Universidade Federal do Rio Grande do Norte
Centro de Tecnologia
Departamento de Engenharia de Computação e Automação
Bacharelado em Engenharia de Computação

Plataforma Sapparchi: Integração de Comunicação Assíncrona e Roteamento Inteligente em Arquiteturas Edge-Fog-Cloud Continuum

Rafael Pinheiro Carlos Maia

Natal – RN, Brasil

Novembro de 2025

Rafael Pinheiro Carlos Maia

Plataforma Sapparchi: Integração de Comunicação Assíncrona e Roteamento Inteligente em Arquiteturas Edge-Fog-Cloud Continuum

Trabalho de Conclusão de Curso de Engenharia de Computação da Universidade Federal do Rio Grande do Norte, apresentado como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação

Orientador: Nélio Alessandro Azevedo Cacho

Universidade Federal do Rio Grande do Norte – UFRN

Departamento de Engenharia de Computação e Automação – DCA

Graduação em Engenharia de Computação

Natal – RN, Brasil

Novembro de 2025

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Maia, Rafael Pinheiro Carlos.
Plataforma Sapparchi: integração de comunicação assíncrona e roteamento inteligente em arquiteturas edge-fog-cloud continuum / Rafael Pinheiro Carlos Maia. - 2025.
77 f.: il.

Trabalho de Conclusão de Curso - TCC (graduação) - Universidade Federal do Rio Grande do Norte, Centro de Tecnologia, Curso de Engenharia de Computação, Natal, RN, 2025.
Orientação: Prof. Dr. Nélio Alessandro Azevedo Cacho.

1. Computação em Névoa - TCC. 2. Edge Computing - TCC. 3. Mensageria Assíncrona - TCC. 4. RabbitMQ - TCC. 5. MQTT - TCC. 6. Sistemas Distribuídos - TCC. I. Cacho, Nélio Alessandro Azevedo. II. Título.

RN/UF/BCZM CDU 004.7

Elaborado por Fernanda de Medeiros Ferreira Aquino - CRB-15/301

Rafael Pinheiro Carlos Maia

Plataforma Sapparchi: Integração de Comunicação Assíncrona e Roteamento Inteligente em Arquiteturas Edge-Fog-Cloud Continuum

Trabalho de Conclusão de Curso de Engenharia de Computação da Universidade Federal do Rio Grande do Norte, apresentado como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação

Orientador: Nélio Alessandro Azevedo Cacho

Trabalho aprovado. Natal, 08 de Dezembro de 2025:

Prof. Dr. Nélio Alessandro Azevedo Cacho
Orientador

Prof. Dr. Frederico Araujo da Silva Lopes
Examinador

Prof. Dr. Arthur Emanoel Cassio da Silva e Souza
Examinador

Natal – RN, Brasil
Novembro de 2025

Aos meus pais, à minha família, aos meus amigos, ao meu orientador, a todos que cruzaram meu caminho e aos que ainda cruzarão, por terem contribuído para quem eu sou hoje, moldando minha trajetória, apoio e experiências que levarei sempre comigo.

AGRADECIMENTOS

Todo começo é, de fato, uma continuação. E se cheguei até aqui, foi porque muitos vieram antes de mim, caminharam ao meu lado ou simplesmente cruzaram meu caminho no momento certo.

Agradeço aos meus pais, por serem o início de tudo. Pelo amor que não precisou de palavras, pelos sacrifícios que nunca fizeram questão de mencionar e pela confiança depositada mesmo quando o caminho parecia incerto. Vocês são a base de qualquer conquista minha.

À minha família, pelo abrigo seguro em meio às tempestades, pelo incentivo constante e por me lembrarem de onde vim e do que sou capaz.

Aos meus amigos e irmãos, pelo apoio mesmo quando não merecia, pelos desabafos e críticas, pela presença, independente de tempo e distância.

Ao meu orientador, Prof. Dr. Nélio Alessandro Azevedo Cacho, pela paciência, pela confiança e pelos ensinamentos que vão muito além do técnico. Mais do que um orientador, um mestre que me mostrou que todo o esforço depositado na busca pelo conhecimento é, invariavelmente, recompensador.

A todos os professores que passaram pela minha trajetória, cada um deixando uma marca, uma lição, uma forma diferente de ver o mundo.

E a todos que, de alguma forma, contribuíram para quem sou hoje, pois nenhum fio se entrelaça por acaso: cada encontro tem seu motivo, sua causa e sua razão.

Por fim, agradeço aos que ainda virão. Porque, como bem disse Szymborska, o livro dos acontecimentos está sempre aberto pela metade, e as melhores páginas podem ainda estar por ser escritas.

"Every beginning
is only a sequel, after all,
and the book of events
is always open halfway through."

Wislawa Szymborska

RESUMO

Este trabalho apresenta o desenvolvimento e integração de um sistema de mensageria assíncrona baseado em RabbitMQ para a plataforma Sapparchi, uma solução de orquestração de tarefas distribuídas em ambientes Edge-Fog-Cloud. A solução proposta implementa comunicação assíncrona com suporte a múltiplos protocolos, incluindo AMQP para nós de Fog e Cloud e MQTT para dispositivos de borda com recursos limitados, além de HTTP como mecanismo de fallback. Foi desenvolvido um algoritmo de roteamento adaptativo que seleciona automaticamente o protocolo mais adequado conforme a camada computacional do nó de destino. O trabalho também inclui a implementação de um mecanismo de monitoramento de saúde de nós com circuit breaker, capaz de detectar falhas e evitar o envio de requisições para workers indisponíveis. Por fim, foram criados mecanismos de registro e re-registro automático que garantem a recuperação do sistema após falhas do Manager ou dos Workers. Os resultados qualitativos indicam aumento da resiliência do sistema e melhor adequação aos cenários de conectividade intermitente típicos de dispositivos de borda.

Palavras-chave: Computação em Névoa. Edge Computing. Mensageria Assíncrona. RabbitMQ. MQTT. Sistemas Distribuídos.

ABSTRACT

This work presents the development and integration of an asynchronous messaging system based on RabbitMQ for the Sapparchi platform, a distributed task orchestration solution for Edge-Fog-Cloud environments. The proposed solution implements asynchronous communication with support for multiple protocols, including AMQP for Fog and Cloud nodes and MQTT for resource-constrained edge devices, with HTTP as a fallback mechanism. An adaptive routing algorithm was developed to automatically select the most suitable protocol according to the computational layer of the target node. The work also includes the implementation of a node health monitoring mechanism with circuit breaker, capable of detecting failures and preventing requests from being sent to unavailable workers. Finally, automatic registration and re-registration mechanisms were created to ensure system recovery after Manager or Worker failures. Qualitative results indicate increased system resilience and better suitability for intermittent connectivity scenarios typical of edge devices.

Keywords: Fog Computing. Edge Computing. Asynchronous Messaging. RabbitMQ. MQTT. Distributed Systems.

LISTA DE ILUSTRAÇÕES

Figura 1 – Visão geral da arquitetura Edge–Fog–Cloud no contexto do Sapparchi. 18
Figura 2 – Uso de RabbitMQ (AMQP) no roteamento de mensagens do Sapparchi. 19
Figura 3 – Arquitetura geral da plataforma Sapparchi. 26
Figura 4 – Topologia de comunicação entre Manager e Workers no Sapparchi. O RabbitMQ atua como broker central, recebendo conexões AMQP de Workers Python/Java e conexões MQTT de Workers ESP32. 32
Figura 5 – Filas principais da arquitetura de mensageria. Filas unicast são exclusivas por Worker, enquanto filas de heartbeat e resultados são compartilhadas. 33
Figura 6 – Arquitetura de filas separadas. Workers publicam na fila intermediária (1), o Manager consolida (2) e publica o resultado final (3). 40
Figura 7 – Fluxo de decisão para escolha de protocolo. A seleção inicial depende da camada; tentativas subsequentes seguem a ordem de prioridade até sucesso ou esgotamento das opções. 47
Figura 8 – Arquitetura do serviço NodeHealth. O dispatcher consulta o estado de saúde antes de enviar mensagens; falhas são reportadas de volta para atualizar o estado. 49
Figura 9 – Fluxo de heartbeats do Manager. 55
Figura 10 – Máquina de estados de registro. Transições ocorrem por eventos de conexão, heartbeats e quedas do Manager. 64
Figura 11 – Arquitetura de duas filas. Fluxo: Worker → fila intermediária → Manager consolida → fila final → Cliente. 65
Figura 12 – Registro inicial: Worker conecta, publica registro, Manager valida e envia ACK. 68
Figura 13 – Re-registro automático: Workers detectam retorno do Manager e re-registram imediatamente. 69
Figura 14 – Re-registro periódico: refresh automático a cada 50 segundos mantém consistência. 70
Figura 15 – Execução de action individual: requisição, dispatch, execução e consolidação. 71
Figura 16 – Execução de task multi-step: orquestração com passagem de resultados entre steps. 72

LISTA DE TABELAS

Tabela 1 – Comparativo entre trabalhos relacionados e a solução proposta 24

Tabela 2 – Campos da estrutura Task: origem e obrigatoriedade 29

Tabela 3 – Rotas configuradas na exchange `amq.topic`. Cada routing key direciona mensagens para consumidores específicos. 32

Tabela 4 – Prioridade de protocolos por camada e justificativa técnica. 41

Tabela 5 – Estados de saúde de nós no sistema NodeHealth. 53

Tabela 6 – Estados de registro de Workers no Sapparchi. 65

Tabela 7 – Comparação entre arquitetura original e arquitetura com filas separadas. 67

LISTA DE ABREVIATURAS E SIGLAS

AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
CLOUD	Camada de computação em nuvem
CPU	*Central Processing Unit*
EDGE	Camada de computação na borda
ESP32	Microcontrolador de baixo custo com WiFi e Bluetooth
FOG	Camada de computação em névoa
HTTP	*HyperText Transfer Protocol*
IoT	*Internet of Things* (Internet das Coisas)
JSON	*JavaScript Object Notation*
MQTT	*Message Queuing Telemetry Transport*
QoS	*Quality of Service*
RAM	*Random Access Memory*
REST	*Representational State Transfer*
TCP	*Transmission Control Protocol*
TLS	*Transport Layer Security*
URN	*Uniform Resource Name*
YAML	*YAML Ain't Markup Language*
ML	*Machine Learning*

SUMÁRIO

1 Introdução **15**
1.1 Motivação 15
1.2 Objetivos 16
1.3 Estrutura do Documento 16

2 Fundamentação Teórica **17**
2.1 Arquitetura Edge-Fog-Cloud 17
2.1.1 O Conceito de Continuum 17
2.2 Mensageria Assíncrona 18
2.2.1 RabbitMQ 18
2.2.2 MQTT 19
2.3 Padrões de Comunicação 19
2.3.1 Request-Reply 19
2.3.2 Publish-Subscribe 20
2.3.3 Message Queue 20
2.4 Plataforma Sapparchi 20
2.4.1 Modelo de Programação 21
2.4.2 Arquitetura Original 21
2.4.3 Limitações Identificadas 21
2.5 Trabalhos Relacionados 22
2.5.1 Ferramentas de Simulação 22
2.5.2 Frameworks de Borda 22
2.5.3 Arquiteturas de Referência 23
2.5.4 Análise Comparativa 23
2.5.5 Contribuição deste Trabalho 24

3 Arquitetura do Sistema **25**
3.1 Visão Geral 25
3.2 Componentes do Sistema 26
3.2.1 Sapparchi Manager 26
3.2.2 Sapparchi Worker (Python) 27
3.2.3 Worker Node ESP32 27
3.3 Modelo de Dados 28
3.3.1 Action 28
3.3.2 Task 28

3.3.3 Message 29

4 Contribuições Desenvolvidas 31

4.1 Integração de Comunicação Assíncrona 31
4.2 Exchanges e Filas Implementadas 32
4.2.1 Implementação no Manager (Java) 34
4.2.2 Implementação no Worker Python 35
4.2.3 Implementação no Worker ESP32 38
4.2.4 Separação de Filas: Intermediárias vs Finais 40
4.3 Roteamento Inteligente por Camadas 41
4.3.1 Algoritmo de Seleção de Protocolo 41
4.3.2 Implementação: ProtocolResolver 41
4.3.3 Transports Polimórficos 42
4.3.4 Dispatcher com Fallback Automático 44
4.3.5 Considerações sobre Latência por Protocolo 48
4.4 Sistema de Monitoramento de Saúde (NodeHealth) 48
4.4.1 Arquitetura do NodeHealth 48
4.4.2 Implementação do NodeHealthService 50
4.4.3 Integração com o Dispatcher 52
4.4.4 Benefícios Observados 53
4.5 Registro e Re-registro Automático 53
4.5.1 Mecanismo de Heartbeat 54
4.5.2 Implementação: ManagerHeartbeatPublisher 56
4.5.3 Implementação: Worker Python 58
4.5.4 Implementação: Worker ESP32 59
4.5.5 Mecanismos Anti-Flood 62
4.5.6 Resumo de Estados 63
4.6 Separação de Filas de Resultados 65
4.6.1 Problema Original 65
4.6.2 Solução Implementada 65
4.6.3 Implementação: Worker Python 66
4.6.4 Implementação: Worker ESP32 67
4.6.5 Vantagens da Separação 67
4.7 Fluxos de Execução 67
4.7.1 Fluxo 1: Registro Inicial 67
4.7.2 Fluxo 2: Re-registro após Reinício do Manager 68
4.7.3 Fluxo 3: Re-registro Periódico 69
4.7.4 Fluxo 4: Execução de Action Individual 70
4.7.5 Fluxo 5: Execução de Task Multi-Step 71

5 Conclusões **73**
5.1 Contribuições 73
5.2 Resultados Qualitativos 74
5.3 Trabalhos Futuros 74
5.4 Considerações Finais 75

Referências **77**

APÊNDICE A Configurações do Sistema **78**
A.1 application.yml (Manager) 78
A.2 config.yaml (Worker Python) 78
A.3 mqtt
config.h (ESP32) 79

1 INTRODUÇÃO

O conceito de Cidades Inteligentes (*Smart Cities*) refere-se ao uso de tecnologias como Computação em Nuvem, Big Data e Internet das Coisas (IoT) para desenvolver soluções voltadas ao bem-estar social dos cidadãos (Souza, 2022). Aplicações nesse contexto abrangem desde o monitoramento de tráfego veicular até sistemas de qualidade do ar e assistência médica remota. Essa diversidade de domínios e tecnologias traz grandes desafios para desenvolvedores e provedores de infraestrutura.

A proliferação de dispositivos IoT e a demanda por processamento em tempo real têm impulsionado o desenvolvimento de arquiteturas computacionais distribuídas. O paradigma Edge-Fog-Cloud emergiu como solução para reduzir latência, economizar largura de banda e processar dados próximo à sua origem (Bonomi et al., 2012).

Nesse contexto, a plataforma Sapparchi foi desenvolvida como uma solução para orquestrar a execução de funções serverless (chamadas *Actions*) e fluxos de trabalho (*Tasks*) em nós distribuídos através das camadas Edge, Fog e Cloud (Souza et al., 2022). A plataforma utiliza um modelo de programação baseado em processamento de fluxo de dados, similar ao adotado por plataformas como OpenWhisk e ThingWorx.

No entanto, a versão original do Sapparchi utilizava comunicação síncrona baseada em HTTP, apresentando limitações significativas em cenários com alta latência, conectividade intermitente e necessidade de processamento assíncrono — características comuns em ambientes de borda com dispositivos de recursos limitados como microcontroladores ESP32[1].

1.1 Motivação

A análise da arquitetura original do Sapparchi revelou diversos desafios que motivaram o desenvolvimento deste trabalho. O primeiro deles diz respeito ao acoplamento temporal imposto pela comunicação síncrona, na qual o Manager permanece bloqueado aguardando respostas dos Workers. Esse modelo se mostra incompatível com dispositivos de conectividade intermitente, comuns em cenários de borda.

Outro aspecto problemático identificado foi a ausência de mecanismos de resiliência. O sistema original não dispunha de estratégias de retry automático, circuit breaker ou recuperação após falhas, o que comprometia a estabilidade em ambientes distribuídos sujeitos a instabilidades de rede.

[1] ESPRESSIF SYSTEMS. *ESP32 Series Datasheet*. Disponível em: https://www.espressif.com/en/products/socs/esp32. Acesso em: 8 dez. 2025.

Dispositivos de borda como o ESP32 também enfrentavam dificuldades com o overhead do protocolo HTTP, que se mostrava inadequado para enlaces de baixa largura de banda e alta latência. Além disso, o roteamento de mensagens era estático, sem considerar as características específicas de cada camada computacional na escolha do protocolo de comunicação.

Por fim, a plataforma carecia de mecanismos adequados de monitoramento, sendo incapaz de detectar falhas ou indisponibilidade dos nós distribuídos de forma proativa.

1.2 Objetivos

O objetivo geral deste trabalho consiste em integrar comunicação assíncrona baseada em mensageria à plataforma Sapparchi, implementando roteamento inteligente por camadas e monitoramento de saúde de nós distribuídos.

Para alcançar esse objetivo, foram definidos os seguintes objetivos específicos: implementar comunicação assíncrona utilizando RabbitMQ com suporte aos protocolos AMQP e MQTT; desenvolver um algoritmo de roteamento adaptativo capaz de selecionar o protocolo mais adequado para cada camada, empregando MQTT para nós Edge e AMQP para nós Fog e Cloud; criar um sistema de monitoramento de saúde de nós (*NodeHealth*) com padrão circuit breaker; implementar mecanismos de registro e re-registro automático de workers; separar as filas de resultados em intermediárias e finais para suporte a tasks compostas por múltiplas actions; e validar funcionalmente a solução proposta através de fluxos de execução em ambiente distribuído.

1.3 Estrutura do Documento

Este trabalho está organizado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica, incluindo conceitos de arquitetura Edge-Fog-Cloud, mensageria assíncrona e a plataforma Sapparchi original. O Capítulo 3 descreve a arquitetura do sistema proposto. O Capítulo 4 detalha as contribuições desenvolvidas. Por fim, o Capítulo 5 apresenta as conclusões e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Arquitetura Edge-Fog-Cloud

A arquitetura Edge–Fog–Cloud é um modelo hierárquico de computação distribuída que organiza recursos computacionais em três camadas distintas (Satyanarayanan, 2017). A camada Cloud, ou Nuvem, concentra recursos computacionais abundantes em data centers centralizados, oferecendo alta capacidade de processamento e armazenamento, porém com maior latência devido à distância física dos dispositivos de origem dos dados. A camada Fog, ou Névoa, posiciona servidores intermediários entre a nuvem e os dispositivos finais, proporcionando latência moderada e capacidade de pré-processamento dos dados antes de seu envio à nuvem. Por fim, a camada Edge, ou Borda, compreende dispositivos próximos à origem dos dados, como gateways e microcontroladores, oferecendo latência reduzida às custas de recursos computacionais mais limitados.

