Implementação IoT de comunicação MQTT utilizando ESP32 com broker em cluster Kubernetes em ambientes Virtualizado, Embarcado e na Nuvem AWS

Douglas Bellomo Cavalcante[1], **Elielder Belchior de Melo**[1]

[1]Escola Politécnica - Universidade de São Paulo (EP-USP)

douglas.b.cavalcante@usp.br, elielder@usp.br

Resumo. *Este artigo apresenta uma implementação IoT (Internet das Coisas) de MQTT com publicações de mensagens com ESP32, usando Kubernetes com autoescalonamento horizontal (HPA) para escalar o broker MQTT nos ambientes virtualizado, embarcado e em nuvem Amazon AWS utilizando o recurso Amazon Elastic Kubernetes Service (EKS). Ao aplicar testes de sequência de mensagens com diferentes frequências, foi observado que o ambiente virtualizado apresentou menor taxa de perda de mensagens. Observa-se, portanto, que a possibilidade de implementação de uma aplicação MQTT em um cluster Kubernetes pode ser viável em ambiente embarcado e na nuvem em contextos específicos, devendo se considerar a criticidade de um ambiente de tempo real.*

Abstract. *This article presents an IoT (Internet of Things) implementation of MQTT with message publishing using ESP32, web technologies, and Kubernetes with horizontal pod autoscaling (HPA) to scale the MQTT broker across virtualized, embedded, and Amazon AWS cloud environments utilizing the Amazon Elastic Kubernetes Service (EKS). By conducting sequence message tests with different frequencies, it was observed that the virtualized environment had the lowest message loss rate. Therefore, it can be concluded that the possibility of implementing an MQTT application in a Kubernetes cluster may be feasible in embedded and cloud environments in specific contexts, taking into account the criticality of a real-time environment.*

1. Introdução

O número de dispositivos conectados à Internet das Coisas (*IoT - Internet of Things*) atingiu 16,6 bilhões até o final de 2023, marcando um ano de crescimento importante para essa tecnologia. De acordo com o relatório *State of IoT Summer 2024* da *IoT Analytics*, esse número deve aumentar para 18,8 bilhões até o final de 2024, embora o ritmo de crescimento seja mais lento em comparação aos anos anteriores. Ainda assim, a expansão da IoT permanece significativa, refletindo o papel crucial que essa tecnologia desempenha na transformação digital em escala global. Empresas de diversos setores continuam a investir em soluções conectadas, aproveitando os benefícios da automação e da conectividade inteligente, o que reafirma a importância crescente da IoT na modernização de processos e na criação de novas oportunidades de inovação. [IoT Analytics 2024]

O crescimento da IoT tem sido acompanhado por uma descentralização nas arquiteturas, com a adoção de modelos distribuídos que utilizam soluções de comunicação entre dispositivos e orquestradores na nuvem. Tecnologias como o MQTT (*Message*

Queuing Telemetry Transport) são usadas para mediar a troca de mensagens entre dispositivos IoT e servidores na nuvem. O MQTT é um protocolo de comunicação leve, projetado para redes com limitações de largura de banda e grande número de dispositivos conectados.

A implementação de *brokers* MQTT em ambientes de orquestração como o Kubernetes permite a gestão escalável e eficiente da comunicação entre dispositivos IoT e a nuvem. O Kubernetes facilita o escalonamento automático de recursos e a alta disponibilidade, possibilitando que infraestruturas de IoT administrem volumes elevados de dispositivos conectados e processos distribuídos, garantindo a resiliência e o gerenciamento adequado dessas redes.

A proposta deste trabalho é mostrar a implementação de uma aplicação MQTT, utilizando uma placa de desenvolvimento ESP32 para publicar mensagens MQTT, com broker MQTT operando em cluster Kubernetes e fazer uma análise comparativa de desempenho, disponibilidade e perda de mensagens entre três ambientes, desktop virtualizado, embarcado e na nuvem. Na seção 2, faremos uma breve revisão de literatura trazendo as discussões sobre viabilidade de uma aplicação IoT, como o MQTT, em um cluster Kubernetes. Na seção 3, há o detalhamento da aplicação de publicação de mensagens MQTT com ESP32. Na seção 4, há a explicação detalhada da implementação do broker MQTT nos ambientes de plataformas citadas acima. Nas seções 5 e 6 discutiremos resultados e traremos conclusões sobre a implementação, que pode ser explorada no repositório do projeto no GitHub[1]

2. Revisão de Literatura

Como demonstrado no artigo [Figueroa et al. 2023], a combinação de containerização com o gerenciamento via Kubernetes possibilita uma implantação eficiente, enquanto o protocolo MQTT se mostrou adequado para a comunicação e gerenciamento, especialmente ao ser adaptado para o envio de dados sensoriais. Além disso, o MQTT oferece vantagens para dispositivos IoT de baixo consumo. A escalabilidade e a distribuição geográfica dos nós Kubernetes ainda podem ser aprimoradas para otimizar o desempenho do sistema.

