



**UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS DE CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO**

JARDEL OSORIO DUARTE

**ANÁLISE DE DESEMPENHO DO RASPBERRY PI COMO BROKER DO
PROTOCOLO MQTT**

**CHAPECÓ
2022**

JARDEL OSORIO DUARTE

**ANÁLISE DE DESEMPENHO DO RASPBERRY PI COMO BROKER DO
PROTOCOLO MQTT**

Trabalho de conclusão de curso apresentado como requisição para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.
Orientador: Prof. Dr. Marco Aurelio Spohn

**CHAPECÓ
2022**

Duarte, Jardel Osorio

Análise de desempenho do Raspberry Pi como broker do protocolo
MQTT / Jardel Osorio Duarte. – 2022.

59 f.: il.

Orientador: Prof. Dr. Marco Aurelio Spohn.

Trabalho de conclusão de curso (graduação) – Universidade Federal
da Fronteira Sul, curso de Ciência da Computação, Chapecó, SC, 2022.

1. Rede IoT. 2. Inspecionar. 3. *Protocolo MQTT*. 4. Raspberry
Pi 4. 5. Corretor. I. Spohn, Prof. Dr. Marco Aurelio, orientador.
II. Universidade Federal da Fronteira Sul. III. Título.

JARDEL OSORIO DUARTE

**ANÁLISE DE DESEMPENHO DO RASPBERRY PI COMO BROKER DO
PROTOCOLO MQTT**

Trabalho de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Dr. Marco Aurelio Spohn

Este trabalho de conclusão de curso foi defendido e aprovado pela banca avaliadora em:
01/04/2022 .

BANCA AVALIADORA

Marco A. Spohn

Prof. Dr. Marco Aurelio Spohn – UFFS

Adriano S. Padilha

Prof. Me. Adriano Sanick Padilha – UFFS

Braulio A. Mello

Profa. Dr. Braulio Adriano de Mello – UFFS

RESUMO

Atualmente, a utilização da rede IoT é uma realidade dentro da sociedade, consequentemente em razão da ubiquidade gerada por meio de projetos de cidades inteligentes e indústria 4.0. Estas coisas possibilitaram abranger em paralelo novos espaços para uma vasta gama de operacionalidades na Internet, como protocolos de rede, servidores em nuvem, exploração e sanitização de dados, entre outros. Existe um imenso horizonte de possibilidades que surgem através de estudos em redes IoT, justamente, por ser composta de objetos comuns relacionados à vida humana. Em suma, para que haja uma validação nestes projetos, é necessário inspecionar todos os componentes com propósito de assegurar um serviço confiável. Em virtude disso, a presente pesquisa contribui com a extração de dados coletados de testes utilizando o *protocolo MQTT*. Para isto, foi definido o dispositivo Raspberry Pi 4, onde é hospedado o corretor Mosquitto, responsável pelo envio e recebimento das múltiplas mensagens. Em conclusão, realizado o acompanhamento do hardware durante os disparos, identificando características do protocolo e obtendo a relação da carga de processamento, consumo de memória e vazão de dados na rede.

Palavras-chave: Rede IoT. Inspecionar. *Protocolo MQTT*. Rasberry Pi 4. Corretor.

ABSTRACT

Currently, the use of the IoT network is a reality within society, consequently due to the ubiquity generated through smart city projects and industry 4.0. These things made it possible to add in parallel new spaces for a wide range of internet operations, such as network protocols, cloud servers, exploration and data sanitization, among others. There is an immense horizon of possibilities that arise through studies in IoT networks, precisely because it is composed of common objects related to human life. In short, for these designs to be validated, it is necessary to inspect all components to ensure reliable service. As a result, this research contributes to the extraction of data collected from tests using the MQTT protocol. For this, the Raspberry Pi 4 device has been defined, where the Mosquitto broker is hosted, being responsible for sending and receiving multiple messages. In conclusion, monitoring of the hardware during the shots was carried out, identifying characteristics of the MQTT protocol and extracting the relationship of processing load, memory consumption and data execution on the network.

Keywords: IoT networks. Inspect. *MQTT protocol*. Raspberry Pi 4. Broker.

AGRADECIMENTOS

Agradeço em primeiro lugar a Deus, pela possibilidade de vida diante de um momento tão triste para a história do mundo, sobretudo do Brasil, que contabiliza neste ano mais de 660 mil mortos por Covid-19. Agradeço aos meus familiares, em principal minha mãe Ana Cristina Osório Duarte, pelo amparo e apoio diante de todo o período universitário. Agradeço também aos professores, que foram essenciais compartilhando todos os ensinamentos durante esta caminhada, em especial meu orientador Prof. Dr. Marco Aurélio Spohn, que possibilitou o desenvolvimento deste trabalho e outros diversos conhecimentos.

Agradeço aos colegas, que além de trilharem lado a lado almejando o título de bacharel em ciência da computação, foram fundamentais dando suporte psicológico e incentivando em muitos momentos desta caminhada, aos estudantes do pró-Haiti, pelo compartilhamento cultural e amizade, também agradeço aos servidores da Universidade Federal da Fronteira Sul, que de forma ou outra, contribuíram com a formação de um caráter humanizado e uma personalidade mais íntegra. Por fim, agradeço aos meus amigos, que em muitos momentos foram fundamentais para que eu suportasse os problemas que surgiram mediante a todos estes anos de estudos.

LISTA DE ILUSTRAÇÕES

Figura 1 – Arquitetura IoT de seis camadas	24
Figura 2 – Arquitetura endpoints IIoT	25
Figura 3 – Arquitetura IoT-A	26
Figura 4 – Modelo funcional do Broker MQTT	30
Figura 5 – Cabeçalho de mensagem do <i>Protocolo MQTT</i>	31
Figura 6 – Autenticações existentes do <i>Protocolo MQTT</i>	32
Figura 7 – Crescimento esperado nos testes any-to-many, Jmeter	40
Figura 8 – Erros de conexões geradas no primeiro teste	42
Figura 9 – Balance da carga de CPU no primeiro teste	42
Figura 10 – Consumo máximo de memória no primeiro teste	42
Figura 11 – Balance da carga de CPU, segundo teste	43
Figura 12 – Balance da carga de CPU no RPi-Monitor, segundo teste	43
Figura 13 – Consumo máximo de memória no segundo teste	44
Figura 14 – Aviso de erros no segundo teste	44
Figura 15 – Balance da carga de CPU, terceiro teste	45
Figura 16 – Balance da carga de CPU no RPi-Monitor, terceiro teste	45
Figura 17 – Consumo máximo de memória no final do terceiro teste	45
Figura 18 – Resultado de latência dos envios	46
Figura 19 – Resultado de latência das entregas	46
Figura 20 – Consumo máximo de CPU no primeiro teste	52
Figura 21 – Aviso no sinal da rede sem fio	52
Figura 22 – Aviso de erro com exceção do tipo: timeout	53
Figura 23 – Carga máxima de CPU no segundo teste	54
Figura 24 – Variáveis da função principal que modelam o cenário	55
Figura 25 – Alterações para intervalos de 2 segundos	55
Figura 26 – Teste pareado com 25 mil clientes, rede sem fio	56
Figura 27 – Teste pareado, sem fio	56
Figura 28 – Teste pareado, cabeado	57
Figura 29 – Gráfico não normalizado, teste 01	58
Figura 30 – Gráficos não normalizados, teste 2 parte 1	58
Figura 31 – Gráficos não normalizados, teste 2 parte 2	58
Figura 32 – Gráficos não normalizados, teste 3 parte 1	59
Figura 33 – Gráficos não normalizados, teste 3 parte 2	59

LISTA DE ABREVIATURAS

API Application Programming Interface

AWS Amazon Web Service

CoAP Constrained Application Protocol

DCaaS Data Centers as a Service

FCC Federal Communications Commission

GCP Google Cloud Platform

IaaS Infrastructure as a Service

IBM International Business Machines Corporation

ICMP Internet Control Message Protocol

IETF Internet Engineering Task Force

IIoT Industrial Internet of Things

IoT-A Internet of Things Architecture

IoT Internet of Things

IP Internet Protocol

IPv6 Internet Protocol version 6

ISO International Organization for Standardization

ITU Telecommunication Union

JDK Java Development Kit

M2M Machine to Machine

MIT Massachusetts Institute of Technology

MQTT Message Queue Telemetry Transport

OASIS Organization for the Advancement of Structured Information Standards

PaaS Platform as a Service

RFC Request For Comments

RFID Radio-Frequency Identification

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

VM Virtual Machine

XML Extensible Markup Language

XMP Extensible Messaging and Presence Protocol

WIFI Wireless Fidelity

LISTA DE TABELAS

Tabela 1 – Especificações do Raspberry Pi 4 modelo B	38
--	----

SUMÁRIO

1	INTRODUÇÃO	19
1.1	PROBLEMÁTICA	20
1.2	OBJETIVOS	21
1.2.1	Objetivos gerais	21
1.2.2	Objetivos específicos	21
1.3	JUSTIFICATIVA	21
2	REVISÃO BIBLIOGRÁFICA	23
2.1	IOT	23
2.1.1	História	23
2.1.2	Arquitetura IoT	24
2.1.3	Rede IoT utilizando o protocolo MQTT	27
2.1.4	Cloud Computing	28
2.2	<i>PROTOCOLO MQTT</i>	29
2.2.1	Broker	29
2.2.2	Publisher & Subscriber	30
2.2.3	Características de Mensagem	31
2.3	TRABALHOS RELACIONADOS	33
2.3.1	Análise de Desempenho de Brokers MQTT em Sistema de Baixo Custo .	33
2.3.2	Análise de desempenho do BeagleBone Black utilizando o protocolo MQTT	34
2.3.3	Considerações	36
3	METODOLOGIA	37
3.1	FAMILIARIZAÇÃO COM O BROKER MOSQUITTO	37
3.2	AMBIENTE DE TESTES	37
3.2.1	Raspberry Pi 4 Model B	37
3.2.2	Softwares de análise	38
3.3	EXPERIMENTO	38
3.3.1	Configurações	39
3.3.2	Design experimental	40
3.3.2.1	Teste com 1 subscriber e 10000 publishers (QoS 1):	40
3.3.2.2	Teste com 10000 subscribers e 1 publisher (QoS 1):	40
3.3.2.3	Teste com 5000 pares publishers/subscribers e 5000 tópicos (QoS 1):	41
3.4	ANÁLISE DOS RESULTADOS	41
3.4.1	Muitos publicadores para um assinante	41
3.4.2	Muitos assinantes para um publicador	42
3.4.3	Teste com pares de clientes Pub/Sub	44
4	CONCLUSÃO	47
	REFERÊNCIAS	49

1 INTRODUÇÃO

A Internet of Things (IoT) surgiu em 1999 através do Massachusetts Institute of Technology (MIT) e foi conceituado para comunicação Machine to Machine (M2M), ou seja, máquinas se comunicando entre si (sensores, conexão de eletrônicos, controladores, etc) por meio de Bluetooth, Radio-Frequency Identification (RFID), Wireless Fidelity (WIFI) e/ou rede cabeada. Estes dispositivos se comunicam em cima de uma camada de rede, utilizando diversos protocolos e servidores na nuvem que hospedam e disseminam as informações publicadas. Somando a quantidade de conexões suportadas e o baixo consumo de energia dos sensores, a rede IoT cresceu de uma maneira exponencial nos desenvolvimentos de projetos de mobilidade urbana, automação residencial e industrial.

Segundo a Telecommunication Union (ITU) (12), a internet das coisas é “*Uma infraestrutura global para a sociedade da informação, permitindo serviços avançados através da interconexão (física e virtual) de coisas baseadas em tecnologias interoperáveis de informação e comunicação, existentes e em evolução*”. Em uma análise estimativa, a IoT tem probabilidade de crescer com ganhos de 25 a 50 bilhões de dispositivos conectados até 2025. De acordo com Manyika (15), as principais áreas de potencial investimento são a saúde, agricultura e fabricantes que utilizam sensores para otimizar a manutenção de equipamentos e proteger a segurança dos trabalhadores. Em sua análise ascendente, é dimensionado um impacto econômico com potencial total de U\$3,9 trilhões a U\$11,1 trilhões por ano até 2025, o que seria aproximadamente 11% da economia mundial.

Para que haja a possibilidade de uma comunicação neste modelo, é necessário que exista uma arquitetura de rede que permita o envio e o recebimento de todas essas informações. Os pacotes de dados necessitam de identificação para serem entregues em seus destinos, essa comunicação é feita através de protocolos, como Transmission Control Protocol (TCP), User Datagram Protocol (UDP), entre outros. Silva (23) documenta, que tanto o TCP quanto o UDP funcionam em cima do Internet Protocol (IP), o IP é responsável pelo endereçamento das máquinas, garantindo que estas estejam identificadas na rede. Outros protocolos comumente visto em rede IoT, são: protocolo Message Queue Telemetry Transport (MQTT), aplicado em cima do TCP/IP, que utiliza de agentes mensageiros nomeados como publisher e subscriber, estes se comunicam mediante um broker atuante como um servidor; Constrained Application Protocol (CoAP), que é um protocolo de padrão Request For Comments (RFC), normalmente utilizado em aplicação de sensores vide sua simplicidade; Extensible Messaging and Presence Protocol (XMPP), um protocolo aberto baseado em Extensible Markup Language (XML), com padrão Internet Engineering Task Force (IETF), projetado para uso de aplicações de mensagens instantâneas em tempo real.

Completando o cenário da comunicação M2M, na maioria dos casos é fundamental a existência de um gateway, onde atua como responsável pela distribuição dos pacotes de mensagens lançados por cada sensor, podendo também ser nomeado como um tradutor de

protocolos nesta mediação. O gateway no IoT está agregado ao conceito de Fog Computing (Computação de borda ou computação em névoa), pois se encontra exatamente entre os sensores e a Cloud (nuvem) e permitem que mais coisas sejam introduzidas em operações de conexão em escala industrial, isto por suportarem diversos protocolos, ou serem implementados como multiprotocolos devido a aplicação em cenários difíceis da indústria 4.0 e cidades inteligentes.

Entretanto, Silva (23) comenta em sua monografia, que existem desafios para o futuro da IoT nos próximos anos, como a regulamentação, padronização e segurança, isto por motivo de ter muitos protocolos implementados para rede IoT, onde cada protocolo tende a resolver somente um problema, se mantendo restrito quanto à utilização geral. Ele também fomenta, que a estimativa da inclusão do IoT em quase todos os dispositivos móveis ocorra em um curto espaço de tempo, sendo os dispositivos: medidores de estacionamento, pneus, estradas, monitores cardiovasculares, termostatos, sensores de presença, prateleiras de supermercados, assim dizendo, uma imensa quantia de dispositivos conectados com a internet das coisas, gerando uma propensa exploração das vulnerabilidade, problemas de privacidade com a disseminação dos dados sensíveis e segurança com possíveis cenários de ataques. Por estas circunstâncias são necessários regulamentos adequados, que evitem arriscar a segurança dos dispositivos e a validade da IoT do futuro.

