UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Iago da Cunha Corrêa

RACK: PROPOSTA DE CONFIGURAÇÃO PARA GARANTIA DE ENTREGA DE MENSAGENS NO APACHE KAFKA

Santa Maria, RS
2022

Iago da Cunha Corrêa

RACK: PROPOSTA DE CONFIGURAÇÃO PARA GARANTIA DE ENTREGA DE MENSAGENS NO APACHE KAFKA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC), Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

ORIENTADORA: Prof.ª Patrícia Pitthan de A. Barcelos

Santa Maria, RS
2022

This study was financied in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) – Finance Code 001

Corrêa, Iago da Cunha
rAck: Proposta de Configuração para Garantia de Entrega de Mensagens no Apache Kafka / Iago da Cunha Corrêa.- 2022.
69 p.; 30 cm

Orientadora: Patrícia Pitthan de Araújo Barcelos
Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Ciência da Computação , RS, 2022

1. Sistemas Distribuídos 2. Mensageria 3. Apache Kafka I. Barcelos, Patrícia Pitthan de Araújo II. Título.

Sistema de geração automática de ficha catalográfica da UFSM. Dados fornecidos pelo autor(a). Sob supervisão da Direção da Divisão de Processos Técnicos da Biblioteca Central. Bibliotecária responsável Paula Schoenfeldt Patta CRB 10/1728.

Declaro, IAGO DA CUNHA CORRêA, para os devidos fins e sob as penas da lei, que a pesquisa constante neste trabalho de conclusão de curso (Dissertação) foi por mim elaborada e que as informações necessárias objeto de consulta em literatura e outras fontes estão devidamente referenciadas. Declaro, ainda, que este trabalho ou parte dele não foi apresentado anteriormente para obtenção de qualquer outro grau acadêmico, estando ciente de que a inveracidade da presente declaração poderá resultar na anulação da titulação pela Universidade, entre outras consequências legais.

Iago da Cunha Corrêa

RACK: PROPOSTA DE CONFIGURAÇÃO PARA GARANTIA DE ENTREGA DE MENSAGENS NO APACHE KAFKA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC), Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

Aprovado em 11 de abril de 2022:

Patrícia Pitthan de A. Barcelos, Dra. (UFSM)
(Presidenta/Orientadora)

João Vicente Ferreira Lima, Dr. (UFSM)

Dalvan Jair Griebler, Dr. (PUCRS)

Santa Maria, RS
2022

DEDICATÓRIA

Dedico este trabalho inteiramente a minha mãe, Rozana, que ao passar por diversos problemas de saúde durante o meu ciclo de mestrado, me ensinou ainda mais sobre resiliência, fé e esperança.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por permitir que eu esteja vivenciando mais este momento em minha trajetória profissional.

Aos meus pais, Rozana, Denis e irmãos, Lívia, Iuri e Mariene pelo amor, apoio e motivação incondicional que foram de fundamental importância durante a pós-graduação.

À minha orientadora Patrícia Pitthan Barcelos, a qual me orienta desde a graduação por meio dos trabalhos de iniciação científica e foi uma das principais responsáveis por me incentivar durante a vida acadêmica. Demonstro aqui minha eterna gratidão pela paciência, amizade, profissionalismo.

À Universidade Federal de Santa Maria, ao Centro de Tecnologia e à Coordenação do Curso de Pós-Graduação em Ciência da Computação pela excelente estrutura de formação oferecida e o ótimo ambiente de aprendizagem cultuado.

Agradeço também aos colegas do Laboratório de Sistemas de Computação pelos debates e trocas de conhecimento que foram indispensáveis para a elaboração deste trabalho. Agradecimento especial ao Lucas Ferreira, amigo fundamental e que me acompanha desde o início da graduação.

Aos meus demais amigos que me acompanharam durante os últimos dois anos e contribuíram com conselhos, orientações e me incentivaram desde o começo do mestrado. Muito obrigado Lucas Lima, Lucas Ritter, Henrique Muniz, Guilherme Rodrigues, Matheus Gomes, Eduardo Maciel, Enrique Chimainski, Fernando Bandinelli, Tamiris Couto e Joel Ferreira.

Talvez não tenha conseguido fazer o melhor, mas lutei para que o melhor fosse feito. Não sou o que deveria ser mas graças a Deus não sou o que era antes.

(Marthin Luther King)

RESUMO

RACK: PROPOSTA DE CONFIGURAÇÃO PARA GARANTIA DE ENTREGA DE MENSAGENS NO APACHE KAFKA

AUTOR: Iago da Cunha Corrêa
ORIENTADORA: Patrícia Pitthan de A. Barcelos

Atualmente, uma das formas de atingir escalabilidade e flexibilidade durante a construção de aplicações complexas é adotando o paradigma de microsserviços. Por meio de microsserviços, aplicações podem ser fragmentadas em processos isolados e independentes que se comunicam entre si. Entretanto, existe uma problemática presente neste paradigma que trata especificamente da comunicação entre os diversos micro processos de uma mesma aplicação. Para a resolução desta problemática, pode-se aplicar o conceito de mensageria do Apache Kafka para realizar a troca de mensagens entre cada serviço. O Apache Kafka é uma plataforma de mensageria e *streaming* de dados que segue um modelo produtor-consumidor. A arquitetura do Kafka conta com o mecanismo de *Reconhecimento Positivo* (ack), o qual visa garantir a entrega de mensagens. Apesar de existirem três níveis de configuração para *ack*, os mesmos apresentam restrições de confiabilidade ou desempenho durante a transmissão de mensagens em redes instáveis, obrigando usuários a priorizarem um destes requisitos. Este trabalho tem por objetivo apresentar o *Reliable Ack* (*rAck*), uma configuração para transmissão de mensagens baseada no monitoramento, identificação e recuperação de mensagens em caso de perda. Uma fase experimental foi conduzida a fim de comparar a configuração *rAck* com os níveis padrões de *ack* em termos de confiabilidade e desempenho. Os experimentos foram estruturados para realizar transmissões de mensagens com introdução de falhas de rede que resultavam em perda de pacotes e atraso na entrega de pacotes. Nos cenários com perda de pacotes, a configuração *rAck* demonstrou melhor desempenho frente aos níveis padrões do Kafka. Já nos cenários com atraso na entrega de pacotes, as falhas introduzidas não impactaram os níveis padrões, mantendo desempenhos melhores que configuração *rAck*. No que tange à confiabilidade, em ambos os cenários a configuração *rAck* foi capaz de recuperar todas as mensagens perdidas. Em contrapartida, os níveis padrões estavam sujeitos a perdas ou duplicações de mensagens.

Palavras-chave: Sistemas Distribuídos. Mensageria. Apache Kafka.