A possibilidade de implementar um *broker* MQTT em uma arquitetura IoT descentralizada se prova eficiente quando considera-se o baixo número de mensagens perdidas, como descrito em [Koziolek et al. 2020], que apresenta alguns cenários comparáveis entre si relacionando sua performance e escalabilidade para um ambiente IoT, no qual as chamadas de mensagens MQTT podem ser feitas em alta quantidade.

Embora o ambiente implementado em nuvem não é adequado para aplicações que exigem alta largura de banda como diversas aplicações IoT, de acordo com [Leskinen 2020], o kubernetes possibilita escalar e implementar aplicações com baixo recurso computacional, com a ferramenta K3s (*Lightweight Kubernetes*). E ainda acrescenta que tendo em vista a complexidade da rede de computadores e seus protocolos, o orquestrador kubernetes operando em computação de borda (*Edge Computing*) viabilizaria ainda mais a implementação de uma aplicação IoT.

Ao se considerar implementações kubernetes para aplicações IoT, é necessário

[1]https://github.com/douglasbcavalcante/tcn-trabalho-final

entender a limitação de recursos e o artigo [Koziolek and Eskandani 2023] aborda essa diferença ao apresentar as diversas distribuições que implementam o orquestrador kubernetes. As soluções de distribuições Kubernetes, tais como K3s e K8s são adequadas para aplicações específicas IoT e para implementar soluções complexas, construídas sobre a orquestração do kubernetes. Entretanto a comparação simples no contexto de execução de uma aplicação entre um computador de uso geral, que oferece mais recursos de memória e processamento e a execução do ambiente kubernetes em um computador de uso específico com recursos restritos de memória e processamento pode levar a execuções incompletas ou impossibilidade no uso de recursos em uma distribuição específica.

Para tornar a implementação com kubernetes ainda mais capaz de gerenciar ambientes com alta quantidade de mensagens e chamadas, o artigo [Centofanti et al. 2023] traz uma aplicação que implementa o *latency-aware scheduling mecanism*, que se refere a um sistema de escalonamento baseado na latência do ambiente kubernetes. Essa implementação empodera o ambiente a escalonar os pods de maneira mais ágil e inteligente para que se tenha um menor nivel de latência nas chamadas da aplicação implementada.

No geral, os trabalhos apontam que a combinação de containerização com Kubernetes e o uso do protocolo MQTT oferece uma solução eficiente para o gerenciamento e comunicação em sistemas IoT.

3. Implementação de Publisher MQTT em ESP32

A aplicação desenvolvida tem como foco o teste de recebimento de mensagens publicadas em um broker MQTT, utilizando o ESP32 como dispositivo de teste. O principal objetivo é garantir que as mensagens publicadas pelo dispositivo cheguem de forma confiável ao *broker*, permitindo que outros dispositivos ou serviços inscritos no mesmo *broker* possam recebê-las. Essa operação é fundamental em um sistema IoT, onde a troca contínua e precisa de dados entre dispositivos remotos é crucial para o correto funcionamento do sistema.

O ESP32 é um microcontrolador de 32 bits desenvolvido pela Espressif, que se destaca por integrar Wi-Fi 2.4 GHz e Bluetooth, tornando-o ideal para aplicações de IoT. Suas principais vantagens incluem conectividade sem fio integrada, baixo custo, alto desempenho e uma ampla gama de interfaces, como GPIO, ADC, DAC e PWM. Possui dois processadores principais de arquitetura RISC operando a até 240 MHz, um coprocessador de baixo consumo de energia, 448 kB de memória ROM interna e 520 kB de memória SRAM [Espressif 2024]. Por haver suporte massivo da comunidade e integração com a IDE do Arduino, sua escolha ocorreu pela facilidade que o ecossistema fornece para desenvolvimento de provas de conceito.

