UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MARCELO CAMPANELLI NOBREGA
MATHEUS ARAUJO BORGES

Análise comparativa de soluções de mensageria e filtragem de dados em tempo real

RIO DE JANEIRO
2025

MARCELO CAMPANELLI NOBREGA
MATHEUS ARAUJO BORGES

Análise comparativa de soluções de mensageria e filtragem de dados em tempo real

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Vinícius Gusmão Pereira de Sá

RIO DE JANEIRO
2025

Nóbrega, Marcelo Campanelli
N754 Análise comparativa de soluções de mensageria e filtragem de dados em tempo real / Marcelo Campanelli Nóbrega e Matheus Araujo Borges. – Rio de Janeiro, 2025.

76 f.

Orientador: Vinícius Gusmão Pereira de Sá.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação)- Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2025.

1. Mensageria. 2. Sistemas distribuídos. 3. Comunicação síncrona. 4. Filtragem de dados. 5. Microserviços. I. Borges, Matheus Araujo. II. Sá, Vinícius Gusmão Pereira de (Orient.). III. Universidade Federal do Rio de Janeiro, Instituto de Computação. IV. Título.

MARCELO CAMPANELLI NOBREGA
MATHEUS ARAUJO BORGES

Análise comparativa de soluções de mensageria e filtragem de dados em tempo real

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 20 de fevereiro de 2024

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
VINICIUS GUSMAO PEREIRA DE SA
Data: 13/03/2025 10:12:48-0300
Verifique em https://validar.iti.gov.br

D.Sc. Vinícius Gusmão Pereira de Sá
(Instituto de Computação- UFRJ)

gov.br Documento assinado digitalmente
SILVANA ROSSETTO
Data: 12/03/2025 17:44:35-0300
Verifique em https://validar.iti.gov.br

D.Sc. Silvana Rossetto
(Instituto de Computação- UFRJ)

gov.br Documento assinado digitalmente
MARIA LUIZA MACHADO CAMPOS
Data: 12/03/2025 17:18:22-0300
Verifique em https://validar.iti.gov.br

Ph.D. Maria Luiza Machado Campos
(Instituto de Computação- UFRJ)

RESUMO

Este trabalho visa contribuir para a área de mensageria síncrona e filtragem de dados. O conceito de mensageria é amplamente utilizado como modelo para comunicação entre sistemas distribuídos, sendo a comunicação síncrona essencial em diversos cenários onde a prioridade é a baixa latência e a continuidade da transmissão, em detrimento da entrega garantida de cada mensagem. Entre esses cenários, destacam-se: processamento de informações em videogames online, monitoramento de radares, rastreamento de objetos em movimento e telemetria em larga escala por meio de sensores IoT(Internet das Coisas), entre outros. Existem diversas soluções de código aberto que podem atuar como transmissores ou redirecionadores de mensagens nesses contextos. Diante disso, este estudo realiza uma análise comparativa de quatro ferramentas amplamente utilizadas (ActiveMQ Artemis, Apache Pulsar, Apache Kafka e RabbitMQ) em cenários onde a perda de mensagens é tolerável. Considerando que essas ferramentas não foram originalmente projetadas exclusivamente para contextos de tolerância a perdas, o trabalho também propõe a implementação de uma solução própria, desenvolvida especificamente para essa condição. A solução proposta utiliza, em sua arquitetura, a linguagem Java e o protocolo UDP, além de incorporar funcionalidades de filtragem de conteúdo e filtragem de frequência (absoluta ou relativa) para redução do espaço amostral.
O objetivo central é avaliar o desempenho, a escalabilidade e a facilidade de uso de cada uma das cinco soluções (as quatro ferramentas mencionadas e a solução própria) em cenários distintos, bem como analisar a solução própria de maneira particular. Os resultados obtidos evidenciam os cenários em que cada ferramenta pode se destacar ou apresentar limitações, especialmente no que diz respeito à latência e à vazão de dados. Além disso, são estabelecidas comparações sob uma perspectiva financeira entre a solução própria e outras ferramentas de código fechado.

Palavras-chave: mensageria; arquitetura de microsserviços; comunicação entre sistemas; desempenho; escalabilidade; latência; Apache Pulsar; ActiveMQ Artemis; Apache Kafka; RabbitMQ; filtragem de dados; tolerância a perdas; protocolo UDP; Java; sistemas distribuídos; telemetria; IoT; vazão de dados; solução proprietária; análise comparativa

ABSTRACT

This work aims to contribute to the field of synchronous messaging and data filtering. The concept of messaging is widely used as a model for communication between distributed systems, with synchronous communication being essential in various scenarios where the priority is low latency and transmission continuity, at the expense of guaranteed delivery of each message. Among these scenarios, the following stand out: processing information in online video games, radar monitoring, tracking moving objects, and large-scale telemetry through IoT (Internet of Things) sensors, among others.

There are several open-source solutions that can act as message transmitters or redirectors in these contexts. Therefore, this study conducts a comparative analysis of four widely used tools (ActiveMQ Artemis, Apache Pulsar, Apache Kafka, and RabbitMQ) in scenarios where message loss is tolerable. Considering that these tools were not originally designed exclusively for loss-tolerant contexts, the work also proposes the implementation of a custom solution, specifically developed for this condition. The proposed solution uses, in its architecture, the Java programming language and the UDP protocol, in addition to incorporating content filtering and frequency filtering (absolute or relative) functionalities for sample space reduction.

The main objective is to evaluate the performance, scalability, and ease of use of each of the five solutions (the four mentioned tools and the custom solution) in different scenarios, as well as to analyze the custom solution in particular. The results highlight the scenarios in which each tool can excel or present limitations, especially regarding latency and data throughput. Furthermore, financial comparisons are established between the custom solution and other closed-source tools, highlighting advantages and disadvantages from an economic perspective.

Keywords: messaging; microservices architecture; communication between systems; performance; scalability; latency; Apache Pulsar; ActiveMQ Artemis; Apache Kafka; RabbitMQ; data filtering; loss tolerance; UDP protocol; Java; distributed systems; telemetry; IoT; data throughput; proprietary solution; comparative analysis.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AMQP	Advanced Message Queuing Protocol
MQTT	Message Queuing Telemetry Transport
RTP	Real-time Transport Protocol
UDP	User Datagram Protocol
M2M	Machine-to-Machine
IOT	Internet of Things
MOM	Message-Oriented Middleware
POJO	Plain Old Java Objects
GCP	Google Cloud Platform
VM	Virtual Machine
AWS	Amazon Web Services
SSD	Solid-State Drive
RAM	Random Access Memory

SUMÁRIO

1 INTRODUÇÃO

A Era da Informação perdura desde meados de 1950. Ao longo desses quase 80 anos, a humanidade presenciou a constante evolução dos sistemas tecnológicos e, por consequência, também o aumento do volume dos dados que são transmitidos na grande rede de computadores. Conforme destacado por Mayer-Schönberger e Cukier (MAYER-SCHöNBERGER; CUKIER, 2013), o volume de dados gerados globalmente tem crescido de forma exponencial, impulsionado por avanços tecnológicos como a Internet das Coisas (*IoT*), que conecta bilhões de dispositivos, e pelo aumento do uso de serviços de *streaming*, que geram fluxos contínuos de informações. Segundo o livro, em 2013, estimava-se que a humanidade gerava 2,5 exabytes de dados por dia, um volume que supera em muito a capacidade de processamento de sistemas tradicionais. Esse crescimento é alimentado não apenas pela quantidade de dispositivos conectados, mas também pela datificação, que é a transformação de atividades cotidianas, como interações sociais e transações comerciais, em dados quantificáveis e analisáveis. A combinação desses fatores tem levado a um cenário em que o volume de dados aumenta a cada ano, um ritmo que desafia a capacidade de armazenamento e processamento existentes. Nesse contexto, a demanda por desempenho na comunicação síncrona entre sistemas distribuídos também aumentou bastante, principalmente em serviços que utilizam em seu negócio o *streaming* de dados.

A mensageria de dados surgiu para melhorar a eficiência e logística da troca de informações entre diferentes aplicações. Combinando o conceito de mensageria com arquitetura de microsserviços (HAQ, 2018), ela possibilita a uma determinada aplicação tratar as mensagens recebidas ao mesmo tempo em que continua realizando suas atividades originais, assim desacoplando serviços, otimizando a troca de informações e permitindo novos horizontes para a interoperabilidade. As ferramentas que implementam esse tipo de mensageria geralmente usam um servidor (*Message Broker*) idealizado para processar e suportar o envio e recebimento de mensagens.

A filtragem síncrona permite que só os dados que interessam cheguem ao consumidor final, poupando o envio de pacotes desnecessários, e permitindo ao usuário que receba somente o que realmente interessa. Os casos de uso de ferramentas para mensageria síncrona de dados podem ser dos mais variados, envolvendo principalmente captação de anomalias, monitoramento de operações, e também análises estatísticas de forma mais geral. Um exemplo de aplicação seria: em uma companhia aérea, podem ser enviadas informações de todos os vôos ativos em um dado momento, essas que podem ser úteis para um determinado sistema que precisa desses dados, consumir apenas as amostras de um certo modelo de avião, ou amostras de dados de aviões em uma região pré-estabelecida, podendo ser possível querer combinar filtros. A mensageria implementada nesse caso poderia favorecer muitos setores do negócio: segurança, possibilitando a geração de alertas

o mais rápido possível caso quaisquer anomalias aparecessem (desvio de rotas, condição das aeronaves, entre outros), supervisão, permitindo com que determinados vôos sejam acompanhados e monitorados, com o melhor nível de detalhe possível, time de análises de dados coletando e gerando informações a partir dos dados e análises feitas, etc. Tais benefícios favorecem uma organização como um todo. Esse é apenas um cenário de aplicação, mas existem diversas possibilidades de situações em que a mensageria pode ser útil.

Existem diversas soluções de código aberto que podem atuar como transmissores ou redirecionadores de mensagens nesses contextos. Diante disso, este estudo realiza uma análise comparativa de quatro ferramentas amplamente utilizadas (ActiveMQ Artemis, Apache Pulsar, Apache Kafka e RabbitMQ) em cenários onde a perda de mensagens é tolerável. Considerando que essas ferramentas não foram originalmente projetadas exclusivamente para contextos de tolerância a perdas, o trabalho também propõe a implementação de uma solução própria, desenvolvida especificamente para essa condição, visando se tornar uma solução viável principalmente para os seguintes casos de uso: processamento de informações de jogadores em videogames online, monitoramento de radares, rastreamento de objetos em movimento e telemetria em larga escala por meio de sensores *IoT*. A solução proposta utiliza, em sua arquitetura, a linguagem Java e o protocolo UDP, além de incorporar funcionalidades de filtragem de conteúdo, com mensagens em formatos de serialização do tipo **ProtoBuffer** (GOOGLE, 2001) e, filtragem de frequência (absoluta ou relativa) para redução do espaço amostral. Essa solução será referenciada como **Carteiro**. Diferencia-se das demais ferramentas, pois o envio e consumo das mensagens ocorrerão necessariamente de forma imediata, caracterizando uma comunicação síncrona, devido à natureza do protocolo utilizado, enquanto nas outras, existe a possibilidade de o consumo ser assíncrono, onde o consumidor final pode receber somente no momento que o convém.

O objetivo central é avaliar o desempenho, a escalabilidade e a facilidade de uso de cada uma das cinco soluções (as quatro ferramentas mencionadas e a solução própria) em cenários distintos, bem como analisar a solução própria de maneira particular. Os resultados obtidos evidenciam os cenários em que cada ferramenta pode se destacar ou apresentar limitações, especialmente no que diz respeito à latência e à vazão de dados. Além disso, são estabelecidas comparações sob uma perspectiva financeira entre a solução própria e outras ferramentas de código fechado.

Por fim, é importante esclarecer que o termo “tempo real”, utilizado no título deste trabalho, refere-se, no contexto específico para o qual a solução própria foi desenvolvida, ao caráter síncrono e imediato da comunicação entre as partes envolvidas na mensageria e filtragem de dados. Embora o termo “tempo real” seja frequentemente associado a sistemas com prazos rígidos e determinísticos, neste trabalho ele é utilizado para enfatizar a sincronia na troca de informações, sem implicar em garantias de prazos estritos.

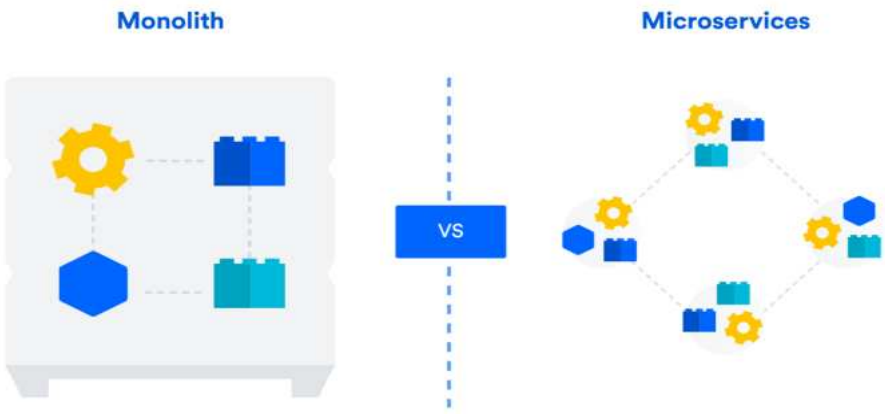
2 CONCEITOS

Antes de introduzir o sistema proposto e as ferramentas escolhidas a serem comparadas, é importante contextualizarmos alguns conceitos fundamentais para estabelecer um entendimento melhor da pesquisa e do tema de mensageria no que tange à transferência de dados de maneira síncrona.

2.1 ARQUITETURA DE MICROSSERVIÇOS

Os microsserviços são um tipo de arquitetura em que um servidor é desenvolvido de modo a ser subdividido em serviços menores, com cada um tendo sua própria independência e responsabilidade, dividindo a atribuição de tarefas e os recursos da aplicação. É ideal que cada microsserviço consiga se comunicar com outros por meio de interfaces simples (APIs) para resolver problemas internos do aplicativo.

Figura 1 – Monolito e Microsserviços



Microservices vs. monolithic architecture, ilustração das arquiteturas mencionadas, Atlassian, Chandler Harris.

A figura 1, extraída da leitura (HARRIS, 2021), ilustra como a arquitetura de monolito (*Monolith*) inclui todas as operações, como por exemplo acesso a banco e/ou interfaces com usuário, o seguinte *website* (HAQ, 2018) exemplifica casos em que uma arquitetura pode ser vantajosa em relação a outra. Os microsserviços se propõem a distribuir e subdividir o serviço "completo" de modo a trabalhar concorrentemente, de maneira distribuída, em que cada segmento só precise processar o menor escopo possível.

Soluções centralizadas, como os sistemas monolíticos, são frequentemente escolhidas quando a simplicidade de desenvolvimento, implantação e manutenção são prioridades.

Elas oferecem um ambiente integrado, onde todas as funcionalidades compartilham um único banco de dados e execução em um mesmo servidor ou conjunto de servidores. Em contrapartida, soluções distribuídas, como os microsserviços, são projetadas para escalabilidade, permitindo que diferentes partes do sistema operem de forma independente. Essa abordagem é vantajosa para aplicações que exigem alta disponibilidade, processamento paralelo e resiliência a falhas, pois um serviço pode continuar operando mesmo que outro falhe. A escolha entre uma arquitetura centralizada ou distribuída depende, portanto, dos requisitos de escalabilidade, desempenho e complexidade de cada aplicação.

Apesar das vantagens dos microsserviços, essa abordagem traz desafios significativos. A maior complexidade na gestão e monitoramento do sistema exige ferramentas específicas para rastrear *logs*, latências e comunicação entre serviços. Além disso, manter a consistência dos dados se torna mais desafiador, pois cada serviço pode possuir seu próprio banco de dados ou armazenar informações de maneira distribuída, necessitando de estratégias para garantir consistência de eventos e mecanismos de coordenação entre serviços. Outro fator a considerar é o *overhead* de comunicação, pois a troca de mensagens entre microsserviços pode adicionar latência e requerer otimizações para garantir eficiência. Dessa forma, a adoção desse modelo deve levar em conta tanto seus benefícios quanto os custos adicionais de implementação e manutenção.

Todavia, essa abordagem combina muito bem com o processamento de mensagens de forma síncrona, justamente por dividir os serviços: caso um monolito (servidor único principal) tenha que se preocupar em processar e receber mensagens de maneira síncrona, ele perderia boa parte da sua capacidade de processamento com essa comunicação, que não necessariamente é sua tarefa essencial, o que resultaria em perda de desempenho e disputa por recursos. Alguns dos atributos proporcionados por esse método são: desacoplamento, agilidade, flexibilidade e escalabilidade.

2.2 PROTOCOLOS DE MENSAGERIA

O uso de protocolos na mensageria é análogo a protocolos de redes ou governamentais, é um conjunto de práticas e padrões que ditam como algum procedimento deve seguir, de forma a estabelecer ordem no decorrer dos eventos.

Os sistemas e aplicações precisam se comunicar de alguma forma, e estruturar/padronizar as formas de interação. Com esse intuito, protocolos são adotados, definindo conjuntos de regras e convenções. Esse uso permite que o desenvolvimento de softwares para comunicação entre sistemas seja padronizado de uma forma bem determinada, não importando em que região/linguagem o código seja originado.

Como posteriormente na pesquisa iremos falar sobre diferentes ferramentas de mensageria, precisamos definir alguns tipos de protocolo de mensageria já existentes, pois cada uma pode disponibilizar diferentes protocolos dentro do escopo da própria ferramenta.

2.2.1 AMQP (Advanced Message Queueing Protocol)

AMQP (LEE, 2024) é um protocolo para enfileiramento de mensagens, que fornece confiabilidade e segurança de forma flexível na troca de mensagens. Nele está presente o recurso em que as mensagens ficam armazenadas em uma fila, até que o consumidor final deseje consumi-las em momento oportuno. É bastante utilizado para aplicações dependentes de eventos.

2.2.2 MQTT (Message Queueing Telemetry Transport)

MQTT (OASIS, 2024) é um protocolo leve de publicação-assinatura projetado para comunicação máquina a máquina (M2M) e aplicativos de Internet das Coisas (IoT). Ele oferece uma solução de mensagens de baixa largura de banda e alta latência para dispositivos com recursos limitados. É bastante usado na coleta de dados por sensores e, por aplicações IoT, além de aplicações de monitoramento.

2.2.3 RTP (Real-time Transport Protocol)

O RTP é um protocolo projetado para aplicações de *streaming* (PARK; PARK, 2000), como conferências de áudio e vídeo, e pode ser usado para mensageria também. Tem foco na entrega de dados de baixa latência e não garante a entrega ou ordenação de mensagens. É o principal protocolo que temos como base para o desenvolvimento da pesquisa.

2.3 MENSAGERIA SÍNCRONA COM MICROSSERVIÇOS

A mensageria síncrona combina os dois conceitos abordados anteriormente para permitir o processamento contínuo de grandes volumes de dados; o seguinte artigo (WANG; GILL; LU, 2019) conceitualiza e mostra casos de uso. Essa integração deve ocorrer de tal forma que seja necessário o desenvolvimento de um software que suporte a implementação de microsserviços relacionados à mensageria. De maneira geral, os microsserviços são as entidades envolvidas na produção/consumo das mensagens, e também a entidade da ferramenta de mensageria (Message Broker), que controla toda a lógica do fluxo dos dados que passam por essa corretagem.

Note que a figura 2, ilustrada com *draw.io* (LTD, 2011), existem diferentes quantidades de produtores, brokers e consumidores. Isso está ilustrado para demonstrar que um sistema de mensageria completo deve permitir que diversos produtores (não apenas um) consigam publicar mensagens no sistema, que possa haver um ou mais nós de message brokers, e também que diferentes consumidores possam consumir a mesma informação, no sentido de que, quando um consumidor recebe uma mensagem, ela ainda estará disponível para outros consumidores receberem a mesma.

Figura 2 – Principais entidades envolvidas na mensageria

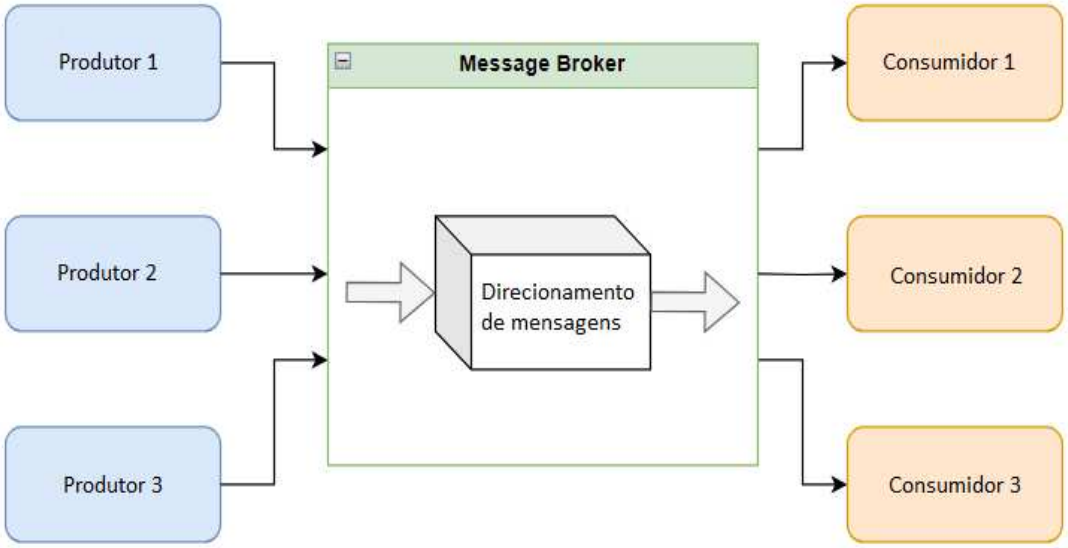


Ilustração simplificada de modelagem de mensageria feita

2.3.1 Produtores

Cada Produtor (Producer) em um sistema de mensageria é uma entidade ou componente que gera e envia mensagens para um intermediário de mensagens (Message Broker). O papel do produtor é criar e publicar mensagens que contêm dados ou eventos que precisam ser processados por outros componentes do sistema. Os produtores são responsáveis por garantir que as mensagens sejam formatadas corretamente e entregues ao broker de forma confiável.

