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

BRUNO BEVILAQUA

SUPORTE À FEDERAÇÃO DE BROKERS MQTT VIA MICROSSERVIÇOS

GERENCIAMENTO DINÂMICO DA TOPOLOGIA DA FEDERAÇÃO

CHAPECÓ
2023

BRUNO BEVILAQUA

SUPORTE À FEDERAÇÃO DE BROKERS MQTT VIA MICROSSERVIÇOS

GERENCIAMENTO DINÂMICO DA TOPOLOGIA DA FEDERAÇÃO

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 Aurélio Spohn

CHAPECÓ
2023

Bibliotecas da Universidade Federal da Fronteira Sul - UFFS

Bevilaqua, Bruno
Suporte à federação de brokers MQTT via microsserviços: Gerenciamento dinâmico da topologia da federação / Bruno Bevilaqua. -- 2023.
36 f.:il.

Orientador: Prof. Dr. Marco Aurélio Spohn

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal da Fronteira Sul, Curso de Bacharelado em Ciência da Computação, Chapecó,SC, 2023.

1. Microsserviços. 2. MQTT. 3. Broker. I. Spohn, Marco Aurélio, orient. II. Universidade Federal da Fronteira Sul. III. Título.

Elaborada pelo sistema de Geração Automática de Ficha de Identificação da Obra pela UFFS com os dados fornecidos pelo(a) autor(a).

BRUNO BEVILAQUA

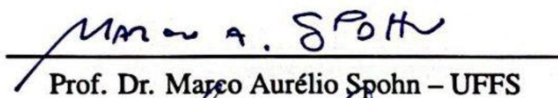
SUPORTE À FEDERAÇÃO DE BROKERS MQTT VIA MICROSSERVIÇOS
GERENCIAMENTO DINÂMICO DA TOPOLOGIA DA FEDERAÇÃO

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

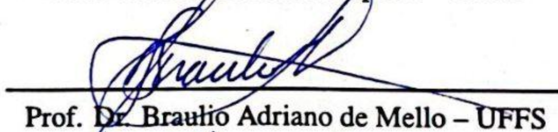
Orientador: Prof. Dr. Marco Aurélio Spohn

Aprovado em: 7/2/2023

BANCA AVALIADORA



Prof. Dr. Marco Aurélio Spohn – UFFS



Prof. Dr. Braulio Adriano de Mello – UFFS

Prof. Dr. Luciano Lores Caimi – UFFS

RESUMO

Conectar e comunicar dispositivos do cotidiano a internet tornando-os inteligentes é a prática que recebe o nome de internet das coisas. Conforme essa área cresce, junto a ela cresce a necessidade de soluções confiáveis para a comunicação entre estes dispositivos. Utilizado para este fim, o *Message Queuing Telemetry Transport (MQTT)*, fornece um protocolo de comunicação através da troca de mensagens fazendo o uso de um servidor central que recebe o nome de *broker*. A utilização de apenas um servidor como propõe sua implementação mais simples, não oferece tanta confiabilidade por possuir um ponto único sem tolerância a falhas, demandando então, que soluções escaláveis e distribuídas como a federação de *brokers* MQTT sejam implementadas. Neste trabalho será proposto uma abordagem para gerenciar dinamicamente uma federação de *brokers*, adicionando funções ainda não implementadas como a de inserção de novos *brokers* na federação em tempo de execução, além de tratamento em caso de falha em um dos servidores. Para garantir uma maior flexibilidade e pensando na adição de novas funcionalidades sem causar um impacto tão grande na solução inicial, o serviço proposto será desenvolvido utilizando a arquitetura de microsserviços.

Palavras-chave: Microsserviços. MQTT. Broker.

ABSTRACT

Connecting and communicating everyday devices to the internet making them smart is the practice that is called the internet of things. As this area grows, along with it the need for reliable solutions for communication between these devices grows. Used for this purpose, the *Message Queuing Telemetry Transport (MQTT)* is able to provide communication to these devices through the exchange of messages using a central server called *broker*. The use of only one server, as its simplest implementation proposes, does not offer as much reliability as it has a single point without fault tolerance, demanding that scalable and distributed solutions such as the federation of *brokers* MQTT be implemented. In this work, an approach to dynamically manage a federation of *brokers* will be proposed, adding functions not yet implemented such as the insertion of new *brokers* in the federation at runtime, in addition to handling in case of failure in one of the servers. To ensure greater flexibility and considering the addition of new features without causing such an impact on the initial solution, the proposed service will be developed using the microservices architecture.

Keywords: Microservices. MQTT. Broker.

LISTA DE ILUSTRAÇÕES

Figura 1 – Ecossistema MQTT 12
Figura 2 – Cluster MQTT 14
Figura 3 – Processo de criação de uma sub-malha. Fonte Spohn (13) 15
Figura 4 – Processo de roteamento de mensagens. Fonte Spohn (13) 15
Figura 5 – Arquitetura monolítica em comparação a arquitetura de microsserviços . . . 17
Figura 6 – Federador MQTT. Fonte Spohn (13) 20
Figura 7 – Resultados do cenário utilizando federação de *brokers*. Fonte Spohn (13) . 21
Figura 8 – Resultados do cenário utilizando *broker* único. Fonte Spohn (13) 22
Figura 9 – Arquitetura de um Federador MQTT. Fonte Ribas; Spohn (10) 22
Figura 10 – Resultados do cenário utilizando federação de *brokers*. Fonte Ribas; Spohn (10) 23
Figura 11 – Resultados do cenário utilizando *broker* único. Fonte Ribas; Spohn (10) . . 24
Figura 12 – Inserção de novos nós a topologia. 28
Figura 13 – Reconexão de um nó através do mecanismo *health check*. 28
Figura 14 – Topologia utilizada no cenário de avaliação de desempenho. 31
Figura 15 – Topologia antes da realocação do nó 8. 32
Figura 16 – Topologia após a realocação do nó 8. 32

LISTA DE TABELAS

Tabela 1 – Latência de publicações da federação. 33

SUMÁRIO

1	**INTRODUÇÃO**	**9**
2	**OBJETIVOS**	**11**
2.1	Objetivo Geral	11
2.2	Objetivos Específicos	11
2.3	Justificativa	11
3	**REVISÃO BIBLIOGRÁFICA**	**12**
3.1	*Protocolo MQTT*	12
3.2	*Escalabilidade*	13
3.2.1	**Vertical**	**13**
3.2.2	**Horizontal**	**13**
3.2.3	**Federação de brokers**	**14**
3.3	Arquitetura de sistemas	16
3.3.1	**Arquitetura monolítica**	**16**
3.3.2	**Arquitetura de microsserviços**	**16**
3.3.3	**Comunicação**	**18**
3.3.4	**Implantação**	**18**
4	**TRABALHOS RELACIONADOS**	**20**
4.1	Federador MQTT por Spohn	20
4.2	Federador MQTT por Ribas e Spohn	22
5	**DESENVOLVIMENTO**	**25**
5.1	Ferramentas	25
5.1.1	**Docker**	**25**
5.1.2	**MongoDB**	**25**
5.1.3	**Eclipse Mosquitto**	**25**
5.1.4	**Eclipse Paho**	**25**
5.1.5	**MQTT *broker latency measure tool***	**25**
5.2	Arquitetura do conjunto	26
5.2.1	**Microsserviço**	**26**
5.2.2	**Federador**	**29**
5.2.3	**Configuração**	**29**
5.3	Cenário de avaliação	30
6	**RESULTADOS**	**32**
6.1	Cenário de teste de disponibilidade	32
6.2	Cenário de teste de desempenho	33
7	**CONCLUSÃO**	**34**
7.1	Trabalhos futuros	34
	REFERÊNCIAS	**35**

1 INTRODUÇÃO

O termo Internet das Coisas (IoT) geralmente se refere a cenários em que a rede, a conectividade e a capacidade de computação se estendem a objetos, sensores e itens de uso diário normalmente não considerados computadores, permitindo que esses dispositivos gerem, troquem e consumam dados com o mínimo de intervenção humana. Essa tecnologia está incorporada em um amplo espectro de produtos, sistemas e sensores em rede, que aproveitam os avanços em poder de computação, miniaturização de eletrônicos e interconexões de rede para oferecer novos recursos que antes não eram possíveis (11).

Os campos de aplicação destas tecnologias são diversos e numerosos, e a cada vez mais se expandem para todas as áreas do dia a dia. As áreas de aplicação mais proeminentes incluem, por exemplo, casas ou edifícios, que incluem sistemas de monitoramento de gastos de eletricidade, gás ou água, e até mesmo sistemas de segurança para o ambiente em geral. A área da saúde possui alta aplicação no monitoramento de doenças crônicas de pacientes através do uso de sensores ou dispositivos vestíveis. Também é possível encontrar projetos de cidades inteligentes baseados em IoT, voltados a controle de tráfego de veículos, iluminação, controle de vagas de estacionamento, entre outros (15).