2.1.1 O Conceito de Continuum

O termo *Edge–Fog–Cloud Continuum* refere-se à visão de que essas camadas não são silos isolados, mas um espectro contínuo de recursos computacionais (Balouek-Thomert et al., 2019). Diferentemente de arquiteturas tradicionais com fronteiras rígidas, o continuum reconhece que os recursos variam gradualmente entre as camadas, sem fronteiras precisas delimitando onde Edge, Fog e Cloud começam ou terminam. Nessa perspectiva, o roteamento de workloads pode ser fluido, permitindo que tarefas sejam migradas ou distribuídas entre camadas de acordo com requisitos da aplicação e políticas de execução. Além disso, os protocolos de comunicação tendem a ser adaptativos, de modo que a escolha entre MQTT, AMQP e HTTP pode considerar o contexto de rede, o perfil do dispositivo e os requisitos de confiabilidade, em vez de ficar rigidamente associada a uma camada fixa.

A plataforma Sapparchi implementa essa visão ao integrar, em um mesmo ecossistema, nós de Edge, Fog e Cloud por meio de uma camada unificada de comunicação. Na versão atual, o roteamento é definido por regras de camada: nós de borda utilizam predominantemente MQTT e HTTP, enquanto nós de névoa podem empregar AMQP, MQTT e HTTP. Embora a escolha de nó e de protocolo ainda não seja realizada de forma dinâmica com base em métricas como latência, disponibilidade ou carga, as três camadas são tratadas como um ambiente contínuo de recursos distribuídos. Esse desenho arquitetural caracteriza um *Edge–Fog–Cloud continuum* e fundamenta trabalhos futuros voltados à incorporação de mecanismos de roteamento efetivamente adaptativos.

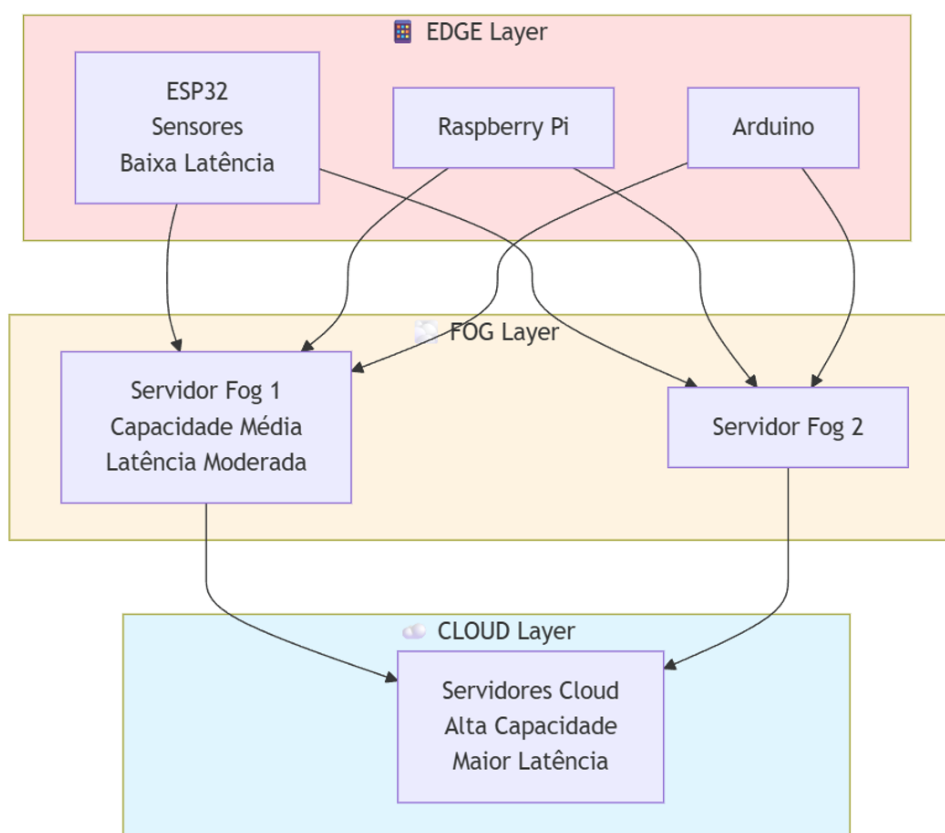

Figura 1 – Visão geral da arquitetura Edge–Fog–Cloud no contexto do Sapparchi.

2.2 Mensageria Assíncrona

Sistemas de mensageria assíncrona desacoplam produtores e consumidores de mensagens por meio de intermediários conhecidos como *message brokers* (Hohpe; Woolf, 2004). Esse modelo de comunicação oferece diversos benefícios para sistemas distribuídos. O desacoplamento temporal permite que produtor e consumidor não precisem estar ativos simultaneamente, enquanto o desacoplamento espacial elimina a necessidade de conhecimento direto entre as partes envolvidas. Além disso, a escalabilidade é favorecida pela possibilidade de múltiplos consumidores processarem mensagens em paralelo, e a resiliência é aumentada pela capacidade de persistir mensagens até a confirmação de processamento.

2.2.1 RabbitMQ

RabbitMQ é um *broker* de mensagens de código aberto que implementa o protocolo AMQP (Advanced Message Queuing Protocol) (Videla; Williams, 2012). O modelo de roteamento do RabbitMQ é baseado em três conceitos principais: exchanges, queues e bindings. As exchanges são componentes responsáveis pelo roteamento de mensagens, podendo operar em diferentes modos como fanout, direct, topic e headers. As queues são

filas que armazenam mensagens até o momento do consumo. Os bindings são regras que associam exchanges a filas, definindo como o roteamento é realizado. O RabbitMQ também oferece suporte à persistência de filas e mensagens, garantindo durabilidade dos dados, além de mecanismos de acknowledgment que confirmam o processamento bem-sucedido das mensagens.

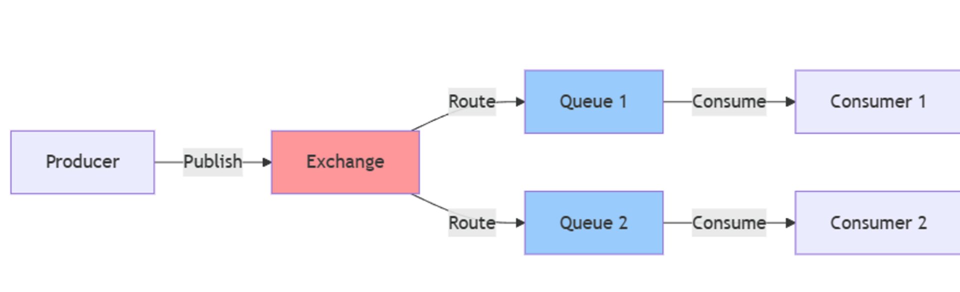

Figura 2 – Uso de RabbitMQ (AMQP) no roteamento de mensagens do Sapparchi.

2.2.2 MQTT

MQTT (Message Queuing Telemetry Transport) é um protocolo leve de mensageria baseado no padrão publish/subscribe, projetado para dispositivos com recursos limitados e conexões instáveis (OASIS, 2024). O protocolo se destaca pelo overhead mínimo de seus cabeçalhos, tornando-o adequado para enlaces com baixa largura de banda. O MQTT oferece três níveis de qualidade de serviço (QoS): nível 0, que entrega a mensagem no máximo uma vez sem confirmação; nível 1, que garante a entrega pelo menos uma vez; e nível 2, que assegura a entrega exatamente uma vez. Outros recursos relevantes incluem as retained messages, que mantêm a última mensagem publicada em um tópico para novos subscribers, e o Last Will and Testament, que permite configurar uma mensagem a ser enviada automaticamente em caso de desconexão inesperada do cliente. Essas características tornam o MQTT particularmente adequado para cenários de IoT, onde baixo consumo de energia e de largura de banda são requisitos frequentes.

2.3 Padrões de Comunicação

Os padrões de comunicação em sistemas distribuídos definem a forma como componentes trocam informações, sendo fundamentais para a arquitetura de aplicações modernas. Esta seção apresenta três padrões amplamente utilizados: Request-Reply, Publish-Subscribe e Message Queue (Hohpe; Woolf, 2004).

2.3.1 Request-Reply

O padrão Request-Reply é um modelo síncrono em que o cliente envia uma requisição e aguarda a resposta do servidor, sendo típico de interfaces HTTP REST (Hohpe; Woolf,

2004). Neste modelo, o solicitante (requestor) envia uma mensagem de requisição através de um canal e bloqueia sua execução até receber a mensagem de resposta correspondente do respondedor (replier). Sua principal vantagem reside na simplicidade de implementação e na adequação a operações de criação, leitura, atualização e exclusão (CRUD). Por outro lado, o acoplamento temporal entre cliente e servidor e a menor tolerância a falhas de comunicação representam suas principais desvantagens (Hohpe; Woolf, 2004).

2.3.2 Publish-Subscribe

O padrão Publish-Subscribe é um modelo assíncrono em que produtores publicam mensagens em tópicos e múltiplos consumidores podem se inscrever para recebê-las (Eugster et al., 2003). Este paradigma promove o desacoplamento total entre as entidades comunicantes em três dimensões: espaço (publishers e subscribers não precisam conhecer uns aos outros), tempo (não precisam estar ativos simultaneamente) e sincronização (publishers não são bloqueados ao produzir eventos) (Eugster et al., 2003). Entre suas vantagens destacam-se o desacoplamento entre produtores e consumidores, a escalabilidade natural do modelo e o suporte a broadcast de mensagens. As desvantagens incluem maior complexidade de gerenciamento e a possibilidade de inconsistência eventual entre os participantes (Eugster et al., 2003).

2.3.3 Message Queue

O padrão Message Queue é um modelo assíncrono em que uma fila intermediária garante a entrega ponto-a-ponto entre produtores e consumidores (Hohpe; Woolf, 2004). A fila atua como um buffer que armazena mensagens até que sejam processadas, permitindo que produtores e consumidores operem em velocidades diferentes e possibilitando o balanceamento de carga entre múltiplos consumidores (Hohpe; Woolf, 2004). Suas vantagens incluem garantia de entrega, buffering de mensagens durante picos de demanda e balanceamento de carga entre múltiplos consumidores. A complexidade adicional de operação e monitoramento das filas constitui sua principal desvantagem (Hohpe; Woolf, 2004).

2.4 Plataforma Sapparchi

A plataforma Sapparchi foi desenvolvida como resultado de pesquisa de doutorado na UFRN, com o objetivo de fornecer um ambiente integrado para desenvolvimento, implantação e execução de aplicações em contextos de cidades inteligentes (Souza, 2022). A plataforma adota o paradigma de computação osmótica, permitindo que serviços migrem entre as camadas Edge, Fog e Cloud de acordo com requisitos de qualidade de serviço (Souza et al., 2022).

2.4.1 Modelo de Programação

O modelo de programação do Sapparchi é baseado em processamento de fluxo de dados, onde *Actions* representam funções serverless e *Tasks* representam composições de múltiplas Actions executadas sequencialmente. Esse modelo é similar ao adotado por plataformas como OpenWhisk, ThingWorx e Kosmos (Souza et al., 2022).

As Actions constituem as menores unidades de processamento da plataforma. Cada Action é caracterizada por execução stateless, sem persistência direta de entidades, e identificada por uma URN no formato `urn:sapparchi:Action:nome.da.action`. A comunicação entre Actions ocorre através de Messages contendo MessageData em formato JSON. As Tasks, por sua vez, são fluxos direcionados de processamento que encadeiam múltiplas Actions, permitindo a composição de serviços complexos a partir de funções atômicas.

2.4.2 Arquitetura Original

A arquitetura original do Sapparchi compreende os seguintes componentes principais (Souza, 2022): o Manager Service, responsável por orquestrar a execução de Actions e Tasks, gerenciando o registro de nós e o roteamento de requisições; o Monitor Service, que coleta métricas de execução dos nós registrados; os Executors, responsáveis pela execução de Actions nos Worker Nodes; os Data Services, que fornecem persistência distribuída entre Fog e Cloud; e o API Gateway, que recebe requisições HTTP e cria identificadores únicos para rastreamento.

Na versão original, a comunicação entre Manager e Workers era predominantemente síncrona, baseada em HTTP REST. Essa abordagem, embora simples, apresenta limitações em cenários com alta latência, conectividade intermitente ou grande volume de requisições concorrentes.

2.4.3 Limitações Identificadas

A análise da plataforma original revelou oportunidades de melhoria, especialmente no que diz respeito à comunicação com dispositivos de borda. O protocolo HTTP apresenta cabeçalhos extensos, inadequados para dispositivos com recursos limitados como ESP32. Além disso, a comunicação bloqueante exige que produtor e consumidor estejam ativos simultaneamente, criando um acoplamento síncrono que dificulta a operação em cenários de conectividade intermitente. A ausência de filas para buffering de mensagens implica que falhas transitórias podem resultar em perda de requisições. Por fim, após falhas do Manager, os Workers não se re-registravam automaticamente, exigindo intervenção manual para restaurar o funcionamento do sistema.

Este trabalho propõe a integração de mensageria assíncrona à plataforma Sapparchi, endereçando essas limitações através do uso de RabbitMQ como broker central, suportando os protocolos AMQP e MQTT.

2.5 Trabalhos Relacionados

Diversas iniciativas na literatura abordam aspectos de computação distribuída em ambientes Edge-Fog-Cloud. Esta seção analisa trabalhos representativos, organizados em três categorias: ferramentas de simulação, frameworks de borda e arquiteturas de referência. Para cada trabalho, discute-se o escopo, as contribuições e as limitações em relação aos requisitos de comunicação assíncrona multiprotocolo e roteamento adaptativo por camada, que constituem o foco deste trabalho.

2.5.1 Ferramentas de Simulação

O **iFogSim** (Gupta et al., 2017) é um simulador para ambientes de Fog Computing construído sobre o CloudSim, permitindo modelar políticas de alocação de recursos, posicionamento de módulos e avaliar métricas como latência e consumo energético. A ferramenta possibilita a experimentação com diferentes topologias e cargas de trabalho em ambiente controlado, sendo amplamente utilizada pela comunidade acadêmica para validação de algoritmos de escalonamento.

Contudo, o iFogSim opera exclusivamente em contexto de simulação, não oferecendo implementação real de protocolos de comunicação. Os módulos simulados comunicam-se através de abstrações que não refletem os desafios de sistemas em produção, como falhas de rede, variações de latência e heterogeneidade de dispositivos. Além disso, não há suporte a múltiplos protocolos de mensageria nem mecanismos de registro dinâmico de nós, aspectos fundamentais para ambientes IoT reais.

2.5.2 Frameworks de Borda

O **EdgeX Foundry** (Linux Foundation, 2024) é um framework de código aberto, mantido pela Linux Foundation, voltado para Edge Computing. Fornece uma arquitetura modular com serviços para coleta de dados de sensores, transformação e exportação para sistemas externos. O framework inclui suporte a comunicação baseada em mensageria através de barramentos internos e conectores para protocolos como MQTT e REST.

Embora robusto para a camada de borda, o EdgeX Foundry não oferece integração explícita com as camadas Fog e Cloud em um modelo de execução contínuo. Sua arquitetura assume que os dados serão exportados para plataformas externas, sem prover mecanismos nativos de roteamento adaptativo ou orquestração de tarefas entre camadas. A comunicação

multiprotocolo existe, porém restrita ao contexto de ingestão de dados na borda, não contemplando o fluxo bidirecional necessário para coordenação de tarefas distribuídas.

O **FogFlow** (Cheng et al., 2018) apresenta um modelo de programação baseado em padrões para serviços IoT em Smart Cities, executando sobre Cloud e Edge. Similar ao Sapparchi, permite dividir aplicações em tarefas executadas em containers Docker, oferecendo abstrações para processamento de dados contextual. O framework implementa descoberta de serviços e orquestração baseada em intenções, facilitando o desenvolvimento de aplicações distribuídas.

Diferentemente do Sapparchi, o FogFlow não suporta migração osmótica de tarefas entre camadas. O escalonamento é determinado no momento da implantação, sem reavaliação dinâmica baseada em condições de carga ou disponibilidade. Ademais, a comunicação entre componentes utiliza predominantemente REST síncrono, não oferecendo as garantias de entrega e desacoplamento temporal proporcionadas por filas de mensagens persistentes.

2.5.3 Arquiteturas de Referência

A **OpenFog Reference Architecture** (OpenFog Consortium, 2017), desenvolvida pelo OpenFog Consortium (atualmente incorporado ao Industrial Internet Consortium), define princípios fundamentais para sistemas de Fog Computing. O documento estabelece requisitos de segurança, escalabilidade, autonomia e hierarquia que orientam o projeto de arquiteturas distribuídas. As diretrizes conceituais da OpenFog influenciaram diversos trabalhos subsequentes, incluindo aspectos do próprio Sapparchi.

Entretanto, a OpenFog Reference Architecture permanece no nível conceitual, sem fornecer implementação concreta de mecanismos de comunicação ou roteamento. As especificações definem o que sistemas de Fog Computing devem oferecer, mas não como implementá-los. Questões práticas como tradução entre protocolos, registro de nós heterogêneos e tratamento de falhas de comunicação não são abordadas em nível de código.

2.5.4 Análise Comparativa

A Tabela 1 sintetiza as características dos trabalhos relacionados em comparação com a solução proposta neste trabalho.

A análise evidencia que os trabalhos existentes atendem parcialmente aos requisitos de sistemas Edge-Fog-Cloud. Ferramentas de simulação como o iFogSim permitem experimentação, mas não geram artefatos executáveis em produção. Frameworks de borda como EdgeX Foundry e FogFlow oferecem implementações reais, porém com foco em camadas específicas ou sem suporte completo a comunicação assíncrona multiprotocolo. Arquiteturas de referência como a OpenFog fornecem diretrizes conceituais valiosas, mas carecem de implementação.

Tabela 1 – Comparativo entre trabalhos relacionados e a solução proposta

Característica	iFogSim	EdgeX	FogFlow	OpenFog	Este trabalho
Ambiente de produção	–	✓	✓	–	✓
Comunicação assíncrona	–	Parcial	–	–	✓
Suporte AMQP	–	–	–	–	✓
Suporte MQTT	–	✓	–	–	✓
Suporte HTTP	–	✓	✓	–	✓
Integração Edge-Fog-Cloud	Sim*	–	Parcial	Sim*	✓
Roteamento por camada	–	–	–	–	✓
Registro dinâmico de nós	–	Parcial	✓	–	✓
Migração osmótica	–	–	–	–	✓

*Apenas conceitual ou em simulação

2.5.5 Contribuição deste Trabalho

O presente trabalho diferencia-se dos anteriores ao contribuir para a plataforma Sapparchi com uma solução integrada de comunicação assíncrona que atende simultaneamente aos seguintes requisitos:

1. **Multiprotocolo**: suporte nativo a AMQP (via RabbitMQ), MQTT (para dispositivos IoT restritos) e HTTP (para integração com sistemas legados), com tradução transparente entre protocolos;

2. **Roteamento adaptativo por camada**: mensagens são direcionadas aos nós apropriados com base na camada de execução (Edge, Fog ou Cloud) e na disponibilidade dos recursos, sem intervenção manual;

3. **Registro dinâmico**: nós podem ingressar ou deixar o sistema em tempo de execução, com propagação automática de catálogos de ações disponíveis;

4. **Resiliência**: mecanismos de heartbeat, reenvio de mensagens e tratamento de falhas garantem a continuidade das operações mesmo em cenários de instabilidade de rede;

5. **Compatibilidade com migração osmótica**: a comunicação desacoplada via filas de mensagens viabiliza a redistribuição de tarefas entre camadas conforme as condições do ambiente, funcionalidade central do paradigma osmótico do Sapparchi.

