Revista Brasileira de Computação Aplicada, Novembro, 2024

DOI: 10.5335/rbca.v16i3.15778
Vol. 16, Nº 3, pp. 37–47
Homepage: seer.upf.br/index.php/rbca/index

ARTIGO ORIGINAL

MQTT Poller: um arcabouço endógeno para avaliação de desempenho do protocolo MQTT em larga escala

MQTT Poller: an endogenous framework for large-scale MQTT protocol performance evaluation

Diogo Maciel da Cunha [1], and Marco Aurélio Spohn [2]

[1]Universidade Estadual de Campinas - Unicamp, [2]Universidade Federal da Fronteira Sul - UFFS

*d249418@dac.unicamp.br;†marco.spohn@uffs.edu.br

Recebido: 19/04/2024. **Revisado:** 11/11/2024. **Aceito:** 30/11/2024.

Resumo

O MQTT é um protocolo da camada de aplicação com suporte à comunicação síncrona e assíncrona entre clientes. Ele emprega um servidor (*broker*) intermediário entre os clientes e um sistema de tópicos no processo de envio e recepção de mensagens. Graças à sua simplicidade e baixo custo de controle, ele é comumente empregado no desenvolvimento de aplicações na Internet das Coisas (*Internet of Things*, IoT), sendo também do interesse de vários trabalhos de pesquisa como, por exemplo, os relacionados à escalabilidade do protocolo. Neste contexto, observa-se que há uma carência de instrumentos de avaliação de desempenho e orquestração de experimentos. Este trabalho apresenta um arcabouço para avaliação de desempenho do protocolo MQTT em uma infinidade de configurações e topologias de sistemas. O desenvolvimento do arcabouço segue um modelo de comunicação endógeno (*i.e.*, baseado no próprio esquema de tópicos do MQTT) para controle e interação entre os elementos da arquitetura. Partindo de uma ferramenta de avaliação de desempenho de referência (*MQTTLoader*), permite-se configurar um cenário de avaliação que possibilite executar múltiplas instâncias dessa ferramenta, sob a coordenação de um orquestrador, em máquinas inclusive hospedadas em diferentes domínios administrativos. Um estudo de caso demonstra as funcionalidades e a praticidade em se utilizar o arcabouço para a realização de experimentos de avaliação de desempenho do MQTT.

Palavras-Chave: MQTT; Publicador/Assinante; Avaliação de desempenho.

Abstract

MQTT is an application layer protocol that supports synchronous and asynchronous client communication. It employs an intermediary server/broker between clients and a topic system to send and receive messages. Thanks to its simplicity and low control cost, it is commonly used in the development of applications for the Internet of Things (IoT), and it is also of interest in several research works, such as those related to the scalability of the protocol. In this context, it is observed that there is a lack of performance evaluation and experiment orchestration instruments. This work presents a framework for evaluating the performance of the MQTT protocol in many scenarios. The development of the framework follows an endogenous communication model (*i.e.*, based on the MQTT topic schema) for control and interaction between the architecture's elements. Starting from a reference performance evaluation tool (*MQTTLoader*), it is possible to configure an evaluation scenario that makes it feasible to run multiple instances of this tool, under the coordination of an orchestrator, on machines possibly hosted in different administrative domains. A case study demonstrates the functionalities and practicality of using the framework to conduct MQTT performance evaluation experiments.

Keywords: MQTT; Publisher/Subscriber; Performance evaluation.

1 Introdução

O MQTT é um protocolo amplamente empregado no desenvolvimento de aplicações de Internet das Coisas (Internet of Things, IoT) que compartilham as seguintes características (Canek et al., 2022): (i) baixa largura de banda, (ii) presença de dispositivos autônomos e (iii) dispositivos com baixa capacidade computacional (*i.e.*, memória principal, armazenamento secundário e processamento limitados). O protocolo se destaca pela sua simplicidade (baixo custo), contemplando os modos de comunicação síncrona e assíncrona com baixo custo operacional e tamanho de mensagens adaptável (Banno and Yoshizawa, 2022). Questões relativas a sua escalabilidade e segurança tem recebido atenção especial da comunidade acadêmica e da indústria (Spohn, 2022; Dinculeană and Cheng, 2019).

A configuração básica do MQTT é baseada em um único servidor, sendo este um potencial gargalo e um ponto único de falhas (Spohn, 2022). Para contornar essas limitações, empregam-se técnicas de escalabilidade vertical e horizontal adaptadas às peculiaridades do formato de comunicação cliente para cliente (Do inglês *client to client*) e ao paradigma de publicação e assinatura (do inglês *publish and subscribe*) (Ribas and Spohn, 2022). As técnicas de escalabilidade vertical focam em otimizações no código do servidor que permitam o emprego eficiente de mais recursos computacionais no contexto de uma mesma máquina. Já a escalabilidade horizontal foca em alternativas de distribuição da carga de trabalho entre um grupo variável de máquinas via técnicas de agrupamento (do inglês, *clustering*) (Detti et al., 2020) e federação (Ribas and Spohn, 2022), obtendo-se assim um grau flexível de disponibilidade.

Para avaliar o desempenho de soluções escaláveis, necessita-se de ferramentas que automatizem o processo de geração de carga (*i.e.*, instanciação de clientes) no sistema. Nesse contexto, nota-se que há um número limitado de ferramentas, publicamente disponíveis, para avaliação do protocolo MQTT, sendo a maioria destas descontinuadas (Spohn, 2022). Além disso, a maioria das ferramentas não possibilita, ou torna enfadonho, o processo de automatização da execução dos experimentos e a consequente coleta das estatísticas de desempenho. Assim, uma grande carga de trabalho é imposta nos interessados pelo tema por precisarem adaptar uma ferramenta para os seus testes e realizar as instanciações de clientes e as coletas de resultados de forma manual.

Nesse contexto, a contribuição principal deste trabalho consiste em oferecer à comunidade um arcabouço que facilite a realização de testes de escalabilidade de servidores MQTT. Para tanto, o sistema (denominado *MQTT Poller*[1]) opera como uma interface de orquestração de instâncias do utilitário *MQTTLoader* (2022), o qual é responsável pela criação de clientes MQTT e pela produção das estatísticas de desempenho.

O restante deste trabalho está organizado conforme descrito a seguir. A Seção 2 apresenta os principais fundamentos e conceitos do protocolo MQTT para contextualização do problema principal tratado nesse trabalho. A Seção 3 apresenta os principais trabalhos relacionados, enquanto a Seção 4 descreve o sistema desenvolvido. A Seção 5 apresenta um estudo de caso visando demonstrar os requisitos funcionais do arcabouço. As conclusões e trabalhos futuros são apresentados na Seção 6.

