UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE SISTEMAS DE INFORMAÇÃO

Paula Zomignani Oliveira

Avaliação Comparativa de Desempenho de Padrões de Projeto para Arquiteturas de Sistemas de Filas de Mensagens

Florianópolis, SC
2024

Paula Zomignani Oliveira

Avaliação Comparativa de Desempenho de Padrões de Projeto para Arquiteturas de Sistemas de Filas de Mensagens

Trabalho de Conclusão de Curso submetido ao Curso de Sistemas de Informação da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Sistemas de Informação.
Orientador: Odorico Machado Mendizabal, Dr.

Florianópolis, SC
2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Oliveira, Paula Zomignani
Avaliação comparativa de desempenho de padrões de projeto para arquiteturas de sistemas de filas de mensagens / Paula Zomignani Oliveira ; orientador, Odorico Machado Mendizabal, 2024.
64 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Sistemas de Informação, Florianópolis, 2024.

Inclui referências.

1. Sistemas de Informação. 2. Sistemas de filas de mensagens. 3. Padrões de projeto. 4. Comparação de desempenho. I. Mendizabal, Odorico Machado. II. Universidade Federal de Santa Catarina. Graduação em Sistemas de Informação. III. Título.

Paula Zomignani Oliveira

Avaliação Comparativa de Desempenho de Padrões de Projeto para Arquiteturas de Sistemas de Filas de Mensagens

O presente trabalho em nível de graduação foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Patricia Della Mea Plentz, Dra.
Instituição UFSC

Bruno Dourado Miranda, Me.
Instituição UFSC

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Bacharel em Sistemas de Informação.

_____

Coordenação do Programa de Graduação

_____

Odorico Machado Mendizabal, Dr.
Orientador

Florianópolis, SC, 2024.

RESUMO

A utilização de sistemas distribuídos tem se tornado extremamente popular e, dentro desse contexto, sistemas de filas de mensagens surgiram como uma solução para enfrentar os desafios relacionados à troca de informações em tais ambientes. Esses sistemas permitem a comunicação entre processos de forma assíncrona, onde processos emissores enviam mensagens para determinada fila e consumidores leem as mensagens que foram postadas na fila. Tal solução pode trazer diversos benefícios aos sistemas que as utilizam, como desacoplamento entre emissores e receptores, facilidade na escalabilidade de serviços, tolerância a falhas e homogeneização das variações de latência, sendo que tais benefícios podem diferir de acordo com o estilo arquitetural de filas de mensagens escolhido. Porém, sistemas de filas de mensagens são de difícil implementação e manutenção devido à vasta gama de requisitos que devem atender. Além disso, a comparação de desempenho entre diferentes padrões de projeto mostra-se desafiadora, pois tais sistemas também são complexos para testar, especialmente em cenários de alta vazão e com requisitos de alta disponibilidade. Dessa forma, este trabalho foi desenvolvido com o objetivo de trazer uma visão sintetizada de diferentes padrões de projeto para sistemas baseados em filas de mensagens, além de realizar uma avaliação comparativa de determinados estilos arquiteturais com ênfase em desempenho.

Palavras-chave: Mensageria. Filas de mensagens. Padrões de projeto. Estilos arquiteturais. Desempenho.

ABSTRACT

The use of distributed systems has become extremely popular, and within this context, message queue services have emerged as a solution to address the challenges associated with information exchange in such environments. Message queueing systems enable asynchronous communication between processes, where producer processes post messages to a designated queue, and consumer processes read the messages that have been posted. This solution can offer numerous benefits to the systems that implement it, such as decoupling of senders and receivers, scalability, fault tolerance, and homogenization of latency variations. Some benefits, however, may vary depending on the architectural style of the message queueing system. Nevertheless, these systems are challenging to implement and maintain due to the wide range of requirements they must meet. Comparing the performance of different design patterns can also be challenging, as these systems are complex to test, particularly in high-throughput scenarios with high availability requirements. In that sense, this study was developed with the objective of providing a synthesized view of different design patterns for message queueing systems, in addition to conducting a comparative evaluation of those design patterns with an emphasis on performance.

Keywords: Messaging. Message Queues. Design Patterns. Architectural Styles. Performance.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A utilização de sistemas distribuídos tem se popularizado com a crescente demanda por aplicações altamente escaláveis, tolerantes a falhas e que sejam capazes de lidar com fluxos de dados intermitentes. Tais sistemas têm, naturalmente, uma necessidade muito grande de trocar informações entre seus componentes e com sistemas externos. Nesse contexto, a mensageria[1] surgiu como uma solução para enfrentar os desafios apresentados por tais ambientes distribuídos.

Filas de mensagens são, então, serviços que permitem a comunicação entre processos de forma assíncrona. Segundo Akbulut e Perros (2019), a comunicação síncrona provida por modelos como o *REST*, apesar de muito útil e popular, não é ideal para todos os cenários. Um exemplo disso são os casos em que um grande volume de dados devem ser trocados entre serviços, onde a comunicação de forma assíncrona se torna mais vantajosa.

Em sistemas de filas de mensagem os serviços produtores enviam mensagens para uma fila e serviços consumidores leem as mensagens que foram postadas na fila, sendo uma mensagem composta por informações armazenadas em um formato acordado entre os serviços (REAGAN, 2018). Tal modelo permite um baixo grau de acoplamento entre os processos, pois como a interação entre eles ocorre por meio de uma fila de mensagens, os mesmos não levam em conta seus pares - apenas o conteúdo das mensagens (REAGAN, 2018). É importante ressaltar que nesses sistemas, um serviço pode ser produtor e consumidor de diferentes filas de mensagens simultaneamente, assim como diversos serviços podem popular ou consumir de uma mesma fila de mensagens.

Sistemas de mensageria podem assumir diversas arquiteturas baseadas em padrões de projeto que trazem diferentes vantagens e objetivos. Alguns destes padrões são Fila de Prioridade, Publicador-Assinante, Verificação de Declaração, Supervisor de Agente do Agendador, entre outros (MICROSOFT AZURE, 2023). Uma gama tão vasta de formatos tornam esses estilos arquiteturais úteis na resolução de problemas comuns em sistemas distribuídos. Alguns ganhos possíveis são a comunicação assíncrona e consequente desacoplamento entre emissores e receptores, facilidade na escalabilidade de serviços, tolerância a falhas e homogeneização das variações de latência (FU; ZHANG; YU, 2020). Assim, filas de mensagens podem ser um artifício vantajoso para os mais diversos sistemas.

No entanto, devido a natureza dos problemas que serviços de mensageria se propõe a solucionar, eles são permeados por desafios. Tais sistemas tendem a ser de difícil implementação e manutenção devido à vasta gama de requisitos que devem

[1] Devido à popularização do termo “Mensageria”, o mesmo será utilizado ao longo desse trabalho para se referir a sistemas de filas de mensagens, também conhecidos como MoM (*middlewares* orientados a mensagens).

atender. Além disso, em razão da variedade de estilos arquiteturais que esses serviços podem assumir, torna-se difícil avaliar qual padrão de projeto é mais adequado para determinadas demandas de um sistema. A comparação de desempenho entre arquiteturas focadas em certo escopo também mostra-se desafiadora, pois além da complexidade de implementação, esses sistemas são complexos para testar, especialmente em situações que simulem uma realidade de alta demanda.

Tendo em vista os desafios levantados, o presente trabalho propõe-se a realizar um levantamento e revisão sobre os padrões de arquitetura comuns para sistemas de filas de mensagem e o contexto em que são utilizados. Propõe-se também a realizar uma análise de desempenho de alguns padrões de projeto para filas de mensagens através da implementação de protótipos de cada um deles e da realização de avaliações experimentais.

1.1 CONTEXTO HISTÓRICO

Seres humanos sempre tiveram a necessidade de se comunicar e, ao longo da história, muitos sistemas de comunicação foram criados com diferentes propósitos. O mesmo é verdade para a tecnologia: sistemas computacionais precisam, desde seus primórdios, trocar informações uns com os outros (JOHANSSON; DOSSOT, 2020). Tal necessidade se intensificou com a avanço da Internet e se torna cada vez mais relevante com o crescimento do uso de arquiteturas distribuídas, utilização de microsserviços e inserção de dispositivos IoT em diversos tipos de tecnologia.

Filas de mensagens são um de vários sistemas de comunicação no mundo da tecnologia, que surgiram para possibilitar uma nova forma de troca de informações entre aplicações. A criação desse modelo de sistema começou em 1983 com o engenheiro Vivek Ranadivé, que se dedicou a procurar uma solução que desempenhasse uma função semelhante a um barramento utilizado em placas eletrônicas, mas em sistemas de *software* (VIDELA; WILLIAMS, 2012). Essa solução deveria permitir conexões com diferentes aplicações de forma simples e habilitar a troca de mensagens entre elas de forma assíncrona, promovendo assim um baixo grau de acoplamento entre as aplicações.

Trazendo uma solução primeiramente focada no setor financeiro, em 1985 Ranadivé disponibilizou um *software bus* onde aplicações poderiam se conectar como consumidoras, estando “inscritas” nos canais de informações que gostariam de receber. Dessa forma nasceu o primeiro *software* de Filas de Mensagens moderno, *The Information Bus (TIB)*, assim como o padrão de projeto Publicador-Assinante que foi adotado (VIDELA; WILLIAMS, 2012).

Rapidamente notou-se que o desacoplamento entre produtor e consumidor proporcionado foi uma grande inovação e que poderia ser utilizada em demais contextos. Assim, a solução criada se expandiu para outras áreas e deu origem vários modelos.

Os benefícios trazidos pelo software criado por Ranadivé não passaram despercebidos e logo grandes empresas dedicaram esforços para desenvolver seus próprios sistemas de mensageria. Em 1993 IBM lançou o IBM *MQSeries*, um conjunto de *softwares* de filas de mensagens, enquanto em 1997 a Microsoft lançou o Microsoft *Message Queue* (MSMQ) (VIDELA; WILLIAMS, 2012). Atualmente existem muitos sistemas de filas de mensagens disponíveis e os mesmos se tornaram indispensáveis para certos modelos de aplicações modernas. Alguns dos programas mais utilizados são Kafka, RabbitMQ, RocketMQ, ActiveMQ e Pulsar (FU; ZHANG; YU, 2020).

1.2 DESEMPENHO E TOLERÂNCIA A FALHAS EM SISTEMAS DE INFORMAÇÃO

Conforme sistemas de informação trouxeram soluções e simplificações benéficas para a o mundo moderno, os mesmos ganharam espaço e se tornaram parte da vida de quase qualquer pessoa. Hoje em dia sistemas de informação estão presentes em muitas áreas: transações bancárias, sistemas médicos, organização de meios de transporte, acesso a serviços governamentais e várias outras esferas da atualidade. Com tamanha dependência desses sistemas, é importante que eles sejam estáveis, ou seja, não apresentem grandes flutuações ou indisponibilidades. Em outras palavras, é necessário que os sistemas tenham um bom desempenho e sejam tolerantes a falhas.

Existem diferentes estratégias para alcançar essas características em um *software*. Uma ferramenta muito utilizada para esse fim são Padrões de Projeto: segundo Gawand, Mundada e Swaminathan (2011) padrões de projeto podem ser definidos como descrições de um conjunto de soluções bem-sucedidas para lidar com um problema conhecido, dentro de um contexto específico. Eles são utilizados para descrever diversos tipos de soluções, e muitos são usados visando a construção de um sistema consistente e robusto - como é o caso de padrões voltados para desempenho e tolerância a falhas.

É importante notar que tais características são difíceis de alcançar em um sistema. Isso porque quanto maior e mais complexo ele for, mais gargalos e erros podem passar despercebidos. Em um programa com alta complexidade e grande volume de código, é desafiador prever e identificar situações que ocasionarão a falhas. Além disso, invariavelmente, soluções que visam tolerância a falhas em um sistema de informação vêm com um custo: recursos que seriam direcionados para o funcionamento regular da aplicação deverão ser utilizados em estratégias para garantir o contínuo funcionamento da mesma em caso de erros. Assim, existem um equilíbrio que deve ser alcançado entre desempenho e tolerância a falhas em um sistema (HANMER, 2007).

1.3 OBJETIVOS

1.3.1 Objetivo Geral

O presente trabalho tem como objetivo principal avaliar padrões de projeto para sistemas que utilizam filas de mensagens com ênfase em desempenho e confiabilidade.

1.3.2 Objetivos Específicos

- Identificar padrões de projeto populares para sistemas de filas de mensagens e seus diferentes escopos;
- Propor cenários de análise dos padrões de projeto levantados;
- Conduzir uma análise quantitativa para comparar o desempenho das diferentes arquiteturas implementadas. Essa análise considerará métricas de vazão, latência e *makespan*, além de diferentes cenários de carga com o objetivo de identificar gargalos e destacar os pontos de maior eficiência do sistema.

1.4 ORGANIZAÇÃO DO TRABALHO

O presente trabalho foi divido em alguns capítulos. Primeiramente, no Capítulo 2, é feita uma revisão sobre Sistemas de Mensageria, seu funcionamento, características básicas e situações onde são comumente utilizados. Além disso, esse capítulo aborda Padrões de Projeto e seus diferentes usos e benefícios em sistemas computacionais. No Capítulo 3, são trazidos os padrões de projeto comuns em sistemas de filas de mensagens, assim como suas arquitetura, funcionamento básico e suas indicações e contraindicações de uso. O trabalho segue com o Capítulo 4 que traz o contexto atual da utilização de padrões de projetos em sistemas de filas de mensagens em uma revisão científica importante sobre o tema. No Capítulo 5 são explanadas as estratégias utilizadas para a implementação dos sistemas utilizados neste trabalho. Finalmente, no Capítulo 6, é feita a descrição dos experimentos realizados, além da exposição e discussão dos resultados obtidos. O trabalho é finalizado no Capítulo 7, com reflexões sobre o desenvolvimento do mesmo e levantamento de pontos relevantes para serem abordados em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

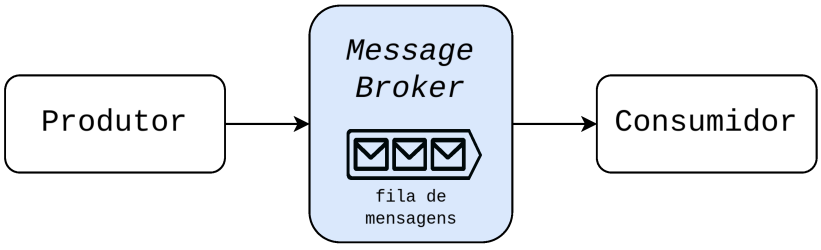
A fundamentação teórica deste trabalho explora conceitos relevantes sobre sistemas de filas de mensagens e padrões de projeto. Dessa forma procura-se proporcionar um embasamento e compreensão adequados sobre tais temas, que são fundamentais para este estudo.

2.1 SISTEMAS DE FILAS DE MENSAGENS

Uma das principais características de filas de mensagens é que elas permitem que processos se comuniquem de forma assíncrona. O funcionamento básico dessa estrutura envolve um serviço que irá adicionar uma mensagem na fila, comumente chamado de serviço produtor, e um serviço que irá periodicamente checar a fila de mensagens procurando por novos eventos, comumente chamado de serviço consumidor (REAGAN, 2018). Devido as mensagens serem enviadas e consumidas diretamente da fila, a aplicação receptora e a aplicação produtora não precisam interagir com ela simultaneamente. Além disso, essas aplicações não precisam ter conhecimento uma da outra, o que pode facilitar a comunicação entre sistemas de diversas formas (REAGAN, 2018).

Uma mensagem, comumente chamada de evento ou pacote, no contexto de filas de mensagens nada mais é do que um conjunto de informações organizadas em um formato determinado e acordado entre os serviços envolvidos. Uma fila de mensagens tipicamente as organiza em uma ordenação FIFO (*First In, First Out*), de forma que o serviço consumidor irá receber os pacotes na ordem em que foram adicionados na fila (REAGAN, 2018). No entanto, os sistemas de mensageria como RabbitMQ, ActiveMQ, entre outros, expandem esses conceitos (HAUNTS, 2015). Esses sistemas, que são *middlewares* referidos como agentes ou *message brokers*, oferecem funcionalidades adicionais em ecossistemas com filas de mensagens (HAUNTS, 2015). A Figura 1 demonstra o funcionamento básico de tais sistemas.