O desenvolvimento de aplicações voltadas ao IoT traz a necessidade de utilizar protocolos de comunicação eficientes, visto que muitas vezes a capacidade dos dispositivos conectados é baixa tanto em poder de processamento, quanto em disponibilidade de rede. Portanto, abordagens assíncronas como o protocolo *Message Queuing Telemetry Transport (MQTT)*, que faz a utilização do mecanismo *Publish/Subscribe (P/S)*, onde, dispositivos publicadores fazem o envio de dados a um servidor central conhecido como *broker*, que por sua vez faz a distribuição das mensagens recebidas para dispositivos interessados em receber essas mensagens (consumidores), acabam tornando-se peças chaves na construção dessas aplicações.

A medida com que essas aplicações crescem, surge a necessidade de escalar a infraestrutura, como na sua implementação mais simples, o MQTT faz o uso de apenas um *broker*, garantir o alto desempenho e a alta disponibilidade acabam se tornando obstáculos já que há um ponto único para falhas. Para atingir a alta escala e disponibilidade, o uso de técnicas como a clusterização, onde utiliza-se um balanceador de carga para direcionar requisições a servidores que encontram-se com maior disponibilidade acabam sendo comuns. Surgindo como uma solução paralela a clusterização, a federação de *brokers* inicialmente proposta por Spohn (14), tem como objetivo ser um modelo auto organizável, onde assinantes locais em *brokers* distintos se inter conectam utilizando malhas para que seja possível fazer o roteamento de publicações oriundas de produtores.

Este protocolo recentemente recebeu duas novas variantes, a primeira também desenvolvida por Spohn (13) e a segunda por Ribas; Spohn (10), estas duas novas versões trazem novos conceitos para contornar necessidades impostas pela primeira versão, como por exemplo, a introdução ao federador, uma aplicação que atua juntamente ao *broker* e que fornece todos

os mecanismos necessários para o funcionamento da federação sem necessitar de alterações no funcionamento original do *broker*.

2 OBJETIVOS

2.1 OBJETIVO GERAL

Gerenciar a realocação de nós desconectados e a inserção de nós ingressantes na topologia virtual de uma federação de *brokers* do protocolo MQTT, utilizando a arquitetura de microsserviços, a fim de garantir uma maior disponibilidade.

2.2 OBJETIVOS ESPECÍFICOS

- Realizar a implementação de um microsserviço para gerenciamento da topologia;
- Realizar a implementação de um federador de *brokers* MQTT;
- Analisar o funcionamento do conjunto desenvolvido com base em métricas coletadas;

2.3 JUSTIFICATIVA

Dia após dia a quantidade de dispositivos conectados a rede IoT aumenta, e junto a isso o interesse pela área também. Estimado pela Huawei, a quantidade de conexões IoT até o ano de 2025 pode chegar a 100 bilhões, e o impacto financeiro na economia global pode atingir de 3,9 a 11,1 trilhões de dólares segundo McKinsey Global Institute (11). Um salto tão grande na demanda acaba por gerar a necessidade de soluções altamente escaláveis para o protocolo MQTT, uma solução relativamente nova e muito promissora é a federação de *brokers*, mas que em contrapartida, as implementações existentes apesar de citarem, não demonstram formas para se obter uma escala da topologia virtual da federação de forma dinâmica em tempo de execução, além de não abordarem tratamentos para casos onde *brokers* ficam indisponíveis. Devido a isso, surge um espaço para o desenvolvimento e análise de uma nova arquitetura capaz de forncer ainda mais disponibilidade e funcionalidades a federação de *brokers*.

3 REVISÃO BIBLIOGRÁFICA

3.1 *PROTOCOLO MQTT*

Projetado para ser um protocolo de transporte de mensagens extremamente leve e de fácil implementação, o *Message Queuing Telemetry Transport (MQTT)* é ideal para utilização em aplicações voltadas ao IoT (9).

Utiliza o mecanismo *Publish/Subscribe (P/S)* no qual, um componente chamado *publisher*, envia mensagens para uma fila chamada de tópico, e outro componente chamado *consumer*, por sua vez, indica seu interesse em receber mensagens deste tópico quando efetua uma operação de inscrição no mesmo (12).

Como demonstra a Figura 1 tanto publicadores como consumidores não têm conhecimento uns dos outros, devido a isso, ambos utilizam um intermediário chamado *broker*, que atua como uma ponte conectando ambos, sua função é filtrar as mensagens de entrada, organizá-las nos tópicos e fazer a distribuição para os interessados (12).

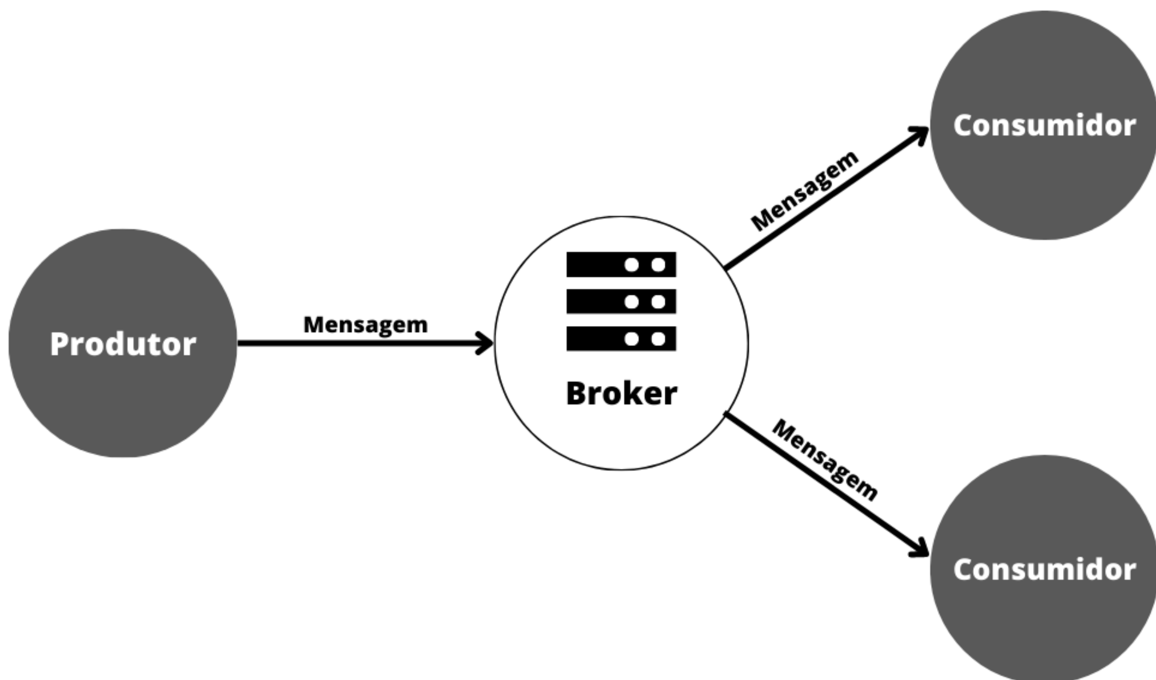


Figura 1 – Ecossistema MQTT

O protocolo MQTT disponibiliza três níveis para qualidade de serviço (QoS), que funcionam como um acordo entre as duas partes (produtores e consumidores) no que diz respeito a garantia de entrega de uma mensagem, os níveis são:

- QoS 0: Neste nível uma mensagem é enviada no máximo uma vez e não oferece garantia de entrega, também é conhecido como "No máximo uma vez".

- QoS 1: Já no nível 1, os dados são enviados pelo menos uma vez e existe a possibilidade de entregas duplicadas, é conhecido também como "Pelo menos uma vez"
- QoS 2: Conhecido como exatamente uma vez, utiliza um *handshake* de quatro vias para fazer o envio "Exatamente uma vez".

3.2 *ESCALABILIDADE*

Em cenários em que o tráfego de dados entre publicadores e assinantes é muito grande e constante, uma simples implantação, contendo um *broker*, acaba gerando um cenário suscetível a falha e possíveis gargalos, demandando então de soluções que envolvem modelos de escala vertical e horizontal, para que seja possível escalar a infraestrutura conforme a demanda aumente.

3.2.1 Vertical

Consiste basicamente em aumentar a capacidade de processamento de um servidor, a fim de melhorar ou explorar todo o recurso disponibilizado pelo mesmo.

No caso de um servidor MQTT além do aumento dos recursos da máquina, múltiplos *brokers* podem ser inicializados em uma mesma máquina para aumentar o fluxo de transmissão de mensagens.

Utilizar a escala vertical pode resolver alguns problemas de desempenho, porém, este modelo de escala não oferece alta disponibilidade, uma vez que todo o sistema está concentrado em apenas um lugar, demandando então, de outras soluções como a clusterização.

3.2.2 Horizontal

Esta estratégia tem como principal fundamento, dividir o processamento dos dados em máquinas distintas, e tem como principal vantagem um aumento na disponibilidade, uma vez que deixa de possuir um ponto único de falha.

Para escalar o MQTT horizontalmente, pode ser utilizado o conceito conhecido como clusterização, que como demonstrado na Figura 2, utiliza a frente dos *brokers* um balanceador de carga, que possui a responsabilidade de determinar qual servidor será o responsável por processar as mensagens provindas dos clientes, essa determinação pode ser feita com base em critérios, como por exemplo, encaminhar ao servidor que possui mais disponibilidade no momento.

Para o correto funcionamento de um cluster, é necessário que os *brokers* possuam uma conexão entre si, para que possa ser feita a sincronização dos dados.