Enfim, a partir da análise do crescimento da internet das coisas, surge a proposta visando explorar alguns destes exemplos citados, em principal o *protocolo MQTT*. O cenário selecionado é definido por um dispositivo Raspberry Pi 4 como servidor/broker e um computador fazendo disparos de publicações e/ou assinaturas, para então explorar o gargalo do dispositivo, com intuito de analisar custos de processamento, o índice de utilização de memória, tráfego da rede e as taxas de envio/entrega das mensagens.

1.1 PROBLEMÁTICA

Partindo das demandas provindas da área de telecomunicações, redes e da visível curva de crescimento da internet das coisas, fica claro através da bibliografia a necessidade de examinar cada dispositivo a ser implementado como broker, razões estas que possam garantir a integridade na execução dos possíveis cenários, principalmente, sabendo que estas implementações estão diretamente relacionadas a vida humana. Para validar estas pesquisas, também é importante que o tratamento dos dados sensíveis cumpram os pilares da segurança de informação (confiabilidade, integridade e disponibilidade como previsto na International Organization for Standardization (ISO) 27001), prevenindo acidentes em razão de vulnerabilidades que levam riscos às futuras implementações na IoT. Com base nestas apostas, surgem os desafios nas diferentes opções de hardwares, com distintas arquiteturas e corporações, os variados protocolos de rede e outras características destas coisas, sendo então necessário a realização de uma análise aprofundada dos dispositivos, identificando o melhor a ser adotado em cada cenário simulado. Nestes testes de desempenho, são incluídas algumas métricas para aferir a confiabilidade do dispositivo

escolhido, como por exemplo: o índice de perda de pacotes, vazão de dados, capacidade de processamento, consumo de memória e testes de vulnerabilidades. Permitindo uma estatística aproximada do funcionamento em aplicações reais.

1.2 OBJETIVOS

1.2.1 Objetivos gerais

Neste trabalho, foi escolhido o Raspberry Pi 4 para hospedar o broker Mosquitto, de modo a atuar como responsável na distribuição dos pacotes enviados pelos clientes durante a escalada, coletando os dados de recursos consumidos por cada componente diante os cenários elaborados, bem como, identificar características do *protocolo MQTT* com intuito de assegurar que o hardware, o software e o protocolo possam ser suficientes em contextos de múltiplos clientes.

1.2.2 Objetivos específicos

- Implementar o broker Mosquitto no Raspberry Pi 4;
- Acompanhar a capacidade de processamento, o consumo de memória e a vazão dos dados;
- Fazer testes com rede cabeadas e rede sem fio;
- Testar diferentes cenários (muitos-para-um, um-para-muitos e muitos-para-muitos);
- Analisar os resultados dos testes de cada cenário;
- Avaliar os resultados dos testes de forma conjunta.

1.3 JUSTIFICATIVA

Apresenta como relevante uma análise do Raspberry Pi 4 em resposta ao broker Mosquitto, valendo-se do modelo publisher/subscriber do *protocolo MQTT*, delineando o desempenho do hardware apontado, sobretudo, considerando cenários homogêneos de casos de uso, buscando garantir a capacidade do dispositivo como um gateway estável em rede IoT. Inclusive, apontar a vazão do *protocolo MQTT* em cenários de múltiplas conexões da internet das coisas. Estrutura que atende atualmente milhares de requisições simultâneas (leituras e publicações de mensagens) e assíncronas entre as partes. Por fim, espera-se com a pesquisa identificar resultados positivos e encorajar mais análises com viés ao tema.

2 REVISÃO BIBLIOGRÁFICA

2.1 IOT

Para Kevin (4), especialista britânico do MIT, a definição para IoT é de “*um novo mundo em que os objetos estarão conectados e passarão a realizar tarefas sem a interferência humana*”. Atualmente, sensores são incluídos em todos os lugares, estes sensores convertem dados físicos brutos em sinais digitais que são transmitidos ao seu centro de controle, permitindo monitorar mudanças climáticas, controle de tráfego, análise preditiva de saúde, etc.

De acordo com Suresh et al. (24) o IoT pode estar sendo implementado na automação residencial, conectando a caixa de distribuição de energia a um único smartphone (ou controlador), em que poderia estar sendo operado remotamente; na automação industrial, através de um dispositivo embarcado servindo como gateway, onde em ambos os cenários, não seria necessário um dispositivo local de armazenamento, podendo então um único sensor capturar sinais e processá-los, encaminhando-os a serviços compartilhados na internet; porém, esta arquitetura varia dependendo do contexto de sua aplicação.

2.1.1 História

Desde o surgimento da Internet, estamos conectando coisas na rede e além dos objetos de interação humana direta (computadores pessoais), neste primeiro momento também ocorreram testes de dispositivos eletrônicos conectados, como a cafeteira Trojan Room, sendo citada por Suresh et al. (24) como primeiro dispositivo aplicado nesses cenários. Logo em seguida, em 1990, John Romkey criou uma torradeira que tornou-se o primeiro dispositivo a ser ligada e desligada através da internet e, em 1994, surge o WearCam, inventado por Steve Mann: o Wearcam era composto por um sistema de 64 processadores e foi o primeiro computador vestível a ser projetado, objetivando-se enviar imagens pencigráficas para uma estação base no telhado de seu edifício.

Conforme Suresh et al. (24), em 1997, o sensoriamento já virava uma realidade. Paul Saffo trouxe a descrição sobre os sensores e seu futuro curso de ação para o ano e, em 1999, o termo IoT nascia através de Kevin Ashton, diretor executivo do AutoIDCentre, MIT. Ainda em 1999, eles também viabilizaram a conexão do item global RFID, baseado em sistemas de radares de aviões da 2^a guerra mundial. Após estes eventos, ocorreu um grande salto na comercialização da Internet das Coisas e, em 2000, a LG anunciou o interesse em desenvolver uma geladeira inteligente, que determinaria o momento de seu reabastecimento. Em 2003, o RFID foi implementado em grande escala no exército dos EUA durante o programa Savi e, no mesmo ano, o Walmart varejo implementou RFID em todas as suas lojas distribuídas no mundo.

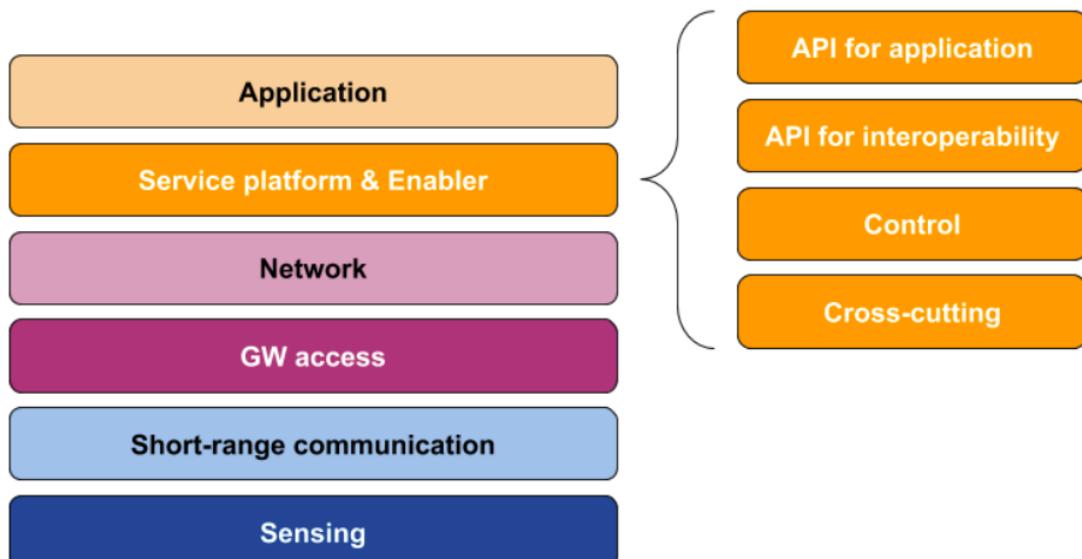
O tema ganhou ainda mais destaque em 2005, com os publicadores The Guardian, Scientific American e Boston Globe, citando em diversos artigos a relevância da IoT e seu curso

futuro, dando visibilidade para posteriormente grupos de empresas lançarem o IPSO Alliance, que possibilitou o uso do Protocolo de Internet (IP) em redes de objetos inteligentes, promovendo ainda mais a IoT. Enfim, em 2011, após a Federal Communications Commission (FCC) aprovar o uso do espectro de espaço em branco, foi lançado o Internet Protocol version 6 (IPv6) que desencadeou um crescimento massivo de interesses neste campo.

2.1.2 Arquitetura IoT

Embora não exista uma representação que esboce uma arquitetura homogênea da IoT, Di Martino et al.(8) incutem que recentemente algumas pesquisas propuseram uma definição de rede IoT, delineando uma maneira de representar a pilha tecnológica, sendo esta a arquitetura de seis camadas proposta por Borgia et al. (Fig. 1).

Figura 1 – Arquitetura IoT de seis camadas



Fonte: Di Martino et al. (8)

A arquitetura proposta é representada na figura 1 e está definida da seguinte forma:

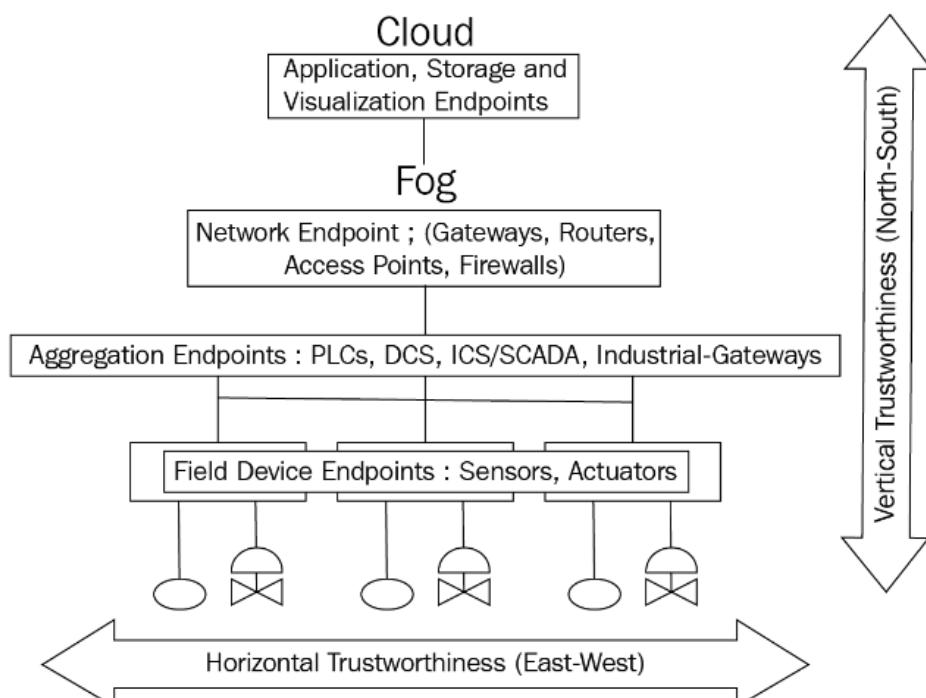
- **Atuadores (Sensing):** Detecção através dos sensores;
- **Conectividade (short-range communication):** Comunicação de curto alcance ou conectividade do sensor através de bluetooth, rede sem fio, RFID, outros;
- **Acesso ao gateway (GW access):** Controlador de sistema embarcado;
- **Conexão de alta largura de banda (Network):** Acesso à rede internet, ocorre de forma cabeados ou de rede sem fio;

- **Tratamento dos dados (Service Platform e enabler):** A quinta camada é o serviço de tratamento dos dados, incluindo softwares e serviços dedicados ao controle oferecidos pelas plataformas;
- **Camada de aplicação (Application):** Serviços de domínio, protocolos para tratar e enviar os dados ou mantê-los.

Em outra visão, Bhattacharjee (6) traz em seu livro que é muito importante arquiteturas confiáveis para Industrial Internet of Things (IIoT), vide ser uma tecnologia-chave na indústria 4.0, que tem revolucionado a maneira como fábricas e organizações industriais se desenvolvem. Focado em uma arquitetura de endpoints (pontos finais ou de ponto a ponto) baseada em risco, o autor trouxe dois princípios importantes, envolvendo motivação e análise de risco, explicadas da seguinte forma:

- **Motivação:** As motivações mais comuns são de proteção, segurança, confiabilidade, resiliência e privacidade dos dispositivos, o que garante integridade e disponibilidade;
- **Análise de risco:** Riscos como explorações, usando dispositivos IoT (bots) como vetores de ataque, risco à garantia de vida e outros.