Figura 1 – *Message Broker*



Fonte: Adaptado de Johansson e Dossot (2020)

Em um *middleware* de mensageria, as mensagens adicionadas em uma fila

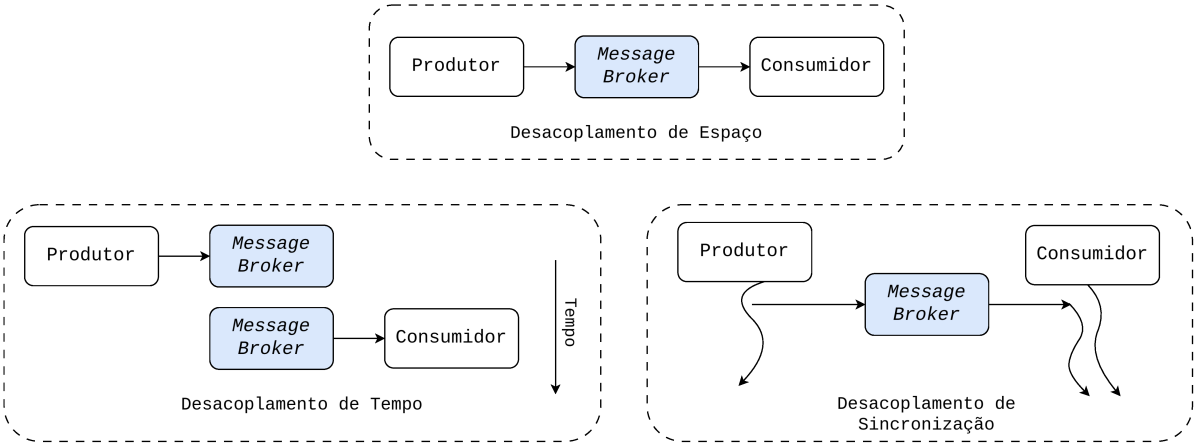
podem ser persistidas a fim de não serem perdidas caso o servidor das mesmas falhe; pode-se também determinar um tempo de expiração para as mensagens (comumente chamado de *time to live*, ou TTL), visando eliminar pacotes muito antigos que não foram consumidos (HAUNTS, 2015). Além disso, *message brokers* possuem *features* de segurança, como permitir acessos apenas a usuários autorizados, filtragem e roteamento de mensagens, determinando para qual fila um dado pacote será enviado, garantia de entrega, onde o produtor recebe uma confirmação do agente de que sua mensagem foi entregue, escalabilidade conforme a demanda do sistema, além de permitir ecossistemas complexos com diversas filas, produtores e consumidores (HAUNTS, 2015).

Existem várias particularidades e funcionalidades em sistemas de filas de mensagens, porém, é importante entender que o agente funcionará como um roteador entre a aplicação e o serviço com quem ela está se comunicando. Assim, a aplicação apenas preocupa-se em entender qual papel está assumindo (produtor ou consumidor) em qual fila e, a partir daí, poderá seguir com a interação com mensagens sem se preocupar com a comunicação com outros serviços (VIDELA; WILLIAMS, 2012).

As filas de mensagens oferecem vários benefícios para os sistemas que as utilizam, incluindo escalabilidade, elasticidade, desacoplamento temporal e espacial e tolerância a falhas. A escalabilidade é alcançada pois múltiplos consumidores podem processar mensagens da mesma fila, o que possibilita que diferentes partes de uma aplicação escalem de forma independente. Isso gera um balanceamento de carga, pois em um pico de demanda, o número de consumidores pode ser aumentado a fim de processar mensagens de determinada fila mais rapidamente (REAGAN, 2018). A elasticidade, por sua vez, permite que as filas de mensagens absorvam picos de trabalho, acumulando mensagens na própria fila e evitando o colapso da aplicação durante surtos de tráfego; dessa forma, é garantido que as tarefas sejam eventualmente concluídas, mesmo quando os recursos imediatos são insuficientes (REAGAN, 2018).

O desacoplamento temporal, espacial, de sincronização e a tolerância a falhas também são benefícios das filas de mensagens (REAGAN, 2018). É importante notar que existem diferentes características trazidas por cada tipo de desacoplamento, conforme demonstrado na Figura 2. No desacoplamento de espaço as partes envolvidas em uma dada interação não precisam ter ciência uma da outra - isso permite substituições e atualizações de segmentos do sistema sem afetar o restante do mesmo, tornando-o mais adaptável e resiliente; já o desacoplamento temporal permite que emissores e receptores operem de forma assíncrona, ou seja, os serviços envolvidos não precisam estar ativamente participando da interação ao mesmo tempo; e, finalmente, no desacoplamento de sincronização as partes não ficam bloqueadas até receberem um retorno de uma requisição enviada (EUGSTER et al., 2003). Já a tolerância a falhas é inerente desse desacoplamento: se um receptor falhar, as mensagens

Figura 2 – Tipos de Desacoplamento



Fonte: Adaptado de Eugster et al. (2003)

enfileiradas são preservadas e processadas quando o consumidor voltar a operar, garantindo o contínuo funcionamento do sistema (REAGAN, 2018).

2.1.1 Tecnologias de Filas de Mensagens

Atualmente existem muitos *message brokers* disponíveis no mercado, sendo que alguns dos mais conhecidos são Apache Kafka, ActiveMQ, RabbitMQ e soluções de filas dentro da plataforma Azure (REAGAN, 2018). A Azure é uma plataforma abrangente baseada em nuvem que oferece serviços integrados, como análises, bancos de dados e mobilidade, visando facilitar o desenvolvimento rápido. Dentro do universo de filas de mensagens, Azure oferece dois tipos de filas: *Service Bus Messaging Queues*, que possuem diversas *features* como *time to live* (TTL) e garantia de ordenação das mensagens, e *Azure Storage Queues*, que possuem menos *features* mas fornecem a possibilidade de filas de tamanho muito superior (REAGAN, 2018).

Já Kafka é uma plataforma de *streaming* que tem como seu principal foco o processamento e transmissão de eventos com alta vazão. A plataforma oferece APIs para diversas linguagens e processamento de fluxos de dados por meio do Kafka Streams e KSQL, permitindo filtragem e agregação de fluxos de dados. Além disso, Kafka inclui a API *Connect* para integração com bancos de dados e pontos de extremidade, tornando-se uma boa ferramenta para transformar dados estáticos em fluxos dinâmicos de eventos e conectar aplicações distintas dentro de uma organização (STOPFORD, 2018).

É importante notar que embora possam existir enfoques diferentes entre essas plataformas, todas se propõem a prover, de alguma forma, um *middleware* que será responsável pela criação, manutenção e orquestração de filas de mensagem.

2.2 RABBITMQ

Para a implementação desse trabalho foi utilizado o *message broker* RabbitMQ. Tal *broker* foi escolhido devido a sua versatilidade e funcionamento simples e intuitivo (JOHANSSON; DOSSOT, 2020). Por existir desde 2006, o RabbitMQ é um agente consolidado e que suporta muitas integrações para desenvolvimento por meio de bibliotecas de cliente, tornando-o flexível e adaptável. Ele oferece recursos como confiabilidade, roteamento complexo de mensagens, alta disponibilidade, ferramentas de gerenciamento, e é compatível com plataformas variadas, incluindo Windows, Linux, Unix, Mac OS, e plataformas de nuvem como Amazon EC2 e Microsoft Azure (JOHANSSON; DOSSOT, 2020).

O sistema RabbitMQ é construído na linguagem de programação Erlang e implementa o padrão AMQP (*Advanced Message Queuing Protocol*) (VIDELA; WILLIAMS, 2012). Esse protocolo propõe-se a conectar dinamicamente informações em tempo real de qualquer publicador a qualquer consumidor interessado, por meio de um barramento de software. Ou seja, é um protocolo padrão aberto que define como sistemas podem trocar mensagens através de um conjunto de regras que devem ser seguidas pelas partes envolvidas na comunicação. Além de definir a interação que ocorre entre um consumidor/produtor e um broker, ele também define a representação das mensagens e comandos que estão sendo trocados. O AMPQ possui definições sobre garantia de entrega, enfileiramento, roteamento das mensagens, entre outros (JOHANSSON; DOSSOT, 2020).

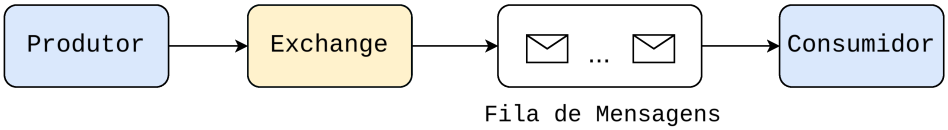
O *broker* é implementado em Erlang devido à linguagem se adequar a sistemas distribuídos, sendo amplamente utilizada na área de telecomunicações, por exemplo. Outro ponto relevante é a alta compatibilidade da linguagem com variados sistemas. Além disso o RabbitMQ faz uso das funcionalidades e tecnologias intrínsecas de Erlang, como *clustering* e utilização da base de dados Mnesia, parte do ecossitema da linguagem, para a persistência de dados do agente. (JOHANSSON; DOSSOT, 2020).

Os elementos básicos do RabbitMQ são: Produtores, Consumidores, Filas de Mensagens e Exchanges. Por “produtor” pode-se entender qualquer programa que envie mensagens para o agente. Os consumidores, por outro lado, são programas que leem (consomem) mensagens das filas. Filas de Mensagens são as estruturas que armazenam temporariamente pacotes sendo trocados entre programas que se comunicam pelo RabbitMQ - essas estruturas são limitadas pela memória e pelos limites de disco do *host*. Um ou muitos produtores podem enviar mensagens que vão para uma mesma fila, e muitos ou um único consumidor pode ler os dados de uma fila (RABBITMQ TUTORIALS, 2024).

No entanto, a proposta do modelo do RabbitMQ é que o serviço produtor nunca envie pacotes diretamente para uma fila de mensagens. Em vez disso, o produtor envia suas mensagens para uma estrutura chamada Exchange. Um *exchange* recebe

mensagens dos produtores e, do outro lado, as encaminha para filas. O mesmo pode ser configurado para descartar determinadas mensagens, enviar as mensagens para todas as filas conectadas a ele (padrão *fanout*), enviar mensagens para apenas filas específicas identificadas através de *bindings* (padrão *direct*), entre outras particularidades (RABBITMQ TUTORIALS, 2024).

Figura 3 – Funcionamento básico do RabbitMQ



Fonte: Adaptado de Johansson e Dossot (2020)

Assim, o funcionamento básico do RabbitMQ ocorre conforme ilustrado na Figura 3: Um produtor envia mensagens para um *exchange*, que as encaminha para a fila (ou filas) de destino. As mensagens são então armazenadas na fila e consumidas por um serviço consumidor. Ao serem consumidas, as mensagens são retiradas da fila (RABBITMQ TUTORIALS, 2024).

2.3 PADRÕES DE PROJETO

Como citado previamente, um padrão de projeto descreve uma estrutura recorrente de componentes que se propõe a resolver um problema conhecido em um contexto específico (BUSCHMANN et al., 1996). De forma mais abrangente, um padrão de projeto propõe uma solução generalista para solucionar determinado tipo de problema.

Padrões de projeto, então, podem ser utilizados com diversas finalidades. Isso é exemplificado por Buschmann et al. (1996), que divide padrões de projeto em algumas categorias, de acordo com a área em que os mesmos se propõe a trazer soluções. São elas: decomposição estrutural, onde padrões trazem uma forma de decompor componentes complexos em partes cooperantes; organização de trabalho, onde descrevem como partes trabalham juntas para resolver problemas complexos; controle de acesso, onde padrões descrevem privilégios de acesso entre componentes; gerenciamento, onde é descrita a gestão de componentes do sistema; e comunicação, onde os padrões descrevem a comunicação e troca de dados entre partes do sistema.

Como exemplo da utilização de padrões de projeto é possível citar alguns padrões muito utilizados na área de tecnologia, especificamente no desenvolvimento de *softwares* orientados a objetos, chamados de *Gang of Four design patterns*. Segundo Hussain, Keung e Khan (2017), os pesquisadores Gamma, Helm, Johnson, e Vlissides introduziram um catálogo com 23 padrões de projeto que descrevem soluções para problemas específicos oriundos da orientação a objetos. Tais padrões são utilizados para

tornar o sistema mais flexível, legível e reutilizável. Dessa forma, o desenvolvimento se torna mais eficiente: códigos com boa organização, de fácil leitura e manutenção tendem a necessitar de menos intervenções e refatorações, ou seja, tornam-se códigos com maior nível de qualidade (HUSSAIN; KEUNG; KHAN, 2017).

De forma parecida, algumas orientações de padrões de projeto são descritos pelos princípios SOLID. Trazendo diretrizes para o desenvolvimento de programas, esses princípios foram introduzidos por Robert Martin e fornecem benefícios relacionados a qualidade de *softwares*. São esses princípios: *The Single Responsibility Principle*, *The Open Close Principle*, *The Liskov Substitution Principle*, *The Interface Segregation Principle* e *The Dependency Inversion Principle*. Além dos benefícios de legibilidade e manutenção, nota-se que a utilização desses princípios tornam o código mais escalável, algo muito valioso para demandas atuais de tecnologia (SINGH; HASSAN, 2015).

Fica claro, então, que padrões de projeto são ferramentas eficientes para lidar com desafios recorrentes e para aumentar qualidade dentro de diferentes áreas da tecnologia da informação. No próximo capítulo serão detalhados alguns padrões de projeto utilizados em sistemas de filas de mensagens, visando proporcionar uma compreensão mais profunda sobre formas para estruturar e otimizar a comunicação assíncrona entre componentes de um sistema.

3 PADRÕES DE PROJETO PARA ARQUITETURAS DE FILAS DE MENSAGENS

Neste capítulo serão explorados alguns padrões de projeto voltados para comunicação - mais especificamente, o capítulo focará em padrões de projetos utilizados para solucionar desafios encontrados em sistemas de filas de mensagens. São eles: Publicador-Assinante, Consumidores Concorrentes, Fila de Prioridade e Verificação de Declaração.

3.1 PADRÃO PUBLICADOR-ASSINANTE (*PUBLISHER-SUBSCRIBER*)

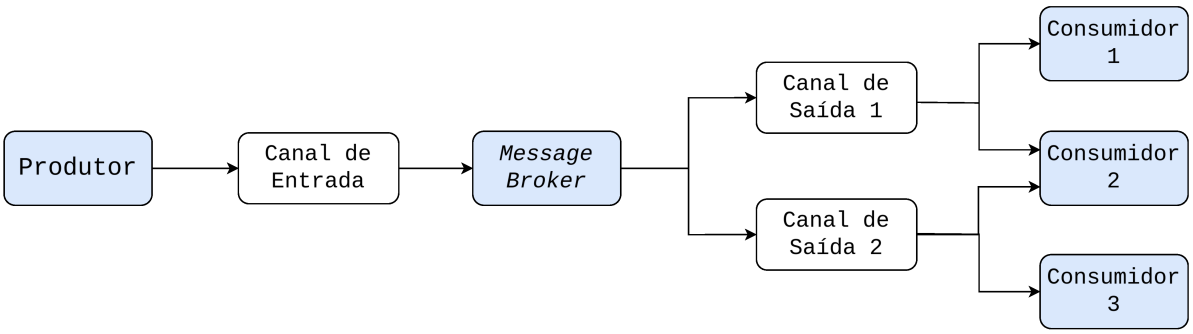
A ideia por trás do padrão de projeto Publicador-Assinante, também chamado de *Publisher-Subscriber*, é permitir que uma aplicação disponibilize mensagens para diversos consumidores com diferentes necessidades. Além disso, a utilização do padrão permite que os serviços produtores disponibilizem suas mensagens sem necessidade de distinção ou conhecimento sobre seus consumidores. O padrão busca alcançar esse objetivo de forma assíncrona e desacoplada, sem levar em consideração a informação almejada por cada um dos receptores, e sem que esses recebam mensagens com conteúdos que não os interessam (MICROSOFT AZURE, 2023). Esse foi o primeiro padrão utilizado em um sistema de filas de mensagens. Segundo Videla e Williams (2012) a primeira versão de um software de mensageria criado por Ranadivé em 1985 utilizava o conceito de um *software bus* onde aplicações podiam se inscrever para receber as informações que desejam, ou seja, o padrão de projeto *Publisher-Subscriber*.

O funcionamento básico de tal padrão é que o serviço produtor envia suas mensagens para um canal específico, chamado de canal de entrada, sem fazer nenhuma distinção em relação a qual serviço irá consumir tal informação. O evento é então acessado por um *middleware* referido como agente, ou *message broker*. Esse *middleware* é a peça do sistema responsável por disponibilizar as mensagens recebidas do produtor nos canais adequados. Diferentes mensagens podem ser destinadas a diferentes canais, chamados de canais de saída, e o *message broker* deverá roteá-las de acordo com o tipo de informação adequada para cada um dos canais de saída (MICROSOFT AZURE, 2023). Os consumidores, por sua vez, se inscrevem apenas nos canais dos quais têm interesse. Dessa forma, consumindo apenas dos canais de saída com conteúdo relevante para seu funcionamento, as aplicações consumidoras não precisam lidar com eventos voltados para contextos dos quais não fazem parte (BELLAVISTA; CORRADI; REALE, 2014).

A Figura 4 demonstra esse funcionamento. Nela, é possível observar que o serviço produtor (ou *Publisher*) posta seus eventos no canal de entrada sem fazer nenhuma distinção em relação a qual serviço irá consumir tal mensagem. O *message broker*, por sua vez, lê as mensagens postadas no canal de entrada e as encaminha para o canal de saída adequado. Os serviços consumidores (ou *Subscribers*) então

consomem exclusivamente as mensagens que são postadas nos canais de saída de seu interesse, sem ter nenhum contato com os demais eventos postados. É importante ressaltar que todos os serviços consumidores de determinado canal consumirão todas as mensagens postadas no mesmo. Além disso, um mesmo serviço pode consumir mensagens de mais de um canal de saída, como é o caso do Consumidor 2 da imagem. Um mesmo serviço também pode ser produtor de certos canais e consumidor de outros.