Apesar de se mostrar mais efetiva que a escala vertical, a clusterização pode apresentar dois principais pontos de falha, a dependência do balanceador de carga, e um possível engasgo na comunicação entre os *brokers*.

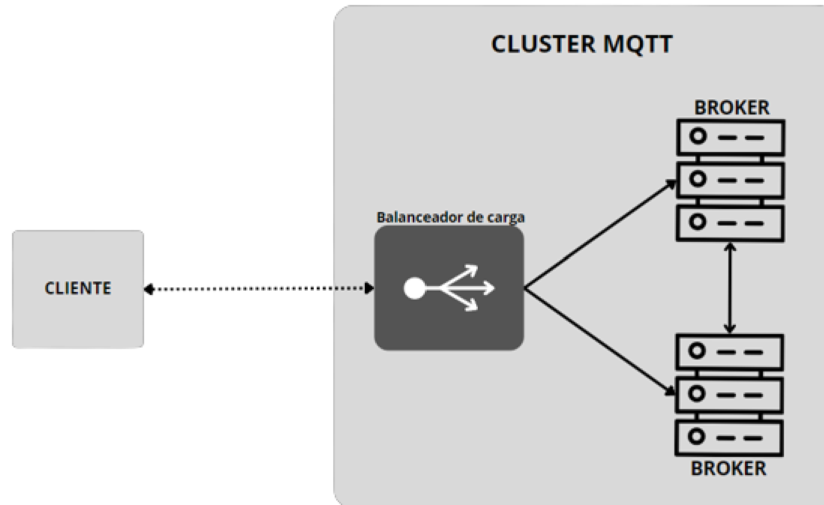


Figura 2 – Cluster MQTT

3.2.3 Federação de brokers

Nesta solução proposta por Spohn (13), *brokers* distribuídos em uma topologia virtual de malha, se auto organizam criando sub-malhas que conectam consumidores que possuem interesse em um tópico em comum.

O processo de criação das sub-malhas inicia a partir de uma inscrição local em um dos *brokers* da federação, este *broker* que recebe a inscrição envia para seus vizinhos um anúncio de núcleo dando origem a uma nova malha, o vizinho ao receber o anúncio armazena a informação de quem é o núcleo e a distância em saltos até o mesmo, e retransmite a informação para seus vizinhos incrementando a distância em 1 salto, o processo é repetido até que todos os *brokers* recebam a informação. Neste ponto podem existir casos em que dois *brokers* efetuam um anúncio de núcleo simultaneamente, ou em tempos semelhantes, e quando isso acontece todos os *brokers* da federação devem convergir para que somente um seja o núcleo, para isso pode ser utilizado como critério para decisão, o menor ou o maior identificador que está associado ao nó.

Quando um dos *brokers* da federação recebe um inscrito local para um tópico ao qual já exista uma sub-malha e que ele ainda não faz parte desta malha, o mesmo envia um anúncio para ingressar a esta malha, este anúncio é enviado em direção ao núcleo pelo caminho mais curto, a quantidade de vizinhos aos quais essa informação é enviada é definido pela redundância da federação, ou seja, se a redundância for 2, o anúncio será enviado a 2 vizinhos até chegar ao núcleo. Todos os *brokers* que fizerem parte do caminho até o núcleo, também passarão a fazer parte da malha.

Já para o roteamento de mensagens enviadas pelos publicadores, existem dois possíveis casos, no primeiro deles, o produtor está fazendo publicações em um tópico a partir de um dos *brokers* que faz parte da malha correspondente ao tópico, neste caso as publicações são simplesmente encaminhadas através da malha pois todos os *brokers* pertencentes tem conhecimento uns dos outros. No segundo caso o publicador envia mensagens para um *broker* que não faz parte

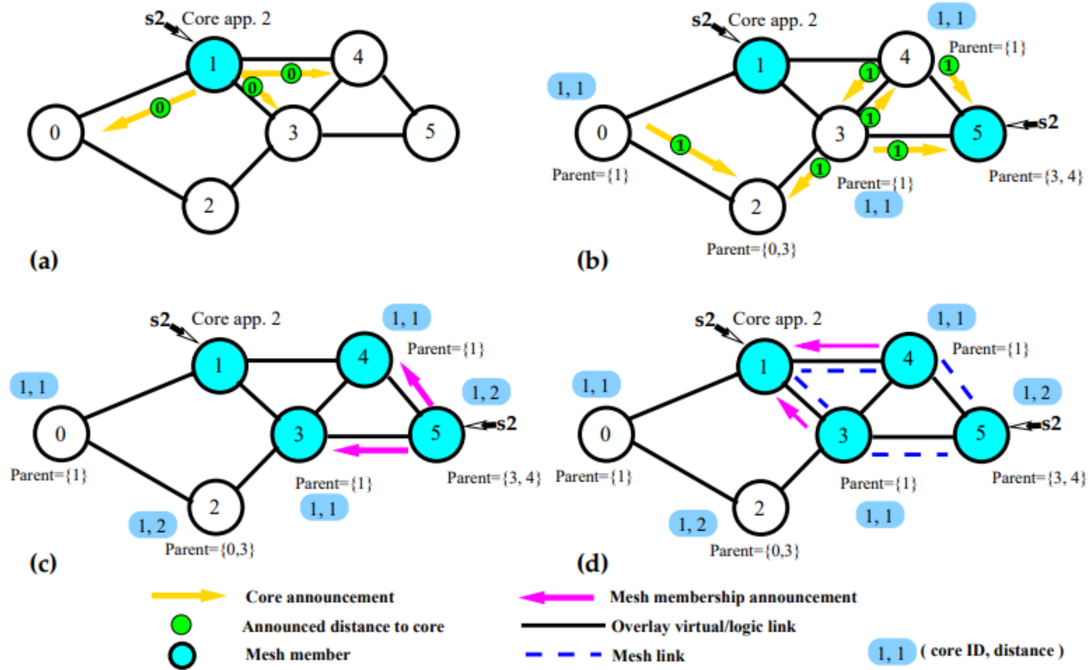


Figura 3 – Processo de criação de uma sub-malha. Fonte Spohn (13)

da malha, para chegar até os consumidores interessados, o protocolo é encaminhar a mensagem para seus vizinhos, e repetir o processo até atingir um dos *brokers* pertencentes a malha, a partir daí, a mensagem é transmitida pela malha como no primeiro caso.

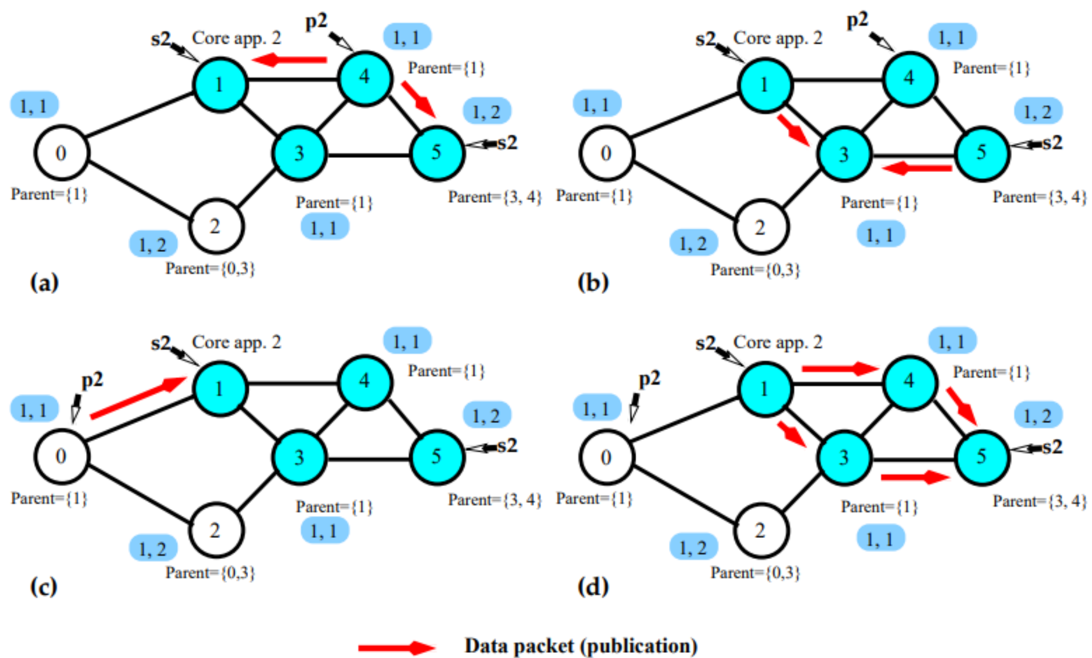


Figura 4 – Processo de roteamento de mensagens. Fonte Spohn (13)

As vantagens de adotar a federação segundo o autor são:

- Não há um ponto único de falha: os clientes podem optar por associar-se a quaisquer

brokers federados.

- Balanceamento de carga: sempre há a possibilidade de escolher entre um conjunto de *brokers* disponíveis e obter o que se precisa.
- Exploração de topologias virtualizadas ou capacidades de redes: a implantação virtualizada completa é realizável por meio de agentes instanciados em máquinas ou contêineres virtuais.

