5.2.5 Códigos fonte das aplicações em plataformas embarcadas

Códigos 5.15 Teste do publisher no ESP32

```

#include <MqtttPublisher.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", "
                    continuous", "{ \"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());
}
```