Dessa forma, enquanto os trabalhos relacionados contribuem com aspectos específicos — simulação, processamento na borda ou diretrizes conceituais —, este trabalho oferece uma implementação concreta e operacional que trata Edge, Fog e Cloud como um continuum de recursos distribuídos, permitindo a execução de aplicações de forma resiliente, desacoplada e adaptativa.

3 ARQUITETURA DO SISTEMA

3.1 Visão Geral

A arquitetura proposta neste trabalho estende a plataforma Sapparchi original (Souza, 2022) com a integração de comunicação assíncrona baseada em mensageria. A estrutura de componentes principais foi mantida, com modificações significativas na camada de comunicação e adição de novos mecanismos de monitoramento e recuperação.

O sistema é composto por três componentes principais. O Sapparchi Manager atua como orquestrador central, implementado em Java com Spring Boot 3.1.2[1] sendo responsável pelo gerenciamento de Workers, roteamento de mensagens e consolidação de resultados. O Sapparchi Worker Python consiste em nós de processamento destinados às camadas Fog e Cloud, com suporte aos protocolos AMQP e HTTP. O Worker Node ESP32[2], desenvolvido como contribuição deste trabalho, constitui nós de processamento para a camada Edge, com suporte a MQTT e HTTP, otimizado para dispositivos com recursos limitados.

A principal diferença em relação à arquitetura original está na camada de comunicação: enquanto a versão original utilizava exclusivamente HTTP síncrono, a versão proposta integra RabbitMQ[3] como broker central, suportando AMQP para nós Fog/Cloud e MQTT para nós Edge.

[1] Spring Boot: framework para desenvolvimento de aplicações em Java baseado no ecossistema Spring. Disponível em: `https://spring.io/projects/spring-boot`.

[2] ESP32: microcontrolador da Espressif Systems com conectividade Wi-Fi e Bluetooth integrado. Informações em: `https://www.espressif.com/en/products/socs/esp32`.

[3] RabbitMQ: broker de mensagens de código aberto baseado no protocolo AMQP. Disponível em: `https://www.rabbitmq.com`.

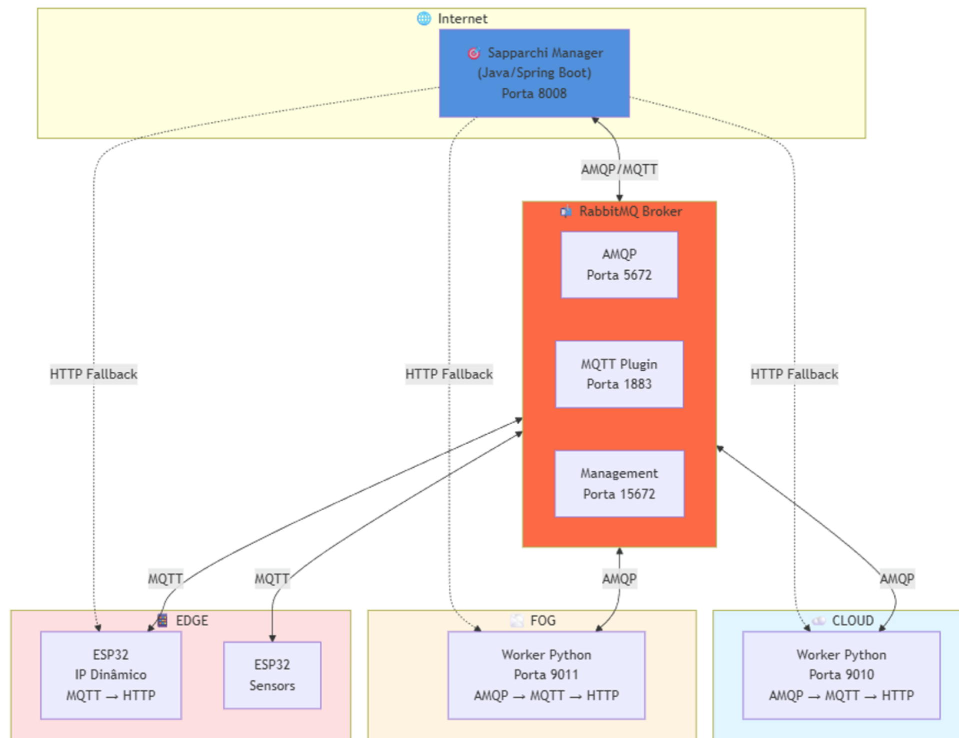

Figura 3 – Arquitetura geral da plataforma Sapparchi.

3.2 Componentes do Sistema

3.2.1 Sapparchi Manager

O Sapparchi Manager é o componente central de orquestração, responsável por coordenar a execução de Actions e Tasks em todo o sistema distribuído. Na versão proposta, foram adicionadas funcionalidades de roteamento inteligente por camada e monitoramento de saúde de nós.

Entre suas responsabilidades estão a orquestração de Tasks e Actions, o registro e gerenciamento de Workers, o roteamento inteligente de requisições considerando a camada de destino, a consolidação de resultados de tasks multi-step e o monitoramento contínuo da saúde dos nós registrados.

O componente foi desenvolvido em Java 21 utilizando Spring Boot 3.1.2 como framework principal. A integração com mensageria é realizada através do Spring Cloud Stream[4], que abstrai a comunicação com o RabbitMQ. O Redis[5] é utilizado como cache para configurações e estado dos nós, enquanto o RabbitMQ atua como broker central para comunicação assíncrona.

[4]Spring Cloud Stream: biblioteca do ecossistema Spring para integração com brokers de mensageria como RabbitMQ e Kafka. Disponível em: `https://spring.io/projects/spring-cloud-stream`.
[5]Redis: armazenamento chave-valor em memória utilizado como cache e estrutura de dados. Disponível em: `https://redis.io`.

3.2.2 Sapparchi Worker (Python)

O Worker Python é utilizado nas camadas Fog e Cloud, fazendo uso da maior capacidade computacional e suportando bibliotecas mais robustas, principalmente para processamento de ML. Este componente é responsável pela execução de Actions, pelo registro automático junto ao Manager, pela publicação de resultados intermediários na fila apropriada e pelo suporte a múltiplos protocolos de comunicação.

A implementação utiliza Python 3.10 ou superior, com FastAPI[6] como servidor HTTP para recebimento de requisições síncronas. A comunicação assíncrona via AMQP é realizada através da biblioteca aio-pika[7], que oferece suporte a operações assíncronas compatíveis com o modelo de programação do FastAPI.

3.2.3 Worker Node ESP32

O Worker ESP32 foi desenvolvido como contribuição deste trabalho para permitir a execução de Actions em dispositivos de borda com recursos limitados. Este componente utiliza preferencialmente MQTT para comunicação, otimizando o uso de banda e energia em cenários típicos de IoT.

As responsabilidades deste componente incluem a execução de Actions em dispositivos embarcados, a inferência de modelos de machine learning utilizando TensorFlow Lite Micro[8], a leitura de sensores conectados ao microcontrolador e a comunicação bidirecional via MQTT com o broker RabbitMQ.

A implementação foi realizada em C++ utilizando o Arduino Framework[9], o que facilita a portabilidade para diferentes variantes do ESP32. A comunicação MQTT é gerenciada pela biblioteca PubSubClient[10], enquanto o parsing de mensagens JSON é realizado pela biblioteca cJSON[11]. Para inferência de modelos de machine learning, foi integrado o TensorFlow Lite Micro, permitindo a execução de modelos previamente treinados diretamente no microcontrolador.

---

[6] RAMÍREZ, Sebastián. *FastAPI*. Disponível em: https://fastapi.tiangolo.com/. Acesso em: 8 dez. 2025.

[7] MOSKOVKIN, Pavel et al. *aio-pika: AMQP client library for asyncio*. Disponível em: https://aio-pika.readthedocs.io/. Acesso em: 8 dez. 2025.

[8] TensorFlow Lite for Microcontrollers (TensorFlow Lite Micro): versão do TensorFlow voltada para execução de modelos em microcontroladores. Disponível em: https://www.tensorflow.org/lite/microcontrollers.

[9] Arduino core para ESP32: plataforma de desenvolvimento baseada na API Arduino para microcontroladores ESP32. Documentação em: https://docs.espressif.com/projects/arduino-esp32/en/latest/.

[10] PubSubClient: biblioteca MQTT para plataformas Arduino. Informações em: https://pubsubclient.knolleary.net.

[11] cJSON: biblioteca em C para manipulação de dados JSON. Repositório em: https://github.com/DaveGamble/cJSON.

3.3 Modelo de Dados

O modelo de dados do Sapparchi é composto por três entidades principais: Actions, Tasks e Messages. Este modelo foi definido na plataforma original (Souza, 2022) e mantido neste trabalho para garantir compatibilidade com implementações existentes. As extensões realizadas neste trabalho adicionam parâmetros opcionais que não alteram a estrutura base, preservando a interoperabilidade com workers desenvolvidos anteriormente.

3.3.1 Action

Uma Action representa a unidade atômica de processamento da plataforma, identificada por uma URN (Uniform Resource Name) no formato:

```
urn:sapparchi:Action:actions.fator_potencia
```

Cada Action possui um keyname que a identifica univocamente e um estado de ciclo de vida (life_state), conforme definido na especificação original (Souza, 2022). Neste trabalho, foram adicionados parâmetros opcionais de configuração — `timeout` e `retry` — que permitem ajustar o comportamento de resiliência por Action, sem quebrar a compatibilidade com implementações que não os utilizem. A estrutura JSON de uma Action é exemplificada a seguir:

```
{
    "keyname": "urn:sapparchi:Action:actions.exemplo",
    "life_state": "ACTIVE",
    "parameters": {
        "timeout": 5000,
        "retry": 3
    }
}
```

Os campos `keyname` e `life_state` são obrigatórios e correspondem à especificação original. O objeto `parameters` é opcional e, quando omitido, o sistema assume valores padrão (timeout de 30 segundos e 3 retentativas), garantindo compatibilidade retroativa.

3.3.2 Task

Uma Task representa um conjunto ordenado de Actions executadas sequencialmente, permitindo a composição de fluxos de processamento mais complexos. Assim como as Actions, as Tasks são identificadas por URNs:

```
urn:sapparchi:Task:inferencia_fator_potencia
```

A estrutura de uma Task inclui um identificador único de requisição (id), o identificador da Task (task_id), a lista ordenada de Actions a serem executadas (target_actions) e os dados de entrada (message_data), todos definidos na especificação original (Souza, 2022). Neste trabalho, foram incorporados metadados de temporização (`created_at` e `processing_time`) para suportar métricas de observabilidade no sistema de mensageria. Um exemplo de estrutura JSON é apresentado a seguir:

```
{
    "id": "streq-1234567890",
    "task_id": "urn:sapparchi:Task:exemplo",
    "target_actions": [
        "urn:sapparchi:Action:actions.fetch_data",
        "urn:sapparchi:Action:actions.fator_potencia"
    ],
    "message_data": "{\"url\": \"http://sensor/data\"}",
    "created_at": 1700000000000,
    "processing_time": 0
}
```

A Tabela 2 resume a origem de cada campo da estrutura Task.

Tabela 2 – Campos da estrutura Task: origem e obrigatoriedade

Campo	Descrição	Origem	Obrigatório
id	Identificador único da requisição	Original	Sim
task_id	URN da Task	Original	Sim
target_actions	Lista ordenada de Actions	Original	Sim
message_data	Dados de entrada serializados	Original	Sim
created_at	Timestamp de criação	Este trabalho	Não
processing_time	Tempo de processamento (ms)	Este trabalho	Não

3.3.3 Message

Uma Message representa a unidade de comunicação entre Manager e Workers, encapsulando a identificação da Action de destino, o identificador da requisição e os dados a serem processados. Esta estrutura permanece inalterada em relação à especificação original (Souza, 2022):

```
{
    "actionTo": "urn:sapparchi:Action:actions.exemplo",
    "requestID": "sreq.1700000000.123456",
    "messageData": "{\"input\": \"dados\"}"
}
```

A manutenção da estrutura de Message sem alterações foi uma decisão de projeto deliberada, uma vez que esta entidade é processada diretamente pelos workers. Qualquer modificação exigiria atualização de todos os workers existentes, comprometendo a compatibilidade retroativa. As extensões de resiliência e temporização são tratadas nas camadas de Action e Task, transparentes para os workers.

4 CONTRIBUIÇÕES DESENVOLVIDAS

Este capítulo detalha as contribuições desenvolvidas para a plataforma Sapparchi, organizadas em cinco eixos principais: integração de comunicação assíncrona via RabbitMQ, roteamento inteligente por camadas, sistema de monitoramento de saúde de nós, mecanismos de registro e re-registro automático, e separação de filas de resultados intermediários e finais.

As implementações descritas a seguir foram desenvolvidas em três componentes: Manager (Java/Spring Boot), Worker Python (aio-pika/FastAPI) e Worker ESP32 (C++/PubSubClient). Para cada contribuição, são apresentados os trechos de código mais relevantes com explicações detalhadas das decisões de implementação.

4.1 Integração de Comunicação Assíncrona

A solução implementada utiliza RabbitMQ como broker central de mensagens, suportando dois protocolos principais para atender aos requisitos heterogêneos das diferentes camadas da arquitetura.

O protocolo AMQP (Advanced Message Queuing Protocol) é utilizado para comunicação com Workers Python e o Manager Java, operando na porta 5672. Trata-se de um protocolo binário eficiente que oferece garantias de entrega, ordenação de mensagens e suporte a transações, características essenciais para as camadas Fog e Cloud onde a confiabilidade é prioritária.

O protocolo MQTT (Message Queuing Telemetry Transport) é empregado para comunicação com Workers ESP32 na camada Edge, operando na porta 1883. Este protocolo foi projetado especificamente para dispositivos com recursos limitados, apresentando overhead reduzido de apenas 2 bytes por mensagem. Essa característica o torna ideal para microcontroladores com memória restrita e conexões de rede intermitentes, comuns em ambientes IoT.

A Figura 4 ilustra a topologia de comunicação resultante, onde o RabbitMQ atua como ponto central de integração entre os diferentes componentes. Observa-se que o Manager comunica-se com Workers Python através de AMQP (conexões representadas em azul) e com Workers ESP32 através do plugin MQTT do RabbitMQ (conexões representadas em verde). Essa arquitetura permite que dispositivos heterogêneos participem do mesmo ecossistema de mensageria sem necessidade de adaptadores externos.

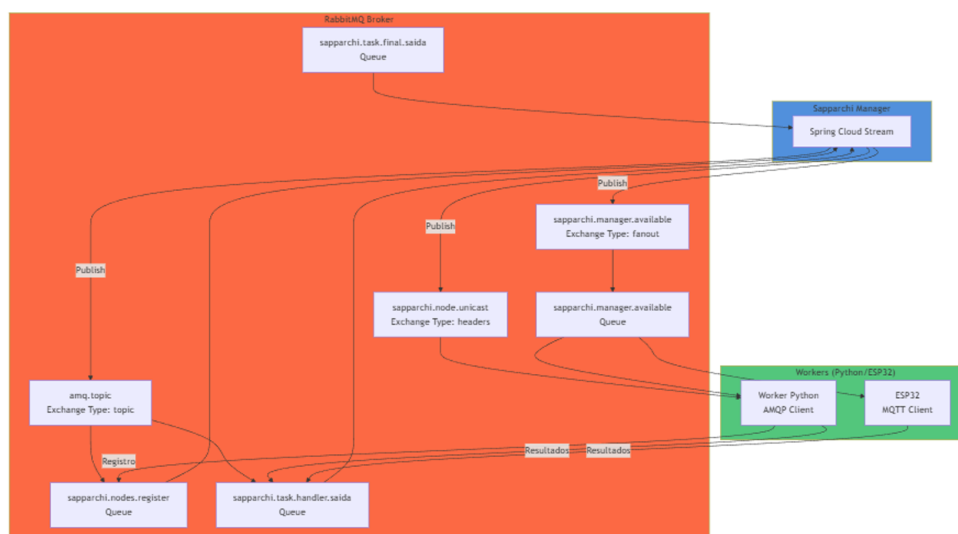

Figura 4 – Topologia de comunicação entre Manager e Workers no Sapparchi. O RabbitMQ atua como broker central, recebendo conexões AMQP de Workers Python/Java e conexões MQTT de Workers ESP32.

4.2 Exchanges e Filas Implementadas

A arquitetura de mensageria utiliza três exchanges principais no RabbitMQ, cada uma com um propósito específico de roteamento.

A exchange `amq.topic`, do tipo *topic*, é a principal via de comunicação da plataforma. Neste tipo de exchange, as mensagens são roteadas com base em padrões de *routing keys*, permitindo que múltiplos consumidores recebam mensagens de acordo com seus interesses. A Tabela 3 apresenta as principais rotas configuradas, indicando o propósito de cada uma e quais componentes as consomem.

Tabela 3 – Rotas configuradas na exchange `amq.topic`. Cada routing key direciona mensagens para consumidores específicos.

Routing Key	Descrição	Consumidores
`sapparchi.nodes.register`	Mensagens de registro de novos nós na plataforma	Manager
`sapparchi.nodes.actions-list`	ACK de confirmação de registro enviado pelo Manager	Workers
`sapparchi.task.handler.saida`	Resultados intermediários de execução de actions	Manager
`sapparchi.manager.available`	Heartbeats periódicos indicando disponibilidade do Manager	Workers

A exchange `sapparchi.manager.available`, do tipo *fanout*, é responsável pelo broadcasting de heartbeats. Diferentemente da exchange *topic*, uma exchange *fanout* envia cópias da mensagem para todas as filas vinculadas, independentemente de *routing keys*. Isso garante que todos os Workers conectados recebam os heartbeats do Manager simultaneamente.

A exchange `sapparchi.node.unicast`, do tipo *headers*, realiza roteamento unicast para Workers específicos. Neste tipo de exchange, o roteamento é baseado em atributos do cabeçalho da mensagem ao invés de *routing keys*. A implementação utiliza o header `nodeId` como critério de seleção, permitindo enviar mensagens diretamente para um Worker específico sem que outros consumidores as recebam.

A Figura 5 apresenta a estrutura de filas resultante dessa configuração. Cada Worker possui uma fila exclusiva para receber mensagens unicast (identificada pelo padrão `sapparchi.node.unicast.<nodeId>`), enquanto filas compartilhadas são utilizadas para broadcast e coleta de resultados.