Na aplicação, o ESP32 é configurado para se conectar a uma rede Wi-Fi utilizando a função `setup_wifi()`. Após estabelecer a conexão, o dispositivo envia mensagens que contêm um timestamp em intervalos regulares por meio da função `client.publish()`. Essa implementação, que pode ser encontrada no GitHub[2], simula um cenário em que o ESP32 atua como um sensor ou dispositivo de monitoramento, publicando dados periodicamente em um *broker* MQTT em um tópico específico. A escolha de um *broker*

[2]https://github.com/douglasbcavalcante/tcn-trabalho-final/tree/main/esp32_mosquitto_pub

MQTT implementado em Kubernetes é fundamental, pois proporciona escalabilidade e resiliência, permitindo ao sistema gerenciar grandes volumes de mensagens e garantir alta disponibilidade, mesmo em ambientes com muitos dispositivos conectados. Além disso, a função `reconnect()` é utilizada para assegurar que o dispositivo restaure a conexão com o broker em caso de interrupções, o que é crucial para a operação contínua em um ambiente IoT dinâmico.

O ponto central do teste é verificar se o *broker* MQTT está recebendo corretamente essas mensagens publicadas, o que garante que o fluxo de dados entre o dispositivo e o sistema de comunicação esteja funcionando conforme esperado. Esse processo de teste é relevante porque, em um sistema IoT, os dados enviados para o *broker* geralmente são redistribuídos para outros dispositivos ou sistemas que dependem dessas informações para operar corretamente. A função `gettimeofday()` é responsável por obter o timestamp, que é então enviado para o broker a partir da função `client.publish()`, confirmando que a mensagem está sendo corretamente gerada e publicada no ciclo configurado.

Ao final, o teste de recebimento das mensagens publicadas no *broker* MQTT é essencial para validar a robustez e confiabilidade da comunicação, certificando-se de que os dados enviados pelo ESP32 estão chegando ao destino corretamente. Essa etapa é fundamental para garantir que os sistemas que dependem dessas mensagens possam operar de forma eficiente e sem interrupções.

4. Implementação de Broker MQTT com Kubernetes

Para a realização dos experimentos foi escolhido a versão do Kubernetes K3s e o orquestrador (*broker)* MQTT HiveMQ. A escolha dessas versões levou em consideração, principalmente, a possibilidade de implantação em hardware de baixa capacidade de processamento, para simular a aplicação em um *gateway* IoT, e abertura para uso em comunidade. Vídeos sobre as implementações e testes podem ser vistos na lista do YouTube[3]

O K3s é uma versão leve do Kubernetes, criada pela Rancher Labs, otimizada para ambientes de borda (*edge computing*), Internet das Coisas (IoT) e pequenos clusters. Ele é uma distribuição totalmente compatível com o Kubernetes padrão, mas reduzida em termos de requisitos de recursos, com componentes mais leves e menos dependências. As vantagens do K3s incluem fácil instalação, baixo consumo de recursos e suporte nativo a ambientes de baixa capacidade, como dispositivos IoT. Por outro lado, suas desvantagens incluem limitações para clusters de larga escala ou com demandas de alto desempenho, já que o K3s não é otimizado para esses cenários [K3s Project Authors 2024].

O HiveMQ é um orquestrador (*broker*) baseado em Java escalável e confiável, com suporte aos protocolos MQTT 3.x e MQTT 5. Foi projetado para facilitar a comunicação entre dispositivos IoT. Uma de suas principais vantagens é o suporte a uma grande quantidade de conexões simultâneas, o que o torna ideal para grandes implementações IoT, como cidades inteligentes e redes industriais. Além disso, oferece suporte a alta disponibilidade, fácil integração com sistemas corporativos e conformidade com padrões de segurança. A versão de comunidade (*HiveMQ Community Edition – CE* foi utilizada nesse projeto, por ser aberta para uso em provas de conceitos e pequenos projetos [HiveMQ CE Project Authors 2024].

[3]https://www.youtube.com/playlist?list=PLNcItX65p6WkYV4MTXmIoNxkXL7z6k-KD

4.1. Broker MQTT em Desktop Virtualizado

No ambiente de computador de uso geral, chamado Desktop, a implementação foi realizada em ambiente virtualizado, utilizando a ferramenta Virtual Box. O ambiente que hospedou a máquina virtual foi um sistema operacional Windows 10, com processador Intel I7 de oitava geração com 16GB de memória RAM. Para a máquina virtual, foram destinados 8GB de RAM e 2 processadores lógicos. No ambiente virtualizado, o sistema operacional Ubuntu 22.04 foi utilizado para a implementação.

