IX Encontro de Iniciação Científica e Tecnológica
IX EnICT
ISSN: 2526-6772
IFSP – *Campus* Araraquara
6 de dezembro de 2025

HELLO WORLD PARA COMUNICAÇÃO ASSÍNCRONA ENTRE APLICAÇÕES: CONHECENDO O RABBITMQ

MARCO ANTÔNIO Z. MONTANA[1], EDNILSON G. ROSSI[2], JANAINA C. ABIB[2]

[1] Discente do curso de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de São Paulo (IFSP) no *Campus* Araraquara. E-mail: m.zanchettta@aluno.ifsp.edu.br

[2] Docente do Instituto Federal de São Paulo (IFSP) no *Campus* Araraquara. E-mail: {ednilsonrossi, janaina}@ifsp.edu.br

Área de conhecimento (Tabela CNPq): Engenharia de Software – 1.03.03.02-2

RESUMO: A comunicação entre aplicações é essencial em sistemas distribuídos, especialmente para integração entre serviços distintos. Os serviços de mensageria permitem comunicação assíncrona, garantindo escalabilidade, resiliência e desempenho. Este artigo apresenta conceitos iniciais dos serviços de mensageria, descreve características do middleware RabbitMQ, além de demonstrar, por meio de um exemplo prático, o envio e consumo de mensagens em formato JSON entre produtor e consumidor. Os resultados evidenciam o desacoplamento de sistemas e a eficiência do RabbitMQ para integração confiável entre aplicações distribuídas.

PALAVRAS-CHAVE: aplicações distribuídas; serviços mensageria; troca de mensagens; integração não bloqueante de sistemas.

INTRODUÇÃO

A comunicação entre componentes internos de um software é fundamental para o correto funcionamento e integração das funcionalidades oferecidas por uma aplicação. Em sistemas orientados a objetos, essa comunicação ocorre, geralmente, por meio de chamadas de métodos, onde um objeto invoca uma operação de outro objeto, recebendo eventualmente um retorno. No contexto de sistemas distribuídos, esse conceito é estendido através do *Remote Procedure Call* (RPC), permitindo que diferentes componentes interajam mesmo quando executados em processos ou máquinas distintas, mantendo a abstração de uma chamada local (BIRRELL; NELSON, 1984; STEEN; TANENBAUM, 2023; COULOURIS et al., 2013).

De acordo com Richards e Ford (2020), com a evolução das aplicações e o aumento da complexidade dos sistemas, surgiram diferentes arquiteturas distribuídas que moldaram a forma como os componentes se comunicam. Segundo os mesmos autores, a Arquitetura Orientada a Serviços (SOA, do inglês *Service-Oriented Architecture*) foi uma das primeiras abordagens amplamente adotadas, propondo a decomposição de aplicações em serviços independentes que se comunicam geralmente via protocolos baseados em XML e SOAP. Newman (2022) observa que, posteriormente, a arquitetura de microsserviços surgiu como uma evolução natural, reforçando a ideia de pequenos serviços autônomos, cada um responsável por uma funcionalidade específica e se comunicando por meio de APIs leves, frequentemente REST ou gRPC. Ainda segundo Newman (2022), conceitos mais recentes como Arquitetura Orientada a Eventos (EDA, do inglês *Event-Driven Architecture*) e Computação sem Servidor (*Serverless Computing*) ampliaram ainda mais a descentralização, enfatizando a comunicação baseada em eventos e o processamento sob demanda, conforme

a ocorrência de determinados gatilhos. A escolha entre comunicação síncrona ou assíncrona é um aspecto central em qualquer arquitetura distribuída, pois influencia diretamente o desempenho, a escalabilidade e a resiliência dos sistemas.

Na comunicação síncrona, o produtor envia uma requisição e aguarda a resposta antes de prosseguir, assim como as chamadas de métodos na programação orientada a objetos. Esse modelo é intuitivo e facilita o controle do fluxo lógico, sendo amplamente utilizado em APIs REST, por exemplo. Contudo, em sistemas de alta demanda, pode gerar forte acoplamento e baixa resiliência: se o serviço de destino estiver indisponível, toda a cadeia de chamadas pode falhar. Um exemplo clássico ocorre em um sistema de e-commerce durante o processo de checkout no qual, se o serviço de pagamento demorar para responder, o serviço de pedidos fica bloqueado, aumentando o tempo de resposta e degradando a experiência do usuário. Esse impacto negativo afeta requisitos de qualidade como desempenho, disponibilidade e escalabilidade.