Figura 4 – Padrão de Projeto Publicador-Assinante



Fonte: Adaptado de Microsoft Azure (2023)

Como mencionado previamente, o padrão Publicador-Assinante permite que serviços se inscrevam em tópicos específicos, recebendo apenas mensagens pertinentes a esses tópicos, e elimina a necessidade de discernimento entre consumidores por parte do produtor (MICROSOFT AZURE, 2023). Outro ponto relevante para esse padrão é o desacoplamento entre subsistemas - mais especificamente, neste padrão é possível obter desacoplamento de espaço, tempo e desacoplamento de sincronização (EUGSTER et al., 2003).

Esses desacoplamentos, além de removerem responsabilidades de discernimento entre diferentes tipos de mensagens e receptores dos serviços, permitem que componentes do sistema sejam facilmente adicionados ou substituídos sem necessidade de adaptação por parte das demais aplicações. Assim, o sistema como um todo se torna muito mais flexível e adaptável, tendo menores chances de tornar-se obsoleto (SCHMIDT; O'RYAN, 2002). Tal flexibilidade também torna o sistema mais escalável, sendo possível adicionar produtores e consumidores com mínimas interferências, além de criar diversos canais e dividir responsabilidades entre diferentes partes da aplicação.

Além disso, como o *middleware* é responsável por disponibilizar as mensagens para os consumidores, os serviços produtores podem gastar mínimos recursos apenas para enviar a mensagem para um canal de entrada e ter seu foco majoritário em suas demais responsabilidades. A disponibilização de mensagens para os consumidores pode ocorrer em um formato de *pull*, onde os *subscribers* consomem mensagens con-

forme sua necessidade, ou *push*, onde os eles recebem as mensagens do *middleware* assim que as mesmas estão disponíveis. Filas de mensagens, que serão aprofundadas nesse trabalho, utilizam um sistema de *pulling* (EUGSTER et al., 2003).

Outro ponto relevante dessa arquitetura é a tolerância a falhas. Devido ao baixo grau de acoplamento e natureza assíncrona da comunicação entre produtores e consumidores, no caso de falha de um serviço, os demais poderão seguir funcionando normalmente (MICROSOFT AZURE, 2023).

No entanto, existem alguns pontos que precisam ser considerados ao seguir com tal arquitetura: como serão realizadas as inscrições de consumidores em diferentes tópicos de mensagens, como será feito o tratamento de mensagens duplicadas ou malformadas e qual o tempo de expiração das mensagens (se for existir algum). Esse padrão de projeto é indicado para situações em que um serviço produz informações relevantes para diversos outros, mas não precisa ter ciência de quais informações são consumidas pelos demais serviços e como são utilizadas (BUSCHMANN et al., 1996). Em contrapartida, tal *design* não é adequado para sistemas que necessitam de comunicação síncrona ou de duas vias entre seus componentes.

3.2 PADRÃO CONSUMIDORES CONCORRENTES (*COMPETING CONSUMERS*)

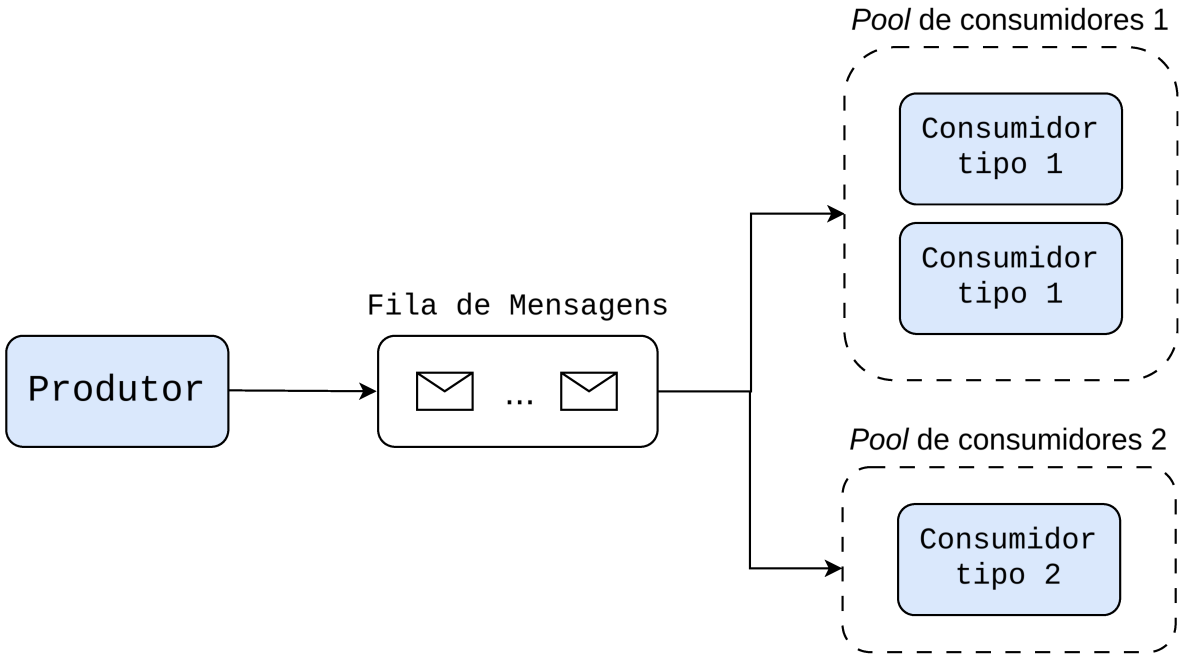
É comum que aplicações possuam um diferente número de requisições ao longo do dia e que existam alguns momentos de pico. Caso a aplicação não esteja preparada para lidar com um grande número de mensagens, essa situação pode gerar um gargalo no sistema - isso pode ocorrer devido a aumentos de demanda, a disponibilização de mensagens ser mais rápida do que seu processamento, entre outros. Um dos padrões de projeto que se propõe a auxiliar na vazão de elevados números de eventos em um momento específico e a lidar com flutuações de demandas é o de Consumidores Concorrentes (KHOMH; ABTAHIZADEH, 2018).

Nesse padrão, também chamado de *Competing Consumers*, as instâncias da aplicação geram mensagens que são postadas em uma fila única e múltiplos serviços são alocados em um mesmo *pool* para consumir e processar (de forma equivalente) as requisições desse canal (HOHPE; WOOLF, 2002). Assim, qualquer um dos serviços consumidores pode processar mensagens de qualquer uma das instâncias da aplicação que esteja alimentando dada fila - ou seja, os serviços consumidores irão competir para acessar o evento. Dessa forma, com consumidores concorrendo para realizar o processamento de mensagens de forma paralela, evitam-se gargalos no sistema (HOHPE; WOOLF, 2002).

Essa dinâmica é demonstrada na Figura 5, onde os consumidores dentro de cada *pool* irão competir pelo acesso e processamento das mensagens postadas na fila, e cada uma das mensagens será processada por apenas um consumidor do *pool*. O número de consumidores dos *pools* pode variar conforme necessidade. Nota-se que

esse padrão de projeto difere do padrão Publicador-Assinante pois, ao contrário do cenário apresentado aqui, no padrão Publicador-Assinante todos os consumidores de uma dada fila recebem e processam cópias de todas as mensagens postadas na fila (MICROSOFT AZURE, 2023).

Figura 5 – Padrão de Projeto Consumidores Concorrentes



Fonte: Adaptado de Microsoft Azure (2023)

Assim, o paralelismo no processamento de mensagens obtido por esse padrão é um ponto relevante de seu funcionamento. O controle de qual serviço consumidor irá processar cada mensagem da aplicação pode variar - a distribuição das mensagens pode ser feita pelo próprio canal de mensagens, pode ser implementado um algoritmo para distribuição das mensagens ou os consumidores podem ser encarregados de tentar acessar as mensagens postadas na fila. Nota-se, no entanto, que a última opção acarretará em uma perda de desempenho se comparada as outras opções (HOHPE; WOOLF, 2002).

Uma vantagem deste padrão é que o sistema torna-se mais tolerante a variações de carga, pois o número de serviços consumidores pode ser alterado conforme demanda (KHOMH; ABTAHIZADEH, 2018). Pode-se inclusive utilizar uma solução auto-escalável, otimizando a utilização de recursos. Além disso, esse padrão de projeto tende a fornecer um tempo de resposta e processamento mais homogêneo, pois é preparado para lidar com flutuações de carga (MICROSOFT AZURE, 2023). Outro ponto positivo do padrão é que com a falha de um serviço consumidor, ainda existirão outros serviços que poderão fazer o processamento das mensagens. Assim, garante-se que esse cenário não causará o bloqueio do sistema, tornando-o mais tolerante a

falhas e confiável (KHOMH; ABTAHIZADEH, 2018).

Nota-se que a ordem das mensagens pode ser alterada nesse padrão, tornando-o pouco recomendado para cenários onde eventos precisam ser processados sequencialmente. Além disso, ele só é aplicável para serviços que geram mensagens que podem ser processadas de forma assíncrona e paralela (MICROSOFT AZURE, 2023).

3.3 PADRÃO FILA DE PRIORIDADE (*PRIORITY-QUEUE*)

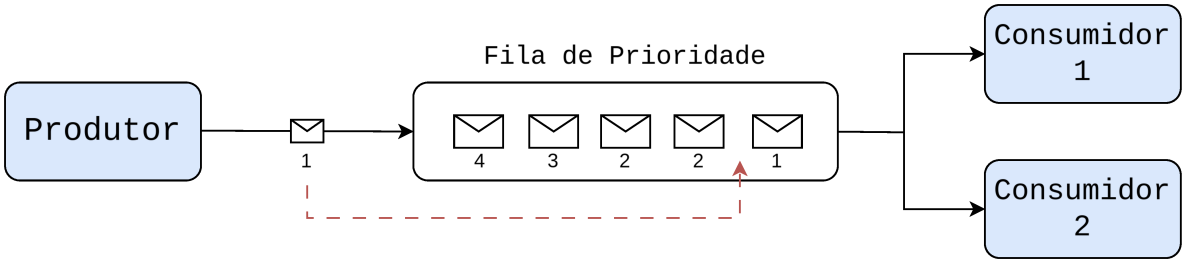
Existem situações em que consumir as mensagens na ordem em que elas chegam na fila (ordem *FIFO*) não é o mais benéfico para o sistema. Isso porque pode existir uma hierarquia entre as mensagens produzidas, onde algumas são mais relevantes e urgentes do que outras. O padrão de projeto Fila de Prioridade, ou *Priority Queue*, pode ser adequado para esses cenários. Nele as mensagens enviadas para uma fila são ordenadas de acordo com a prioridade associada a cada uma delas (RÖNNGREN; AYANI, 1997).

Esse padrão é indicado para situações em que certos tipos de mensagens devem ter prioridade para serem processados pelos serviços consumidores. Nesse caso, mesmo que uma mensagem prioritária seja enviada à fila depois de uma mensagem menos importante, ela deve ter preferência para ser consumida (MICROSOFT AZURE, 2023). O padrão Fila de Prioridade é amplamente utilizado em uma gama de aplicações como sistemas operacionais, sistemas em tempo real, entre outros (RÖNNGREN; AYANI, 1997).

Entende-se que existem duas operações básicas em um algoritmo de Fila de Prioridade: a operação de adicionar (ou enfileirar) um elemento x, com prioridade $x.p$ na fila, e a operação de recuperar um elemento da fila. O elemento recuperado deve ser sempre o com maior prioridade entre os presentes na fila (DRAGICEVIC; BAUER, 2008). Esse comportamento é ilustrado na Figura 6: nela, um serviço produtor envia uma nova mensagem para a fila de prioridade. A mensagem enviada tem prioridade 1, então a mesma é inserida na fila a frente de todas as mensagens com prioridade 2, 3 e 4; além disso, a nova mensagem é inserida atrás da mensagem com prioridade 1 que já havia sido introduzida na fila previamente, respeitando a ordem FIFO entre mensagens com a mesma prioridade.

Filas de prioridade utilizam uma escala para organizar as mensagens recebidas, que define a ordem de importância de cada categoria de mensagem - essa escala chama-se “escala de prioridade” (SHAVIT; ZEMACH, 1999). O padrão de projetos Fila de Prioridade pode ter um número fixo de pontos em sua escala de prioridades, conhecido em inglês como *bounded range priority queue*, ou essa escala pode ser flexível. Porém, padrões com escala fixa tem um funcionamento e implementação mais simples (SHAVIT; ZEMACH, 1999). Um dos grandes desafios desse padrão de projetos é sua complexa implementação, especialmente em cenários de concorrência

Figura 6 – Padrão de Projeto Fila de Prioridade



Fonte: Adaptado de Microsoft Azure (2023)

(DRAGICEVIC; BAUER, 2008). Este trabalho não focará na implementação básica desse padrão, mas sim em sua utilização em sistemas de mensageria.

Em sistemas de mensageria existem algumas possibilidades para utilização de um padrão de Fila de Prioridades: uma delas é a utilização de MoMs que suportam filas de mensagem com prioridade, onde o serviço que está postando a mensagem na fila a identifica de acordo com seu nível prioritário, e as mensagens já postadas na fila são reorganizadas para respeitar tal nível. Esse cenário pode possibilitar a redução de recursos alocados para o processamento de mensagens, mas ainda garante a rápida leitura das mensagens mais importantes (MICROSOFT AZURE, 2023).

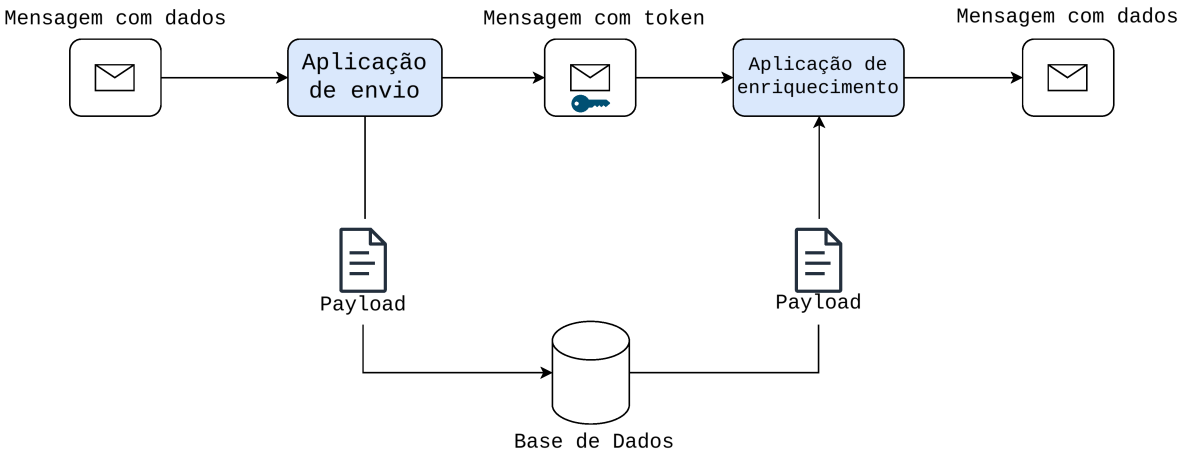
Já em sistemas que não possuem suporte para filas de mensagem com prioridade, pode-se adotar uma arquitetura com múltiplas filas: dessa forma, cada nível de prioridade terá sua própria fila e os serviços consumidores lerão primeiro as mensagens das filas mais prioritárias, para depois seguir para as demais filas. Além disso, pode-se alocar um maior número de serviços consumidores para filas com alta prioridade, fornecendo uma forma adicional de acelerar o processamento de mensagens mais relevantes (MICROSOFT AZURE, 2023).

3.4 PADRÃO DE VERIFICAÇÃO DE DECLARAÇÃO (*CLAIM-CHECK*)

O padrão de arquitetura de Verificação de Declaração, ou *Claim-Check*, foi criado para aplicações que devem enviar, receber e processar mensagens muito grandes. Com o intuito de manter a performance e consumo de recursos desses sistemas otimizados, tal padrão propõe como solução a divisão dessas mensagens em um *claim-check* e um *payload*, que serão manejados de formas distintas (MICROSOFT AZURE, 2023).

A Figura 7 demonstra o funcionamento desse padrão. Primeiramente, a mensagem é recebida por uma aplicação de envio, que tem o papel de gerar uma chave única para a mensagem. Em seguida, esse componente extrai o corpo da mensagem (*payload*) e o armazena em uma memória persistente, como por exemplo um banco de dados, utilizando a chave gerada como identificador. Com os dados armazenados

Figura 7 – Padrão de Verificação de Declaração



Fonte: Adaptado de Microsoft Azure (2023)

em uma memória externa, a aplicação de envio adiciona a informação da chave na mensagem (verificação de declaração) e a envia para uma fila ou para a aplicação enriquecedora. Em estruturas de mensagens mais simples, onde não existem *headers* com informações relevantes, a chave também pode ser enviada diretamente à fila. Ao receber a mensagem, a aplicação enriquecedora utiliza a informação de chave para acessar os dados que foram armazenados em uma memória externa e enriquecer a mensagem com os mesmos - a partir desso momento, a mensagem pode ser processada normalmente (HOHPE; WOOLF, 2002). Um ponto relevante deste padrão é que, após recuperar o *payload* e reinseri-lo na mensagem, o serviço enriquecedor deve retirar a informação de chave que havia sido adicionada à mensagem anteriormente (RITTER, 2014).