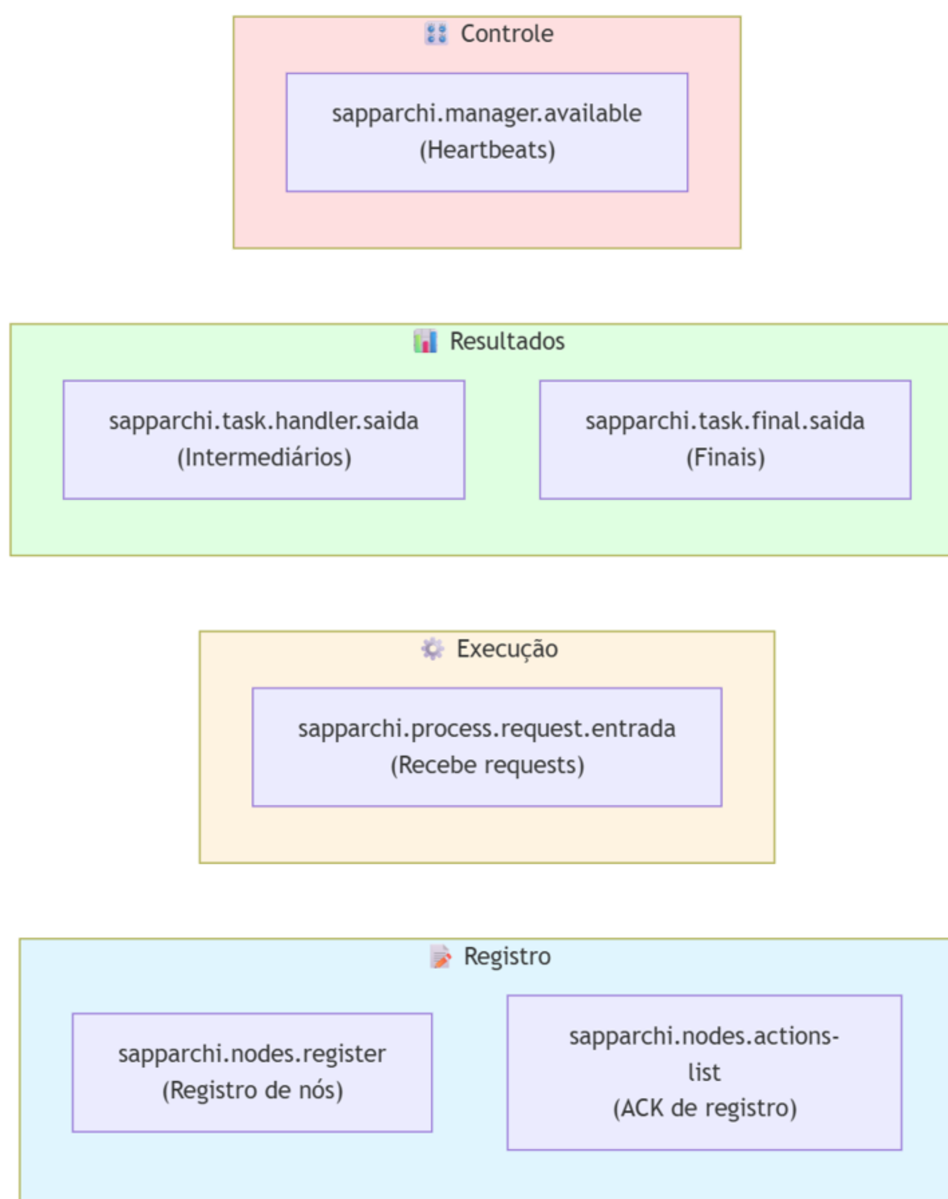

Figura 5 – Filas principais da arquitetura de mensageria. Filas unicast são exclusivas por Worker, enquanto filas de heartbeat e resultados são compartilhadas.

4.2.1 Implementação no Manager (Java)

A configuração do Spring Cloud Stream define os bindings de entrada e saída para comunicação com o broker. O Código 4.1 apresenta os principais bindings configurados no arquivo `application.yml`.

Listing 4.1 – Configuração de bindings do Spring Cloud Stream para comunicação com RabbitMQ.

```
spring.cloud.stream.bindings:
  # Input: Recebe requests de tasks
  processAndSendMessage-in-0:
    destination: sapparchi.process.request.entrada

  # Input: Recebe resultados intermediarios dos Workers
  getTaskMessages-in-0:
    destination: sapparchi.task.handler.saida
    group: sapparchi.manager

  # Output: Publica resultado final consolidado
  taskFinalResponse-out-0:
    destination: sapparchi.task.final.saida

  # Output: Heartbeats para Workers
  sendAvailableMessage-out-0:
    destination: sapparchi.manager.available
    producer.exchangeType: fanout
```

Na configuração apresentada, as linhas 3–4 definem o binding de entrada para recepção de requisições de tasks, onde `destination` especifica o nome da fila RabbitMQ. As linhas 7–9 configuram o consumo de resultados intermediários, com o parâmetro `group` definindo um grupo de consumidores para balanceamento de carga quando múltiplas instâncias do Manager estiverem ativas. As linhas 12–13 definem a fila de saída para resultados finais, enquanto as linhas 16–18 configuram o canal de heartbeats com `exchangeType: fanout` para garantir broadcasting a todos os Workers.

O processamento de mensagens é implementado através de beans funcionais do Spring, conforme ilustrado no Código 4.2. Este padrão permite que o framework gerencie automaticamente a conexão com o broker e a serialização de mensagens.

Listing 4.2 – Handlers para processamento de tasks e resultados no Manager.

```
@Bean
public Consumer<TaskRequest> taskHandler() {
    return request -> {
        for (String action : request.getTarget_actions()) {
```

```
5             Message msg = buildMessage(action, request);
6             String nodeId = findNodeForAction(action);
7             eventGateway.sendToWorker(nodeId, msg.toString());
8         }
9     };
10 }
11
12 @Bean
13 public Consumer<String> getTaskMessages() {
14     return resultJson -> {
15         Message result = Message.fromString(resultJson);
16         InFlightTaskState state = inFlightByMessageId
17             .remove(result.getRequestID());
18
19         if (state.isLastStep()) {
20             eventGateway.sendTaskFinalResponse(state.consolidate
                   ());
21         } else {
22             dispatchNextAction(state);
23         }
24     };
25 }
```

O método `taskHandler` (linhas 1–10) é invocado automaticamente quando uma nova requisição de task chega na fila de entrada. O loop na linha 4 itera sobre as actions definidas na task, construindo uma mensagem para cada uma (linha 5), identificando o Worker apropriado através do método `findNodeForAction` (linha 6) e despachando a mensagem via gateway AMQP (linha 7).

O método `getTaskMessages` (linhas 12–24) processa os resultados intermediários publicados pelos Workers. A linha 15 deserializa o JSON recebido em um objeto `Message`, enquanto as linhas 16–17 recuperam e removem atomicamente o estado da task do mapa de tasks em andamento usando o `requestID` como chave. A verificação na linha 19 determina se todos os steps foram concluídos: em caso positivo, o resultado é consolidado e publicado na fila final (linha 20); caso contrário, o próximo step é despachado (linha 22).

4.2.2 Implementação no Worker Python

O cliente AMQP em Python utiliza a biblioteca `aio-pika` para comunicação assíncrona com o broker. O Código 4.3 apresenta a classe `AmqpClient` com os métodos principais de inicialização, publicação e consumo.

Listing 4.3 – Cliente AMQP assíncrono para Worker Python.

```
import aio_pika
from aio_pika import ExchangeType

class AmqpClient:
    async def init(self, host, port, username, password):
        url = f"amqp://{username}:{password}@{host}:{port}/"
        self._connection = await aio_pika.connect_robust(url)
        self._channel = await self._connection.channel()
        await self._channel.set_qos(prefetch_count=16)

    async def publish_topic(self, routing_key, body, exchange="
       amq.topic"):
        ex = await self.ch.declare_exchange(
            exchange, ExchangeType.TOPIC, durable=True
        )
        msg = aio_pika.Message(
            body.encode("utf-8"),
            delivery_mode=aio_pika.DeliveryMode.PERSISTENT,
        )
        await ex.publish(msg, routing_key=routing_key)

    async def consume_unicast_for(self, node_id, callback):
        exch = await self.ch.declare_exchange(
            "sapparchi.node.unicast", ExchangeType.HEADERS,
               durable=True
        )
        queue = await self.ch.declare_queue(
            f"sapparchi.node.unicast.{node_id}.python", durable=
               True
        )
        await queue.bind(exch, routing_key="",
                         arguments={"x-match": "all", "nodeId":
                            node_id})
        await queue.consume(callback, no_ack=False)
```

O método `init` (linhas 5–9) estabelece a conexão com o broker. A linha 7 utiliza `connect_robust`, que implementa reconexão automática em caso de falhas de rede — funcionalidade essencial para ambientes distribuídos instáveis. A linha 9 configura o `prefetch_count` para 16, limitando o número de mensagens não confirmadas que o Worker pode receber simultaneamente, evitando sobrecarga de memória em picos de

demanda.

O método `publish_topic` (linhas 11–19) publica mensagens na exchange `amq.topic`. As linhas 12–14 declaram a exchange como durável (`durable=True`), garantindo que ela sobreviva a reinícios do broker. A linha 17 define `delivery_mode=PERSISTENT`, instruindo o RabbitMQ a persistir a mensagem em disco antes de confirmar o recebimento, prevenindo perda de dados em caso de falha do broker.

O método `consume_unicast_for` (linhas 21–30) configura o consumo de mensagens unicast. As linhas 22–24 declaram a exchange do tipo `HEADERS`, que roteia mensagens com base em atributos do cabeçalho. As linhas 25–27 criam uma fila exclusiva para o Worker, nomeada com o padrão que inclui o `node_id`. As linhas 28–29 vinculam a fila à exchange com o argumento `x-match: all` e `nodeId`, garantindo que apenas mensagens destinadas a este Worker específico sejam entregues. A linha 30 inicia o consumo com `no_ack=False`, exigindo confirmação explícita de processamento.

A publicação de resultados segue o padrão apresentado no Código 4.4, onde todos os resultados são direcionados para a fila intermediária, independentemente de qualquer `reply_to` presente na mensagem original.

Listing 4.4 – Publicação de resultados na fila intermediária.

```
1  RESULT_QUEUE = "sapparchi.task.handler.saida"
2
3  async def _publish_result_message(out, *, reply_to=None,
       correl_id=None):
4      """Publica resultado na fila intermediaria.
5      O Manager consolida e publica o resultado final."""
6      body = str(out)
7      amqp = AmqpClient.instance()
8
9      # Ignora reply_to, usa SEMPRE a fila padrao
10     target_q = RESULT_QUEUE
11
12     await amqp.publish_topic(
13         routing_key=RESULT_QUEUE,
14         body=body,
15         exchange="amq.topic"
16     )
17     log.info("Resultado publicado rk=%s bytes=%d", RESULT_QUEUE,
           len(body))
```

A constante `RESULT_QUEUE` na linha 1 define o nome da fila intermediária. O método `_publish_result_message` (linhas 3–17) recebe o resultado da execução e parâmetros

opcionais. A linha 6 serializa o objeto para string. Crucialmente, a linha 10 ignora o parâmetro `reply_to`, garantindo que todos os resultados sejam enviados para a mesma fila onde o Manager os aguarda. As linhas 12–16 publicam na exchange `amq.topic` com a routing key apropriada.

4.2.3 Implementação no Worker ESP32

O Worker ESP32 opera com restrições significativas de memória (520KB de RAM) e processamento, exigindo implementações otimizadas. O Código 4.5 apresenta as definições de tópicos MQTT, suportando dois formatos de separador para compatibilidade com diferentes configurações de broker.

Listing 4.5 – Definição de tópicos MQTT para Worker ESP32.

```
#pragma once

// Heartbeats do Manager (dois formatos para compatibilidade)
#define SAP_MANAGER_AVAILABLE_DOT   "sapparchi.manager.available"
#define SAP_MANAGER_AVAILABLE_SLASH "sapparchi/manager/available"

// Registro de nos
#define SAP_NODES_REGISTER_DOT   "sapparchi.nodes.register"

// Resultados de execucao
#define SAP_RESULT_OUT_DOT   "sapparchi.task.handler.saida"
```

As definições com sufixo `_DOT` (linhas 4, 8, 11) utilizam ponto como separador, compatível com o formato AMQP padrão do RabbitMQ. As definições com sufixo `_SLASH` (linha 5) utilizam barra, formato nativo do MQTT. O Worker subscreve ambos os formatos para garantir recebimento de mensagens independentemente da configuração do plugin MQTT do broker.

O Código 4.6 apresenta a inicialização do cliente MQTT com configurações otimizadas para o ESP32.

Listing 4.6 – Inicialização do cliente MQTT no ESP32.

```
#include <PubSubClient.h>

static WiFiClient net_client;
static PubSubClient mqtt_client(net_client);

void mqtt_init() {
```

```
7       mqtt_client.setBufferSize(2048);  // Buffer maior para
            mensagens
8       mqtt_client.setServer(MQTT_HOST, MQTT_PORT);
9       mqtt_client.setCallback(mqtt_on_message);
10      mqtt_connect_internal();
11  }
```

A linha 7 configura o buffer para 2048 bytes, necessário para mensagens JSON maiores que o padrão de 256 bytes da biblioteca. A linha 8 define o endereço do broker, enquanto a linha 9 registra a função de callback que será invocada para cada mensagem recebida. A linha 10 estabelece a conexão inicial.

O Código 4.7 apresenta a função de publicação de resultados, implementando retry automático com reconexão em caso de falha.

Listing 4.7 – Publicação de resultados com retry automático no ESP32.

```
1   bool mqtt_publish_result(const char* request_id,
2                            const char* action_to,
3                            const char* message_json) {
4       if (!request_id || !action_to) return false;
5
6       cJSON* root = cJSON_CreateObject();
7       cJSON_AddStringToObject(root, "requestID", request_id);
8       cJSON_AddStringToObject(root, "actionTo", action_to);
9       cJSON_AddStringToObject(root, "messageData",
10                              message_json ? message_json : "{}");
11
12      char* payload = cJSON_PrintUnformatted(root);
13      cJSON_Delete(root);
14      if (!payload) return false;
15
16      bool ok = mqtt_client.publish(SAP_RESULT_OUT_DOT, payload);
17
18      if (!ok) {
19          Serial.println("[MQTT] Falha. Reconectando...");
20          mqtt_client.disconnect();
21          delay(50);
22          if (mqtt_connect_internal()) {
23              ok = mqtt_client.publish(SAP_RESULT_OUT_DOT, payload)
                    ;
24          }
25      }
```

```
26
27     free(payload);
28     return ok;
29 }
```

A linha 4 valida os parâmetros obrigatórios. As linhas 6–10 constroem o objeto JSON utilizando a biblioteca `cJSON`, escolhida por seu baixo consumo de memória. A linha 12 serializa o objeto para string sem formatação (mais compacto), e a linha 13 libera a estrutura JSON — padrão importante em C para evitar vazamentos de memória. A linha 14 trata o caso de falha na serialização.

A linha 16 tenta a primeira publicação no tópico de resultados. O bloco condicional das linhas 18–25 implementa o mecanismo de retry: em caso de falha, desconecta (linha 20), aguarda 50ms (linha 21) para estabilização da rede, reconecta (linha 22) e tenta novamente (linha 23). A linha 27 libera a memória do payload serializado antes de retornar.

4.2.4 Separação de Filas: Intermediárias vs Finais

Uma das principais contribuições implementadas foi a separação entre filas de resultados intermediários e finais, conforme ilustrado na Figura 6. Na arquitetura original, Workers publicavam resultados diretamente na fila final, causando inconsistências em tasks multi-step onde resultados parciais chegavam ao cliente antes da conclusão.

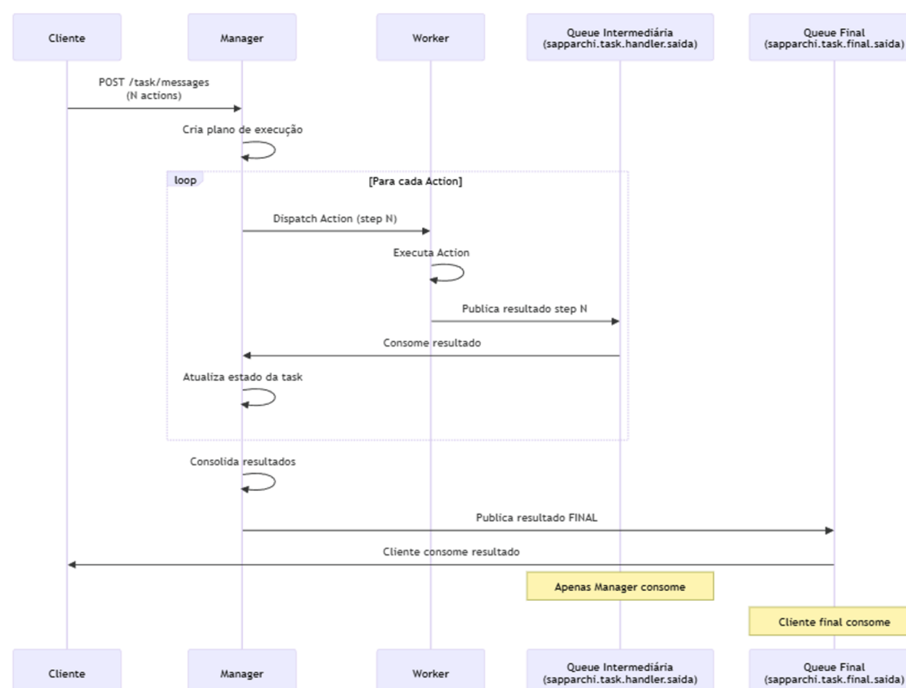

Figura 6 – Arquitetura de filas separadas. Workers publicam na fila intermediária (1), o Manager consolida (2) e publica o resultado final (3).

O diagrama ilustra o fluxo de dados em três etapas numeradas. Na etapa (1), Workers de diferentes camadas (Edge, Fog, Cloud) publicam resultados de steps individuais na fila `sapparchi.task.handler.saida`. Na etapa (2), o Manager consome essas mensagens, atualiza o estado interno da task e verifica se todos os steps foram concluídos. Na etapa (3), apenas quando a task está completa, o Manager publica o resultado consolidado na fila `sapparchi.task.final.saida`, de onde o cliente o consome.

Os benefícios dessa separação incluem: (i) consistência, pois o cliente recebe apenas resultados completos; (ii) rastreabilidade, com cada step individual disponível para debugging; (iii) resiliência, permitindo retry de steps específicos sem afetar os demais; e (iv) escalabilidade, possibilitando múltiplas instâncias do Manager consumindo a fila intermediária em paralelo.

4.3 Roteamento Inteligente por Camadas

O roteamento adaptativo constitui uma das principais inovações da implementação, selecionando automaticamente o protocolo de comunicação mais adequado com base na camada do Worker de destino.

4.3.1 Algoritmo de Seleção de Protocolo

A estratégia de seleção considera as características de cada camada. Para nós das camadas Cloud e Fog, a ordem de tentativa prioriza AMQP, seguido de MQTT e HTTP como fallback. Para nós da camada Edge, a prioridade é MQTT seguido de HTTP, omitindo AMQP devido às limitações de recursos desses dispositivos.

A Tabela 4 apresenta a justificativa técnica para cada escolha, considerando as características típicas dos dispositivos em cada camada.

Tabela 4 – Prioridade de protocolos por camada e justificativa técnica.

Camada	Protocolo	Justificativa
CLOUD	AMQP	Servidores com recursos abundantes, suporte a transações e filas duráveis.
FOG	AMQP	Gateways intermediários com capacidade para cliente AMQP completo.
EDGE	MQTT	Microcontroladores com memória limitada (<520KB RAM no ESP32).

4.3.2 Implementação: ProtocolResolver

O Código 4.8 apresenta o componente `ProtocolResolver`, responsável por determinar a ordem de protocolos a serem tentados para cada camada.