3.3 ARQUITETURA DE SISTEMAS

A arquitetura é o que permite que os sistemas evoluam e forneçam um determinado nível de serviço ao longo de seu ciclo de vida. Na engenharia de software, a arquitetura se preocupa em fornecer uma ponte entre a funcionalidade do sistema e os requisitos de atributos de qualidade que o sistema deve atender (2).

3.3.1 Arquitetura monolítica

O modelo de arquitetura chamado de monolítico pode ser caracterizado como uma aplicação na qual os módulos não podem ser executados independentemente. Apesar de ser mais comum, podem apresentar problemas conforme a aplicação cresce, monólitos de grande porte são difíceis de manter e evoluir devido à sua complexidade, rastrear bugs requer longas leituras através de sua base de código, os monólitos também sofrem com o problema de dependências externas, onde casos em que é necessário adicionar ou atualizar bibliotecas resultam em sistemas inconsistentes (4).

Para projetos de grande porte, a reinicialização geralmente acarreta tempos de inatividade consideráveis, dificultando o desenvolvimento, teste e manutenção do projeto. Além disso, apresentam um aprisionamento tecnológico para desenvolvedores, que são obrigados a usar a mesma linguagem e estruturas definidas no início do desenvolvimento (4).

Conforme os monólitos crescem a demanda por recursos de máquina tende a crescer junto, necessitando então a aplicação de soluções escaláveis que muitas vezes tornam-se inviáveis devido a alta complexidade do software.

3.3.2 Arquitetura de microsserviços

O termo “microsserviços” foi introduzido pela primeira vez em 2011 em um workshop de arquitetura de software como forma de descrever as ideias comuns dos participantes em padrões de arquitetura, mais recentemente, as principais empresas de consultoria de software e empresas de design de produtos descobriram que a abordagem de microsserviços é uma arquitetura atraente que permite que equipes e organizações de software sejam mais produtivas

em geral e criem produtos de software frequentemente mais bem-sucedidos. Empresas como Amazon, Netflix, eBay e LinkedIn optaram pelo uso desta arquitetura para implantar seus grandes serviços através de pequenos componentes (2).

Como demonstra a Figura 5 em comparação com a arquitetura monolítica, os micros serviços devem ser componentes independentes implantados conceitualmente de forma isolada e equipados com ferramentas de persistência de memória dedicada (por exemplo, bancos de dados), tornando o resultado final, uma aplicação distribuída. Como todos os componentes de uma arquitetura de micros serviços são micros serviços, o seu diferencial deriva da composição e coordenação de seus componentes por meio de mensagens Dragoni et al. (4).

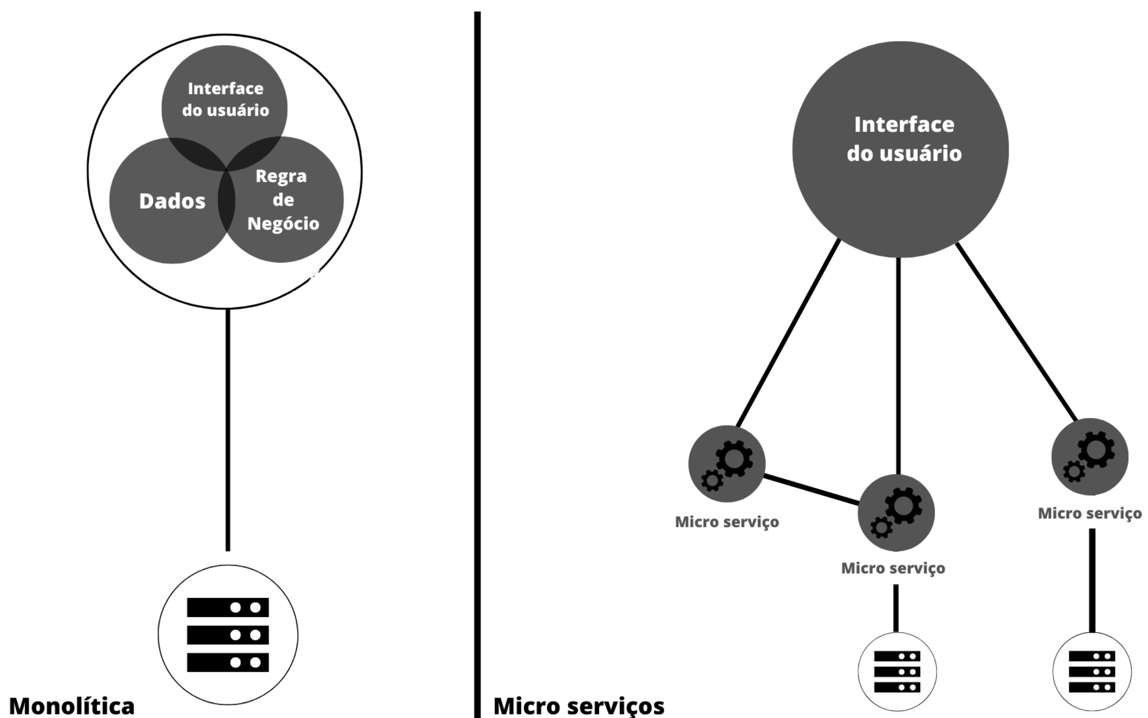


Figura 5 – Arquitetura monolítica em comparação a arquitetura de microsserviços

Resultando de uma junção da computação orientada a serviço (SOC) e das arquiteturas orientadas a serviço (SOA), a arquitetura de microsserviços foi desenvolvida abstraindo níveis de complexidade que para suas antecessoras eram necessários, para que os desenvolvedores possam focar somente na programação de sistemas simples e efetivos.

Apesar de nem todos os conceitos acerca da arquitetura de microsserviços estarem definidos, os autores Alshuqayran; Ali; Evans (2) e Dragoni et al. (4) citam como benefícios da utilização desta arquitetura um grande aumento na produtividade por parte das equipes já que os desenvolvedores não precisam atuar em todas partes do sistema, sendo possível focar apenas em módulos específicos, a aplicação de testes automatizados, além do monitoramento e rastreamento de falhas também podem ter seus processos agilizados. O fato de os ambientes serem isolados e independentes viabiliza integrações e entregas contínuas tornando o sistema como um todo mais confiável, além de torná-lo altamente escalável, de fácil instalação e manutenção. Ao utilizar a

comunicação via mensagens, os desenvolvedores não precisam prender-se a utilização de apenas uma tecnologia, podendo utilizar as que mais se adequarem em cada microsserviço.

A adoção desta arquitetura por outro lado pode não ser uma tarefa simples, decompor uma aplicação monolítica identificando partes que podem ser modularizadas especialmente em aplicações que possuem partes legadas pode ser um grande desafio, outro desafio que pode ser encontrado na etapa de implementação é a definição de padrões de comunicação a serem adotados.

3.3.3 Comunicação

Para comunicar partes de um sistema monolítico de forma rigorosa basta utilizar os relacionamentos do banco de dados, já na arquitetura de microsserviços, onde todos os módulos possuem uma base de dados própria, surge uma necessidade de comunicá-los utilizando padrões apropriados de comunicação que podem ser classificados como síncronos e assíncronos (1).

Utilizando modelos como o REST baseado em HTTP, na abordagem síncrona, quando feita uma solicitação o cliente deve aguardar uma resposta para dar continuidade em seu fluxo, este modelo requer que o serviço requisitado esteja em pleno funcionamento, caso contrário, se o cliente não receber uma resposta o mesmo deve ser notificado (1)

Na comunicação assíncrona, normalmente utiliza-se filas de mensagem, neste modelo baseado na arquitetura orientada a eventos, o cliente faz o envio de uma mensagem que pode ser uma solicitação, e não bloqueia seu processamento até receber a resposta, pois na maioria das vezes o cliente não necessita desta resposta.

Há ainda casos em que ambos os métodos podem ser adotados resultando em um modelo classificado como híbrido.

3.3.4 Implantação

Embora seja fácil implantar um aplicativo na abordagem monolítica, implantar sistemas baseados na arquitetura de microsserviços pode se tornar um desafio, especialmente quando milhares de módulos compõem um sistema, sem contar casos onde o mesmo serviço é escalado inúmeras vezes devido a alta demanda.

Surgindo como uma aliada a esta arquitetura, a computação em nuvem fornece a possibilidade de dimensionar aplicativos para servidores virtuais de forma eficaz já que é possível ajustar dinamicamente seus recursos de computação (1). Plataformas como a Amazon Web Services, Microsoft Azure e Google Cloud, além de garantirem segurança e alta disponibilidade, fornecem seus recursos cobrando sob demanda, permitindo que a infraestrutura a ser utilizada escale de modo dinâmico juntamente com o crescimento da aplicação, evitando assim custos iniciais com infraestruturas e datacenters fixos.

Segundo Aksakalli et al. (1) inúmeros padrões de implantação podem ser identificados, dentre eles podem ser citados, instância de serviço por máquina virtual (VM), onde cada serviço é empacotado como uma imagem de VM, permitindo criar ambientes totalmente isolados evitando que recursos de computação reservados sejam atingidos. Instância de serviço por contêiner, contêineres são mecanismos de virtualização executados no nível do sistema operacional, e podem ser limitados a consumirem somente recursos desejados, cada servidor, seja ele físico ou virtual pode rodar inúmeros contêineres, este modelo também permite o uso de orquestradores como o Kubernetes ou o Docker Swarm que automatizam a implantação, o dimensionamento e a gestão de aplicações em contêineres.