Como alternativa, e não necessariamente como substituição, adota-se a comunicação assíncrona, na qual mensagens são enviadas a um intermediário e processadas de forma independente pelos destinatários. Esse modelo promove maior desacoplamento, tolerância a falhas e escalabilidade, sendo amplamente utilizado em aplicações que processam grandes volumes de dados ou precisam continuar operando mesmo diante da indisponibilidade temporária de alguns serviços. Por exemplo, em um sistema de entrega de pedidos, o serviço de pagamento pode produzir e enviar uma mensagem confirmando o sucesso da transação para uma fila, e o serviço de notificação consome essa mensagem posteriormente para enviar o e-mail de confirmação ao cliente, sem bloquear o fluxo principal do pedido. Entre as ferramentas que viabilizam a comunicação assíncrona, destaca-se o RabbitMQ, um *message broker* ou *Message Oriented Middleware* (MOM) amplamente utilizado por sua confiabilidade, simplicidade e suporte a múltiplos protocolos. Ele atua como um *middleware* responsável por intermediar a troca de mensagens entre produtores e consumidores, garantindo que os dados sejam entregues mesmo em situações de falha temporária na rede ou no sistema de destino (HOHPE; WOOLF, 2003). Contudo, é importante ressaltar que a abordagem de mensageria não é a única forma de comunicação assíncrona. Outras abordagens incluem o uso de *event streaming*, como no Apache Kafka (APACHE SOFTWARE FOUNDATION, 2025), que prioriza o fluxo contínuo de dados em tempo real; o *event sourcing*, que armazena o histórico completo das mudanças de estado de um sistema; e mecanismos baseados em filas internas ou bancos de dados temporários, onde o processamento é realizado posteriormente por tarefas agendadas (*batch processing*). Cada abordagem apresenta vantagens e desafios distintos, sendo escolhida conforme os requisitos de desempenho, consistência e acoplamento desejados.

Este trabalho busca explorar os conceitos iniciais de comunicação assíncrona entre componentes de sistemas distribuídos, apresentando o RabbitMQ como uma solução prática para o serviço mensageria confiável, mas também reconhecendo alternativas de comunicação que complementam o ecossistema de arquiteturas distribuídas modernas.

OBJETIVO

O objetivo geral deste artigo é apresentar conceitos iniciais de troca de mensagens assíncronas entre aplicações, com o uso da ferramenta RabbitMQ.

De forma mais específica, este trabalho visa mostrar formas da comunicação assíncrona em sistemas distribuídos, descrevendo sua relevância para o desempenho e a escalabilidade das aplicações; detalhar a arquitetura e os principais componentes da ferramenta RabbitMQ, incluindo o produtor, o *exchange*, a fila e o consumidor, apresentando um exemplo prático de envio e consumo de mensagens que ilustram seu funcionamento. Por fim, evidenciar as vantagens do uso da mensageria em comparação com o modelo de comunicação síncrona, destacando seus benefícios em termos de desacoplamento, tolerância a falhas e eficiência na troca de informações entre componentes distribuídos.

METODOLOGIA

Este estudo adotou uma abordagem exploratória e aplicada, com o objetivo de introduzir os conceitos fundamentais de mensageria e ilustrar o uso do RabbitMQ em um ambiente de sistemas distribuídos.

Inicialmente, foi conduzida uma pesquisa bibliográfica sobre sistemas distribuídos e comunicação assíncrona, com base em livros, artigos científicos e documentação técnica. O estudo da documentação oficial do RabbitMQ (RABBITMQ, 2025) permitiu compreender seus principais componentes, padrões de funcionamento e boas práticas de configuração. Para apoiar a compreensão prática, foram analisados materiais instrutivos, incluindo vídeos técnicos, como os disponibilizados pelo canal Full Cycle no YouTube (FULL CYCLE, 2023), que abordam a implementação do RabbitMQ em cenários reais.

Com base no referencial teórico e nas práticas estudadas, foi desenvolvida uma aplicação de exemplo para demonstrar a troca de mensagens entre serviços utilizando o RabbitMQ. Essa implementação teve como finalidade validar, de forma prática, os conceitos de comunicação assíncrona e o papel do *message broker* na integração de sistemas distribuídos.

DESENVOLVIMENTO E RESULTADOS

As pesquisas e análises realizadas ao longo do estudo possibilitaram compreender o funcionamento dos sistemas de mensageria e seus principais conceitos. A partir desse embasamento teórico, optou-se pela utilização do RabbitMQ como ferramenta para aplicação prática e validação dos conhecimentos adquiridos. Na sequência, apresenta-se uma descrição sucinta da ferramenta e de seus principais recursos.