Listing 4.8 – Componente ProtocolResolver para seleção de protocolo por camada.

```
@Component
public class ProtocolResolver {

    public List<ProtocolType> orderFor(LayerType layer) {
        return switch (layer) {
            case EDGE -> List.of(
                ProtocolType.MQTT,
                ProtocolType.HTTP
            );
            case CLOUD, FOG -> List.of(
                ProtocolType.AMQP,
                ProtocolType.MQTT,
                ProtocolType.HTTP
            );
        };
    }
}
```

O método `orderFor` (linhas 4–16) utiliza a sintaxe de *switch expression* do Java 14+ para retornar uma lista imutável de protocolos ordenados por prioridade. Para a camada `EDGE` (linhas 6–9), MQTT é prioritário por seu baixo overhead de 2 bytes por mensagem, com HTTP como fallback universal. Para as camadas `CLOUD` e `FOG` (linhas 10–14), AMQP é preferido por suas garantias de entrega e suporte a transações, com MQTT como segunda opção (útil para comunicação com dispositivos Edge) e HTTP como última alternativa.

4.3.3 Transports Polimórficos

A implementação utiliza o padrão Strategy através da interface `NodeTransport`, permitindo que diferentes protocolos sejam utilizados de forma intercambiável. O Código 4.9 apresenta a interface e as três implementações.

Listing 4.9 – Interface NodeTransport e implementações para AMQP, MQTT e HTTP.

```
public interface NodeTransport {
    boolean send(String destination, String payload) throws
       Exception;
    ProtocolType protocol();
}

@Component
```

```
7  public class AmqpNodeTransport implements NodeTransport {
8      private final ManagerNodeEventGatewayStreamBridge bridge;
9  
10     @Override
11     public boolean send(String nodeId, String payload) {
12         return bridge.sendToWorker(nodeId, payload);
13     }
14 
15     @Override
16     public ProtocolType protocol() { return ProtocolType.AMQP; }
17 }
18 
19 @Component
20 public class MqttNodeTransport implements NodeTransport {
21     private final StreamBridge bridge;
22 
23     @Override
24     public boolean send(String target, String payload) {
25         String topic = "sapparchi/node/" + target + "/in";
26         return bridge.send("mqttActionsList-out-0",
27             MessageBuilder.withPayload(payload)
28                 .setHeader("mqtt_topic", topic)
29                 .build()
30         );
31     }
32 
33     @Override
34     public ProtocolType protocol() { return ProtocolType.MQTT; }
35 }
36 
37 @Component
38 public class HttpNodeTransport implements NodeTransport {
39     private final RestTemplate rest;
40 
41     @Override
42     public boolean send(String endpoint, String payload) {
43         var req = RequestEntity.post(endpoint + "/sapparchi/
               messages")
44             .contentType(MediaType.APPLICATION_JSON)
45             .body(payload);
46         try {
```

```
47                rest.exchange(req, String.class);
48                return true;
49            } catch (Exception e) {
50                return false;
51            }
52        }
53
54        @Override
55        public ProtocolType protocol() { return ProtocolType.HTTP; }
56    }
```

A interface `NodeTransport` (linhas 1–4) define o contrato para todos os transportes: o método `send` (linha 2) recebe destino e payload, retornando sucesso ou falha; o método `protocol` (linha 3) identifica o tipo de protocolo implementado.

A classe `AmqpNodeTransport` (linhas 6–17) implementa o transporte via AMQP. O método `send` (linhas 10–13) delega para o `StreamBridge` do Spring Cloud Stream, que utiliza a exchange de headers para roteamento unicast baseado no `nodeId`.

A classe `MqttNodeTransport` (linhas 19–34) implementa o transporte via MQTT. A linha 25 constrói o tópico dinâmico no formato `sapparchi/node/<target>/in`. As linhas 26–30 utilizam o `StreamBridge` com o header `mqtt_topic` para indicar ao plugin MQTT do RabbitMQ o tópico de destino.

A classe `HttpNodeTransport` (linhas 36–54) implementa o fallback HTTP. As linhas 42–45 constroem uma requisição POST para o endpoint do Worker. O bloco try-catch (linhas 46–51) trata exceções de rede, retornando `false` em caso de falha para permitir tentativa com o próximo protocolo.

4.3.4 Dispatcher com Fallback Automático

O Código 4.10 apresenta o dispatcher que itera sobre os protocolos até obter sucesso, implementando fallback automático transparente.

Listing 4.10 – Dispatcher com fallback automático entre protocolos.

```
1  private void dispatchActionToWorker(InFlightTaskState state) {
2      Node node = state.currentNode;
3      Message message = state.currentMessage;
4
5      List<ProtocolType> order = protocolResolver.orderFor(node.
           getLayerType());
6
7      boolean dispatched = false;
```

```
8
9       for (ProtocolType proto : order) {
10          NodeTransport transport = transports.get(proto);
11
12          if (transport == null) {
13              LOG.warning("Transport nao disponivel: " + proto);
14              continue;
15          }
16
17          try {
18              dispatched = transport.send(node.getNodeId(), message
                    .toString());
19
20              if (dispatched) {
21                  LOG.info("Dispatch OK via " + proto + " -> " +
                        node.getNodeId());
22                  nodeHealth.markOk(node);
23                  break;
24              }
25          } catch (Exception ex) {
26              LOG.warning(proto + " falhou: " + ex.getMessage());
27          }
28      }
29
30      if (!dispatched) {
31          LOG.severe("Todos os protocolos falharam para " + node.
                getNodeId());
32          nodeHealth.markFail(node, "dispatch-failed");
33          handleDispatchFailure(state);
34      }
35  }
```

As linhas 2–3 extraem o nó de destino e a mensagem do estado da task. A linha 5 obtém a lista ordenada de protocolos apropriada para a camada do nó. O loop das linhas 9–27 itera sobre cada protocolo na ordem de prioridade: a linha 10 recupera a implementação de transport correspondente; as linhas 12–15 tratam o caso de transport não disponível (configuração incompleta); a linha 18 tenta o envio; em caso de sucesso (linha 20), o nó é marcado como saudável no serviço NodeHealth (linha 22) e o loop é interrompido (linha 23); em caso de exceção (linhas 25–26), o erro é registrado e o próximo protocolo é tentado.

Se nenhum protocolo for bem-sucedido (linhas 29–33), o nó é marcado como

problemático (linha 31) e um tratamento de falha é acionado (linha 32), que pode incluir reescalonamento para outro Worker ou notificação de erro ao cliente.

A Figura 7 ilustra o fluxo de decisão em formato de diagrama, facilitando a compreensão do algoritmo de fallback.

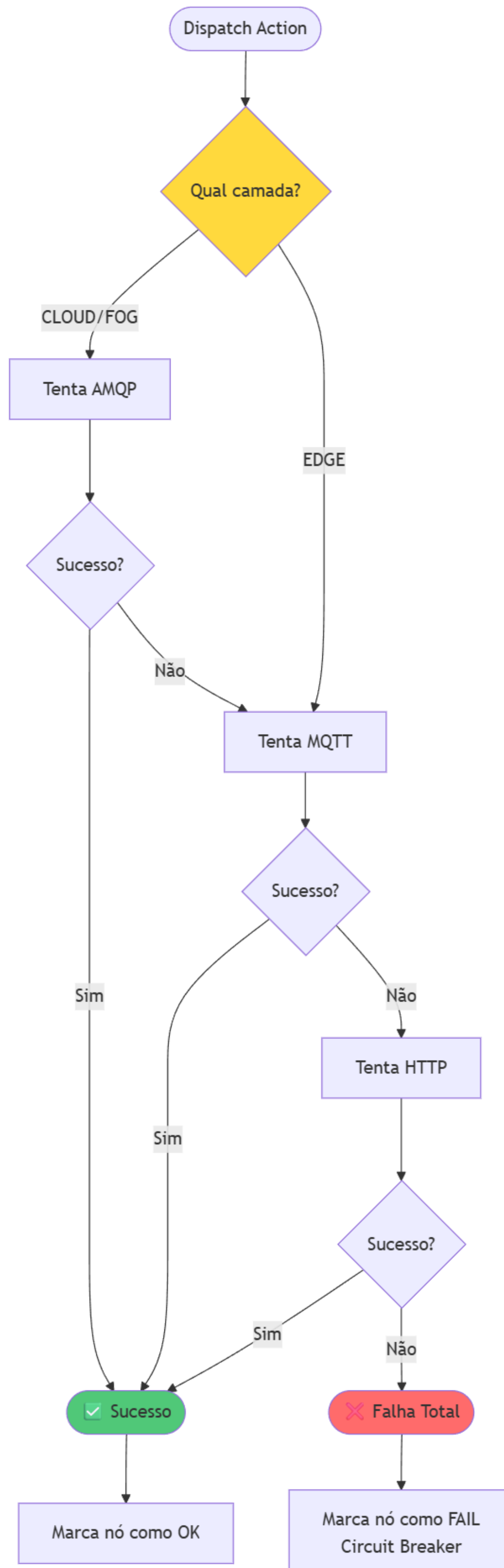

Figura 7 – Fluxo de decisão para escolha de protocolo. A seleção inicial depende da camada; tentativas subsequentes seguem a ordem de prioridade até sucesso ou esgotamento das opções.

O diagrama apresenta um fluxograma que inicia com a identificação da camada do nó de destino. Para cada camada, uma lista ordenada de protocolos é obtida do `ProtocolResolver`. O fluxo então entra em um loop de tentativas: cada protocolo é testado sequencialmente; em caso de sucesso, o fluxo termina com a mensagem entregue e o nó marcado como saudável; em caso de falha, o próximo protocolo da lista é tentado. Se todos os protocolos falharem, o nó é marcado como indisponível no NodeHealth e uma ação de recuperação é iniciada.

4.3.5 Considerações sobre Latência por Protocolo

Embora este trabalho não tenha realizado uma avaliação experimental de métricas de desempenho, é possível discutir de forma qualitativa o impacto dos diferentes protocolos utilizados no Sapparchi.

O AMQP é um protocolo binário mais robusto, com maior overhead de cabeçalho (tipicamente dezenas de bytes), porém oferecendo recursos avançados de roteamento, filas duráveis, transações e garantias de entrega. Essas características o tornam adequado para cenários de Fog e Cloud onde a confiabilidade é prioritária sobre a latência mínima.

O MQTT é um protocolo leve, com cabeçalhos reduzidos (2 bytes fixos mais tópico), suporte a diferentes níveis de QoS e projetado especificamente para dispositivos com recursos limitados e conexões instáveis. Essas características o tornam ideal para a camada Edge.

O HTTP é um protocolo amplamente suportado e simples de integrar, útil como fallback universal. Entretanto, apresenta overhead maior por requisição (cabeçalhos textuais extensos) e ausência nativa de filas e broker, o que pode impactar a latência em cenários de alta carga.

4.4 Sistema de Monitoramento de Saúde (NodeHealth)

Em sistemas distribuídos, nós podem falhar de forma transitória (problemas de rede temporários, sobrecarga momentânea) ou permanente (hardware defeituoso, configuração incorreta). Identificar e evitar nós problemáticos é essencial para manter a qualidade do serviço. O sistema NodeHealth implementa o padrão *circuit breaker*, interrompendo temporariamente o envio de mensagens para nós que apresentam falhas consecutivas.

4.4.1 Arquitetura do NodeHealth

A Figura 8 apresenta a arquitetura do serviço de monitoramento de saúde, ilustrando como o NodeHealth interage com os demais componentes do Manager.

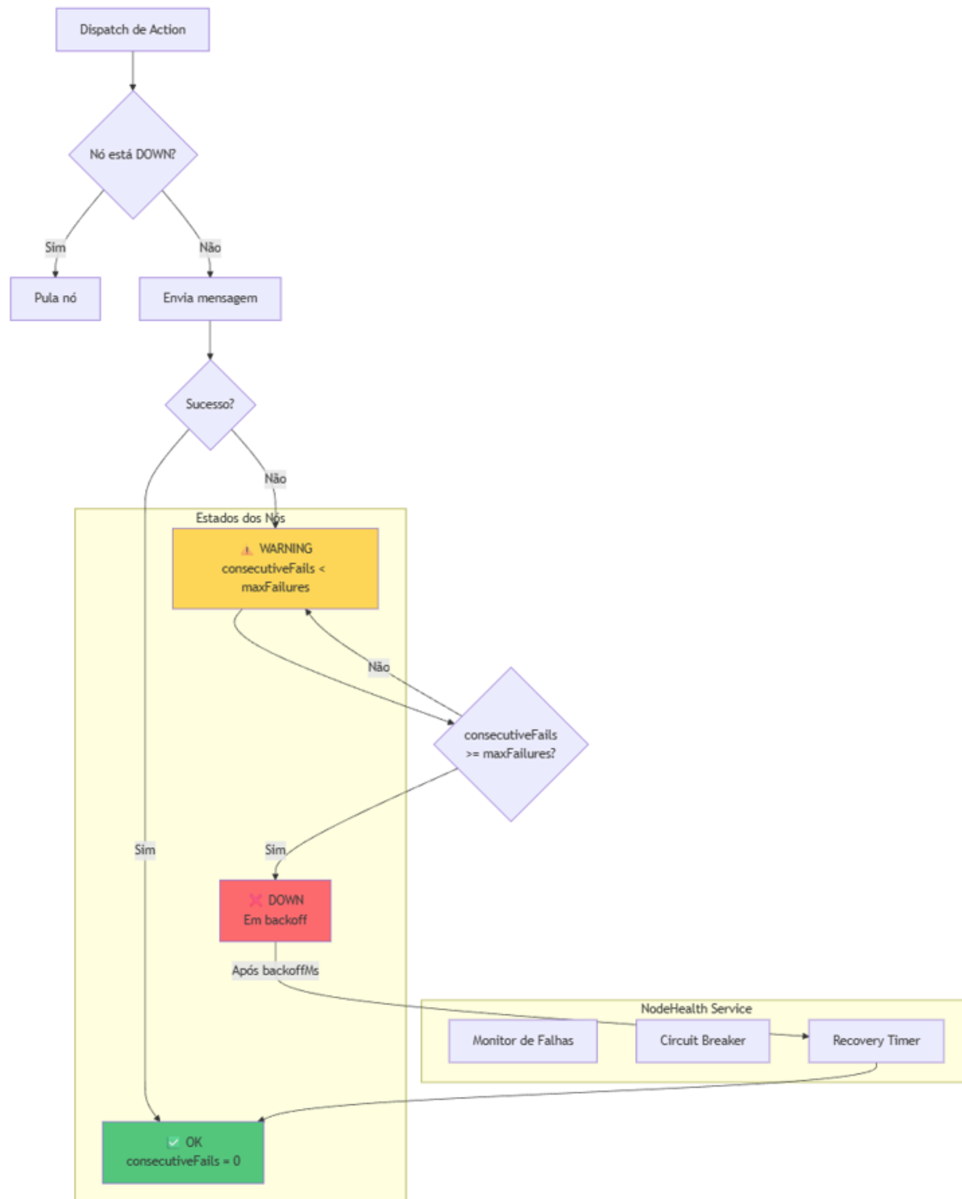

Figura 8 – Arquitetura do serviço NodeHealth. O dispatcher consulta o estado de saúde antes de enviar mensagens; falhas são reportadas de volta para atualizar o estado.

O diagrama mostra três componentes principais interagindo: o Dispatcher, que consulta o NodeHealth antes de cada envio; o NodeHealthService, que mantém um mapa concorrente com o estado de saúde de cada nó; e os Workers, cujas respostas (sucesso ou timeout/erro) são utilizadas para atualizar o estado. As setas numeradas indicam o fluxo: (1) consulta de disponibilidade via `isDown()`, (2) tentativa de envio ao Worker, (3) feedback do resultado (sucesso ou falha), (4) atualização de estado via `markOk()` ou `markFail()`.

4.4.2 Implementação do NodeHealthService

O Código 4.11 apresenta a implementação completa do serviço de monitoramento, incluindo a estrutura de dados interna e os métodos de verificação e atualização de estado.

Listing 4.11 – Serviço NodeHealthService com implementação de circuit breaker.

```
@Service
public class NodeHealthService {

    @Value("${sapparchi.manager.node-max-failures:3}")
    private int maxFailures;

    @Value("${sapparchi.manager.node-backoff-ms:60000}")
    private long backoffMs;

    private final Map<String, Health> byKey = new
       ConcurrentHashMap<>();

    static final class Health {
        volatile int consecutiveFails = 0;
        volatile long downUntilMillis = 0L;
        volatile long lastOkMillis = 0L;
    }

    private String key(Node n) {
        if (n.getNodeId() != null && !n.getNodeId().isBlank())
            return "ID:" + n.getNodeId();
        return "EP:" + String.valueOf(n.getEndpoint());
    }

    public boolean isDown(Node n) {
        Health h = byKey.get(key(n));
        long now = System.currentTimeMillis();
        return h != null && h.downUntilMillis > now;
    }

    public void markOk(Node n) {
        Health h = byKey.computeIfAbsent(key(n), k -> new Health
           ());
        h.consecutiveFails = 0;
        h.downUntilMillis = 0L;
        h.lastOkMillis = System.currentTimeMillis();
```

```
35      }
36
37      public void markFail(Node n, String reason) {
38          Health h = byKey.computeIfAbsent(key(n), k -> new Health
                ());
39          h.consecutiveFails++;
40
41          if (h.consecutiveFails >= maxFailures) {
42              h.downUntilMillis = System.currentTimeMillis() +
                    backoffMs;
43              LOG.warning("[CIRCUIT OPEN] " + key(n) +
44                  " (fails=" + h.consecutiveFails + ", reason=" +
                        reason + ")");
45          } else {
46              LOG.warning("[WARN] Falha " + h.consecutiveFails + "/
                    " +
47                  maxFailures + " para " + key(n));
48          }
49      }
50  }
```

As linhas 4–8 definem os parâmetros configuráveis via `application.properties`: `maxFailures` determina o número de falhas consecutivas necessárias para abrir o circuit breaker (padrão: 3); `backoffMs` define o tempo em milissegundos que o nó permanece indisponível após atingir o limite (padrão: 60 segundos).

A linha 10 declara o mapa concorrente que armazena os estados de saúde, utilizando `ConcurrentHashMap` para garantir thread-safety em ambiente multi-threaded onde múltiplas tasks podem ser processadas simultaneamente. A classe interna `Health` (linhas 12–16) encapsula o estado de cada nó com campos `volatile` para garantir visibilidade entre threads: `consecutiveFails` conta falhas consecutivas, `downUntilMillis` armazena o timestamp até o qual o nó está em backoff, e `lastOkMillis` registra o último sucesso.

O método `key` (linhas 18–22) gera uma chave única para cada nó, preferindo o `nodeId` quando disponível e usando o endpoint como fallback. O prefixo “ID:” ou “EP:” evita colisões entre os dois tipos de identificação.

O método `isDown` (linhas 24–28) verifica se um nó está em estado de backoff. A linha 26 obtém o timestamp atual, e a linha 27 compara com `downUntilMillis`: se o tempo de backoff ainda não expirou, o nó é considerado indisponível e deve ser evitado.

O método `markOk` (linhas 30–35) é chamado após um envio bem-sucedido. A linha 31 utiliza `computeIfAbsent` para obter ou criar o registro de saúde. As linhas 32–34

resetam os contadores, efetivamente fechando o circuit breaker e tornando o nó disponível novamente.