2 MQTT

O MQTT é um protocolo de aplicação que adota o paradigma de comunicação publicador/assinante, *P/S (Publisher/Subscriber)*. Clientes comunicam-se por intermédio de um servidor, *broker*, enviando e recebendo mensagens rotuladas em tópicos (*MQTT*, 2019). Os clientes podem assinar e publicar em tópicos especificando um nível de qualidade de serviço (QoS). Como o *broker* é o intermediário na comunicação entre clientes, tem-se à disposição suporte a comunicação assíncrona. Nessa modalidade, caso o cliente não esteja conectado, as mensagens são armazenadas no servidor para posterior entrega quando da reconexão do cliente. A comunicação assíncrona favorece sistemas com dispositivos limitados em recursos e com enlaces de comunicação sujeitos a perdas de pacotes acentuadas: além de transferir parte significativa do controle ao servidor, permite que o cliente receba todas as mensagens quando as condições de comunicação se tornarem mais favoráveis.

O *broker* é responsável por receber e encaminhar mensagens entre clientes, controlar os tópicos de comunicação e gerenciar as sessões. Ele é instanciado, normalmente, em uma máquina com mais recursos do que os clientes porque é responsável pela maior parte da carga de controle da aplicação. Em decorrência, torna-se um gargalo e ponto único de falha (Spohn, 2022), pois sem ele não há comunicação entre os clientes. Clientes conectam-se ao servidor via TCP, identificando-se (*i.e.*, usuário e senha) e passando informações de controle de sessão (*e.g.*, *clear session flag* e *keepalive timeout*).

Destaca-se que a primeira conexão de um cliente com o *broker* é tratada como o início de uma sessão, a qual pode ser limpa (*i.e.*, assinaturas anteriores são descartadas) ou persistente (assinaturas anteriores são mantidas). O *broker* utiliza um identificador de cliente (*ClientID*) para controlar as sessões. O cliente também pode encerrar a conexão com o *broker* a qualquer momento, enviando uma mensagem de desconexão (*MQTT*, 2019). Ao retomar uma comunicação persistente, o *broker* enviará ao cliente todas as mensagens não lidas relativas aos tópicos assinados anteriormente, respeitando-se a hierarquia de qualidade de serviço e as *flags* de controle (*e.g.*, retenção da última mensagem publicada no tópico).

As mensagens trocadas com o servidor são rotuladas em tópicos: o servidor mantém o controle de todos os assinantes interessados em cada tópico, realizando a entrega das mensagens conforme a modalidade de inscrição de cada cliente. O protocolo adota níveis de Qualidade de Serviço (*Quality of Service (QoS)*) e a *flag* de retenção para garantir que mensagens prioritárias não se percam no processo, decidindo assim o procedimento adequado para que as mensagens cheguem da forma acordada em seus destinos. Portanto, ao assinar ou publicar em um tópico deve-se explicitar o nível de QoS pretendido, bem como a sinalização

[1]Todos os códigos fontes relacionados ao sistema desenvolvido estão publicamente disponíveis no GitHub em `https://github.com/namelew/MQTTPoller`.

para retenção da última mensagem publicada.

A retenção da última mensagem publicada em um tópico permite que os assinantes recebam esta mensagem imediatamente após uma primeira conexão ou reconexões. Desta forma, também pode servir como um mecanismo de consulta do estado atual da entidade atrelada ao publicador (*e.g.*, status atual de um dispositivo como, por exemplo, ligado ou desligado). Ressalta-se que ocorre a retenção de apenas uma (a última) mensagem por tópico. Em contrapartida, com relação ao QoS de uma conexão, há um gerenciamento mais minucioso em relação às mensagens ainda não entregues aos assinantes. A opção de QoS 0 funciona na modalidade de melhor esforço (*i.e.*, sem confirmação a nível de aplicação); ou seja, mensagens publicadas enquanto o cliente estiver desconectado serão perdidas. A opção de QoS 1 oferece entrega confiável, mas podem ocorrer duplicatas durante o processo de entrega. Na opção de QoS 2, também confiável, não há a ocorrência de duplicatas. Para as opções de QoS 1 ou 2, tem-se a possibilidade de manter armazenadas no broker as mensagens que ainda não foram entregues aos assinantes: quando um cliente realizar uma nova conexão, todas as mensagens serão entregues; ou seja, tem-se entrega confiável a nível de aplicação.

Atualmente, as duas versões do MQTT mais utilizadas são a 3.1.1 e a 5 (ambas oferecem as funcionalidades apresentadas anteriormente) (*MQTT*, 2019). Dentre as melhorias presentes na v5, destacam-se: assinaturas compartilhadas, configuração otimizada de parâmetros de sessão (*e.g.*, tamanho máximo da mensagem aceita pelo assinante), pseudônimos de tópicos (*topic aliases*), definição do tempo de vida de uma mensagem e um limite de retransmissões para as qualidades de serviço 1 e 2, reduzindo consideravelmente a carga de controle do *broker* (Spohn, 2022).

Quando o enfoque está em avaliar o desempenho de um servidor MQTT, há um conjunto bem definido de parâmetros que podem ser explorados (vide Tabela 1) (Spohn, 2022). Por se tratar de uma rede centralizada na entidade do *broker*, muitos dos parâmetros de configuração dizem respeito à demanda de trabalho conferida a ele (*e.g.*, quantidade de clientes, periodicidade e tamanho das mensagens). Entretanto, outros aspectos da rede também devem ser analisados como, por exemplo, as estatísticas referentes a perda de pacotes, latência e vazão. O quantitativo de clientes (publicadores e assinantes) permite, gradativamente, avaliar a reação do servidor à carga de controle referente às sessões e conexões. O fluxo de dados, manipulável através do número de tópicos, tamanho e frequência das mensagens, qualidade de serviço, e sessões persistentes ampliam o conjunto de cenários para avaliação. A localização física de todos os elementos envolvidos (*i.e.*, broker e clientes) também merece atenção na definição dos cenários de avaliação.

3 Trabalhos Relacionados

Até onde foi possível constatar, existe um número reduzido de ferramentas de avaliação de desempenho do MQTT. Todas as ferramentas pesquisadas flexibilizam a configuração dos parâmetros citados na seção anterior; no entanto, elas diferem no processo de execução dos experimentos e nos procedimentos de apresentação das métricas de desempenho. Algumas das ferramentas incorporam parâmetros adicionais como, por exemplo, assinaturas compartilhadas e a geração de mensagens codificadas. Entretanto, a principal diferença entre elas está na instanciação dos clientes: publicadores e assinantes via um único processo (*MQTTLoader* e *MQTT broker latency measure tool*) e processos dedicados para cada categoria de clientes (*MQTT-Malaria* e *MQTT-JMeter*).

Objetivando testar a escalabilidade do recurso de assinatura compartilhada disponível na versão 5 do MQTT, Banno and Yoshizawa (2022) desenvolveram uma ferramenta denominada *MQTTLoader* (*MQTTLoader*, 2022). Implementada em linguagem *Java*, ela pode ser executada em qualquer sistema operacional, inclusive possibilitando a instanciação de clientes em múltiplas máquinas simultaneamente. No entanto, fica a cargo do usuário realizar a orquestração dos experimentos e a coleta das estatísticas. A saída padrão dos resultados de desempenho é via terminal mas, opcionalmente, pode-se optar pela geração de um arquivo em formato CSV com detalhamento de todos os eventos relevantes registrados. As grandezas computadas são a vazão máxima e média, número de mensagens publicadas e recebidas e latência máxima e média.