É possível fazer diversas analogias que podem auxiliar no entendimento desse padrão. Uma delas é de um restaurante que armazena os casacos de clientes em um cômodo próprio, pois assim os mesmos não precisam carregar um item que não será utilizado naquele momento. Para isso, um funcionário do local coleta os casacos dos clientes quando os mesmos chegam e os entrega uma senha única. O casaco é, então, armazenado em um compartimento que faz referência à mesma senha que foi entregue ao cliente. Ao finalizar a refeição, antes de ir embora do restaurante, o cliente entrega sua senha única ao funcionário, que irá coletar o casaco do local que faz referência a mesma senha e devolvê-lo ao cliente. Assim como no padrão *claim-check*, um item que não será necessário em dado momento é armazenado em um local externo e recuperado através de uma senha única no momento em que volta a ser necessário (DUELL; GOODSEN; RISING, 1997).

Essa solução é indicada para situações onde as mensagens carregam *payloads* grandes. Apesar de os dados da mensagem precisarem ser recuperados pela apli-

cação que irá processar ou disponibilizar a mesma, essa solução ainda oferece uma possível vantagem no transporte dos dados. Isso porque enviar mensagens grandes, como imagens, é custoso para o sistema e pode ocasionar queda de desempenho no envio das mesmas. Também pode-se extrapolar os limites ou sobrecarregar o *middleware* de envio de mensagens ocasionando falhas (MICROSOFT AZURE, 2023). Além disso, existe a possibilidade da mensagem passar por diversos componentes do sistema que não utilizam seus dados ou ainda ser encriptada para envio. Esses passos exigem um elevado poder de processamento que pode ser economizado ao armazenar o corpo da mensagem em uma estrutura externa e recuperá-lo apenas quando necessário (HOHPE; WOOLF, 2002).

É indicado, no entanto, que os *payloads* sejam deletados da memória externa após certo período, visando não escalar demasiadamente os custos da solução. Além disso, procurando manter a latência do serviço otimizada, é indicado que essa solução possa ser aplicada apenas para mensagens que excedem determinado tamanho (MICROSOFT AZURE, 2023).

3.5 OUTROS PADRÕES

Além dos padrões de projeto discutidos anteriormente, existem diversos outros utilizados para diferentes finalidades no contexto de filas de mensagens. Abaixo são citados e brevemente explicados alguns desses padrões.

- Padrão Solicitação/Resposta Assíncrona (*Asynchronous Request-Reply*): esse padrão é utilizado para situações em que um serviço não tem a possibilidade de aguardar a resposta de uma solicitação feita a um serviço externo, como uma API. Nesse caso, a API retorna imediatamente uma resposta informando que a chamada realizada foi bem sucedida e, após realizar o processamento necessário, envia a mensagem contendo o resultado para uma fila de mensagens (MICROSOFT AZURE, 2023);
- Padrão de Supervisor de Agente do Agendador (*Scheduler Agent Supervisor*): esse padrão busca garantir a integridade do sistema como um todo. Nele, uma aplicação que possui diversas tarefas deve concluir cada uma delas com sucesso ou deve tentar resolver as falhas que ocorram ao longo do processo. No caso de falhas mais permanentes, a aplicação deve ser capaz de restaurar o sistema para um estado consistente e garantir a integridade de toda a operação - para isso, um Supervisor monitora o *status* de cada tarefa agendada pelo Agendador (MICROSOFT AZURE, 2023);
- Comboio Sequencial (*Sequential Convoy*): um padrão de projeto que propõe uma forma de processar uma sequência de mensagens sem alterar sua ordem, porém de forma que seja possível aumentar a vazão do sistema. Nele,

as mensagens enviadas são relacionadas a categorias dentro do sistema de enfileiramento e cada consumidor irá processar eventos apenas de uma categoria, uma mensagem por vez (MICROSOFT AZURE, 2023);

- Coreografia (*Choreography*): padrão utilizado para distribuir a lógica de fluxo de trabalho entre componentes dentro de um sistema. Nele, as mensagens recebidas passam por um orquestrador, que será responsável por delegá-las aos respectivos serviços. Cada serviço, pos sua vez, será responsável apenas por processar as mensagens, sem ter conhecimento do fluxo de trabalho do sistema em geral (MICROSOFT AZURE, 2023).

Esses padrões, no entanto, não serão abordados no presente estudo.

4 TRABALHOS RELACIONADOS

O presente capítulo tem como objetivo fornecer uma análise de estudos anteriores que estão relacionados ao tema deste projeto. Com isso, procura-se promover uma maior compreensão do estado atual da arte na área de padrões de projeto e sistemas de filas de mensagens. As investigações prévias são examinadas visando destacar as metodologias, abordagens teóricas e resultados relevantes encontrados nos trabalhos analisados, além de pontos de convergência e divergência com o presente estudo.

4.1 *DESIGN PATTERNS* PARA *DATA SCIENCE*

A monografia "*Design Patterns* para *Data Science*", escrita por Ferreira (2021), salienta o fato de que estudos a respeito de padrões de projeto são usuais na área de engenharia de *software* e arquitetura de sistemas, mas ainda pouco difundidos dentro do campo da ciência de dados. Dessa forma, o autor se propõe a preencher essa lacuna trazendo uma revisão sobre alguns padrões de projeto voltados para a área de ciência de dados, exemplos de utilização dos mesmos e uma análise qualitativa dos padrões elencados.

Para isso, a monografia traz uma análise de 7 padrões de projeto: Qualidade de Dados, Modelo de Dados Canônico, Controle de Versionamento, Identificador Único, Fluxo de Dados Contínuo, Caixa Preta de APIs e Remoção de Códigos Experimentais Mortos. Para cada um dos padrões elencados é trazido o problema que o mesmo se propõe a resolver, a solução proposta pelo padrão, suas consequências positivas e negativas para a aplicação e alguns exemplos de utilização do mesmo.

Após prover uma revisão teórica de cada um dos padrões de projeto selecionados, o autor traz uma análise qualitativa dos mesmos através de um questionário *online* respondido por 17 profissionais da área. Esse questionário, através de 23 perguntas, buscou mapear dados dos respondentes, sua familiaridade com e entendimento dos padrões de projeto elencados e, finalmente, sua avaliação sobre o quanto tais *design patterns* podem auxiliar a solucionar problemas com que os profissionais se deparam no dia a dia. Dos respondentes, apenas 52,9% afirmaram ter familiaridade com a utilização de padrões de projeto na área de ciência de dados. Apesar da familiaridade restrita com os padrões, 88,2% dos respondentes concordam totalmente ou parcialmente com o fato de que tais *designs* relacionam-se com problemas que os mesmo se deparam em suas atividades profissionais. Além disso, uma porção majoritária dos respondentes afirma que o conjunto de padrões de projeto apresentados são úteis para suas atividades de *data science* (94,1%) e que usariam esses padrões em suas atividades (100%). Assim, é evidente que o trabalho apresenta uma boa aceitação dos padrões de projeto por ele trazidos.

Nota-se que da mesma forma que no presente trabalho, Ferreira (2021) apre-

senta uma revisão geral de alguns padrões de projeto existentes em sua área de interesse e os avalia. Salienta-se, no entanto, que as áreas de foco de cada um dos trabalhos divergem: este estudo é direcionado a padrões de projeto em sistemas de filas de mensagens, enquanto o trabalho "*Design Patterns* para *Data Science*" busca trazer padrões relacionados a área de ciência de dados. Além disso, o método de avaliação selecionado pelo autor diverge do utilizado aqui: o autor optou por realizar uma análise qualitativa dos padrões apresentados - para isso, usa um questionário respondido por profissionais da área. Em contrapartida, este trabalho foca em trazer uma análise quantitativa dos padrões de projeto elencados através da implementação e testagem dos mesmos, e da subsequente avaliação de métricas de desempenho como vazão, latência e *makespan*.

4.2 *DESIGN PATTERNS TO IMPLEMENT SAFETY AND FAULT TOLERANCE*

O artigo intitulado *Design Patterns to Implement Safety and Fault Tolerance*, escrito por Gawand, Mundada e Swaminathan (2011) traz uma análise de Padrões de Projeto utilizados com o viés de adicionar segurança e tolerância à falhas em aplicações. Para isso, o mesmo apresenta uma revisão sobre táticas de segurança na arquitetura de *softwares*, uma explicação sobre os padrões de projeto abordados no artigo e as conclusões extraídas através do estudo.

Os padrões de projeto abordados no artigo foram *Triple Modular Redundancy Pattern*, *The Reflective State Pattern* e *The Fault Pattern*. Visando apresentar os padrões estudados de forma clara, objetiva e aplicável em diversos contextos, os autores os trazem representados através de diagramas UML e os explicam de forma complementar no corpo do texto. Além disso, são apresentados pequenos trechos de código que exemplificam o *design* apresentado.

Gawand *et al.* concluem que padrões de projeto são ferramentas que podem aumentar a flexibilidade, qualidade e produtividade de aplicações. A apresentação dessas ferramentas é potencializada com o uso de diagramas UML, tornando as soluções apresentadas facilmente aplicadas em vários cenários. Nota-se que, assim como essa monografia, o artigo explora o uso padrões de projeto visando tornar uma dada aplicação mais confiável. No entanto, neste estudo, outros benefícios que podem ser obtidos com o uso de *design patterns* também são abordados, como por exemplo, tolerância a falhas. Outro ponto de divergência dos trabalhos é que o artigo faz uma revisão teórica e apresentação visual dos padrões levantados, enquanto neste trabalho é realizada, além de uma revisão teórica, implementação e análise quantitativa dos padrões.

4.3 *PERFORMANCE MODELING AND ANALYSIS OF MESSAGE-ORIENTED EVENT-DRIVEN SYSTEMS*

Performance modeling and analysis of message-oriented event-driven systems é um artigo escrito por Sachs, Kounev e Buchmann (2013). Nele é apresentada uma metodologia abrangente de modelagem para sistemas orientados a mensagens, que é ilustrada por meio de um estudo de caso no domínio da gestão da cadeia de suprimentos. Tal artigo tem o intuito de trazer insumos a respeito de desempenho e escalabilidade do sistema. Para isso, utilizam uma aplicação representativa para avaliar a eficácia de uma técnica de modelagem de desempenho sob diferentes tipos de cargas de trabalho.

Os autores apresentam a aplicação e seu funcionamento básico, para então apresentar a análise de desempenho realizada. O *software Queueing Petri net Modeling Environment* foi utilizado para tal análise, onde os autores coletaram dados relacionados a utilização de CPU, taxa de entrega das mensagens e taxa de erros do sistema. O artigo também introduziu um conjunto de padrões genéricos de modelagem de desempenho que podem ser utilizados como blocos de construção ao modelar sistemas orientados por mensagens e orientados por eventos.

Vários cenários diferentes, variando a intensidade da carga de trabalho e a mistura de interações, foram considerados e a precisão dos modelos desenvolvidos foi avaliada. Os resultados demonstraram a eficácia e praticidade da abordagem de modelagem proposta através de ganhos de desempenho no sistema.

Nota-se que, assim como a presente trabalho, o artigo apresentado busca realizar uma análise de desempenho em um sistema que utiliza um *middleware* de filas de mensagens como forma de comunicação. Tal análise é feita de forma quantitativa e é baseada em métricas de desempenho coletadas com o sistema em funcionamento sob diferentes cargas. A mesma abordagem é utilizada nesta monografia - no entanto, aqui busca-se realizar uma análise para avaliar a influência de diversos padrões de projetos no desempenho de um sistema e como esses padrões se comportam com a variação do volume e tipo de mensagens trocadas.

5 IMPLEMENTAÇÃO DOS PADRÕES DE PROJETO

Neste capítulo serão detalhadas as tecnologias utilizadas e o processo de desenvolvimento dos quatro padrões de projeto abordados neste trabalho. Também serão descritas as ferramentas e bibliotecas empregadas, assim como as estratégias de codificação adotadas durante o processo de implementação de cada padrão. A implementação de todos os padrões foi realizada utilizando o *message broker* RabbitMQ e a linguagem de programação Java 17.

Um conceito muito relevante no RabbitMQ são os *exchanges*, que são componentes responsáveis por distribuir mensagens em filas de acordo com diferentes tipos de regras de roteamento. Existem quatro variações de *exchange*: *direct*, que envia mensagens para filas específicas com base em uma chave de roteamento; *topic*, que permite roteamento com base em padrões flexíveis usados nas chave de roteamento; *fanout*, que distribui a mensagem para todas as filas vinculadas ao *exchange*, sem considerar chaves de roteamento; e *headers*, que usa os atributos das mensagens (*headers*) para determinar o roteamento (RABBITMQ TUTORIALS, 2024).

Outro ponto relevante sobre o funcionamento deste *middleware* é que o padrão de distribuição utilizado nas filas é o de Consumidores Concorrentes, onde as mensagens postadas serão divididas entre receptores conectados a uma mesma fila. A distribuição entre consumidores é feita de maneira *round-robin*, ou seja, cada mensagem é entregue a um consumidor em ordem sequencial, garantindo uma divisão equilibrada da carga de trabalho entre os mesmos (VIDELA; WILLIAMS, 2012). Esse funcionamento básico será levado em consideração nas implementações de alguns padrões.

Elementos utilizados para obtenção de dados neste estudo, como marcação de tempo de execução do programa e cálculos de métricas, foram abstraídos das demonstrações de implementação nas seções seguintes. A implementação integral de cada um dos padrões pode ser encontrada nos anexos deste trabalho.

5.1 PADRÃO PUBLICADOR-ASSINANTE

5.1.1 Serviço Produtor

O serviço produtor do padrão de projetos Publicador-Assinante, demonstrado na Figura 8, tem uma arquitetura de simples implementação. Seguindo um molde que se repetirá nos demais produtores e consumidores implementados, primeiramente é necessário importar as classes *Channel*, *Connection* e *ConnectionFactory* da biblioteca *RabbitMQ Java Client Library*, utilizada nesse projeto. Essas classes serão usadas para criar componentes básicos necessários para o funcionamento do serviço produtor.

Em seguida é declarada uma constante com o nome do *exchange* ao qual o

serviço produtor enviará as mensagens, observada na linha 1 da Figura 8. É importante que o nome dessa variável seja o mesmo no serviço produtor e no serviço consumidor, garantindo que as diferentes aplicações consigam se comunicar através da conexão ao mesmo *exchange*.

O serviço então precisa criar uma conexão com o servidor RabbitMQ. Essa conexão fará diversas abstrações, como por exemplo processos de autenticação, negociação de versão do protocolo utilizado, entre outros (RABBITMQ TUTORIALS, 2024). No código demonstrado, o servidor do *middleware* roda na mesma rede que os serviços Produtor e Consumidor, sendo acessado por meio de um IP interno ("10.10.1.1"). Caso fosse necessário estabelecer uma conexão com o servidor a partir de outra rede, seria preciso declarar o *hostname* ou o endereço IP externo correspondente.

Em seguida, na linha 7, é criado um *Channel*, ou canal de comunicação. Essa é a principal interface de comunicação entre a aplicação (produtor ou consumidor) e o *broker* RabbitMQ - todas as operações de envio e recebimento de mensagens ocorrem através de um *Channel*. A declaração da conexão criada, assim como do *Channel*, ocorrem dentro de uma estrutura *try-with-resources* do Java, garantindo que estes recursos serão fechados após sua utilização.

Figura 8 – Implementação Serviço Produtor - Publicador-Assinante

```
1     private final static String EXCHANGE_NAME = "exchange-pubsub";
2
3     public static void main(String[] args) throws Exception {
4         ConnectionFactory factory = new ConnectionFactory();
5         factory.setHost("10.10.1.1");
6
7         try (Connection connection = factory.newConnection(); Channel
    channel = connection.createChannel()) {
8
9             channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
10            // creates 5 bytes payload
11            String payload = MetricsUtils.createPayload(5);
12
13            for(int i=0; i < 1000; i++) {
14                channel.basicPublish(EXCHANGE_NAME, "", null, payload.
    getBytes());
15            }
16
17        }
18    }
19
```

Fonte: Autoria própria

O *exchange* é declarado em um padrão *fanout*, garantindo que todas as filas conectadas ao mesmo receberão cópias das mensagens enviadas. Finalmente, dentro de um laço com o número de iterações desejado - no caso dessa demonstração, 1000 - o serviço produtor envia mensagens para o *exchange* criado através do método *ba-*

sicPublish, na linha 13. Esse método recebe como parâmetros o nome do *exchange*, uma chave de roteamento (não necessária para *exchanges* do tipo *fanout*), propriedades da mensagem e a própria mensagem a ser enviada. Nota-se que o conteúdo da mensagem, criado na linha 11 do código, serve apenas o propósito de criar pacotes de um tamanho específico. Em um cenário real, este conteúdo seria populado com informações relevantes para a aplicação.