ABSTRACT

RACK: PROPOSED OF CONFIGURATION FOR MESSAGE DELIVERY GUARANTEE IN APACHE KAFKA

AUTHOR: Iago da Cunha Corrêa
ADVISOR: Patrícia Pitthan de A. Barcelos

Nowadays, one of the ways to achieve scalability and flexibility during implementation of applications is by adopting the microsservices paradigm. Through microservices, applications can be fragmented into isolated and independent processes that communicate with each other. However, a concern present in the microservice paradigm specifically deals with the communication between microservices of a same application. To solve this concern, the Apache Kafka messaging concept can be applied to exchange messages between each service. Apache Kafka is a messaging and data *streaming* platform that follows a producer-consumer model. Kafka's architecture relies on the *Positive Recognition* (ack) mechanism, which aims to guarantee the delivery of messages. Although there are three configuration levels for *ack*, they present reliability or performance restrictions during the transmission of messages in unstable networks, forcing users to prioritize one of these requirements. This work aims to present *Reliable Ack* (*rAck*), a configuration for message transmission based on monitoring, identification, and recovery of messages in case of loss. An experimental phase was conducted to compare the *rAck* configuration with the standard *ack* levels in terms of reliability and performance. The experiments were structured to carry out message transmissions with the introduction of network failures that resulted in packet loss and delay in delivery. In the packet loss scenarios, the *rAck* configuration showed better performance when compared to Kafka's default levels. In scenarios with delays in the delivery of packages, the introduced failures did not impact the default levels, maintaining better performances than *rAck* configuration. Regarding reliability, in both scenarios, the *rAck* configuration was able to recover all lost messages. In contrast, the default levels were subject to message loss or duplication.

Keywords: Distributed Systems. Messaging. Apache Kafka.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
TTL	Time to Live
$TRAK$	Testing the Reliability of Apache Kafka
$AMQP$	Advanced Message Queuing Protocol
IOT	Internet of Things
$MQTT$	Message Queuing Telemetry Transport
QoS	Quality of Service
$VANET$	Veicular Ad Hoc Network

SUMÁRIO

1 INTRODUÇÃO

O aumento contínuo na quantidade de dados produzidos pela sociedade tem gerado cada vez mais demanda por serviços de Tecnologia da Informação (TI) mais robustos, confiáveis e escaláveis. Atualmente, uma das principais formas de suprir tal demanda tem se baseado na adoção de aplicações distribuídas, fragmentadas em microsserviços (DRAGONI et al., 2017).

Em Dragoni et al. (2017) observa-se uma definição que descreve microsserviços como um conjunto de processos independentes, coesos e que interagem entre si através de mensagens. Analisando tal definição, identifica-se que uma das principais preocupações do modelo arquitetural de microsserviços refere-se à confiabilidade na entrega de mensagens entre serviços compõem determinada aplicação.

No contexto de microsserviços, garantir que uma mensagem seja entregue entre emissor e receptor é de fundamental importância. Isto ocorre pois ao adotar a arquitetura de microsserviços na implementação de aplicações, cada microsserviço apenas identifica o momento certo de executar suas funcionalidades mediante notificações, ou seja, mensagens. Neste cenário, eventuais perdas de mensagens no canal de comunicação entre os microsserviços prejudica o bom funcionamento da arquitetura como um todo.

Para abstrair as dificuldades impostas pela comunicação entre microsserviços existem diversos protocolos de comunicação baseados em mensageria (Luzuriaga et al., 2015). Similarmente, ferramentas também são implementadas para realizar comunicação centrada em mensagens, como é o caso do RabbitMq (PIVOTAL, 2015) e o Apache Kafka (FOUNDATION, 2011). Destaca-se que este trabalho centra-se no Apache Kafka e no seu mecanismo de garantia de entrega de mensagens.

O Kafka é uma plataforma *open source* de processamento distribuído e tolerante a falhas que pode ser aplicada tanto em mensageria quanto em *streaming* de dados. Implementado para ser capaz de enviar grandes quantidades de mensagens em um pequeno intervalo do tempo, o Kafka demonstra-se uma opção viável para aplicações distribuídas em microsserviços onde a comunicação, o transporte e o processamentos dos dados necessite ocorrer em tempo hábil.

A estrutura do Kafka baseia-se em um modelo produtor-consumidor. Neste sentido, um produtor Kafka é responsável por transmitir uma mensagem a um *cluster* Kafka. Por sua vez, uma mensagem sempre é transmitida a um *cluster* Kafka constituído por um ou mais *brokers*. Cada *broker* executa uma instância do Kafka, recebendo as mensagens para que elas sejam consumidas por aplicações consumidoras.

Caracteristicamente tolerante a falhas, o Kafka apresenta dois mecanismos que garantem a tanto a disponibilidade de mensagens quanto a garantia de entrega, sendo eles: o Fator de Replicação (FR) e o Reconhecimento Positivo. A aplicabilidade de ambos

os mecanismos contribui para que as consequências resultantes de eventuais falhas em *brokers* sejam minimizadas (NARKHEDE; SHAPIRA; PALINO, 2017).

O mecanismo de Reconhecimento Positivo (*Positive Acknowledgement*, ou simplesmente *ack*), implementa a confiabilidade na entrega de mensagens (NARKHEDE; SHAPIRA; PALINO, 2017). O mecanismo de Reconhecimento Positivo apresenta três configurações distintas, a saber: *Fire-and-Forget*, *Leader Confirmation* e *Replicas Confirmation*. O nível *Fire-and-Forget* não apresenta confirmação de entrega de mensagem, sendo este um nível sujeito a perdas. Diferentemente, em *Leader Confirmation* é retornada uma confirmação de entrega da mensagem ao emissor sempre que a mesma chega ao destino. Já em *Replicas Confirmation*, a confirmação da entrega é retornada ao emissor sempre que, além de entregue no destino, a mensagem também foi replicada em diversas máquinas, aumentando assim a disponibilidade da mesma.

Observando o cenário de aplicações distribuídas em microsserviços, identificou-se a existência de aplicações que exigem que dados sejam transmitidos ao seu destino em tempo hábil tanto para processamento quanto para apoio a tomada de decisão (JHA et al., 2019) (Alaasam; Radchenko; Tchernykh, 2019) (KUL et al., 2021). Ao utilizar Kafka, essas aplicações são prejudicadas pelo mecanismo de *ack*, ainda mais em transmissões em redes instáveis, que podem resultar em perdas de mensagens. Isto ocorre pois ao priorizar desempenho e assumir a possibilidade de perder mensagens, quando uma perda de mensagem ocorre de fato, perde-se uma notificação ou um dado que seria vital a execução de uma tarefa. Diferentemente, ao priorizar confiabilidade, compromete-se o desempenho da aplicação, podendo ocasionar em gargalos de desempenho em microsserviços específicos.