4 TRABALHOS RELACIONADOS

Neste capítulo serão apresentados dois trabalhos onde os autores propuseram modelos para viabilização de uma federação de *brokers*, ambos os trabalhos possuem relação direta com o presente trabalho. Para cada um deles será apresentado os conceitos utilizados para a construção, bem como os resultados obtidos.

4.1 FEDERADOR MQTT POR SPOHN

A versão original de uma federação de *brokers* requer que alterações na implementação dos *brokers* sejam feitas, tendo como objetivo contornar essa necessidade, Spohn (13) propôs um novo modelo a federação introduzindo então o conceito de federador.

Neste modelo, uma aplicação baseada no subsistema Pub/Sub é construída e associada a um *broker* para que faça a construção e a manutenção das malhas. Segundo o autor, fazer as alterações diretamente no *broker* poderia resultar em um desempenho melhor, porém, com a possibilidade de virtualização de recursos de máquina e de comunicação, a implantação de *brokers* em contêiner dedicados associados a cada federador seria uma estratégia que poderia ser aproveitada para obter melhores resultados de desempenho.

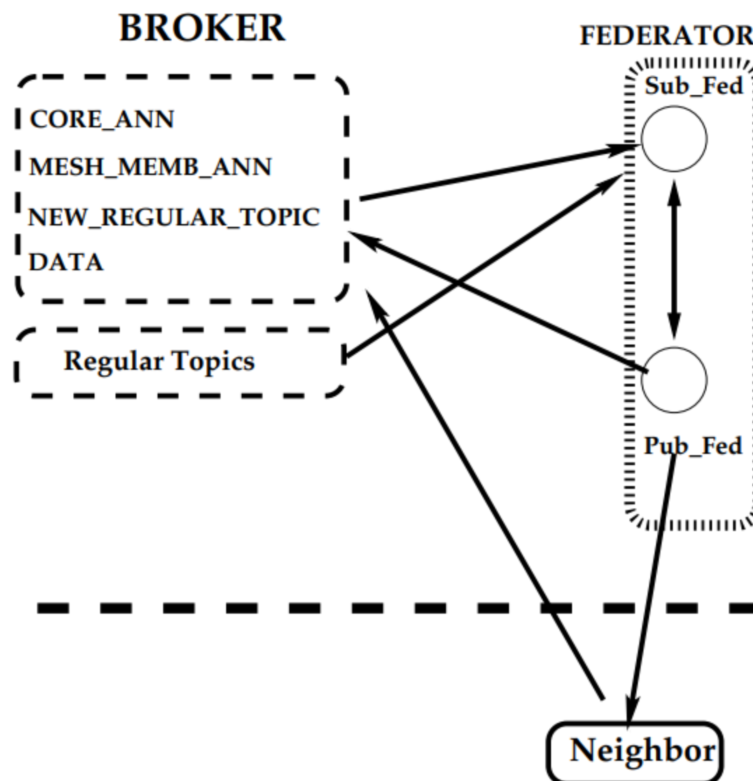


Figura 6 – Federador MQTT. Fonte Spohn (13)

Como demonstrado na Figura 6, esta aplicação é baseada em duas camadas principais. A camada **Pub Fed** é responsável por publicar mensagens de controle e fazer o roteamento de

mensagens para os *brokers* vizinhos, enquanto a camada **Sub Fed** tem como responsabilidade receber e fazer o controle de publicações recebidas de *brokers* vizinhos.

Para o funcionamento do modelo o Sub Fed necessita estar inscrito em um conjunto de tópicos de controle, são eles:

- CORE_ANN: Para receber anúncios provindos de núcleos vizinhos.
- MESH_MEMB_ANN: Para receber anúncios de ingresso na malha.
- NEW_REGULAR_TOPIC: Para receber assinaturas ou primeira publicação em tópicos regulares
- DATA: Para receber dados publicados por clientes em tópicos regulares, para então serem repassados e atingirem os clientes interessados.

Outro ponto fundamental para funcionamento, é a necessidade de que os clientes dos *brokers* façam uma publicação informando uma inscrição ou primeira publicação em um tópico regular através do tópico NEW_REGULAR_TOPIC, caso contrário, não há a garantia de que as publicações serão entregues.

Para construção do cenário de avaliação, foi construída uma topologia virtual em formato de grade 3x3, onde cada nó era composto por um *broker* Mosquitto inicializado em um container Docker. Os testes foram feitos em dois cenários, o primeiro com dois assinante no mesmo *broker* simulando uma implementação de *broker* único, e o segundo utilizando a solução federada onde dois consumidores foram dispostos em nós diferentes da topologia, para ambos os cenários os assinantes se encontravam a uma distância de 1 e 4 saltos do publicador.

Um publicador recebeu duas configurações, na primeira foram realizadas 500 publicações, e na segunda 1000 publicações, cada uma delas possuía o tamanho de 64 bytes e a taxa de envio de publicações era entre 10 e 20 publicações por segundo.

Como mostra a Figura 7, na solução federada, o consumidor que se encontrava a um salto do publicador (nó 7) apresentou metade do atraso em comparação ao que se encontrava mais distante (nó 2).

Publ.	**Subscriber at Node 2**	**Subscriber at node 7**
500	34.64 ± 3.21 ms	18.65 ± 3.2 ms
1000	34.77 ± 3.17 ms	18.61 ± 3.17 ms

Figura 7 – Resultados do cenário utilizando federação de *brokers*. Fonte Spohn (13)

No cenário de *broker* único, os resultados também mostram um atraso maior no nó mais distante do publicador, como demonstra a Figura 8, o autor também percebeu um atraso maior em comparação ao cenário federado, o qual foi justificado devido a necessidade de lidar com o dobro de carga.

Publ.	**Broker at Node 2** Delay (ms)		**Broker at Node 7** Delay (ms)	
	Sub. A	*Sub. B*	*Sub. A*	*Sub. B*
500	34.59 ± 3.13	42.47 ± 4.37	18.63 ± 3.2	26.69 ± 4.77
1000	34.58 ± 3.23	42.58 ± 4.48	18.47 ± 3.22	26.42 ± 4.56

Figura 8 – Resultados do cenário utilizando *broker* único. Fonte Spohn (13)

4.2 FEDERADOR MQTT POR RIBAS E SPOHN

Tendo como base o federador proposto por Spohn (13), Ribas; Spohn (10) realizaram uma análise de desempenho da federação de *brokers*, e além da análise, propôs também uma reimplementação aplicando mudanças na arquitetura original e dando origem a uma nova versão, o federador foi desenvolvido utilizando a linguagem Rust, a biblioteca de cliente MQTT Paho, além do *runtime* Tokio, que faz o gerenciamento de *tasks*, unidades de processamento assíncrono, que diferente das *threads* do sistema, não são gerenciados pelo escalonador e sim pelo *runtime* Tokio, o que as torna muito mais leves de criar, executar e destruir.

Em sua implementação, Ribas; Spohn (10) introduziram o conceito de trabalhadores de tópicos, cada trabalhador de tópico é responsável por fazer o gerenciamento da malha de um tópico específico, logo, para cada tópico federado, a um trabalhador associado, portanto a criação dos mesmos é feita de forma dinâmica à medida em que tópicos federados são utilizados.

De acordo com a Figura 9 que demonstra a arquitetura do federador, publicações que passam por tópicos federados ou de controle são encaminhadas a um componente nomeado *dispatcher*, que possui a responsabilidade de identificar o trabalhador de tópico responsável e fazer o encaminhamento da mensagem para este trabalhador.

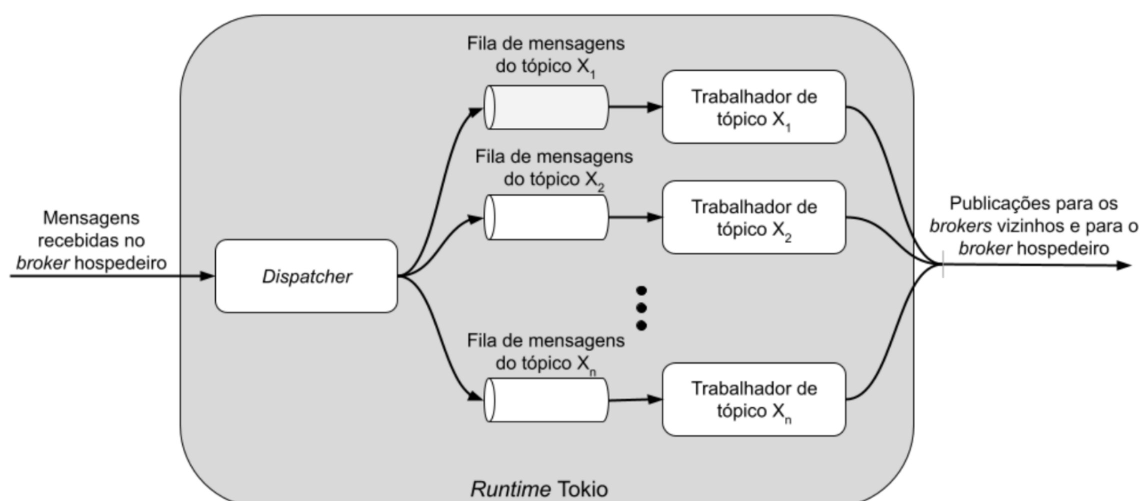


Figura 9 – Arquitetura de um Federador MQTT. Fonte Ribas; Spohn (10)

Para explorar toda a redundância disponibilizada, as publicações oriundas de um *broker*

não participante da malha são encaminhadas para todos os pais disponíveis em cada nó até o núcleo, diferente da original onde as mesmas eram encaminhadas em um único caminho.