O método `markFail` (linhas 37–49) registra uma falha. A linha 39 incrementa o contador de falhas consecutivas. O bloco condicional das linhas 41–48 verifica se o limiar foi atingido: em caso positivo, a linha 42 calcula o timestamp até o qual o nó permanecerá indisponível, abrindo o circuit breaker; caso contrário, apenas um warning é registrado (linhas 45–47) para monitoramento.

4.4.3 Integração com o Dispatcher

O Código 4.12 demonstra como o dispatcher utiliza o serviço de saúde para selecionar apenas nós disponíveis, iterando até encontrar um candidato saudável.

Listing 4.12 – Seleção de nó saudável com verificação de circuit breaker.

```
1  private Node selectHealthyNode(String actionKey, LayerType layer)
2          throws SapparchiException {
3      Node node = nodesRepository.getNextNodeByLayer(actionKey,
           layer);
4
5      int attempts = 0;
6      final int MAX_ATTEMPTS = 10;
7
8      while (node != null && nodeHealth.isDown(node) && attempts <
           MAX_ATTEMPTS) {
9          LOG.fine("No " + node.getNodeId() + " em backoff, proximo
               ...");
10         node = nodesRepository.getNextNodeByLayer(actionKey,
               layer);
11         attempts++;
12     }
13
14     if (node == null || nodeHealth.isDown(node)) {
15         throw new SapparchiException(
16             "Nenhum no saudavel para " + actionKey + " na camada
                 " + layer
17         );
18     }
19
20     return node;
21 }
```

A linha 3 obtém o primeiro candidato do repositório de nós, que implementa round-robin para distribuição de carga. As linhas 5–6 definem o limite de tentativas para evitar loops infinitos. O loop das linhas 8–12 itera enquanto o nó estiver em backoff (`nodeHealth.isDown(node)`), buscando o próximo candidato (linha 10) e incrementando o contador (linha 11). A verificação das linhas 14–18 lança uma exceção se nenhum nó saudável foi encontrado após todas as tentativas, permitindo que camadas superiores tratem o erro apropriadamente (por exemplo, enfileirando a mensagem para retry posterior).

A Tabela 5 resume os estados possíveis de um nó monitorado e seus comportamentos associados.

Tabela 5 – Estados de saúde de nós no sistema NodeHealth.

Estado	**Condição**	**Comportamento**
OK	`consecutiveFails == 0`	Disponível para envio normal.
WARNING	`0 < consecutiveFails < max`	Disponível, mas instável; monitorado.
DOWN	`consecutiveFails >= max`	Circuit breaker aberto; indisponível.
RECOVERING	`downUntilMillis < now`	Backoff expirado; disponível novamente.

4.4.4 Benefícios Observados

Embora este trabalho não tenha realizado uma avaliação experimental quantitativa do NodeHealth, é possível discutir de forma qualitativa os benefícios esperados a partir da arquitetura proposta.

Em um cenário em que um Worker Fog fica offline por falha de hardware, sem o NodeHealth o dispatcher continuaria enviando mensagens para esse nó, acumulando timeouts (tipicamente 30 segundos cada) e aumentando artificialmente a latência percebida pelo cliente. Com o mecanismo de saúde, após 3 falhas consecutivas (configurável), o nó é marcado como DOWN e temporariamente removido da rota de envio por 60 segundos, período durante o qual as mensagens são direcionadas para outros Workers saudáveis.

De maneira semelhante, em cenários de Workers Edge com conectividade intermitente (comum em redes móveis ou WiFi instável), o uso de circuit breaker evita sucessivas tentativas de envio para nós instáveis, favorecendo o redirecionamento automático para nós Fog ou Cloud mais estáveis.

4.5 Registro e Re-registro Automático

Em sistemas distribuídos, diversos eventos podem comprometer o estado de registro dos nós: quedas de energia em dispositivos Edge, reinícios do Manager para manutenção

ou atualização, perda temporária de conectividade de rede. O mecanismo implementado garante que todos os nós permaneçam registrados após essas falhas sem necessidade de intervenção manual ou reinício completo do sistema.

4.5.1 Mecanismo de Heartbeat

A Figura 9 ilustra o fluxo de heartbeats entre Manager e Workers, demonstrando como o sistema mantém a sincronização de estado através de mensagens periódicas.

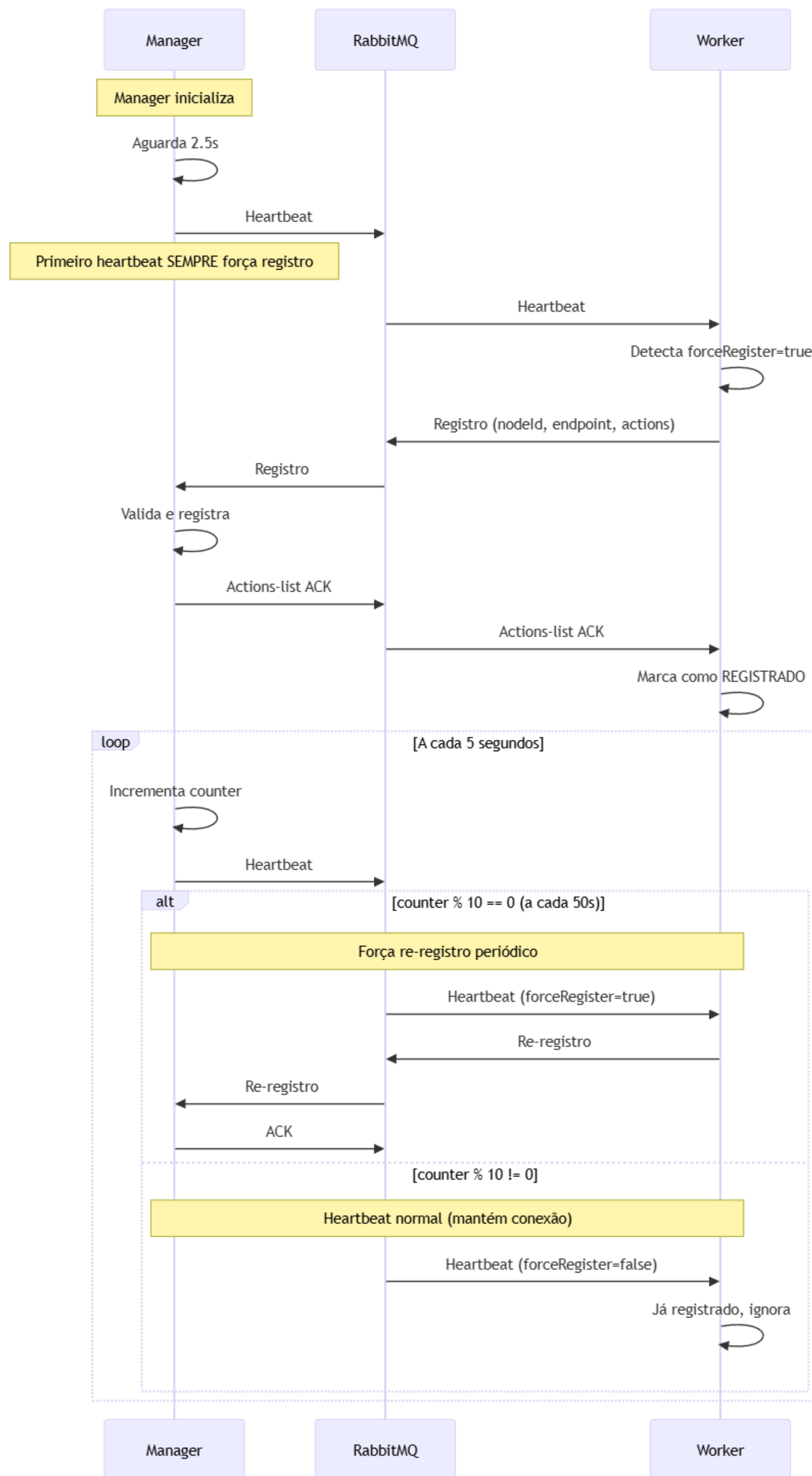

Figura 9 – Fluxo de heartbeats do Manager. Heartbeats periódicos com `forceRegister=true` a cada N ciclos garantem re-sincronização automática.

O diagrama de sequência mostra três entidades: Manager, RabbitMQ e Worker. O Manager publica heartbeats periodicamente (a cada 5 segundos por padrão) na exchange fanout `sapparchi.manager.available`. Heartbeats regulares contêm `forceRegister=false`, indicando operação normal. A cada 10 heartbeats (aproximadamente 50 segundos), um heartbeat especial com `forceRegister=true` é enviado, sinalizando aos Workers que devem re-registrar-se para atualizar suas informações no Manager, independentemente de seu estado atual de registro.

4.5.2 Implementação: ManagerHeartbeatPublisher

O Código 4.13 apresenta o componente responsável por publicar heartbeats periódicos, incluindo a lógica de re-registro forçado.

Listing 4.13 – Publicador de heartbeats do Manager com re-registro periódico.

```
@Component
public class ManagerHeartbeatPublisher {

    private final AtomicLong heartbeatCount = new AtomicLong(0);

    @Value("${sapparchi.manager.heartbeat.force-register-every
       :10}")
    private long forceRegisterEvery;

    record Heartbeat(boolean available, boolean forceRegister,
                     long ts, int schema) {}

    @EventListener(ApplicationReadyEvent.class)
    public void onReady() {
        Executors.newSingleThreadScheduledExecutor()
            .schedule(() -> heartbeatOnce(true), 2500, TimeUnit.
               MILLISECONDS);
    }

    @Scheduled(fixedRateString = "${sapparchi.manager.heartbeat-
       ms:5000}")
    public void heartbeat() {
        long count = heartbeatCount.incrementAndGet();
        boolean shouldForce = (count % forceRegisterEvery == 0);
        heartbeatOnce(shouldForce);

        if (shouldForce) {
```

```
25              LOG.info("Heartbeat #" + count + " com forceRegister=
                   true");
26          }
27      }
28
29      private void heartbeatOnce(boolean force) {
30          String json = mapper.writeValueAsString(
31              new Heartbeat(true, force, System.currentTimeMillis()
                   , 1)
32          );
33
34          // AMQP fanout para Workers Python
35          bridge.send("sendAvailableMessage-out-0",
36              MessageBuilder.withPayload(json).build());
37
38          // MQTT via amq.topic para Workers ESP32
39          bridge.send("mqttActionsList-out-0",
40              MessageBuilder.withPayload(json)
41                  .setHeader("mqtt_topic", "sapparchi.manager.
                       available")
42                  .build());
43      }
44  }
```

A linha 4 declara um contador atômico para rastrear o número de heartbeats enviados desde o boot do Manager. As linhas 6–7 configuram o intervalo de re-registro forçado através de propriedade externa (padrão: a cada 10 heartbeats). O record `Heartbeat` (linhas 9–10) define a estrutura da mensagem com campos para disponibilidade, flag de re-registro, timestamp (para dedupe) e versão do schema.

O método `onReady` (linhas 12–16) é executado quando a aplicação Spring Boot termina de inicializar. A linha 15 agenda o primeiro heartbeat com `forceRegister=true` após 2,5 segundos de delay, garantindo que Workers que conectaram antes do Manager completar a inicialização se registrem imediatamente.

O método `heartbeat` (linhas 18–27), anotado com `@Scheduled`, é executado automaticamente a cada 5 segundos (configurável). A linha 20 incrementa atomicamente o contador; a linha 21 calcula se este é um heartbeat de re-registro forçado usando o operador módulo; a linha 22 delega para `heartbeatOnce`. As linhas 24–26 registram em log quando um re-registro é forçado, facilitando diagnóstico.

O método `heartbeatOnce` (linhas 29–43) realiza a publicação efetiva. As linhas

30–32 serializam o record para JSON. As linhas 35–36 enviam via exchange fanout AMQP para Workers Python. As linhas 39–42 enviam via MQTT com header específico para Workers ESP32, utilizando o mesmo binding de saída mas com o header `mqtt_topic` que o plugin MQTT do RabbitMQ reconhece.

4.5.3 Implementação: Worker Python

O Código 4.14 apresenta o watcher de disponibilidade que monitora o Manager e força re-registro quando detecta transições de estado.

Listing 4.14 – Watcher de disponibilidade do Manager no Worker Python.

```
_IS_REGISTERED = False
_LAST_MANAGER_UP = None
_LAST_HEARTBEAT_TS = 0

async def start_register_on_manager_available(poll_seconds: int =
    3):
    """Monitora Manager e forca re-registro em transicoes DOWN->
       UP."""
    global _LAST_MANAGER_UP

    while True:
        up = _manager_is_up()  # HTTP GET /actuator/health

        if _LAST_MANAGER_UP is False and up:
            log.info("Manager voltou! Re-registro imediato...")
            mark_unregistered()
            await send_register_message(force=True)

        elif _LAST_MANAGER_UP is True and not up:
            log.info("Manager caiu! Marcando como nao registrado.
               ")
            mark_unregistered()

        _LAST_MANAGER_UP = up
        await asyncio.sleep(poll_seconds)

async def handle_manager_heartbeat(payload_str: str):
    """Processa heartbeats recebidos do Manager."""
    global _LAST_HEARTBEAT_TS, _IS_REGISTERED

    data = json.loads(payload_str)
```

```
29      force_register = data.get("forceRegister", False)
30      hb_ts = data.get("ts", 0)
31
32      # Dedupe por timestamp
33      if hb_ts > 0 and hb_ts == _LAST_HEARTBEAT_TS:
34          return
35      _LAST_HEARTBEAT_TS = hb_ts
36
37      # Re-registra se forcado OU se nao esta registrado
38      if force_register or not _IS_REGISTERED:
39          log.info("Heartbeat: force=%s, registered=%s. Enviando
               registro...",
40                   force_register, _IS_REGISTERED)
41          await send_register_message(force=True)
```

As linhas 1–3 declaram variáveis globais para rastrear o estado: `_IS_REGISTERED` indica se o Worker está registrado, `_LAST_MANAGER_UP` armazena a última condição conhecida do Manager, e `_LAST_HEARTBEAT_TS` guarda o timestamp do último heartbeat processado (para dedupe).

O método `start_register_on_manager_available` (linhas 5–22) executa em loop infinito a cada `poll_seconds`. A linha 10 verifica a disponibilidade do Manager através de requisição HTTP ao endpoint `/actuator/health`. O bloco das linhas 12–15 detecta a transição `DOWN → UP`: quando o Manager estava indisponível e agora está disponível, o Worker marca-se como não registrado (linha 14) e envia registro imediato (linha 15), ignorando qualquer debounce. O bloco das linhas 17–19 trata a transição `UP → DOWN`: o Worker apenas marca-se como não registrado, preparando-se para re-registro quando o Manager retornar.

O método `handle_manager_heartbeat` (linhas 24–41) processa heartbeats recebidos via AMQP. As linhas 28–30 extraem os campos relevantes do JSON. As linhas 33–35 implementam dedupe por timestamp, ignorando heartbeats duplicados que podem chegar por múltiplos caminhos (fanout + topic). As linhas 38–41 decidem se deve re-registrar: se `forceRegister` está ativo OU se o Worker não está registrado, envia registro com `force=True`.

4.5.4 Implementação: Worker ESP32

O Código 4.15 demonstra o registro automático no ESP32, executado imediatamente após conexão bem-sucedida com o broker MQTT.

Listing 4.15 – Registro automático ao conectar no Worker ESP32.

```
static bool mqtt_connect_internal() {
    if (mqtt_client.connected()) return true;

    String client_id = String("esp32-") + make_node_id();
    bool connected = mqtt_client.connect(
        client_id.c_str(), MQTT_USER, MQTT_PASS
    );

    if (connected) {
        Serial.println("[MQTT] Conectado ao broker");

        // Reset estado antes de registrar
        mqtt_state.registration_confirmed = false;
        mqtt_state.heartbeat_count = 0;

        // Subscreve topicos de controle
        subscribe_control_and_unicast();

        // Registro automatico imediato
        Serial.println("[MQTT] Enviando registro automatico...");
        publish_registration_once(true);
        mqtt_state.last_register_attempt_ms = millis();
    }

    return connected;
}
```

A linha 2 verifica se já está conectado, evitando reconexões desnecessárias. As linhas 4–7 tentam estabelecer conexão com o broker, usando um client ID único baseado no MAC address do dispositivo e as credenciais configuradas.

O bloco das linhas 9–23 é executado apenas em caso de sucesso. As linhas 12–14 resetam o estado interno, garantindo um estado limpo para o novo ciclo: `registration_confirmed` volta a `false` (aguardando ACK) e o contador de heartbeats é zerado. A linha 17 subscreve os tópicos de controle (heartbeats, actions-list, unicast). Finalmente, as linhas 20–22 publicam o registro imediatamente com `force=true`, garantindo que o Worker seja conhecido pelo Manager o mais rápido possível após conectar.

O Código 4.16 apresenta a função de publicação do registro com todos os metadados necessários.

Listing 4.16 – Publicação de registro com metadados do Worker ESP32.

```
static void publish_registration_once(bool force) {
```

```
2       cJSON* root = cJSON_CreateObject();
3       if (!root) return;
4
5       String node_id = make_node_id();
6       String endpoint = local_endpoint_http();
7
8       cJSON_AddStringToObject(root, "endpoint", endpoint.c_str());
9       cJSON_AddStringToObject(root, "layer", "EDGE");
10      cJSON_AddStringToObject(root, "type", "ESP32");
11      cJSON_AddStringToObject(root, "nodeId", node_id.c_str());
12      cJSON_AddBoolToObject(root, "forceRegister", force);
13
14      cJSON* actions = cJSON_AddArrayToObject(root, "actions");
15      cJSON_AddItemToArray(actions,
16          cJSON_CreateString("urn:sapparchi:Action:actions.
                fetch_data"));
17      cJSON_AddItemToArray(actions,
18          cJSON_CreateString("urn:sapparchi:Action:actions.
                fator_potencia"));
19
20      char* payload = cJSON_PrintUnformatted(root);
21      cJSON_Delete(root);
22      if (!payload) return;
23
24      mqtt_client.publish(SAP_NODES_REGISTER_DOT, payload);
25      Serial.printf("[MQTT] Registro publicado: %s\n",
            SAP_NODES_REGISTER_DOT);
26
27      free(payload);
28  }
```

As linhas 2–3 criam o objeto JSON raiz e tratam falha de alocação. As linhas 5–6 obtêm o identificador único (baseado no MAC) e o endpoint HTTP local (para fallback). As linhas 8–12 adicionam os campos obrigatórios: endpoint, camada (EDGE para ESP32), tipo do dispositivo, nodeId e flag de forceRegister. As linhas 14–18 adicionam o array de actions suportadas por este Worker, usando URNs no formato Sapparchi. As linhas 20–22 serializam e liberam a estrutura JSON. As linhas 24–27 publicam no tópico de registro e liberam a memória.

4.5.5 Mecanismos Anti-Flood

Três mecanismos complementares previnem flood de registros duplicados que poderiam sobrecarregar o Manager. O primeiro é o dedupe por fingerprint no Manager, apresentado no Código 4.17.