Figura 2 – Arquitetura endpoints IIoT



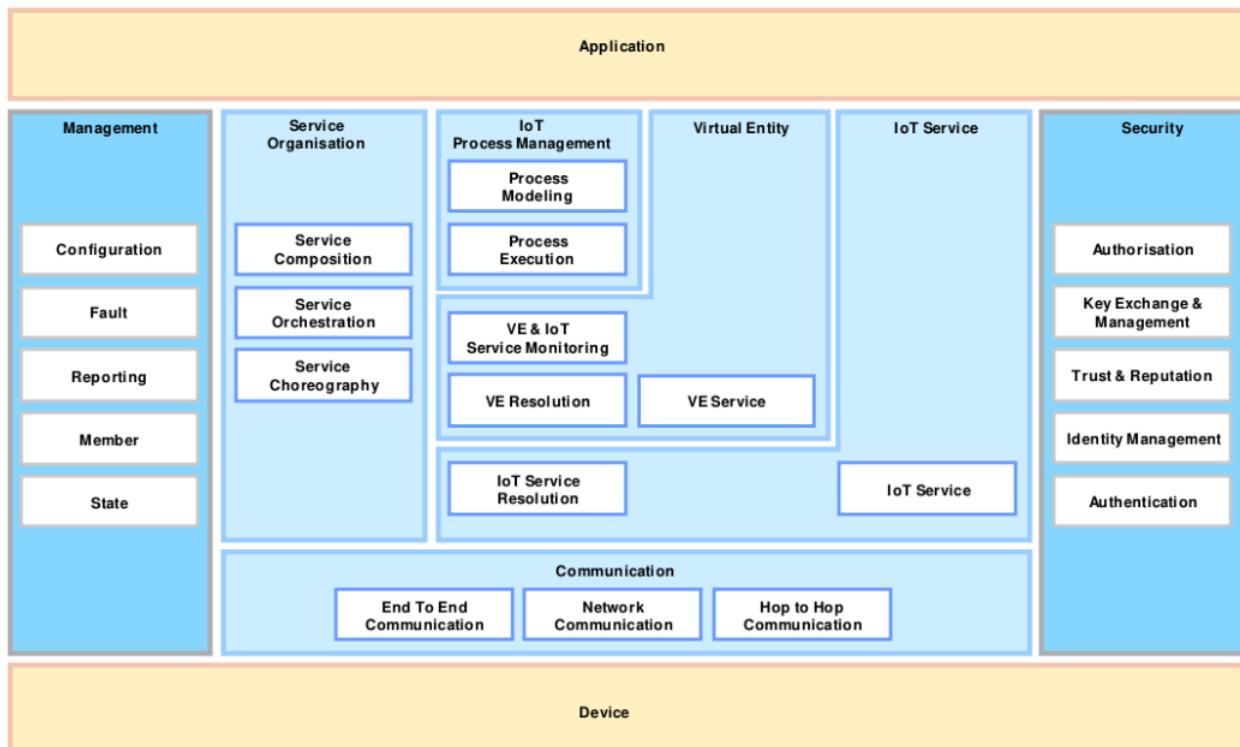
Fonte: Bhattacharjee (6)

Considerando que estas aplicações envolvem custos e podem consumir muitos recursos, a segurança de um sistema corporativo industrial não pode estar totalmente associada a firewalls e

gateways utilizando-se de protocolos de segurança nas topologias convencionais. Bhattacharjee (6) trouxe a estratégia de proteção de endpoints com boa relação no modelo de quatro camadas, fomentando a utilização de criptografias em todos os blocos da arquitetura, para a criação delineada de um ambiente confiável de ponta a ponta, sendo os métodos mais adotados a criptografia a nível de hardware, protocolos criptografados, armazenamentos criptografados e técnicas de isolamento (Blockchain e Containers).

No IIoT, os desafios encontrados também são devido aos dados atravessarem limites organizacionais, os acessos remotos acontecem de interfaces terminais de outros domínios e os endpoints dos dispositivos vêm de uma variedade de fornecedores. No entanto, é possível simplificar o projeto geral de segurança, utilizando recursos de nível de Application Programming Interface (API) consistentes em todos os terminais, tanto na borda quanto na nuvem, além da adesão de arquiteturas modulares e escaláveis. A figura 2 mostra vários pontos de verificação no ciclo de vida do modelo endpoints.

Figura 3 – Arquitetura IoT-A



Fonte: Bassi et al. (5).

Existem também arquiteturas que esboçam outros cenários na IoT além dos padrões abstratos, como a Internet of Things Architecture (IoT-A) que se encontra representada na figura 3. Essa arquitetura foi desenvolvida pela fundação da comissão europeia, e inclui capitalização sobre os benefícios de plataformas orientadas para o consumidor, incluindo serviços, hardware e software. Para um melhor entendimento, o European Lighthouse Integrated Project (20) apresentou um modelo arquitetônico construído durante três anos (de setembro de 2010 até

novembro de 2013), que tinha como propósito criar uma arquitetura como modelo de referência, junto a definição de um conjunto de blocos de construção chave, vistos como fundamentais para promover uma futura Internet das Coisas.

Partindo de um paradigma experimental, o IoT-A combinou o raciocínio de cima para baixo sobre os princípios arquitetônicos, com as diretrizes de design com simulação e prototipagem focados em consequências técnicas do projeto. A arquitetura possui as seguintes camadas de abstração: visão física; visão de contexto; visão funcional; visão de informação e visão de implementação. O modelo proposto prevê resultados de interoperabilidade (e.g., delinear conceitos abstratos para o projeto e seus protocolos, interfaces e algoritmos) e mecanismos de eficiência na integração de serviços para internet do futuro, infraestrutura de resolução, descobertas escalonáveis de captação de recursos da IoT (entidades reais e associações), entre outros.

2.1.3 Rede IoT utilizando o protocolo *MQTT*

A adoção do *protocolo MQTT* na IoT cresceu bastante nos últimos anos, o que pode justificar essa adoção é o fato de ser um protocolo leve, que suporta mensagens significantemente grandes e faz isto através de um broker. A complexidade pode ser considerada simples em alguns casos de uso, pois utilizando de um broker do MQTT em uma Virtual Machine (VM) já é suficiente para uma comunicação existir. Essa desenvoltura permite que a execução de projetos residenciais sejam desenvolvidos por um simples entusiasta, tendo-se como exemplo típico o monitoramento de luzes ou câmeras em partes de uma casa.

Em suma, esta simplicidade também é encontrada na literatura. Durante as pesquisas para realização deste trabalho, foram encontrados artigos, monografias, livros em diferentes áreas da ciência, abordando o desenvolvimento de rede IoT em paralelo com o *protocolo MQTT*. Um exemplo que chamou a atenção foi a monografia de Muenchen (17), que teve como objetivo a aplicação de um sistema capaz de detectar fumaças e diferentes gases inflamáveis através de uma placa ESP-32 e um sensor de fumaça MQ2. O software implementado tinha funções de hibernação, visando o baixo consumo de energia, quando esse não estava em cenários de ameaça. Em síntese, este projeto idealizou o encaminhamento das publicações de indícios de incêndios de casas de show, bares, lojas de varejo e outros serviços de atendimento.

Nesta pesquisa, o MQTT foi o protocolo adotado para o envio das mensagens para nuvem, o que garantiria um melhor retorno do serviço de segurança pública, razões que podem ser justificadas pelo fato das mensagens serem entregues em tempo real, bastando somente um computador conectado a internet, assinar os tópicos e aguardar os alertas que forem publicados. Muenchen (17), também comenta, a necessidade de uma configuração de geolocalização nos dispositivos(gateways), tencionando a assertividade durante as buscas de ocorrências. Enfim, o *protocolo MQTT* é visto como um padrão de rede IoT e ao pensarmos nas inúmeras possibilidades de aplicações, surge um horizonte diante dos olhos visando uma melhor qualidade de vida.

2.1.4 Cloud Computing

Segundo a página PET de Sistemas de Informação da Universidade Federal de Santa Maria (UFSM)(18), o termo Cloud Computing foi criado em 1997 pelo professor Ramnath Chellappa, e foi utilizado para se referir a algo que está no ar, fazendo uma relação com sistemas que não estão hospedados em servidores físicos locais, mas que estão externos, acessíveis via Internet.

Acabando com parte dos desafios que o IoT ainda tinha em 2000, a Cloud Computing chega ao amplo público com a Salesforce e com a gigante Amazon, estes primeiros modelos de serviços consistiam em aluguéis de computadores virtuais (VMs), que disponibilizavam serviços e aplicativos da plataforma. Logo em seguida, vieram também as nuvens Google e Microsoft, que ofertavam serviços adicionais com custos menores. Entretanto, somente em 2006 com a pioneira Amazon Web Service (AWS), esse modelo de serviço se consolidou em plataformas focadas a manipulação de computadores compartilhados. Logo em seguida, também surgiram a Google Cloud Platform (GCP)(2008), Watson(2010) International Business Machines Corporation (IBM) e Azure(2010) Microsoft.

Deve ser levado em consideração a grande manipulação de dados da IoT, um dos pilares do modelo compartilhado em nuvem, tendo em vista que os dispositivos de baixo custo (sensores, controladores, etc.) não possuem armazenamento em grande escala, sendo fundamental que a nuvem sirva de armazenamento, hospedagem e replicação das mensagens publicadas por estes controladores. Porém, existem inúmeros serviços ofertados nestas plataformas, onde é importante também salientar características de integridade, confiabilidade, flexibilidade e escalabilidade dos dados.

Entre os serviços destacados para rede IoT, estão o armazenamento dos dados na própria plataforma (hospedagem), sanitização dos dados e a virtualização dos dados nas VMs. Ao planejarmos a virtualização do MQTT, um exemplo seria a máquina operando como broker na nuvem, encaminhando as mensagens recebidas pelo protocolo de rede, evitando a necessidade de servidores físicos locais.

Segundo o setor de dados da Associação Brasileira de Empresas de Software (ABES) (1), os gastos de Infrastructure as a Service (IaaS) somados com os gastos de Platform as a Service (PaaS) em nuvem pública no Brasil devem atingir em 2021 cerca de US\$ 3,0 bilhões, o que representa um crescimento da adoção em 46,5% em relação a 2020. O modelo de nuvem privada também cresceu em um bom ritmo, totalizando US \$614 milhões no ano, crescimento devido ao Data Centers as a Service (DCaaS), que avançou 15,5% comparado à 2020.

2.2 PROTOCOLO MQTT

Conforme a página oficial do *protocolo MQTT*(16), este protocolo foi criado em 1999 pelo Dr. Andy Stanford-Clark da IBM em conjunto com Arlen Nipper da Arcom (agora Eurotech). O atual CEO da Huawei, Michael Yuan (27), trouxe que o *protocolo MQTT* foi inicialmente implementado com o objetivo de atuar em cenários heterogêneos da IoT e, nesse primeiro momento, era vinculado desde a aplicações em pipelines de petróleo a satélites e, logo em seguida, tornou-se um padrão para comunicação na rede IoT.

Em um artigo publicado pela IBM developer, Yuan (27) comenta que o *protocolo MQTT* é um protocolo de mensagem com suporte de comunicação assíncrona entre as partes (emissor e receptor) e, portanto, é escalável em ambientes de redes não confiáveis. O MQTT não possui enfileiramento de mensagens, mas adota um modelo de publicação e assinatura mediado por um servidor, operando em cima do TCP/IP. No final de 2014, o MQTT tornou-se oficialmente um padrão aberto da Organization for the Advancement of Structured Information Standards (OASIS) e assim possibilitou suporte a linguagens de programação populares, o que garantiu implementações de softwares livres.

A página oficial do *protocolo MQTT*(16) traz uma recomendação relacionada a segurança das trocas de mensagens; entretanto, na aplicação com esses padrões o usuário deve informar o nome e uma senha com um pacote MQTT, onde a criptografia na rede pode ser tratada com Secure Sockets Layer (SSL), atual Transport Layer Security (TLS), a documentação também reconhece modelos deseguranças adicionais de APIs, embora não embutidos no protocolo a fim de mantê-lo simples e leve.

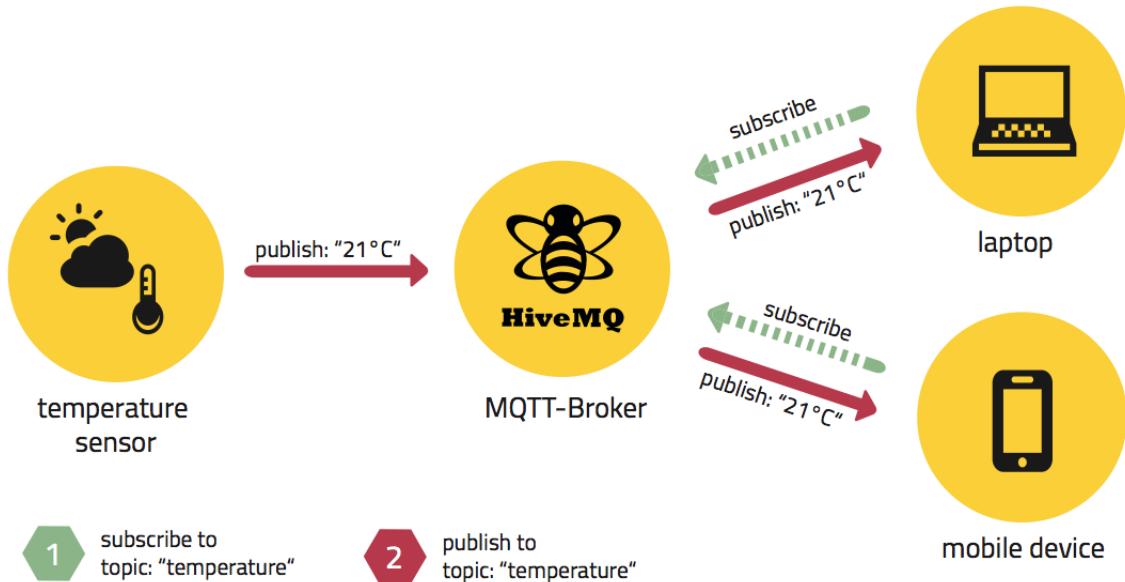
2.2.1 Broker

O broker opera como um servidor e, como o MQTT opera com um modelo de publicação e assinatura, é fundamental que todas as execuções sejam mediadas pelo servidor. O broker recebe as mensagens dos clientes publicadores e, em seguida, as envia para os clientes assinantes dos tópicos publicados, sendo estes os únicos identificadores comuns entre as partes. Segundo Yuan (27), um cliente tanto pode ser um sensor de IoT em campo ou um aplicativo em um datacenter que processa dados de IoT.

O *protocolo MQTT* define dois tipos de entidades na rede, um mensageiro broker e inúmeros clientes (publishers e subscribers). A figura 4 mostra o exato instante em que um publicador do tópico "temperature" envia mensagens a dois clientes assinantes através de um broker.

- O cliente conecta-se ao broker. Ele pode assinar qualquer "tópico" de mensagem no broker. Esta conexão pode ser uma conexão TCP/IP simples ou uma conexão TLS com criptografia para mensagens sensíveis. O evento pode ser observado no instante 1 (Fig. 4), subscribe to topic: "temperature".

Figura 4 – Modelo funcional do Broker MQTT



Fonte: Götz (11)

- O cliente publica as mensagens em um tópico, enviando a mensagem e o tópico ao broker. Instante 2 (Fig. 4), publish to topic: "temperature", é possível notar na aresta do gráfico, que a mensagem do tipo publish possui o conteúdo "21 °C".
- Em seguida, o broker encaminha a mensagem a todos os clientes que assinam esse tópico.

2.2.2 Publisher & Subscriber

Para que haja a comunicação entre as pontas, o publisher e o subscriber devem estar conectados a um broker. Em geral, os clientes não têm um endereço específico padrão e não há conexão direta entre eles, apenas um tópico chave. Existem três cenários que podem ser implementados, representados na lista abaixo:

- Any to any: Um publicador para um assinante.
- Any to many: Um publicador para muitos assinantes ou muitos publicadores para um assinante.
- Many to many: Muitos publicadores para muitos assinantes.

2.2.3 Características de Mensagem

Atualmente, existem diversos brokers do *protocolo MQTT* e cada um tem um modo de operar diferente. O broker Mosquitto é um destes, apresentando por definição um tópico somente de leitura, com o nome `$SYS`, publicando todos os logs de informação do serviço neste tópico, os quais são: timestamp; uptime; status dos clientes (ativados ou desativados); históricos de mensagens recebidas e todas as mensagens na memória.