O RabbitMQ é uma ferramenta de mensageria de código aberto (*Message Broker Open Source*) que usa o protocolo AMQP. Ele funciona como um intermediário entre quem envia a mensagem (produtor) e quem recebe mensagens (consumidor), garantindo que elas cheguem aos seus destinos mesmo se algum serviço estiver temporariamente fora do ar. Além disso, permite organizar filas de mensagens, confirmar entregas e controlar como elas são roteadas. Por isso, é muito usado em sistemas distribuídos que precisam ser confiáveis e escaláveis, como *e-commerce*, processamento de pedidos, envio de notificações ou integração entre microsserviços.

A arquitetura do RabbitMQ é composta por quatro componentes primários, todos operando de forma desacoplada: Produtor, *Exchange*, Fila e Consumidor. O Produtor é a aplicação responsável pela criação e emissão das mensagens. Sua função limita-se a publicar a mensagem em um *Exchange*, sem a necessidade de conhecimento sobre a identidade ou o estado operacional do receptor, promovendo um alto nível de desacoplamento. O Consumidor, por sua vez, é a aplicação que se conecta à Fila, recebe as mensagens e executa o processamento lógico. Para garantir a integridade e evitar a perda de dados, o Consumidor é responsável por enviar uma Confirmação de Recebimento (*Acknowledgement* – ACK) ao RabbitMQ, sinalizando que a mensagem foi processada com sucesso e pode ser permanentemente removida.

A Fila (*Queue*) atua como a estrutura de buffer persistente dentro do RabbitMQ. Ela armazena as mensagens roteadas até que o Consumidor correspondente esteja pronto para processá-las, assegurando a durabilidade e a ordem de entrega.

O *Exchange* é o componente central do RabbitMQ, atuando como agente de roteamento de mensagens. O Produtor não envia a mensagem diretamente à Fila, mas sim ao *Exchange*, que emprega uma lógica configurável para determinar a destinação da mensagem. O roteamento é determinado pela avaliação da *Routing Key* (Chave de Roteamento) fornecida pelo Produtor, em conjunto com as configurações de *Binding* (Ligação). O *Binding* é a estrutura que estabelece a regra de conexão entre um *Exchange* e uma Fila, definindo precisamente como cada mensagem deve ser encaminhada.

O RabbitMQ oferece diferentes mecanismos de roteamento (RABBITMQ, 2025):

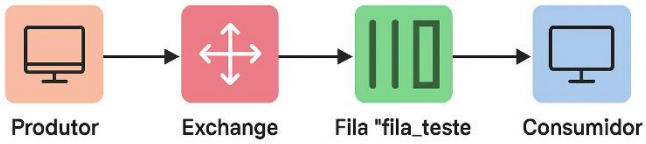
- *Direct Exchange*: roteia a mensagem exclusivamente para Filas cuja *Binding Key* corresponda exatamente à *Routing Key*.

- *Fanout Exchange*: executa um roteamento de dispersão (*broadcast*), enviando a mensagem para todas as Filas vinculadas, independentemente da *Routing Key*.
- *Topic Exchange*: permite um roteamento flexível, baseado na correspondência de padrões (*wildcards*) definidos na *Routing Key*.
- *Headers Exchange*: utiliza atributos no cabeçalho da mensagem para roteamento, oferecendo uma alternativa ao uso da *Routing Key*.

A combinação desses componentes permite que a arquitetura do RabbitMQ estabeleça um fluxo de mensagens robusto e assíncrono. Esta característica é crucial para garantir a confiabilidade operacional, permitindo que o sistema resista a falhas temporárias dos Consumidores.

Para ilustrar o processo, em um exemplo prático implementado, o fluxo de mensagens segue a seguinte sequência: o Produtor envia a mensagem, o *Exchange* realiza o roteamento, e a mensagem é armazenada na Fila denominada "fila_teste". Finalmente, o Consumidor processa o dado e envia a Confirmação (ACK). A Figura 1 apresenta o diagrama deste fluxo.

FIGURA 1. Fluxo de comunicação entre Produtor e Consumidor no RabbitMQ.



Fonte: Elaborada pelo autor.

Para demonstrar a funcionalidade da arquitetura RabbitMQ, foi desenvolvido um exemplo prático focado no envio de uma mensagem no formato JSON e seu consumo por uma aplicação distinta. Este fluxo de execução ilustra o mecanismo de desacoplamento entre o Produtor e o Consumidor.