5.1.2 Serviço Consumidor

A implementação do serviço consumidor do padrão Publicador-Assinante, ou *Publisher-Subscriber*, é demonstrada na Figura 9. Além de todas as classes importadas e utilizadas no serviço produtor deste mesmo padrão, o serviço consumidor também necessita da classe *DeliverCallback*, utilizada no processamento de mensagens. Seguindo o mesmo funcionamento do serviço produtor deste mesmo padrão e das demais implementações, devem ser criadas uma conexão com o servidor do RabbitMQ e declarado um canal de comunicação com o mesmo, demonstrados nas linhas 4 a 7. Nota-se que nesse caso não é utilizada uma estrutura *try-with-resources*, pois não é desejado que o serviço consumidor seja fechado após o consumo de cada mensagem. Essa forma de declaração se repete nas implementações dos demais serviços receptores deste trabalho.

Figura 9 – Implementação Serviço Consumidor - Publicador-Assinante

```
1     private static final String EXCHANGE_NAME = "exchange-pubsub";
2
3     public static void main(String[] args) throws Exception {
4         ConnectionFactory factory = new ConnectionFactory();
5         factory.setHost("10.10.1.1");
6         Connection connection = factory.newConnection();
7         Channel channel = connection.createChannel();
8
9         String tempQueue = channel.queueDeclare().getQueue();
10        channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
11        channel.queueBind(tempQueue, EXCHANGE_NAME, "");
12
13        ...
14
15        DeliverCallback deliverCallback = (consumerTag, delivery)->{
16           String message = new String(delivery.getBody(), "UTF-8");
17        };
18
19        channel.basicConsume(tempQueue, true, deliverCallback,
    consumerTag -> {});
20
21    }
22
```

Fonte: Autoria própria

O serviço consumidor declara o *exchange* da mesma forma que o serviço produtor pois, caso o consumidor seja iniciado antes do produtor, é importante garantir que essa estrutura existirá, prevenindo assim erros de execução. Adicionalmente, o serviço consumidor deve declarar a fila que se conectará ao *exchange*. Para um padrão Publicador-Assinante é necessário que cada consumidor possua uma fila própria - dessa forma, quando o *exchange* com o padrão *fanout* enviar as mensagens recebidas para todas as filas, cada um dos consumidores receberá uma cópia da mensagem. Para isso é utilizado o método *queueDeclare()*, na linha 9, sem parâmetros específicos - isso criará uma fila não-durável, exclusiva, auto-deletável e com um nome gerado automaticamente. Essa fila é vinculada ao *exchange* por meio de uma operação de *bind*, observada na linha 11 do código. Os diferentes consumidores não podem se conectar a uma mesma fila, pois caso o fizessem, assumiriam o padrão de Consumidores Concorrentes que é padrão do agente.

Finalmente, devem ser feitas as configurações para que o serviço consiga consumir e processar as mensagens enviadas para a fila criada. Para isso, é declarada a função *deliverCallback*, na linha 15, que é chamada automaticamente a cada mensagem lida pelo serviço. Nesse caso, a função recebe dois parâmetros: o *consumerTag*, que identifica o consumidor, e um objeto do tipo *delivery*, do qual é obtido o conteúdo da mensagem recebida. Caso houvesse mais algum processamento a ser realizado com as mensagens recebidas, ele deveria ser executado dentro deste bloco.

A função *basicConsume*, na linha 19, é responsável por conectar o consumidor à fila e consumir as mensagens postadas. O primeiro parâmetro declarado na mesma é o nome da fila a ser consumida, seguido de um valor *booleano* que informa se o *acknowledgment* automático do recebimento das mensagens está ativado. Os dois últimos parâmetros identificam a função de *callback* declarada previamente, assim como outro *callback* que é acionado caso o consumidor seja cancelado - nesse caso, nenhuma ação é especificada.

5.2 PADRÃO CONSUMIDORES CONCORRENTES

5.2.1 Serviço Produtor

Com um funcionamento parecido ao do serviço produtor do padrão Publicador-Assinante, o produtor do padrão Consumidores Concorrentes difere apenas no tipo de *exchange* utilizado. Aqui, conforme a linha 6 da Figura 10, utiliza-se o funcionamento *direct* de roteamento. Isso significa que o *exchange* enviará as mensagens recebidas apenas para filas que estão conectadas a ele com uma chave de roteamento específica, que deve ser a mesma chave de roteamento declarada pelo consumidor. Essa chave é declarada na linha 2 da Figura 10 e é utilizada na operação *basicPublish*, demonstrada na linha 10 do código, no momento de envio das mensagens.

Figura 10 – Implementação Serviço Produtor - Consumidores Concorrentes

```
1     private final static String EXCHANGE_NAME = "exchange-cc";
2     private static final String ROUTING_KEY = "key-cc";
3
4     ...
5
6     channel.exchangeDeclare(EXCHANGE_NAME, "direct");
7     String payload = getPayloadFromFile();
8
9     for(int i=0; i < 1000; i++) {
10        channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY, null, payload.
    getBytes());
11    }
12
```

Fonte: Autoria própria

5.2.2 Serviço Consumidor

A implementação do serviço consumidor do padrão de projeto Consumidores Concorrentes também se aproxima da implementação realizada para o padrão Publicador-Assinante, contendo apenas algumas modificações demonstradas na Figura 11. Uma delas é que, além da declaração do *exchange* ao qual o consumidor se conectará, é declarada uma fila específica. Isso ocorre pois todos os consumidores interessados em receber mensagens deverão estar consumindo da mesma fila - assim, a carga de mensagens será dividas entre os consumidores conectados na fila, seguindo a lógica de Consumidores Concorrentes.

Por esse motivo a fila declarada não é gerada com um nome aleatório, e sim com um nome que será utilizado por todos os consumidores, observado na linha 9. É então realizada uma operação de *bind* na linha 11 vinculando a fila e o *exchange* declarados por meio de uma *routing key*, ou uma chave de roteamento. Essa chave é a mesma que foi utilizada pelo serviço produtor ao enviar as mensagens.

Ao conectar diversos serviços consumidores para processarem mensagens de uma mesma fila no RabbitMQ, o *exchange* define, por padrão, uma distribuição no estilo *Round Robin* — ou seja, envia cada mensagem sucessivamente para um consumidor diferente. Para implementar o padrão de Consumidores Concorrentes no qual cada consumidor recebe mensagens da fila conforme sua disponibilidade, a função *basicConsume* foi configurada com o parâmetro de *acknowledgment* automático definido como *false*, como mostrado na linha 18 do código. Dessa forma, o RabbitMQ considera que uma mensagem foi consumida com sucesso apenas ao receber um *acknowledgment* manual. Esse processo ocorre dentro da função *deliverCallback*, onde a chamada à função *channel.basicAck* é feita na linha 15, confirmando o recebimento da mensagem somente após o consumidor ter concluído todo o processamento necessário. Após isso, o serviço consumidor estará apto a receber mais mensagens da fila.

Figura 11 – Implementação Serviço Consumidor - Consumidores Concorrentes

```
1         private static final String EXCHANGE_NAME = "exchange-cc";
2         private final static String QUEUE_NAME = "queue-cc";
3         private static final String ROUTING_KEY = "key-cc";
4         private static final int PREFETCH_COUNT = 1;
5
6         ...
7
8         channel.basicQos(PREFETCH_COUNT);
9         channel.queueDeclare(QUEUE_NAME, false, false, false, null);
10        channel.exchangeDeclare(EXCHANGE_NAME, "direct");
11        channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, ROUTING_KEY);
12
13        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
14            String message = new String(delivery.getBody(), "UTF-8");
15            channel.basicAck(delivery.getEnvelope().getDeliveryTag(),
    false);
16        };
17
18        channel.basicConsume(QUEUE_NAME, false, deliverCallback,
    consumerTag -> {});
19
```

Fonte: Autoria própria

Além disso, o *prefetch count* do consumidor foi definido como 1, conforme demonstrado na linha 9 do código. Com essa configuração, garante-se que cada consumidor receba apenas uma mensagem por vez.

5.3 PADRÃO FILA DE PRIORIDADE

5.3.1 Serviço Produtor

A implementação do serviço produtor do padrão Fila de Prioridade, demonstrada na Figura 12, difere da implementação realizada no padrão Publicador-Assinante apenas devido à utilização da propriedade de prioridade da mensagem. No contexto do RabbitMQ, a prioridade configurada permite que mensagens mais prioritárias sejam consumidas antes das mensagens com prioridades mais baixas quando há um acúmulo de mensagens na fila.

Neste código a declaração de prioridade das mensagens ocorre na linha 2. Por utilizar uma função randômica, distribuição dos números gerados será uniforme entre os valores possíveis. Nota-se, no entanto, que em um cenário real esse valor deveria ser definido de acordo com a lógica de negócio adotada. Na implementação demonstrada, as possibilidades de prioridade de mensagens são valores de 1 a 3, sendo 3 a prioridade máxima possível. A classe *AMQP.BasicProperties* encapsula as propriedades da uma mensagem - nesse caso, a propriedade de prioridade é adicionada através do método *priority*, na linha 5, e esse pacote de propriedades é

então passado como parâmetro do método *basicPublish*, na linha 8 do código.

Figura 12 – Implementação Serviço Produtor - Fila de Prioridade

```
1       for(int i=0; i < 1000; i++) {
2           int priority = ThreadLocalRandom.current().nextInt(1, 4);
3
4           AMQP.BasicProperties props = new AMQP.BasicProperties.
    Builder()
5                   .priority(priority)
6                   .build();
7
8           channel.basicPublish(EXCHANGE_NAME, "", props, payload.
    getBytes("UTF-8"));
9       }
10
```

Fonte: Autoria própria

5.3.2 Serviço Consumidor

O padrão Fila de Prioridade possui uma implementação de serviço consumidor muito semelhante ao padrão Publicador-Assinante. A particularidade do mesmo se encontra apenas na declaração da fila, demonstrada na linha 1 da Figura 13, onde é definida uma propriedade de prioridade da fila. Tal propriedade é chamada de "*x-max-priority*" e, no presente trabalho, foi definida com o valor 3 - ou seja, existem 3 níveis de prioridade que podem ser utilizados para caracterizar as mensagens adicionadas nessa estrutura. Essa configuração é o que permite ao RabbitMQ priorizar mensagens com maior preferência em detrimento de mensagens com prioridades menores.

Figura 13 – Implementação Serviço Consumidor - Fila de Prioridade

```
1       channel.queueDeclare(QUEUE_NAME, true, false, false, Map.of("x-
    max-priority", 3));
2
```

Fonte: Autoria própria

5.4 PADRÃO DE VERIFICAÇÃO DE DECLARAÇÃO

5.4.1 Serviço Produtor

O padrão de Verificação de Declaração segue a mesma formatação dos demais serviços produtores, no entanto, esse padrão conta com uma estrutura externa para armazenamento dos *payloads* das mensagens antes de enviá-las. Isso otimiza o tráfego no RabbitMQ, especialmente em cenários de mensagens grandes, que podem sobrecarregar as estruturas do *broker*. Neste trabalho a ferramenta Redis (*Remote*

Dictionary Server), um banco de dados em memória e orientado a chave-valor, foi utilizada para armazenamento externo. Esse banco foi utilizado pois é altamente performático, capaz de armazenar dados temporariamente e responder a consultas de forma rápida, o que o torna ideal para aplicações de *caching*, filas de mensagens e processamento de dados em tempo real (CARLSON, 2013).

Para a implementação do padrão utilizou-se a biblioteca Jedis. Da mesma forma que é necessário criar uma conexão com o servidor do *middleware* RabbitMQ, também é necessário estabelecer uma conexão com o servidor do banco de dados. Isso é feito na linha 1 da Figura 14. Posteriormente, na linha 7, é criado um identificador único para representar o *payload* da mensagem que será enviada - esse valor é então utilizado como chave para armazenar esse corpo da mensagem no Redis e, em seguida, é enviado no lugar da mensagem original, conforme demonstrado nas linhas 8 e 9 do código. O mesmo valor será utilizado pelo serviço consumidor para recuperar o corpo da mensagem.

Figura 14 – Implementação Serviço Produtor - Verificação de Declaração

```
1        try (Connection connection = factory.newConnection(); Channel
   channel = connection.createChannel(); Jedis jedis = new Jedis("
   10.10.1.2", 6379)) {
2            channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
3            // creates 5 bytes payload
4            String payload = MetricsUtils.createPayload(5);
5
6            for (int i = 0; i < 1000; i++) {
7                String claimCheckId = UUID.randomUUID().toString();
8                jedis.set(claimCheckId, payload);
9                channel.basicPublish(EXCHANGE_NAME, "", null,
   claimCheckId.getBytes());
10               }
11           }
12
```

Fonte: Autoria própria

5.4.2 Serviço Consumidor

Para a implementação do serviço consumidor do padrão de Verificação de Declaração (*Claim-Check Pattern*), é necessário estabelecer uma conexão com o banco de dados Redis, assim como foi feito no serviço produtor. Essa conexão é configurada na linha 1 da Figura 15. Em seguida, nas linhas 6 e 7, pode-se observar que o identificador único da mensagem, utilizado como chave para armazenamento de seu corpo no banco Redis, foi recebido pelo serviço consumidor. Esse dado é então utilizado para recuperar o *payload* da mensagem do banco de dados. O restante da implementação desse serviço segue o padrão já demonstrado nos demais serviços consumidores.

Figura 15 – Implementação Serviço Consumidor - Verificação de Declaração

```
1     Jedis jedis = new Jedis("10.10.1.2", 6379);
2
3     ...
4
5     DeliverCallback deliverCallback = (consumerTag, delivery) -> {
6         String claimCheckId = new String(delivery.getBody(), "UTF-8");
7         String payload = jedis.get(claimCheck);
8     };
9
```

Fonte: Autoria própria

6 AVALIAÇÃO DOS RESULTADOS

Após a implementação dos padrões de projeto discutidos nos capítulos anteriores, foram realizados testes para análise de desempenho de cada um dos protótipos desenvolvidos. Os testes tiveram como objetivo comparar as arquiteturas por meio da coleta de métricas, a fim de identificar as diferenças de performance entre os padrões de projeto e situações em que cada um deles se destaca.

Buscou-se avaliar a eficiência dos padrões em processar grandes volumes de mensagens em um intervalo de tempo, a capacidade do sistema de concluir o processamento completo de um lote de mensagens e a capacidade do sistema de responder rapidamente a mensagens individuais. Para isso, foi selecionado um conjunto de métricas que refletem aspectos críticos do desempenho desses sistemas e como os mesmos são afetados pelos diferentes padrões de projeto.

Os resultados analisam o comportamento de dos padrões em termos de vazão (número de mensagens consumidas por segundo), *makespan* (tempo total para o serviço consumidor ler todas as mensagens enviadas pelo serviço produtor), latência (tempo que capa mensagem permaneceu na fila) e capacidade do sistema de lidar com diferentes tamanhos de mensagens. Essas métricas, em conjunto, fornecem uma visão ampla da performance, permitindo identificar os cenários mais adequados para cada padrão e eventuais gargalos no sistema. Nas sessões abaixo são descritos os procedimentos adotados para coleta das métricas analisadas, assim como *insights* sobre as vantagens e desvantagens de cada abordagem.

6.1 EXECUÇÃO DOS EXPERIMENTOS

Para a realização dos experimentos foi utilizada a plataforma Emulab, um ambiente de pesquisa e desenvolvimento que fornece infraestrutura para experimentação em sistemas distribuídos (EIDE et al., 2006). Os códigos desenvolvidos para cada um dos padrões de projeto selecionados foram testados em uma variedade de cenários, tendo as métricas de vazão, latência e *makespan* coletadas para análise. Cada etapa do processo de experimentação, assim como a análise dos resultados obtidos, são discutidos nas seções abaixo.

6.1.1 Ambiente de Teste

A plataforma Emulab oferece diversas máquinas com configurações específicas que podem ser reservadas e utilizadas por seus usuários. Nos experimentos deste trabalho foram utilizadas 5 máquinas diferentes, todas com as mesmas configurações. Esses equipamentos são acessados remotamente via SSH (*Secure Shell*), um protocolo de rede usado para estabelecer uma conexão segura entre dois dispositivos.

Todos os computadores utilizados nos testes possuíam a mesma configuração: 64 GB de memória RAM e um processador Intel Xeon E5-2630v3, com arquitetura de 64 bits e 8 núcleos de CPU. O sistema operacional utilizado, Ubuntu 22.04, foi configurado com uma reserva de 4 GB de memória, além de contar com um disco de 2 TB para armazenamento - assim, garantiu-se suporte adequado para a coleta e retenção dos dados experimentais.

6.1.2 Preparação do Ambiente

Após sua alocação, as máquinas foram distribuídas da seguinte forma: uma foi dedicada à execução do servidor do *message broker* RabbitMQ, outra ao servidor do banco de dados Redis, uma terceira para rodar os serviços produtores dos padrões de projeto implementados, e as demais foram destinadas aos serviços consumidores. Para a execução dos experimentos, foi necessário instalar os programas correspondentes em cada servidor: RabbitMQ na máquina 1, Redis na máquina 2 e o pacote JDK, para a execução de programas Java, nas máquinas dedicadas aos serviços produtores e consumidores.