O autor descreve como publicações federadas todas aquelas feitas em tópicos que iniciam com o prefixo "federated", desta forma, utilizando o recurso de curinga multinível disponibilizado pelo MQTT, onde consumidores podem receber publicações de múltiplos tópicos a partir de uma única inscrição, o federador realiza uma inscrição utilizando o curinga "federated/".

Em sua implementação também são utilizados 4 tópicos de controle que possibilitam o uso de coringas multiníveis, são eles:

- federator/core_ann/: Para receber anúncios de núcleo.
- federator/memb_ann/: Para receber anúncios de ingresso na malha.
- federator/routing/: Para rotear publicações federadas.
- federator/beacon/: Para o recebimento de beacons informando a existência de assinantes locais.

Para construção do cenário de avaliação Ribas; Spohn (10) utilizaram os mesmos cenário de Spohn (13) e para medir a latência das publicações os autores utilizaram a ferramenta MQTT broker latency measure tool, a qual foi alterada para comportar a federação **TUAempty citation**.

Ao comparar os dois cenários, a solução federada apresentou latência menor nos assinantes mais distantes tanto para 500 como para 1000 publicações enquanto a solução centralizada apresentou latências menores para os assinantes mais próximos.

Em ambos os cenários, com 500 publicações, os assinantes mais próximos ao publicador obtiveram uma latência menor comparada aos mais distantes, e quando feitas 1000 publicações os assinantes mais distantes tiveram uma redução, enquanto os mais próximos tiveram um aumento.

Como justificam os autores, a diferença entre os dois cenários pode ser causado devido a uma sobrecarga na federação causada pelo roteamento de mensagens, já que o mesmo explora toda redundância possível, já no cenário de *broker* único o *overhead* é mínimo pois existe apenas um único caminho.

Os resultados obtidos na experimentação encontram-se detalhados nas Figuras 10 e 11.

Publicações	Assinante no *broker* 3	Assinante no *broker* 8
500	17,21±9,17 ms	9.51±6,58 ms
1000	16,71±9,53 ms	12,35±10,70 ms

Figura 10 – Resultados do cenário utilizando federação de *brokers*. Fonte Ribas; Spohn (10)

Publicações	*Broker* no nodo 3		*Broker* no nodo 8	
	Sub 1	Sub 2	Sub 1	Sub 2
500	19,81±16,21 ms	18,80±13,05 ms	3,33±4,35 ms	3,34±4,35 ms
1000	17,66±15,47 ms	17,68±15,49 ms	5,04±7,05 ms	5,10±7,12 ms

Figura 11 – Resultados do cenário utilizando *broker* único. Fonte Ribas; Spohn (10)

5 DESENVOLVIMENTO

Nesta seção serão descritas as principais ferramentas e a arquitetura utilizada na implementação de um microsserviço e um federador. O microsserviço, responsável pela criação e manutenção da topologia da federação de *brokers*, utiliza algoritmos para encontrar vizinhos para um nó ingressante e monitoramento da topologia em tempo de execução, além de armazenar as informações da topologia em um banco de dados MongoDB. Já o federador, por sua vez, fornece os mecanismos para o funcionamento do protocolo de federação de *brokers*.

5.1 FERRAMENTAS

5.1.1 Docker

Docker (3) é um sistema de virtualização a nível de sistema operacional capaz de fornecer ambientes totalmente isolados chamados de containers. Um container agrupa todo o software e suas dependências, o que acaba por agilizar tanto o processo de desenvolvimento quanto o de *deploy*, além de fornecer um ambiente seguro para execução em máquinas distintas.

5.1.2 MongoDB

MongoDB (8) é um software de banco de dados orientado a documentos classificado como um banco NoSQL, seu modelo documental facilita a sua utilização poupando tempo de desenvolvimento e manutenção.

5.1.3 Eclipse Mosquitto

Também desenvolvido pela Eclipse Foundation, o Eclipse Mosquitto (5) é um *broker* que implementa o protocolo MQTT. Destaca-se por ser leve e confiável podendo ser utilizado tanto em placas de baixo desempenho quanto em grandes servidores.

5.1.4 Eclipse Paho

Sendo um projeto da Eclipse Foundation, o Eclipse Paho (6) é uma biblioteca de código aberto que fornece a implementação de um cliente capaz de se conectar a *brokers* MQTT e usufruir de suas funcionalidades, possui suporte a inúmeras linguagens de programação.

5.1.5 MQTT *broker latency measure tool*

Tendo como principal função a de executar *benchmarks* em *brokers*, o MQTT *broker latency measure tool* (7) dispõe de uma vasta gama de configurações, além da coleta de diversas

métricas que podem ser utilizadas para mensurar o desempenho de um *broker*. para utilização neste projeto a ferramenta foi modificada com a finalidade de ser utilizada como um pacote para a linguagem Go.

5.2 ARQUITETURA DO CONJUNTO

Ambas as aplicações, um microsserviço e um federador, foram desenvolvidas utilizando a linguagem de programação Go e disponibilizadas[1] como imagens Docker. Os detalhes específicos do funcionamento do conjunto estão descritos a seguir.

5.2.1 Microsserviço

Para a implementação[2] do microsserviço optamos por utilizar o modelo de comunicação híbrida, que combina tanto a comunicação síncrona quanto a assíncrona. Para fornecer suporte a comunicação síncrona, utilizamos o modelo de *REST API's*, já para a comunicação assíncrona, utilizamos clientes MQTT que publicam mensagens diretamente no *broker* hospedeiro dos federadores através do tópico "federated_topology_ann". Na camada de dados, para fazer o armazenamento de informações referentes a topologia, utilizamos um banco de dados MongoDB.

Para cumprir seu papel de criação e manutenção da topologia, o microsserviço utiliza dois algoritmos, o primeiro deles é responsável por encontrar vizinhos para um nó ingressante e o outro, nomeado de *health check*, recebe a responsabilidade de fazer o monitoramento da topologia em tempo de execução.

Para receber solicitações de ingresso, o microsserviço disponibiliza um endpoint *HTTP (/api/v1/join)* para onde os federeradores devem enviar uma solicitação do tipo *POST* contendo no corpo um objeto JSON com o atributo "ip", este atributo recebe como valor a URL de conexão para o *broker* hospedeiro do federador. Um exemplo da requisição pode ser observado na Listagem 5.1.

```
curl --location --request POST 'http://127.0.0.1/api/v1/join' \
    --header 'Content-Type: application/json' \
    --data-raw '{"ip": "tcp://127.0.0.1:1883"}'
```

Listagem 5.1 – Solicitação de ingresso de um federador

O microsserviço busca sempre conectar novos nós a 2 outros, entretanto, enquanto não existirem nós suficientes para contemplar este requisito, os dois federadores que ingressarem primeiro serão simplesmente conectados um ao outro. Cada nó recebe um identificador inteiro e crescente baseado na ordem de ingresso na topologia, este identificador, juntamente com os demais dados referentes ao nó, são armazenados no banco de dados através de um documento com 6 campos, são eles:

[1] https://hub.docker.com/u/brunobevilaquaa
[2] https://gitlab.com/bruno.bevilaquaa/fed-topology-manager

- id: Identificador do federador.
- ip: URL para conexão com o *broker* hospedeiro do federador.
- neighbors: *Array* de objetos contendo o id e o ip de cada vizinho.
- neighborsAmount: Número de vizinhos.
- latency: Métrica coletada pelo *health check*.
- latestHealthCheck: Carimbo de data/hora da última execução do *health check* para aquele nó.

Na Listagem 5.2 é possível visualizar um documento que contém os dados de um federador.

```
{
    "id": 0,
    "ip": "tcp://127.0.0.1:1883"
    "neighbors": [{
        "id": 1,
        "ip": "tcp://127.0.0.1:1884"
    }],
    "neighborsAmount": 1,
    "latency": 0.10,
    "latestHealthCheck": "2023-01-01 00:00:00.000Z"
}
```

Listagem 5.2 – Dado referente a 1 nó com 1 vizinho

Após a inserção do primeiro nó, a cada 5 segundos, o algoritmo de *health check* dá início a uma varredura em todos os nós da topologia com a finalidade de coletar uma métrica específica, o tempo de latência entre uma publicação e uma resposta. Com esta métrica é possível determinar o desempenho do *broker* hospedeiro, e é utilizada principalmente como critério para inserção de novos federadores.

A partir do momento em que existirem pelo menos 2 nós na topologia, o microsserviço passa a realizar processo representado na Figura 12 no momento da inserção de novos nós. Ao receber uma solicitação de ingresso, o microsserviço faz a busca por 2 nós. O primeiro escolhido será o nó que possui a menor latência e número de vizinhos inferior a redundância máxima configurada para a topologia, já o segundo, utilizado para o complemento da redundância, é escolhido apenas com o critério de possuir a menor quantidade de vizinhos dentre todos os nós.