O *MQTT broker latency measure tool* é uma ferramenta desenvolvida em *Go* por Jianhui e Xiang (*MQTT broker latency measure tool*, 2018). Pelas características da linguagem, pode ser executada em uma ampla gama de sistemas operacionais mas, ao contrário do *MQTTLoader*, a ferramenta precisa ser compilada para a arquitetura alvo. Seus parâmetros são semelhantes ao da ferramenta anterior, embora adote algumas regras mais restritivas como, por exemplo, exige um quantitativo par de clientes e a definição dos parâmetros de configuração apenas via *flags* de comando ao invés de um arquivo de configuração. Em relação às métricas de desempenho, estas são mais amplas que na ferramenta anterior, incluindo a quantidade de publicações bem sucedidas e falhas e a latência mínima e máxima, com avaliação da dispersão, para mensagens publicadas e recebidas. A saída destes dados pode ser via terminal, em formato texto ou JSON. Ao contrário do *MQTTLoader*, não possibilita a execução de experimentos em múltiplas máquinas. O desenvolvimento da ferramenta encontra-se descontinuado, com a última atualização datada de agosto de 2021.

O *MQTT-Malaria* (*mqtt-malaria*, 2013) é uma ferramenta desenvolvida em linguagem *Python*. Ela possibilita manipular livremente o número de publicadores e assinantes, as características e o quantitativo de mensagens e a alocação de clientes em múltiplos processos. Os processos *malaria subscribe* e *malaria publish* desempenham as funções dos assinantes e dos publicadores, respectivamente. O *malaria subscribe* recebe como parâmetros o quantitativo de publicadores, mensagens por publicador, endereço do *broker*, QoS da assinatura, tópico e identificador do cliente (*i.e.*, *clientID*). A execução é encerrada após atingir a cota, computada como sendo o resultado do produto do número de publicadores e do número de mensagens por publicador. O retorno ao final da execução inclui o número de clientes monitorados, número total de mensagens recebidas, tempo total do experimento, vazão, quantidade

Tabela 1: Avaliação de desempenho do MQTT: parâmetros de configuração

Parâmetro	Valores	Descrição
broker	*string*	endereço de rede do *broker* alvo
n. publicadores	0-n	quantidade de publicadores
n. assinantes	0-n	quantidade de assinantes
n. mensagens	0-n	quantidade de mensagens publicadas
tamanho mensagem	0-n *bytes*	tamanho em *bytes* das mensagens
flag de retenção	true,false	marcador de retenção
tópico	*string*	nome do tópico de teste
QoS assinatura	0-2	qualidade de serviço das assinaturas
QoS publicação	0-2	qualidade de serviço das publicações

de mensagens duplicadas e a média, desvio padrão, e os valores mínimo e máximo da latência. O processo *malaria publish* recebe como argumentos o *clientID*, endereço do *broker*, QoS de publicação, número total de mensagens, tamanho da mensagem, frequência de publicações (*i.e.*, mensagens por segundo) e a quantidade de processos que serão instanciados. O publicador encerra quando finalizar o envio de todas as mensagens. Dentre os resultados produzidos ao final, tem-se a percentagem de mensagens enviadas corretamente e as medidas referentes à latência de publicação: média, desvio padrão, máximo e mínimo. No momento, o desenvolvimento da ferramenta está descontinuado, com a última atualização datada de 2019 e o último *release* referente a 2013.

O *MQTT-JMeter* (*mqtt-jmeter*, 2014) é um *plugin* para avaliar o desempenho do MQTT com o *JMeter* (uma ferramenta para realizar testes de carga em sistemas computacionais) (Apache Software Foundation, 2024). Ele possibilita criar planos de teste tanto para publicadores quanto para assinantes. No plano de assinatura, são definidos parâmetros comuns às ferramentas anteriores como, por exemplo, a quantidade máxima de mensagens recebidas, semelhante ao *malaria subscribe*, e o tempo máximo de espera, parecido com o *MQTTLoader*. No plano de publicação, há opções particulares como, por exemplo, a definição da codificação da mensagem (UTF-8, UTF-16, US-ASCII,UTF-16LE, UTF-16BE, ISO-8859-1), o conteúdo desta (gerado automaticamente, fixo, texto ou um *array* de valores aleatório) e a estratégia de publicação nos tópicos (*round-robin* ou *random*). Os dois planos (*i.e.*, publicadores e assinantes) são independentes, suportando, inclusive, clientes externos (*i.e.*, não instanciados pela ferramenta). No que tange a saída dos resultados de desempenho, a ferramenta se destaca por apresentar uma interface gráfica que permite a geração de gráficos e listagens detalhados. No momento, o desenvolvimento da ferramenta encontra-se descontinuado, com a última atualização datada de 2014.

4 MQTT Poller

Partindo da ferramenta *MQTTLoader*, desenvolveu-se um arcabouço (denominado *MQTT Poller*) que possibilita orquestrar a execução de experimentos de avaliação de desempenho do MQTT em cenários de complexidade variada. Os requisitos a serem cumpridos são: (i) compatibilidade com os parâmetros da ferramenta alvo, (ii) tratamento de erros, (iii) coleta e apresentação dos resultados finais, e (iv) viabilidade de instanciação das entidades envolvidas (clientes e servidores) em múltiplas máquinas [2]. Todos os códigos fontes relacionados ao sistema desenvolvido estão publicamente disponíveis no GitHub em `https://github.com/namelew/MQTTPoller`.

4.1 Arcabouço

Os dois principais elementos/entidades da arquitetura do arcabouço são o orquestrador e os trabalhadores (*workers*). Optou-se pelo próprio mecanismo P/S do MQTT como método de comunicação e coordenação entre as entidades, aproveitando-se da modalidade de comunicação assíncrona para gerenciar a execução dos experimentos, reduzindo ou eliminando a possibilidade de perda na coleta de resultados. Como linguagem principal de desenvolvimento, optou-se por Go (The Go Authors, 2024), sobretudo pelo seu suporte a programação concorrente, o tratamento de entrada e saída de rede e a sua crescente comunidade. Ademais, a interface com o usuário foi separada da parte principal da aplicação, para facilitar manutenções futuras e permitir uma maior personalização. Para tanto, esta foi desenvolvida em *Typescript* com o *framework Next.js* (*Next.js by Vercel - The React Framework*, 2022) e sua comunicação com a parte principal da aplicação ocorre via uma API REST executada sobre o protocolo HTTP em comunicação direta com o orquestrador (Fig. 1).

4.1.1 *Orquestrador*

O orquestrador é responsável por receber as requisições dos usuários, disparar os experimentos e coletar os resultados (Fig. 2). As requisições são recebidas via um interpretador HTTP (porta definida pelo usuáro), produzindo um retorno no formato JSON com os resultados (vide exemplo na Fig. 4). Cada orquestrador gerencia um grupo de trabalhadores, os quais são diretamente responsáveis pela execução dos experimentos. Como enfatizado anteriormente, diz-se que se adota uma abordagem endógena de comunicação por utilizar o próprio protocolo MQTT como meio de comunicação entre o orquestrador e os trabalhadores, via um conjunto específico de tópicos de controle. O orquestrador identifica unicamente cada trabalhador, gerenciando sua execução e tratando eventuais erros de execução.