2.3.2 Corretor de mensagens

Um Intermediário de Mensagens (Message Broker) é uma entidade central e essencial em um sistema de mensageria, que recebe, armazena e encaminha mensagens dos produtores para os consumidores. O broker é responsável por garantir a entrega ordenada e confiável das mensagens, gerenciar filas de mensagens e facilitar a comunicação entre os diferentes componentes do sistema. Ele também pode oferecer recursos adicionais, como persistência de mensagens, roteamento, filtragem, balanceamento de carga e garantia de entrega.

2.3.3 Consumidores

Um Consumidor (Consumer) em um sistema de mensageria é uma entidade ou componente que recebe e processa mensagens provenientes do intermediário de mensagens.

O papel do consumidor é ler as mensagens recebidas do broker, processar os dados ou eventos contidos nelas e realizar as ações necessárias baseadas nessas informações. Os consumidores podem ser configurados para tratar mensagens de maneira síncrona ou assíncrona e são responsáveis por confirmar a recepção das mensagens para garantir que elas sejam removidas da fila e não sejam processadas novamente.

2.3.4 Persistência de mensagens

A persistência de mensagens em um software de mensageria (WALTHER et al., 2018) refere-se à capacidade do sistema de armazenar mensagens de forma duradoura, garantindo que elas não sejam perdidas em caso de falhas no sistema, como quedas de energia, erros de software ou falhas de hardware. Este recurso é crucial em aplicações onde a entrega confiável de mensagens é essencial, pois assegura que todas as mensagens enviadas sejam eventualmente entregues aos destinatários, independentemente de qualquer interrupção temporária.

Ela é geralmente implementada através do armazenamento das mensagens em discos rígidos ou outros meios de armazenamento não voláteis antes de serem entregues aos consumidores. Isso permite que o sistema recupere e reenvie as mensagens em caso de falhas, mantendo a integridade e a continuidade dos dados.

Outra vantagem que a implementação da persistência traz é o desacoplamento na comunicação entre os sistemas, pois a entidade consumidora pode receber as mensagens de maneira assíncrona, de forma que não é necessária uma comunicação imediata.

Porém, há cenários em que a persistência de mensagens pode ser descartada e desabilitada, como em streams de dados, ou cenários onde a entrega rápida e o processamento contínuo dos dados são mais importantes do que a garantia de que todas as mensagens sejam recebidas.

No entanto, vale destacar que para a proposta de implementação presente nessa pesquisa, optou-se por não utilizar a persistência de mensagens. Esta escolha é motivada pela natureza dos problemas a serem atendidos (streams de dados) e pela tolerância à perda de mensagens. Dessa forma, a implementação utiliza o protocolo UDP, que, diferente do TCP, é um protocolo que não garante a entrega, a ordem ou a integridade das mensagens, mas é mais veloz no envio das mesmas.

Dessa forma, será priorizada a baixa latência e o desempenho, em relação à confiabilidade absoluta. Foi decidido que esse seria o modelo mais adequado em um cenário em que a perda de algumas mensagens é aceitável.

2.3.5 Padrões de canalização de mensagens

Os padrões de canalização de mensagens são formas padronizadas de projetar e implementar caminhos de comunicação em sistemas de mensageria. Esses padrões abordam

desafios comuns em mensagens, como roteamento de mensagens, transformação e tratamento de diferentes cenários de comunicação. Eles ajudam a criar arquiteturas de mensagens mais robustas e mais fáceis de desenvolver/manter.

Nos baseamos na literatura (HOHPE; WOOLF, 2003) para introduzir os dois padrões mais comumente utilizados na mensageria:

- **Canal Ponto-a-Ponto**:
  Um canal ponto a ponto garante que apenas um receptor consuma qualquer mensagem. Se o canal tem vários receptores, apenas um deles pode consumir com sucesso uma mensagem específica. Se vários receptores tentarem consumir uma única mensagem, o canal garante que apenas um deles seja bem-sucedido, de modo que os receptores não precisem se coordenar entre si. O canal ainda pode ter múltiplos receptores para consumir múltiplas mensagens simultaneamente, mas apenas um único receptor consome qualquer mensagem. Esse modelo introduz o fenômeno de competição por mensagens.

- **Canal Publicação/Subscrição**:
  Quando é necessário a replicação de mensagens para todos receptores interessados, publicação/subscrição é a melhor alternativa de canalização. Existem alguns padrões para envio de mensagens em broadcast. O padrão mais comum é com uso de tópicos. Funciona de tal maneira em que as mensagens são publicadas em um tópico, e vários assinantes podem receber as mensagens desse tópico. Este padrão separa os produtores (publicadores) e os consumidores (assinantes) de mensagens, permitindo uma comunicação flexível e escalável.

Figura 3 – Modelo de Publicação/Subscrição

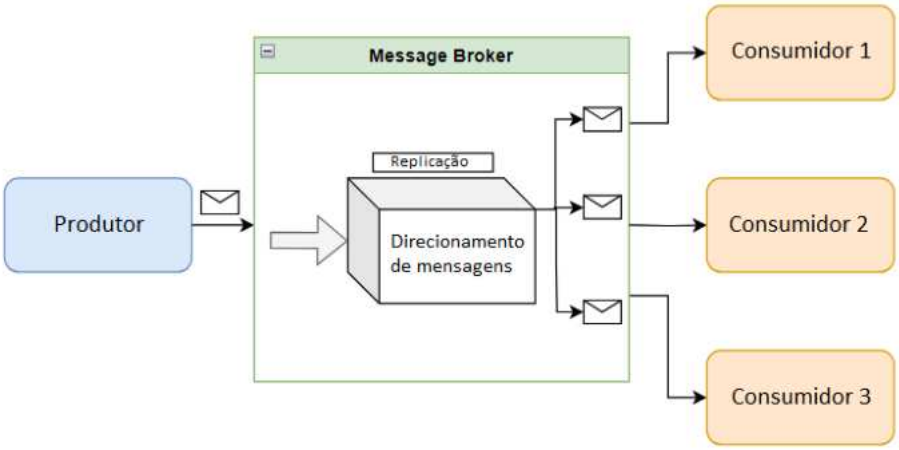


Ilustração de só um produtor enviando a mesma mensagem para mais de um consumidor

A figura 3, modelada com o *draw.io* (LTD, 2011) representa a ilustração de só um produtor enviando a mesma mensagem para mais de um consumidor.

2.3.6 Abordagem de arquitetura

Cada ferramenta/software que implementa esses microsserviços precisa ter como fundamento as entidades citadas anteriormente, essas que podem diferir quanto às suas abordagens em relação à construção da arquitetura das partes e como se relacionam.

2.3.7 Métricas em Mensageria

Métricas em mensageria são indicadores quantitativos que permitem avaliar o desempenho e a saúde de um *middleware* orientado à mensagem (MOM). Essas métricas são cruciais para garantir a qualidade do serviço, identificar gargalos e tomar decisões informadas sobre a otimização do sistema.

- **Throughput**:
  Representa o número de mensagens processadas por unidade de tempo (por exemplo, mensagens por segundo). Indica a capacidade do sistema de lidar com grandes volumes de mensagens, sendo fundamental para aplicações com alta demanda.

- **Latência**:
  Mede o tempo que uma mensagem leva para ser processada desde o envio até a entrega. A baixa latência é crucial em aplicações que envolvem comunicação síncrona, como sistemas de *trading* ou jogos online.

- **Consumo de Recursos**:
  Mede o uso de recursos do sistema, como CPU, memória e disco. Permite identificar gargalos de recursos e otimizar o uso do sistema.

- **Número de Conexões**:
  Indica o número de clientes conectados ao sistema. Ajuda a avaliar a capacidade do sistema de lidar com um grande número de clientes simultâneos.

- **Taxa de Perda**:
  A taxa de perda de mensagens é uma métrica crucial em sistemas de mensageria que indica a proporção de mensagens enviadas que não são entregues ao destinatário. É a porcentagem de mensagens que se perdem no caminho.

- **Taxa de Consumo**:
  A taxa de consumo de mensagens é uma métrica que indica a proporção de mensagens consumidas pelo destinatário em relação ao total produzido que deveria ser entregue ao mesmo. É a porcentagem de mensagens que não se perdem no caminho.

O monitoramento das métricas em sistemas de mensageria é fundamental para garantir a saúde e o desempenho desses sistemas. Ao acompanhar indicadores como *throughput*, latência e taxa de erro, é possível identificar problemas de forma precoce, otimizar recursos, garantir a qualidade do serviço e tomar decisões estratégicas sobre a infraestrutura de mensagens. Essas métricas permitem que as organizações garantam a escalabilidade de seus sistemas, atendendo às demandas de aplicações cada vez mais complexas e exigentes.

3 TRABALHOS RELACIONADOS

Uma das motivações para se implementar um sistema de mensageria usando microsserviços é a possibilidade de se desenvolver um sistema que possibilite a detecção de anomalias de forma síncrona. Isso foi feito anteriormente, por exemplo, por um grupo da Universidade de Tongji, em Shanghai, que publicou o artigo (DU; YUHE, 2018), onde foi implementado monitoramento ativo de microsserviços conteinerizados, coletando dados de contêineres e processando-os para identificar anomalias de desempenho.

A comparação entre ferramentas de código aberto é incentivada e necessária para determinar melhores casos de uso. O artigo (CHY et al., 2023) faz um estudo em que as ferramentas RocketMQ, Kafka, Artemis e Pulsar são comparadas. É o trabalho que mais se assemelha à presente pesquisa, diferindo justamente nas ferramentas que entram na abordagem, pois a presente pesquisa inclui a comparação com uma proposta de implementação própria, e ao invés do RocketMQ, analisa o RabbitMQ. O artigo também se aprofunda em métricas, como coletor de lixo e utilização de CPU e memória.

O artigo (TALHAOUI, 2017) se relaciona com o presente trabalho, pois aborda as questões de processamento de *streamings*, ou seja, aborda a ponta do consumidor de mensagens do Carteiro, se ele for o software utilizado para a mensageria. Nele são discutidas as melhores formas de processamento, apresentando ferramentas como Apache Hadoop MapReduce, Apache Spark, e Storm. Também é apresentada uma proposta de arquitetura de módulo de processamento em tempo real.

O estudo (T.A MARK D. GRIFFITHS, 2004) aborda o tema de coleta de dados de jogadores de videogames para pesquisa. As formas apresentadas de coletar dados, por ser um artigo antigo, não abordam a ingestão de informações de forma síncrona. O presente trabalho se relaciona, pois abre uma porta para a ingestão síncrona de informações de jogadores, com os seus devidos consensos, e pode poupar muito trabalho manual ou processamento de grandes volumes de dados.

No tema de Internet das Coisas (IoT), a mensageria em síncrona é de notável relevância. O artigo (D'SILVA et al., 2017) utiliza o Kafka, juntamente ao Spark para tratar o processamento de eventos de IoT. A relação do presente estudo demonstra outras ferramentas possíveis para mensageria síncrona, com o fim de comparar com o Kafka.

O artigo de (SOUSA et al., 2021) introduz ferramentas para conduzir o tratamento de informações de forma síncrona, apresentando softwares como o Apache Flume, Apache Spooq, Apache Kafka e Spark. Ele propõe uma solução híbrida de ferramentas para ter resultados ideais, se diferenciando do trabalho presente, que avalia cada ferramenta individualmente.

4 FERRAMENTAS EXISTENTES

Nesta seção, esta pesquisa irá descrever de forma introdutória o funcionamento de algumas ferramentas escolhidas a serem comparadas com nosso Carteiro (proposta de implementação), detalhando as principais escolhas e arquiteturas de software.

Foram escolhidas quatro diferentes ferramentas de mensageria de código aberto, com o intuito de comparar desempenho e resultados com a proposta de implementação da pesquisa. São ferramentas que foram escolhidas por serem amplamente utilizadas por organizações, e por serem de zero custo financeiro para implementação: **RabbitMQ** (VMWARE, 2007), **Apache ActiveMQ Artemis** (FOUNDATION, 2015) , **Apache Kafka** (FOUNDATION, 2011b) e **Apache Pulsar** (FOUNDATION, 2016).

4.1 APACHE ACTIVEMQ ARTEMIS

Desenvolvido pela Apache Software Foundation (FOUNDATION, 1999), ActiveMQ Artemis (FOUNDATION, 2015) é um projeto de código aberto com o objetivo de permitir que sejam construídos sistemas assíncronos de mensagens, sendo adaptado para ser multi-protocolar (aceitar diferentes protocolos de mensageria), de alto desempenho, passível de ser clusterizado e facilmente incorporável.

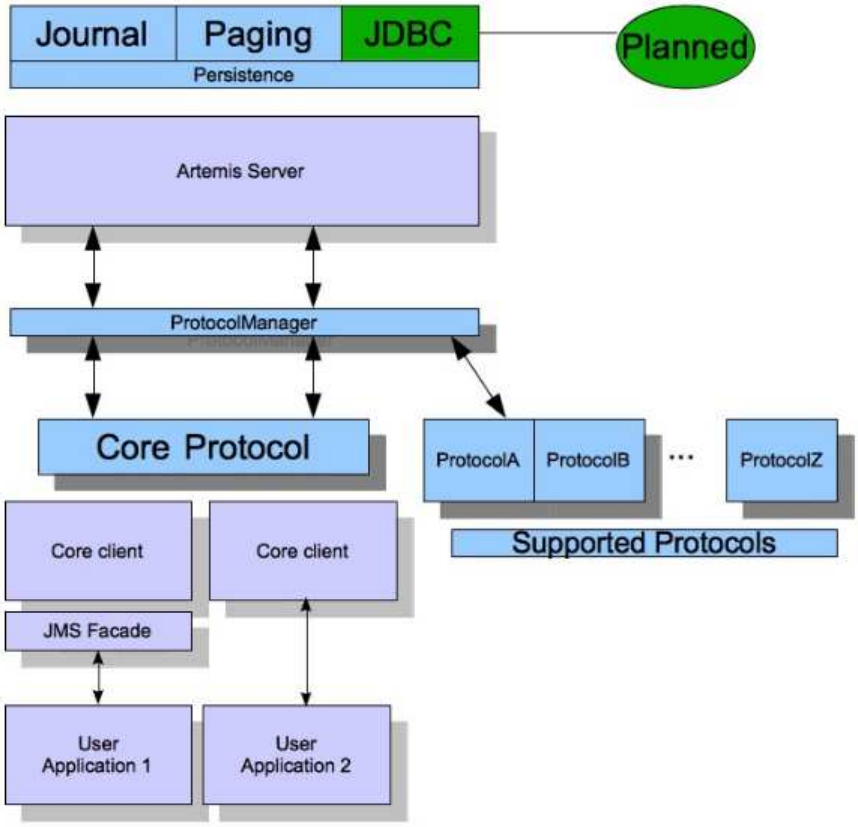
O núcleo do software foi projetado como um conjunto de Plain Old Java Objects (POJOs) (IBM, 2000). Cada servidor Apache ActiveMQ Artemis possui seu próprio diário persistente que propõe alto desempenho, o qual ele usa para mensagens.

A interação com o broker é feita de maneira que os clientes, potencialmente em diferentes máquinas físicas, possam se conectar via API com o Artemis, tendo três opções de API.

- Core client API, é uma API Java desenvolvida para ser intuitiva, alinhada internamente com o núcleo do broker. Oferece um conjunto inteiro de funcionalidades, sem complexidades encontradas no *Java Messaging Service* (JMS) (ORACLE, 2001).
- JMS 2.0 client API, é a API padrão do JMS para os clientes.
- Jakarta Messaging 3.0 client API. Essa alternativa é essencialmente parecida com a JMS2.0 API, a única diferença é na nomenclatura dos pacotes em que é usado o nome *jakarta*, ao invés de *javax*.

Os protocolos suportados são: AMQP, OpenWire, MQTT, STOMP, HornetQ e os nativos do Artemis.

Figura 4 – Arquitetura de alto nível do Artemis



A arquitetura do núcleo do Artemis é bem simples e resumida. Na figura 4 (FOUNDATION, 2021), é possível enxergar as camadas e imaginar o fluxo da mensageria. Na parte de baixo da imagem, está presente a camada de interação do cliente com a API; no meio, está o gerenciador de protocolos. A requisição sai do nível mais inferior e passa pelo gerenciador para, então, chegar ao servidor e, se necessário, passar posteriormente para a camada de persistência.

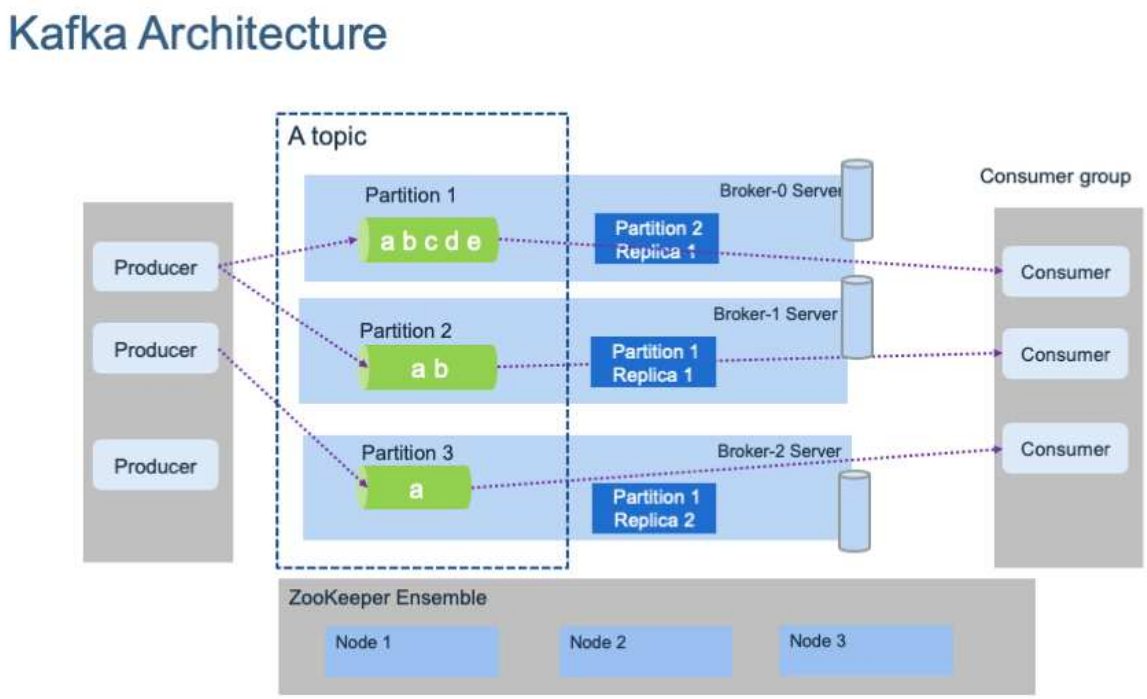
4.2 APACHE KAFKA

O Kafka (FOUNDATION, 2011b), projeto desenvolvido também pela Apache Software Foundation (FOUNDATION, 1999), é um sistema distribuído que consiste em servidores e clientes que se comunicam através de um protocolo de rede TCP de alto desempenho. Ele pode ser implantado em máquinas virtuais e contêineres, em ambientes locais e em nuvem.

Esse sistema é executado como um cluster de um ou mais servidores que podem abranger vários datacenters ou regiões de nuvem. Alguns desses servidores formam a camada de armazenamento, chamada de corretores (brokers). Para permitir a implementação de casos de uso de nível crítico, um cluster Kafka é altamente escalável e tolerante a falhas: se

algum de seus servidores falhar, os outros servidores assumirão seu trabalho para garantir operações contínuas sem qualquer perda de dados, ou seja, o Kafka garante a persistência dos dados e não é possível simplesmente desabilitar essa funcionalidade. Para ainda melhor garantia de persistência, um cluster do Kafka utiliza o ZooKeeper, que é um software também da Apache para gerenciamento de disponibilidade de informações.

Figura 5 – Arquitetura de alto nível do Kafka



A figura 5, extraída da leitura (AKIN, 2023), ilustra a arquitetura de alto nível do Kafka. A implementação das entidades está disponível em diversas linguagens, os clientes estão disponíveis para Java e Scala, incluindo a biblioteca Kafka Streams de nível superior, para Go, Python, C/CPP e muitos outros programas e linguagens, bem como APIs REST para interagir com os brokers.