O fluxo de execução, que utiliza o *Exchange* para roteamento preciso, é definido pelas seguintes etapas sequenciais:

1. A aplicação Produtor cria uma mensagem no formato JSON e a publica no *Exchange*.
2. O *Exchange* executa a lógica de roteamento e direciona a mensagem para a fila denominada "fila_teste".
3. A aplicação Consumidor que está escutando a fila recebe a mensagem.
4. O Consumidor processa o conteúdo da mensagem e, ao final da execução, envia uma Confirmação de Recebimento (ACK) ao RabbitMQ.
5. Em resposta ao ACK, a mensagem é removida da fila, garantindo que não haja reprocessamento indesejado.

A aplicação Produtor (Figura 2a) foi desenvolvida com o objetivo de enviar a mensagem JSON para a fila específica "fila_teste". O código implementa uma sequência de ações necessárias para o estabelecimento da comunicação e publicação:

1. Estabelecimento da conexão com o *broker* RabbitMQ.
2. Criação de um canal de comunicação dedicado.
3. Asseguramento da existência e durabilidade da fila "fila_teste".
4. Envio da mensagem JSON para o *Exchange*, utilizando a chave de roteamento que corresponde à fila de destino.
5. Encerramento formal do canal e da conexão.

Neste exemplo, o *Direct Exchange* foi empregado para garantir que a mensagem fosse diretamente direcionada para uma fila específica que atuou como armazenamento temporário até a requisição pelo Consumidor. Como no exemplo não foi definido um tipo explícito de *Exchange*, o RabbitMQ aplica o *Default Exchange* que é do

tipo *direct*. A *routing key* empregada é o próprio nome da fila, de forma que essa é a configuração mais simples para utilização do RabbitMQ.

A aplicação Consumidor (Figura 2b) foi implementada para escutar ativamente a fila, "fila_teste" e processar as mensagens recebidas. O Consumidor executa o seguinte protocolo de comunicação assíncrona:

1. Estabelecimento da conexão com o broker RabbitMQ.
2. Criação de um canal de comunicação.
3. Asseguramento da existência e durabilidade da fila, "fila_teste" para iniciar a escuta.
4. Consumo das mensagens da fila de forma assíncrona.
5. Processamento de cada mensagem recebida e subsequente envio da Confirmação (ACK) ao broker para a remoção definitiva da mensagem da fila.

A arquitetura implementada, utilizando o *Direct Exchange*, ilustra a capacidade do RabbitMQ de garantir tanto a ordem de entrega quanto a confiabilidade do processamento por meio da mecânica de confirmação.

FIGURA 2a e 2b. Estrutura do código do Produtor e do Consumidor no RabbitMQ

```javascript
// producer.js
const amqp = require('amqplib');

async function sendMessage() {
  try {
    const connection = await amqp.connect('amqp://localhost');
    const channel = await connection.createChannel();
    const queue = 'fila_teste';

    await channel.assertQueue(queue, { durable: true });

    const mensagem = { id: 1, texto: 'Olá RabbitMQ!' };
    channel.sendToQueue(queue, Buffer.from(JSON.stringify(mensagem)));

    console.log('Mensagem enviada:', mensagem);
    await channel.close();
    await connection.close();
  } catch (error) {
    console.error('Erro:', error);
  }
}

sendMessage();
```

```javascript
// consumer.js
const amqp = require('amqplib');

async function receiveMessage() {
  try {
    const connection = await amqp.connect('amqp://localhost');
    const channel = await connection.createChannel();
    const queue = 'fila_teste';

    await channel.assertQueue(queue, { durable: true });

    console.log('Aguardando mensagens...');
    channel.consume(queue, (msg) => {
      if (msg !== null) {
        const mensagem = JSON.parse(msg.content.toString());
        console.log('Mensagem recebida:', mensagem);
        channel.ack(msg); // confirma que a mensagem foi processada
      }
    });
  } catch (error) {
    console.error('Erro:', error);
  }
}

receiveMessage();
```

Fonte: Elaborada pelo autor.

Os resultados obtidos com o exemplo prático demonstram o funcionamento eficiente do RabbitMQ como sistema de mensageria para comunicação assíncrona entre aplicações. A implementação permitiu que uma mensagem JSON fosse enviada de um produtor para uma fila e consumida de forma confiável por um consumidor, sem bloqueios ou perda de dados. Ainda, os resultados corroboram com o que é discutido na literatura sobre mensageria e sistemas distribuídos. Segundo Hohpe e Woolf (2003), a utilização de filas e o *exchanges* garante desacoplamento entre aplicações, permitindo maior escalabilidade e tolerância a falhas, característica também observada no exemplo prático desenvolvido.