Além disso, foi instalado o programa NTP (*Network Time Protocol*) nos servidores dos serviços produtores e consumidores. NTP é um protocolo que tem como objetivo manter os relógios de sistemas sincronizados, o que é importante para a coleta de métricas precisas e fidedignas nos experimentos. Após a instalação e sincronização dos relógios dos equipamentos necessários, o ambiente estava preparado para a execução dos testes.

Os programas desenvolvidos foram compilados e empacotados em arquivos JAR (*Java Archive*) – esses arquivos contêm o conjunto de classes Java, bibliotecas e demais recursos necessários para a execução do programa. Os mesmos foram enviados aos servidores via protocolo SCP (*Secure Copy Protocol*), que permite a transferência segura de arquivos entre computadores utilizando o protocolo SSH para autenticação e proteção dos dados. Após o ambiente ser preparado e os programas compilados e enviados para os servidores Emulab, os experimentos foram executados.

6.1.3 Execução dos Experimentos e Coleta de Métricas

Foram definidos diferentes cenários para a testagem dos padrões de projeto implementados. Uma das variações nesses cenários é que todos os experimentos foram executados em nas seguintes situações:

1. Pré-populando as filas de mensagens e iniciando o serviço consumidor apenas após todas as mensagens terem sido publicadas na fila;
2. Executando simultaneamente os serviços produtor e consumidor a fim de que as mensagens fossem consumidas à medida que eram publicadas.

Isso porque as métricas de *makespan* e vazão foram coletadas medindo o tempo que o serviço consumidor de cada padrão de projeto levou para esvaziar as filas pré-populadas. A métrica de latência, no entanto, seria afetada negativamente no cenário 1, pois o tempo de permanências das mensagens na fila seria expandido. Dessa forma, essa métrica foi coletada apenas no cenário 2, onde os serviços produtores e consumidores foram executados simultaneamente.

Além disso, cada padrão de projeto foi testado com mensagens de diferentes tamanhos: pequenas, com apenas 5 *bytes*, e grandes, de 2.5 MB. Essa variação teve como objetivo avaliar o comportamento do sistema ao lidar com cargas de diferentes magnitudes. Por fim, os padrões foram avaliados tanto em cenários com um único consumidor quanto com 10 consumidores distintos lendo as mensagens enviadas pelo produtor, permitindo mensurar o impacto de múltiplos assinantes interagindo simultaneamente com o sistema. O volume de mensagens permaneceu constante, com 50000 mensagens enviadas por experimento. A Tabela 1 resume os cenários de teste aplicados a cada padrão de projeto.

Tabela 1 – Cenários de Teste

Cenário	**Fila**	**Tamanho da Mensagem**	**Nº de Consumidores**
Cenário 1	Pré-populada	5 bytes	1 Consumidor
Cenário 2	Pré-populada	5 bytes	10 Consumidores
Cenário 3	Pré-populada	2.5 MB	1 Consumidor
Cenário 4	Pré-populada	2.5 MB	10 Consumidores
Cenário 5	Pós-populada	5 bytes	1 Consumidor
Cenário 6	Pós-populada	5 bytes	10 Consumidores
Cenário 7	Pós-populada	2.5 MB	1 Consumidor
Cenário 8	Pós-populada	2.5 MB	10 Consumidores

Fonte: Autoria própria

Para medir o *makespan*, foi registrado um *timestamp* no momento em que o serviço consumidor (ou os serviços, nos cenários com 10 consumidores) iniciou o processamento das mensagens da fila. Após esvaziá-la, um novo *timestamp* foi registrado, permitindo que o tempo total necessário para consumir todas as mensagens postadas fosse calculado pela diferença entre esses dois valores. A vazão do sistema foi determinada dividindo o número total de mensagens consumidas pelo *makespan* em cada cenário. Em ambos os casos, as métricas foram coletadas com as filas de mensagens previamente populadas.

A métrica de latência, por outro lado, foi obtida executando simultaneamente os serviços produtores e consumidores. Para isso, o serviço produtor adicionava um *timestamp* como propriedade de cada mensagem antes de enviá-la para a fila. Ao receber a mensagem, o serviço consumidor registrava um novo *timestamp*, permitindo

calcular o tempo total de permanência da mensagem na fila como a diferença entre esses dois valores. Devido ao alto volume de mensagens enviadas, a latência foi coletada para apenas 4% das mensagens consumidas em cada experimento. Para evitar vieses, as mensagens analisadas foram selecionadas de forma aleatória.

6.2 ANÁLISE DE RESULTADOS

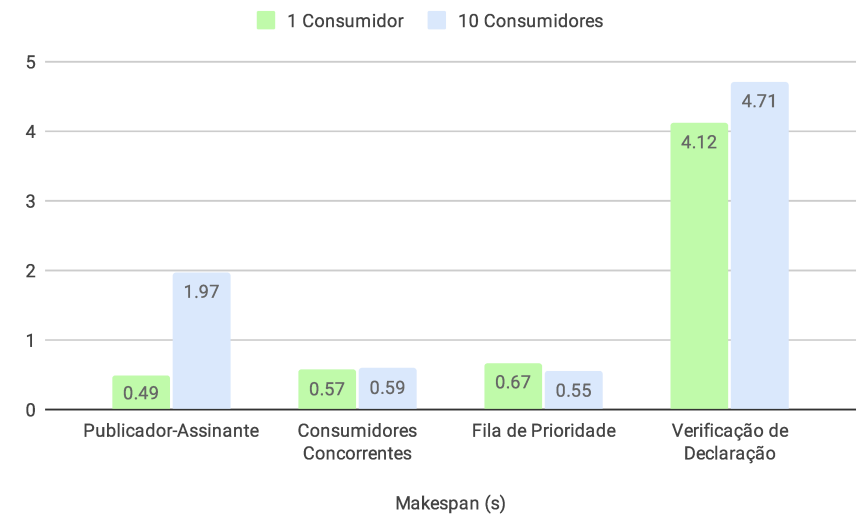
Nesta seção são apresentados os resultados obtidos nos testes de desempenho dos diferentes padrões de projeto implementados, com o objetivo de analisar a eficiência de cada um na obtenção de mensagens em um sistema de filas que utiliza o *message broker* RabbitMQ. Os resultados relacionados a *makespan* (medido em segundos), vazão (medida em mensagens por segundo) e latência (medida em milissegundos) são exibidos e discutidos nos parágrafos a seguir.

A Figura 16 apresenta uma análise comparativa das médias de *makespan* (em segundos) para cada padrão de projeto implementado, considerando cenários com mensagens pequenas (Figura 16(a)) e com mensagens grandes (Figura 16(b)). Os gráficos indicam que o aumento do número de consumidores no sistema, atuando tanto de forma concorrente quanto multi-assinante, resulta em uma diminuição na capacidade de consumo e processamento das mensagens. É relevante destacar que o agente RabbitMQ divide a carga de mensagens entre consumidores vinculados a uma mesma fila de forma *Round Robin*. Por esse motivo, apenas o padrão de projeto Publicador-Assinante, que dedica uma fila exclusiva para cada consumidor do sistema, apresenta um aumento significativo no tempo de consumo das mensagens da fila. Nos demais padrões, o crescimento do tempo de consumo causado pelo maior número de consumidores e pela divisão de recursos do sistema é parcialmente compensado pela concorrência na leitura das mensagens, o que impede um crescimento expressivo no *makespan*.

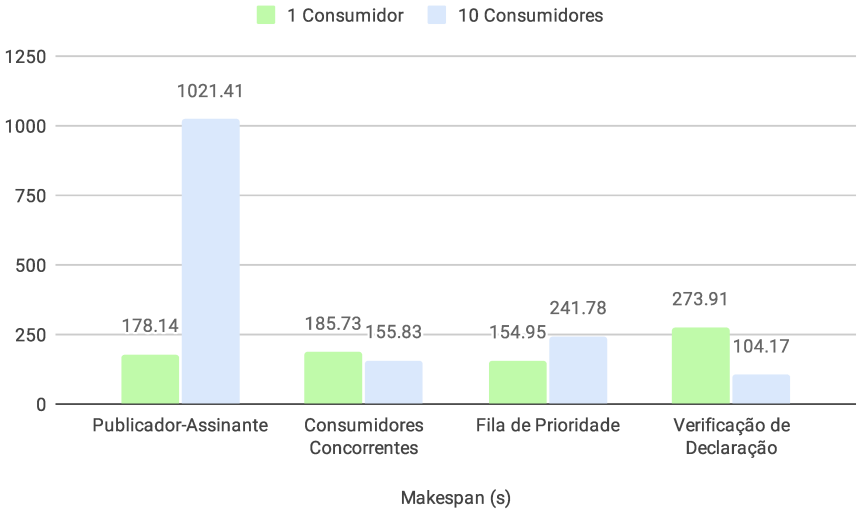
As medições de vazão, exibidas na Figura 17, complementam as análises de *makespan*. Observa-se que, em todos os cenários, a vazão para um único consumidor é superior à vazão com 10 consumidores no sistema. Isso porque a concorrência por recursos e pelo acesso aos pacotes disponibilizados pelo RabbitMQ geram impactos, conforme discutido anteriormente. Embora o padrão de projeto Publicador-Assinante apresente uma queda menos acentuada na vazão em comparação com os outros padrões quando o número de consumidores aumenta, o *makespan* desse padrão é significativamente maior. Esse aumento no *makespan* deve-se à falta de concorrência pelas mensagens nesse modelo. A Figura 17 também demonstra que o impacto causado pelo aumento de consumidores sem concorrência, demonstrado na vazão do padrão Publicador-Assinante, é inferior ao impacto observado nos demais padrões, onde os consumidores competem pela obtenção das mensagens.

O tamanho das mensagens enviadas no sistema também causa efeitos relevan-

Figura 16 – *Makespan* médio (a) e (b)



(a) Medição para mensagens pequenas

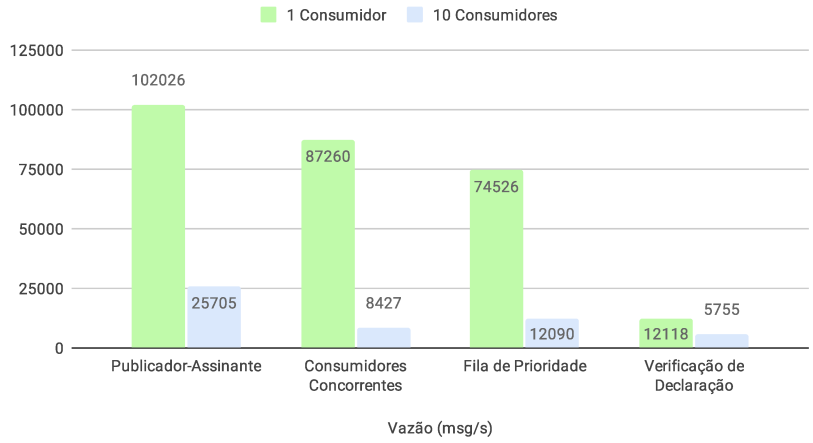


(b) Medição para mensagens grandes
Fonte: Autoria própria

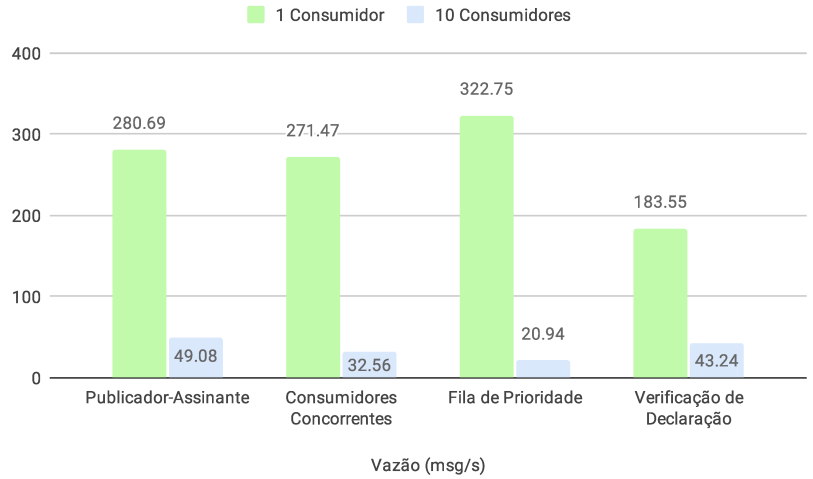
tes. Todos os padrões de projeto foram testados tanto com mensagens pequenas, de 5 bytes, quanto com mensagens grandes, de 2.5 MB - em ambos os casos com o envio de 50000 mensagens. A vazão do padrão Publicador-Assinante foi aproximadamente 363 vezes maior ao enviar mensagens de 5 bytes em comparação com o envio de mensagens de 2.5 MB. No padrão Consumidores Concorrentes, a vazão para mensagens pequenas foi aproximadamente 321 vezes maior. Diferenças proporcionais também podem ser observadas nas medidas de *makespan*. Nota-se, então, que popular o sistema com mensagens grandes causa um impacto mais significativo no sistema do que o aumento no número de consumidores.

As métricas de *makespan* e de vazão também tornam evidente a ineficiência do padrão de projetos Verificação de Declaração ao lidar com mensagens pequenas,

Figura 17 – Vazão média (a) e (b)



(a) Medição para mensagens pequenas



(b) Medição para mensagens grandes
Fonte: Autoria própria

com concorrência entre consumidores para obtenção das mensagens ou não. Isso porque, em situações onde o tamanho e volume de mensagens não sobrecarrega o *message broker*, o custo associado ao armazenamento do corpo das mensagens em uma estrutura externa e posterior obtenção dos mesmos é maior do que o custo de simplesmente enviar as mensagens íntegras pela fila.

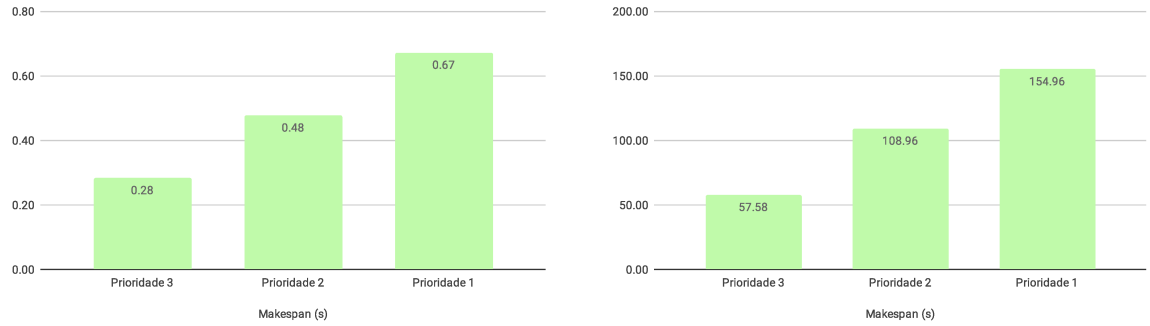
Ao lidar com mensagens grandes, no entanto, a vantagem associada a esse padrão começa a ser observada: o menor tempo registrado para a obtenção de mensagens de 2.5 MB, por uma margem de 50 segundos, foi associando o padrão de projetos Verificação de Declaração com 10 consumidores. É possível inferir, assim, que cenários com mensagens ainda maiores podem salientar essa vantagem. Além disso, em situações onde as mensagens a serem enviadas são maiores do que o suportado pelo *broker* ou que, em combinação com um alto volume de pacotes, podem sobrecarregar o agente causando falhas, esse padrão se torna uma solução eficaz.

6.2.1 Fila de Prioridades

O padrão de projeto Fila de Prioridades é suportado pelo agente RabbitMQ, e recomenda-se que não sejam definidos mais de 5 níveis de prioridade. Isso porque recursos de CPU e memória do servidor do *message broker* precisam ser alocados para a ordenação da fila de mensagens conforme as prioridades definidas: internamente, o RabbitMQ cria e mantém uma sub-fila para cada nível de prioridade. Recursos adicionais, especialmente de CPU, também são comprometidos no serviço consumidor (RABBITMQ DOCUMENTATION, 2024). Na implementação realizada para este trabalho foram definidos 3 níveis de prioridade, sendo “3” o mais prioritário e “1” o menos.

A distribuição de prioridades entre as mensagens foi feita de maneira randômica e equilibrada, de modo que aproximadamente um terço das mensagens enviadas foi vinculado a cada nível. Na Figura 18 (a) observa-se que, ao lidar com pacotes pequenos, as mensagens de maior prioridade são consumidas, em média, 1.7 vezes mais rápido do que as de prioridade 2, e 2.39 vezes mais rápido do que as de prioridade 1, menos prioritárias. Ao processar mensagens grandes, a diferença de velocidade de consumo é ainda maior: os pacotes com prioridade 3 são consumidos 1.89 vezes mais rapidamente do que os de prioridade 2, e 2.69 vezes mais rápido do que as mensagens de prioridade 1.