A distribuição k3s foi escolhida para o ambiente virtualizado pois oferece uma solução leve e eficiente, pois a busca do projeto foi padronizar o trabalho entre diferentes ambientes de desenvolvimento. O Minikube é uma distribuição Kubernetes mais comumente usada em ambientes de maior recurso computacional, mas o k3s, por sua vez, aproveita-se da simplicidade e o menor consumo de recursos, permitindo que o mesmo ambiente seja utilizado em diversos ambientes, independentemente de suas capacidades. Essa padronização facilita a extração de análises sobre resultados, promovendo maior consistência no fluxo de trabalho e simplificando a transição entre os ambientes.

A implementação da aplicação pode ser explorada no repositório Github[4] do projeto e acompanha as demais implementações de broker MQTT em Kubernetes, utilizando k3s e o broker HiveMQ, apresentados anteriormente.

Como parte fundamental da implementação está a instalação da aplicação, o escalonamento horizontal de *pods* por carga e uma limitação específica de 10 réplicas e 50% de limite de carga por *pod* para a implementação, como pode-se observar no seguinte comando `$ sudo kubectl autoscale deployment hivemq-broker-depl --cpu-percent=50 --min=1 --max=10` . Essa particularidade foi adotada pelo generalidade do ambiente Desktop, executando outras aplicações e compartilhando recursos com uma máquina virtual. Para que o aproveitamento da implementação não prejudicasse outras tarefas no sistema operacional, tanto baremetal no ambiente de hospedagem da máquina virtual, quanto no sistema operacional virtualizado.

Os testes foram realizados como descrito abaixo:

- *Teste 1*: um dispositivo ESP32 publicando a cada 1 s;
- *Teste 2*: um dispositivo ESP32 publicando a cada 100 ms;
- *Teste 3*: um dispositivo ESP32 publicando a cada 10 ms.

4.2. Broker MQTT em Ambiente Embarcado

Para simulação de um ambiente de gateway foi escolhido o Raspberry Pi como plataforma para instalação do K3s e do HiveMQ. O Raspberry Pi é um computador de placa única, de baixo custo, amplamente utilizado em educação, prototipagem e projetos de eletrônica. Suas vantagens incluem a capacidade de rodar um sistema operacional completo (como o Linux), uma vasta gama de portas de entrada/saída (GPIO), compatibilidade com várias linguagens de programação e uma comunidade ativa que oferece suporte e recursos [Raspberry Pi Foundation 2024].

[4]https://github.com/douglasbcavalcante/tcn-trabalho-final/tree/main/x86

Foram utilizados dois modelos de Raspberry Pi para o projeto: o Raspberry Pi 3B, que utiliza um SoC (*System-on-a-Chip*) BCM2837 (processador quad-core ARM Cortex-A53 de 64 bits, 1,2 GHz de *clock* e GPU VideoCore IV) e 1 GB de memória RAM, e o Raspberry Pi 4B, que possui SoC BCM2711 (processador quad-core ARM Cortex-A72 de 64 bits, 1,5 GHz de *clock* e GPU VideoCore IV) e 8 GB de memória RAM. O sistema operacional *Raspberry Pi OS* foi escolhido, por ser baseado em Linux Debian e ter suporte oficial do fabricante.

O procedimento de instalação do *broker* HiveMQ CE sob o kubernetes K3s pode ser visto no repositório Github[5] do projeto. Iniciou-se pela instalação do cluster kubernetes baixando o *script* com o comando `$ sudo curl -sfL https://get.k3s.io | sh -`. O *script* se encarrega de toda a instalação, inicialização do serviço do K3s e disponibilização do aplicativo `kubectl`.

Em sequência foi feita a instalação do HiveMQ CE com a sequência de comandos disponibilizada no roteiro no GitHub. Os arquivos de configuração `hivemq-deployment.yaml` e `hivemq-service.yaml` também estão disponíveis no repositório GitHub do projeto.

Importante observar que somente o Raspberry Pi 4B conseguiu rodar o ecossistema, sendo que apesar do Raspberry Pi 3B ter permitido a execução dos *deployments*, ele não permitiu a criação de *pods* pelo HPA. Isso, provavelmente, se deve às limitações de recurso do equipamento, principalmente quanto à quantidade de memória RAM.

Em sequência foram realizados os seguintes testes de orquestração com o *broker* instalado no Raspberry Pi 4B, os dispositivos ESP32 publicando a data e hora (com milissegundos) no tópico `data_hora` , o consumo das publicações sendo realizada por um PC rodando um programa em python que se subscreve no mesmo tópico de publicação, e com todos os equipamentos conectados em uma mesma rede WiFi. Os registros de publicação, de recebimento das publicações e as quantidades de *pods* foram registradas em arquivos *csv* para análise.