Comparando com a comunicação síncrona tradicional, o RabbitMQ dá indícios de ser mais eficiente, pois elimina o bloqueio do produtor e evita dependência direta do tempo de processamento do consumidor. Essa vantagem é essencial em sistemas necessite de alta disponibilidade e confiabilidade. Além disso, a implementação e execução da aplicação prática mostrou a flexibilidade do RabbitMQ, permitindo o envio de mensagens estruturadas (JSON) e sua entrega confiável mesmo em caso de interrupções temporárias do serviço consumidor.

CONCLUSÕES

A realização deste estudo permitiu compreender de forma prática e conceitual a importância da abordagem de mensageria em sistemas distribuídos e a contribuição da ferramenta RabbitMQ para a comunicação assíncrona

entre aplicações. Os objetivos propostos foram alcançados, uma vez que foi possível introduzir os conceitos teóricos de troca de mensagens, descrever a arquitetura do RabbitMQ e desenvolver um exemplo funcional de envio e consumo de mensagens.

Durante o processo de desenvolvimento, foram encontradas algumas dificuldades relacionadas à configuração do ambiente e à compreensão inicial do funcionamento das filas e canais do RabbitMQ. Entretanto, por meio do estudo da documentação oficial e de testes práticos, foi possível superar esses desafios e consolidar o entendimento sobre o fluxo de mensagens assíncronas.

Os resultados obtidos evidenciam as vantagens do uso de mensageria em relação à comunicação síncrona tradicional, principalmente no que se refere ao desacoplamento entre sistemas, à resiliência e à escalabilidade das aplicações. Dessa forma, o RabbitMQ sinaliza ser uma ferramenta eficiente e versátil para aplicações que necessitam de integração entre múltiplos serviços. Este trabalho apresenta parte dos resultados obtidos no projeto de iniciação científica, correspondendo ao estudo introdutório sobre o RabbitMQ. O projeto tem como objetivo comparar o RabbitMQ e o Kafka como soluções de comunicação assíncrona entre aplicações. Como continuidade desta pesquisa, pretende-se aprofundar o domínio do RabbitMQ, avançar no estudo e na implementação do Kafka, analisar suas diferentes abordagens e aplicações e, por fim, realizar um comparativo entre esses brokers em um ambiente distribuído com comunicação assíncrona.

AGRADECIMENTOS

Os autores agradecem ao INSTITUTO FEDERAL DE SÃO PAULO e ao PROGRAMA DE APOIO À CIÊNCIA E TECNOLOGIA (PACTec) pelo apoio acadêmico e financeiro que contribuiu para o desenvolvimento deste trabalho.

REFERÊNCIAS

APACHE SOFTWARE FOUNDATION. Apache Kafka: a distributed event streaming platform. [S.l.: s.n.], [2025?]. Disponível em: https://kafka.apache.org/. Acesso em: 30 out. 2025.

BIRRELL, Andrew D.; NELSON, Bruce J. Implementing remote procedure calls. ACM Transactions on Computer Systems, New York, v. 2, n. 1, p. 39-59, fev. 1984. Disponível em: https://dl.acm.org/doi/10.1145/2080.357392. Acesso em: 30 out. 2025.

COULOURIS, George et al. Sistemas distribuídos: conceitos e projeto. Porto Alegre: Bookman Editora, 2013.

FULL CYCLE. Full Cycle Week: Escalabilidade, Microsserviços e Sistemas. YouTube, 21 jul. 2023. Disponível em: https://www.youtube.com/watch?v=hlcQ_WOyNNs. Acesso em: 30 out. 2025.

HOHPE, G.; WOOLF, B. Enterprise integration patterns: designing, building, and deploying messaging solutions. Boston, MA: Addison-Wesley, 2003.

NEWMAN, Sam. Criando Microsserviços: Projetando sistemas com componentes menores e mais especializados. 2ª ed. São Paulo: Novatec Editora, 2022.

RABBITMQ. RabbitMQ Documentation. [S.l.]: Broadcom, [2025?]. Disponível em: https://www.rabbitmq.com/docs. Acesso em: 30 out. 2025.

RICHARDS, M.; FORD, N. Fundamentals of software architecture. Sebastopol, CA: O'Reilly Media, 2020.

STEEN, M. van; TANENBAUM, A. S. Distributed Systems. 4. ed. [S.l.]: distributed-systems.net, 2023. Disponível em: https://www.distributed-systems.net/index.php/books/ds4/. Acesso em: 30 out. 2025.