4.2.1 ZooKeeper

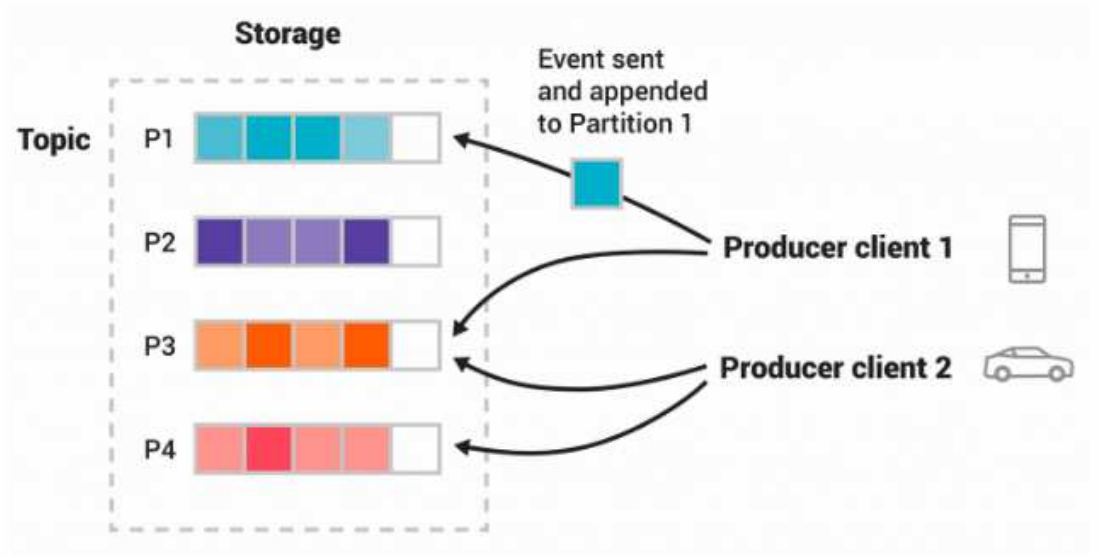
No cluster de Kafka, o ZooKeeper (FOUNDATION, 2008) é usado para persistir os estados dos componentes e da plataforma, e é executado também em cluster para garantir alta disponibilidade. Um servidor ZooKeeper é o líder e outros são usados no backup. Com o ZooKeeper, os brokers do Kafka não precisam guardar estados de produtores e consumidores, toda coordenação fica com o ZooKeeper. Outra vantagem é que o controle de acesso também fica do lado desse gerenciador.

4.2.2 Tópicos e partições

Cada tópico no Kafka sempre suporta **n** produtores e **n** consumidores. A maior diferença entre o tópico do Kafka e outras ferramentas é que as mensagens não são deletadas após o consumo. Existem políticas de retenção para quanto tempo os eventos devem ficar armazenados nos tópicos para posteriormente serem descartados.

Os tópicos são particionados, o que significa que um tópico está espalhado por diretórios localizados em diferentes brokers de Kafka. Esse posicionamento distribuído de seus dados é muito importante para a escalabilidade, pois permite que clientes leiam e gravem os dados de/para vários corretores ao mesmo tempo. Quando um novo evento é publicado em um tópico, ele é anexado a uma das partições do tópico. Eventos com a mesma chave de evento (por exemplo, um ID de cliente ou veículo) são gravados na mesma partição, e Kafka garante que qualquer consumidor de uma determinada partição de tópico sempre lerá os eventos dessa partição exatamente na mesma ordem em que foram gravados.

Figura 6 – Exemplo de tópico particionado



Na figura 6, extraída da documentação do Kafka (FOUNDATION, 2020), dois produtores diferentes estão enviando, independentemente um do outro, novos eventos para o tópico, gravando mensagens nas partições do tópico. Eventos com a mesma chave (indicados pela cor na figura) são gravados na mesma partição. Observe que ambos os produtores podem gravar na mesma partição.

4.3 APACHE PULSAR

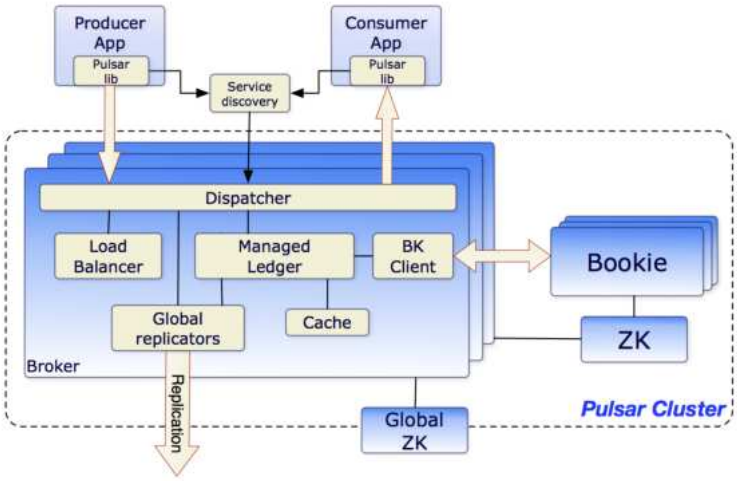
O Apache Pulsar (FOUNDATION, 2016) é uma solução que propõe alto desempenho para mensageria entre diferentes servidores. Foi originalmente desenvolvido pelo Yahoo,

e posteriormente ficou sob administração da Apache Software Foundation. Foi desenvolvido com Java, se baseando no protocolo TCP. Ele oferece georeplicação de mensagens, compactação de tópicos e suporta comunicação por parte do cliente com APIs disponíveis para Java, C++, Go, Python, Node.js e C#.

O pulsar foi construído no padrão de publicação/subscrição, com uso de tópicos e reconhecimento de mensagens por parte do consumidor, para sinalizar quando consumiu uma mensagem com sucesso. Por padrão, quando uma mensagem chega a algum tópico, e existe um ou mais consumidores subscritos a ele, a mensagem é retida até que todos os consumidores enviem o "acknowledgment" de que consumiram com sucesso. Assim, a ferramenta consegue saber se pode deletar o evento do sistema.

A arquitetura geral, em um alto nível, é composta por um ou mais clusters de Pulsar, instâncias ou clusters de BookKeeper (FOUNDATION, 2011a) e ZooKeeper (FOUNDATION, 2008). Cada cluster consegue permitir replicação de dados entre os nós integrantes e tem os seguintes componentes ilustrados na figura 7 (FOUNDATION, 2024):

Figura 7 – Arquitetura geral do Pulsar



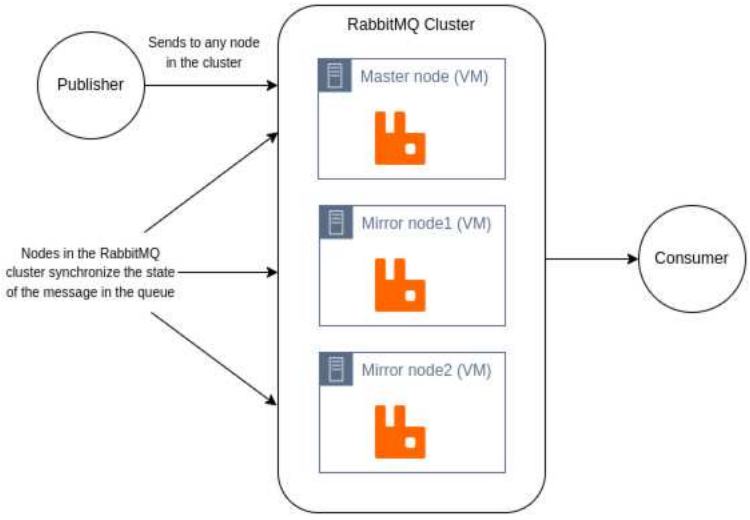
- **Brokers**, um ou mais brokers gerenciam balanceamento de carga, se comunicam com os BookKeepers e depende de um cluster ZooKeeper para coordenação de clientes.
- **Cluster de BookKeeper**, para lidar com armazenamento persistente de mensagens.
- **Cluster de ZooKeeper**, específico para lidar com tarefas de coordenação entre clusters Pulsar.

4.4 RABBIT MQ

O RabbitMQ (VMWARE, 2007) é uma implementação de código aberto do protocolo AMQP (LEE, 2024) desenvolvido com Erlang (uma linguagem de programação multiuso para desenvolvimento de sistemas simultâneos e distribuídos) e também possui uma comunidade muito ativa. Erlang oferece suporte inerente à computação distribuída (sincronizando os cookies nos nós do cluster Erlang), portanto, o RabbitMQ não depende de um gerenciador de cluster de terceiros como o ZooKeeper. O cluster RabbitMQ tem dois modos: modo de cluster normal e modo de cluster espelhado.

Esse MOM alcança alta disponibilidade e busca garantir a persistência por meio do espelhamento de filas. O conceito de fila no RabbitMQ é semelhante à partição do Kafka. Cada fila espelhada consiste em um mestre e um ou mais espelhos, onde cada Broker contém todos os dados de uma fila. O mestre está hospedado em um nó comumente chamado de nó mestre. Cada fila possui seu próprio nó mestre. Todas as operações para uma determinada fila são aplicadas primeiro no nó mestre da fila e depois propagadas para espelhos. Quando ocorre uma falha, os consumidores podem ser encaminhados para processar os dados da fila espelhada em outros Brokers. O Exchange pode ser usado como agente de encaminhamento, o que ajuda a implementar a conversão de rotas e encaminhar mensagens para a fila correspondente.

Figura 8 – Arquitetura geral do RabbitMQ



A figura 8, extraída da leitura (UMMUSALMA, 2023), ilustra o modo de cluster com espelhamento. Para identificar o modo normal, é possível desconsiderar os nós de espelho na imagem. É importante a observação de que o modo de espelhamento prejudica a escalabilidade de uma mensageria com grande demanda. Pois seria necessária a replicação completa das mensagens em cada nó espelho.

A ferramenta é um MOM de uso geral que oferece suporte a vários protocolos padronizados e a múltiplas funcionalidades, como enfileiramento de prioridade e enfileiramento

de atraso. Porém, perde um pouco em rapidez devido ao mecanismo transacional. Foi escolhida para o estudo, pois pode ter a função de persistência desabilitada, diminuindo um pouco do overhead e também possibilita filtragem de conteúdo.

4.5 OPENMESSAGING BENCHMARK - FERRAMENTA DE TESTES

Para padronizar os testes, ao invés de implementarmos a simulação de mensageria para cada ferramenta utilizando seus próprios métodos, escolhemos usar uma ferramenta já existente de benchmark: o *OpenMessaging Benchmark* é um software de código aberto, sustentado pela TheLinuxFoundation, projetado especificamente para avaliar o desempenho e a escalabilidade de sistemas de mensagens, principalmente em ambientes de nuvem. Essa ferramenta oferece um conjunto abrangente de testes e métricas para avaliar diversos aspectos de um sistema de mensagens, como *throughput*, latência, durabilidade e escalabilidade. Os seguintes componentes apresentam as principais vantagens de utilizar a ferramenta:

- **Padronização e Comparabilidade**: A ferramenta oferece um conjunto padronizado de testes e métricas, o que permite comparar diferentes sistemas de mensagens de forma justa e consistente.
- **Flexibilidade**: Permite configurar cenários de teste personalizados, ajustando parâmetros como carga de trabalho, tamanho das mensagens e topologia da rede para simular diferentes condições reais.
- **Comunidade Ativa**: Conta com uma comunidade ativa de desenvolvedores e usuários, o que garante um alto nível de maturidade e confiabilidade.
- **Código Aberto**: Sendo de código aberto, permite a customização e a verificação do código fonte, garantindo transparência e confiabilidade.
- **Foco em Mensageria**: É especificamente projetada para avaliar sistemas de mensagens, oferecendo métricas e testes relevantes para este tipo de sistema.
- **Escalabilidade**: Ao ser projetado para rodar em infraestruturas de nuvem, permite realizar testes em ambientes escaláveis e flexíveis. Isso facilita a simulação de cargas de trabalho variadas e a avaliação do desempenho de sistemas de mensagens em diferentes cenários, desde pequenas aplicações até grandes plataformas distribuídas. A nuvem oferece recursos computacionais sob demanda, permitindo ajustar a capacidade de teste de acordo com as necessidades específicas.

Devido às vantagens listadas, o *OpenMessaging Benchmark* (OPENMESSAGING, 2018) é uma ferramenta poderosa para avaliar o desempenho de sistemas de mensagens

distribuídas. O software consegue fazer a comunicação com os serviços de MOM, criando recursos dentro das ferramentas, essas que podem estar hospedadas em diferentes máquinas. Ele realiza isso através de dois componentes principais: driver e worker. Cada um desses componentes desempenha um papel crucial no processo de benchmarking.

4.5.1 Driver

O driver é o orquestrador da operação. Ele é responsável por configurar todos os parâmetros dos testes, como a quantidade de tópicos, o tamanho das mensagens e a frequência com que as mensagens são enviadas e recebidas. Além disso, o driver gerencia os workers, distribuindo as tarefas entre eles e coletando os resultados de cada tarefa. Após a conclusão dos testes, o driver compila todos os dados coletados e gera relatórios detalhados, permitindo uma análise completa do desempenho do sistema de mensagens.

4.5.2 Worker

O worker é a força de trabalho. Cada worker executa tarefas individuais, como enviar ou receber mensagens. Ele se conecta ao sistema de mensagens que está sendo testado e realiza as operações definidas pelo driver. Ao final de cada tarefa, o worker envia os resultados de seu desempenho, como o tempo que levou para processar cada mensagem, de volta para o driver.

Outro componente importante para a realização da coleta de estatísticas é o workload.

4.5.3 Workload

É um formato de arquivo em **.yaml** que tem como objetivo definir o cenário de teste a ser simulado; ele define todas as variáveis existentes para o benchmark de maneira declarativa. Definindo diferentes workloads, é possível avaliar a capacidade do sistema de lidar com uma ampla variedade de situações, o que é fundamental para a comparação das ferramentas em diferentes cenários.

Variáveis de benchmark a serem definidas pelo workload:

- **Número de tópicos**: Quantos canais de comunicação serão utilizados.
- **Número de partições**: Quantidade de partições por tópico (para ferramentas de mensageria que suportam partições).
- **Tamanho das mensagens**: O volume de dados contido em cada mensagem, podendo variar de tamanho em bytes.
- **Taxa de produção**: O *throughput* com que as mensagens são geradas, é possível pensar em quantidade de mensagens por segundo.

- **Número de produtores por tópico**: Quantidade de produtores de mensagens por canal de comunicação.
- **Número de consumidores por tópico**: Quantidade de consumidores de mensagens por canal de comunicação.
- **Duração do teste**: O tempo total que o teste irá durar.

Figura 9 – Exemplos de cenários de testes

Workload	Topics	Partitions per topic	Message size	Subscriptions per topic	Producers per topic	Producer rate (per second)	Consumer backlog size (GB)	Test duration (minutes)
`simple-workload.yaml`	1	10	1 kB	1	1	10000	0	5
`1-topic-1-partition-1kb.yaml`	1	1	1 kB	1	1	50000	0	15
`1-topic-1-partition-100b.yaml`	1	1	100 bytes	1	1	50000	0	15
`1-topic-16-partitions-1kb.yaml`	1	16	1 kB	1	1	50000	0	15

A figura 9, coletada na documentação do projeto OMB (FOUNDATION, 2022), mostra diferentes workloads com variáveis configuradas de maneiras distintas, com cada linha representando um cenário diferente de teste.

Além de todas essas facilidades, como resolução de comunicação e integração com as ferramentas de mensageria, o software do *OpenMessaging Benchmark* tem uma natureza dockerizável, o que significou uma possibilidade de subir apenas um docker (DOCKER, 2013) configurado, isolado, para testar todas as ferramentas de código aberto, de maneira totalmente desacoplada.

5 CARTEIRO - PROPOSTA DE IMPLEMENTAÇÃO

O que chamamos de Carteiro (ARAUJO, 2025a) é uma ferramenta de desenvolvimento própria que se caracteriza como um corretor de mensagens (Message Broker). Sua principal funcionalidade é permitir que clientes realizem subscrições, podendo definir regras personalizadas de filtragem e/ou amostragem de mensagens. Diferentemente das demais ferramentas analisadas neste trabalho, que possuem aplicações mais amplas e diversos casos de uso, o Carteiro foi projetado com um foco específico: transmissão de mensagens de maneira síncrona. Esse enfoque permite otimizar seu desempenho e funcionalidade para cenários onde a persistência e a resiliência dos dados não são requisitos prioritários, priorizando, assim, a eficiência na entrega das mensagens.

5.1 ESCOLHAS DE DESENVOLVIMENTO

Nessa seção, iremos abordar as escolhas realizadas para desenvolvimento de uma solução própria, justificando as decisões tomadas e apresentando como o Carteiro foi estruturado.

5.1.1 Linguagem de Programação e Framework

Optamos por utilizar Java como linguagem de programação principal, principalmente, devido à experiência prévia com a linguagem. Além disso, Java é amplamente utilizado na implementação de ferramentas de mercado como Kafka, ActiveMQ e Pulsar, o que reforça sua adequação ao cenário de mensageria. Para simplificar a criação de APIs RESTful, adotamos o framework Spring Boot (TANZU, 2022), que foi empregado na implementação do módulo de subscrição. O Spring Boot também facilitou o processo de implantação, pois inclui um servidor web incorporado (*embedded web server*), permitindo uma execução simplificada, especialmente em ambientes na nuvem.

5.1.2 Protocolo de Mensagens

Dado o foco em mensageria síncrona, escolhemos o protocolo UDP para envio e recebimento de mensagens, visando menor latência e alta vazão de mensagens. O conteúdo das mensagens é codificado utilizando Protocol Buffers (GOOGLE, 2001), um protocolo eficiente para *parsing* e que oferece a possibilidade de *parsing* condicional. Isso permite ignorar o conteúdo completo das mensagens quando não é necessário, processando apenas metadados ou cabeçalhos, resultando em ganhos significativos de desempenho.

As mensagens da aplicação seguem um formato padronizado definido em Protocol Buffers, sendo estruturadas a partir de uma mensagem base. Essa mensagem contém um

cabeçalho e o conteúdo da mensagem propriamente dito. O cabeçalho é composto pelos seguintes campos:

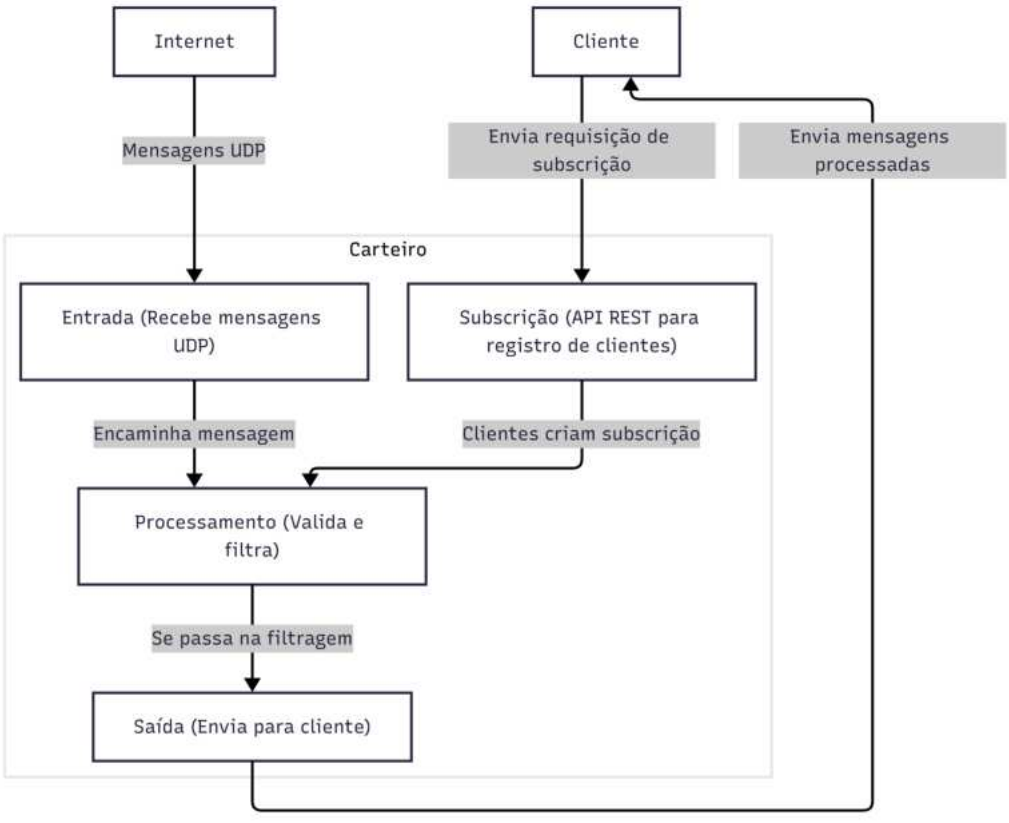
- **Tipo**: uma *string* que define a categoria da mensagem;
- **Timestamp**: um valor que indica o instante de criação da mensagem;
- **Identificador**: um identificador alfanumérico único da mensagem;
- **Data**: uma cadeia de bytes que contém o conteúdo da mensagem, que por sua vez também é codificado em *Protocol Buffers*.

Essa estrutura possibilita a interpretação eficiente das mensagens, permitindo que o *parsing* condicional utilize apenas o cabeçalho quando necessário e que o conteúdo seja decodificado sob demanda. Dessa forma, o desempenho da aplicação é otimizado, reduzindo o tempo de processamento e o consumo de recursos computacionais.

5.1.3 Divisão em Módulos

A aplicação foi estruturada em quatro módulos principais para organizar as funcionalidades e otimizar o processamento, tal como ilustrado na figura 10.