O orquestrador mantém a persistência dos dados relacionados aos experimentos e aos trabalhadores, utilizando-

[2] Exemplo: pode-se instanciar dois processos do *MQTTLoader* em máquinas distintas de forma que uma das máquinas execute 10 publicadores e a outra execute 5 assinantes, assumindo-se o mesmo tópico alvo.

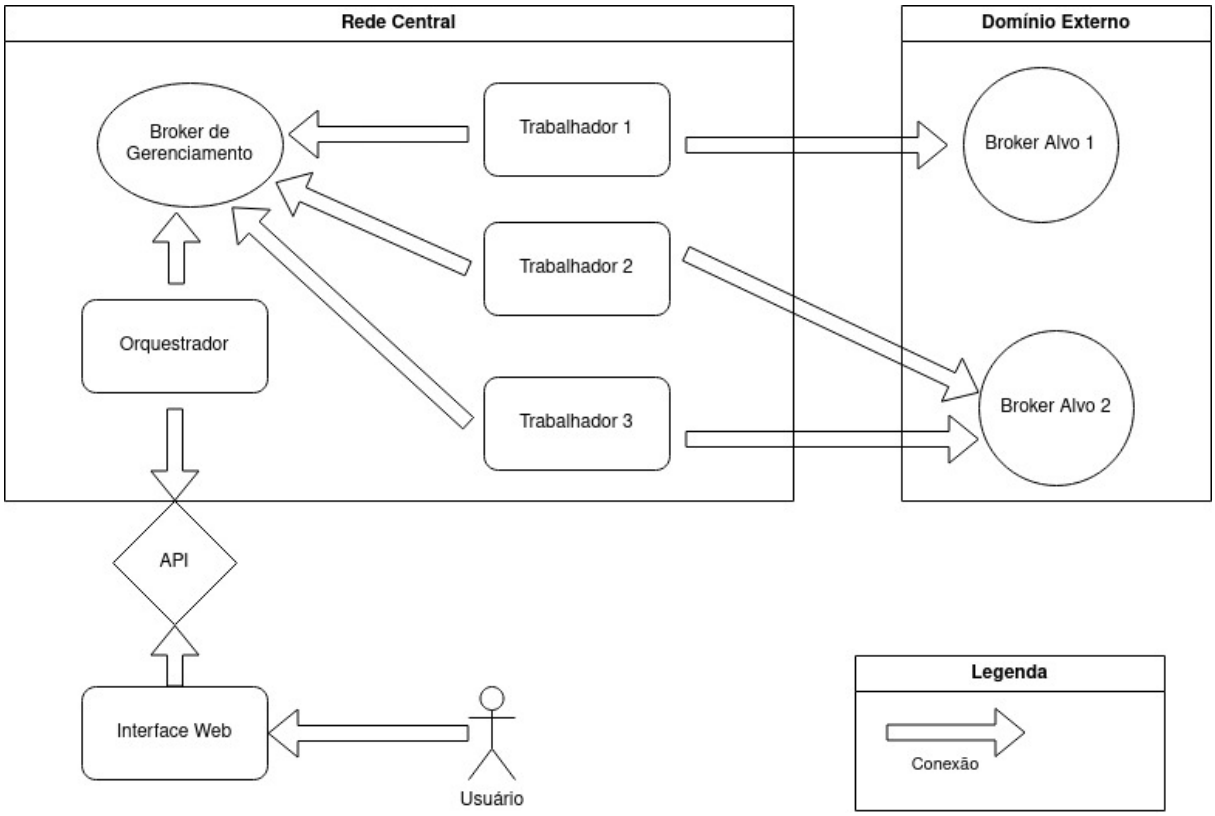

Figura 1: Arcabouço: arquitetura geral e agentes externos

se arquivos em codificação binária e duas árvores B baseadas na implementação disponibilizada por Josh Baker (*btree*, 2014). Para o armazenamento e indexação em memória principal, emprega-se uma árvore B para os trabalhadores e outra para os experimentos. Essas estruturas são armazenadas em dois arquivos cada, um deles contendo os dados e outro com os índices da árvore. Em intervalos de três segundos, realiza-se uma comparação entre os valores armazenados nos arquivos e em memória, persistindo as diferenças entre ambos, sempre dando primazia ao valor em memória. Ao iniciar, o orquestrador carrega em memória principal os dados armazenados em ambos os arquivos.

Para a comunicação com os trabalhadores, tem-se um cliente MQTT implementado com auxílio da biblioteca *Paho MQTT* (*Eclipse Paho*, 2019) e os seguintes tópicos de controle (Fig. 3): (i) registro de novo trabalhador, (ii) autenticação (*login*) de trabalhador, (iii) controle de estado do trabalhador e (iv) controle de experimentos. Há uma limitação da biblioteca *Paho* em linguagem Go que impossibilita a assinatura de tópicos em multinível (*i.e.*, não é possível que o orquestrador assine todos os tópicos dos trabalhadores simultaneamente).

É através do tópico de registro que um trabalhador recebe suas credenciais/identificação no sistema. O gerenciamento do estado do trabalhador e dos seus experimentos fica atrelado ao seu identificador. O tópico de *login* serve tanto para a primeira conexão como para eventuais reconexões; ou seja, é através dele que o trabalhador inicia sua sessão na aplicação e por onde informa uma reconexão. Os demais tópicos serão detalhados na seção referente aos trabalhadores; no entanto, adianta-se que servem ao propósito de gerenciamento dos trabalhadores (vide Tabela 2).

No tópico de registro, o orquestrador recebe como identificador temporário do trabalhador uma *string* de 16 *bytes* definida conforme a normativa RFC 4122 (*A Universally Unique IDentifier (UUID) URN Namespace*, 2005) e gerada pela biblioteca disponibilizada pelo *Google* (*uuid: Go package for UUIDs based on RFC 4122 and DCE 1.1: Authentication and Security Services.*, 2011). Após receber a requisição, o orquestrador publica em um tópico de resposta, formado pela concatenação de um nome padrão com o identificador temporário do trabalhador, o *token* de identificação do trabalhador, o qual também é uma *string* no formato padrão. Para realizar o *login*, o trabalhador deve publicar esse identificador no tópico de *login* assinado pelo orquestrador. O identificador do trabalhador também é concatenado a todos os tópicos deste, transformando-os em subtópicos.