- *Teste 1*: um dispositivo ESP32 publicando a cada 1 s;
- *Teste 2*: um dispositivo ESP32 publicando a cada 10 ms;
- *Teste 3*: um dispositivo ESP32 publicando a cada 100 ms; e
- *Teste 4*: dois dispositivos ESP32 publicando a cada 100 ms.

4.3. Broker MQTT na AWS com EKS

A criação do cluster na AWS com EKS se inicia com a formação de uma nuvem privada virtual (VPC) que atendem os requisitos da Amazon EKS. Para isso, executa-se o comando `$ aws cloudformation create-stack` com argumentos de região (utilizada a *us-east-1*), nome da VPC e *template* (utilizado o padrão sugerido no tutorial [Amazon 2024]).

[5]https://github.com/douglasbcavalcante/tcn-trabalho-final/tree/main/raspberry_pi

Uma vez criada a VPC, cria-se o cluster utilizando a interface *web* de gerenciamento da nuvem Amazon AWS e configura-se o `kubectl`local para operar remotamento o cluster. Em seguida cria-se os nós para o cluster utilizando o AWS EC2. Os nós do cluster são criados a partir do seu próprio ambiente de configuração na interface de gerenciamento da AWS. Para os testes em questão foram criadas duas instâncias do tipo *t3.medium* com quantidades padrões de cpu e memória. Em seguida efetua-se o *deploy* do HiveMQ, do *metrics-server* e efetua-se a configuração do HPA. O procedimento é exatamente o apresentado na Seção 4.2, porém o número máximo de pods foi ajustado para 20. O processo completo de *deploy* pode ser visto no vídeo disponível em https://youtu.be/gyVQQZnajuk.

Após concluído o processo de implantação do cluster EKS com a aplicação HiveMQ na AWS, os seguintes testes foram realizados, seguindo a mesma lógica dos testes feitos na Seção 4.1 e 4.2 e pode ser visto no GitHub[6]:

- *Teste 1*: um dispositivo ESP32 publicando a cada 1 s;
- *Teste 2*: um dispositivo ESP32 publicando a cada 10 ms;
- *Teste 3*: dois dispositivos ESP32 publicando a cada 10 ms;
- *Teste 4*: um dispositivo ESP32 publicando a cada 1 ms;

Além desses testes, foram realizados mais dois com publicações realizadas por um PC, de forma a estressar o cluster:

- *Teste 5*: PC publicando a cada 100 ns;
- *Teste 6*: PC publicando a cada 1 ms;

5. Análise de Resultados

Os testes consistiram na publicação da data e hora (com milissegundos) pelo ESP32, via MQTT, no broker implementado em três ambientes distintos, sendo que o relógio do dispositivo estava sincronizado com o servidor ntp.br. Um PC rodou um programa que subscrevia no mesmo tópico para receber a informação, escrevendo um log contendo a mensagem e a data e hora (com milissegundos) de recebimento. As seções a seguir descrevem os resultados obtidos em cada ambiente.

5.1. Resultados do teste com Desktop Virtualizado

A Tabela 1 mostra os resultados obtidos nos três testes realizados com disparo de mensagens com ESP32, e a implementação MQTT do HiveMQ no ambiente kubernetes K3s em ambiente virtualizado.

Observa-se na baixa taxa de perda de mensagens que a implementação foi executada com estabilidade e a aplicação conseguiu suprir a manutenção da disponibilidade do servidor, chamado *broker* MQTT. A escalabilidade dos *pods* mesmo com tempo reduzido

[6]https://github.com/douglasbcavalcante/tcn-trabalho-final/tree/main/aws

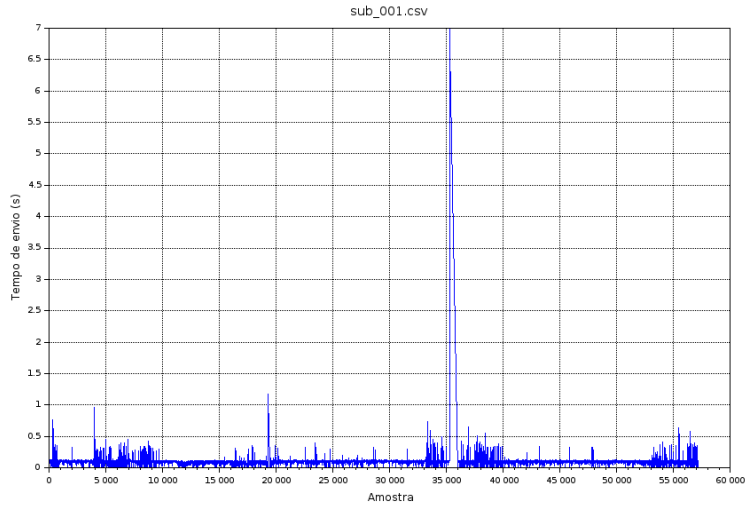
Table 1. Resultados dos testes do ESP32 publicando em ambiente virtualizado