Listing 4.17 – Dedupe por fingerprint para evitar registros duplicados no Manager.

```
private final ConcurrentHashMap<String, String> lastFingerprint =
    new ConcurrentHashMap<>();
private final ConcurrentHashMap<String, Instant> lastSeen =
    new ConcurrentHashMap<>();
private static final Duration REG_TTL = Duration.ofMinutes(5);

private boolean isDuplicate(String nodeKey, String newFingerprint
   ) {
    String last = lastFingerprint.get(nodeKey);
    Instant seen = lastSeen.get(nodeKey);

    boolean ttlValid = seen != null &&
        Duration.between(seen, Instant.now()).compareTo(REG_TTL)
           < 0;

    return last != null && last.equals(newFingerprint) &&
       ttlValid;
}
```

As linhas 1–5 declaram as estruturas de armazenamento: `lastFingerprint` guarda o hash SHA-256 do último registro de cada nó; `lastSeen` registra o timestamp; `REG_TTL` define o tempo de vida do cache (5 minutos). O método `isDuplicate` (linhas 7–15) calcula se o registro é duplicado: as linhas 11–12 verificam se o registro anterior ainda é válido (dentro do TTL); a linha 14 retorna `true` apenas se o fingerprint for idêntico E o TTL não tiver expirado. Registros com `forceRegister=true` ignoram esta verificação.

O segundo mecanismo é o dedupe por timestamp nos Workers (já apresentado no Código 4.14, linhas 33–35), que ignora heartbeats com timestamp já processado.

O terceiro mecanismo é o backoff entre tentativas no ESP32, apresentado no Código 4.18.

Listing 4.18 – Backoff entre tentativas de registro no ESP32.

```
static const unsigned long REGISTER_RETRY_BACKOFF_MS = 10000;
static unsigned long last_register_attempt_ms = 0;

```

```
4 // No handler de heartbeat:
5 unsigned long now = millis();
6 if ((now - last_register_attempt_ms) < REGISTER_RETRY_BACKOFF_MS)
      {
7     return;  // Aguarda backoff
8 }
9 last_register_attempt_ms = now;
```

A linha 1 define o intervalo mínimo entre tentativas (10 segundos). As linhas 5–8 verificam se tempo suficiente passou desde a última tentativa; se não, o registro é ignorado. A linha 9 atualiza o timestamp para a próxima verificação.

4.5.6 Resumo de Estados

A Figura 10 apresenta a máquina de estados de registro de um Worker, ilustrando as transições possíveis entre estados.

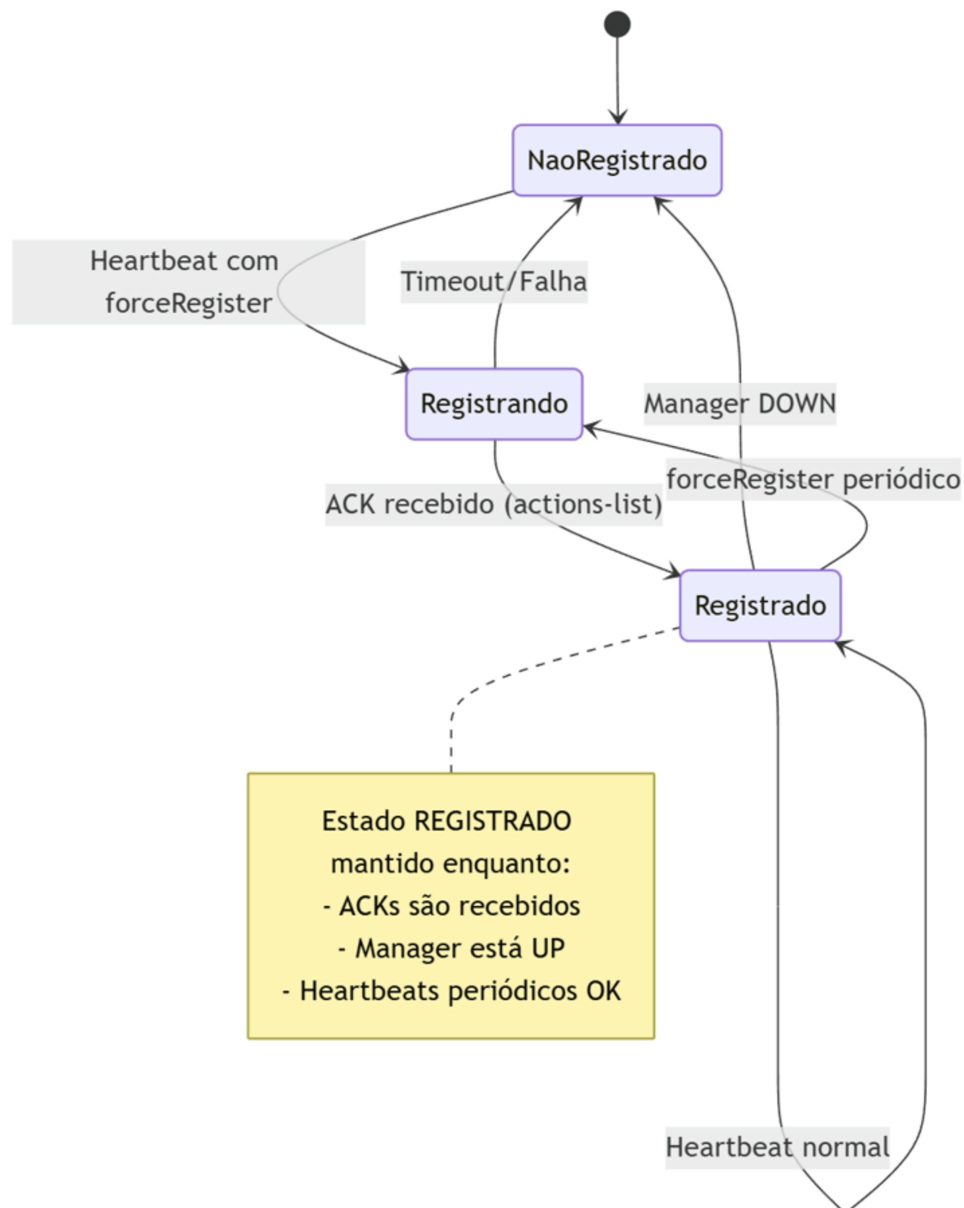

Figura 10 – Máquina de estados de registro. Transições ocorrem por eventos de conexão, heartbeats e quedas do Manager.

O diagrama apresenta três estados principais conectados por setas de transição. O estado `NaoRegistrado` é o estado inicial após boot ou quando o Manager está indisponível. Ao estabelecer conexão com o broker, o Worker transiciona para `Registrando`, onde aguarda o ACK do Manager. Ao receber o ACK (mensagem `actions-list`), transiciona para `Registrado`, o estado de operação normal. Se o Manager cair (detectado via watcher ou timeout de heartbeat), retorna para `NaoRegistrado`.

A Tabela 6 detalha cada estado com seus triggers e ações associadas.

Tabela 6 – Estados de registro de Workers no Sapparchi.

Estado	**Descrição**	**Trigger de entrada**	**Ações**
NaoRegistrado	Worker sem confirmação de registro.	Boot, Manager DOWN, timeout.	Publica registro automaticamente.
Registrando	Aguardando ACK do Manager.	Conexão MQTT/AMQP estabelecida.	Aguarda `actions-list`.
Registrado	Registro confirmado.	ACK recebido.	Executa actions; responde heartbeats.

4.6 Separação de Filas de Resultados

4.6.1 Problema Original

Na versão inicial da plataforma, Workers publicavam resultados diretamente na fila final `sapparchi.task.final.saida`. Essa arquitetura ocasionava três problemas principais: inconsistência em tasks de múltiplos steps, onde resultados parciais chegavam ao cliente antes da consolidação; perda de contexto, pois o Manager não tinha oportunidade de processar resultados intermediários; e fragmentação, com o cliente recebendo múltiplas mensagens para uma única task ao invés de um resultado consolidado.

4.6.2 Solução Implementada

A arquitetura de duas filas separa claramente os resultados intermediários dos finais, conforme ilustrado na Figura 11.

Figura 11 – Arquitetura de duas filas. Fluxo: Worker → fila intermediária → Manager consolida → fila final → Cliente.

O diagrama mostra o fluxo de dados em três etapas sequenciais. Na primeira etapa, Workers de todas as camadas publicam resultados de steps individuais na fila `sapparchi.task.handler.saida`. O Manager consome estas mensagens, identifica a task correspondente pelo `requestID`, e atualiza o estado interno. Na segunda etapa, quando todos os steps de uma task são concluídos, o Manager executa a consolidação, agregando todos os resultados parciais em um único objeto JSON estruturado. Na terceira etapa, o resultado consolidado é publicado na fila `sapparchi.task.final.saida`, de onde o cliente (ou sistema externo) o consome.

4.6.3 Implementação: Worker Python

O Código 4.19 apresenta a implementação completa da publicação de resultados no Worker Python.

Listing 4.19 – Publicação de resultados na fila intermediária (Worker Python).

```
RESULT_QUEUE = "sapparchi.task.handler.saida"

async def _publish_result_message(out, *, reply_to: Optional[str]
    = None,
                                  correl_id: Optional[str] = None
                                    ):
    """
    Publica SEMPRE na fila de resultados intermediarios.
    Manager e responsavel por consolidar e publicar resultado
       final.
    """
    body = str(out)
    amqp = AmqpClient.instance()

    # Ignora reply_to, usa SEMPRE a fila padrao
    target_q = RESULT_QUEUE

    try:
        await amqp.publish_topic(
            routing_key=RESULT_QUEUE,
            body=body,
            exchange="amq.topic"
        )
        log.info("Resultado publicado rk=%s bytes=%d",
           RESULT_QUEUE, len(body))
    except Exception:
        log.exception("Falha ao publicar resultado em %s",
           RESULT_QUEUE)
```

A linha 1 define a constante com o nome da fila intermediária. O método aceita parâmetros `reply_to` e `correl_id` (linhas 3–4) para compatibilidade com o padrão RPC do AMQP, mas crucialmente ignora o `reply_to` (linha 13), garantindo que todos os resultados sigam o mesmo caminho. As linhas 16–20 publicam na exchange `amq.topic` com a routing key da fila intermediária. O bloco try-except (linhas 15–23) garante que falhas de publicação sejam registradas para diagnóstico.

4.6.4 Implementação: Worker ESP32

O Worker ESP32 utiliza a mesma fila que o Worker Python, garantindo consistência na arquitetura. A implementação já foi apresentada no Código 4.7, onde a constante `SAP_RESULT_OUT_DOT` aponta para `sapparchi.task.handler.saida`.

4.6.5 Vantagens da Separação

A Tabela 7 compara a arquitetura original com a nova arquitetura de filas separadas em diversos aspectos.

Tabela 7 – Comparação entre arquitetura original e arquitetura com filas separadas.

Aspecto	**Arquitetura Original**	**Arquitetura com Separação**
Consolidação	Não havia; cliente recebia fragmentos.	Manager consolida todos os steps.
Rastreamento	Impossível rastrear steps individuais.	Cada step registrado separadamente.
Resiliência	Perda de steps sem possibilidade de retry.	Retry granular por step.
Debugging	Difícil identificar falhas.	Logs detalhados de cada etapa.
Escalabilidade	Limitada a um consumidor.	Múltiplos managers em paralelo.

4.7 Fluxos de Execução

Esta seção detalha os principais fluxos de execução implementados, ilustrando a interação entre os componentes em cenários típicos de operação.

4.7.1 Fluxo 1: Registro Inicial

A Figura 12 apresenta o diagrama de sequência do registro inicial de um Worker recém-inicializado.

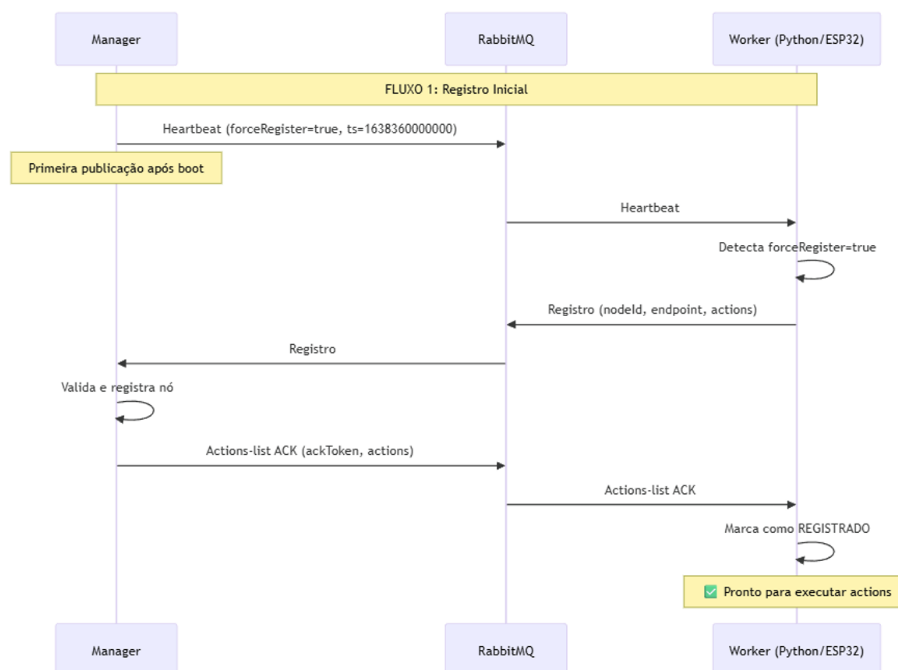

Figura 12 – Registro inicial: Worker conecta, publica registro, Manager valida e envia ACK.

O fluxo inicia com o Worker estabelecendo conexão com o broker (MQTT para ESP32, AMQP para Python). Imediatamente após a conexão bem-sucedida, o Worker publica uma mensagem de registro no tópico `sapparchi.nodes.register`. A mensagem contém todos os metadados necessários: `nodeId` (identificador único baseado em MAC ou UUID), `endpoint` (URL para fallback HTTP), `layer` (EDGE, FOG ou CLOUD), `type` (ESP32, Python, etc.), `actions` (lista de URNs das actions suportadas) e `forceRegister` (flag indicando prioridade).

O Manager recebe o registro via binding na exchange `amq.topic`, valida os campos obrigatórios (verificando camada válida, endpoint acessível, etc.), e armazena as informações no repositório de nós. Em seguida, publica um ACK na routing key `sapparchi.nodes.actions-list` contendo o token de confirmação, o `nodeId` processado e a lista de actions aceitas. O Worker recebe o ACK, verifica que o `nodeId` corresponde ao seu, e transiciona para o estado `REGISTRADO`.

4.7.2 Fluxo 2: Re-registro após Reinício do Manager

A Figura 13 ilustra o cenário de recuperação automática após falha do Manager.

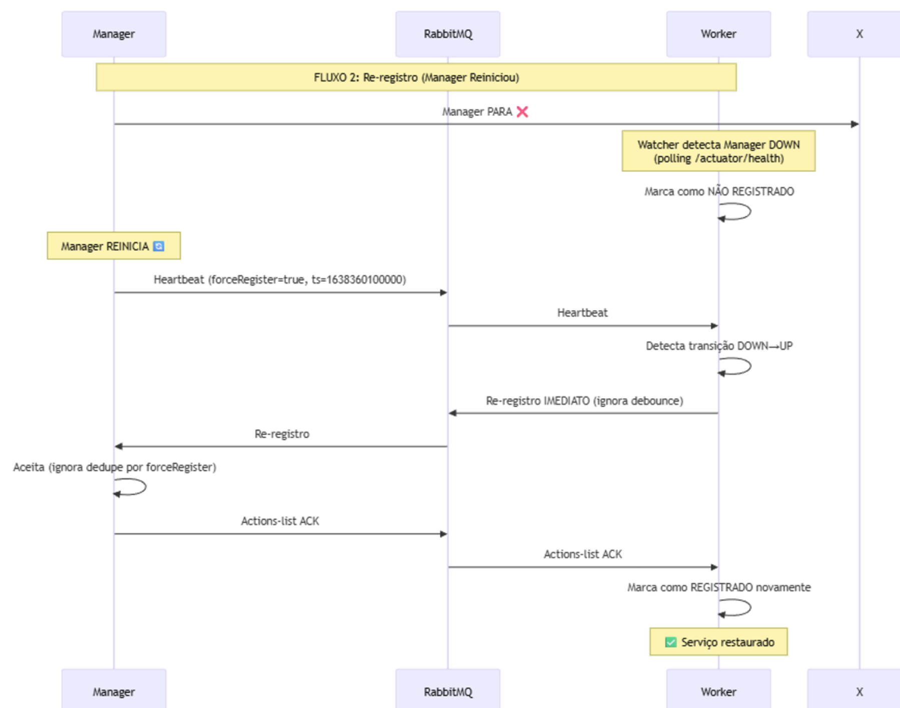

Figura 13 – Re-registro automático: Workers detectam retorno do Manager e re-registram imediatamente.

O fluxo inicia com o Manager parando (seja por manutenção planejada, falha de hardware ou crash de software). Workers Python detectam a indisponibilidade através do watcher que monitora o endpoint `/actuator/health` a cada 3 segundos. Workers ESP32 detectam pela ausência de heartbeats após o timeout configurado. Ambos transitam para o estado `NaoRegistrado`.

Quando o Manager retorna e completa sua inicialização, seu primeiro heartbeat (enviado após 2,5 segundos de delay) contém `forceRegister=true`. Simultaneamente, Workers Python detectam a transição `DOWN → UP` através do watcher HTTP. Ambos os mecanismos disparam re-registro imediato com `force=true`, ignorando qualquer debounce ou backoff, garantindo que o Manager reconstrua seu conhecimento sobre os nós disponíveis em segundos.

4.7.3 Fluxo 3: Re-registro Periódico

A Figura 14 apresenta o fluxo de re-registro periódico em condições normais de operação.

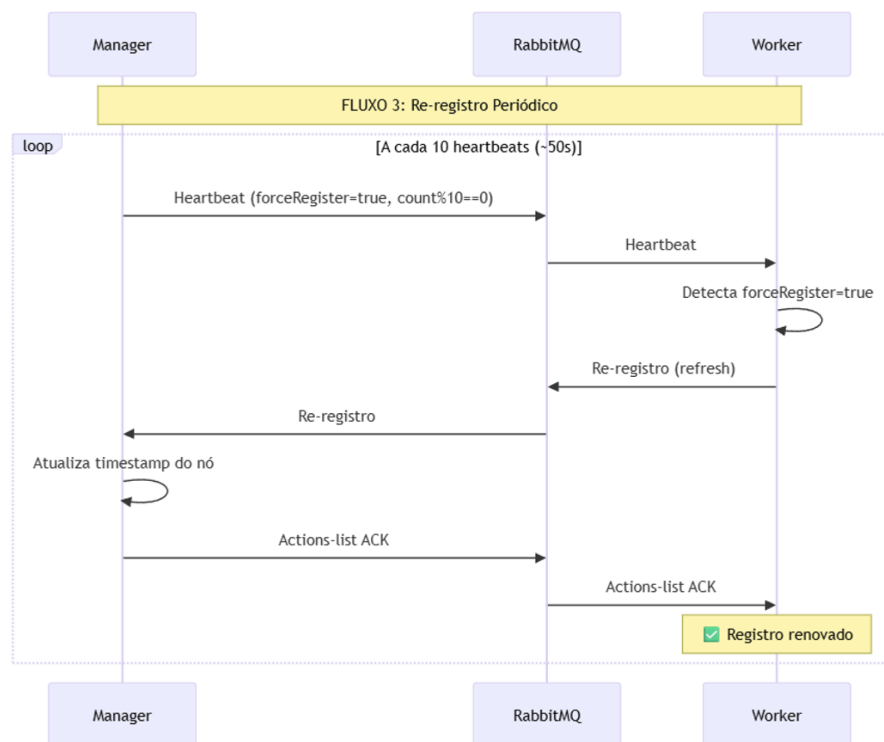

Figura 14 – Re-registro periódico: refresh automático a cada 50 segundos mantém consistência.