O modelo de comunicação do *protocolo MQTT* é baseado em solicitação e resposta, onde o pedido da autenticação ocorre através da solicitação do cliente (request) e uma resposta do servidor (response). De acordo com a especificação do MQTT (14), o cabeçalho fixo das mensagens é composto por dois bytes, como visto na figura a seguir.

Figura 5 – Cabeçalho de mensagem do *Protocolo MQTT*

bit	7	6	5	4	3	2	1	0
byte 1	Message Type			DUP flag	QoS level		RETAIN	
byte 2	Remaining Length							

Fonte: IBM; Eurotech (14)

A figura 5 representa a estrutura do cabeçalho, onde o byte 1 é fragmentado em quatro partes: 4 bits (7 à 4) reservados para o tipo de mensagem (Message Type); DUP flag no bit 3; dois bits para QoS level (2 e 1); e por fim o RATAIN flag no bit 0.

O byte 2 representa o número de bytes restantes na mensagem atual, incluindo dados no cabeçalho variável e a carga útil (payload). O esquema de codificação de comprimento variável, usa um único byte para representar mensagens de até 127 bytes. A especificação do protocolo garante configurações de recálculos definidos para mensagens maiores, com uma carga máxima de 256MB.

- **DUP flag (Duplicate Delivery):** Persistência na entrega, é o sinalizador que deve ser definido quando o cliente ou servidor tentar entregar novamente mensagens do tipo PUBLISH, PUBREL, SUBSCRIBE ou UNSUBSCRIBE, só ocorre quando o QoS level é maior que zero (0).
- **QoS level(Quality of Service):** Definição do tipo de garantia de entrega que o cliente pretende para uma mensagem e possui três níveis de seleção; O nível 0 consiste em não receber garantias (fire and forget); o nível 1 pelo menos uma vez a entrega é confirmada (Acknowledged delivery); e o nível 2 exatamente uma vez a entrega é garantida (Assured delivery).
- **RETAIN flag:** Garante a persistência de uma mensagem, ou seja, quando um publicador envia uma mensagem ao broker com a flag retain ativada, esta mensagem será retida no

broker e toda vez que um assinante se conectar ao tópico que contém a mensagem, ela será lançada.

Figura 6 – Autenticações existentes do *Protocolo MQTT*

Mnemonic	Enumeration	Description
Reserved	0	Reserved
CONNECT	1	Client request to connect to Server
CONNACK	2	Connect Acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish Acknowledgment
PUBREC	5	Publish Received (assured delivery part 1)
PUBREL	6	Publish Release (assured delivery part 2)
PUBCOMP	7	Publish Complete (assured delivery part 3)
SUBSCRIBE	8	Client Subscribe request
SUBACK	9	Subscribe Acknowledgment
UNSUBSCRIBE	10	Client Unsubscribe request
UNSUBACK	11	Unsubscribe Acknowledgment
PINGREQ	12	PING Request
PINGRESP	13	PING Response
DISCONNECT	14	Client is Disconnecting
Reserved	15	Reserved

Fonte: IBM; Eurotech (14)

A figura 6 representa diferentes tipos de autenticações e respostas geradas pelo broker do *protocolo MQTT*. As mensagens reservadas são:

- **CONNECT** - Quando uma conexão de soquete TCP/IP solicita conexão de um cliente para um broker, uma sessão de nível de protocolo é criada utilizando o fluxo Connect.
- **CONNACK** - O servidor envia uma mensagem em resposta a uma solicitação Connect de um cliente, podendo assumir valores como: Conexão confirmada; Conexão recusada: versão de protocolo inaceitável; Conexão recusada: identificador rejeitado; Conexão recusada: servidor indisponível; Conexão recusada: nome de usuário ou senha incorretos; Conexão recusada: não autorizada; Reservado para uso futuro.
- **PUBLISH** - Está diretamente associada a um único tópico (assunto ou canal) e quando enviada a um servidor é distribuída aos assinantes interessados neste mesmo tópico.
- **PUBACK** - É a resposta de um servidor broker para uma Publish com QoS de nível 1.

- *PUBREC* - É a resposta de um servidor broker para uma Publish com QoS de nível 2.
- *PUBREL* - É a resposta de um publicador a uma Pubrec do servidor, ou a resposta do servidor a uma Pubrec de um assinante, é a terceira mensagem no fluxo de QoS em nível 2.
- *PUBCOMP* - É a resposta do servidor a uma Pubrel de um publicador ou a resposta de um assinante de uma mensagem Pubrel do servidor, sendo a última mensagem no fluxo do protocolo em QoS 2.
- *SUBSCRIBE* - Permite que um cliente registre interesses em um ou mais tópicos com o servidor broker e, por consequência, também é possível especificar o nível de QoS em qual o assinante deseja receber as mensagens publicadas no tópico.
- *SUBACK* - Representa uma mensagem enviada pelo servidor ao cliente para confirmar o recebimento de uma mensagem Subscribe; a Suback contém uma lista de níveis de QoS concedidos.
- *UNSUBSCRIBE* - Representa uma mensagem enviada do cliente ao servidor para cancelar a assinatura de tópicos nomeados.
- *UNSUBACK* - Representa a resposta enviada pelo servidor ao cliente confirmado o recebimento de uma mensagem de cancelamento Unsubscribe.
- *PINGREQ* - A mensagem PINGREQ representa um "você está vivo?" enviada de um cliente conectado ao servidor, também conhecida como PING, keepalive, heartbeat, entre outros.
- *PINGRESP* - Representa a resposta enviada por um servidor a uma mensagem PINGREQ e significa "sim, estou vivo".
- *DISCONNECT* - Representa uma mensagem enviada do cliente para o servidor para indicar que ele está prestes a fechar sua conexão TCP/IP de forma limpa e segura.

2.3 TRABALHOS RELACIONADOS

2.3.1 Análise de Desempenho de Brokers MQTT em Sistema de Baixo Custo

Neste trabalho, a pesquisa teve foco em analisar diferentes brokers do *protocolo MQTT* e foi desenvolvido pelos integrantes do grupo de Redes de Computadores, Engenharia de Software e Sistemas(GREAT) da Universidade Federal do Ceará (UFC), Andrei Torres, Atslands Rocha e José Neuman de Souza. O artigo foi publicado no 34º congresso da Sociedade Brasileira de Computação, em 2016, e apresentado no XV Workshop em Desempenho de Sistemas Computacionais e de Comunicação. (25)

Durante este projeto, foi utilizado o Raspberry Pi 2 como gateway empregando os brokers Apollo (Java), Mosca, Ponte (Javascript), Mosquitto (C) e o broker eMQTT (Erlang).

O design experimental aplicado foi definido na seguinte ordem:

- Iniciar a captura dos dados;
- Aguardar 60 segundos;
- Iniciar 200 conexões de subscribers a cada 30 segundos;
- Ao atingir a carga máxima de 10.000 subscribers, mantê-la durante 180 segundos;
- Encerrar 25 conexões a cada 1 segundo;
- Aguardar 5 minutos após encerrar a última conexão;
- Encerrar captura de dados.

Os resultados obtidos foram os seguintes: durante a comparação do processamento, o eMQTT implementado em Erlang teve a maior carga de processamento, provavelmente devido ao foco de aplicação em clusterização e escalabilidade, já o Mosca e o Ponte, implementados em Nodejs (Javascript) (importante salientar que o Ponte implementa o MQTT a partir do Mosca, sendo normal uma similaridade nas análises) obtiveram desempenhos equivalentes com o Mosquitto em C, ambos consumindo menos de 25% do processamento total do dispositivo. Com referência ao consumo de memória, destaca-se novamente o Mosquitto, consumindo somente 9,5 MB comparado ao consumo de memória de 275 MB do Mosca, Ponte e do terceiro colocado eMQTT. No que corresponde a rede, foi analisado o índice de mensagens entregues, o eMQTT alcançou o primeiro lugar, seguido do Ponte e Mosca; entretanto, o Mosquitto só atingiu 68% dos pacotes entregues, aparentemente por ter atingido um pico máximo após 15 minutos (em torno de 4000 pacotes). O Apollo não obteve estabilidade em nenhuma fase dos testes e por isto não foi citado durante esta relação, acredita-se que por razões de memória.

2.3.2 Análise de desempenho do BeagleBone Black utilizando o *protocolo MQTT*

Este trabalho teve como objetivo analisar o desempenho do BeagleBone Black operando como broker do *protocolo MQTT* e foi desenvolvido por Gabriel Augusto Veiga Rodrigues, tendo como requisito à obtenção parcial do título de Bacharel em Ciência da Computação da Universidade Tecnológica Federal do Paraná (UTFPR) (21).

Para execução dos testes, foi utilizado o dispositivo BeagleBone Black como gateway e implementado no dispositivo o broker Mosquitto. Foram realizadas três fases de testes, contendo um total de 6000 publicadores em cada teste.

Cenário experimental do teste 1 (objetivo: alcançar o nível máximo de carga no menor tempo):

- Iniciar coleta de desempenho;
- Aguardar 30 segundos;
- Iniciar com 0 conexões e aumentar 1200 conexões a cada 15 segundos;
- Ao atingir a carga máxima de 6 mil conexões, mantê-las durante 60 segundos;
- Encerrar 60 conexões a cada 1 segundo;
- Encerrar teste.

A duração do teste foi de 4 minutos e 33 segundos.

Cenário experimental do teste 2 (objetivo: uso normal do gateway mantendo um balanço no total e tempo de conexões):

- Iniciar coleta de desempenho;
- Aguardar 30 segundos;
- Iniciar com 0 conexões e aumentar 300 conexões a cada 30 segundos;
- Ao atingir a carga máxima de 6.000 conexões, mantê-la durante 90 segundos;
- Encerrar 30 conexões a cada 1 segundo;
- Encerrar teste.

A duração do teste foi de 16 minutos e 28 segundos.

Cenário experimental do teste 3 (objetivo: teste de larga escala de tempo, tanto para alcançar a carga máxima como para mantê-la ativa):

- Iniciar coleta de desempenho;
- Aguardar 30 segundos;
- Iniciar com 0 conexões e aumentar 150 conexões a cada 30 segundos;
- Ao atingir a carga máxima de 6.000 conexões, mantê-la durante 180 segundos;
- Encerrar 30 conexões a cada 1 segundo;
- Encerrar teste.

A duração do teste foi de 32 minutos e 58 segundos.

Os resultados alcançados foram: para os três testes o BeagleBone obteve um carga alta de processamento, chegando a utilizar todo o processamento, sendo que nos primeiros 30 segundos houve uma estabilidade variando de 5% à 20% do uso, porém, a partir da chegada do primeiro pacote, o dispositivo utilizou 100% do processamento; No consumo de memória, a distribuição foi da seguinte forma: no primeiro teste, o pico máximo foi de 87MB; no segundo teste, o uso de memória teve pico de 97MB alcançados no momento 151 s à 170 s e, no último teste, alcançou o pico máximo de 100MB, onde o consumo aumentava em média 14MB em cada 30 segundos de evento. Por fim, nos dados relacionados à perda de pacotes não houveram perdas a serem contabilizadas em nenhum dos eventos executados.

2.3.3 Considerações

Os trabalhos que foram adotados nesta relação, coube pela similaridade na construção dos modelos elaborados, durante a análise dos trabalhos relacionados, foi observado que ainda existem cenários a serem abordados, tal como: teste pareado (many-to-many); documentação de configurações de permissões de ilimitadas conexões no broker Mosquitto; ampliação de análises de erros gerados pelo software Jmeter, este buscando identificar as exceções de cada caso encontrado, vide serem caracterizados pelo *protocolo MQTT* em serviço mediante o software; inclusão da métrica de latência para identificação da eficiência do protocolo no contexto arquitetado e enfim, considerar a capacidade atual do dispositivo adotado.

3 METODOLOGIA

3.1 FAMILIARIZAÇÃO COM O BROKER MOSQUITTO

O Mosquitto é um agente de mensagens de software livre e foi projetado pela Eclipse Foundation (9). Por ser leve e suportar diversas versões do *protocolo MQTT*, foi o mais adequado para instalar no sistema embarcado. Durante os meses de aprendizagem, o broker Mosquitto se demonstrou estável em diferentes aplicações, oferecendo as seguintes funcionalidades: a implementação de websockets baseado no projeto Paho, fácil instanciação em máquinas virtuais na nuvem e em aplicações no Android.

Em um terminal Linux, pode-se testar o broker Mosquitto seguindo os seguintes passos:

- Download e instalação do módulo Mosquitto disponível no website mosquitto.org.
- O comando `mosquitto` executa o broker no computador local e é possível utilizar a flag `-d` para executar em segundo plano, então no terminal linux digite `$ mosquitto -d`
- Em outra janela do terminal, é possível usar o comando `mosquitto_sub` para conectar-se ao broker local e assinar um tópico, para isso digite `$ mosquitto_sub -t 'temperature'`
- Em uma outra janela do terminal, é possível usar o comando `mosquitto_pub` para conectar-se ao broker local, em seguida, publicar uma mensagem em um tópico, sendo assim digite `$ mosquitto_pub -t 'temperature' -m '21°C'`

Após estes passos, na janela em que a flag `mosquitto_sub -t 'temperature'` foi executada, irá aparecer na tela a mensagem `21 °C`, o que demonstra que o broker entregou a mensagem corretamente.

3.2 AMBIENTE DE TESTES

Foi utilizado um notebook com sistema operacional Linux Ubuntu versão 20.04 (processador i7 e 8GB de memória) para a inicialização dos clientes através da rede sem fio e, em conjunto, um Raspberry pi 4 para implementar o broker Mosquitto, este operando em rede cabeadas.

3.2.1 Raspberry Pi 4 Model B

A página oficial do dispositivo, recomenda a utilização do sistema operacional Raspbian (Raspberry Pi OS) e neste caso, foi utilizado a versão Raspberry Pi OS Bullseye x86, baseado em Debian, qual incorpora um kernel linux (kernel versão 5.10). Como armazenamento, foi utilizado um cartão de memória Adata (2), com capacidade máxima de 32G. As principais especificações do Raspberry Pi 4 estão detalhadas na tabela 1.