Figura 18 – *Makespan* médio em Fila de Prioridade (a) e (b)



(a) Medição para mensagens pequenas (b) Medição para mensagens grandes

Fonte: Autoria própria

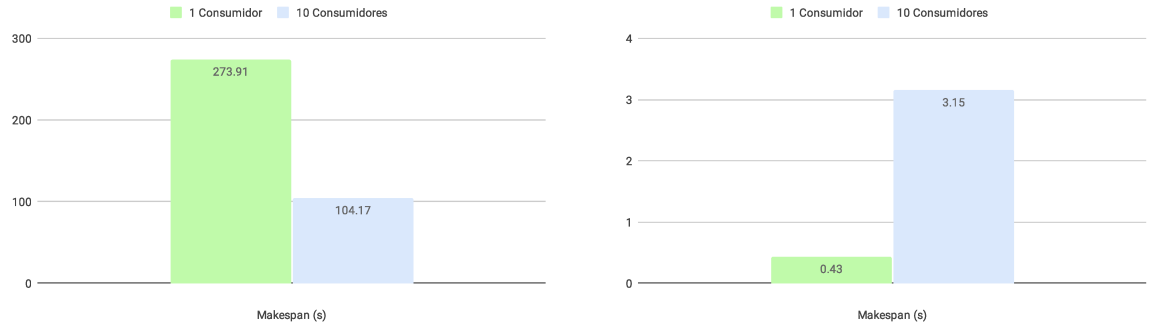
Os impactos causados pela complexidade adicional desse padrão de projetos podem ser observados nas medidas de *makespan* (Figura 16) e vazão (Figura 17). As métricas de consumo de mensagens pequenas com apenas um consumidor e de mensagens grandes com 10 consumidores são as mais afetadas, apresentando intervalos maiores do que outros padrões implementados. Ainda assim, as vantagens apresentadas para obtenção de pacotes prioritários são relevantes e podem ser de grande valia para sistemas com necessidades e urgências diferentes para cada tipo de pacote recebido.

Uma consideração importante sobre o uso do padrão de projeto Fila de Prioridades no RabbitMQ é que, em cenários onde as mensagens são consumidas imediatamente após serem postadas na fila, o *broker* não consegue priorizá-las de forma eficaz. Isso ocorre porque o curto tempo de permanência das mensagens na fila não permite a reordenação das mesmas. Nesses casos, a utilização do padrão Fila de Prioridades não é recomendada, já que as mensagens serão consumidas na ordem em que foram postadas, tornando desnecessária a alocação de recursos adicionais para a estruturação do padrão.

6.2.2 Verificação de Declaração

As Figuras 19 e 20 mostram o *makespan* e vazão médios para o padrão de projetos Verificação de Declaração ao lidar com mensangens grandes. Essas figuras exibem tanto um cenário onde o *payload* da mensagem é recuperado do banco de dados Redis, quanto um onde apenas o identificador único do *payload* é recebido pelo serviço consumidor, sem metrificar o tempo necessário para a obtenção do *payload* do banco de dados. Essa comparação é relevante pois o serviço consumidor não necessariamente precisa acessar o corpo da mensagem assim que a recebe. Nesses cenários, o recebimento do identificador para a posterior obtenção do *payload* é suficiente e garante o acesso aos dados armazenados no Redis quando necessário.

Figura 19 – *Makespan* médio em Verificação de Declaração (a) e (b)

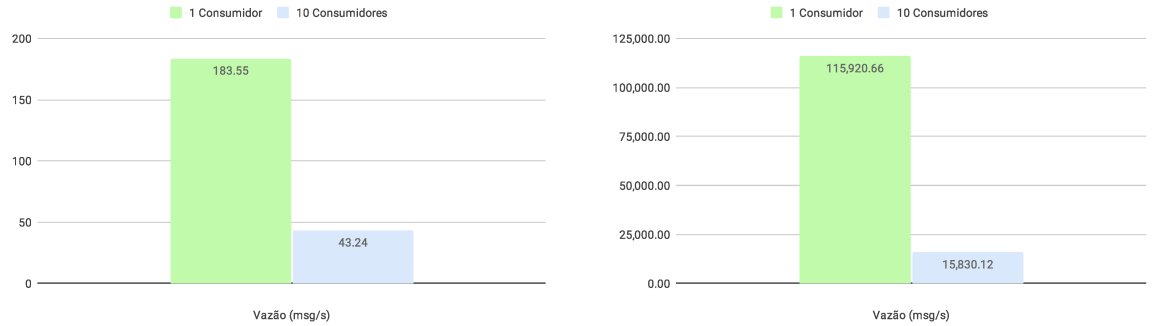


(a) Medição com recuperação do *payload* (b) Medição com recuperação do Id

Fonte: Autoria própria

Os identificadores únicos utilizados como chave para armazenar o *payload* das mensagens no Redis têm, em média, 36 bytes. Dessa forma, como esperado, as métricas obtidas ao recuperar apenas os Ids das mensagens se aproximam das observadas ao recuperar mensagens pequenas. Assim, em situações onde as informações do *payload* não são necessárias imediatamente para o serviço consumidor, o padrão de projetos Verificação de Declaração demonstra uma vantagem relevante. Isso porque as mensagens são enviadas e consumidas de forma muito eficiente, com medidas de *makespan* e vazão mais de 600 vezes superiores às medições realizadas ao recuperar

Figura 20 – Vazão média em Verificação de Declaração (a) e (b)



(a) Medição com recuperação do *payload* (b) Medição com recuperação do Id

Fonte: Autoria própria

o *payload* imediatamente. Nesses casos, a obtenção do corpo da mensagem, que é mais custosa para o sistema, pode ser realizada conforme a necessidade do serviço consumidor.

6.2.3 Latência

As Tabelas 2 e 3 mostram métricas de latência para mensagens pequenas e grandes, enviadas com 10 consumidores lendo as mensagens postadas nas filas. Para evitar a saturação do sistema durante a coleta dessas medidas, foi inserida uma pausa de alguns milissegundos entre o envio das mensagens - assim, garante-se que o intervalo medido corresponda ao tempo necessário para uma mensagem ser postada e, em seguida, retirada da fila por um consumidor disponível, sem acúmulo de pacotes no sistema. Além disso, são avaliadas as métricas de desvio padrão, a fim de analisar o quanto os valores da amostra variam em relação a média, e de Percentil 90, visando entender até qual valor de latência máximo tiveram 90% das observações da amostra.

Tabela 2 – Latência de Mensagens Pequenas

Padrão de Projeto	**Média (ms)**	**Desvio Padrão**	**Percentil 90**
Publicador-Assinante	2.63	0.55	3
Consumidores Concorrentes	0.83	0.73	2
Fila de Prioridades	0.16	0.47	0
Verificação de Declaração	1.98	0.33	2

Fonte: Autoria própria

Ao analisar a Tabela 2 fica claro que o maior valor médio de latência é vinculado ao padrão de projetos Publicador-Assinante, seguido pelo padrão Verificação de Declaração, Consumidores Concorrentes e, finalmente, Fila de Prioridade. É importante

salientar, no entanto, que o desvio padrão dos padrões Publicador-Assinante e Verificação de Declaração são baixos, indicando uma latência mais estável. Os padrões de projeto Consumidores Concorrentes e Fila de Prioridades, no entanto, tiveram valores de desvio padrão proporcionalmente mais altos, indicando que métricas de latência identificadas nas observações tem maior variação em torno da média.

Ainda assim, a latência de todos os padrões de projeto é baixa: a métrica de Percentil 90 indica que 90% das mensagens enviadas no padrão de projetos Publicador-Assinante tiveram seu tempo de latência menor ou igual a 3 milissegundos, enquanto os padrões Consumidores Concorrentes e Verificação de Declaração obtiveram percentil 90 igual a 2, e o padrão Fila de Prioridades teve o valor desse indicador igual a 0. As mensagens, então, são consumidas quase imediatamente após serem postadas na fila, tornando a troca de informações entre produtor e consumidor muito eficiente.

Tabela 3 – Latência de Mensagens Grandes

Padrão de Projeto	**Média (ms)**	**Desvio Padrão**	**Percentil 90**
Publicador-Assinante	241.93	16.04	254
Consumidores Concorrentes	47.31	2.53	52
Fila de Prioridades	48.81	2.48	52
Verificação de Declaração	28.39	2.04	31.7

Fonte: Autoria própria

A Tabela 3 mostra os valores obtidos com o envio de mensagens de 2.5 MB nas filas. Seguindo o mesmo padrão encontrado nas métricas de *makespan* e de vazão, o padrão de projetos Publicador-Assinante possui a latência média mais alta por uma ampla margem. Isso é um resultado esperado pois seus consumidores não estão concorrendo pela obtenção das mensagens; eles estão, cada um, recebendo uma cópia da mensagem enviada, o que torna o processo mais oneroso. Apesar de ser o maior valor absoluto de desvio padrão entre os padrões mensurados, o valor de 16.05 segundos é relativamente baixo em relação à média (aproximadamente 6.6%), o que indica uma variabilidade moderada.

Os padrões Consumidores Concorrentes e Fila de Prioridades apresentam valores similares de latência média e desvio padrão, sugerindo baixa variabilidade e tempos de resposta consistentes. Em ambos os padrões 90% das observações tiveram seu valor de latência menor ou igual a 52 milissegundos, o que indica que a concorrência dos consumidores e o tamanho das mensagens tornou o desempenho desses padrões mais homogêneo.

Já o padrão Verificação de Declaração apresenta a menor latência média, um resultado consistente com as medições de *makespan* e vazão obtidas. Com um desvio

padrão de aproximadamente 7.2% da média, o padrão exibe a maior variabilidade dentre as arquiteturas testadas. O indicador de percentil 90, no entanto, é aproximadamente 1.64 vezes menor que os valores obtidos pelos padrões de projeto Consumidores Concorrentes e Fila de prioridades, e 8 vezes inferior do que essa métrica para o padrão Publicador-Assinante. Esses resultados evidenciam sua alta eficiência ao lidar com mensagens grandes.

7 CONCLUSÃO

Neste trabalho foram apresentados conceitos básicos e fundamentos sobre duas áreas relevantes no desenvolvimento de sistemas distribuídos: padrões de projeto e sistemas de filas de mensagens. Inicialmente foram identificados padrões de projeto relevantes para sistemas de mensageria. Esse levantamento baseou-se na análise e estudo da produção científica atual e de livros da área de tecnologia, assim como em documentações de organizações reconhecidas no setor.

Em seguida, foram aprofundados quatro padrões de projeto amplamente aplicados em sistemas de filas de mensagens: Publicador-Assinante, Consumidores Concorrentes, Fila de Prioridades e Verificação de Declaração. Esses padrões foram estudados para entendimento de seu funcionamento básico, aplicabilidade e suas principais vantagens e desvantagens. Eles foram escolhidos como foco do estudo devido a sua popularidade dentro de sistemas de filas de mensagens e por terem enfoques distintos - assim, buscou-se entender a variação de desempenho de cada padrão em situações que buscam solucionar em comparação com demais. O resultado do estudo foi condensado em uma produção textual visando trazer uma visão sintetizada sobre os padrões de projetos selecionados, além de uma análise abrangente sobre sistemas de filas de mensagens.

Além dos padrões analisados, outras arquiteturas comumente empregadas em sistemas de mensageria também foram brevemente explicadas, proporcionando uma visão mais ampla das possibilidades da área. Com isso, espera-se que este trabalho contribua para o entendimento e aplicação eficaz de padrões de projeto em sistemas de filas de mensagens, promovendo o desenvolvimento de soluções mais eficientes e confiáveis.

Para complementar o estudo teórico, foram implementados protótipos dos padrões selecionados, seguidos de uma análise de desempenho dos mesmos em diferentes cenários. Os testes, realizados na plataforma Emulab, avaliaram essas arquiteturas por meio da coleta de métricas de *makespan*, vazão e latência, a fim de identificar as diferenças entre os padrões e destacar as situações em que cada um deles é mais ou menos vantajoso.

Os resultados obtidos tornaram evidente que a adição de consumidores em um sistema de mensageria causa variações perceptíveis no desempenho do mesmo. O padrão Consumidores Concorrentes, por exemplo, mostra que a competição pelas mensagens resulta em uma perda significativa de vazão entre os consumidores do sistema. Ainda assim, a distribuição da carga entre eles faz as mensagens serem consumidas com maior velocidade em um cenário de concorrência, o que torna essa dinâmica vantajosa para aumentar a performance sistema como um todo, mesmo com perdas de desempenho individuais para cada consumidor.

O padrão de projetos Publicador-Assinante também ocasiona uma queda de desempenho com a adição de serviços consumidores. Como, nesse caso, todas as mensagens são enviadas para cada assinante, o *makespan* do sistema é prejudicado. Ainda assim, uma das grandes vantagens de um sistema de mensageria é a possibilidade de disponibilizar informações de forma assíncrona a diversos clientes - essa característica crucial torna o padrão Publicador-Assinante muito relevante e utilizado.

Outro ponto destacado foi a limitação do padrão de Fila de Prioridades em cenários onde as mensagens são consumidas logo após serem postadas. Nesse tipo de situação, o *broker* RabbitMQ não consegue organizar as mensagens conforme sua prioridade antes de serem consumidas. Com isso, os recursos investidos para implementar uma fila prioritária acabam não trazendo benefícios reais, já que as mensagens são lidas em uma ordem FIFO (*First In First Out*).

O padrão Verificação de Declaração, ou *Claim-Check*, também apresentou limitações, especialmente em cenários com mensagens pequenas. O custo associado ao armazenamento externo de *payloads* quando a carga do sistema é pequena supera as vantagens de desempenho, tornando a estratégia ineficiente. Em situações menos exigentes, é mais eficaz enviar as mensagens íntegras pela fila, sem a complexidade adicional do armazenamento externo.

Conclui-se, portanto, que cada padrão de projeto possui cenários nos quais se destacam, e esses contextos devem ser avaliados com cuidado. A aplicação de padrões inadequados em determinados cenários pode gerar custos adicionais de tempo e recursos, sem oferecer os benefícios esperados. No entanto, quando aplicados a situações às quais se adéquam, todos esses padrões demonstraram vantagens relevantes.

7.1 TRABALHOS FUTUROS

Para ampliar o escopo e a aplicabilidade deste estudo, pode-se efetuar análises semelhantes utilizando outras tecnologias de mensageria além do RabbitMQ. Ferramentas como Apache Kafka, ActiveMQ e Google Pub/Sub possuem arquiteturas e funcionalidades distintas, o que pode influenciar o comportamento dos padrões de projeto em diferentes cenários. Essa abordagem permitiria comparar não apenas os padrões de projeto, mas também o impacto que as características específicas de cada tecnologia têm no desempenho dos sistemas de mensageria.

Outro aspecto importante a ser explorado é a análise dos padrões de projeto sob cenários de maior sobrecarga do sistema. Experimentos futuros podem considerar volumes mais altos de mensagens, cargas contínuas por longos períodos e variações na complexidade das mensagens. Esses testes ajudariam a compreender os limites de escalabilidade dos padrões e identificar possíveis gargalos de desempenho, bem como estratégias para mitigar tais problemas.

Adicionalmente, a inclusão de outros padrões de projeto de sistemas de mensageria, como os mencionados no Capítulo 3, poderia enriquecer os resultados e ampliar a compreensão sobre as melhores práticas em diferentes contextos. Analisar seu desempenho em comparação com os padrões já estudados contribuiria para um mapeamento ainda mais completo das opções disponíveis para o desenvolvimento de sistemas robustos.

Outra vertente de estudo seria a análise do desempenho dos padrões de projeto com mensagens de tamanhos maiores. Esse tipo de teste permitiria avaliar os limites do sistema em termos de consumo de memória, vazão e latência, além de identificar possíveis gargalos ou falhas específicas no processamento de mensagens pesadas. Isso seria especialmente relevante para sistemas que lidam com dados multimídia ou grandes arquivos.

Por fim, um estudo mais aprofundado sobre a saturação e os pontos de falha dos sistemas de mensageria seria altamente relevante. Testes que simulam situações como a quebra de consumidores, produtores ou perda de conexão com o *broker* poderiam revelar a resiliência e a capacidade de recuperação do sistema. Esses experimentos seriam valiosos para projetar sistemas mais confiáveis e para identificar mecanismos de *fallback* ou redundância que minimizem o impacto de falhas críticas no funcionamento dos sistemas de filas de mensagens.

REFERÊNCIAS

AKBULUT, Akhan; PERROS, Harry G. Performance Analysis of Microservices Design Patterns. **JOURNAL OF IEEE INTERNET COMPUTING**, 2019.

BELLAVISTA, Paolo; CORRADI, Antonio; REALE, Andrea. Quality of Service in Wide Scale Publish–Subscribe Systems. **IEEE Communications Surveys & Tutorials**, 2014.

BUSCHMANN, Frank et al. **Pattern-Oriented Software Architecture**: A System of Patterns. 1. ed. Estados Unidos: [s.n.], 1996.

CARLSON, Josiah L. **Redis in Action**. [S.l.]: Manning, 2013.

DRAGICEVIC, Kristijan; BAUER, Daniel. A survey of concurrent priority queue algorithms, p. 1–6, 2008.

DUELL, Michael; GOODSEN, John; RISING, Linda. Non-software examples of software design patterns, p. 120–124, 1997.