Para disparar um experimento, emprega-se um subtópico dentro do escopo dos trabalhadores. O orquestrador publica uma requisição nesse tópico e aguarda pelo retorno com os resultados produzidos pelo trabalhador. Para isso, utiliza-se duas informações, os *logs* gerados pelo trabalhador, que informam quando um experimento inicia/finaliza, e uma variável de tolerância definida pelo usuário. A tolerância é o atraso permitido entre as sinalizações (*i.e.*, *heartbeat*) do trabalhador, também utilizado para o controle dos experimentos. O tempo máximo de espera (em segundos) pela finalização de um experimento é obtido

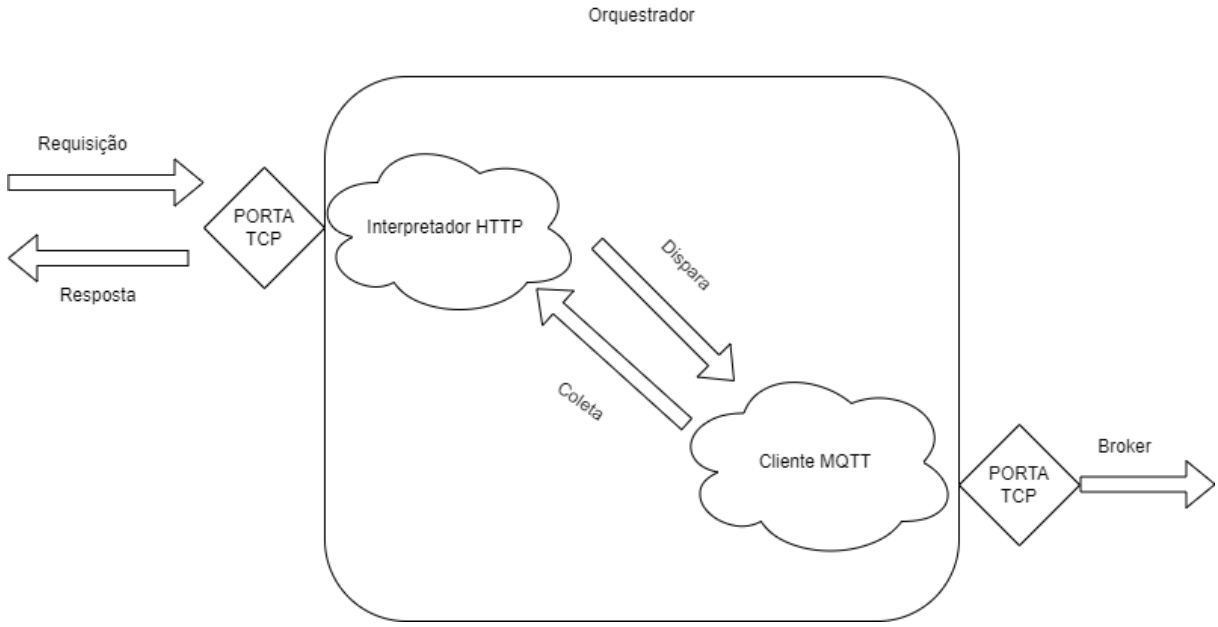

Figura 2: Estrutura de comunicação do orquestrator

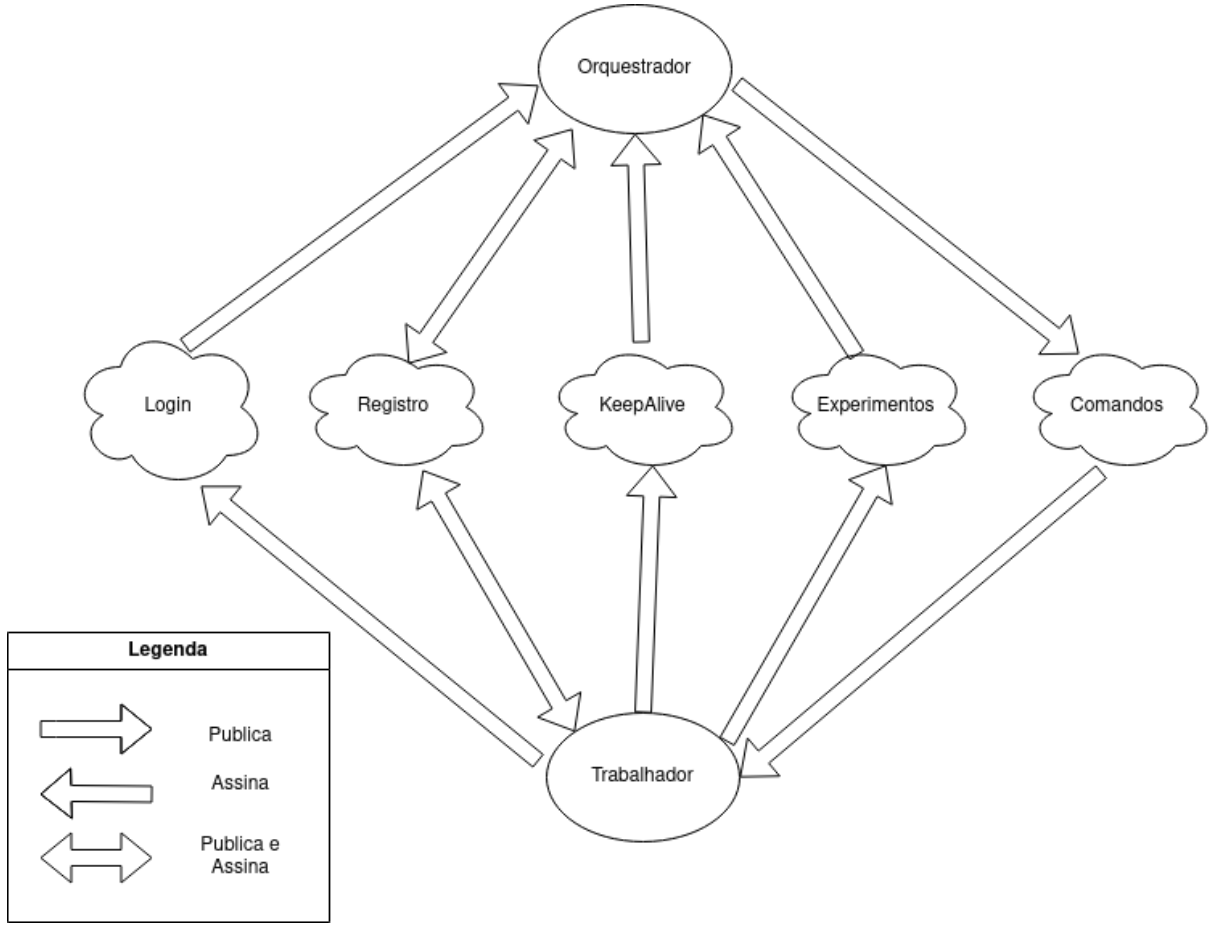

Figura 3: Sistema de tópicos do orquestrador

```
{
   "id": ["e233e988-56f2-4834-853a-085e96a95b67"],
   "description":{
      "broker": "localhost",
      "port": 1883,
      "mqttVersion": 3,
      "numPublishers": 100,
      "numSubscribers": 100,
      "qosPublisher": 2,
      "qosSubscriber": 2,
      "topic": "test",
      "payload": 10000,
      "numMessages": 100000,
      "execTime": 120
   }
}
```