Figura 10 – Arquitetura geral do Carteiro



5.1.4 Módulo de Entrada

Implementado em Java puro, utiliza a biblioteca `DatagramPacket`, nativa do Java, que possibilita o envio e recebimento de mensagens UDP. Nesse módulo, ela foi usada exclusivamente para receber mensagens dos produtores por meio de uma única porta exposta. O papel do módulo é receber mensagens e imediatamente as encaminhar ao módulo de processamento.

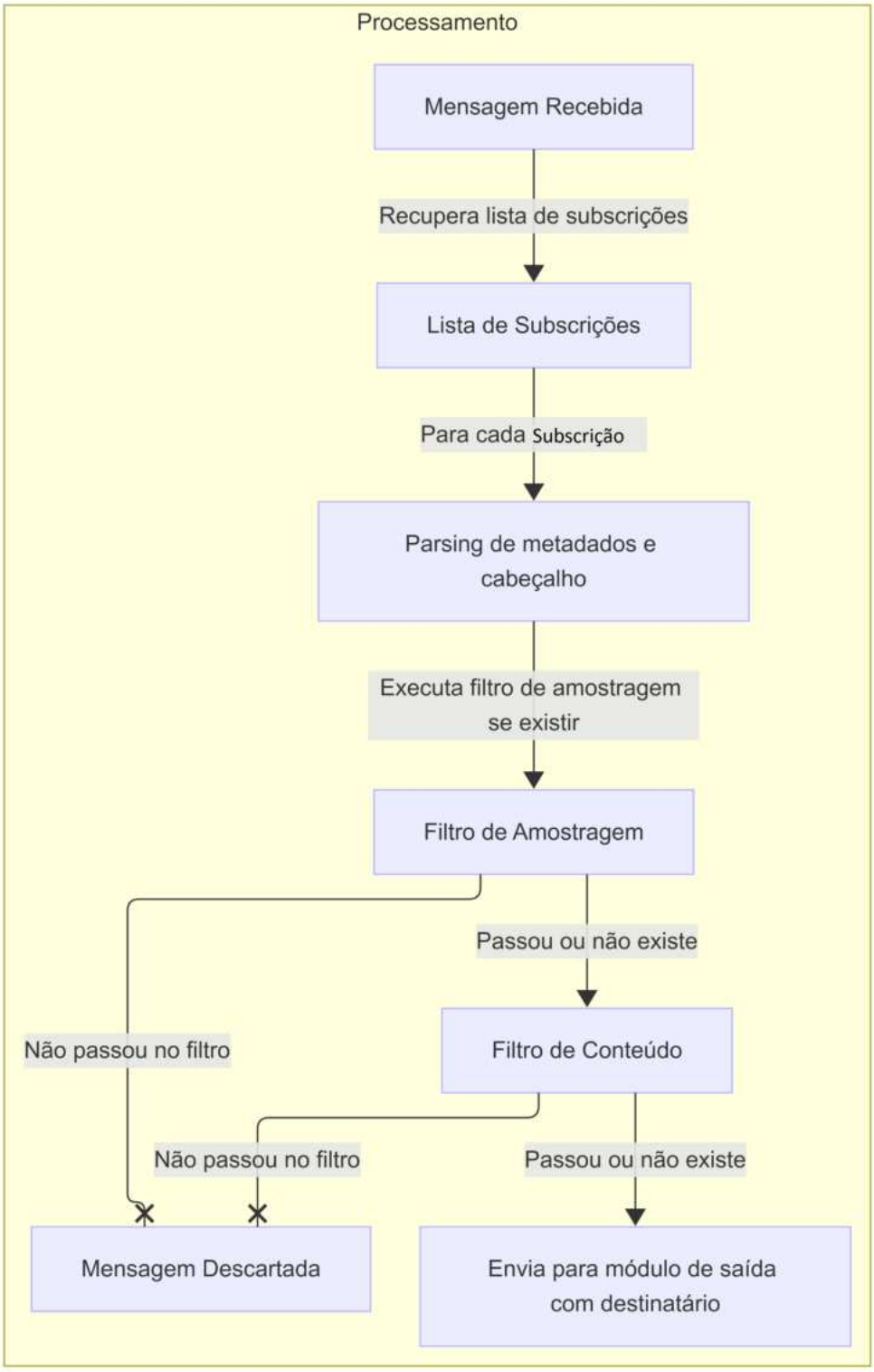
5.1.5 Módulo de Processamento

Responsável pela validação e aplicação de filtros de mensagens. Cada mensagem é processada para todas as subscrições existentes, seguindo o fluxo:

1. Realiza o *parsing* de metadados e cabeçalhos.
2. Executa os filtros de amostragem. Caso a mensagem não passe, avança para a próxima subscrição.
3. Caso passe, verifica os filtros de conteúdo. Apenas então o conteúdo da mensagem é *parseado* e os filtros são aplicados.
4. Se a mensagem atender a todos os critérios, é encaminhada ao módulo de envio.

Esse processo, ilustrado na imagem 11, ocorre de forma assíncrona, permitindo o processamento paralelo das mensagens. Para garantir segurança em ambientes com múltiplas *threads*, para armazenamento de dados acessados por múltiplas *threads*, utilizamos estruturas de dados como `ConcurrentHashMap`, nativa do Java e que permite gravar e recuperar dados de forma concorrente. Além disso, também utilizamos um gerador de números aleatórios, também oferecido nativamente pelo Java, compatível com processamento concorrente.

Figura 11 – Fluxo de Processamento



5.1.6 Módulo de Envio

Recebe o conteúdo da mensagem, o endereço IP e a porta do cliente destinatário e, tal qual o módulo de entrada, utiliza `DatagramPacket` para envio de mensagens. Este módulo também foi implementado em Java puro.

5.1.7 Módulo de Subscrição

Implementado como uma API RESTful usando Spring Boot. Permite que os clientes:

- Registrem subscrições enviando um arquivo JSON com os filtros desejados.
- Cancelem subscrições usando o identificador gerado ao criar a subscrição.

Cada subscrição segue uma estrutura pré-definida, na qual o usuário deve informar os seguintes parâmetros:

- **IP de destino**: endereço para onde as mensagens filtradas serão enviadas.
- **Porta de destino**: porta utilizada para o recebimento das mensagens.
- **Especificações de filtragem**: critérios que determinam quais mensagens serão entregues.
- **Tipo da mensagem**: define a categoria de mensagens às quais a subscrição está vinculada.

Dessa forma, cada subscrição pode receber apenas mensagens de um único tipo.

5.2 FILTRAGEM DE MENSAGENS

O Carteiro oferece a função de filtragem de mensagens, ou seja, permite que usuários definam parâmetros que irão definir quais mensagens serão enviadas para os consumidores finais. Nessa seção iremos detalhar como isso é realizado.

5.2.1 Filtro de Conteúdo

Utilizamos a biblioteca *Spring Expression Language* (SpEL) (VMWARE, 2022), que possibilita consultas dinâmicas e operações sobre objetos em tempo de execução. Para o *Carteiro*, as expressões SpEL são usadas exclusivamente para retornos booleanos, indicando se uma mensagem atende aos critérios de filtro. Além disso, as expressões são compiladas no momento da subscrição em objetos do tipo `Expression`, otimizando o desempenho no momento da execução.

5.2.2 Filtros de Amostragem

O *Carteiro* suporta dois tipos de filtros de amostragem:

1. **Frequência Absoluta:** Define o intervalo mínimo de tempo entre envios consecutivos de mensagens, utilizando a data e hora do sistema operacional para garantir o cumprimento.
2. **Frequência Relativa:** Implementada de duas formas:
   a) Cada mensagem tem uma probabilidade f (entre 0 e 1) de ser enviada.
   b) Envio de uma mensagem a cada p mensagens recebidas.

Os filtros de frequência são mutuamente exclusivos, ou seja, uma subscrição pode conter apenas uma das opções de filtragem. Ambos os tipos de amostragem podem ser combinados com filtros de conteúdo.

5.3 FERRAMENTAS DE BENCHMARK

Para realizar os testes de desempenho no *Carteiro*, optamos por desenvolver uma solução própria composta por duas aplicações, denominadas *Produtor* (ARAUJO, 2025c) e *Consumidor* (ARAUJO, 2025b). A decisão de não implementar um *plugin* para o *OpenMessaging Benchmark*, apesar de sua possibilidade, foi motivada pela complexidade e pelo desvio de escopo que tal abordagem traria ao projeto.

A solução desenvolvida foi projetada para simular cenários de uso real, onde uma aplicação envia mensagens para um *broker* e outra as consome. Implementamos essas ferramentas de forma separada, permitindo a execução independente e replicação em múltiplas instâncias de produtores e consumidores, com quantidades distintas. Essa abordagem nos proporcionou maior flexibilidade na realização de testes e maior aderência a cenários reais. Vale acrescentar que, embora não tenha sido desenvolvido o *plugin* para o OMB, essa ferramenta foi utilizada como referência para desenvolvimento das aplicações dos testes de desempenho.

5.3.1 Decisões de Implementação

Ambas as ferramentas foram desenvolvidas em Java, não apenas pela experiência prévia da equipe com a linguagem e pela performance oferecida, mas também para manter o padrão de desenvolvimento já utilizado no *Carteiro*. Além disso, utilizamos o framework Spring Boot (TANZU, 2022), que já havia sido empregado no desenvolvimento do *Carteiro*, facilitando a criação de APIs RESTful e o *deployment* na nuvem, graças ao servidor web incorporado.

5.3.2 Arquitetura do Produtor

A aplicação *Produtor* (ARAUJO, 2025c) foi projetada com dois módulos principais, sendo um para cumprir funcionalidades necessárias para iniciar uma simulação e coletar contagem de mensagens enviadas e, outra para enviar mensagens para um broker de Carteiro.

5.3.2.1 Módulo de Configuração de Simulação

Este módulo é responsável por receber os parâmetros de configuração da simulação, fornecidos através de uma API RESTful. Suas funcionalidades incluem:

- **Iniciar Simulação:** Recebe um objeto JSON contendo as especificações da simulação, como:
  - Taxa de envio de mensagens.
  - Duração da simulação.
  - Endereço IP e porta de destino.
  - Duração da fase de aquecimento (*warmup*).
  - *Throughput* máximo desejado.
- **Consultar Contagem de Envios:** Retorna a quantidade de mensagens enviadas na última simulação.

5.3.2.2 Módulo de Envio

Após a configuração da simulação, este módulo executa o envio das mensagens conforme os parâmetros definidos. O fluxo inclui:

1. **Fase de *warmup*:** Envio de mensagens marcadas com a *flag* correspondente, que são desconsideradas nas métricas finais.
2. **Envio regular:** Contabiliza a quantidade de mensagens enviadas durante a simulação.

As mensagens enviadas são geradas de forma aleatória durante a inicialização da aplicação, utilizando uma lista pré-processada para garantir maior eficiência no envio. O tamanho das mensagens pode ser configurado através de uma variável de ambiente, que define o tamanho das cadeias de caracteres geradas. Isso permite ajustar o tamanho das mensagens em bytes de forma prática e precisa, um ponto crucial para os testes de desempenho conduzidos.

5.3.3 Arquitetura do Consumidor

A aplicação *Consumidor* (ARAUJO, 2025b) foi estruturada em três módulos principais, sendo um para cumprir funcionalidades de uma simulação, outro para recebimento de mensagens, e outro para processar métricas referentes à mensagens obtidas durante uma janela de tempo.

5.3.3.1 Módulo de Configuração de Simulação

Semelhante ao módulo de configuração do *Produtor*, este módulo possui endereços de API RESTful para:

- **Iniciar Simulação:** Recebe parâmetros, como o endereço do *Carteiro* e as definições de subscrição, registra a subscrição no *Carteiro* e inicia o recebimento de mensagens.
- **Consultar Resultados:** Processa as estatísticas da simulação e retorna as métricas calculadas, limpando os dados registrados para permitir novas simulações.

5.3.3.2 Módulo de Recebimento

Este módulo consiste em um servidor UDP que recebe mensagens em uma única porta, registra informações relevantes e descarta mensagens fora do escopo. O fluxo é o seguinte:

1. Recebe uma mensagem.
2. Verifica a presença da *flag* de *warmup* e descarta mensagens marcadas.
3. Registra informações de latência, instante de chegada e outras estatísticas de forma assíncrona, otimizando o desempenho.

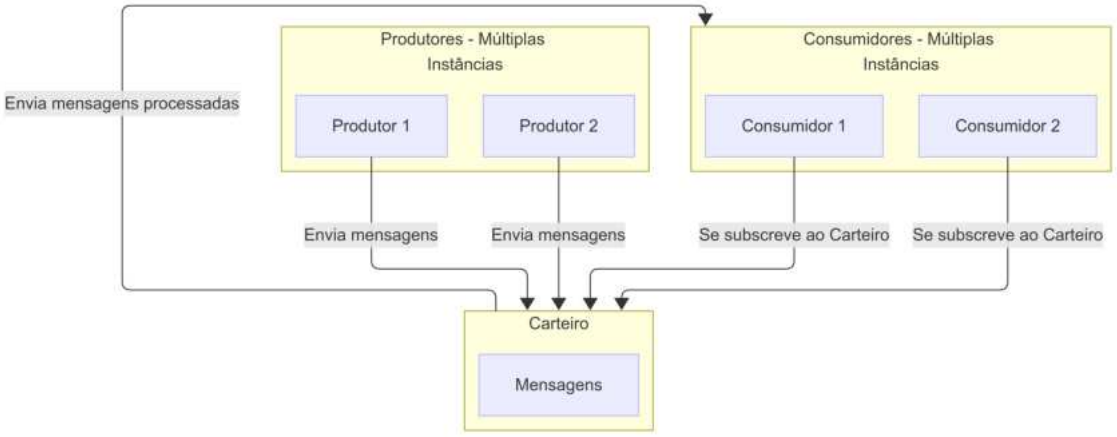
5.3.3.3 Módulo de Processamento

Quando acionado, este módulo calcula métricas como:

- *Throughput* médio.
- Latência média.
- Percentis de latência.
- Total de mensagens recebidas.

Os resultados são formatados em um arquivo JSON e retornados para o usuário.

Figura 12 – Visualização do benchmark



A figura 12 é uma ilustração, feita com *draw.io* (LTD, 2011), da arquitetura de testes do Carteiro apresentada na presente seção, em que uma ou múltiplas instâncias de produtores enviam mensagens de acordo com uma especificação arbitrária de simulação para o Carteiro, que em seguida processa e envia mensagens para os consumidores inscritos naquela devida simulação.

6 METODOLOGIA E COLETA DE ESTATÍSTICAS

Neste capítulo, será descrito todo o processo projetado para comparação das ferramentas de código aberto com o Carteiro. Serão detalhadas as escolhas e decisões tomadas para rodar as simulações, tanto quanto as escolhas do ambiente para diminuir o viés e chegar à fase de coleta de estatísticas de maneira consistente.

6.1 VIABILIDADE DE FERRAMENTAS

Antes de implementar cada ferramenta em um ambiente padronizado para simulações, foi necessário testar de maneira local a implantação e o funcionamento de cada uma. Para isso, foi usado o **Docker** (DOCKER, 2013), que ofereceu uma série de vantagens que contribuem para a reprodutibilidade, eficiência e flexibilidade dos experimentos. Ao isolar os ambientes, automatizar a configuração e facilitar a gestão de recursos, foi possível abstrair bastante do processo de implantação manual das ferramentas, melhorando o foco na coleta de resultados.

Para cada uma das quatro ferramentas a serem comparadas (RabbitMQ , Artemis, Pulsar e Kafka), foram criados e configurados contêineres a serem utilizados com o fim de rodar os testes de desempenho. As versões das ferramentas são disponibilizadas com imagens de contêiner já prontas no **DockerHub** (DOCKER, 2014), ou disponibilizam códigos para subida de clusters configuráveis no GitHub, permitindo alteração nos arquivos de configuração do Docker (**Dockerfile** e **docker-compose.yaml**).

6.2 PADRONIZAÇÃO DOS TESTES

A seguir, detalharemos as escolhas técnicas realizadas para a condução dos benchmarks, com foco nas ferramentas e na infraestrutura de nuvem empregadas. Essas decisões foram cruciais para garantir a consistência e a reprodutibilidade dos resultados, permitindo comparações precisas entre as diferentes ferramentas MOM. Ao padronizar o ambiente de teste, eliminamos variáveis externas que poderiam influenciar os resultados e comprometer a confiabilidade da análise.

6.3 ESCOLHA DE INFRAESTRUTURA EM NUVEM GCP

A escolha da **Google Cloud Platform** (GCP) (CLOUD, 2008) como infraestrutura para a execução dos testes do presente trabalho foi fundamentada em diversos fatores que a tornam uma plataforma consistente para este tipo de aplicação.

Um dos principais motivadores foi a disponibilidade do crédito de 300 dólares para testar a plataforma no tier gratuito. Essa iniciativa facilitou significativamente a tomada

de decisão, permitindo que fossem experimentados os serviços da GCP de forma prática e sem custos iniciais.

Além disso, a GCP oferece uma imagem de máquina virtual otimizada para contêineres, o que se mostrou fundamental para a execução eficiente do Carteiro. Essa imagem pré-configurada agiliza o processo de setup e garante um ambiente de execução consistente.

Outro fator relevante é a possibilidade de subir as próprias imagens de docker para o Artifact Registry da GCP. Essa funcionalidade permite total controle sobre o ambiente de execução, garantindo que todas as dependências e configurações estejam de acordo com os requisitos do Carteiro.

A GCP também se destaca por sua ampla gama de serviços gerenciados e escaláveis, permitindo a rápida implantação e configuração dos ambientes de teste. A flexibilidade da plataforma possibilita a criação de recursos computacionais sob demanda, adaptando-se às necessidades específicas de cada experimento.

6.3.1 Imagem de máquina otimizada para aplicações conteinerizadas

A escolha de uma imagem de máquina virtual (VM) otimizada para contêineres foi um fator crucial para garantir a consistência e eficiência dos testes realizados na Google Cloud Platform. Essa imagem vem pré-instalada com os softwares e ferramentas essenciais para o funcionamento de contêineres, como o Docker, e é otimizada para recursos como memória e CPU, proporcionando um ambiente leve e ágil. Dessa maneira, é possível melhorar a qualidade das simulações em relação às outras imagens de sistema operacional que a GCP oferece.

6.3.2 Redução do viés para testes de desempenho

A escolha do serviço de nuvem da Google e, em particular, o uso de máquinas virtuais (VMs) otimizadas para contêineres e suas robustas capacidades de rede, foi fundamental para reduzir o viés nos testes de desempenho realizados neste trabalho. Abaixo foram listados alguns pontos e escolhas que foram pensados para alicerçar a redução dos vieses de diferentes ambientes para efetivamente rodar as simulações com o menor viés possível.

- **VMs Independentes**: Cada ferramenta de mensageria foi implementada em uma VM independente, de tal maneira que a VM escolhida serviu exclusivamente para hospedar o serviço daquele MOM em específico, garantindo que as configurações de hardware e software fossem consistentes e isoladas de outros processos. Isso minimizou a interferência de outras aplicações ou serviços na coleta de dados, proporcionando resultados mais precisos e confiáveis.

- **Imagens de Máquina Padronizadas**: O uso de imagens de máquina pré-configuradas e otimizadas para contêineres assegurou que todas as VMs possuíam a mesma base

de software e configurações, eliminando a variabilidade causada por diferenças de sistema operacional ou pacotes instalados.

- **Redes Virtuais Isoladas**: As redes virtuais da GCP permitiram criar ambientes de teste isolados, com recursos de rede dedicados para cada experimento. Isso evitou interferências de tráfego externo e garantiu que os testes fossem realizados em condições controladas seguindo o mesmo padrão de rotas e tráfego de pacotes.

6.4 ARQUITETURA DA INFRAESTRUTURA DE TESTES

Após entender as necessidades e vantagens de utilizar um ambiente de nuvem para simular cenários de testes, foi projetada uma solução para implantar as aplicações de maneira com que todas as ferramentas utilizassem recursos equivalentes e o mesmo ambiente. Foi decidido que o tamanho das máquinas seguisse o padrão mínimo necessário para estressar todos os softwares ao máximo, e que esse tamanho seria definido para toda implantação de software, de tal maneira que os recursos das máquinas virtuais seriam nivelados por cima, e nenhuma ferramenta carecesse de recursos para rodar os testes.

6.4.1 Uma máquina virtual para cada ferramenta de mensageria

A decisão de alocar uma máquina virtual (VM) individual para cada ferramenta de mensageria foi estratégica e visou garantir um ambiente de testes altamente isolado, escalável e controlável. Dessa maneira, cada software de MOM fica hospedado em apenas uma VM, podendo estar clusterizado em Docker ou não, sem concorrer com outros processos e garantindo que todos os recursos ficarão focados para a ferramenta.

6.4.2 Uma máquina virtual para o OpenMessaging Benchmark

Foi decidido também isolar o orquestrador *OpenMessaging Benchmark* em uma máquina virtual separada, que também garante um ambiente isolado, livre de interferências das ferramentas de mensageria que estão sendo testadas. Isso permite um controle mais preciso sobre a execução dos testes, com carga de mensagens e variáveis padronizadas, e sobre a coleta de dados. Além de simplificar o gerenciamento e a configuração dos testes. Isso facilita a criação, a execução e o monitoramento dos experimentos. Essa máquina foi projetada para rodar todos os cenários de testes sequencialmente para cada ferramenta, e armazenar em seu disco o resultado dos processamentos.