EIDE, Eric et al. Integrated Scientific Workflow Management for the Emulab Network Testbed. **Proceedings of the 2006 USENIX Annual Technical Conference**, 2006.

EUGSTER, Patrick Th. et al. The many faces of publish/subscribe. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, 2003.

FERREIRA, Victor A. P. **Design Patterns para Data Science**. 2021. Diss. (Mestrado) – Universidade Federal do Rio Grande do Sul.

FU, GUO; ZHANG, YANFENG; YU, GE. A Fair Comparison of Message Queuing Systems. **IEEE Access**, 2020.

GAWAND, Hemangi; MUNDADA, R. S.; SWAMINATHAN, P. Design Patterns to Implement Safety and Fault Tolerance. **International Journal of Computer Applications**, 2011.

HANMER, Robert S. **Patterns for Fault Tolerant Software**. 1. ed. Naperville, IL, Estados Unidos: [s.n.], 2007.

HAUNTS, Stephen. **Message Queueing with RabbitMQ Succinctly**. Morrisville, NC, Estados Unidos: [s.n.], 2015.

HOHPE, Gregor; WOOLF, Bobby. **Enterprise Integration Patterns**. [S.l.: s.n.], 2002.

HUSSAIN, Shahid; KEUNG, Jacky; KHAN, Arif Ali. The Effect of Gang-of-Four Design Patterns Usage on Design Quality Attributes, p. 263–273, 2017.

JOHANSSON, Lovisa; DOSSOT, David. **RabbitMQ Essentials**: Build distributed and scalable applications with message queuing using RabbitMQ. 2. ed. Birmingham, Reino Unido: [s.n.], 2020.

KHOMH, Foutse; ABTAHIZADEH, S Amirhossein. Understanding the impact of cloud patterns on performance and energy consumption. **Journal of Systems and Software**, Elsevier, v. 141, p. 151–170, 2018.

MICROSOFT AZURE. **Azure Architecture Center - Messaging Patterns**. 2023. Disponível em: <https://learn.microsoft.com/en-us/azure/architecture/patterns/category/messaging>. Acesso em: 5 nov. 2023.

RABBITMQ DOCUMENTATION. **RabbitMQ Tutorials**. 2024. Disponível em: <https://www.rabbitmq.com/docs/>. Acesso em: 8 nov. 2024.

RABBITMQ TUTORIALS. **RabbitMQ Tutorials**. 2024. Disponível em: <https://www.rabbitmq.com/tutorials>. Acesso em: 8 out. 2024.

REAGAN, Rob. **Web Applications on Azure: Developing for Global Scale**. [S.l.]: Apress, 2018. ISBN 978-1-4842-2975-0.

RITTER, Daniel. Using the business process model and notation for modeling enterprise integration patterns. **arXiv preprint arXiv:1403.4053**, 2014.

RÖNNGREN, Robert; AYANI, Rassul. A comparative study of parallel and sequential priority queue algorithms. **ACM Trans. Model. Comput. Simul.**, Association for Computing Machinery, New York, NY, USA, p. 157–209, 1997.

SACHS, Kai; KOUNEV, Samuel; BUCHMANN, Alejandro. Performance modeling and analysis of message-oriented event-driven systems. **Software & Systems Modeling**, Springer, v. 12, p. 705–729, 2013.

SCHMIDT, Douglas C.; O'RYAN, Carlos. Patterns and performance of distributed real-time and embedded publisher-subscriber architectures. **The Journal of Systems and Software**, 2002.

SHAVIT, Nir; ZEMACH, Asaph. Scalable concurrent priority queue algorithms. Association for Computing Machinery, Atlanta, Georgia, USA, p. 113–122, 1999.

SINGH, Harmeet; HASSAN, Syed Imtiyaz. Effect of solid design principles on quality of software: An empirical assessment. **International Journal of Scientific & Engineering Research**, v. 6, n. 4, p. 1321–1324, 2015.

STOPFORD, Ben. **Designing Event-Driven Systems**. [S.l.]: O'Reilly Media, Incorporated, 2018.

VIDELA, Alvaro; WILLIAMS, Jason J.W. **RabbitMQ in Action**: Distributed messaging for everyone. 2. ed. Estados Unidos: [s.n.], 2012.

ANEXO A – CÓDIGO FONTE

O código fonte utilizado para o desenvolvimento deste trabalho está disponível em: https://github.com/paulazomig/MessageBroker.

ANEXO B – ARTIGO ACADÊMICO

Análise de Desempenho de Padrões de Projeto para Sistemas de Filas de Mensagens

Paula Zomignani Oliveira[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

Abstract. *This paper presents a performance analysis of design patterns used in message queuing systems, focusing on evaluating the efficiency of each architecture in terms of throughput, makespan, and latency. Four patterns were implemented and tested: Publisher-Subscriber, Competing Consumers, Priority Queue, and Claim-Check. The results obtained in different scenarios, using the message broker RabbitMQ, reveal that each pattern excels in specific situations, with advantages and disadvantages that should be considered when choosing the best architecture for a given system.*

Resumo. *Este trabalho apresenta uma análise de desempenho de padrões de projeto utilizados em sistemas de filas de mensagens, com foco na avaliação da eficiência de cada arquitetura em termos de vazão, makespan e latência. Foram implementados e testados quatro padrões: Publicador-Assinante, Consumidores Concorrentes, Fila de Prioridades e Verificação de Declaração. Os resultados obtidos em diferentes cenários, utilizando o message broker RabbitMQ, revelam que cada padrão se destaca em situações específicas, com vantagens e desvantagens que devem ser consideradas na escolha da melhor arquitetura para um dado sistema.*

1. Introdução

A crescente demanda por aplicações altamente escaláveis e tolerantes a falhas, capazes de lidar com fluxos de dados intermitentes, impulsiona o uso de sistemas distribuídos. A troca de informações entre componentes e sistemas externos se torna crucial neste contexto, e a mensageria surge como solução para os desafios de comunicação em ambientes distribuídos.

Sistemas de filas de mensagens permitem a comunicação assíncrona entre processos, oferecendo vantagens em cenários onde grandes volumes de dados precisam ser trocados [Akbulut and Perros 2019]. Os serviços produtores enviam mensagens para uma fila, e os consumidores as leem, com a interação ocorrendo através do conteúdo da mensagem, sem necessidade de conhecimento mútuo entre os serviços [Reagan 2018].

A mensageria oferece diversos padrões de projeto, cada um com suas características e objetivos. A comunicação assíncrona, o desacoplamento, a escalabilidade e a tolerância a falhas são alguns dos benefícios que podem ser obtidos com a utilização desses padrões [FU et al. 2020].

No entanto, a complexidade desses sistemas traz desafios, como a dificuldade na implementação e manutenção, a escolha do padrão mais adequado para uma determinada demanda, e a complexidade de testes em cenários de alta demanda.

Este trabalho visa analisar o desempenho de padrões de projeto para filas de mensagens, através da implementação de protótipos e avaliações experimentais. O objetivo é investigar as características de cada padrão e determinar quais cenários são mais adequados para sua aplicação.

1.1. Sistemas de Filas de Mensagens

Sistemas de filas de mensagens baseiam-se na comunicação assíncrona entre serviços produtores e consumidores. Os produtores inserem mensagens em uma fila, e os consumidores as retiram e processam de forma independente. A fila geralmente segue o padrão FIFO (First In, First Out), com a mensagem mais antiga sendo a primeira a ser consumida [Reagan 2018].

Message brokers, como RabbitMQ, ActiveMQ e Kafka, atuam como intermediários entre os serviços, oferecendo funcionalidades como persistência de mensagens, tempo de expiração (TTL), segurança, roteamento, garantia de entrega e escalabilidade [Haunts 2015].

As filas de mensagens proporcionam diversos benefícios, como a escalabilidade, que permite a utilização de múltiplos consumidores para processar mensagens da mesma fila, possibilitando que diferentes partes de uma aplicação escalem independentemente. A elasticidade das filas garante que picos de trabalho sejam absorvidos, acumulando mensagens e evitando o colapso da aplicação. Além disso, o desacoplamento temporal, espacial e de sincronização entre produtores e consumidores é possibilitado por filas de mensagens, promovendo maior flexibilidade e independência entre os serviços. Outro ponto importante é a tolerância a falhas, pois se um receptor falhar, as mensagens são preservadas e processadas posteriormente, garantindo a continuidade do sistema.

1.2. Padrões de Projeto

Padrões de projeto oferecem soluções para problemas recorrentes no desenvolvimento de software, visando aumentar a qualidade, a flexibilidade e a reutilização do código [Buschmann et al. 1996]. Alguns padrões com foto em comunicação foram selecionados para serem aprofundados nesse trabalho. Eles são detalhados a seguir.

1.2.1. Padrões de Projeto para Filas de Mensagens

Diversos padrões de projeto são utilizados em sistemas de filas de mensagens, incluindo:

- Publicador-Assinante (*Publisher-Subscriber*): Um serviço publica mensagens em um canal específico, e vários serviços consumidores podem se inscrever para receber as mensagens daquele canal;
- Consumidores Concorrentes (*Competing Consumers*): Múltiplos consumidores concorrem para consumir mensagens de uma única fila, o que permite um processamento paralelo e um melhor desempenho em cenários de alta demanda;
- Fila de Prioridade (*Priority Queue*): As mensagens são ordenadas de acordo com a prioridade definida, permitindo que mensagens mais importantes sejam consumidas primeiro;

- Verificação de Declaração (*Claim-Check*): As mensagens grandes são divididas em uma parte de declaração e uma parte de payload. O payload é armazenado em uma estrutura externa, como um banco de dados, e a declaração, com o identificador do payload, é enviada pela fila.

2. Metodologia

Para avaliar o desempenho dos padrões de projeto, foram implementados protótipos utilizando o *message broker* RabbitMQ e a linguagem de programação Java 17. Foram realizados testes em diferentes cenários, utilizando a plataforma Emulab, um ambiente de pesquisa e desenvolvimento que fornece infraestrutura para experimentação em sistemas distribuídos [Eide et al. 2006]. Os experimentos foram configurados para avaliar o desempenho dos padrões em relação à vazão, *makespan* e latência.

2.1. Cenários de Teste

Os testes foram realizados em cenários com diferentes configurações, variando o tamanho das mensagens (pequenas, com 5 bytes, e grandes, com 2.5 MB), o número de consumidores (1 e 10), e a forma de preenchimento da fila - pré-populada com 50.000 mensagens antes do início do consumo, ou com mensagens inseridas e consumidas simultaneamente.

A Tabela 1 resume os cenários de teste aplicados a cada padrão de projeto.

Table 1. Cenários de Teste

Cenário	**Fila**	**Tamanho da Mensagem**	**Nº de Consumidores**
Cenário 1	Pré-populada	5 bytes	1 Consumidor
Cenário 2	Pré-populada	5 bytes	10 Consumidores
Cenário 3	Pré-populada	2.5 MB	1 Consumidor
Cenário 4	Pré-populada	2.5 MB	10 Consumidores
Cenário 5	Pós-populada	5 bytes	1 Consumidor
Cenário 6	Pós-populada	5 bytes	10 Consumidores
Cenário 7	Pós-populada	2.5 MB	1 Consumidor
Cenário 8	Pós-populada	2.5 MB	10 Consumidores

Autoria própria

2.2. Métricas de Desempenho

As seguintes métricas foram coletadas para avaliar o desempenho dos padrões:

- *Makespan*: Tempo total para o serviço consumidor ler todas as mensagens enviadas pelo serviço produtor;
- Vazão: Número de mensagens consumidas por segundo;
- Latência: Tempo que uma mensagem permanece na fila.

3. Resultados

Os resultados da análise de desempenho demonstram que cada padrão de projeto se destaca em situações específicas. O padrão Publicador-Assinante, por exemplo, apresenta a menor queda de desempenho com o aumento do número de consumidores, mas

seu *makespan* é maior devido à falta de concorrência, já que cada consumidor recebe uma cópia da mensagem. O padrão Consumidores Concorrentes, por sua vez, apresenta maior vazão com a concorrência entre os consumidores, porém a ordem das mensagens pode ser alterada. A Fila de Prioridades, como esperado, garante que as mensagens prioritárias sejam consumidas mais rapidamente, mas sua complexidade impacta o *makespan*.

O padrão Verificação de Declaração é ideal para mensagens grandes, pois seu desempenho é superior a outros padrões nesse contexto. No entanto, sua ineficiência com mensagens pequenas, devido ao custo do armazenamento externo, é um ponto a ser considerado.

3.1. Análise do *Makespan*

Os resultados mostraram que o padrão Publicador-Assinante apresentou o maior *makespan*, principalmente em cenários com 10 consumidores. Isso se deve ao fato de que cada consumidor recebe uma cópia da mensagem, aumentando o tempo total de processamento. O padrão Consumidores Concorrentes apresentou um *makespan* menor, especialmente em cenários com mensagens grandes, devido à concorrência entre consumidores, que divide a carga de trabalho.

A Fila de Prioridades também teve um *makespan* considerável, especialmente com mensagens pequenas, devido à complexidade da ordenação. Por fim, o padrão Verificação de Declaração, ao ser usado com mensagens grandes, apresentou o menor *makespan*, devido à redução do tráfego na fila, pois apenas a declaração é enviada e o payload é recuperado do armazenamento externo.

3.2. Análise da Vazão

A vazão do sistema foi maior no padrão Consumidores Concorrentes, especialmente em cenários com mensagens pequenas, devido à concorrência. O Publicador-Assinante apresentou uma vazão menor, mas ainda eficiente, principalmente em cenários com mensagens grandes. Já a Fila de Prioridades teve uma vazão menor que os outros padrões, devido à complexidade da ordenação. O padrão Verificação de Declaração, quando utilizado com mensagens grandes, apresentou uma vazão similar ao padrão Consumidores Concorrentes, mostrando sua eficiência na redução do tráfego na fila.

3.3. Análise da Latência

A latência média foi maior no padrão Publicador-Assinante, especialmente para mensagens grandes, devido à falta de concorrência. Os outros padrões apresentaram latência menor e mais consistente, com baixo desvio padrão, demonstrando que a concorrência e a estratégia de dividir a carga de trabalho contribuem para um tempo de resposta mais rápido.

4. Discussão

Os resultados da análise de desempenho indicam que a escolha do padrão de projeto ideal para um sistema de filas de mensagens depende de diversos fatores, incluindo o tamanho das mensagens, o número de consumidores, a necessidade de priorização de mensagens e a tolerância a alterações na ordem das mensagens.

O padrão Publicador-Assinante é ideal para cenários onde a publicação de mensagens para diversos consumidores é crucial, mas a performance em relação ao *makespan* é menos relevante. O padrão Consumidores Concorrentes é mais eficiente em cenários de alta demanda, com múltiplos consumidores processando mensagens em paralelo. A Fila de Prioridades garante que as mensagens mais importantes sejam consumidas primeiro, mas a complexidade do padrão pode impactar o desempenho. O padrão Verificação de Declaração é útil para o envio de mensagens grandes, reduzindo o tráfego na fila, mas é menos eficiente com mensagens pequenas.

5. Conclusão

A análise dos resultados demonstra que cada padrão de projeto apresenta vantagens e desvantagens que devem ser consideradas ao escolher a melhor arquitetura para um sistema de filas de mensagens.

Este estudo fornece subsídios para a escolha e implementação de padrões de projeto em sistemas de filas de mensagens, com base em um entendimento profundo do desempenho de cada padrão em diferentes cenários.

5.1. Trabalhos Futuros

Diversas áreas podem ser exploradas em pesquisas futuras:

- Análise de desempenho utilizando outras tecnologias de mensageria, como Apache Kafka, ActiveMQ e Google Pub/Sub;
- Investigação do desempenho dos padrões em cenários de maior sobrecarga do sistema, com volumes mais altos de mensagens, cargas contínuas e mensagens mais complexas;
- Inclusão de outros padrões de projeto de sistemas de mensageria no estudo;
- Análise do desempenho dos padrões com mensagens de tamanhos ainda maiores;
- Investigação da saturação e dos pontos de falha dos sistemas de mensageria.

A realização dessas pesquisas contribuirá para um entendimento ainda mais completo dos padrões de projeto para sistemas de filas de mensagens, auxiliando no desenvolvimento de soluções mais eficientes, confiáveis e resilientes.

6. References

References

Akbulut, A. and Perros, H. G. (2019). Performance analysis of microservices design patterns. *JOURNAL OF IEEE INTERNET COMPUTING*.

Buschmann, F., Meunier, R., Rohnert, H., Sornmerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture*. Estados Unidos, 1 edition.

Eide, E., Stoller, L., Stack, T., Freire, J., and Lepreau, J. (2006). Integrated scientific workflow management for the emulab network testbed. *Proceedings of the 2006 USENIX Annual Technical Conference*.

FU, G., ZHANG, Y., and YU, G. (2020). A fair comparison of message queuing systems. *IEEE Access*.

Haunts, S. (2015). *Message Queueing with RabbitMQ Succinctly*. Morrisville, NC, Estados Unidos.

Reagan, R. (2018). *Web Applications on Azure: Developing for Global Scale*. Apress.