Como demonstrado na Figura 13 O *health check* também é responsável pela manutenção da topologia. Caso a coleta de uma métrica falhe duas vezes consecutivas para um mesmo federador, o mesmo é considerado desconectado. A partir da identificação da falha, utilizando

a conexão assíncrona o microsserviço faz o envio de uma mensagem aos vizinhos do nó *offline* informando que o mesmo não encontra-se mais disponível. Na sequência, uma busca pelos vizinhos do nó indisponível é feita, para cada vizinho é verificado se o mesmo ainda possui conexão com pelo menos um nó da topologia, caso algum deles fique totalmente desconectado, o microsserviço executará o processo de obtenção de novos vizinhos para o federador em questão, realocando-o de modo em que o mesmo retome a conexão com a federação.

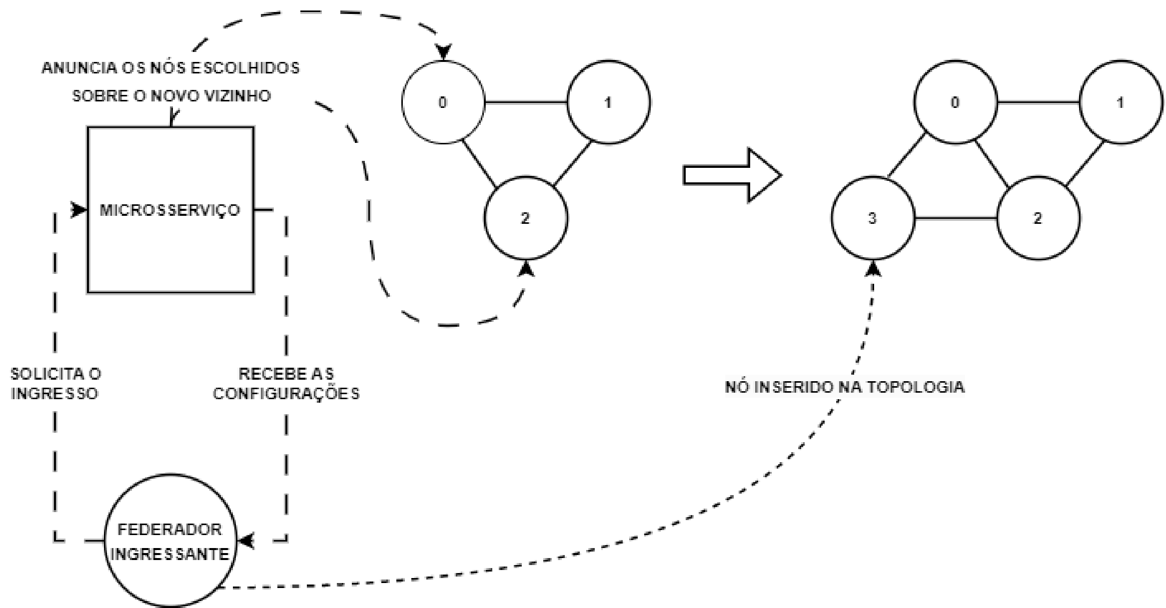


Figura 12 – Inserção de novos nós a topologia.

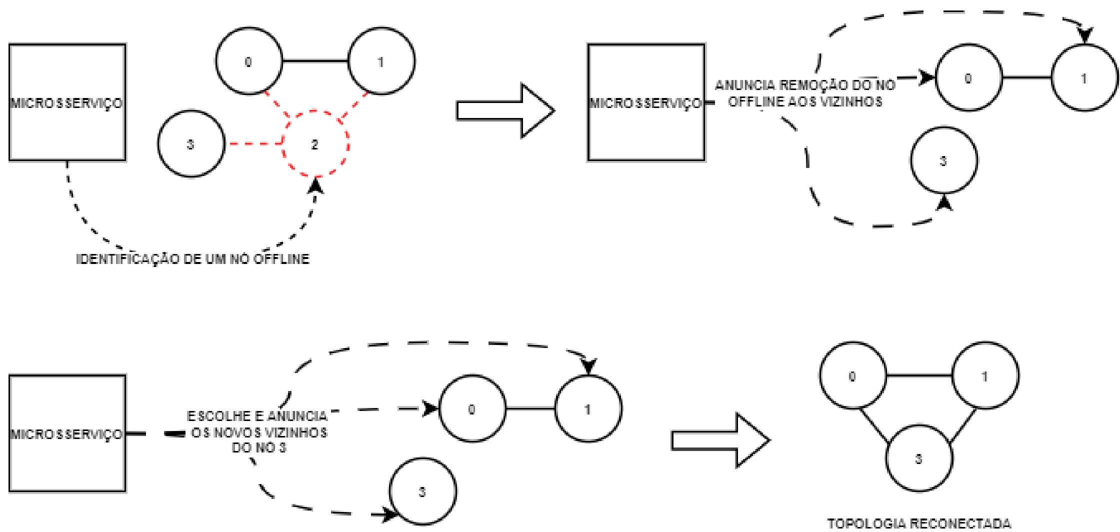


Figura 13 – Reconexão de um nó através do mecanismo *health check*.

5.2.2 Federador

No desenvolvimento[3] do federador utilizamos a mesma estrutura proposta por Ribas; Spohn (10). Nesta etapa o federador passou por uma tradução do código original escrito na linguagem de programação Rust para a linguagem de programação Go, mesmo assim, alterações foram feitas no federador tornando possível a integração das funcionalidades contempladas pelo microsserviço.

Inicialmente adicionamos ao federador a capacidade de fazer requisições HTTP ao microsserviço para que o mesmo busque as informações necessárias para sua execução.

Além disso, o federador também assina o novo tópico federated_topology_ann ao qual o microsserviço faz o envio de anúncios da topologia. Cada anúncio contém instruções para adição ou remoção de um vizinho, desta forma, o federador pode realizar a conexão ou desconexão com outro federador quando necessário.

Para processar as intruções recebidas, no mesmo nível do *dispatcher* foi adicionado um *handler* que altera os dados conhecidos pelos trabalhadores de tópicos.

5.2.3 Configuração

A configuração de ambas as aplicações é feita através de variáveis de ambiente.

Para o federador apenas duas configurações são necessárias:

- TOPOLOGY_MANAGER_URL: URL de conexão com o microsserviço.
- ADVERTISED_LISTENER: URL para conexão externa para com o *broker* hospedeiro do federador.

Já o microsserviço, além de suas próprias configurações, centraliza também as configurações da federação, necessitando de mais parâmetros de configuração:

- MONGO_URL: URL de conexão com o banco de dados MongoDB.
- CORE_ANN_INTERVAL: Valor positivo concatenado a uma unidade de tempo abreviada ("ms", "s", "m", "h"). Define o intervalo entre anúncios de núcleo.
- BEACON_INTERVAL: Valor positivo concatenado a uma unidade de tempo abreviada ("ms", "s", "m", "h"). Define o intervalo entre *beacons* recebidos por assinantes.
- FED_REDUNDANCY: Valor positivo que define a redundância das sub-malhas criadas pela federação.
- TOP_MAX_REDUNDANCY: Valor positivo que define a redundância máxima para um nó na topologia.

[3] `https://gitlab.com/bruno.bevilaquaa/mqtt-fed`

O exemplo da Listagem 5.3 é referente a um arquivo no formato YAML, que serve como *template* para a execução de uma federação através da ferramenta Docker Compose, disponibilizada pelo Docker. Este arquivo contempla os dados para execução de 3 containers, referentes ao banco de dados, microsserviço e federador, bem como suas variáveis de configuração.

```yaml
version: '3'
services:
  mongodb:
    image: mongo:latest
    environment:
      - MONGO_INITDB_DATABASE=root

  microsservice:
    image: brunobevilaquaa/fed-topology-manager
    depends_on:
      - mongodb
    environment:
      - MONGO_URL=mongodb://mongodb:27017/root
      - CORE_ANN_INTERVAL=2s
      - BEACON_INTERVAL=2s
      - FED_REDUNDANCY=2
      - TOP_MAX_REDUNDANCY=5

  federator:
    image: brunobevilaquaa/mqtt-fed
    depends_on:
      - microsservice
    ports:
      - '1883:1883'
    environment:
      - TOPOLOGY_MANAGER_URL=http://topology-manager:8080
      - ADVERTISED_LISTENER=tcp://mqtt-fed:1883
```

Listagem 5.3 – Configuração para *deploy* dos containers do conjunto

5.3 CENÁRIO DE AVALIAÇÃO

Para a avaliação, dois cenários de implantação foram criados, o primeiro utilizando computação em nuvem e o segundo uma rede local. Para ambos os cenários o microsserviço foi utilizado para gerar a topologia da federação.

Ao fazer a utilização da computação em nuvem buscamos explorar ambientes encontrados em implantações reais, tendo como principal objetivo a construção de um cenário para teste de disponibilidade da federação. Com este teste é possível visualizar como o microsserviço adapta a topologia quando ocorre uma eventual desconexão de um federador. Para construção do cenário foram utilizadas as plataformas AWS, Azure e Google Cloud para computação em nuvem, nelas

um total de 10 máquinas virtuais foram criadas, 9 para federação e 1 para o microsserviço, todas dispostas em regiões geográficas diferentes nos continentes América, Europa, África e Ásia. Para o teste, inicialmente coletamos a topologia gerada pelo microsserviço e em seguida interrompemos a execução de 2 federadores simulando uma desconexão dos mesmos.