Tabela 1 – Especificações do Raspberry Pi 4 modelo B

Processador	Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
Memória	8GB LPDDR4-3200 SDRAM (depending on model)
Placa WIFI	2.4 GHz and 5.0 GHz IEEE 802.11ac wireless
Placa de rede	Gigabit Ethernet até 1.000 Mb/s
Bluetooth	5.0, BLE
Portas USB	2 USB 3.0; 2 USB 2.0.
Audio	4-pole stereo audio and composite video port
Vídeo	2 × micro-HDMI ports
Gráficos	OpenGL ES 3.1, Vulkan 1.0
Pinos	40 pin GPIO header
Armazenamento	Micro-SD card slot for loading operating system and data storage
Voltagem	5V DC via USB-Connector (min. 3A*), 5V DC via GPIO header(min. 3A*)

Fonte: Pi Foundation (19)

3.2.2 Softwares de análise

Foram definidos dois softwares para auxiliar nas cargas de publicações e assinaturas. Nos testes any to many, foi utilizado o Apache JMeter (3), que é um software de código aberto focado em escalabilidade e é desenvolvido em Java. Para análise da latência da entrega dos pacotes no teste many to many, foi utilizado o projeto MQTT broker latency measure tool (13), desenvolvido em Golang e está disponível no Github.

Durante o monitoramento do dispositivo, foi utilizado o RPi-Monitor (22) e o Collectl (7). O RPi-Monitor permite o acompanhamento em tempo real através de gráficos com uma atualização de 10 segundos e carga mínima de 1 minuto. Já o Collectl obtém atualizações em 1 segundo e foi fundamental para as métricas da pesquisa. Ambas ferramentas permitem o monitoramento de CPU, memória e rede.

3.3 EXPERIMENTO

Ao delinear o experimento, os testes foram separados em três partes: dois deles any to many, com o primeiro tendo muitos publicadores para um assinante e, o segundo, um publicador para muitos assinantes; por fim, um teste many to many com muitos publicadores para muitos assinantes. Pode-se considerar este último um dos desafios da pesquisa, por incluir as métricas da taxa de envio/entrega do *protocolo MQTT*. Os testes no software Jmeter objetivaram uma escala regular de tempo, com intervalos na escalada definidos com metade do valor total estabelecido como tempo de espera, gerando disparos de pequenas quantidade de clientes (no máximo 80 conexões) em um curto intervalo de tempo, fixo em 5 segundos; No último teste, o crescimento objetivou uma escala maior de tempo, considerando uma quantidade menor de clientes, definido em 10 conexões, com um intervalo estabelecido em 2 segundos.

3.3.1 Configurações

Os sistemas operacionais Ubuntu e Respbian são baseados em Debian e possuem um limite de arquivos a serem abertos, definidos por padrão em no máximo 1024 arquivos simultâneos. Para que os testes fossem possíveis, uma alteração no arquivo *system.conf* foi fundamental. Neste caso, a rota utilizada para encontrar o arquivo foi */etc/systemd/system.conf*. O parâmetro *DefaultLimitNOFILE* foi definida em 65535 nos dois dispositivos. Entretanto, o broker Mosquitto tem um limite rígido de conexões, 4096, e em sua documentação, para ajustar este limite, deve-se abrir o arquivo *mosquitto.conf* que se encontra em */etc/mosquitto/mosquitto.conf* e incluir a variável *max_connections -1*: o valor -1 consiste em ilimitadas conexões. No manual do *mosquitto.conf* consta que é aconselhado uma alteração no arquivo *limits.conf*, encontrado no diretório */etc/security/limits.conf*. Neste contexto, inseriu-se duas linhas que reforçam os limites rígidos e também de software do sistema, ambos definidos em 65535. Enfim, para que as atualizações fizessem efeito, tornou-se necessário reiniciar o daemon do sistema e também o serviço Mosquitto.

O Jmeter e o Collectl funcionam em cima do servidor Apache (10); logo, foi necessário habilitar o serviço antes da execução dos testes. O Jmeter, por ser implementado totalmente em Java, torna-se necessária a instalação da Java Development Kit (JDK). Para finalizar as configurações, foi necessário incluir a biblioteca *mqtt-jmeter* (26) na raiz do projeto Apache Jmeter. A característica que chamou a atenção nessa biblioteca foi a implementação do cliente a partir do broker HiveMQ.

O RPi-monitor plota no eixo Y a capacidade máxima do processador e no eixo X o tempo de execução do teste. Os dados lançados no eixo Y variam em relação a seleção de tempo no eixo X. Em razão, fez-se necessário a normalização do vetor Y (de 0 até 1,5 Ghz) e também, do valor selecionado de um determinado ponto (número racionais) para porcentagens. Curvas de crescimento: a linha amarela tem uma atualização de minuto a minuto, demonstrando parcialmente o crescimento da curva em uma média de um minuto de carga; a linha azul representa uma atualização a cada 5 minutos; a linha vermelha corresponde a uma atualização de intervalos de 15 minutos, trazendo uma curva da média total. Em sua documentação, não há referência de implementação direcionada a apenas um núcleo do processador e o Mosquitto opera em apenas um núcleo, destacando-se que o Raspberry Pi 4 possui quatro núcleos; em consequência, foi considerado um desvio padrão no crescimento da curva, assumindo-se uma dispersão de até 0,132 entre as máximas dos gráficos e os log do Collectl; ou seja, $\approx 13\%$. Embora o erro seja relativamente alto, não afetou nas métricas da pesquisa devido a contribuição do software Collectl.

3.3.2 Design experimental

3.3.2.1 Teste com 1 subscriber e 10000 publishers (QoS 1):

- Assina o tópico a ser publicado;
- Inicia o teste e espera 20 segundos;
- Envia 40 publicações;
- Escala 80 novas publicações a cada 5 segundos;
- Assim que finalizar a escala, espera 120 segundos;
- Desconecta 60 clientes por segundo e encerra o teste.

3.3.2.2 Teste com 10000 subscribers e 1 publisher (QoS 1):

- Inicia o teste;
- Espera 20 segundos;
- Envia 40 assinaturas;
- Escala 80 novas assinaturas a cada 5 segundos;
- Ao finalizar todas assinaturas, durante a espera de 120 segundos, gera uma publicação no tópico assinado;
- Desconecta 60 clientes por segundo e encerra o teste.

Figura 7 – Crescimento esperado nos testes any-to-many, Jmeter



Fonte: O autor

3.3.2.3 Teste com 5000 pares publishers/subscribers e 5000 tópicos (QoS 1):

- Inicia o teste;
- Gera 10 Conexões/SUB a cada 2 segundos, cada subscriber mostra interesse em um tópico específico e, assim que finalizar 5000 assinaturas, mantém a execução;
- Realiza 10 Conexões/PUB com intervalo de 2 segundos, cada cliente publica 10 vezes em um tópico específico;
- Assim que finalizar 50000 publicações, espera 3 segundos e desconecta os 5000 assinantes aleatoriamente;
- Imprime a taxa de sucesso.

3.4 ANÁLISE DOS RESULTADOS

3.4.1 Muitos publicadores para um assinante

Durante o primeiro teste, foi observado uma instabilidade iniciando em 6000 publicadores, provavelmente causada pela qualidade do serviço (QoS 1: inclusão da DUP flag e persistência na entrega). Este gargalo gerou uma ampliação no serviço do Collectl, como mostra a figura 8: o aplicativo imprimiu duplicadamente a cada segundo, sendo uma linha referente ao serviço Mosquitto e outra relacionando os erros ocasionados enquanto o teste executava. Portanto, o log trouxe um total de 161 erros (ou 1,61% das publicações) no Internet Control Message Protocol (ICMP), em grande parte sendo exceções do tipo: tempo esgotado (timeout); fato curioso foram os erros de pacotes UDP, uma vez que, o MQTT infiltra através do TCP, sendo assim, estes ocorreram em razão do gargalo gerado, afetando outros serviços em execução durante o teste, não foi possível identificar o motivo dos 3 erros de IP, embora acredita-se que foram gerados em razão de falhas da rede durante um curto tempo do evento (sinal da rede reduzido, enfraquecido ou até mesmo sem funcionamento). A variação dos dados foi acima dos demais testes, porém, foi considerado, justamente para buscar identificar cada erro caracterizado pelo protocolo *MQTT*.

O processamento variava, em média, mais de 20% do uso total a cada segundo. Durante esta oscilação, o Collectl chegou a apontar uma carga máxima de 98% do recurso (Fig. 9, lado esquerdo): o pico ocorreu no final do teste, um pouco antes das 10000 desconexões. O RPi-Monitor apresentou uma variação de carga entre 0,9 Ghz e 1,2 Ghz no teste; $\approx 75\%$ da capacidade total, a figura 9 à direita, esboça esta variação durante o evento. A duplicação do Collectl contabilizou também um aumento significativo do uso da memória: no ponto inicial, a memória consumida foi de 88MB (em consequência dos demais processos abertos e ao buffer de memória) e chegou a atingir 140MB durante o evento (Fig. 10); ou seja, totalizando uma máxima de 52MB.

Figura 8 – Erros de conexões geradas no primeiro teste

#	CPU				Memory				Network				TCP					
#Time	cpu	sys	inter	ctxsw	Free	Buff	Cach	Inac	Slab	Map	KBIn	PktIn	KBOut	PktOut	IP	Tcp	Udp	Icmp
02:34:19	52	23	17M	7331K	3G	34M	3G	334M	110M	811M	2110K	24228K	1841K	27167K	2	12	6	160
02:34:19	82	36	7140	3851	3G	34M	3G	334M	138M	830M	1415	17215	1074	15811	0	0	0	1
02:34:20	63	36	8406	1908	3G	34M	3G	334M	137M	831M	1411	15893	1517	22666	0	0	0	0
02:34:20	52	23	7094	3006	3G	34M	3G	334M	110M	811M	887	9939	773	11145	0	0	0	0
02:34:21	67	36	4134	2231	3G	34M	3G	334M	137M	831M	1313	15272	1397	20629	0	0	0	0
02:34:22	52	23	17M	7339K	3G	34M	3G	334M	110M	811M	2114K	24276K	1845K	27226K	2	12	6	161
02:34:22	66	36	4113	2373	3G	34M	3G	334M	138M	831M	1428	16690	1314	19549	1	0	0	0
02:34:23	52	23	17M	7341K	3G	34M	3G	334M	110M	811M	2116K	24293K	1846K	27245K	3	12	6	161
02:34:23	71	33	6238	3829	3G	34M	3G	334M	138M	831M	1324	15705	1270	18912	0	0	0	0
02:34:24	52	23	17M	7345K	3G	34M	3G	334M	110M	811M	2117K	24309K	1847K	27264K	3	12	6	161
02:34:24	70	37	5432	4271	3G	34M	3G	334M	138M	832M	1315	15179	1426	21287	0	0	0	0
02:34:25	52	23	17M	7350K	3G	34M	3G	334M	110M	811M	2118K	24324K	1849K	27285K	3	12	6	161
02:34:25	68	35	4578	3177	3G	34M	3G	334M	138M	832M	1282	15038	1397	20814	0	0	0	0
02:34:26	52	23	17M	7353K	3G	34M	3G	334M	110M	811M	2120K	24339K	1850K	27306K	3	12	6	161
02:34:26	71	40	6134	4986	3G	34M	3G	334M	137M	831M	1295	14814	1557	23188	0	0	0	0
02:34:27	52	23	17M	7358K	3G	34M	3G	334M	110M	811M	2121K	24354K	1851K	27329K	3	12	6	161
02:34:27	68	33	8215	3368	3G	34M	3G	334M	137M	831M	1350	15998	1336	19840	0	0	0	0
02:34:28	52	23	7089	3007	3G	34M	3G	334M	110M	811M	888	9955	775	11172	0	0	0	0
02:34:28	70	36	5079	3883	3G	34M	3G	334M	138M	832M	1443	17072	1340	20015	0	0	0	0

Fonte: O autor

Figura 9 – Balance da carga de CPU no primeiro teste

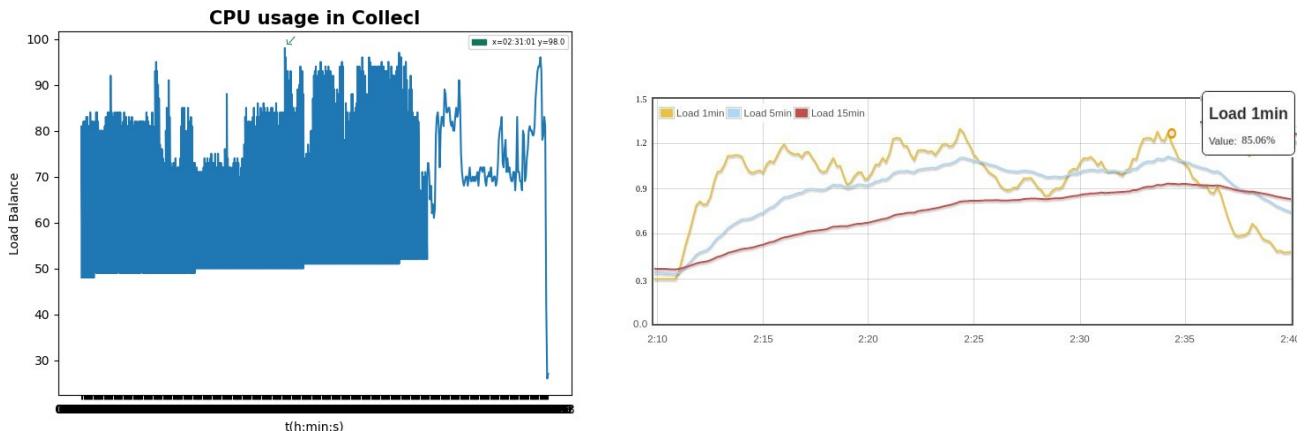
**Fonte: O autor**

Figura 10 – Consumo máximo de memória no primeiro teste

#	CPU				Memory				Network				TCP					
#Time	cpu	sys	inter	ctxsw	Free	Buff	Cach	Inac	Slab	Map	KBIn	PktIn	KBOut	PktOut	IP	Tcp	Udp	Icmp
02:32:36	93	51	7033	2589	3G	34M	3G	334M	140M	828M	1838	20599	1393	20421	0	0	0	0
02:32:37	51	23	16M	7040K	3G	34M	3G	334M	109M	810M	1956K	22416K	1693K	24948K	0	12	6	160
02:32:37	93	50	5585	2720	3G	34M	3G	334M	139M	828M	1677	19212	1472	21590	0	0	0	0
02:32:38	51	23	16M	7043K	3G	34M	3G	334M	109M	810M	1958K	22436K	1695K	24970K	0	12	6	160
02:32:38	95	42	5008	2924	3G	34M	3G	334M	140M	828M	1559	18095	1606	23618	0	0	0	0