Este trabalho visa apresentar uma nova configuração para o mecanismo de *acks* do Apache Kafka. Esta configuração, denominada *Reliable Ack* ou, simplesmente *rAck*, consiste na implementação e incorporação de três módulos à arquitetura do Kafka, visando aumentar a confiabilidade de transmissões de mensagens sem confirmação de entrega. Por meio do desenvolvimento deste estudo, espera-se contribuir com os seguintes itens:

- Desenvolver um estudo aprofundado do mecanismo de *acks* do Kafkfa, bem como detalhar o funcionamento de cada um dos níveis de confiabilidade existentes;
- Apresentar uma nova configuração para transmissões de mensagens em produtores Kafka que seja equilibrada em termos de confiabilidade e desempenho;
- Comparar o configuração apresentada neste estudo com os níveis existentes de *acks* do Kafka;
- Contribuir com o cenário de transmissões de mensagens entre microsserviços, sobretudo em redes de transmissão sujeitas a falhas.

Este trabalho está organizado em cinco capítulos. O Capítulo 2 apresenta uma fundamentação teórica sobre a arquitetura base do Apache Kafka, destacando o seu mecanismo de Reconhecimento Positivo e também apresentando a relação existente entre o Kafka e o Zookeeper. Ainda no Capítulo 2, descreve-se trabalhos identificados como correlatos. O Capítulo 3 descreve a configuração *rAck*, nível de configuração de *ack* desenvolvido neste trabalho, detalhando o funcionamento dos módulos implementados. No Capítulo 4 detalha-se a condução da fase experimental deste estudo, apresentando e discutindo os resultados obtidos nos experimentos. Por fim, o Capítulo 5 apresenta as considerações finais e direciona os trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta uma fundamentação teórica que trata a respeito dos temas abordados no presente trabalho. A Seção 2.1 descreve as principais características do Apache Kafka, com ênfase em sua arquitetura. Na Seção 2.2 é detalhado o funcionamento do mecanismo de Reconhecimento Positivo para a confirmação de entrega de mensagens de produtores Kafka. Por fim, a Seção 2.3 apresenta o Apache Zookeeper, um *framework open-source* comumente aplicado na gestão de sistemas distribuídos.

2.1 APACHE KAFKA

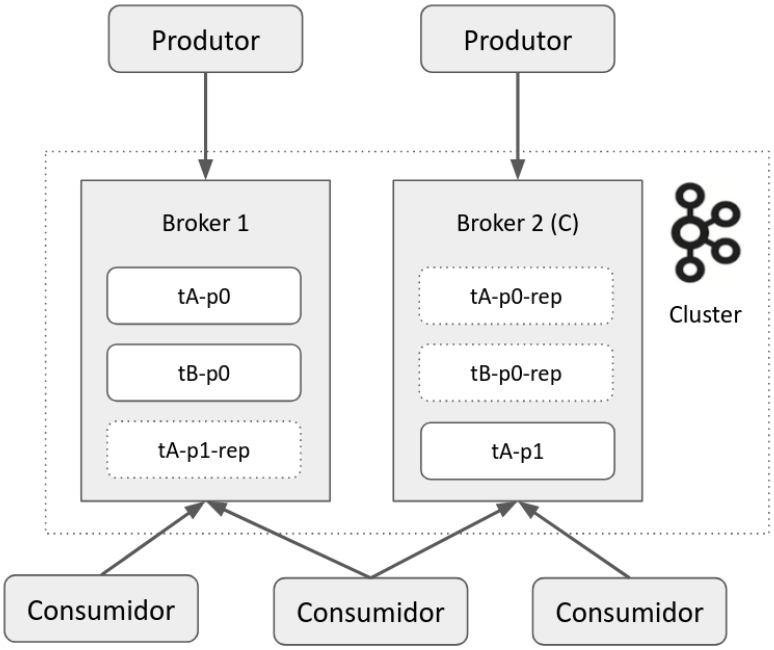
O Apache Kafka é uma plataforma *open-source* de processamento distribuído destinada a oferecer um serviço de mensageria. Este serviço de mensageria centra-se em um modelo produtor-consumidor, onde um produtor é responsável por criar uma mensagem e enviá-la a um *cluster* Kafka. A mensagem enviada está apta a ser lida por um consumidor, que pode ser tanto uma aplicação externa, por meio de uma API (*Application Programming Interface*), quanto um cliente Kafka. Um *cluster* Kafka é composto por diversas máquinas denominadas *brokers* (também conhecidas como servidores Kafka) que executam paralelamente os serviços do Kafka, cooperando entre si de forma descentralizada (NARKHEDE; SHAPIRA; PALINO, 2017).

Ao ser enviada para os *brokers* Kafka, uma mensagem é armazenada em entidades denominadas “tópicos". Os tópicos são categorias lógicas que delimitam as mensagens de acordo com o contexto em que foram produzidas e, posteriormente, consumidas. Os tópicos Kafka são armazenados fisicamente em disco nos *brokers*, em formato de partições (NARKHEDE; SHAPIRA; PALINO, 2017). As partições, por sua vez, são unidades paralelas cuja principal função é armazenar as mensagens. Um único tópico pode ser constituído por uma ou mais partições e todas as partições de um mesmo tópico são distribuídas através dos *brokers* para agregar paralelismo de leitura e escrita.

A Figura 2.1 ilustra dois produtores Kafka enviando mensagens a um *cluster* Kafka composto por dois *brokers* responsáveis por armazenar dois tópicos, sendo eles o tópico A (tA) e o tópico B (tB). O tópico A é composto pelas partições p0 (tA-p0) e p1 (tA-p1), enquanto o tópico B é composto apenas pela partição p0 (tB-p0). Observa-se ainda consumidores realizando leituras das mensagens armazenadas nas partições dos tópicos presentes nos *brokers* 1 e 2.

Uma das principais características do Apache Kafka é ser tolerante a falhas. Assim, o Kafka é capaz de suportar falhas nos *brokers* e se manter operante. Os mecanismos de tolerância a falhas do Kafka são o Fator de Replicação e o Reconhecimento Positivo (*Po-*

Figura 2.1 – Tópicos e partições distribuídos em um *cluster* Kafka.



Fonte: Autor.

sitive Acknowledgement) ou simplesmente *ack*. Este trabalho concentra-se no mecanismo de *ack*, conforme aborda a Seção 2.2.

O Fator de Replicação é uma técnica de tolerância a falhas baseada em redundância que tem por objetivo aumentar a disponibilidade. No Kafka, a replicação ocorre em nível de partição. As partições de cada tópico são armazenadas no disco dos *brokers* em arquivos *append-only*, i.e., arquivos que somente podem ser incrementados. Cada incremento nos arquivos das partições consiste em uma nova mensagem recebida. Os arquivos de partições são replicados e distribuídos através dos *brokers* para aumentar a disponibilidade das mensagens.

Apesar de atuarem de forma descentralizada, um dos *brokers* sempre é definido como controlador. Este *broker* controlador tem o papel de realizar a gestão de todos tópicos e partições distribuídos em todas as máquinas *brokers*. Na Figura 2.1, o controlador está representado por meio do *broker* 2.

A quantidade de réplicas para cada arquivo de partição é definida por uma variável de controle estipulada no momento da criação de cada tópico. Dentre as réplicas de cada partição, uma é eleita como a líder. A réplica líder é responsável por receber requisições de leitura e escrita, enquanto as réplicas não líderes possuem como tarefa buscar o perfeito sincronismo de mensagens com a líder.