Figura 4: Exemplo de requisição de experimento aceita pelo orquestrador

Tabela 2: Tópicos do orquestrador

Nome	Nomenclatura	Ação	QoS
Registro	Orquestrator/Register	Assina	2
Resposta de Registro	Orquestrator/Register/[ID Trabalhador]	Publica	2
Login	Orquestrator/Login	Assina	2
Batidas de Coração	[ID Trabalhador]/KeepAlive	Assina	0
Resultados Experimentos	[ID Trabalhador]/Experiments/Results	Assina	2
Estado Experimentos	[ID Trabalhador]/Experiments/Status	Assina	2
Tópico de Comando	[ID Trabalhador]/Command	Publica	2

Tabela 3: Rotas da API de comunicação

Nome	Rota	Método
Lista de Trabalhadores	/orquestrator/worker	GET
Detalhes de Trabalhador	/orquestrator/worker/:id	GET
Lista de Experimentos	/orquestrator/experiment	GET
Detalhes Experimento	/orquestrator/experiment/:id	GET
Requisitar Experimento	/orquestrator/experiment/start	POST
Cancelar Experimento	/orquestrator/experiment/cancel/:id	POST
Remover Experimento	/orquestrator/experiment/:id	DELETE

como o resultado da multiplicação dos valores do parâmetro *exectime* do *MQTTLoader* e da tolerância. Em caso de *timeout*, o orquestrador sinaliza um erro na execução do experimento, mas continua esperando por possíveis resultados atrasados.

Para realizar a comunicação com a interface gráfica, foi utilizado um interpretador HTTP, implementado com o *framework Echo* (*High performance, extensible, minimalist Go web framework Echo*, 2019), o qual recebe requisições em uma porta TCP definida pelo usuário. Ele funciona como um servidor HTTP que executa em paralelo ao cliente MQTT, interpretando os comandos do usuário e disparando os experimentos. Nele, existem rotas para o gerenciamento dos trabalhadores e dos experimentos (vide Tabela 3). Com as duas primeiras rotas, obtem-se a informação de quais trabalhadores estão registrados no orquestrador e seus estados atuais. Com as outras rotas, pode-se iniciar novos experimentos, cancelar experimentos em execução, visualizar os parâmetros e resultados dos experimentos já finalizados e excluir registros.

As rotas de visualização e exclusão de registros utilizam os métodos GET e DELETE do HTTP, respectivamente, e não recebem parâmetros pelo corpo da requisição mas podem receber parâmetros no cabeçalho para filtrar o retorno dos dados ou identificar o registro que será excluído. Apenas a rota de criação de experimento recebe um JSON no corpo da requisição, que contém a lista de trabalhadores que executará o experimento e os parâmetros que eles utilizarão no *MQTTLoader*.

4.1.2 *Trabalhadores*

Os trabalhadores são as entidades responsáveis por instanciar os experimentos e coletar seus resultados. A solicitação de um novo experimento chega através de um tópico específico (Comando): o trabalhador, enquanto assinante, recebe a solicitação enviada via publicação do orquestrador. A espera por novos comandos sempre é precedida pelo processo de autenticação do trabalhador e, após esta, sinaliza periodicamente (via publicações de *heartbeats*) ao orquestrador, para que este esteja ciente da presença ativa do trabalhador. Ao receber o comando para iniciar um novo experimento, o trabalhador carrega os parâmetros no arquivo de configuração do *MQTTLoader* e cria uma nova instância deste para, efetivamente, executar os clientes relacionados ao experimento. Após a finalização da instância do *MQTTLoader*, o trabalhador coleta a saída com os resultados, converte estes para o formato JSON e também coleta o arquivo CSV correspondente aos *logs* do experimento. Por fim, retorna ao orquestrador todos os resultados coletados, incluindo a saída de eventuais erros para auditoria. A Fig. 5 ilustra os principais elementos da arquitetura do trabalhador.

O trabalhador não persiste dados relativos aos experimentos; no entanto, mantém em memória principal todas as informações, indexadas via uma estrutura em árvore AVL, referentes aos experimentos instanciados (em execução) desde a inicialização do trabalhador. Cada nodo na árvore refere-se a uma instância do *MQTTLoader*: caso o trabalhador receba do orquestrador uma solicitação de cancelamento de um experimento, basta localizá-lo na árvore e, de posse do identificador do processo correspondente, eliminar ele. O trabalhador explora o recurso de multithreading da linguagem Go para manter a conversação com o orquestrador enquanto os experimentos executam de forma concorrente.

4.1.3 *Interface Web*

A interface *web* é uma forma de facilitar o gerenciamento da ferramenta e permitir mais flexibilidade para o usuário. Ela é composta por duas páginas principais: o gerenciamento dos trabalhadores e a listagem dos experimentos. Por meio da primeira página, o usuário pode requisitar a execução dos experimentos. Nela é listada todos os trabalhadores registrados no orquestrador e o estado de suas conexões. Para iniciar um experimento, o usuário deve selecionar os trabalhadores onde deseja executá-lo e informar os parâmetros de configuração do *MQTTLoader*. Após isso, a requisição é enviada para a API do orquestrador, que repassa os dados para os trabalhadores escolhidos e fica à espera dos resultados.

Na página dos experimentos, o usuário pode acompanhar os experimentos que foram requisitados e os resultados obtidos. Cada registro armazena os parâmetros utilizados e os dados coletados por cada trabalhador onde o experimento foi executado. Nessa página, também é possível realizar o *download* dos valores brutos retornados pelos trabalhadores, no formato JSON, e os arquivos de *log* coletados por cada trabalhador, no formato CSV. Adicionalmente, também são listados eventuais erros que ocorreram durante a execução dos experimentos, bem como é permitido ao usuário apagar registros existentes.

5 Estudo de Caso

Para avaliar as funcionalidades do arcabouço, definiu-se doze cenários de testes divididos em duas categorias: monotrabalhador e multitrabalhador. Estes testes têm como objetivo avaliar três casos de uso do *MQTTLoader*, incluindo situações com clientes distribuídos entre diferentes trabalhadores. Nos testes com monotrabalhador, o resultado do experimento contém uma medição completa de todas as métricas de publicação e assinatura, pois