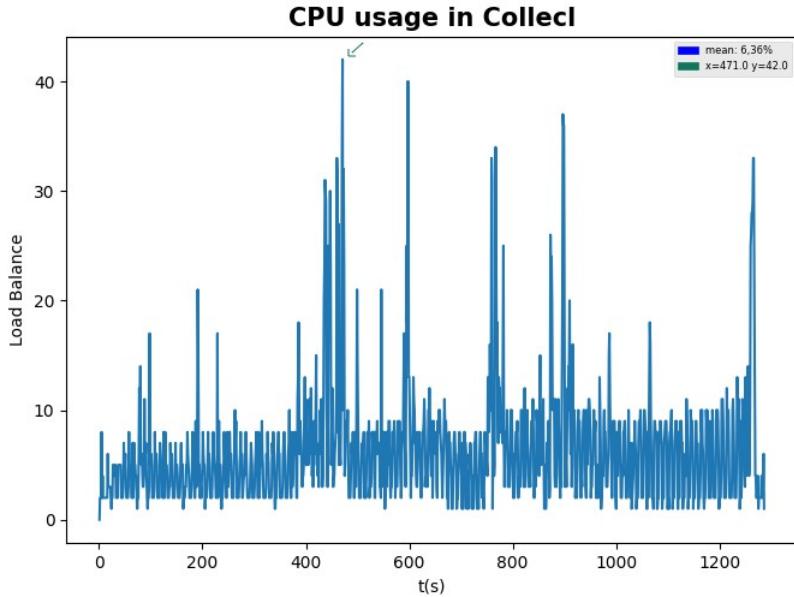
Fonte: O autor

3.4.2 Muitos assinantes para um publicador

Neste segundo teste, o consumo dos recursos foram bem reduzidos comparado com o teste anterior. O Collectl trouxe um pico máximo de processamento de 42% (Fig. 11), ocorrendo após 471s do início da coleta dos dados, com uma média total de 6,36%;

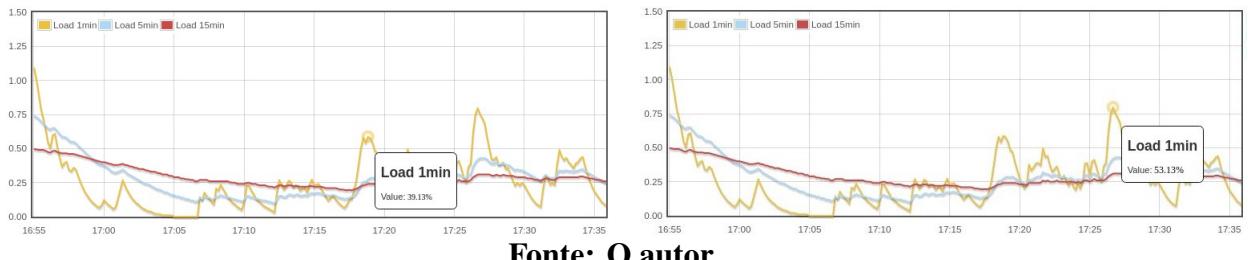
O RPi-Monitor apresentou os picos de 39.13% e 53.13%, como mostram as imagens do gráfico 12. Isto significa que ambos os softwares obtiveram resultados satisfatório quanto ao consumo da CPU.

Figura 11 – Balance da carga de CPU, segundo teste



Fonte: O autor

Figura 12 – Balance da carga de CPU no RPi-Monitor, segundo teste



Fonte: O autor

A memória ocupada estava em 107MB no início do teste e aumentou de forma linear (em média 1MB por minuto), alcançando 135MB (28MB de consumo adicional) quando finalizou a assinatura dos 10000 subscribers. Em seguida, tem-se as entregas das publicações, contabilizando um pico de 157MB (50MB de consumo adicional). Este evento pode ser melhor analisado na Figura 13. Uma explicação para isso pode ser a carga da mensagem "CC UFFS" que somava um total de 22 bytes (este aumento durou somente 6 segundos do teste).

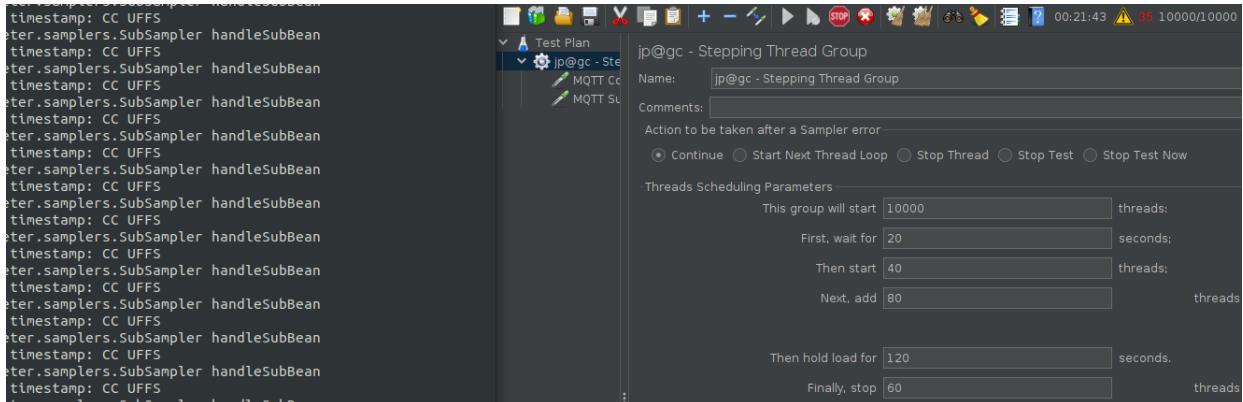
Durante as conexões não ocorreram falhas; porém, no momento em que foi publicado simultaneamente nas 10000 assinaturas, houve uma perda de 35 publicações, totalizando 0,35% das mensagens enviadas (no canto superior direito da figura 14 é possível perceber o erro apontado pelo software Jmeter). O teste durou 25 min 58s.

Figura 13 – Consumo máximo de memória no segundo teste

#Time	CPU	Memory	Network
	#cpu sys inter	ctxsw Free Buff Cach Inac Slab Map KBIn PktIn KBOut PktOut	
05:31:58	7 2 1159	1544 5G 36M 1G 325M 135M 777M	4 56 3 45
05:31:59	13 4 1265	1628 5G 36M 1G 325M 135M 777M	9 136 7 114
05:31:60	14 5 1611	2301 5G 36M 1G 325M 135M 777M	9 131 7 110
05:32:00	11 1 987	1021 5G 36M 1G 325M 135M 776M	8 121 7 101
05:32:01	4 0 694	740 5G 36M 1G 325M 135M 776M	7 114 6 95
05:32:02	5 1 553	437 5G 36M 1G 325M 135M 774M	6 85 5 71
05:32:03	11 1 344	312 5G 36M 1G 325M 135M 774M	0 0 0 0
05:32:04	25 0 259	113 5G 36M 1G 325M 135M 774M	0 1 0 0
05:32:05	26 0 381	297 5G 36M 1G 325M 135M 774M	0 0 0 0
05:32:06	28 0 687	822 5G 36M 1G 325M 135M 775M	0 0 0 0
05:32:07	28 0 563	692 5G 36M 1G 325M 135M 775M	0 1 0 0
05:32:08	29 6 1919	554 5G 36M 1G 325M 142M 774M	134 2024 259 3255
05:32:09	33 18 2975	502 5G 36M 1G 325M 157M 774M	1527 22710 1543 20375
05:32:10	33 13 3063	779 5G 36M 1G 325M 157M 774M	1397 20646 1585 20954
05:32:11	22 8 2909	452 5G 36M 1G 325M 153M 774M	1046 15448 1518 20236
05:32:12	14 4 3356	747 5G 36M 1G 325M 141M 774M	437 6450 418 5733
05:32:13	4 0 1153	397 5G 36M 1G 325M 137M 774M	59 881 64 870
05:32:14	4 0 616	197 5G 36M 1G 325M 135M 774M	22 326 25 336
05:32:15	2 0 275	150 5G 36M 1G 325M 135M 774M	5 67 5 66
05:32:16	2 0 365	429 5G 36M 1G 325M 135M 774M	0 1 0 1

Fonte: O autor

Figura 14 – Aviso de erros no segundo teste



Fonte: O autor

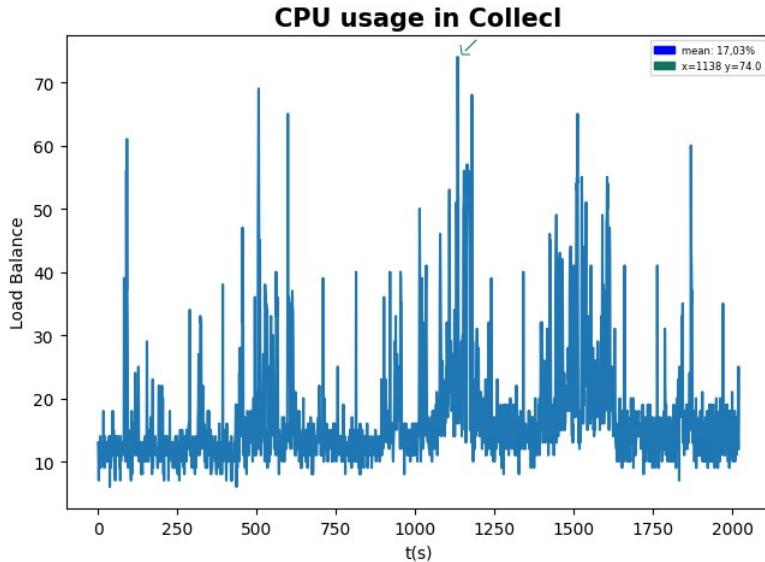
3.4.3 Teste com pares de clientes Pub/Sub

Durante o evento, os consumos do sistema embarcado foram os seguintes: a carga máxima da unidade de processamento foi de 74% no log do Collectl (Fig. 15); entretanto, as figuras do RPi-Monitor (Fig.16) apresentam 87.2% como carga máxima e uma distribuição contínua no intervalo de 0,5Ghz e 0,75Ghz.

O teste teve um tempo de 33 min 24s, sendo iniciado às 19:10:15 e finalizado em 19:43:39. Após 16 min 40s (em 19:26:55), todos os subscribers haviam assinado os tópicos, dando início às publicações que se estenderam até o fim do teste.

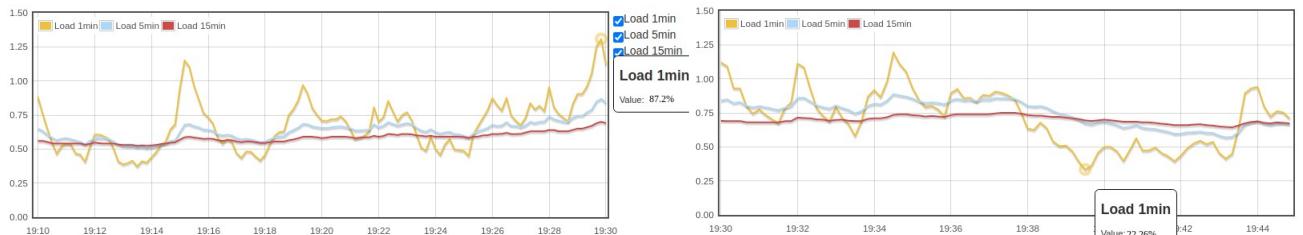
A memória ocupada partiu de 54MB e atingiu o pico máximo em 68MB (durante o cálculo da taxa de sucesso). Assim, o consumo foi de 11MB durante mais de 14 min 34s de teste, e de 14MB enquanto registrava os resultados: a figura 17 mostra o momento em que a memória teve um pico e posteriormente retorna para o valor inicial.

Figura 15 – Balance da carga de CPU, terceiro teste



Fonte: O autor

Figura 16 – Balance da carga de CPU no RPi-Monitor, terceiro teste



Fonte: O autor

Figura 17 – Consumo máximo de memória no final do terceiro teste

#<-----CPU----->		-Memory----->								-Network----->			
#cpu	sys	inter	ctxsw	Free	Buff	Cach	Inac	Slab	Map	KBIn	PktIn	KBOut	PktOut
12	2	887	1039	6G	12M	957M	85M	65M	934M	30	460	26	396
15	2	1027	1118	6G	12M	957M	85M	65M	934M	31	457	27	391
16	2	1078	1113	6G	12M	957M	85M	65M	934M	32	486	26	402
14	2	920	1017	6G	12M	957M	85M	65M	934M	29	442	25	385
16	7	3420	1150	6G	12M	957M	85M	68M	934M	417	6376	208	3193
25	12	3994	1423	6G	12M	957M	85M	56M	934M	1022	15692	1105	16787
12	0	599	1027	6G	12M	957M	85M	54M	934M	1	17	1	16

Fonte: O autor

Não houveram perdas na relação de entrega; sendo assim, o teste foi íntegro quanto ao recurso de rede, garantindo 100% na entrega das mensagens. A Figura 18 traz os dados da latência de envio (publicadores), com os seguintes destaques: o tempo total dos publicadores foi de 1000 s 405 ms, o tempo mínimo de uma publicação foi de 1,665 ms e o tempo máximo foi de 366,249ms; em média, os envios aconteceram em 10,430 ms, com um desvio padrão médio de 4,013 ms, contabilizando uma taxa média de 93,038 envios de mensagens por segundo.

Figura 18 – Resultado de latência dos envios

```
===== TOTAL PUBLISHER (5000) =====
Total Publish Success Ratio: 100.000% (50000/50000)
Total Runtime (sec): 1000.405
Average Runtime (sec): 0.117
Pub time min (ms): 1.665
Pub time max (ms): 366.249
Pub time mean mean (ms): 10.430
Pub time mean std (ms): 4.013
Average Bandwidth (msg/sec): 93.038
Total Bandwidth (msg/sec): 465188.641
```

Fonte: O autor

A taxa na entrega para os assinantes, foram: latência mínima de 2,028 ms; latência máxima foi de 366,531 ms; a média total da dispersão de latência apresentou 5,517 ms e a média total da taxa resultou 14,495 ms como mostra a figura 19.

Figura 19 – Resultado de latência das entregas

```
===== TOTAL SUBSCRIBER (5000) =====
Total Forward Success Ratio: 100.000% (50000/50000)
Forward latency min (ms): 2.028
Forward latency max (ms): 366.531
Forward latency mean std (ms): 5.517
Total Mean forward latency (ms): 14.495

2022/02/26 19:43:39 All jobs done.
```

Fonte: O autor

4 CONCLUSÃO

O dispositivo Raspberry Pi 4 demonstrou-se estável durante boa parte dos testes, assegurando que seria capaz de operar em cenários de múltiplos clientes. Durante os índices de variações, no teste de publicadores, foi observado um consumo demasiado do dispositivo e, embora o fator do gargalo ter sido o elementar causador durante a métrica daquele evento, o Raspberry não chegou a ter um consumo total dos recursos (i.e., sem prejudicar na continuidade do serviço); Ou seja, o dispositivo operou como esperado. Enfim, em outros testes com o mesmo modelo, foi concluído que estas causas eram legítimas e, de fato, foi possível obter a entrega de todas as mensagens (principalmente nos testes cabeado). Durante os demais testes, o dispositivo nem chegou a entrar em um cenário considerado de estresse, visto que poucas vezes passou de 50% do uso total do recurso.