Conforme mostra a Figura 2.1, o fator de replicação utilizado pelos tópicos A e B é 2. Sendo assim, para cada partição presente em A e B existem 2 réplicas, sendo uma líder

e outra não líder. As partições réplicas do tópico A são tA-p0-rep e tA-p1-rep, já a partição réplica do tópico B é tB-p0-rep, visto que B apresenta apenas uma partição. Durante o processo de transmissão de mensagens, apenas partições líderes são responsáveis por receber requisições de leitura e escrita. Então, caso um produtor realize uma transmissão de mensagem ao tópico A, a mensagem pode ser destinada tanto para tA-p0, quanto para tA-p1. Quando um tópico apresenta mais de uma partição, o produtor distribui a transmissão das mensagens entre as partições de forma igualitária, fazendo com que as mensagens de um tópico não fiquem presentes em apenas uma partição.

Quando um *broker* apresenta uma falha que lhe afete a operacionalidade, o *broker* Kafka controlador verifica se no *broker* falho existe uma partição líder de determinado tópico. Caso exista, o controlar elege alguma das partições réplicas para ser a nova líder. Novamente na Figura 2.1, caso o *broker* 1 fique indisponível, o *broker* 2, sendo o controlador, irá definir as partições tA-p0-rep e tB-p0-rep para serem as partições líderes de seus respectivos tópicos.

Um atributo fundamental para partições não líderes é o *in-sync* replicas. Dentre todas réplicas não líderes de uma partição, existem as que conseguem se manter em perfeito estado de sincronia com a partição líder, conseguindo apresentar as mesmas mensagens que a partição líder. Quando uma partição líder apresenta uma falha, o *broker* controlador busca avaliar qual a melhor partição a assumir a liderança, sendo as partições *in-sync* as mais aptas a assumir tal função. Isto ocorre pois tais partições são capazes de assumir a liderança sem que determinadas mensagens tenham se perdido junto com a partição líder.

A distribuição das réplicas ocorre de modo a impedir que duas réplicas de uma mesma partição sejam colocadas no mesmo *broker*. Deste modo, caso um *broker* apresente falhas, as réplicas assumem a liderança e passam a receber requisições de leitura e escrita, evitando perdas de partições e indisponibilidade de mensagens.

Enquanto o Fator de Replicação implementa a disponibilidade de mensagens através de diversos *brokers*, o mecanismo de *ack* se preocupa com a confiabilidade empregada em transmissões de mensagens. Por padrão, o Kafka utiliza o protocolo de TCP para realizar a comunicação entre produtores e *brokers*.

Por realizar comunicação sob o protocolo TCP, toda e qualquer mensagem que for transmitida pelo Kafka na rede será encapsulada em um pacote TCP. Quando um pacote que armazena uma mensagem eventualmente se perde durante a transmissão, é dever o protocolo TCP identificar perda – o que ocorre por meio de *ack* na camada de transporte do modelo OSI – e realizar uma retransmissão do pacote perdido, sendo tais retransmissões de pacotes transparentes ao Kafka.

Entretanto, apesar de que as transmissões de mensagens no Kafka já utilizarem por padrão retransmissões de pacotes, os produtores Kafka adicionam uma camada adicional de confiabilidade denominada Reconhecimento Positivo.

2.2 RECONHECIMENTO POSITIVO NO APACHE KAFKA

No Kafka, o Reconhecimento Positivo (*Positive Acknowledgement*, ou simplesmente *ack*) baseia-se no mesmo conceito dos protocolos de comunicação utilizados nas camadas de transporte do modelo OSI (TANENBAUM; WETHERALL, 2011). Neste sentido, um *ack* na camada de transporte do modelo OSI consiste em uma confirmação de entrega de um pacotes que é retornada ao emissor sempre que o receptor recebeu o pacote com sucesso. Caso um *ack* não seja retornado ao emissor, subtende-se que o pacote não foi recebido, sendo necessário iniciar uma retransmissão. Analogamente, o mecanismo de Reconhecimento Positivo específico do Kafka também realiza confirmação de entrega de mensagens por meio de *ack*. Deste modo, um produtor Kafka, ao realizar uma transmissão de mensagem, também aguarda que um *ack*, ou seja, uma confirmação de entrega, seja retornado pelo *broker* receptor da mensagem (NARKHEDE; SHAPIRA; PALINO, 2017).

Mesmo sabendo que o protocolo TCP já efetua retransmissões de pacotes, ainda existem casos em que, devido a instabilidade ou grandes distâncias geográficas, o pacote é perdido definitivamente. Nessas situações a perda recorrente de pacotes pode impactar na transmissão satisfatória de mensagens. Assim, a camada adicional proposta pelo Reconhecimento Positivo no Kafka visa assegurar que mesmo em situações de perda definitiva de pacotes, os produtores de mensagens consigam identificar tais perdas (por meio da camada de garantia de entrega adicional) e realizar uma nova transmissão da mensagem.

Quando um produtor Kafka está configurado para receber confirmações de entrega (*ack*) e uma mensagem é perdida, consequentemente não há como nenhuma confirmação ser retornada ao emissor da mensagem. Sendo assim, na falta de uma confirmação de entrega, o produtor entende que é necessário iniciar a retransmissão. Por padrão, o tempo utilizado por produtores Kafka para aguardar um *ack* de *brokers* antes de iniciar uma retransmissão é de 30 segundos.

Apesar do reconhecimento positivo ser capaz de assegurar que as mensagens irão chegar ao destino apesar das perdas, adicionar uma camada a mais de confiabilidade tende a prejudicar o desempenho das transmissões. Visando não restringir-se apenas à confiabilidade, o mecanismo de Reconhecimento Positivo implementa três níveis de configuração, os quais diferem entre si tanto em relação à confiabilidade como ao desempenho. Tais níveis apresentam propostas de funcionamento distintas, permitindo que o usuário defina o que priorizar durante transmissões de mensagens, confiabilidade ou desempenho.