Parâmetro	Teste 1	Teste 2	Teste 3
Tempo entre pub.	1 s	100 ms	10 ms
Quant. ESP32	1	1	1
Tempo de exper.	93 s	368 s	595 s
Máximo PODs HPA	1	10	10
Final IP ESP32	.18	.18	.18
Núm. esp. reg.	93	3677	59503
Núm. reg. rec.	94	3661	57128
Diferença esp. - rec.	-1	16	2375
Dif. Perc.	-0,5 %	0,43 %	3,99 %
Média tempo rec.	123,9 ms	117,5 ms	149,7 ms
Desv. Padrão tempo rec.	16,3 ms	28,5 ms	45,1 ms
Esperança temp. rec.	120,1 ms	97,2 ms	47,3 ms

Fonte: Autores

entre mensagens conseguiu manter a implementação com níveis aceitáveis de perda, principalmente para os testes 1 e 2. Essa estabilidade se deve, em grande parte, ao poder do recurso computacional do ambiente de desktop virtualizado e pelo fato da comunicação acontecer em uma rede privada, que reduz perdas por consequência da própria rede.

Figure 1. Resultados do Teste 3



Fonte: Autores.
Legenda: a linha azul mostra o envio em segundos em relação as amostras recebidas.

Para o teste 3, no entanto, o número de mensagens perdidas reflete exatamente o momento em que a carga de processamento se elevou e a escalabilidade das réplicas de *pods* da implementação não acompanhou as chamadas, como pode-se observar no desvio observado na figura 1 e no vídeo https://youtu.be/pfRJ5pPSOls?si=NAdTbIixlSXNeMRw

5.2. Resultados do teste com Ambiente Embarcado

A Tabela 2 mostra os resultados obtidos nos quatro testes realizados na implementação do K3s e do HiveMQ no Raspberry Pi 4B.

Table 2. Resultados dos testes do ESP32 publicando em *broker* no Raspberry Pi

Parâmetro	**Teste 1**	**Teste 2**	**Teste 3**	**Teste 4**	
Tempo entre pub.	1 s	10 ms	100 ms	100 ms	
Quant. ESP32	1	1	1	2	
Tempo de exper.	215 s	562 s	417 s	486 s	
Máximo PODs HPA	1	20	1	1	
Final IP ESP32	.107	.107	.107	.107	.134
Núm. esp. reg.	215	56208	4168	4860	4860
Núm. reg. rec.	216	21905	3525	1435	1322
Diferença esp. - rec.	-1	34303	643	3425	3538
Dif. Perc.	-0,5 %	61,0 %	15,4 %	70,5 %	72,3 %
Média tempo rec.	90,3 ms	319,5 ms	485,0 ms	4.419,7 ms	3104,3 ms
Desv. Padrão tempo rec.	138,5 ms	631,5 ms	1095,0 ms	6025,6 ms	4151,3 ms
Esperança temp. rec.	59,5 ms	260,7 ms	379,3 ms	3853,0 ms	2768,0 ms

Fonte: Autores

Observa-se que há, claramente, uma deterioração do tempo de resposta e da quantidade de mensagens efetivamente recebidas conforme se aumenta a taxa de envio. O teste em 100 ms demonstra que há perdas consideráveis na perda de pacotes (15,4 %), e abaixo dessa taxa há perdas de pacotes na acima de 60 s. Vê-se também que a latência de envio da mensagem é grande para as taxas altas, chegando a médias de 4,4 s. Observa-se, também, que o escalonamento ocorreu somente no Teste 2, o que pode indicar que o Teste 4 pode ter sido influenciado por fatores externos.