6.4.3 Arquitetura dos testes do Carteiro com três máquinas virtuais

Foi decidido separar a arquitetura de testes do Carteiro em relação aos obtidos com o *OpenMessaging Benchmark*, porém ainda visando as mesmas métricas que o orquestrador de testes alcança, e utilizando a mesma configuração de máquina. Essa decisão foi tomada

por motivos dessa ação levar menos tempo de desenvolvimento e possuir maior facilidade para depuração, dado que desenvolver uma integração com o *OpenMessaging Benchmark* levaria um tempo considerável.

Para garantir o correto funcionamento e isolamento dos componentes da aplicação Carteiro, foram configuradas três máquinas virtuais (VMs) distintas na infraestrutura de testes, cada uma dedicada a hospedar uma das imagens Docker essenciais para o fluxo completo do sistema de mensageria: uma para o **Produtor**, uma para o **Carteiro** e outra para o **Consumidor**.

Alocar VMs dedicadas para cada um permitiu com que os processos de envio, intermediação e consumo de mensagens fossem executados sem interferências de compartilhamento de recursos, evitando gargalos que poderiam surgir se rodassem em um ambiente de VM compartilhada.

A separação permitiu diagnósticos mais precisos, facilitando a identificação de problemas de latência ou *throughput* em pontos arbitrários, sem interferência de outros componentes.

Além disso, cada VM pode ser configurada com logs específicos para cada processo (produção, entrega e consumo), permitindo um melhor acompanhamento do fluxo de mensagens, análise de desempenho e diagnóstico de falhas.

6.4.4 Formatos de implementação de ferramentas

Dado que foi decidido comparar diferentes ferramentas entre si e carteiro, a diversidade de arquiteturas e modelos de implementação de MOMs exigiu uma abordagem personalizada na configuração dos ambientes de teste. Esta seção justifica a escolha de configurações standalone ou cluster para cada ferramenta, considerando suas características e os objetivos dos testes.

6.4.4.1 Configuração Standalone

Foi optado usar o modo standalone (nó único) para o Artemis, RabbitMQ, que apesar de permitirem configuração em cluster, não exigem, estruturalmente, uma configuração distribuída para atender aos cenários de teste propostos, uma vez que a arquitetura nativa de ambos permite gerenciar altos volumes de mensagens sem a necessidade de múltiplos nós. O Carteiro também se encaixa na categoria de modo standalone, pois foi implementado como um serviço em nó único, e não há possibilidade de rodá-lo em múltiplos nós.

Além disso, as configurações em modo standalone são simplificadas, de tal maneira que não foi preciso gastar tanto tempo em desenvolvimento para deixar as ferramentas já configuradas rodando em apenas um nó.

6.4.4.2 Configuração em Cluster

Tanto o Apache Kafka quanto o Apache Pulsar são plataformas de mensageria projetadas para operar em arquiteturas distribuídas. Elas utilizam múltiplos nós para oferecer alta disponibilidade e balanceamento de carga, o que é essencial para cenários de testes que requerem alto *throughput*. Dessa maneira, para cada um deles, foram configurados três nós Docker, para simular uma infraestrutura mais similar a ambientes de produção, e para aproveitar a comunicação com o *OpenMessaging Benchmark*, que já é pré-configurada para testar essas duas ferramentas em formatos de cluster.

Nessa subseção, é importante ressaltar que durante a implementação local dessas ferramentas, nas etapas de configuração e pré-testes, foi identificado que uma máquina com apenas **16GB** de memória RAM se mostrou insuficiente para processar todos os cenários de testes, principalmente porque os MOMs clusterizados como Kafka e Pulsar careceram de recursos em cenários de alto *throughput*.

Ademais, mesmo com configurações distintas (cluster vs. standalone), esses testes capturam as capacidades inerentes de cada sistema e destacam suas forças e fraquezas em relação a latência, escalabilidade e uso de recursos. Posteriormente, nesse texto, serão apresentadas as definições dos cenários de testes, em que todas as ferramentas serão comparadas entre si, utilizando as mesmas entradas.

6.4.4.3 Configuração de persistência de mensagens

Na estrutura de testes projetada, buscamos comparar o desempenho dos MOMs (ActiveMQ Artemis, RabbitMQ, Apache Pulsar e Apache Kafka) em cenários onde a persistência de dados é desnecessária, com tolerância à perda de mensagens. Esse tipo de configuração foi necessária para alinhar os testes com o comportamento desejado do Carteiro, nosso sistema de mensagens, que não implementa persistência e tolera perda de dados para garantir menor latência e maior *throughput* em contextos de *streaming*.

- **ActiveMQ Artemis, RabbitMQ e Apache Pulsar**: A configuração para desabilitar a persistência de dados nesses MOMs foi simples e direta, bastando modificar o valor de uma variável de ambiente de true para false. Essa modificação, que dependendo da ferramenta, pode ser feita diretamente pelos comandos de cada ferramenta ou com alteração no arquivo de driver de configuração do *OpenMessaging Benchmark*. Com isso, todos os MOMs nessas configurações passaram a operar em modo "não persistente", atendendo ao requisito de tolerância à perda e otimizando o desempenho.

- **Apache Kafka**: Para o Kafka, a remoção da persistência foi bem mais complexa, pois o sistema foi projetado para preservar mensagens por padrão, oferecendo tole-

rância a falhas e retenção prolongada. Para simular um cenário sem persistência, foi necessário ajustar configurações específicas diretamente pelos comandos do Kafka, como definir período extremamente curto para variável que define tempo de retenção (**log.retention.ms**), e ajuste do tamanho máximo do log de armazenamento para limitar o acúmulo de mensagens a serem consumidas (**log.segment.bytes**).

Essas configurações garantem que cada MOM seja testado em seu desempenho mais próximo possível de uma configuração não persistente, o que permite uma comparação justa dos cenários onde a perda é tolerável e o foco está em alto *throughput* com baixa latência.

6.4.5 Padrão de Configuração de máquinas virtuais

Para garantir a consistência e a comparabilidade dos testes, o padrão de configuração de máquinas virtuais adotado neste trabalho se baseou na utilização de uma imagem de máquina da família **cos-stable** (Container-Optimized OS). A escolha desta imagem foi justificada na subseção 4.3.1.

Cada VM foi configurada no modelo **e2-standard-8**, que oferece 4 núcleos, 8 vCPUs e 32GB de memória, configuração robusta e balanceada para o processamento de grandes volumes de dados com baixos tempos de resposta. Além disso, as VMs foram configuradas para permitir roteamento para o domínio público, facilitando o gerenciamento de rotas e configurações de rede necessárias para a comunicação eficiente entre as instâncias e os serviços externos de monitoramento e benchmark.

6.5 CENÁRIOS DE TESTES E COLETA DE RESULTADOS

Nesta seção, serão descritos os diferentes cenários definidos para comparar as ferramentas de mensageria selecionadas (Kafka, Pulsar, RabbitMQ, ActiveMQ Artemis) e o Carteiro. Foram estabelecidos **seis cenários distintos** que permitem uma análise abrangente do desempenho de cada ferramenta em termos de *throughput*, latência e escalabilidade. Esses cenários foram criados para simular variações na carga de dados e nas taxas de envio de mensagens, buscando identificar as diferenças entre cada ferramenta em condições de uso variadas.

A Tabela 1, abaixo, apresenta uma visão geral das principais características das ferramentas analisadas e da solução própria. Antes de comparar seus desempenhos, é relevante considerar atributos como a linguagem de implementação, representada pelo campo **Linguagem**, o formato de arquitetura adotado, referenciado pelo campo **Arquitetura**, a configuração da persistência na ferramenta, representado pelo campo **Persistência**, que caso esteja preenchido com “Sim” , é devido a impossibilidade de removê-la por completo, e caso esteja preenchido com “Não”, significa que a ferramenta operou sem o modo persistente e, por fim, a dificuldade de implantação, representada pelo campo **Configuração**.

Essas variáveis impactam diretamente não apenas o comportamento das ferramentas em diferentes cenários de uso, mas também a viabilidade de sua adoção em sistemas que demandam baixa latência, alta disponibilidade ou facilidade de gerenciamento.

Tabela 1 – Comparação de atributos entre ferramentas e solução própria

	Linguagem	**Arquitetura**	**Persistência**	**Configuração**
Artemis	Java	Nó-único	Não	Fácil
Apache Kafka	Java e Scala	Cluster	Sim	Complexa
Apache Pulsar	Java	Cluster	Não	Complexa
RabbitMQ	Erlang	Nó-único	Não	Média
Carteiro	Java	Nó-único	Não	Média

Além disso, foram definidos cenários específicos para o Carteiro que incluem testes adicionais com filtragem por conteúdo e por frequência - tanto absoluta quanto periódica, e com variação de número de produtores e consumidores. Com esses cenários, é possível avaliar a eficiência do Carteiro ao lidar com streams de dados e sua capacidade de realizar filtragem de mensagens, sem persistência e com tolerância à perda de dados. Dessa forma, podemos avaliar tanto aspectos exclusivos do Carteiro quanto sua corretude.

6.5.1 Cenários de comparação entre todas ferramentas e Carteiro

Com o objetivo de comparar as ferramentas entre si e com o Carteiro, foram definidos testes seguindo um padrão de configuração com um produtor e um consumidor, operando sobre apenas um tópico com uma única partição, com uma duração de testes de 8 minutos após fase de warmup de 1 a 3 minutos. Esse período inicial de aquecimento assegura que as métricas coletadas reflitam com mais precisão o desempenho real das ferramentas sob teste, eliminando o impacto de inicializações ou processos de carga inicial que poderiam distorcer os resultados finais.

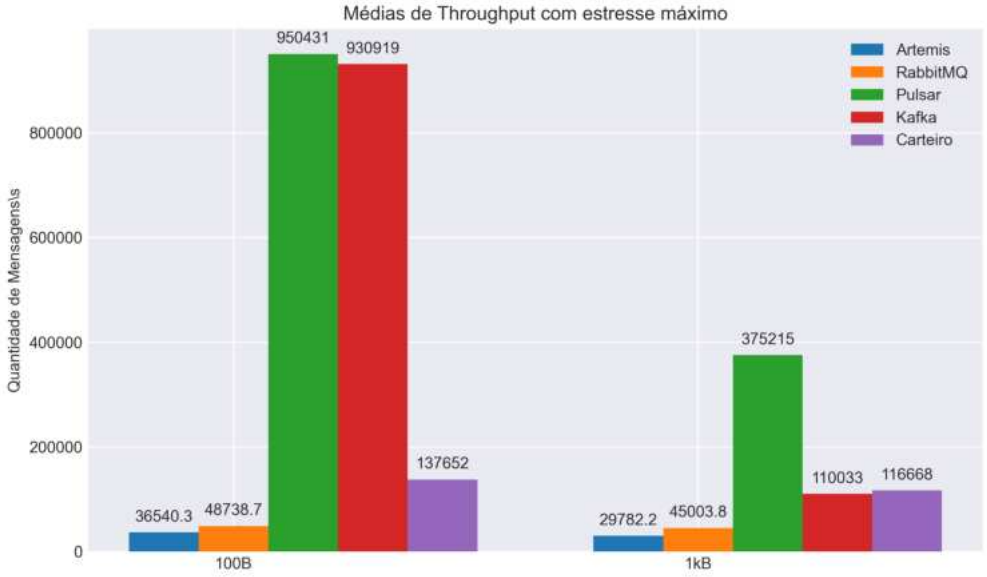
Esse padrão foi selecionado por contemplar o cenário com menor complexidade para poder comparar as ferramentas de forma mais efetiva, e por ter compatibilidade com todas as ferramentas, dado que nem todas suportam partições.

Todas as variáveis descritas abaixo foram configuradas para servirem de entrada para testes no *OpenMessaging Benchmark* e no Carteiro. No OMB, elas entram como arquivos **.yaml**, e, no Carteiro, entram como parâmetros de chamada API Rest POST, no endereço da API do produtor.

6.5.1.1 Comparação de estresse máximo entre ferramentas

No gráfico abaixo extraímos a vazão média de mensagens tolerada em cenário de *throughput* máximo para cada ferramenta, com tamanho de mensagem variando entre 100B e 1kB. É importante ressaltar que nessa análise específica, não estamos considerando latência, apenas o *throughput*.

Figura 13 – Vazão média com estresse máximo



Analisando o gráfico presente na figura 13, é possível concluir que o Carteiro apresentou um desempenho competitivo em termos de *throughput*, especialmente considerando que opera em um único nó, assim como RabbitMQ e Artemis. Ele também demonstrou uma boa tolerância a cenários de alto *throughput* dentre as ferramentas que operaram em nó único, pois se posicionou em níveis mais próximos a middlewares de maior complexidade como Kafka e Pulsar, que foram testadas em clusters com três nós, fator esse que naturalmente aumenta a capacidade de transmitir altos volumes de dados, e isso é evidenciado no gráfico.

Também é notório que todas as ferramentas experimentaram uma queda no *throughput* ao aumentar o tamanho da mensagem de 100 bytes para 1 kilobyte quando medido em mensagens por segundo, conforme esperado. No entanto, ao analisar o *throughput* em bits por segundo, observamos um ganho com mensagens maiores. Isso pode ser atribuído ao fato de que as ferramentas analisadas são projetadas para lidar com mensagens de tamanho moderado a grande, sendo que Kafka suporta mensagens de até 1 MB, Pulsar até 5 MB, RabbitMQ até 10 MB e Artemis depende da memória disponível e configuração de armazenamento. Dessa forma, o tamanho inicial de 100 B pode não ter sido o mais eficiente em termos de aproveitamento dos buffers e do processamento interno dessas ferramentas. As ferramentas de nó único apresentaram menor perda de *throughput* em mensagens por segundo com o aumento do tamanho das mensagens porque suas arquiteturas são mais simples, locais e menos impactadas por *overheads* de comunicação e armazenamento. Já as ferramentas clusterizadas sofrem mais com os custos de coordenação entre nós e dependência de I/O de disco, o que explica a maior redução de desempenho

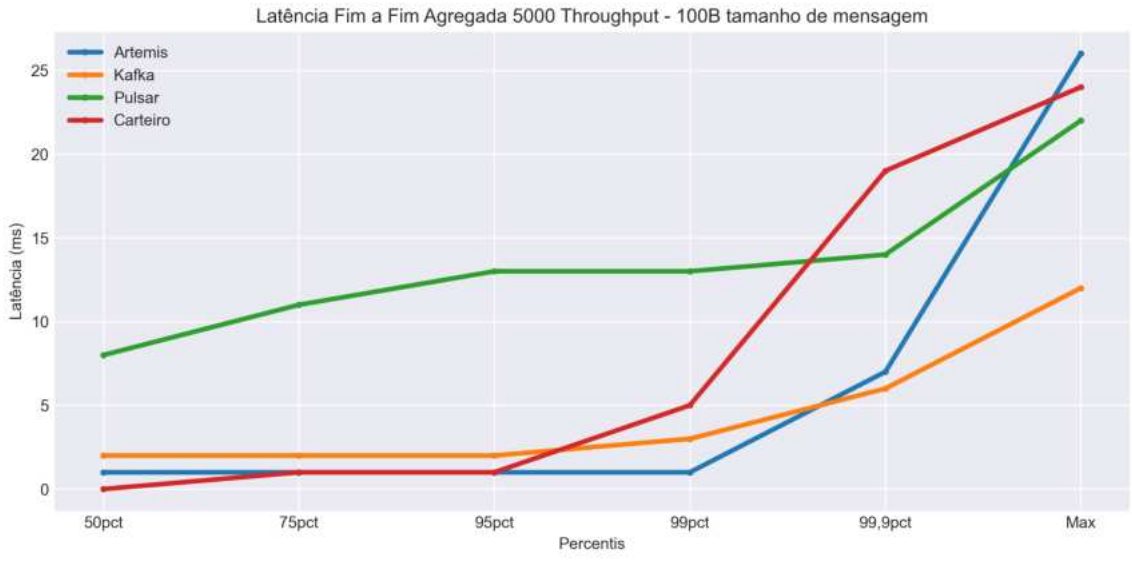
observada em Pulsar e Kafka, especialmente este último, cuja queda foi significativa. Essa redução é atribuída à dependência do armazenamento em disco, que se torna um gargalo nesse cenário. No caso do Kafka, a escalabilidade nesses casos exige também a ampliação da infraestrutura de armazenamento para mitigar esse impacto.

6.5.1.2 Cenário 1 - Baixo throughput e baixa carga

- **Taxa de Produção**: A taxa de envio dos produtores foi configurada para 5000 mensagens por segundo, gerando um volume de dados relativamente pequeno para processamento das corretoras de mensagens.
- **Tamanho das Mensagens**: As mensagens enviadas terão tamanho de 100 bytes, com o conteúdo sendo carregado a partir do arquivo "payload/payload-100b.data" no OMB, e no Carteiro são geradas a partir de um modelo de mensagem protobuf com 100b de mensagens.

Abaixo vemos os resultados em gráfico que compara percentis de latência agregada fim a fim, isto é, desde que a mensagem foi produzida até ser consumida. É importante ressaltar que, dentre as 5 ferramentas, devido ao RabbitMQ ter apresentado uma latência muito superior às demais, chegando a ser discrepante em todos os cenários, foi decidido removê-lo dos gráficos para melhorar a visualização do restante dos MOMs.

Figura 14 – Cenário 1



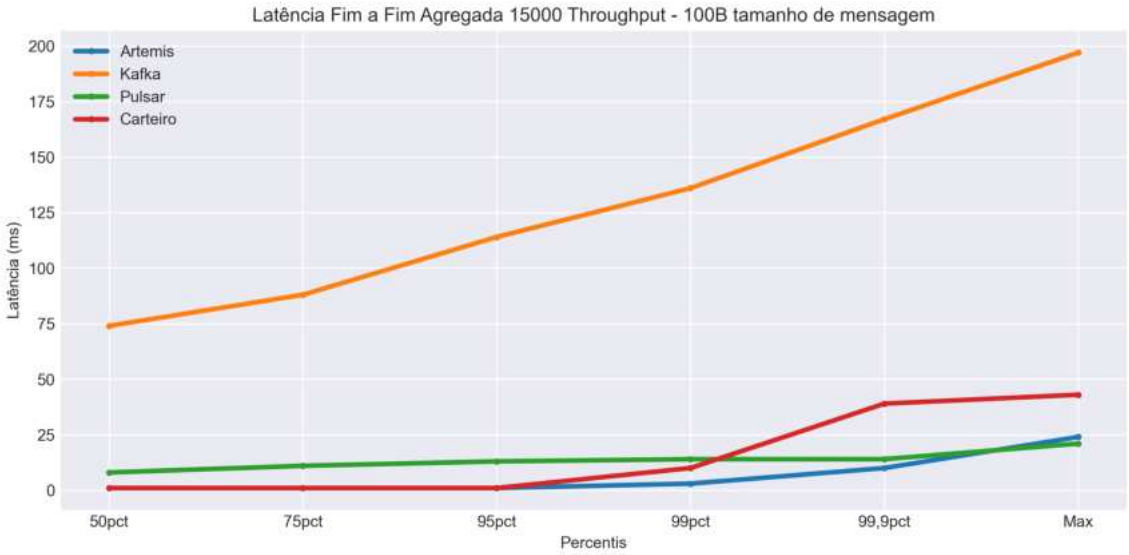
Interpretando o gráfico da figura 14, é possível perceber que até o percentil 50, isto é, cobrindo até metade da amostra global de mensagens, o Carteiro entregou as mensagens mais rápido que os outros MOMs, fato que é explicado pela simplicidade de sua arquitetura. Nos percentis que medem os piores casos, isto é, do percentil 99 até o 100, que representa o pior caso, o Kafka se destacou por causa da sua maior estabilidade, enquanto

as outras ferramentas se comportaram muito similarmente, entregando com cerca de 25 ms.

6.5.1.3 Cenário 2 - Médio throughput e baixa carga

- **Taxa de Produção**: A taxa de envio dos produtores foi configurada para 15000 mensagens por segundo, gerando um volume de dados médio/grande para processamento das corretoras de mensagens.
- **Tamanho das Mensagens**: As mensagens enviadas terão tamanho de 100 bytes, com o conteúdo sendo carregado a partir do arquivo "payload/payload-100b.data"no OMB, e no Carteiro são geradas a partir de um modelo de mensagem protobuf com 100B de mensagens.

Figura 15 – Cenário 2



Em um cenário com aumento de *throughput* para 15000 mensagens por segundo, a figura 15 mostra que até o percentil 95 o Carteiro e Artemis se comportaram muito bem, entregando com ótima velocidade, enquanto Pulsar e Kafka apresentaram maior latência, com o último se destacando negativamente. Essa discrepância pode ter se dado pelo *overhead* das ferramentas clusterizadas, porém, entre essas duas, o Pulsar performou melhor.

Nos últimos percentis, o Artemis e Pulsar foram os que se mantiveram mais estáveis, o Carteiro apresentou leve aumento, e o Kafka subiu a latência uniformemente.