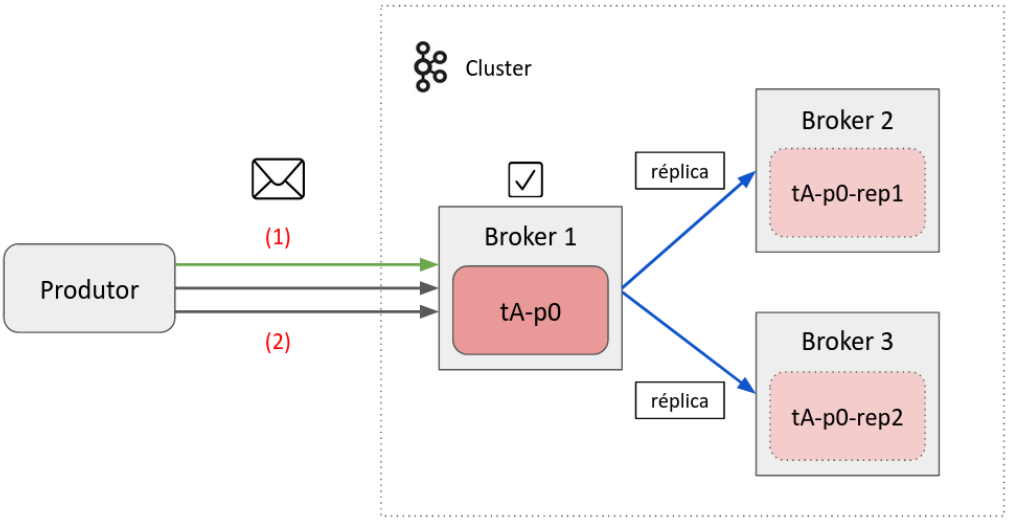
Os níveis de configuração para *ack* alteram a exigência de um produtor perante à persistência de cada mensagem nas partições presentes nos *brokers* (NARKHEDE; SHAPIRA; PALINO, 2017). Produtores Kafka permitem três níveis de configuração para o mecanismo de *acks*. A primeira configuração é denominada ***Fire-and-Forget*** e está associada ao valor `ack = 0`. Com esta configuração, produtores consideram uma mensagem entregue tão logo ela seja enviada através da rede. Assim, não se faz necessário uma

confirmação de entrega dos *brokers* antes de enviar a próxima mensagem. Ao utilizar esta configuração, o produtor Kafka não consegue identificar perdas de mensagens e estas se tornam permanentes, não havendo meios para retransmissão.

A Figura 2.2 exibe um produtor Kafka que está realizando transmissões de mensagens com o mecanismo de *ack* configurado em *Fire-and-Forget*. Na Figura, o produtor está transmitindo mensagens para o tópico tA. Deste modo, ao transmitir mensagens para este tópico, elas automaticamente são direcionadas à partição líder deste tópico, representada na Figura pela partição p0, localizada no *broker* 1. A primeira etapa, identificada pelo número 1 na Figura, exibe a transmissão de uma única mensagem caracterizada pela seta na cor verde.

Visto que este produtor está configurado com o nível de configuração *Fire-and-Forget*, o produtor não aguarda uma confirmação de entrega a ser retornada pela partição líder do tópico. Sendo assim, o produtor já transmite as mensagens subsequentes. Na etapa 2, é possível observar as próximas transmissões, caracterizadas pelas setas na cor cinza.

Figura 2.2 – Nível de configuração *Fire-and-Forget*.



Fonte: Autor.

Em paralelo com as transmissões de mensagens, o *broker* 1 inicia a replicação das mensagens para as partições réplicas de p0. Na Figura 2.2, a replicação das mensagens para as partições réplicas de p0 (tA-p0-rep1 e tA-p0-rep2) está representada pelas setas na cor azul.

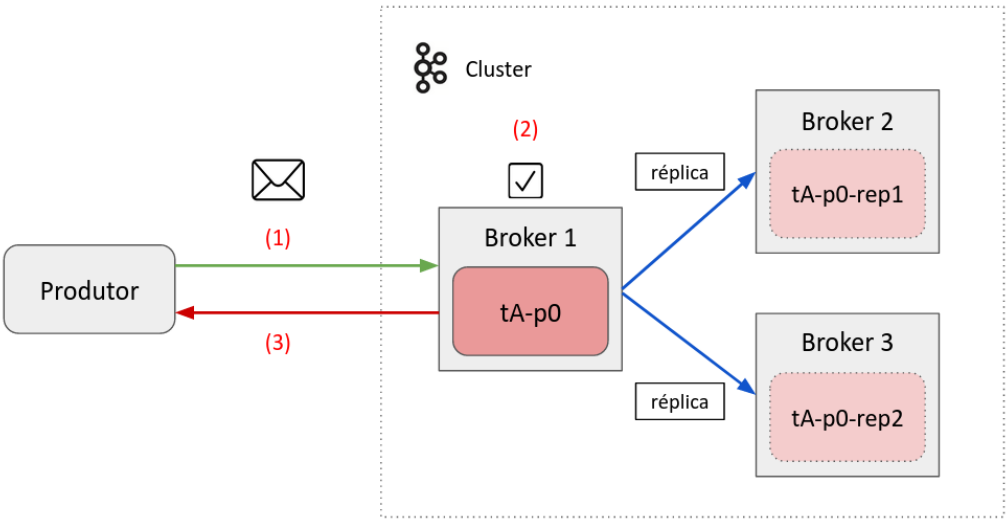
O segundo nível de configuração para *acks* é chamado de ***Leader Confirmation*** e está associado valor `ack = 1`. Nesta configuração, o produtor recebe uma confirmação de entrega sempre que a mensagem enviada foi recebida pela partição líder do tópico destino. Caso o produtor envie uma mensagem e a partição líder deixe de retornar um *ack* ao produtor, a mensagem é retransmitida. Com a configuração *Leader Confirmation*,

produtores conseguem identificar perdas efetivas de mensagens devido à ausência de *acks*. Entretanto, perdas de mensagens ainda são observadas em casos onde a partição líder de um tópico apresenta uma falha antes de replicar todas as suas mensagens entre as réplicas daquela partição.

A Figura 2.3 exibe um produtor Kafka que realiza transmissões de mensagens com o mecanismo de *ack* configurado em *Leader Confirmation*. Como este nível sugere, uma confirmação de entrega é retornada ao produtor sempre que a mensagem chega com sucesso ao destino. Na Figura, é observado que o produtor realiza a transmissão de uma mensagem ao tópico tA, que é recebida pela partição líder p0, presente no *broker* 1. A mensagem transmitida está representada pela seta na cor verde (etapa 1). Uma vez transmitida a mensagem, o produtor aguarda uma confirmação de entrega. Neste caso, o produtor não inicia a transmissão de uma nova mensagem. Já na etapa 2 da Figura, observa-se que a mensagem foi recebida pela partição p0. Com isso, o *broker* 1 busca observar qual o nível de configuração para o mecanismo de *ack* é exigido para a mensagem recebida. Ao identificar que a mensagem exige uma confirmação de entrega, o mesmo *broker* 1 retorna um *ack* ao produtor. Tal confirmação de entrega pode ser observada por meio da seta na cor vermelha, presente na etapa 3. Após a conclusão da etapa 3, o produtor pode iniciar a transmissão das mensagens subsequentes.

A Figura 2.3 também permite observar que, assim que a mensagem alcança a partição p0 (*broker* 1), imediatamente é inicializado o processo de replicação nas partições réplicas, ou seja, tA-p0-rep1 e tA-p0-rep2. Esta replicação ocorre concomitantemente ao retorno da confirmação ao produtor, visto que o nível *Leader Confirmation* exige apenas que a mensagem tenha sido recebia pela partição líder.

Figura 2.3 – Nível de configuração *Leader Confirmation*.