O broker Mosquitto atendeu às demandas como esperado, garantindo diversas configurações e, mesmo operando em apenas um núcleo, teve um bom retorno em relação ao consumo total dos recursos, podendo ser observado através do baixo consumo da utilização da memória: por exemplo, no teste final o pico máximo atingiu somente 14MB.

O protocolo *MQTT* apresentou um desempenho satisfatório, evidenciando que é possível atender/trocar mensagens em até 1,6 ms no contexto analisado, sendo comparável com outros protocolos projetados exclusivamente para aplicações de mensagens instantâneas e assíncrona entre as partes.

REFERÊNCIAS

- 1 ABES, Associação Brasileira de Empresas de Software. Mercado Brasileiro de Software: Panorama e Tendências 2021. In: 1. ed. São Paulo, SP, Brasil: ABES, 2021. p. 40. Disponível em: <<https://abessoftware.com.br/dados-do-setor/>>. Acesso em: 30 ago. 2021.
- 2 ADATA. Premier microSDHC/SDXC UHS-I Class10 Memory Card. [S.l.: s.n.]. Disponível em: <<https://www.adata.com/uk/consumer/203>>. Acesso em: 4 abr. 2022.
- 3 APACHE, JMeter™. What can I do with it? [S.l.]: The Apache Software foundation. Disponível em: <<https://jmeter.apache.org/>>. Acesso em: 3 ago. 2021.
- 4 ASHTON, Kevin. That ‘Internet of Things’ Thing. **RFID Journal**, jun. 2009. Disponível em: <<https://www.rfidjournal.com/that-internet-of-things-thing>>. Acesso em: 22 ago. 2021.
- 5 BASSI, Alessandro et al. **Enabling Things to Talk**: Designing IoT solutions with the IoT Architectural Reference Model. Berlin, Germany: Springer-Verlag Berlin Heidelberg, 2013. p. 349. ISBN 978-3-642-40403-0. Disponível em: <<https://link.springer.com/book/10.1007/978-3-642-40403-0>>. Acesso em: 4 set. 2021.
- 6 BHATTACHARJEE, Sravani. **Practical Industrial Internet of Things Security**: A practitioner’s guide to securing connected industries. Birmingham, Englan, UK: Packt Publishing Ltd, jul. 2018. p. 324. ISBN 9781788832687. Disponível em: <<https://www.packtpub.com/product/practical-industrial-internet-of-things-security/9781788832687>>. Acesso em: 4 set. 2021.
- 7 COLLECTL Latest Version. [S.l.]: Source Forge. Disponível em: <<http://collectl.sourceforge.net/>>. Acesso em: 15 jan. 2022.
- 8 DI MARTINO, B. et al. Internet of things reference architectures, security and interoperability: A survey. **Internet of Things**, Dipartimento di Ingegneria, Università della Campania Luigi Vanvitelli, Aversa (CE), Via Roma 29, Italy, v. 1-2, p. 99–112, 2018. ISSN 2542-6605. DOI: <https://doi.org/10.1016/j.iot.2018.08.008>.
- 9 ECLIPSE, Mosquitto™. An open source MQTT broker. [S.l.]: Eclipse Foundation. Disponível em: <<https://mosquitto.org/>>. Acesso em: 20 set. 2021.
- 10 FOUNDATION, Apache Software. **HTTP Server Project**. [S.l.: s.n.]. Disponível em: <<https://httpd.apache.org/>>. Acesso em: 15 jan. 2022.

- 11 GÖTZ, Christian. **MQTT 101**: How to Get Started with the lightweight IoT Protocol. [S.l.]: Eclipse Foundation, 2014. Disponível em: <https://www.eclipse.org/community/eclipse_newsletter/2014/october/article2.php>. Acesso em: 7 set. 2021.
- 12 GROUP, ITU-T Study. **New ITU standards define the Internet of Things and provide the blueprints for its development**. [S.l.: s.n.], 2012. Disponível em: <<http://newslog.itu.int/archives/245>>. Acesso em: 5 ago. 2021.
- 13 HUAWEI. **MQTT broker latency measure tool**. [S.l.: s.n.]. Disponível em: <<https://github.com/hui6075/mqtt-bm-latency>>. Acesso em: 15 jan. 2022.
- 14 IBM, International Business Machines Corporation; EUROTECH. **MQTT V3.1 Protocol Specification**. [S.l.: s.n.], 2010. Disponível em: <<http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html>>. Acesso em: 20 ago. 2021.
- 15 MANYIKA, James et al. The Internet of Things: Mapping the value beyond the hype. In: [s.l.]: McKinsey Global Institute, jun. 2015. Disponível em: <<https://apo.org.au/node/55490>>. Acesso em: 8 ago. 2021.
- 16 MQTT. **FAQ**. [S.l.: s.n.]. Disponível em: <<https://mqtt.org/faq/>>. Acesso em: 20 jul. 2021.
- 17 MUENCHEN, Jean. **Uma proposta de detecção de incêndio utilizando protocolo MQTT para aplicações IOT**. Santa Maria, RS, Brasil: Universidade Federal de Santa Maria, jun. 2018. Disponível em: <<https://repositorio.ufsm.br/handle/1/15799>>. Acesso em: 24 ago. 2021.
- 18 PET, Sistemas de informação. **Computação em Nuvem**. [S.l.]: Universidade Federal de Santa Maria-UFSM, 2020. Disponível em: <<https://www.ufsm.br/pet/sistemas-de-informacao/2020/09/15/computacao-em-nuvem/>>. Acesso em: 20 jul. 2021.
- 19 PI FOUNDATION, Raspberry. **Raspberry Pi 4 Tech Specs**. [S.l.]: Raspberry Pi Org. Disponível em: <<https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/>>. Acesso em: 16 set. 2021.
- 20 PROJECT, European lighthouse integrated. **Internet of Things: Architecture**. [S.l.]: 7 th framework programme, ago. 2019. Disponível em: <<https://cordis.europa.eu/project/id/257521>>. Acesso em: 31 ago. 2021.
- 21 RODRIGUES, Gabriel Augusto Veiga. **Análise de desempenho do beaglebone black utilizando o protocolo MQTT**. Ponta Grossa, PR, Brasil: Universidade Tecnológica Federal do Paraná, jun. 2019. Disponível em: <<http://repositorio.roca.utfpr.edu.br/jspui/handle/1/15991>>. Acesso em: 24 ago. 2021.

- 22 RPI-EXPERIENCES. **RPi-Monitor documentation**. [S.l.: s.n.]. Disponível em: <<https://rpi-experiences.blogspot.com/>>. Acesso em: 3 out. 2021.
- 23 SILVA, Leandro Jamir. **Internet Das Coisas**. Palhoça, SC, Brasil: Universidade do Sul de Santa Catarina[UNISUL],, 2017. Disponível em: <<https://repositorio.animaeducacao.com.br/handle/ANIMA/11220>>. Acesso em: 18 ago. 2021.
- 24 SURESH, P. et al. A state of the art review on the Internet of Things (IoT) history, technology and fields of deployment. In: 2014 International Conference on Science Engineering and Management Research (ICSEMR). [S.l.: s.n.], nov. 2014. p. 1–8. DOI: [10.1109/ICSEMR.2014.7043637](https://doi.org/10.1109/ICSEMR.2014.7043637).
- 25 TORRES, Andrei B. B.; ROCHA, Atslands R.; SOUZA, José Neuman de. Análise de Desempenho de Brokers MQTT em Sistema de Baixo Custo. In: XV Workshop em Desempenho de Sistemas Computacionais e de Comunicação. Porto Alegre, RS, Brasil: Sociedade Brasileira da Computação - SBC, 2016. (15), p. 2804–2815. DOI: <https://doi.org/10.5753/wperformance.2016.9727>.
- 26 XMETER. **MQTT JMeter Plugin**. [S.l.: s.n.]. Disponível em: <<https://github.com/xmeter-net/mqtt-jmeter>>. Acesso em: 15 jan. 2022.
- 27 YUAN, Michael. Conhecendo o MQTT: Por que o MQTT é um dos melhores protocolos de rede para a Internet das Coisas? **Article IBM developer**, out. 2017. Disponível em: <<https://developer.ibm.com/br/technologies/iot/articles/iot-mqtt-why-good-for-iot/>>. Acesso em: 1 set. 2021.

APÊNDICE A - Dados coletados nos testes de publicadores.

- Exato momento em que ocorre a carga máxima do processador no log do Collectl; apresentado na coluna 2 e com tempo 02:32:01 da figura 20, primeiro teste resultado na pesquisa.

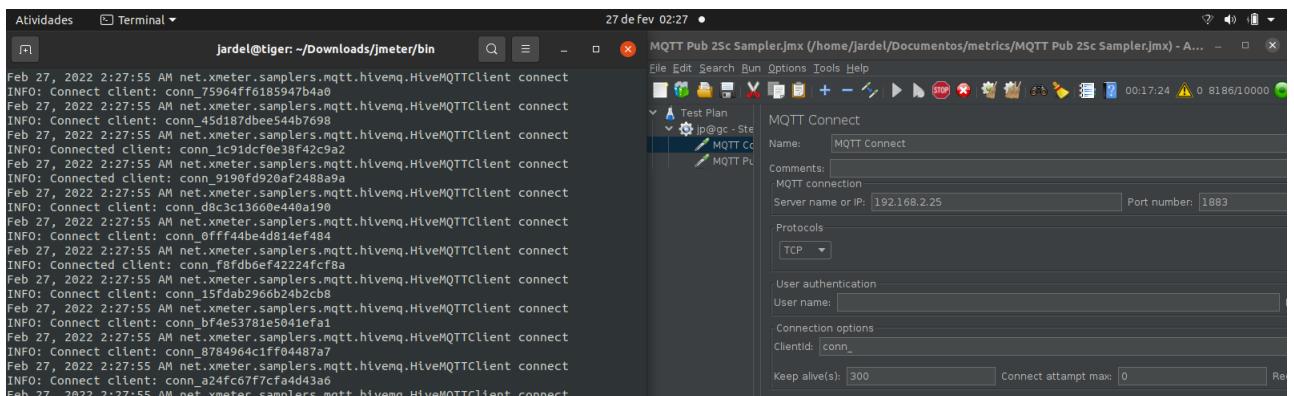
Figura 20 – Consumo máximo de CPU no primeiro teste

#	CPU												Memory				Network			
Time	cpu	sys	inter	ctxsw	Free	Buff	Cach	Inac	Slab	Map	KBIn	PktIn	KBOut	PktOut	IP	Tcp	Udp	Icmp		
02:32:00	50	23	16M	6925K	3G	34M	3G	334M	108M	810M	1908K	21857K	1648K	24263K	0	12	6	160		
02:32:00	84	44	7401	2718	3G	34M	3G	334M	137M	828M	1621	18273	1339	19631	0	0	0	0		
02:32:01	50	23	16M	6928K	3G	34M	3G	334M	108M	810M	1910K	21875K	1649K	24283K	0	12	6	160		
02:32:01	98	47	6480	2757	3G	34M	3G	334M	138M	828M	1486	16885	1279	18686	0	0	0	0		
02:32:02	50	23	16M	6930K	3G	34M	3G	334M	109M	810M	1911K	21892K	1650K	24302K	0	12	6	160		
02:32:02	96	44	6311	3182	3G	34M	3G	334M	138M	829M	1415	16041	1365	20032	0	0	0	0		
02:32:03	50	23	16M	6934K	3G	34M	3G	334M	109M	810M	1912K	21908K	1651K	24322K	0	12	6	160		
02:32:03	90	37	9339	3852	3G	34M	3G	334M	138M	828M	713	7967	828	12161	0	0	0	0		
02:32:04	50	23	16M	6937K	3G	34M	3G	334M	109M	810M	1913K	21916K	1652K	24334K	0	12	6	160		
02:32:04	93	38	9064	3518	3G	34M	3G	334M	138M	828M	897	10028	936	13796	0	0	0	0		
02:32:05	50	23	16M	6941K	3G	34M	3G	334M	109M	810M	1914K	21926K	1653K	24348K	0	12	6	160		
02:32:05	86	42	10019	2987	3G	34M	3G	334M	138M	828M	1539	17597	1187	17515	0	0	0	0		

Fonte: O autor

- Aviso de falha na rede sem fio antes do final dos envios dos publicadores (primeiro teste resultado), encontrado no canto superior direito da figura 21. Embora esta sinalização, os pacotes foram enviados ao servidor.

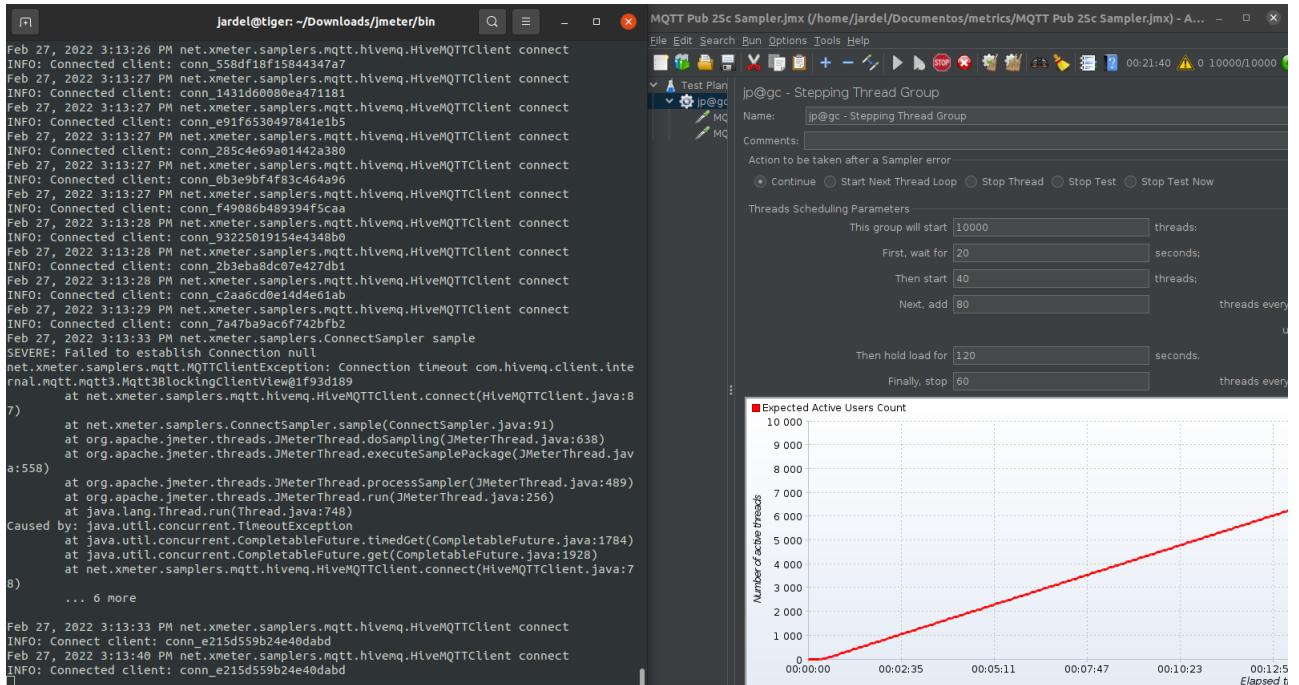
Figura 21 – Aviso no sinal da rede sem fio



Fonte: O autor

- Em outro teste de publicadores (não utilizado como resultados na pesquisa), é possível verificar os erros de timeout (Soft. Jmeter, a esquerda da Fig. 22) após enviar todos os 10000 clientes no lado direito superior da imagem; logo em seguida, em uma tentativa de conexão do cliente *conn_e215d559b24e40dabd*, a resposta "connected" demorou 7 segundos de tempo (lado esquerdo da figura 22).