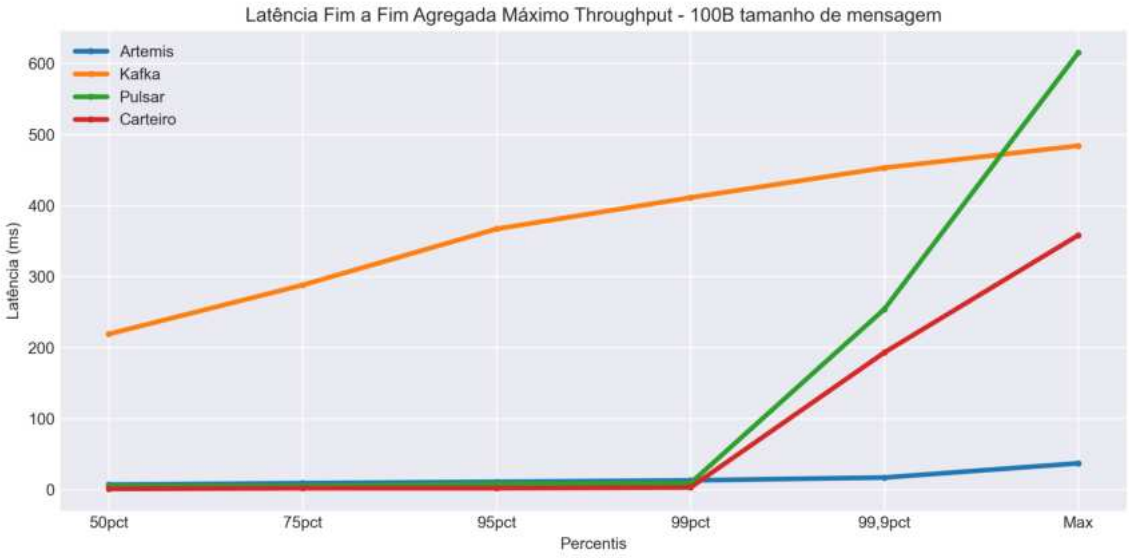
6.5.1.4 Cenário 3 - Máximo throughput e baixa carga

- **Taxa de Produção**: A taxa de envio dos produtores foi configurada para o máximo de mensagens por segundo capazes de serem geradas pelo produtor, gerando um

volume de dados bem estressante para processamento das corretoras de mensagens.

- **Tamanho das Mensagens**: As mensagens enviadas terão tamanho de 100 bytes, com o conteúdo sendo carregado a partir do arquivo "payload/payload-100b.data" no OMB, e no Carteiro são geradas a partir de um modelo de mensagem protobuf com 100B de mensagens.

Figura 16 – Cenário 3



Nesse cenário, com resultados representados pela figura 16, é importante levar em conta o gráfico anteriormente apresentado, que mostrou o *throughput* médio por segundo em cenário de estresse máximo para mensagens com 100B. Em que a posição das ferramentas com maior quantidade de *throughput* médio por segundo foi a seguinte: Pulsar, Kafka, com cerca de 950 mil mensagens por segundo, em seguida, Carteiro com 137 mil e Artemis com 36 mil. Essas métricas são importantes, pois existe uma correlação entre *throughput* máximo alcançado e a degradação da latência em percentis mais elevados.

O Pulsar, Artemis e Carteiro se comportaram com estabilidade até o percentil 99, com o Carteiro se destacando apresentando menor latência; já o Kafka demonstrou latência bastante acima.

Após o percentil 99, o Artemis conseguiu manter estabilidade, o que é explicado por causa do baixo *throughput*, e as outras ferramentas tiveram picos maiores de latência, com o Pulsar até ultrapassando o pior caso do Kafka.

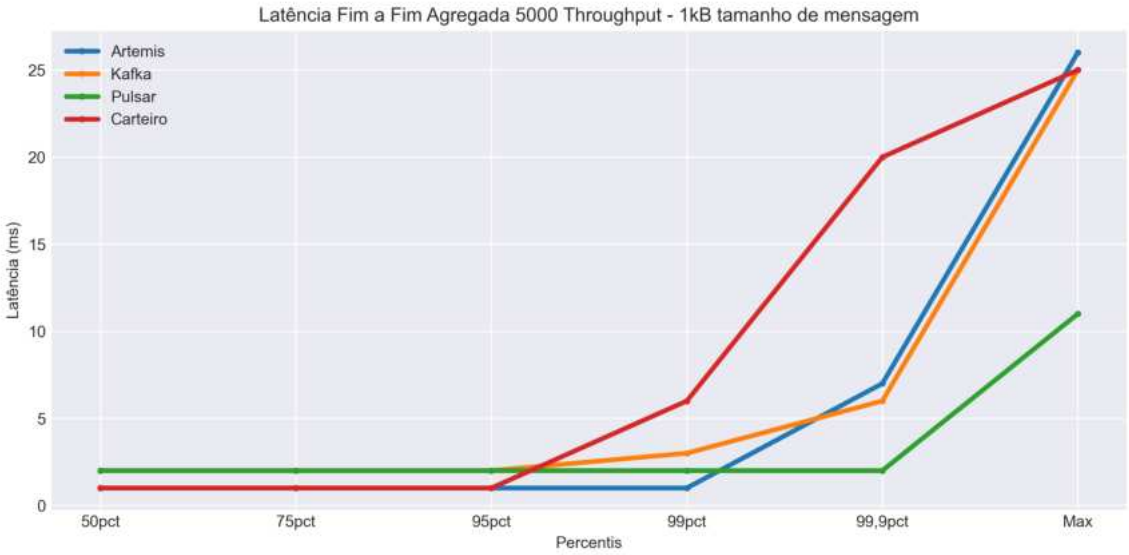
6.5.1.5 Cenário 4 - Baixo throughput e média carga

Os cenários 4, 5 e 6 são análogos aos cenários 1, 2 e 3, respectivamente; a única variação é o tamanho da mensagem, que sai de 100B para 1kB. A ideia é comparar os

resultados com esse aumento de carga, levando em conta tanto o processamento quanto o tráfego de rede que é aumentado.

- **Taxa de Produção**: A taxa de envio dos produtores foi configurada para 5000 mensagens por segundo, gerando um volume de dados relativamente pequeno para processamento das corretoras de mensagens.
- **Tamanho das Mensagens**: As mensagens enviadas terão tamanho de 1 kB, com o conteúdo sendo carregado a partir do arquivo "payload/payload-1kb.data"no OMB, e no Carteiro são geradas a partir de um modelo de mensagem protobuf com 1kB de mensagens.

Figura 17 – Cenário 4



A figura 17 mostra que a variação do tamanho das mensagens não afetou significativamente nem o Carteiro, nem o Artemis, que demonstraram comportamento bem parecido com quando processaram mensagens de 100 B. O Kafka apresentou uma piora significativa nos piores casos, por aumentar o uso do armazenamento do disco, e o Pulsar demonstrou melhoria em relação ao cenário análogo anterior, o que pode ser explicado devido à sua arquitetura que otimiza fluxo de dados, o Pulsar utiliza segmentos para escrever dados em disco de forma otimizada. Para mensagens maiores, essa segmentação permite menos operações de escrita e leitura, reduzindo o tempo necessário para processar cada mensagem. Em mensagens muito pequenas, a fragmentação pode aumentar a latência devido ao maior número de operações.

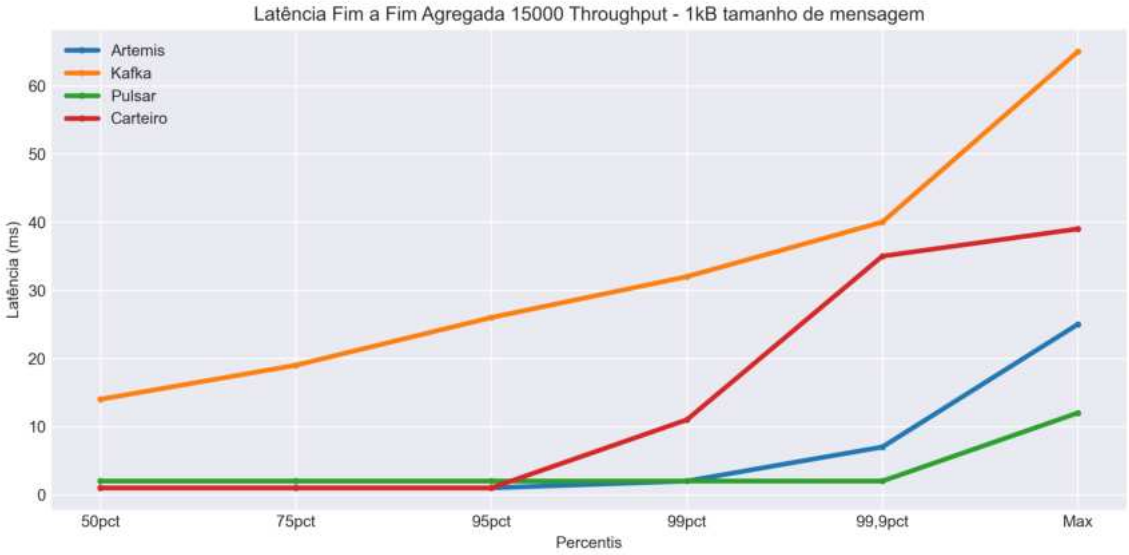
6.5.1.6 Cenário 5 - Médio throughput e média carga

- **Taxa de Produção**: A taxa de envio dos produtores foi configurada para 15000 mensagens por segundo, gerando um volume de dados médio/grande para proces-

samento das corretoras de mensagens.

- **Tamanho das Mensagens**: As mensagens enviadas terão tamanho de 1 kB, com o conteúdo sendo carregado a partir do arquivo "payload/payload-1kb.data" no OMB, e no Carteiro são geradas a partir de um modelo de mensagem protobuf com 1kB de mensagens.

Figura 18 – Cenário 5



Nesse cenário, em que o *throughput* é aumentado, a figura 18 mostra um comportamento parecido com o último cenário. Artemis e Carteiro praticamente não foram afetados pelo aumento do tamanho das mensagens, enquanto Pulsar melhorou seu desempenho, e o Kafka piorou, devido à necessidade de escalar seu disco para manter o desempenho.

6.5.1.7 Cenário 6 - Máximo throughput e média carga

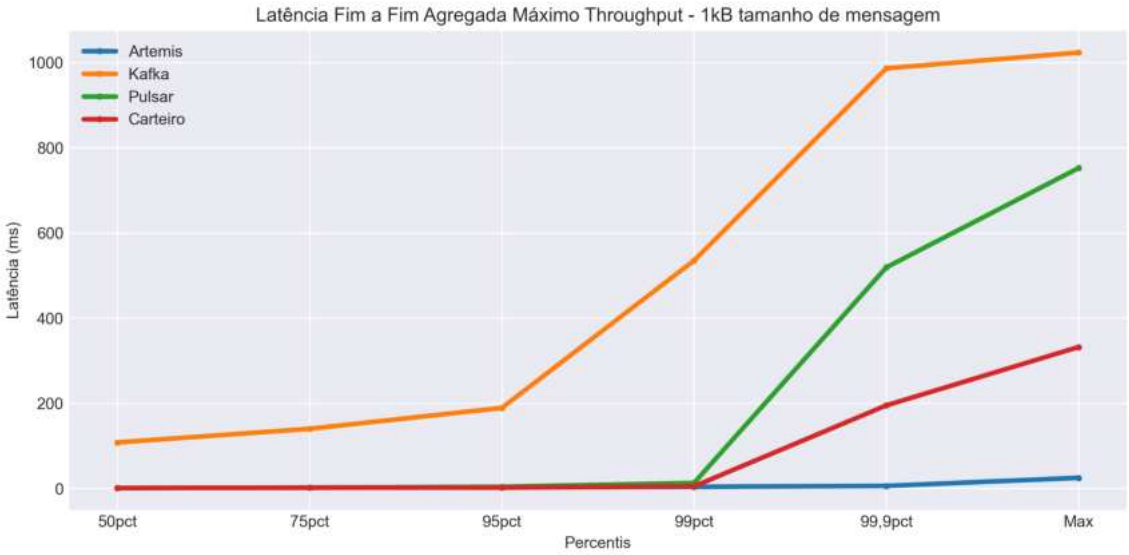
- **Taxa de Produção**: A taxa de envio dos produtores foi configurada para o máximo de mensagens por segundo capazes de serem geradas pelo produtor, gerando um volume de dados bem estressante para processamento das corretoras de mensagens.

- **Tamanho das Mensagens**: As mensagens enviadas terão tamanho de 1 kB, com o conteúdo sendo carregado a partir do arquivo "payload/payload-1kb.data" no OMB, e no Carteiro são geradas a partir de um modelo de mensagem protobuf com 1kB de mensagens.

Novamente temos um cenário que precisamos levar em consideração os *throughputs* médios obtidos pelas ferramentas.

A figura 19 mostra que com estresse máximo e aumento do tamanho das mensagens, o Carteiro, Artemis e Pulsar apresentaram comportamento muito parecido em relação

Figura 19 – Cenário 6



ao cenário com mensagens menores, com ligeiro aumento na latência nos últimos percentis. Levando em consideração o *throughput* médio, é possível concluir que o Carteiro e Pulsar apresentaram melhor desempenho. Enquanto o Kafka não conseguiu manter uma latência baixa em nenhum dos percentis apresentados, o Artemis foi mais estável e rápido justamente por não possuir uma tolerância grande para cenários com alto *throughput*.

6.5.2 Cenários de testes isolados para Carteiro

Além dos cenários de comparação com outras ferramentas, foram projetados cenários específicos para testar exclusivamente o Carteiro. Esses cenários foram elaborados com o objetivo de avaliar tanto a corretude quanto o desempenho da aplicação, considerando suas funcionalidades específicas.

Os testes focaram em situações que explorassem características próprias do Carteiro, como filtragem de mensagens por conteúdo e frequência (absoluta e periódica). Dessa forma, foi possível aferir se a aplicação entrega resultados consistentes de acordo com os parâmetros configurados, além de medir sua eficiência em cenários variados.

Esses cenários exclusivos foram fundamentais para validar a implementação e garantir que o Carteiro atenda às especificações desejadas, enquanto identifica possíveis áreas de melhoria para sua evolução futura.

6.5.2.1 Definição do Formato de Mensagem

Para a realização dos testes no Carteiro, foi utilizado um formato de mensagem baseado no protocolo Protobuf denominado `Deteccao`, que simula uma detecção de radar de trânsito. Este formato inclui informações como o radar responsável pela detecção e os dados do veículo registrado, conforme o código a seguir:

```
syntax = "proto3";

message Radar {
  int64 id_radar = 1;
  string localizacao = 2;
  int32 limite_velocidade = 3;

  enum EstadoRadar {
    NULO = 0;
    INATIVO = 1;
    ATIVO = 2;
  }
  EstadoRadar estado_radar = 4;

  string modelo_radar = 5;
}

message Veiculo {
  enum TipoVeiculo {
    NAO_ESPECIFICADO = 0;
    CARRO = 1;
    CAMINHONETE = 2;
    ONIBUS = 3;
    CAMINHAO = 4;
    TRATOR = 5;
    MOTO = 6;
  }
  TipoVeiculo tipo_veiculo = 1;
  int32 velocidade_registrada = 2;
  string placa = 3;
}

message Deteccao {
  int64 id_deteccao = 1;
  uint64 data_deteccao = 2;
  Veiculo veiculo = 3;
  Radar radar = 4;
}
```

Código 1 – Formato Protobuf da Mensagem Deteccao

Os campos da mensagem `Deteccao` foram definidos para simular um cenário deter-

minístico, garantindo a consistência dos testes e a possibilidade de aferir a corretude dos resultados. Em particular, o campo `modelo_radar` varia circularmente entre os valores arbitrários {"Alerta", "Buzina", "Controle", "Detetor", "Escaneio"}, enquanto o campo `velocidade_registrada` assume valores entre 80 e 89, também de maneira circular. O tamanho das mensagens foi projetado para ter 100 B de tamanho. Essa abordagem garante que as mensagens geradas para os testes sigam padrões conhecidos, facilitando as análises de filtragem por conteúdo e frequência.

6.5.2.2 Definição de Cenários por Requisição POST

A configuração dos cenários para os testes do Carteiro foi realizada através de requisições HTTP do tipo POST, enviadas para endpoints específicos do sistema. Essas requisições são divididas em duas partes principais: a definição da simulação, enviada ao `endpoint` do produtor, e a configuração das subscrições, enviada ao `endpoint` do consumidor.

A definição da simulação descreve as características gerais do teste, como a taxa de mensagens por segundo (`messageRate`), a duração total (`duration`) e o tempo de aquecimento (`warmupDuration`). Um exemplo de configuração enviado ao `endpoint` do produtor é apresentado abaixo:

```
{
    "messageRate": 15000,
    "duration": 480,
    "host": "34.45.17.211",
    "port": 9876,
    "warmupDuration": 180,
    "maxThroughput": false
}
```

Código 2 – Exemplo de configuração da simulação no Produtor

Já a configuração das subscrições define os critérios de recebimento de mensagens no lado do consumidor. Esses critérios incluem filtros de conteúdo, frequências de envio e o tipo de mensagem a ser processada. Um exemplo de configuração enviada ao `endpoint` do consumidor é apresentado a seguir:

```
{
    "ip": "localhost",
    "port": 9877,
    "textFilter": "getRadar().getModeloRadar().contains('e')",
    "frequency": null,
    "period": null,
    "messageType": "Deteccao"
}
```

Código 3 – Exemplo de configuração de subscrição no consumidor

O campo `messageType` no JSON do consumidor garante que apenas mensagens do tipo `Deteccao` sejam processadas, seguindo o formato pré-definido em protobuf (detalhado na seção anterior). Isso assegura consistência no formato das mensagens utilizadas nos testes.

Adicionalmente, o campo `textFilter` permite configurar filtros de conteúdo utilizando expressões do Spring Expression Language (SpEL). No exemplo acima, o filtro seleciona apenas mensagens cujo modelo do radar contenha a letra 'e'. Por fim, os campos `frequency` e `period` possibilitam a personalização do envio, controlando a periodicidade ou probabilidade de as mensagens serem entregues, ajustando-se às especificações de cada cenário de teste.

6.5.2.3 Cenário Padrão Carteiro - Médio throughput

Este cenário foi projetado para servir como uma base de comparação ao avaliar o desempenho do Carteiro. Nele, configurou-se um *throughput* médio de 15.000 mensagens por segundo, sem qualquer tipo de amostragem por filtros ou frequência, permitindo a análise de métricas como latência, taxa de processamento e eficiência geral da aplicação.

```
{
    "messageRate": 15000,
    "duration": 480,
    "host": "34.45.17.211",
    "port": 9876,
    "warmupDuration": 180,
    "maxThroughput": false
}
```

Código 4 – Simulação Médio Throughput

```
{
    "ip": "localhost",
    "port": 9877,
    "textFilter": "",
    "frequency": null,
    "period": null,
    "messageType": "Deteccao"
}
```

Código 5 – Subscrição Médio Throughput

6.5.2.4 Cenário 1 Carteiro - 3 Produtores com Máximo Throughput

Para este cenário, foi configurada uma simulação com três produtores gerando mensagens no máximo *throughput* possível, enquanto apenas um consumidor recebia as mensagens. O objetivo deste teste foi avaliar o limite máximo de desempenho do Carteiro (*cap*), identificando o ponto em que o sistema começa a perder mensagens devido a limitações no processamento ou na capacidade de entrega.

A configuração dos 3 produtores foi realizada através da seguinte requisição POST replicada para cada um deles:

```
{
    "messageRate": 15000,
    "duration": 480,
    "host": "34.45.17.211",
    "port": 9876,
    "warmupDuration": 180,
    "maxThroughput": true
}
```

Código 6 – Configuração da Simulação nos Produtores

Nesta configuração, o parâmetro `maxThroughput` foi definido como `true`, instruindo o sistema a gerar mensagens na capacidade máxima. A simulação foi projetada para durar 480 segundos, com um período inicial de aquecimento (`warmupDuration`) de 180 segundos para estabilizar o sistema antes da coleta dos dados principais.

Já a configuração da subscrição no lado do consumidor foi definida da seguinte forma:

```
{
    "ip": "localhost",
    "port": 9877,
    "textFilter": "",
    "frequency": null,
    "period": null,
    "messageType": "Deteccao"
}
```

Código 7 – Configuração da Subscrição no consumidor

No lado do consumidor, nenhuma filtragem foi aplicada (`textFilter` vazio), garantindo que todas as mensagens do tipo `Deteccao` fossem recebidas e processadas.

Esse cenário foi projetado para testar os limites do sistema sob alta carga, medindo não apenas o desempenho máximo do Carteiro, mas também identificando a quantidade de mensagens que poderiam ser perdidas em condições extremas. Assim, os resultados deste cenário fornecem *insights* cruciais sobre a resiliência e escalabilidade do sistema.

6.5.2.5 Cenário 2 Carteiro - Médio Throughput e 3 Consumidores

Este cenário foi projetado para testar o desempenho do Carteiro ao replicar mensagens para múltiplos consumidores. Foi configurada uma simulação com um único produtor gerando mensagens em um *throughput* médio de 15000 mensagens por segundo, enquanto três consumidores estavam conectados e recebiam as mensagens simultaneamente. O objetivo deste teste foi avaliar se o Carteiro apresenta perda de desempenho ou aumento na latência ao realizar a replicação de mensagens para múltiplos destinatários.

```
{
    "messageRate": 15000,
    "duration": 480,
    "host": "34.45.17.211",
    "port": 9876,
    "warmupDuration": 180,
    "maxThroughput": false
}
```

Código 8 – Configuração da Simulação no Produtor

A configuração acima será padrão para todos os cenários seguintes, por isso, a partir do próximo cenário, será omitida e apenas serão apresentadas as configurações de subscrições a fim de reduzir a prolixidade.

```
{
    "ip": "localhost",
    "port": 9877,
    "textFilter": "",
    "frequency": null,
    "period": null,
    "messageType": "Deteccao"
}
```

Código 9 – Configuração da Subscrição nos Consumidores

Cada um dos três consumidores utilizou uma configuração semelhante, com o `textFilter` vazio para garantir que todas as mensagens do tipo `Deteccao` fossem recebidas sem filtragem adicional. O campo `messageType` assegurou que apenas mensagens no formato protobuf `Deteccao` fossem processadas, mantendo a consistência entre os cenários de teste.