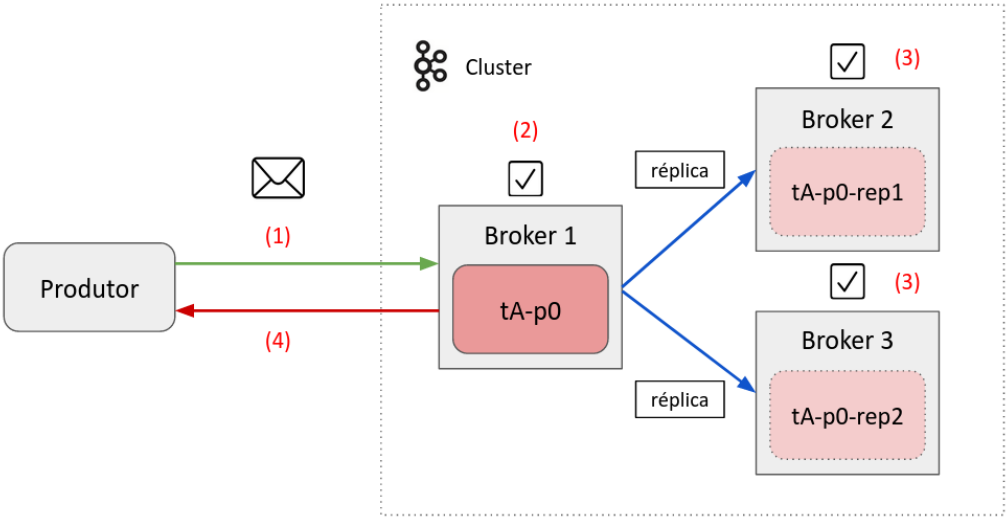
Fonte: Autor.

O terceiro nível de configuração, denominado *Replicas Confirmation* está associ-

ado ao valor `ack = -1`. Com esta configuração, o produtor da mensagem recebe uma confirmação de entrega sempre que, além de persistida na partição líder, a mensagem também foi persistida em todas as réplicas daquela partição. Esta configuração para *ack* é a mais confiável, visto que, para cada envio de mensagem, há uma confirmação sempre que a mensagem está persistida em diferentes *brokers*. Assim, mesmo que a partição líder apresente uma falha, uma nova partição é eleita líder sem perdas.

A Figura 2.4 apresenta uma produção de mensagem com nível de configuração *Replicas Confirmation*. Como observado anteriormente, com este nível de configuração, uma confirmação de entrega é retornada ao produtor sempre que a mensagem está presente na partição líder do tópico destino e já foi replicada. Na Figura, é possível observar que a etapa 1 consiste na transmissão da mensagem pelo produtor ao tópico A (tA), sendo recebida pela partição líder p0 (tA-p0), presente no *broker* 1. A mensagem transmitida está representada pela seta na cor verde. Uma vez recebida pela partição líder, é verificado o nível de configuração do mecanismo de *ack* exigido pela mensagem (etapa 2). Ao constatar que é uma produção com nível *Replicas Confirmation*, o *broker* 1 aguarda a conclusão da replicação da mensagem nas partições réplicas de p0, sendo tA-p0-rep1 presente no *broker* 2 e tA-p0-rep2 presente no *broker* 3. A replicação da mensagem está representada nas setas azuis em direção ao *broker* 2 e *broker* 3. Uma vez concluída a replicação da mensagem (etapa 3), o *broker* 1 retorna uma confirmação de entrega ao produtor, representada pela seta vermelha (etapa 4). Ao final da etapa 4, o produtor pode dar início à transmissão de novas mensagens.

Figura 2.4 – Nível de configuração *Replicas Confirmation* no Kafka.



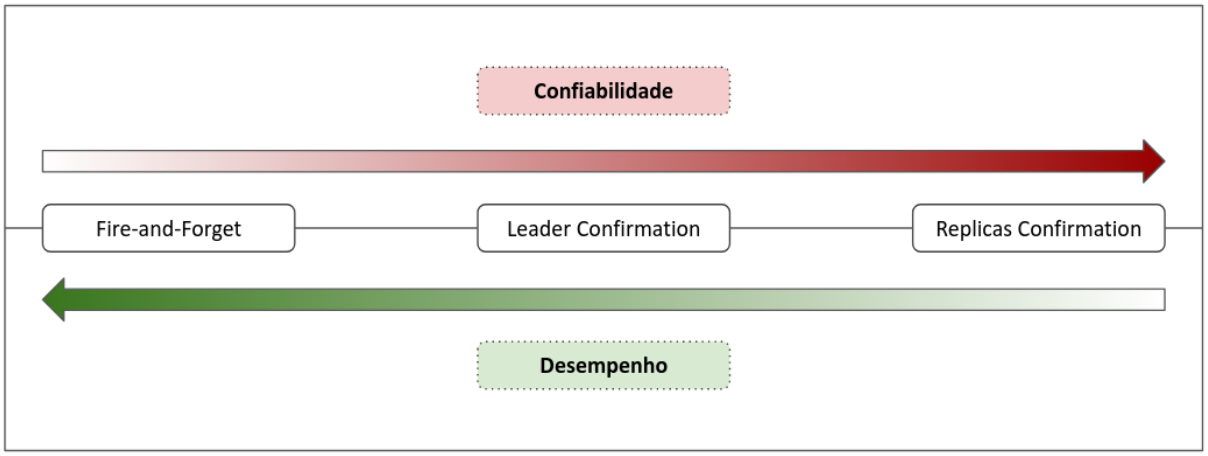
Fonte: Autor.

Conforme exibido nas Figuras 2.2, 2.3 e 2.4, observa-se que os níveis de configuração padrão para *ack* apresentam etapas distintas no processo de produção de mensagem. Enquanto o nível *Fire-and-Forget* apresenta apenas a etapa de transmissão, *Leader*

Confirmation e *Replicas Confirmation* apresentam etapas associadas a exigências de confiabilidade impostas pelas restrições de implementação de cada nível.

Observando os níveis de configuração padrão para *ack* no Kafka, identifica-se também que o custo envolvido para agregar confiabilidade a transmissões de mensagens implica em uma redução no desempenho produtores, ou seja, na sua vazão (GOODHOPE et al., 2012). Isto ocorre pois conforme aumenta-se gradativamente a confiabilidade desejada em *ack*, mais etapas são exigidas aos produtores para garantir que a mensagem chegue com sucesso a uma das partições líderes do tópico. A Figura 2.5 demonstra um comparativo entre a confiabilidade e o desempenho dos níveis de configuração de *acks* no Kafka.

Figura 2.5 – Confiabilidade versus desempenho dos níveis de configuração de *acks* no Kafka.



Fonte: Autor.

Conforme exibido na Figura 2.5, *Fire-and-Forget* é o nível de configuração que apresenta o melhor desempenho. Isto ocorre devido ao fato de que produções de mensagens com este nível não exigem nenhuma confirmação de entrega. Sendo assim, o intervalo de envio entre uma mensagem e outra é o menor possível. Em contraste, *Fire-and-Forget* é o nível com a menor confiabilidade, visto que a ausência de confirmações de entrega impossibilita a recuperação de mensagem em caso de perda. Durante a fase experimental deste estudo, o comportamento esperado do nível *Fire-and-Forget* se confirmou. Este nível apresentou o melhor desempenho dentre todos os níveis testados entretanto também demonstrou que é suscetível a perda de mensagens.