O teste ao qual foi utilizado uma rede local teve como principal objetivo a criação de um cenário para teste de desempenho. Por ser um ambiente mais controlado e livre de interferências que a rede e o desempenho computacional podem proporcionar, métricas mais assertivas puderam ser coletadas. O microsserviço foi configurado para gerar topologias com redundância máxima 5, já a federação foi configurada para possuir redundância 3. Ao total foram utilizados 12 federadores gerando a topologia ilustrada na Figura 14. A partir dela, dois assinantes foram posicionados nos federadores 0 e 5 respectivamente. Um publicador responsável por fazer o envio de 1000 mensagens, cada uma com 64 bytes, foi posicionado no nó 11, a 1 salto do federador 0 e a 3 saltos do federador 5.

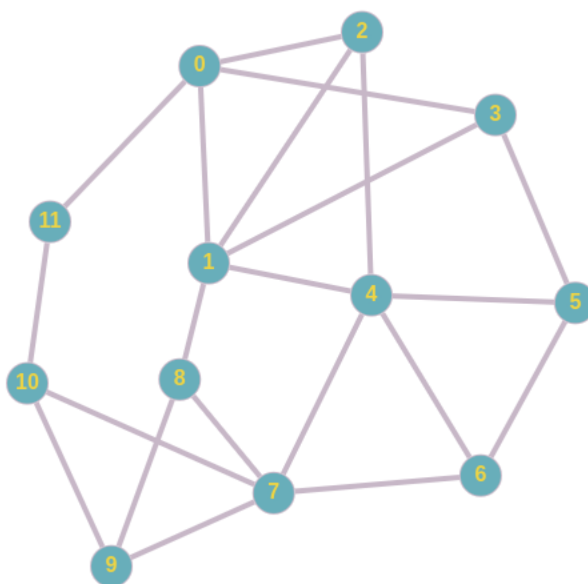


Figura 14 – Topologia utilizada no cenário de avaliação de desempenho.

6 RESULTADOS

6.1 CENÁRIO DE TESTE DE DISPONIBILIDADE

A topologia obtida inicialmente no cenário utilizando computação em nuvem encontra-se relatada na Figura 15. É possível notar que o nó 8 possui a menor quantidade de vizinhos da topologia, o que o torna um alvo mais fácil de isolamento, portanto, ao parar a execução de seus vizinhos (0 e 7), faz-se necessário uma realocação.

Ao realizar a realocação, como demonstra a Figura 16, o microsserviço fez a escolha de novos vizinhos para o federador 8. O federador 3 foi escolhido como o de maior desempenho, isto é possível afirmar já que o mesmo possui o maior número de conexões dentre todos, e o federador 1 que foi obtido para complemento de redundância. É possível observar que a reconexão do mesmo foi realizada com sucesso.

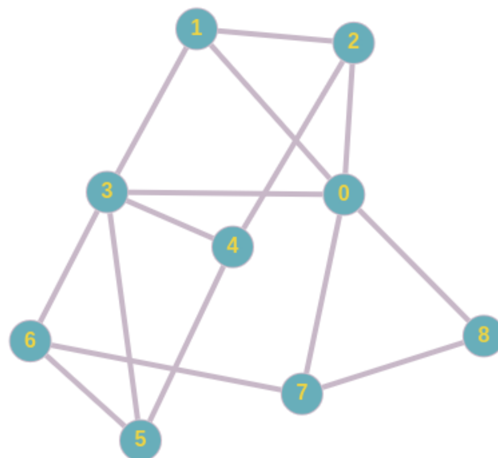


Figura 15 – Topologia antes da realocação do nó 8.

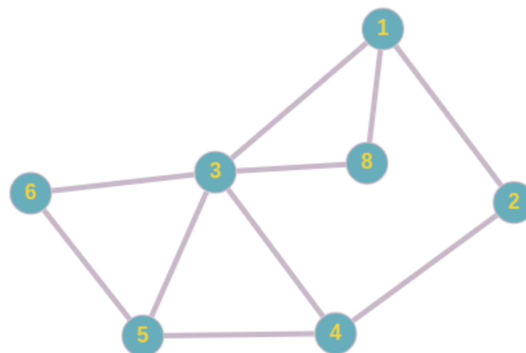


Figura 16 – Topologia após a realocação do nó 8.

6.2 CENÁRIO DE TESTE DE DESEMPENHO

Os resultados coletados utilizando o cenário de rede local encontram-se na tabela 1. Ao comparar a latência média das publicações entre os dois consumidores é possível notar que o *broker* que se encontra mais distante do publicador apresenta valores maiores. Ao analisar estes resultados deve ser levado em conta o *overhead* de publicações gerados durante as retransmissões, fator que acaba colaborando para um aumento considerável da latência.

Publicações	**Assinante no *broker* 0**	**Assinante no *broker* 5**
1000	445.796ms	1062.230ms

Tabela 1 – Latência de publicações da federação.

7 CONCLUSÃO

Com o presente trabalho, foi possível concluir que a infraestrutura proposta consegue agregar funcionalidades a federação de *brokers*, de uma forma em que os federadores não precisam lidar com tarefas além das necessárias.

Os resultados coletados mostram que o microsserviço implementado consegue fazer com êxito seu papel, e que mesmo sendo uma infraestrutura centralizada, devido às suas propriedades de auto escala herdadas da arquitetura de microsserviços, não oferece riscos para a execução da federação. Além disso, a nova versão do federador e o microsserviço, implementados e disponibilizados na linguagem Go, podem servir como base para pesquisa e desenvolvimento de novas variantes do protocolo.

7.1 TRABALHOS FUTUROS

A partir do momento em que a arquitetura de microsserviços foi incluída no ambiente da federação, uma nova janela de possibilidades foi aberta. Para trabalhos futuros, podem ser citados a inclusão de novos microsserviços capazes de fornecer ainda mais funcionalidades a federação, como mecanismos de autenticação, autorização e esquemas de validação de mensagens.

REFERÊNCIAS

1 AKSAKALLI, Işıl Karabey et al. Deployment and communication patterns in microservice architectures: A systematic literature review. **Journal of Systems and Software**, Elsevier, v. 180, p. 111014, 2021.

2 ALSHUQAYRAN, Nuha; ALI, Nour; EVANS, Roger. A systematic mapping study in microservice architecture. In: IEEE. 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA). [S.l.: s.n.], 2016. P. 44–51.

3 DOCKER Develop faster. Run anywhere. [S.l.: s.n.]. Disponível em: `https://www.docker.com`. Acesso em: 02 jan. 2023.

4 DRAGONI, Nicola et al. Microservices: yesterday, today, and tomorrow. **Present and ulterior software engineering**, Springer, p. 195–216, 2017.

5 ECLIPSE FOUNDATION. **Eclipse Mosquitto, An open source MQTT broker**. [S.l.: s.n.]. Disponível em: `https://mosquitto.org`. Acesso em: 02 jan. 2023.

6 ________. **Paho**. [S.l.: s.n.]. Disponível em: `https://www.eclipse.org/paho`. Acesso em: 02 jan. 2023.

7 JIANHUI, ZHANG XIANG. **MQTT broker latency measure tool**. [S.l.: s.n.]. Disponível em: `https://github.com/hui6075/mqtt-bm-latency`. Acesso em: 02 jan. 2023.

8 MONGODB. [S.l.: s.n.]. Disponível em: `https://www.mongodb.com`. Acesso em: 02 jan. 2023.

9 MQTT The Standard for IoT Messaging. [S.l.: s.n.]. Disponível em: `https://mqtt.org`. Acesso em: 22 jul. 2022.

10 RIBAS, Nicolas Kolling; SPOHN, Marco Aurélio. A New Approach to a Self-Organizing Federation of MQTT Brokers. **Journal of Computer Science**, Science Publications, v. 18, n. 7, p. 687–694, jul. 2022. DOI: `10.3844/jcssp.2022.687.694`. Disponível em: <https://thescipub.com/abstract/jcssp.2022.687.694>.

11 ROSE, Karen; ELDRIDGE, Scott; CHAPIN, Lyman. The internet of things: An overview. **The internet society (ISOC)**, Reston, VA, v. 80, p. 1–50, 2015.

12 SONI, Dipa; MAKWANA, Ashwin. A survey on mqtt: a protocol of internet of things (iot). In: INTERNATIONAL conference on telecommunication, power analysis and computing techniques (ICTPACT-2017). [S.l.: s.n.], 2017. v. 20, p. 173–177.

13 SPOHN, Marco Aurélio. An Endogenous and Self-organizing Approach for the Federation of Autonomous MQTT Brokers. In: ICEIS (1). [S.l.: s.n.], 2021. P. 834–841.

14 ________. Publish, subscribe, and federate! **arXiv preprint arXiv:2006.03675**, 2020.

15 WORTMANN, Felix; FLÜCHTER, Kristina. Internet of things. **Business & Information Systems Engineering**, Springer, v. 57, n. 3, p. 221–224, 2015.