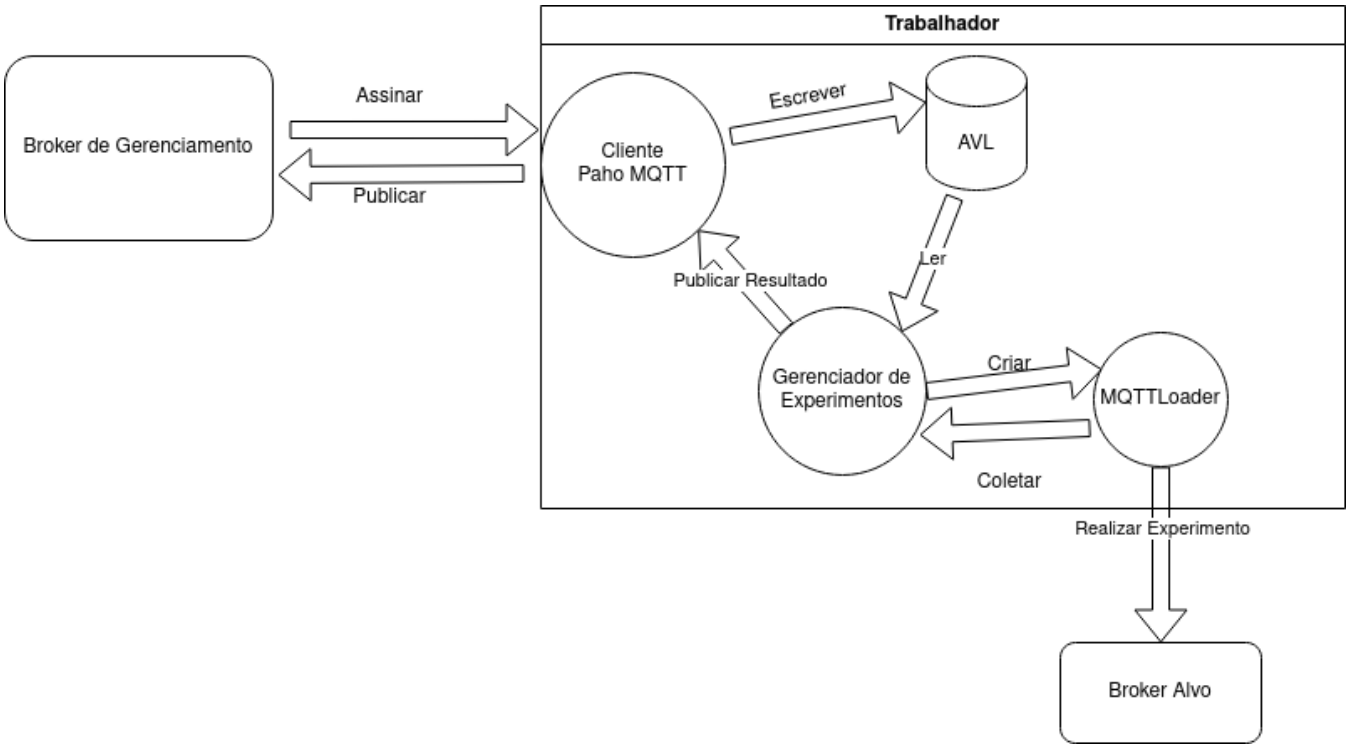

Figura 5: Arquitetura do trabalhador

todos os clientes são gerados em um único processo da ferramenta. Nos casos de multitrabalhador, publicadores e assinantes são separados em processos distintos, portanto o resultado de cada um individualmente contém medições apenas dos seus clientes.

Três cenários de avaliação comumente empregados em testes de escalabilidade são: (i) comunicação um para um, (ii) comunicação de vários para um e (iii) comunicação de um para vários (Spohn, 2022). Avaliar a escalabilidade da comunicação um para um é, possivelmente, o cenário de maior estresse para o servidor, pois exige uma alta quantidade de conexões TCP simultâneas. O segundo caso, também chamado de *fan-in*, é caracterizado por um número maior de publicadores do que assinantes. Já o terceiro caso, também chamado de *fan-out*, ocorre quando há mais assinantes do que publicadores. Estes três cenários foram executados com e sem o uso de assinaturas compartilhadas, uma *feature* disponível apenas na versão 5 do protocolo.

Para avaliar os resultados, realizou-se uma análise do comportamento do sistema ao executar os testes em dois cenários de configuração da rede: (i) rede virtual gerenciada pela plataforma *Docker* e (ii) rede cabeada local. A rede local contém quatro máquinas físicas, com processadores *Intel Core* i7-3770 e 8 GB de memória RAM, executando o sistema operacional Linux (distribuição *Ubuntu*, versão 22). Foram instanciados dois trabalhadores na rede virtual e três na rede física, um orquestrador, uma interface *web* e dois *brokers* Mosquitto, um para gerenciamento e outro para os clientes (*i.e.*, publicadores e assinantes).

Os processos na rede local foram distribuídos da seguinte forma entre as quatro máquinas:

- Uma máquina para os dois *brokers*;
- Outra com o orquestrador e a interface *web*;
- Uma terceira com dois trabalhadores;
- E, por fim, uma quarta máquina com apenas um trabalhador.

Em ambas as redes, os processos foram executados em *containers* provenientes de imagens disponíveis no *Docker Hub*, com as interfaces da rede física empregadas nos testes em rede local. Ao final, as estatísticas referentes à vazão e latência foram coletadas e os arquivos de *log* armazenados localmente para fins de auditoria.

5.1 Resultados e Discussão

Tendo-se ciência que um estudo de caso não possibilita generalizar, procurou-se avaliar os aspectos de funcionamento do sistema proposto considerando os requisitos apresentados na formulação do problema. Portanto, não se pretende aqui apresentar os resultados estatísticos de cada experimento, atentando-se apenas em avaliar os requisitos funcionais do sistema.

Em linhas gerais, pode-se constatar que o sistema conseguiu atender as expectativas em relação a definição, execução e geração dos resultados dos experimentos. No entanto, apesar da execução dos experimentos em si transcorrer dentro do tempo estimado, o cálculo das métricas de desempenho acabou sendo a fase mais demorada do processo de experimentação como um todo.

Observou-se um comportamento inesperado da ferramenta *MQTTLoader* durante uma única execução, levando-a a um estado de *loop*. Como essa situação foge ao escopo

desse trabalho (*i.e.*, a ferramenta que instancia os clientes é apenas um dos requisitos do sistema), adotou-se uma estratégia de tratamento de eventuais situações semelhantes ajustando um tempo limite de espera pelo trabalhador.

5.1.1 Rede Virtual

Neste cenário, antecipa-se maior demanda da máquina hospedeira tendo em consideração que se executa todos os elementos do sistema em *containers* individuais, compartilhando os recursos da máquina física. No geral, observou-se uma utilização acentuada de recursos pelos trabalhadores durante o processo de cálculo das métricas. Apesar de não comprometer as operações dos outros *containers*, notou-se que se inviabilizou a instanciação de um trabalhador adicional.

Outro aspecto relevante concerne a carga de rede, considerando-se que todos os clientes executam na mesma máquina física. Entretanto, notou-se que, para os cenários avaliados, este não foi o maior gargalo, pois o *broker* é quem coordena/regula a comunicação entre clientes. Todavia, a carga da rede não comprometeu a operação regular dos trabalhadores. Destaca-se os seguintes comportamentos para os seis cenários monitorados:

- A comunicação um para um apresentou a maior carga de trabalho para o servidor durante a execução.
- Em *fan-in*, registrou-se a maior vazão média e máxima, tanto para publicadores quanto para assinantes.
- No cenário de *fan-out*, observou-se a maior latência e, por conseguinte, o maior tempo de cálculo das estatísticas.
- Já os testes com assinatura compartilhado foram os que menos estressaram tanto o *broker* quanto o trabalhador.