O nível *Leader Confirmation* apresenta desempenho e confiabilidade intermediários. Este nível possui confirmação de entrega, portanto, é possível realizar retransmissões em caso de perda de mensagens. Entretanto, o fato de intercalar confirmações entre transmissões de mensagens degrada o seu desempenho, principalmente quando comparado com a configuração *Fire-and-Forget*. No que se refere à confiabilidade, *Leader*

Confirmation está sujeita a perdas de mensagens sempre que uma partição líder apresenta falha antes de replicar uma mensagem. Na fase experimental deste estudo, não foram observadas perdas de mensagens com o nível *Leader Confirmation* porém, este nível demonstrou-se suscetível a duplicações de mensagens.

Por fim, ao utilizar a configuração *Replicas Confirmation*, produtores recebem confirmação sempre que a mensagem tiver sido replicada em diversos *brokers*. Esta configuração apresenta a maior confiabilidade dentre as três disponíveis no Kafka. Entretanto, o desempenho de *Replicas Confirmation* tende a ser o pior, visto que o recebimento de uma confirmação de entrega somente ocorre quando a mensagem foi persistida em diferentes *brokers*, o que ocasiona degradação no desempenho de produtores. Ao final dos experimentos conduzidos, observou-se que o nível *Replicas Confirmation* apresentou o pior desempenho dentre os níveis padrões e, apesar de também não terem sido observadas perdas de mensagens, este nível também demonstrou duplicações das mesmas.

No âmbito de produtores de mensagens Kafka, há uma relação entre o desempenho do produtor e o mecanismo de *ack*. Esta relação diz respeito ao comportamento efetuado pelo produtor mediante confirmações de entrega de mensagens, já que de acordo com a confiabilidade escolhida, mais etapas são empregues na espera por confirmações de entrega. Como consequência, o mecanismo de Reconhecimento Positivo pode influenciar no desempenho de produtores afetando a métrica de vazão. A vazão é uma das principais métricas aplicadas na avaliação de desempenho de produtores Kafka (ZHANG, 2015) (DOBBELAERE; ESMAILI, 2017). A vazão mede a quantidade de mensagens enviadas em um intervalo de tempo.

O Reconhecimento Positivo, assim como o Fator de Replicação correspondem a mecanismos de tolerância a falhas implementados pelo Kafka. Apesar da existência destes mecanismos, o Kafka utiliza inerentemente diversos mecanismos de tolerância a falhas da ferramenta Apache Zookeeper, sendo estes mecanismos essenciais à operação do Kafka. Enquanto os mecanismos específicos do Kafka abordam diretamente a confiabilidade de transmissões e a disponibilidade de mensagens, os mecanismos do Zookeeper se concentram na gestão e manutenção do *cluster* Kafka, conforme descrito na Seção 2.3.

2.3 RELAÇÃO ENTRE APACHE KAFKA E APACHE ZOOKEEPER

Como os *brokers* Kafka que compõe um *cluster* são instâncias independentes entre si, faz-se necessário recorrer externo a um recurso que possibilite a manutenção e a coordenação dos *brokers*. Para tanto, o Kafka utiliza o Apache Zookeeper.

O Zookeeper é uma plataforma comumente associada ao suporte, coordenação e sincronismo de outros sistemas distribuídos, como o Kafka. Para realizar este suporte, estão presentes no Zookeeper diversos algoritmos de eleição de líder voltados à gestão

de liderança, assim como recursos voltados para a observação da operacionalidade de *brokers*, como o mecanismo de *heartbeats*. O Zookeeper também provê uma estrutura de dados em árvore aplicada na organização do espaço de nomes (*namespaces*).

A definição de qual *broker* Kafka será o controlador, ou seja, o responsável por gerir as relações de liderança entre as réplicas de partição de um tópico, ocorre por meio dos algoritmos de eleição de líder do Zookeeper. Sendo assim, a gestão de liderança no Kafka acontece em duas etapas. Primeiramente, o Zookeeper auxilia na eleição do *broker* controlador e, após isso, é função deste controlador estabelecer as relações de liderança entre as réplicas de partição.

Para eleger um controlador, o Zookeeper inicialmente verifica a operacionalidade cada um dos *brokers* através de mensagens *heartbeat* (JUNQUEIRA; REED, 2013). As mensagens *heartbeat* consistem em avisos periódicos enviados pelos *brokers* ao Zookeeper notificando-o de seu estado atual. Caso um *broker* pare de enviar mensagens *heartbeat* por um período preestabelecido, o mesmo é marcado como inoperante e deixa de receber requisições de leitura e escrita. Caso o *broker* inoperante seja o controlador, uma nova eleição de líder é disparada.

A arquitetura do Zookeeper apresenta ainda uma estrutura de dados em forma de árvore onde cada nó é denominado *Znode*. Cada nó apresenta um nome o qual também é seu caminho dentro da árvore. Além disso, cada *Znode* pode ou não armazenar algum conteúdo. É por meio da árvore de *Znodes* que o Kafka centraliza e gerencia os metadados de cada um dos tópicos presentes em um *cluster*. Deste modo, os metadados dos tópicos estão presentes em *Znodes* específicos de cada tópico, sendo possível acessá-los quando necessário.

Outro recurso do Zookeeper utilizado pelo Kafka são os *watches*. *Watches* são ações de gatilho único que geram notificações sempre que um determinado evento é realizado em algum *Znode* sob monitoramento (JUNQUEIRA; REED, 2013). Dentre os eventos que podem ser monitorados destacam-se a adição e remoção de Znodes, a alteração do conteúdo de um *Znode* e a adição de um *Znode* filho. Para receber uma notificação, um cliente Zookeeper deve registrar um *watch* no caminho do *Znode* alvo. Após a ocorrência do evento e gerenciamento da notificação, o registro do *watch* deve ser repetido caso o monitoramento ainda seja necessário. O Kafka utiliza o recurso de *watches* sempre que notificações devem ser enviadas aos *brokers* quando há alterações nos metadados de tópicos.

2.4 TRABALHOS RELACIONADOS

A literatura dispõe de diversos trabalhos que apresentam como principal tema a garantia de entrega durante transmissões de dados. Tais trabalhos têm como proposta

apresentar mecanismos ou protocolos de comunicação voltados para a confiabilidade em transmissões, seja de mensagens ou pacotes (por meio de redes de computadores). Outros trabalhos apresentam análise da confiabilidade de mecanismos de garantia de entrega de ferramentas já existentes, como no caso do Apache Kafka e o seu mecanismo de *acks*.