Com heartbeats a cada 5 segundos e `forceRegisterEvery=10`, um heartbeat com `forceRegister=true` é enviado aproximadamente a cada 50 segundos. Esse mecanismo é necessário por diversos motivos: caches Redis podem perder informações e precisam ser atualizados; dispositivos Edge com DHCP podem mudar de IP entre reconexões; Workers podem carregar novas actions dinamicamente via hot-reload; e o processo ajuda a limpar entradas obsoletas de nós que foram removidos sem desregistro explícito.

4.7.4 Fluxo 4: Execução de Action Individual

A Figura 15 ilustra a execução de uma única action, o caso mais simples de processamento.

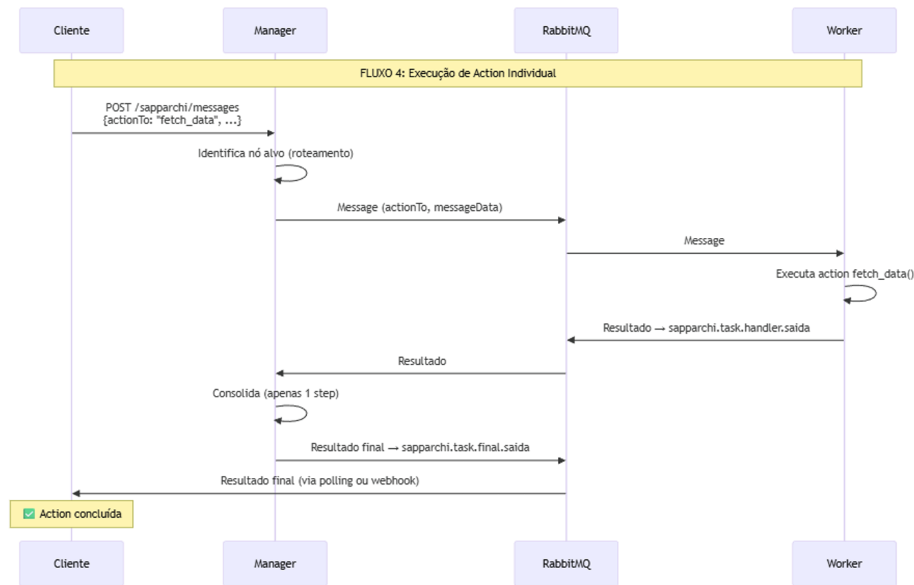

Figura 15 – Execução de action individual: requisição, dispatch, execução e consolidação.

O cliente envia uma requisição ao Manager especificando a action desejada (via URN) e os dados de entrada em formato JSON. O Manager consulta o repositório de nós para encontrar um Worker saudável que suporte a action, verificando disponibilidade via NodeHealth. Usando o ProtocolResolver, determina a ordem de protocolos e despacha a mensagem.

O Worker recebe a mensagem, localiza o executor correspondente à URN, e invoca a lógica de negócio. Ao concluir (com sucesso ou erro), serializa o resultado e publica na fila intermediária. O Manager consome o resultado, verifica que se trata de uma action individual (sem task associada no registro), e republica diretamente na fila final para consumo pelo cliente.

4.7.5 Fluxo 5: Execução de Task Multi-Step

A Figura 16 apresenta o fluxo mais complexo: uma task composta por múltiplas actions encadeadas com dependências.

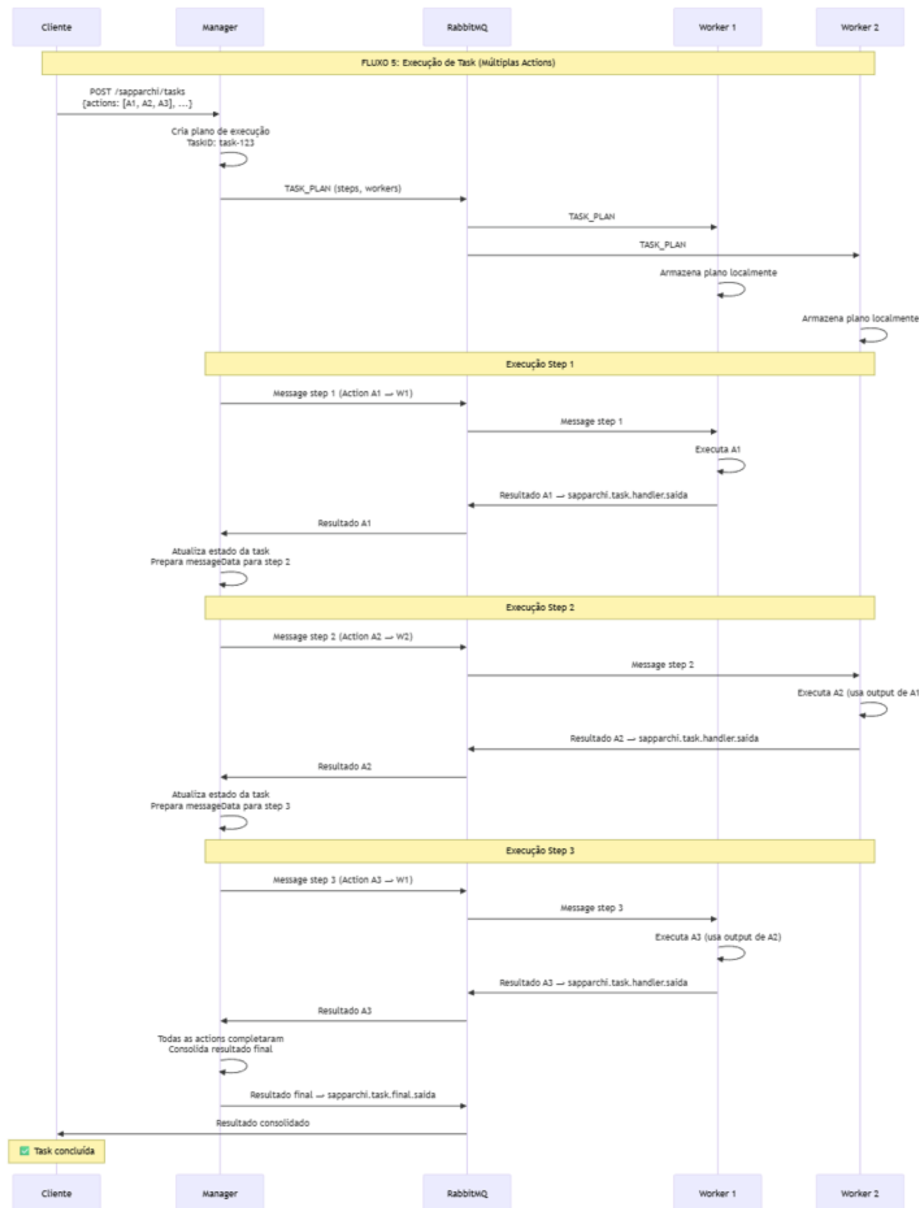

Figura 16 – Execução de task multi-step: orquestração com passagem de resultados entre steps.

O cliente submete uma task especificando múltiplas actions e suas dependências. O Manager cria um TASK_PLAN que define a ordem de execução respeitando dependências (steps sem dependências podem executar em paralelo). Para cada step, o Manager: identifica o Worker apropriado, injeta no `messageData` os resultados dos steps dependentes, e despacha a mensagem.

Conforme cada Worker conclui seu step e publica na fila intermediária, o Manager atualiza o estado da task, verifica se novas ações ficaram desbloqueadas (dependências satisfeitas), e continua a orquestração. Quando o último step conclui, o Manager executa a consolidação final, agregando todos os resultados em uma estrutura JSON hierárquica com metadados de temporização, e publica na fila final.

5 CONCLUSÕES

5.1 Contribuições

Este trabalho apresentou uma solução completa de integração de mensageria assíncrona à plataforma Sapparchi, abrangendo diversos aspectos da comunicação e orquestração em ambientes distribuídos Edge-Fog-Cloud.

No eixo de comunicação assíncrona, foi implementada uma arquitetura baseada em RabbitMQ com suporte aos protocolos AMQP e MQTT, além de HTTP como mecanismo de fallback. Essa abordagem elimina o acoplamento temporal entre Manager e Workers, permitindo que mensagens sejam enfileiradas e processadas de forma independente.

Quanto ao roteamento, foi desenvolvido um algoritmo de seleção automática de protocolo baseado na camada computacional do nó de destino. Nós da camada Edge utilizam preferencialmente MQTT devido ao seu baixo overhead, enquanto nós das camadas Fog e Cloud empregam AMQP por suas garantias de entrega. O sistema também implementa seleção de workers saudáveis com fallback automático entre protocolos.

O sistema de registro e monitoramento constitui outra contribuição relevante. Foi implementado um mecanismo de heartbeat com flag `forceRegister` que permite ao Manager solicitar re-registro de todos os Workers periodicamente ou após reinicializações. Os Workers monitoram a disponibilidade do Manager através de um watcher que detecta transições de estado UP e DOWN, re-registrando-se automaticamente quando necessário. Re-registros periódicos a cada 50 segundos garantem a consistência do catálogo de nós.

A separação de filas de resultados em intermediárias e finais permite que o Manager consolide os resultados de tasks compostas por múltiplas actions. Cada Worker publica seus resultados na fila intermediária `sapparchi.task.handler.saida`, e o Manager é responsável por rastrear o progresso de cada task e publicar o resultado consolidado na fila final `sapparchi.task.final.saida`.

A orquestração avançada de tasks foi implementada com suporte a múltiplas actions encadeadas através de um plano de execução (TASK_PLAN) que define dependências entre steps. O sistema realiza encadeamento automático de outputs entre steps e consolida os resultados finais de forma transparente.

Por fim, mecanismos de resiliência e recuperação foram incorporados, incluindo recuperação automática em caso de falha do Manager, retry automático de mensagens falhadas e dedupe de heartbeats e registros para evitar processamento redundante.

5.2 Resultados Qualitativos

Do ponto de vista de desempenho, a adoção da comunicação assíncrona no Sapparchi tornou a interação entre *Manager* e *Workers* mais fluida. A eliminação do acoplamento temporal típico do HTTP síncrono reduziu a ocorrência de chamadas bloqueantes e de *timeouts* percebidos na prática, resultando em um comportamento mais estável mesmo em cenários com múltiplos nós ativos.

Quanto à resiliência, o sistema mostrou-se capaz de lidar melhor com falhas temporárias tanto do *Manager* quanto dos *Workers*. A combinação de *heartbeats*, re-registro automático e separação entre filas intermediárias e finais permitiu que tarefas em execução fossem retomadas ou reencaminhadas após oscilações de rede ou reinicializações de componentes, sem necessidade de intervenção manual.

De forma geral, a arquitetura proposta apresentou um comportamento mais robusto e previsível do que a abordagem baseada exclusivamente em HTTP síncrono.

5.3 Trabalhos Futuros

A arquitetura desenvolvida neste trabalho abre espaço para diversas evoluções que podem aprimorar ainda mais o desempenho, a observabilidade e a inteligência do sistema. Entre as direções mais promissoras, destacam-se três frentes principais: compressão de payloads, dashboard de monitoramento e roteamento baseado em aprendizado de máquina.

A compressão de payloads representa uma oportunidade significativa de otimização, especialmente em cenários onde o volume de dados trafegados entre os componentes é elevado. A avaliação de algoritmos como gzip, LZ4 e Snappy para mensagens acima de determinado limiar de tamanho pode reduzir substancialmente o consumo de largura de banda, beneficiando principalmente dispositivos de borda com conexões limitadas. A implementação dessa funcionalidade envolveria a definição de políticas de compressão configuráveis por camada, permitindo que nós Edge utilizem algoritmos mais leves enquanto nós Fog e Cloud podem empregar técnicas de maior taxa de compressão.

O desenvolvimento de um dashboard de monitoramento integrado constitui outra evolução relevante para a plataforma. A integração com ferramentas como Grafana e Prometheus permitiria a visualização em tempo real de métricas essenciais, incluindo latência de processamento, throughput de mensagens, taxa de erro por worker e estado de saúde dos nós. Além da visualização, essa infraestrutura habilitaria a configuração de alertas automáticos para detecção precoce de anomalias, como degradação de desempenho em determinada camada ou indisponibilidade prolongada de workers específicos. Essa capacidade de observabilidade é fundamental para a operação do sistema em ambientes de produção.

Por fim, a incorporação de técnicas de aprendizado de máquina para roteamento representa uma evolução mais sofisticada do algoritmo de seleção de workers. Atualmente, o roteamento considera apenas a camada do nó e seu estado de saúde. Um modelo preditivo treinado com histórico de execuções poderia considerar fatores adicionais como latência média por tipo de action, carga atual dos workers, localização geográfica e padrões temporais de demanda. Essa abordagem permitiria decisões de roteamento mais inteligentes, direcionando cada requisição para o worker com maior probabilidade de processá-la com menor latência e maior confiabilidade.

5.4 Considerações Finais

A partir da necessidade de orquestrar tarefas distribuídas em um ambiente Edge–Fog–Cloud, este trabalho propôs a integração de mensageria assíncrona à plataforma Sapparchi. A solução desenvolvida introduziu o uso de RabbitMQ como broker central, o suporte a múltiplos protocolos de comunicação (AMQP, MQTT e HTTP) e um conjunto de mecanismos de roteamento, monitoramento e registro de nós que tornam o sistema mais aderente ao conceito de *computação contínua*.

Do ponto de vista prático, a migração de um modelo baseado exclusivamente em HTTP síncrono para uma abordagem centrada em filas e mensagens resultou em uma interação mais fluida entre o *Manager* e os *Workers*. Em dispositivos de borda, o uso de MQTT aliado ao modelo assíncrono mostrou-se mais compatível com as restrições de recursos típicas desses ambientes. A necessidade de manter conexões HTTP constantes foi reduzida, e o sistema passou a lidar melhor com desconexões intermitentes, sem exigir intervenção manual para retomar o processamento.

De modo semelhante, o mecanismo de *heartbeats*, o re-registro automático de nós e o serviço de *NodeHealth* contribuíram para um comportamento mais previsível diante de falhas temporárias de *Workers* ou do próprio *Manager*, favorecendo a recuperação gradual do sistema após reinicializações ou perdas de conectividade.

A separação entre filas de resultados intermediários e finais também se mostrou conceitualmente adequada para cenários de tarefas multi-etapas. Essa organização permitiu uma orquestração mais clara das *Tasks*, com tratamento explícito de cada *Action*, melhorando a capacidade de depuração e abrindo espaço para estratégias futuras de escalonamento e balanceamento de carga.

Por fim, a arquitetura proposta posiciona o Sapparchi como uma base promissora para aplicações reais em cenários de Internet das Coisas e computação distribuída, nas quais a combinação de borda, névoa e nuvem é essencial. Entre os domínios potenciais de aplicação, destacam-se cidades inteligentes, monitoramento industrial, agricultura de

precisão e sistemas de saúde conectados, nos quais a capacidade de orquestrar tarefas em diferentes camadas, de forma resiliente e desacoplada, é um requisito central. A consolidação dessa arquitetura, aliada às extensões sugeridas na seção de trabalhos futuros, indica um caminho claro para a evolução do Sapparchi como plataforma de pesquisa e de prototipagem em ambientes Edge–Fog–Cloud.

REFERÊNCIAS

BALOUEK-THOMERT, D. et al. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. *The International Journal of High Performance Computing Applications*, SAGE Publications, v. 33, n. 6, p. 1159–1174, 2019.

BONOMI, F. et al. Fog computing and its role in the internet of things. In: ACM. *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. [S.l.], 2012. p. 13–16.

CHENG, B. et al. Fogflow: Easy programming of iot services over cloud and edges for smart cities. *IEEE Internet of Things Journal*, IEEE, v. 5, n. 2, p. 696–707, 2018.

EUGSTER, P. T. et al. The many faces of publish/subscribe. *ACM Computing Surveys*, ACM, v. 35, n. 2, p. 114–131, 2003.

GUPTA, H. et al. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, Wiley Online Library, v. 47, n. 9, p. 1275–1296, 2017.

HOHPE, G.; WOOLF, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston: Addison-Wesley, 2004. ISBN 978-0-321-20068-6.

Linux Foundation. *EdgeX Foundry: Open source edge computing platform*. 2024. https://www.edgexfoundry.org/. Acesso em: 15 nov. 2024.

OASIS. *MQTT: The Standard for IoT Messaging*. 2024. https://mqtt.org/. Acesso em: 15 nov. 2024.

OpenFog Consortium. *OpenFog Reference Architecture for Fog Computing*. [S.l.], 2017.

SATYANARAYANAN, M. The emergence of edge computing. *Computer*, IEEE, v. 50, n. 1, p. 30–39, 2017.

SOUZA, A. et al. Sapparchi: an osmotic platform to execute scalable applications on smart city environments. In: IEEE. *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. [S.l.], 2022. p. 289–298.

SOUZA, A. E. C. d. S. e. *SAPPARCHI: A Scalable Platform to Execute Applications on Computational Smart City Environments*. Tese (Tese de Doutorado) — Universidade Federal do Rio Grande do Norte, 2022.

VIDELA, A.; WILLIAMS, J. *RabbitMQ in Action: Distributed Messaging for Everyone*. [S.l.]: Manning Publications, 2012.

APÊNDICE A – CONFIGURAÇÕES DO SISTEMA

A.1 application.yml (Manager)

```yaml
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: sapparchi
    password: sapparchi123

  cloud:
    stream:
      bindings:
        sendAvailableMessage-out-0:
          destination: sapparchi.manager.available
          binder: rabbit
        mqttActionsList-out-0:
          destination: amq.topic
          binder: rabbit

sapparchi:
  manager:
    heartbeat-ms: 5000
    heartbeat:
      force-register: false
      initial-delay-ms: 2500
      force-register-every: 10
```

A.2 config.yaml (Worker Python)

```yaml
sapparchi:
  node:
    nodeId: "python-worker-01"
    layer: "FOG"
    type: "PYTHON"
    endpoint: "http://192.168.1.20:8080"
```

```
7
8   rabbitmq:
9     host: "192.168.1.10"
10    port: 5672
11    username: "sapparchi"
12    password: "sapparchi123"
13
14  actions:
15    path: "./actions"
```

A.3 mqtt config.h (ESP32)

```
1  #define MQTT_HOST "192.168.1.10"
2  #define MQTT_PORT 1883
3  #define MQTT_USER ""
4  #define MQTT_PASS ""
5  #define MQTT_KEEPALIVE_SEC 30
6  #define MQTT_SOCKET_TIMEOUT_S 15
7
8  #define RESULT_QUEUE_DOT "sapparchi.task.handler.saida"
9  #define FORCE_REREGISTER_EVERY 10
10 #define REGISTER_RETRY_BACKOFF_MS 15000
```