Este cenário permite avaliar como o Carteiro gerencia a replicação de mensagens para múltiplos consumidores, fornecendo informações importantes sobre o impacto no desempenho e na latência em situações de carga moderada. A análise dos resultados pode indicar se o sistema é escalável para cenários de múltiplos consumidores ou se ajustes são necessários.

6.5.2.6 Cenário 3 Carteiro - Filtragem em Campo Numérico

Este cenário foi desenvolvido para avaliar o desempenho do Carteiro quando utiliza filtragem baseada em um campo numérico específico, além de verificar a corretude na redução do espaço amostral de mensagens recebidas. Nesse caso, foi configurada uma simulação com um único produtor enviando mensagens em um *throughput* médio de 15000 mensagens por segundo, enquanto um consumidor estava conectado e recebendo apenas mensagens que satisfizessem o critério de filtragem.

Vale destacar que esse e os próximos cenários terão a mesma configuração de simulação, que passará a ser omitida a partir da próxima subsubseção.

A configuração do produtor foi realizada por meio da seguinte requisição POST:

```
{
    "messageRate": 15000,
    "duration": 480,
    "host": "34.45.17.211",
    "port": 9876,
    "warmupDuration": 180,
    "maxThroughput": false
}
```

Código 10 – Configuração da Simulação no Produtor

Nesta configuração, o produtor operou em um *throughput* médio, com duração total de 480 segundos e período de aquecimento (`warmupDuration`) de 180 segundos para estabilizar o sistema antes da coleta de métricas.

A configuração do consumidor, incluindo o filtro aplicado, foi a seguinte:

```
{
    "ip": "localhost",
    "port": 9877,
    "textFilter": "getVeiculo().getVelocidadeRegistrada()>=85",
    "frequency": null,
    "period": null,
    "messageType": "Deteccao"
}
```

Código 11 – Configuração da Subscrição no Consumidor

O campo `textFilter` foi utilizado para implementar um filtro que permitisse apenas mensagens em que o campo `velocidadeRegistrada` do veículo fosse maior ou igual a 85. Esse critério deve reduzir pela metade o espaço amostral das mensagens recebidas, já que as velocidades foram configuradas de forma determinística para variar entre 80 e 89 de forma circular.

Este cenário foi pensado para atender dois objetivos principais:

- Avaliar se o uso de filtros em campos numéricos causa impacto significativo no desempenho do Carteiro, especialmente em cenários de médio *throughput*.
- Validar a corretude da implementação do filtro, garantindo que apenas 50% das mensagens originais sejam recebidas pelo consumidor, de acordo com o critério estabelecido.

Com isso, é possível analisar se o Carteiro mantém um bom desempenho mesmo ao processar e filtrar mensagens de maneira síncrona, e se o mecanismo de filtragem opera de forma precisa e confiável.

6.5.2.7 Cenário 4 Carteiro - Filtragem em campo de String

Este cenário foi definido para avaliar o desempenho do Carteiro quando utiliza filtragem baseada em um campo de *string* específico, além de verificar a corretude na redução do espaço amostral de mensagens recebidas. Nesse caso, foi configurada uma simulação padrão anteriormente descrita, enquanto um consumidor está conectado e recebendo apenas mensagens que satisfizessem o critério de filtragem.

A configuração do consumidor, incluindo o filtro aplicado, foi a seguinte:

```
1 {
2     "ip": "localhost",
3     "port": 9877,
4     "textFilter": "getRadar().getModeloRadar().contains('e')",
5     "frequency": null,
6     "period": null,
7     "messageType": "Deteccao"
8 }
```

Código 12 – Configuração da Subscrição no Consumidor

O campo `textFilter` foi utilizado para implementar um filtro que permitisse apenas mensagens em que o campo `modeloRadar` do veículo contivesse o caractere '**e**'. Esse critério deve reduzir para 80% o espaço amostral das mensagens recebidas, já que os modelos variam entre os valores {"Alerta", "Buzina", "Controle", "Detetor", "Escaneio"}, em que apenas o segundo valor da estrutura será desconsiderado pelo filtro, por não possuir o caractere desejado.

6.5.2.8 Cenário 5 Carteiro - Filtragem por período modular

Esse cenário foi projetado principalmente para aferir a corretude da filtragem por período modular, em que a subscrição pode escolher receber uma mensagem a cada n mensagens.

```
{
    "ip": "localhost",
    "port": 9877,
    "textFilter": "",
    "frequency": null,
    "period": 3,
    "messageType": "Deteccao"
}
```

Código 13 – Configuração da Subscrição no Consumidor

O campo `period` foi usado para definir que a subscrição deseja receber uma a cada três das mensagens que são produzidas, independentemente do conteúdo presente nelas, com o esperado do resultado sendo que o consumidor receba 33% do total de mensagens produzidas.

6.5.2.9 Cenário 6 Carteiro - Filtragem por frequência absoluta

Nesse cenário, o objetivo é testar a implementação da frequência obtida probabilisticamente, em que a mensagem é enviada com probabilidade `p` para reduzir a frequência total.

```
{
    "ip": "localhost",
    "port": 9877,
    "textFilter": "",
    "frequency": 0.75,
    "period": null,
    "messageType": "Deteccao"
}
```

Código 14 – Configuração da Subscrição no Consumidor

Com uma amostragem suficientemente grande, garantida pelo padrão de simulação que foi utilizado, o esperado é que, da amostragem total de mensagens produzidas pelo produtor, apenas 75% sejam recebidas pelo consumidor que configurou tal subscrição.

6.5.2.10 Cenário 7 Carteiro - Filtragem combinada por frequência e conteúdo

Esse cenário visa avaliar o desempenho e a corretude do Carteiro quando se combinam filtros por frequência e por conteúdo, considerando tanto filtros em campos *string* quanto numéricos ao mesmo tempo.

```
{
    "ip": "localhost",
```

```
3     "port": 9877,
4     "textFilter": "getRadar().getModeloRadar().contains('e') &&
          getVeiculo().getVelocidadeRegistrada()>=85",
5     "frequency": 0.5,
6     "period": null,
7     "messageType": "Deteccao"
8 }
```

Código 15 – Configuração da Subscrição no Consumidor

Nessa subscrição, a frequência absoluta é reduzida para 50% das mensagens, e além disso, são aplicados filtros com lógica de **AND** para o `modeloRadar` conter o caractere '**e**' ao mesmo tempo em que a `velocidadeRegistrada` seja maior ou igual a 85. Esses filtros combinados reduzem bastante o espaço amostral.

- **Frequência**: reduz espaço amostral para 50%
- **Filtro em VelocidadeRegistrada**: reduz novo espaço amostral para 50%
- **Filtro em ModeloRadar**: reduz novo espaço amostral para 80%

Com essa combinação de filtros, o esperado é que apenas 20% do total de mensagens enviadas tenha sido consumido devido às configurações da subscrição.

6.5.2.11 Cenário 8 Carteiro - Filtragem combinada com pouca redução de espaço amostral

Foi definido um último cenário nesse trabalho, a fim de avaliar novamente uma combinação de filtros, mas com redução de espaço amostral significativamente menor do que a configuração do cenário 7.

```
1 {
2     "ip": "localhost",
3     "port": 9877,
4     "textFilter": "getRadar().getModeloRadar().contains('e') &&
          getVeiculo().getVelocidadeRegistrada()<=88",
5     "frequency": 0.8,
6     "period": null,
7     "messageType": "Deteccao"
8 }
```

Código 16 – Configuração da Subscrição no Consumidor

Nessa subscrição, a frequência absoluta é reduzida para 80% das mensagens, e também são aplicados filtros com lógica de **AND** para o `modeloRadar` conter o caractere '**e**' ao mesmo tempo em que a `velocidadeRegistrada` seja menor ou igual a 88. Esses filtros combinados reduzem bastante o espaço amostral.

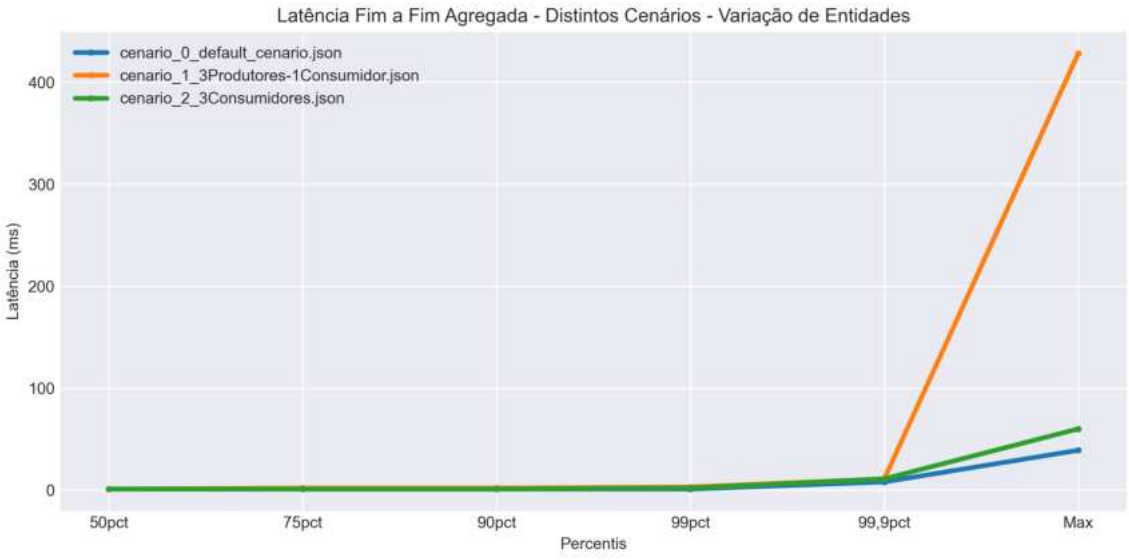
- **Frequência**: reduz espaço amostral para 80%
- **Filtro em VelocidadeRegistrada**: reduz novo espaço amostral para 90%
- **Filtro em ModeloRadar**: reduz novo espaço amostral para 80%

Com essa combinação de filtros, o esperado é que apenas 57,6% do total de mensagens enviadas tenha sido consumido devido às configurações da Subscrição.

6.5.2.12 Resultados de cenários com variações de Entidades

Nessa subseção serão apresentados os resultados dos cenários em que houve variação de entidades, sendo esses os cenários 1 e 2 para os testes unitários do Carteiro. Nele medimos os percentis de latências obtidos nos distintos cenários, em comparação com o teste padrão e a taxa de mensagens consumidas com sucesso, isto é, o percentual entre mensagens consumidas pelos usuários finais em relação ao total de mensagens produzidas pelos produtores.

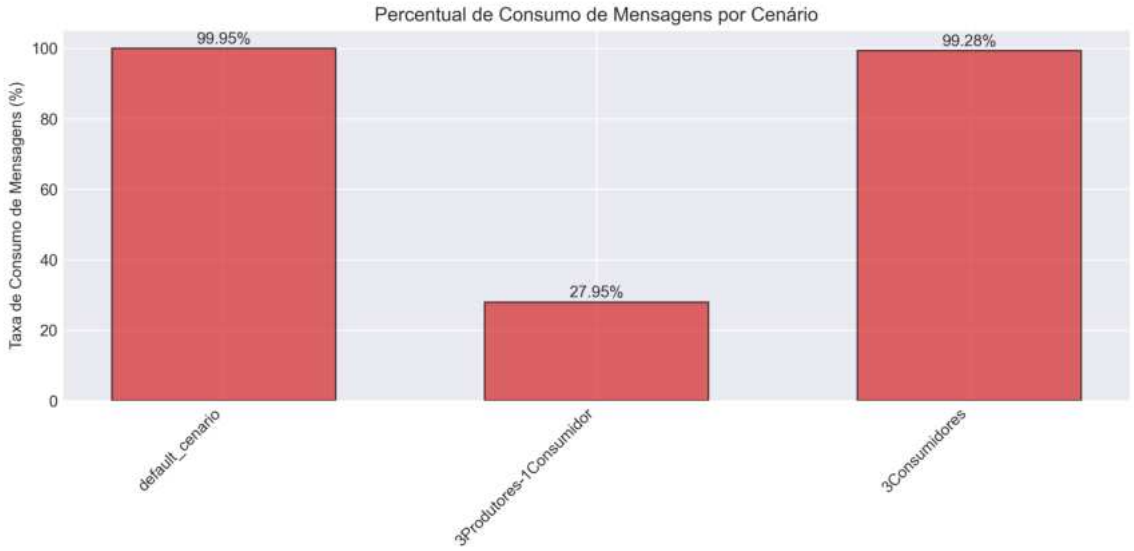
Figura 20 – Variação de Entidades



As figuras 20 e 21 mostram os resultados obtidos em 3 diferentes cenários, evidenciando comparações em termos de latência e taxa de consumo. O cenário 0 é o que serve como base de comparação para avaliar as variações. Nele, a latência se mostra estável em todos os percentis, com o pior caso não ultrapassando os 70 ms, e a taxa de consumo é 99,95%, indicando que o padrão é existir uma perda de 0,05% entre mensagens produzidas e consumidas.

O cenário 1 mostra um impacto da sobrecarga gerada por múltiplos consumidores na sua taxa de consumo; boa parte das mensagens (cerca de 70%) foi descartada pelo Carteiro. Isso é explicado porque um único Carteiro tem um limite de *throughput* que foi

Figura 21 – Taxa de Consumo com Variação de Entidades



atingido, e também impactou o seu pior caso em relação à latência, o que era esperado. Apesar disso, até o percentil 99,9, a latência seguiu o padrão do cenário 0, o que é bem satisfatório.

No cenário 2, também tivemos um comportamento de latência praticamente igual ao cenário base até o percentil 99,9. Seu pior caso variou em relação ao cenário padrão conforme esperado, pois agrega um *overhead* para replicação de mensagens. Em relação à taxa de consumo de mensagens, também foi satisfatória, tendo uma perda ínfima em relação ao caso padrão devido ao *overhead* de replicação.

6.5.2.13 Resultados de cenários com variações em filtros em campo único

Nessa subseção serão apresentados resultados de testes em cenários em que houveram filtros numéricos ou em *string* em campo único, para analisar tanto a corretude quanto o desempenho em relação à latência.

As figuras 22 e 23 mostram os resultados obtidos em 3 diferentes cenários, incluindo um cenário base, evidenciando comparações em termos de latência e taxa de consumo. Esses resultados servem para analisar tanto a corretude, no que tange ao percentual esperado devido às propostas de cenários, quanto ao desempenho quando se compara um filtro de *string* com um filtro numérico.

A corretude está evidenciada, dado que os cenários previam as perdas apresentadas no gráfico devido ao filtro de mensagens de cada um. Quanto ao desempenho, ambas performaram com latência acima do cenário base apenas nos últimos percentis, devido às operações adicionais que foram requisitadas para filtrar o conteúdo das mensagens. É interessante notar que o filtro em *string* apresentou um pior caso acima do que o filtro numérico, fato que pode ser explicado pela operação do filtro de *string* precisar percorrer

Figura 22 – Variação de Filtro em Campo único

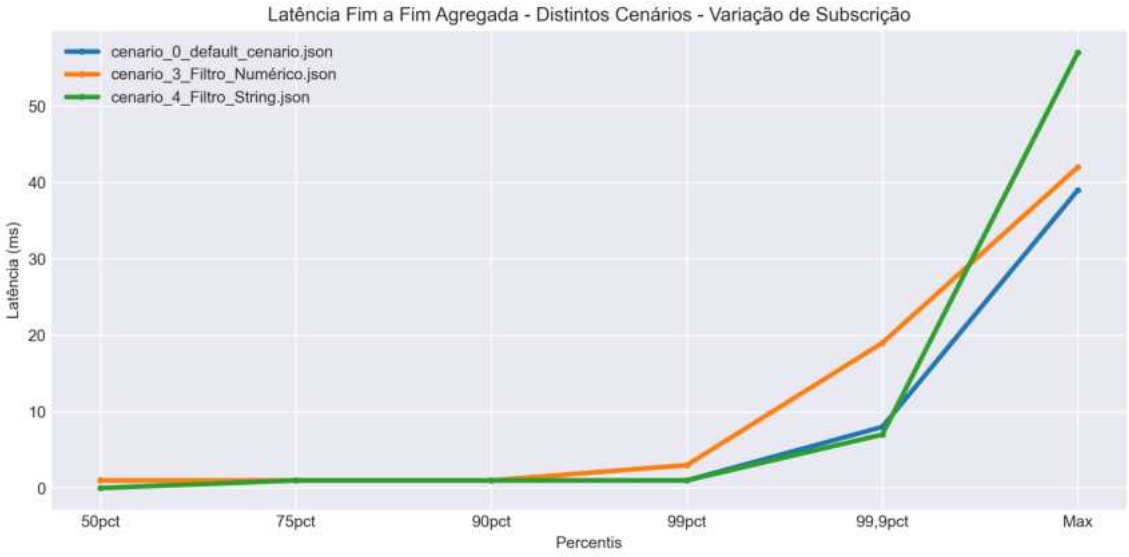
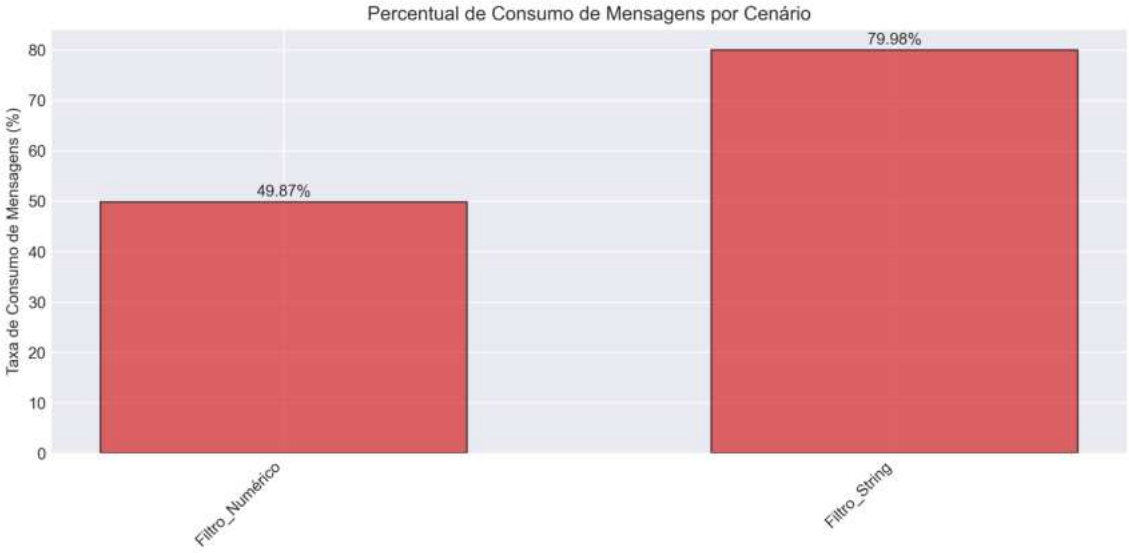


Figura 23 – Taxa de Consumo com Filtro em Campo único



uma cadeia de caracteres para ser realizada, adicionando um *overhead* e causando maior latência.

6.5.2.14 Resultados de cenários com variações em frequência e combinação com filtros de conteúdos

Essa subseção também visa à corretude, mas também permite que sejam extraídas informações sobre latência quando são utilizados filtros por frequência dos dados.