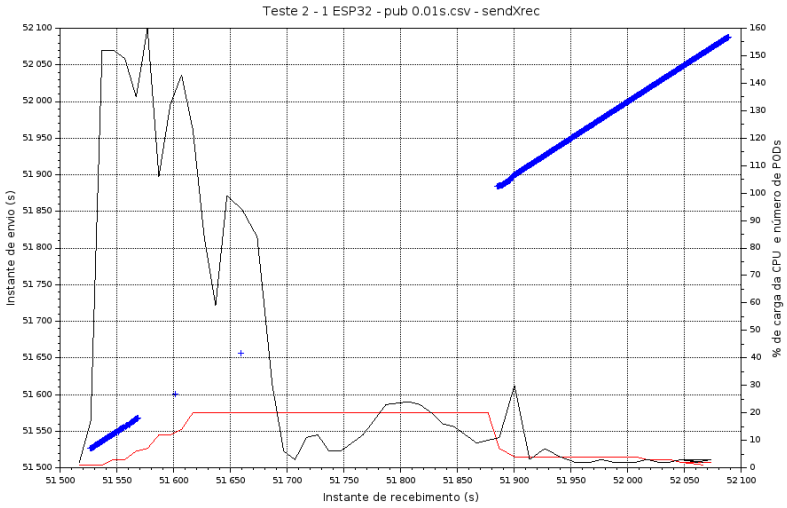
As Figuras 2 e 3 mostram o comparativo entre os instantes de recebimento e de envio dos dados nos testes 2 e 3, respectivamente. Os trechos em branco na curva azul representam os momentos em que não houve envio do dado do *broker* para o *subscriber*, e sua interrupção coincide com o aumento de carga. Vê-se que o aumento de carga no Teste 2 levou ao escalonamento ao máximo de PODs (20) e, no Teste 3 não houve essa necessidade.

5.3. Resultados do teste na AWS com EKS

A Tabela 3 mostra os resultados obtidos nos seis testes realizados na implementação do K3s e do HiveMQ no EKS da AWS.

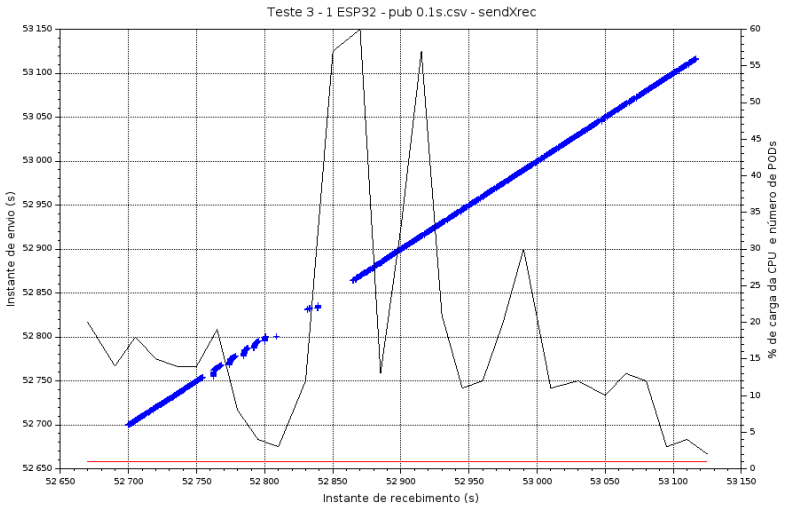
Pelos resultados obtidos pode-se observar que o cluster não precisou escalonar em nenhuma condição quando recebendo dados dos ESP32. No teste com envio a 1 s pode-se observar um tempo anormal de recebimento, dado por algumas amostras que chegaram a demorar mais de 17 s. Esse comportamento demanda mais investigação. A perda de mensagens para envios na casa abaixo de 100 ms ainda foi alta, apesar da capacidade de processamento da nuvem. Isso pode indicar que o teste pode ter sido afetado pela rede. Por fim, o melhor comportamento ocorreu com o PC enviando a 1 ms, onde não houve perda de mensagens e o tempo de recebimento foi da ordem de 250 ms.

Figure 2. Resultados do Teste 2



Fonte: Autores.
Legenda: azul – cada cruz indica uma mensagem enviada; vermelho – número de PODs; preto – carga da CPU.

Figure 3. Resultados do Teste 3



Fonte: Autores.
Legenda: azul – cada cruz indica uma mensagem enviada; vermelho – número de PODs; preto – carga da CPU.