Figura 22 – Aviso de erro com exceção do tipo: timeout



Fonte: O autor

APÊNDICE B - Dados coletados no teste de assinantes.

- Exato momento em que ocorre a carga máxima do processador sendo computado no log do Collectl, coluna 2, com tempo 05:18:09 da figura 23.

Figura 23 – Carga máxima de CPU no segundo teste

#Time	#cpu	sys	inter	ctxsw	Free	Buff	Cach	Inac	Slab	Map	KBIn	PktIn	KBOut	PktOut
05:18:06	10	2	1292	1325	56	36M	1G	325M	116M	772M	9	129	8	112
05:18:07	20	4	2700	4329	56	36M	1G	325M	116M	773M	9	128	8	112
05:18:08	35	4	6063	9554	56	36M	1G	325M	117M	774M	9	129	8	112
05:18:09	42	11	7199	11448	56	36M	1G	325M	117M	774M	10	133	8	116
05:18:10	10	1	1739	2418	56	36M	1G	325M	117M	774M	6	88	5	76
05:18:11	32	9	6344	11800	56	36M	1G	325M	117M	774M	0	2	0	0
05:18:12	23	4	5654	8798	56	36M	1G	325M	117M	774M	0	3	0	0
05:18:13	4	2	636	1071	56	36M	1G	325M	117M	774M	0	0	0	0
05:18:14	5	2	764	1001	56	36M	1G	325M	117M	775M	0	1	0	0
05:18:15	8	2	1747	3301	56	36M	1G	325M	117M	774M	3	36	2	32
05:18:16	9	2	1095	1243	56	36M	1G	325M	117M	774M	9	129	8	112
05:18:17	9	1	1063	1036	56	36M	1G	325M	117M	774M	9	130	8	112
05:18:18	8	2	885	780	56	36M	1G	325M	117M	773M	10	134	8	114
05:18:19	10	3	1185	1209	56	36M	1G	325M	117M	773M	9	122	7	108
05:18:20	6	0	583	338	56	36M	1G	325M	117M	773M	7	97	6	82
05:18:21	2	0	229	201	56	36M	1G	325M	117M	774M	0	0	0	0

Fonte: O autor

APÊNDICE C - Dados coletados nos testes de pares de clientes.

- Parte geradora do Design experimental do terceiro teste, que se encontra no arquivo principal do programa:

Figura 24 – Variáveis da função principal que modelam o cenário

```
func main() {
    counter := 0
    var (
        broker     = flag.String("broker", "tcp://192.168.2.25:1883", "MQTT broker endpoint as scheme://host:port")
        topic      = flag.String("topic", "broker_RPI", "MQTT topic for outgoing messages")
        username   = flag.String("username", "", "MQTT username (empty if auth disabled)")
        password   = flag.String("password", "", "MQTT password (empty if auth disabled)")
        pubqos    = flag.Int("pubqos", 1, "QoS for published messages")
        subqos    = flag.Int("subqos", 1, "QoS for subscribed messages")
        size       = flag.Int("size", 100, "Size of the messages payload (bytes)")
        count      = flag.Int("count", 10, "Number of messages to send per pubclient")
        clients   = flag.Int("clients", 5000, "Number of clients pair to start")
        keepalive  = flag.Int("keepalive", 60, "Keep alive period in seconds")
        format     = flag.String("format", "text", "Output format: text|json")
        quiet      = flag.Bool("quiet", false, "Suppress logs while running")
    )
}
```

Fonte: O autor

- Funções que criam o publicador e o assinantes com intervalos implementados:

Figura 25 – Alterações para intervalos de 2 segundos

```
for i := 0; i < *clients; i++ {
    sub := &SubClient{
        ID:          i,
        BrokerURL:  *broker,
        BrokerUser: *username,
        BrokerPass: *password,
        SubTopic:   *topic + "-" + strconv.Itoa(i),
        SubQoS:     byte(*subqos),
        KeepAlive:  *keepalive,
        Quiet:      *quiet,
    }
    counter = counter + 1
    if counter == 10 {
        time.Sleep(2 * time.Second)
        counter = 0
    }
    go sub.run(subResCh, subDone, jobDone)
}

for i := 0; i < *clients; i++ {
    c := &PubClient{
        ID:          i,
        BrokerURL:  *broker,
        BrokerUser: *username,
        BrokerPass: *password,
        PubTopic:   *topic + "-" + strconv.Itoa(i),
        MsgSize:    *size,
        MsgCount:   *count,
        PubQoS:     byte(*pubqos),
        KeepAlive:  *keepalive,
        Quiet:      *quiet,
    }
    counter = counter + 1
    if counter == 10 {
        time.Sleep(2 * time.Second)
        counter = 0
    }
    go c.run(pubResCh)
}
```

Fonte: O autor

- Teste com 25000 pares de clientes, 25000 tópicos e 1 milhão de publicações, onde cada cliente publicava 40 vezes em apenas um tópico específico, rede sem fio (Fig. 26). É possível observar que 80 entregas não foram efetuadas, gerando um tipo de dado (not-a-number) nas duas variáveis finais (média do desvio padrão e média total das entregas), motivado pela não totalização das entregas, se considerando que o vetor final não atribui os valores nulos, ou assim dizendo, 80 campos vazios no total dos dados. Esse falha ocorreu em razão do final do teste, pois o design experimental teve um crescimento de 80 clientes a cada 5 s, e quando finalizou, o programa esperou somente 3 s e desconectou todos

os clientes aleatoriamente, logo, o broker não entregou dados de 2 assinantes previstos no teste.

Figura 26 – Teste pareado com 25 mil clientes, rede sem fio

```
===== TOTAL PUBLISHER (25000) =====
Total Publish Success Ratio: 100.000% (1000000/1000000)
Total Runtime (sec): 1565.923
Average Runtime (sec): 2.843
Pub time min (ms): 1.811
Pub time max (ms): 9652.595
Pub time mean mean (ms): 68.938
Pub time mean std (ms): 30.968
Average Bandwidth (msg/sec): 16.377
Total Bandwidth (msg/sec): 409418.362

===== TOTAL SUBSCRIBER (25000) =====
Total Forward Success Ratio: 99.992% (999920/1000000)
Forward latency min (ms): 1.904
Forward latency max (ms): 9652.584
Forward latency mean std (ms): NaN
Total Mean forward latency (ms): NaN
```

Fonte: O autor

- Teste com o mesmo design experimental do resultante (5 mil clientes), em rede sem fio (Fig. 27). Não resultado na pesquisa.

Figura 27 – Teste pareado, sem fio

```
===== TOTAL PUBLISHER (5000) =====
Total Publish Success Ratio: 100.000% (50000/50000)
Total Runtime (sec): 1000.716
Average Runtime (sec): 0.214
Pub time min (ms): 2.184
Pub time max (ms): 585.636
Pub time mean mean (ms): 18.149
Pub time mean std (ms): 11.353
Average Bandwidth (msg/sec): 55.209
Total Bandwidth (msg/sec): 276047.131

===== TOTAL SUBSCRIBER (5000) =====
Total Forward Success Ratio: 100.000% (50000/50000)
Forward latency min (ms): 2.254
Forward latency max (ms): 585.537
Forward latency mean std (ms): 12.325
Total Mean forward latency (ms): 18.618
```

Fonte: O autor

- Teste com o mesmo design experimental do resultante (5 mil clientes), em rede cabeada (Fig. 28). Não resultado na pesquisa.

Figura 28 – Teste pareado, cabeado

```
===== TOTAL PUBLISHER (5000) =====
Total Publish Success Ratio: 100.000% (50000/50000)
Total Runtime (sec): 1000.327
Average Runtime (sec): 0.021
Pub time min (ms): 0.241
Pub time max (ms): 13.562
Pub time mean mean (ms): 1.688
Pub time mean std (ms): 0.987
Average Bandwidth (msg/sec): 627.347
Total Bandwidth (msg/sec): 3136736.338

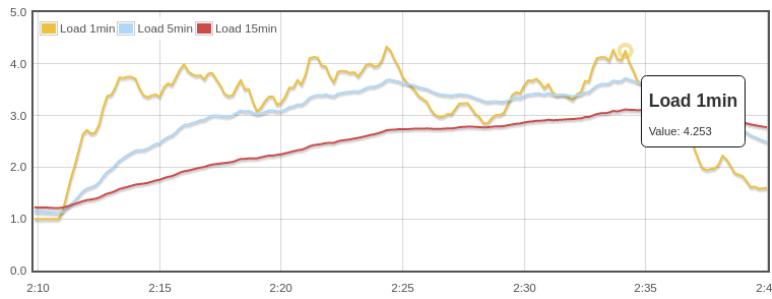
===== TOTAL SUBSCRIBER (5000) =====
Total Forward Success Ratio: 100.000% (50000/50000)
Forward latency min (ms): 0.248
Forward latency max (ms): 13.501
Forward latency mean std (ms): 0.967
Total Mean forward latency (ms): 1.660
```

Fonte: O autor

APÊNDICE D - Inclusão dos gráficos utilizados antes da normalização.

- A figura 29 representa o gráfico utilizado no trabalho durante a carga de CPU, antes da normalização em porcentagem.

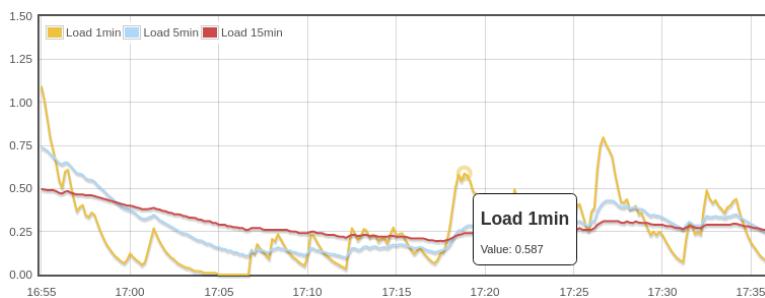
Figura 29 – Gráfico não normalizado, teste 01



Fonte: O autor

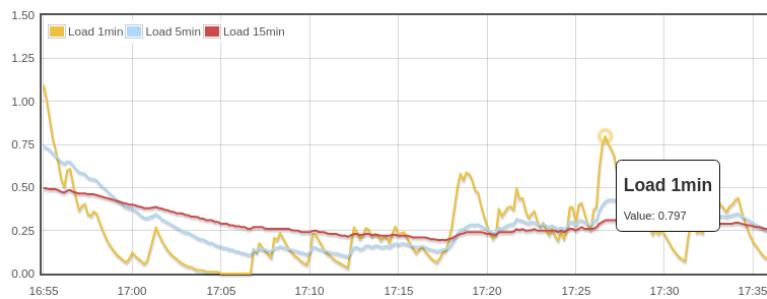
- As figuras 30 e 31 representam os gráficos utilizados no trabalho durante a carga CPU, no teste de assinantes, antes da normalização em porcentagens.

Figura 30 – Gráficos não normalizados, teste 2 parte 1



Fonte: O autor

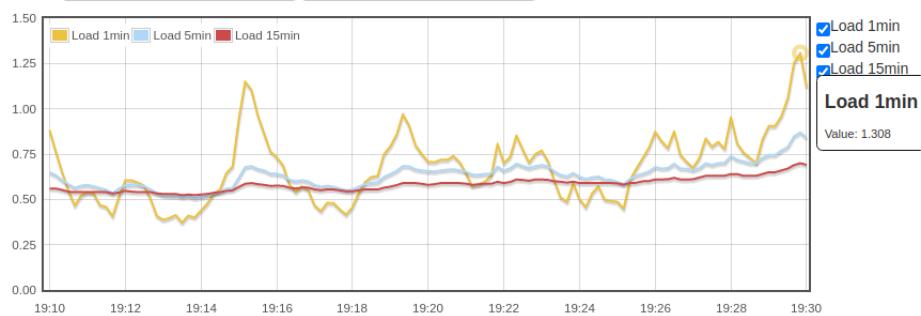
Figura 31 – Gráficos não normalizados, teste 2 parte 2



Fonte: O autor

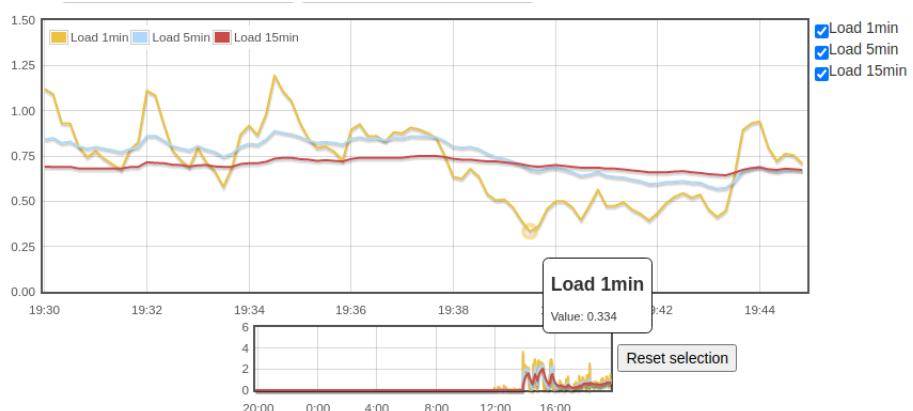
- As figuras 32 e 33 representam os gráficos utilizados no trabalho durante a carga de CPU, terceiro teste, antes da normalização em porcentagens.

Figura 32 – Gráficos não normalizados, teste 3 parte 1



Fonte: O autor

Figura 33 – Gráficos não normalizados, teste 3 parte 2



Fonte: O autor