As figuras 24 e 25 mostram os resultados obtidos em 5 diferentes cenários, incluindo um cenário base, evidenciando comparações em termos de latência e taxa de consumo. Analisando primeiro a corretude, todos os cenários obtiveram suas taxas esperadas de

Figura 24 – Variação em frequência e filtragem combinada de conteúdo

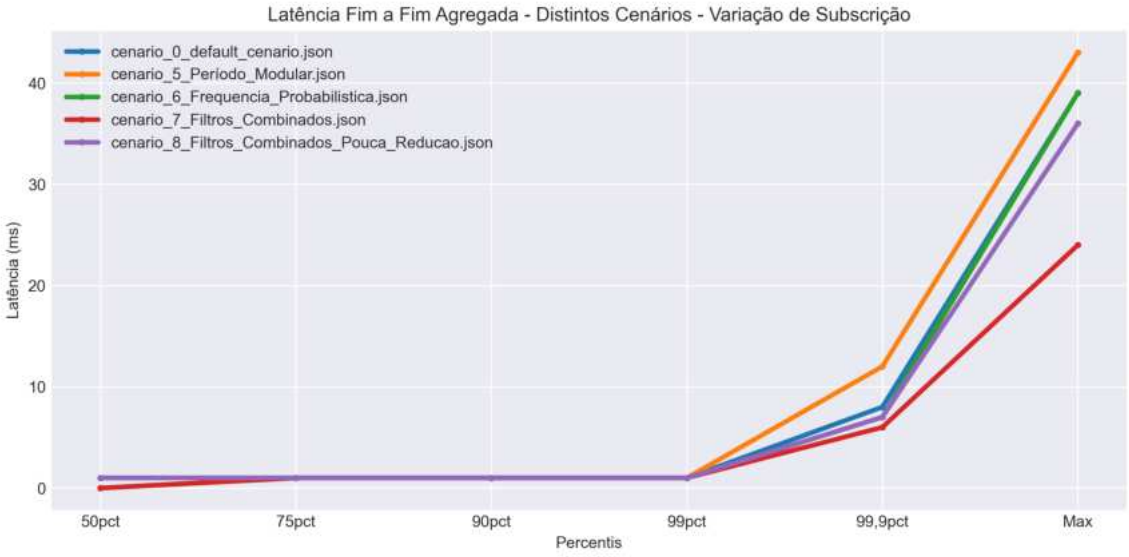
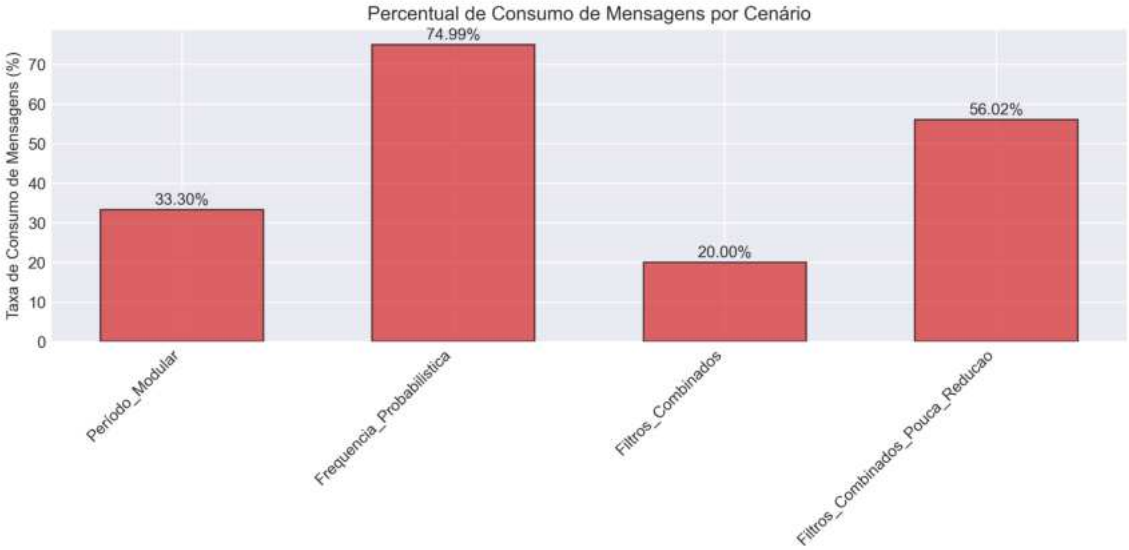


Figura 25 – Taxa de Consumo com Variação em frequência e filtragem combinada de conteúdo



maneira fiel, a maior discrepância foi do cenário com filtros combinados e pouca redução de amostra global, em que o esperado seria 57% e o real foi 56%, isso evidencia que os filtros funcionaram de maneira correta.

O cenário com filtragem de frequência por período modular foi o que demonstrou maior latência em comparação com a base. Esse atraso pode ser explicado pela quantidade de operações adicionais necessárias pela lógica do envio por período modular, que foi explicada na seção da apresentação do Carteiro. O filtro por Frequência probabilística é uma linha quase sobreposta ao cenário base, pois o único adicional é enviar a mensagem com probabilidade p. É notável no gráfico que existiram cenários que desempenharam com latência menor ao cenário base; é contra-intuitivo adicionar filtros e a latência aumentar,

mas faz sentido, pois os filtros por frequência reduzem o espaço amostral antes da filtragem por conteúdo, no sentido de diminuir a quantidade de mensagens processadas, e isso contribui para que o Carteiro consiga entregar as mensagens com mais velocidade ainda. É possível concluir que, com maior redução de espaço amostral devido a filtro de frequência, mais rápido a mensagem será entregue.

6.6 ANÁLISE DE CUSTOS DE SERVIÇOS DE MENSAGERIA

Embora este estudo tenha sido baseado majoritariamente em ferramentas de código aberto, é importante considerar as alternativas comerciais amplamente utilizadas no mercado. Para essa análise, foram comparados os custos das seguintes soluções: **Amazon Kinesis**, **Google Cloud Pub/Sub** e **Azure Web PubSub**, além da solução própria, o **Carteiro**, hospedada em **Google Cloud Platform (GCP)** (CLOUD, 2008) e **Amazon Web Services (AWS)** (SERVICES, 2006).

A metodologia de precificação considerou o caso de uso do Carteiro em sua capacidade máxima, processando 116.668 mensagens por segundo, cada uma com 1 kB de tamanho, as enviando exclusivamente para um único consumidor, totalizando aproximadamente $3{,}024 \times 10^{14}$ bytes por mês. Para um cenário conservador, assumiu-se um funcionamento ininterrupto, 24 horas por dia, ao longo de 30 dias, simulando o custo mais alto possível para cada ferramenta.

Também é importante ressaltar que provedores de nuvem como GCP e AWS organizam seus serviços em regiões geográficas independentes, que contêm zonas de disponibilidade, que são subdivisões físicas com redundância e alta conectividade. A comunicação entre zonas é mais rápida e econômica do que entre regiões.

6.6.1 Azure Web PubSub

O **Azure Web PubSub** (MICROSOFT, 2021) precifica seu uso considerando unidades de processamento e custo de envio de mensagens. Cada unidade abstrai a capacidade de conexões simultâneas para publicação e assinatura de mensagens. Para este estudo, utilizou-se 1 unidade, suficiente para até 1.000 conexões simultâneas, operando 730 horas mensais, ao custo de **$0,0671 por hora**, totalizando **$48,97 mensais**.

Além disso, o custo do tráfego de mensagens é calculado a partir do volume mensal, considerando 2 KiB como tamanho fixo por mensagem. É fundamental ressaltar que mensagens com tamanho inferior a este valor são agrupadas até atingirem tal limite. Consequentemente, o tamanho reduzido das mensagens em questão (1 kB) não exerce influência sobre o cálculo final dos custos. Com um tráfego estimado de 295.316 bilhões de KiB, chega-se a 147,6 bilhões de mensagens mensais. A Azure concede 30,42 milhões de mensagens gratuitas por mês, e o custo final da ferramenta ficou em **$147.627,58 mensais**.

6.6.2 Amazon Kinesis

A precificação do **Amazon Kinesis** (SERVICES, 2013) considera os seguintes fatores:

- Volume de dados ingeridos ($0,08 por GB).
- Volume de dados exportados ($0,04 por GB).
- Região de hospedagem do serviço.

Para reduzir custos, a **região US-East** foi escolhida. Com um volume mensal de 302.403,46 GB de dados ingeridos e exportados, o custo total foi de **$36.288**. Adicionando **$28,80 de impostos**, a estimativa final ficou em **$36.317 mensais**.

6.6.3 Google Cloud Pub/Sub

O modelo de precificação do **Google Cloud Pub/Sub** (CLOUD, 2016) leva em conta:

- Volume de dados publicados por dia (em TB).
- Número de subscrições.
- Configuração de persistência de mensagens (não aplicável neste caso).

Como a solução não requer retenção de mensagens, este último fator não impacta os custos. O tráfego diário estimado foi de 10,1 TB, com uma única subscrição. Dessa forma, o custo mensal final foi de **$22.351**.

6.6.4 Carteiro na GCP

A solução **Carteiro** foi hospedada na **Google Cloud Platform**, utilizando a calculadora de custos da própria GCP para estimar os valores. Diferente das ferramentas comerciais, que incluem todo o serviço de mensageria em um único pacote, no Carteiro foi necessário calcular separadamente os custos de computação e rede.

6.6.4.1 Custos de Computação

A máquina virtual utilizada foi configurada com:

- 4 núcleos, 8 vCPUs, 32 GiB de memória RAM e 10 GB de armazenamento SSD.
- Funcionamento de 730 horas/mês.

O custo da computação foi **$197,37 mensais**.

6.6.4.2 Custos de Rede

Os custos de rede foram divididos em:

- Endereços de IP públicos: **$18,24 mensais**.
- Transferência de dados, variando conforme o destino:
  - Para a internet: **$15.933,38**.
  - Para outras regiões da GCP: **$5.632,70**.
  - Para a mesma região, mas em outra zona: **$2.816,35**.
  - Para a mesma zona: Gratuito.

6.6.5 Carteiro na AWS

A solução também foi testada em **Amazon Web Services (AWS)**, utilizando uma máquina EC2 equivalente à da GCP, configurada com:

- 4 núcleos, 8 vCPUs, 32 GiB de memória RAM e 10 GB de armazenamento SSD.
- Funcionamento de 730 horas/mês.

O custo da computação na AWS foi de **$141,47 mensais**.
Os custos de rede na AWS foram estimados da seguinte maneira:

- Mesma zona: Gratuito.
- Mesma região, mas zona diferente: **$3.072**.
- Outra região: **$6.144**.
- Internet: **$19.251**.

Assim, os custos totais do Carteiro na AWS foram:

- **$141,47** (tráfego dentro da mesma zona).
- **$3.213,47** (tráfego dentro da mesma região, mas em outra zona).
- **$6.285,47** (tráfego para outra região).
- **$19.392,67** (tráfego para a internet).

6.6.6 Comparação geral

A Tabela 2 apresenta um comparativo dos custos estimados:

Tabela 2 – Comparação de custos estimados das soluções de mensageria (ordem decrescente)

Serviço	**Custo Mensal Estimado (USD)**
Azure Web PubSub	147.627,58
Amazon Kinesis	36.317,00
Google Cloud Pub/Sub	22.351,00
Carteiro (AWS - tráfego internet)	19.392,67
Carteiro (GCP - tráfego internet)	16.148,99
Carteiro (AWS - tráfego entre regiões)	6.285,47
Carteiro (GCP - tráfego entre regiões)	5.848,30
Carteiro (GCP - tráfego entre zonas)	3.031,96
Carteiro (AWS - tráfego entre zonas)	3.213,47
Carteiro (GCP - tráfego dentro da mesma zona)	215,61
Carteiro (AWS - tráfego dentro da mesma zona)	141,47

7 CONCLUSÃO

Os resultados obtidos dos cenários de testes comparando todas as ferramentas demonstram que cada MOM analisado possui um caso de uso ideal, dependendo dos requisitos de *throughput*, latência e infraestrutura disponível.

O Apache Kafka destacou-se como a solução mais apropriada para cenários onde o *throughput* extremamente elevado é essencial, especialmente quando há possibilidade de escalabilidade horizontal e expansão da infraestrutura de armazenamento em disco, mesmo que à custa de maior latência e dificuldade de configuração.

O Apache Pulsar também mostrou-se adequado para aplicações de alto volume de dados e *throughput*, mas exige uma infraestrutura robusta para ser bem aproveitado, além de apresentar alta complexidade em suas configurações.

Em contraste, o ActiveMQ Artemis revelou-se mais adequado para cenários onde o volume de mensagens é reduzido, mas a latência deve ser minimizada ao máximo, mesmo nos piores casos, sendo uma escolha viável para aplicações que exigem resposta quase imediata; também demonstrou ser facilmente configurável e implantável.

O RabbitMQ, por sua vez, não foi favorecido pelos cenários testados, mas sua arquitetura de roteamento avançada e suporte a transações garantem confiabilidade e entrega segura de mensagens, sendo amplamente utilizado em sistemas que priorizam integridade e controle sobre o fluxo das mensagens; e apresenta facilidade na sua implantação e configuração.

Por fim, o Carteiro, solução desenvolvida neste estudo, dentre aquelas que operaram em nó único, nos cenários de testes propostos, destacou-se como a ferramenta com maior *throughput*, além de apresentar baixa latência, sendo uma alternativa viável para sistemas que exigem alta taxa de entrega de mensagens de maneira síncrona sem necessidade de persistência.

Em cenários de larga escala, no entanto, foi observado que o Carteiro começou a enfrentar dificuldades, especialmente quando comparado a ferramentas que operam em clusters. Essa limitação decorre do fato de o Carteiro operar em nó único, enquanto as ferramentas de mercado se beneficiam de arquiteturas distribuídas para lidar com volumes extremos de mensagens.

Apesar dessa limitação, o Carteiro mostrou-se vantajoso para a maioria dos cenários de teste avaliados. Além de sua simplicidade de implementação e baixos requisitos de infraestrutura, a análise de precificação evidenciou que, quando comparado às soluções comerciais analisadas, ele é competitivo em custo de manutenção. Isso é especialmente válido em implementações que operam na mesma região ou na mesma zona de disponibilidade dos consumidores, onde os custos de transferência de dados são significativamente reduzidos.

Outro fator relevante identificado foi a capacidade do Carteiro de otimizar custos operacionais por meio de filtragem inteligente de mensagens. Como permite aplicar filtros por conteúdo ou por amostragem, consumidores localizados em diferentes regiões ou conectados via internet pública, situações que normalmente elevam os custos de transferência, podem receber apenas mensagens realmente relevantes para suas necessidades. Dessa forma, o Carteiro se apresenta como uma possível ferramenta estratégica para reduzir despesas operacionais ao minimizar o volume de dados trafegado em redes de alto custo.

Adicionalmente, foi identificada uma possibilidade de melhoria na solução própria, através de mudanças em sua arquitetura e na reestruturação e refatoração de alguns de seus módulos; poderíamos ter ganhos em desempenho. A introdução de um balanceador de carga, como o Nginx, permitiria distribuir as mensagens entre vários nós do Carteiro, ampliando sua capacidade de processamento, mas possivelmente com perda em latência. Em conjunto, o uso de um banco de dados em memória, como o Redis, possibilitaria o gerenciamento de subscrições de forma concorrente entre diferentes nós de Carteiro.

Com essas adaptações, mesmo que complexas, o Carteiro poderia evoluir para uma solução escalável horizontalmente. Essa abordagem permitiria aliar os benefícios de simplicidade e baixos custos à capacidade de atender a requisitos mais complexos e exigentes.

Em síntese, o Carteiro não apenas se demonstrou eficiente para os objetivos inicialmente propostos, como também apresentou potencial para evolução. Além de, nos cenários testados, ser competitivo em desempenho com as ferramentas *open-source*, também se mostrou uma alternativa viável em termos de custos frente às soluções comerciais, podendo vir a ser uma opção simples e econômica no contexto de *streaming* de mensagens.

REFERÊNCIAS

AKIN, E. **Kafka Architecture**. 2023. https://medium.com/@cobch7/kafka-architecture-43333849e0f4. Accessed: 2025-03-06.

ARAUJO, M. N. M. **Carteiro**. 2025. https://github.com/MatheusABorges/carteiroTCC. Accessed: 2025-03-06.

ARAUJO, M. N. M. **Carteiro Consumer**. 2025. https://github.com/MatheusABorges/MessageConsumerTCC. Accessed: 2025-03-06.

ARAUJO, M. N. M. **Carteiro Producer**. 2025. https://github.com/MatheusABorges/MessageProducerTCC. Accessed: 2025-03-06.

CHY, M. S. H. et al. Comparative evaluation of java virtual machine-based message queue services: A study on kafka, artemis, pulsar, and rocketmq. **Electronics**, v. 12, n. 23, 2023. ISSN 2079-9292. Disponível em: https://www.mdpi.com/2079-9292/12/23/4792.

CLOUD, G. **Google Cloud Platform (GCP)**. 2008. https://cloud.google.com/. Accessed: 2025-03-06.

CLOUD, G. **Google Cloud Pub/Sub**. 2016. https://cloud.google.com/pubsub?hl=pt-BR. Accessed: 2025-03-06.

DOCKER, I. **Docker**. 2013. https://www.docker.com/. Accessed: 2025-03-06.

DOCKER, I. **Docker Hub**. 2014. https://hub.docker.com/. Accessed: 2025-03-06.

D'SILVA, G. M. et al. Real-time processing of iot events with historic data using apache kafka and apache spark with dashing framework. **RTEICT**, p. 1805–1809, 2017.

DU, T. X. Q.; YUHE. Anomaly detection and diagnosis for container-based microservices with performance monitoring. **Springer Nature Switzerland**, v. 48, n. 6, p. 561–572, 2018.

FOUNDATION, A. S. **Apache Software Foundation**. 1999. https://www.apache.org/. Accessed: 2025-03-06.

FOUNDATION, A. S. **Apache ZooKeeper**. 2008. https://zookeeper.apache.org/. Accessed: 2025-03-06.

FOUNDATION, A. S. **Apache BookKeeper**. 2011. https://bookkeeper.apache.org/. Accessed: 2025-03-06.

FOUNDATION, A. S. **Apache Kafka**. 2011. https://kafka.apache.org/. Accessed: 2025-03-06.

FOUNDATION, A. S. **Apache ActiveMQ Artemis**. 2015. https://activemq.apache.org/components/artemis/. Accessed: 2025-03-06.

FOUNDATION, A. S. **Apache Pulsar**. 2016. https://pulsar.apache.org/. Accessed: 2025-03-06.

FOUNDATION, A. S. **Main Concepts and Terminology**. 2020. https://kafka.apache.org/documentation/. Accessed: 2025-03-06.

FOUNDATION, A. S. **Artemis Core Architecture Documentation**. 2021. https://activemq.apache.org/components/artemis/documentation/2.17.0/architecture.htmlh. Accessed: 2025-03-06.

FOUNDATION, A. S. **Pulsar Architecture Overview**. 2024. https://pulsar.apache.org/docs/4.0.x/concepts-architecture-overview/. Accessed: 2025-03-06.

FOUNDATION, T. L. **Benchmarking workloads**. 2022. https://openmessaging.cloud/docs/benchmarks/. Accessed: 2025-03-06.

GOOGLE. **Protocol Buffers**. 2001. https://protobuf.dev/. Accessed: 2025-03-06.

HAQ, S. ul. **Introduction to Monolithic Architecture and MicroServices Architecture**. 2018. https://www.wallarm.com/what/what-is-amqp#:~:text=AMQP%20refers%20to%20Advanced%20Message,between%20client%20and%20broker%20parties.

HARRIS, C. **Microsserviços versus arquitetura monolítica**. 2021. https://www.atlassian.com/br/microservices/microservices-architecture/microservices-vs-monolith. Accessed: 2025-03-06.

HOHPE, G.; WOOLF, B. **Enterprise Integration Patterns, Designing, Building, and Deploying Message Solutions**. [S.l.]: Addison-Wesley Professional, 2003. 736 p.

IBM. **POJOs (Plain Old Java Objects)**. 2000. https://www.ibm.com/docs/pt-br/rsm/7.5.0?topic=architecture-pojos-plain-old-java-objects. Accessed: 2025-03-06.

LEE, I. **Advanced Message Queuing Protocol - AMQP**. 2024. https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63.

LTD, J. **draw.io (now diagrams.net)**. 2011. https://app.diagrams.net/. Accessed: 2023-10-15.

MAYER-SCHöNBERGER, V.; CUKIER, K. **Big Data: A Revolution That Will Transform How We Live, Work, and Think**. Boston, MA: Houghton Mifflin Harcourt, 2013. ISBN 978-0544002692.

MICROSOFT. **Azure Web PubSub**. 2021. https://azure.microsoft.com/pt-br/products/web-pubsub. Accessed: 2025-03-06.

OASIS. **MQTT 5 Specification**. 2024. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

OPENMESSAGING. **OpenMessaging Benchmark**. 2018. https://openmessaging.cloud/docs/benchmarks/. Accessed: 2025-03-06.

ORACLE. **Java Message Service (JMS)**. 2001. https://www.oracle.com/java/technologies/java-message-service.html. Accessed: 2025-03-06.

PARK, J.; PARK, J. Rtp/rtcp based real-time protocol over ethernet for distributed control system. **IFAC Proceedings Volumes**, v. 33, n. 30, p. 107–112, 2000. ISSN 1474-6670. 16th IFAC Workshop on Distributed Computer Control Systems (DCCS 2000), Sydney, Australia, 29 November-1 December 2000. Disponível em: https://www.sciencedirect.com/science/article/pii/S1474667017367381.

SERVICES, I. A. W. **Amazon Web Services (AWS)**. 2006. https://aws.amazon.com/. Accessed: 2025-03-06.

SERVICES, I. A. W. **Amazon Kinesis**. 2013. https://aws.amazon.com/pt/pm/kinesis/. Accessed: 2025-03-06.

SOUSA, R. et al. Software tools for conducting real-time information processing and visualization in industry: An up-to-date review. **Applied Sciences**, v. 11, n. 11, 2021. ISSN 2076-3417. Disponível em: https://www.mdpi.com/2076-3417/11/11/4800.

T.A MARK D. GRIFFITHS, V. E. R. Online data collection from video game players: Methodological issues. **Mary Ann Liebert**, v. 7, n. 5, p. 512–518, 2004.

TALHAOUI, M. A. Real-time data stream processing challenges and perspectives. **IJCSI**, v. 14, n. 5, p. 6–12, 2017.

TANZU, V. **Spring Boot**. 2022. https://spring.io/projects/spring-boot. Accessed: 2025-03-06.

UMMUSALMA. **RabbitMQ cluster Setup using Ansible**. 2023. https://devops.supportsages.com/rabbitmq-cluster-setup-using-ansible-c35eccbd47bc. Accessed: 2025-03-06.

VMWARE. **Spring Expression Language (SpEL)**. 2022. https://docs.spring.io/spring-framework/docs/3.0.x/reference/expressions.html. Accessed: 2025-03-06.

VMWARE, I. **RabbitMQ**. 2007. https://www.rabbitmq.com/. Accessed: 2025-03-06.

WALTHER, J. B. et al. The effect of message persistence and disclosure on liking in computer-mediated communication. **Media Psychology**, Taylor & Francis, v. 21, n. 2, p. 308–327, 2018.

WANG, C.; GILL, C.; LU, C. Frame: Fault tolerant and real-time messaging for edge computing. In: **2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)**. [S.l.: s.n.], 2019. p. 976–985.