5.1.2 Rede Local

Como a intenção não é uma avaliação de desempenho limite do MQTT, procurou-se observar, novamente, o comportamento dos elementos do arcabouço como um todo. Em linhas gerais, faz-se as mesmas observações realizadas para o cenário anterior. Notou-se, todavia, que há uma latência maior na comunicação entre as entidades porque agora elas estão distribuídas em máquinas distintas. Apesar de hospedarmos os dois *brokers* (de controle e de comunicação entre clientes) na mesma máquina, não se observou nenhum impedimento em termos de carga acentuada. No entanto, caso o objetivo seja encontrar o ponto de saturação do cenário em avaliação, recomenda-se hospedar o *broker* de controle em uma máquina à parte.

A distribuição da carga de computação e comunicação entre máquinas distintas também refletiu em uma evoluçao em termos de vazão, tanto para publicadores como assinantes. Relacionado a isso também está uma diminuição em retransmissões, as quais foram mais acentuadas no cenário anterior porque toda a carga, de computação e comunicação, estava sobre a mesma máquina física. Por se tratar de um cenário com alguns trabalhadores em máquinas distintas, notou-se que há um impacto na precisão do cálculo das estatísticas de desempenho; no entanto, passíveis de serem aplicados ajustes adequados.

6 Conclusão

Com o crescimento das pesquisas envolvendo a escalabilidade do protocolo MQTT, notou-se a premente necessidade de ferramentas de avaliação de desempenho em larga escala. As poucas ferramentas disponibilizadas pela comunidade estão descontinuadas ou não oferecem facilidades para a execução de experimentos de alta complexidade. Nesse contexto, desenvolveu-se um arcabouço que possibilita a orquestração de múltiplas instâncias de uma ferramenta de referência para geração de clientes e coleta de estatísticas de desempenho. O sistema desenvolvido foi denominado de *MQTT Poller* e a ferramenta base escolhida foi o *MQTTLoader* (*MQTTLoader*, 2022). A arquitetura do sistema compreende um orquestrador e trabalhadores (*workers*), estes responsáveis pela efetiva execução de instâncias do *MQTTLoader*, coleta dos resultados e envio destes ao orquestrador.

Optou-se por uma abordagem endógena (*i.e.*, baseada no MQTT) na comunicação entre os elementos da arquitetura do sistema; ou seja, utiliza-se um *broker* dedicado à comunicação e controle do sistema. Para tanto, definiu-se todo o mecanismo de comunicação entre os elementos através de tópicos do MQTT. Para facilitar a utilização do sistema, desenvolveu-se uma interface *web* via um *frontend* em *TypeScript* utilizando o *framework Next.js*, possibilitando agilidade na criação e visualização de experimentos. O orquestrador recebe a demanda do usuário e repassa para um ou mais trabalhadores executarem as instâncias necessárias do *MQTTLoader*. Após a obtenção dos resultados destes, pode-se optar por visualizá-los na interface *web*.

Para avaliar o comportamento e as funcionalidades do sistema, apresentou-se estudos de caso. Os resultados indicam que o sistema atende todos os requisitos funcionais estabelecidos. No entanto, tem-se como principal dependência a ferramenta de referência (*i.e.*, *MQTTLoader*) empregada para instanciação dos clientes (*i.e.*, publicadores e assinantes), a qual demonstrou ter limitações e possíveis falhas que podem impactar a execução de experimentos de maior escala.

6.1 Trabalhos Futuros

Considerando as limitações do *MQTTLoader*, uma relevante contribuição seria estender o arcabouço a outras ferramentas de referência. Nesse sentido, também poderia ser pesquisada uma interface padrão de especificação de experimentos e, a partir desta, produzir a conversão para o formato de configuração de cada uma das ferramentas básicas suportadas pelo sistema.

O retorno dos experimentos está, atualmente, limitado ao tamanho máximo de uma mensagem no MQTT (*i.e.*, 256 MB). Pode aparentar suficiente, mas pode ser um fator limitante para experimentos mais complexos. Para tratar retornos de maior volume, necessita-se adotar outro mecanismo de resposta dos trabalhadores.

Outro aspecto que necessita mais atenção, trata-se da ausência de rotinas mais específicas para recuperação de erros. Na maioria dos casos, o sistema identifica eventuais erros e notifica o usuário sem, no entanto, comprometer o funcionamento do sistema. De qualquer forma, pode-se

ter como resultado indesejado a perda parcial ou total de um determinado experimento.

Referências

Apache Software Foundation (2024). Apache jmeter. Disponível em https://jmeter.apache.org/.

A Universally Unique IDentifier (UUID) URN Namespace (2005). Disponível em https://datatracker.ietf.org/doc/html/rfc4122.

Banno, R. and Yoshizawa, T. (2022). A scalable iot data collection method by shared-subscription with distributed mqtt brokers, *Mobile Networks and Management: 11th EAI International Conference, MONAMI 2021, Virtual Event, October 27-29, 2021, Proceedings*, Springer, pp. 218–226. https://doi.org/10.1007/978-3-030-94763-7_17.

btree (2014). Disponível em https://github.com/tidwall/btree.

Canek, R., Borges, P. and Taconet, C. (2022). Analysis of the impact of interaction patterns and iot protocols on energy consumption of iot consumer applications, *Distributed Applications and Interoperable Systems: 22nd IFIP WG 6.1 International Conference, DAIS 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, Springer, pp. 131–147. https://doi.org/10.1007/978-3-031-16092-9_9.

Detti, A., Funari, L. and Blefari-Melazzi, N. (2020). Sublinear scalability of mqtt clusters in topic-based publish-subscribe applications, *IEEE Transactions on Network and Service Management* **17**(3): 1954–1968. https://doi.org/10.1109/TNSM.2020.3003535.

Dinculeană, D. and Cheng, X. (2019). Vulnerabilities and limitations of mqtt protocol used between iot devices, *Applied Sciences* **9**(5). Disponível em https://www.mdpi.com/2076-3417/9/5/848.

Eclipse Paho (2019). Disponível em https://projects.eclipse.org/projects/iot.paho.

High performance, extensible, minimalist Go web framework Echo (2019). Disponível em https://echo.labstack.com.

MQTT (2019). Disponível em http://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

MQTT broker latency measure tool (2018). Disponível em https://github.com/hui6075/mqtt-bm-latency.

mqtt-jmeter (2014). Disponível em https://github.com/tuanhiep/mqttjmeter.

MQTTLoader (2022). Disponível em https://github.com/dist-sys/mqttloader.

mqtt-malaria (2013). Disponível em https://github.com/etactica/mqtt-malaria.

Next.js by Vercel - The React Framework (2022). Disponível em https://nextjs.org.

Ribas, N. K. and Spohn, M. A. (2022). A new approach to a self-organizing federation of mqtt brokers, *Journal of Computer Science* **18**(7): 687–694. https://doi.org/10.3844/jcssp.2022.687.694.

Spohn, M. A. (2022). On mqtt scalability in the internet of things: Issues, solutions, and future directions, *Journal of Electronics and Electrical Engineering* pp. 4–4. https://doi.org/10.37256/jeee.1120221687.

The Go Authors (2024). The go programming language, Official Go website. Disponível em https://golang.org.

uuid: Go package for UUIDs based on RFC 4122 and DCE 1.1: Authentication and Security Services. (2011). Disponível em https://github.com/google/uuid.