Table 3. Resultados dos testes do ESP32 publicando em *broker* no AWS EKS

Parâmetro	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5	Teste 6
Tempo entre pub.	1 s	10 ms	10 ms	1 ms	100 ns	1 ms
Quant. ESP32	1	1	2	1	PC	PC
Tempo de exper.	183 s	425 s	360 s	281 s	82 s	292 s
Máximo PODs HPA	1	1	1	1	13	6
Núm. esp. reg.	183	42522	35969	281095	1200954	243120
Núm. reg. rec.	184	24154	14591	28744	17225	243120
Diferença esp. - rec.	-1	18368	21378	252351	1183729	0
Dif. Perc.	-0,5 %	43,2 %	59,4 %	89,8 %	98,6 %	0 %
Média tempo rec.	1,4 s	379,4 ms	636,9 ms	390,7 ms	20,7 s	282,9 ms
Desv. pad. tp. rec.	3 s	452,9 ms	1,4 s	686,1 ms	12,3 s	590,8 ms
Esperança temp. rec.	1,3 s	141,9 ms	358,5 ms	242,9 ms	19,4 s	250,1 ms

Fonte: Autores

6. Conclusão

Os resultados obtidos ao longo dos testes demonstram que a implementação do sistema MQTT foi capaz de operar de maneira estável e com baixa taxa de perda de mensagens em cenários controlados, como no ambiente virtualizado de desktop. A escalabilidade dos *pods* foi um fator chave para manter o sistema eficiente, mesmo com tempos de envio reduzidos, enquanto a rede privada contribuiu para minimizar perdas associadas à infraestrutura de comunicação. Embora as perdas e latências tenham aumentado nos testes com taxas de envio mais rápidas, especialmente abaixo de 100 ms, o sistema foi capaz de manter um desempenho dentro de limites aceitáveis em condições mais moderadas, sugerindo que o ambiente virtualizado é adequado para aplicações de baixa taxa de envio.

Por outro lado, a análise do desempenho em ambiente embarcado revela a necessidade de uma abordagem mais cuidadosa. A elevada latência e as perdas de pacotes em taxas de envio mais rápidas indicam que o hardware e a infraestrutura de comunicação desempenham papéis cruciais na eficácia do sistema. O sistema apresentou limitações quando exposta a condições de alta demanda, evidenciando que o ambiente embarcado requer não apenas um hardware eficiente, mas também uma rede confiável para assegurar a entrega de mensagens em tempo real.

A implementação de soluções em tempo real em ambientes de nuvem para aplicações de Internet das Coisas apresenta desafios que podem comprometer a eficácia e a eficiência do sistema. Apesar disso, o teste com o PC, substituindo o ESP32, enviando a 1 ms mostrou um desempenho superior, sem perda de mensagens e com latências relativamente baixas, reforçando a importância do ambiente e da infraestrutura de rede para o sucesso da aplicação. Embora a nuvem ofereça recursos escaláveis e poderosos para processamento de dados e armazenamento, a latência relacionada à comunicação com servidores remotos pode dificultar a operação em tempo real.

References

[Amazon 2024] Amazon (2024). Getting started with amazon EKS – AWS management console and AWS CLI - amazon EKS.

https://docs.aws.amazon.com/eks/latest/userguide/getting-started-console.html.

[Centofanti et al. 2023] Centofanti, C., Tiberti, W., Marotta, A., Graziosi, F., and Cassioli, D. (2023). Latency-Aware Kubernetes Scheduling for Microservices Orchestration at the Edge. 2.

[Espressif 2024] Espressif (2024). Esp32 series - datasheet version 4.7.

[Figueroa et al. 2023] Figueroa, C., Knowles, T., Kukreja, V., and Lung, C.-H. (2023). IoT Management with Container Orchestration. 3.

[HiveMQ CE Project Authors 2024] HiveMQ CE Project Authors (2024). Hivemq community edition docs. https://github.com/hivemq/hivemq-community-edition/wiki.

[IoT Analytics 2024] IoT Analytics (2024). State of IoT 2024: Number of connected IoT devices growing 13% to 18.8 billion globally. https://iot-analytics.com/number-connected-iot-devices/.

[K3s Project Authors 2024] K3s Project Authors (2024). What is k3s? https://docs.k3s.io/.

[Koziolek and Eskandani 2023] Koziolek, H. and Eskandani, N. (2023). Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift. 1.

[Koziolek et al. 2020] Koziolek, H., Grüner, S., and Rückert, J. (2020). A Comparison of MQTT Brokers for Distributed IoT Edge Computing. 10.

[Leskinen 2020] Leskinen, A. (2020). Applicability of Kubernetes to Industrial IoT Edge Computing System. 1.

[Raspberry Pi Foundation 2024] Raspberry Pi Foundation (2024). Raspberry pi for home. https://www.raspberrypi.com/for-home/.