Dentre os trabalhos identificados como correlatos, apenas o trabalho de Wu, Shang e Wolter (2019) destina-se a análise do mecanismo de *acks* do próprio Kafka. Os autores apresentam a ferramenta TRAK (*Testing the Reliability of Apache Kafka*), utilizada para avaliar a garantia de entrega do mecanismo de *ack*. Com esta ferramenta, os autores criam um ambiente de testes com contêineres Docker e introduzem falhas na rede durante a transmissão de mensagens. A análise da confiabilidade ocorre através da métrica de taxa de entrega de mensagens, a qual também é aplicada na metodologia do presente estudo. Os autores concluem que, em cenários de transmissões de mensagens com redes sujeitas a perda de pacotes ou atraso na entrega de pacote, os níveis de *acks* estão sujeitos a perdas ou duplicações de mensagens.

Em Bhimani e Panchal (2018), os autores apresentam uma proposta de implementação do protocolo *Advanced Message Queuing Protocol* (AMQP) à transmissões de mensagens focadas para dispositivos de *Internet of Things* (IoT). A proposta exibe a adição de dois novos recursos ao protocolo. O primeiro recurso é destinado a situações em que um cliente é desconectado inesperadamente. Neste recurso, o *broker* tende a armazenar todas as mensagens que seriam destinadas especificadamente ao cliente desconectado, encaminhando-as aos clientes inscritos sob o mesmo tópico do cliente falho. Após isso, o *broker* também envia uma mensagens aos clientes operantes para notificá-los que há um cliente que perdeu sua conexão. No segundo recurso, clientes receberão uma atualização imediata quando se inscreverem em um determinado tópico. Este recurso tem por finalidade evitar que clientes recém conectados demorem longos intervalos de tempo até que recebam a sua primeira mensagem.

O trabalho de Lee et al. (2013) realiza uma análise da correlação entre os níveis de QoS (*Quality of Service*) do protocolo MQTT em redes de transmissão reais por meio da internet. Para conduzir os experimentos, os autores realizaram transmissões de mensagens com dois tipos de clientes. O primeiro tipo eram estações de trabalho que transmitiam mensagens a *brokers* por meio da internet em conexões cabeadas. Já o segundo, eram dispositivos móveis realizando transmissões ao *broker* por meio da internet em uma conexão 3g. Durante as transmissões de mensagens, pacotes eram coletados em um intervalo de 5 minutos com o objetivo de analisar a confiabilidade de cada nível de QoS do protocolo. Desta forma, os autores conseguiram analisar ordem de entrega pacotes, quantidade de retransmissões efetuadas pelo protocolo e, por fim, analisar a quantidade de pacotes perdidos em cenários de transmissões reais. Como conclusão do trabalho, foi observado que para todos os níveis de QoS do protocolo MQTT, o tamanho do pacote transmitido na rede influencia diretamente no atraso de entrega assim como na probabilidade de perda.

Em Luzuriaga et al. (2014) um teste do protocolo de mensageria AMQP é apresentado. Para realizar os testes, uma rede WiFi foi criada para que um produtor de mensagens realizasse transmissões a um *broker*. Ainda, na mesma máquina do *broker* havia um consumidor com o objetivo de coletar as mensagens transmitidas pelo produtor. Cada mensagem consumida é registrada em *log* juntamente com seu tempo de envio e o tempo de recepção. Deste modo, sempre que o canal de transmissão do produtor sofre alteração, gera-se inconsistências que permitem a observação de perdas de mensagens ou entregas fora de ordem. Ao realizarem as análises, os autores concluíram que o protocolo AMQP é robusto e confiável a ponto de ser capaz de entregar todas mensagens. Entretanto, há uma relação entre o tamanho em bytes da mensagem enviada e a quantidade de mensagens perdidas quando há mudanças repentinas no canal de transmissão do produtor. Esta correlação é devido ao fato de que ao trocar o canal de transmissão, o produtor armazena mas mensagens em um *buffer* para transmiti-las até que o novo canal de transmissão seja estabelecido. Perdas ocorrem quando a capacidade do *buffer* não é suficiente para comportar todas as mensagens, obrigando o produtor a descartar mensagens quando o *buffer* se encontra sem espaço.

Luzuriaga et al. (2015) apresentam em seu trabalho uma análise experimental do comportamento dos protocolos de mensageria AMQP e *Message Queuing Telemetry Transport* (MQTT) em redes móveis e instáveis visto que ambos os protocolos ainda não foram testados nesses ambientes. Para realizar a análise, um produtor de mensagens foi submetido a transmissões em uma rede com dois canais de transmissão e, forçam o produtor a trocar de canal durante a transmissões de mensagens. Desta forma, foi possível observar como a transição de um canal para outro pode impactar nas transmissões. Observou-se durante os experimentos que a transição entre canais afeta a ordem de entrega no protocolo AMQP, porém isso não ocorre no protocolo MQTT. Ainda, os autores também observaram que a transição entre os canais de transmissão tende a não gerar perdas de mensagens, já que ambos os protocolos inserem as mensagens pendentes de transmissão em um *buffer* pare transmiti-las quando o canal de transmissão já está definido. Entretanto, caso o número de mensagens pendente de transmissão seja superior ao suportado pelo *buffer*, podem ocorrer perdas de mensagens em ambos os protocolos.

O trabalho de Marinov, Nenova e Iliev (2019) apresenta um protocolo de mensageria para *Veicular Ad Hoc Network* (VANET), ou seja, um protocolo de mensageria especializado em comunicação veicular. A criação deste novo protocolo é motivada pela demanda de um protocolo veicular capaz de diminuir acidentes em estradas. Para tal, o protocolo apresentado especializa-se em confiabilidade onde a perda de mensagens é evitada através de criações de novas rotas de entrega de mensagens entre veículos. As rotas de transmissão confiáveis são definidas de acordo com o tempo de conexão existente entre dois carros. Desta forma, garante-se que a mensagem será transmitida pela rota que demonstre menor interferências na rede de comunicação.

Em Sundarasekar et al. (2019), os autores apresentam um algoritmo de transferência de dados confiável para redes de sensores aquáticos com sinal acústico. O principal objetivo dos autores é apresentar um algoritmo com a capacidade de manter a conexão entre os sensores saudável por meio do gerenciamento inteligente da energia dos senhores. Para isso, o algoritmo utiliza técnicas de aprendizado de máquina para transferir dados entre sensores de forma eficiente e com o menor custo energético possível, aumentando assim a vida útil dos sensores e, consequentemente, mantendo a conexão entre os sensores operante por mais tempo. Por fim, o algoritmo apresentado atingiu o objetivo proposto pelos autores ao mitigar colisão de dados entre sensores. Evitando colisão de dados diminuí-se as sobrecargas de processamento aplicadas em retransmissões, reduzindo o gasto excessivo de energia.

Dos trabalhos mencionados foram identificadas funcionalidades a cerca de transmissões confiáveis de mensagens como, por exemplo, o armazenamento de cópias de mensagens visando recuperação. A identificação destas funcionalidades foi útil na implementação da configuração *rAck*, visto que a principal contribuição de *rAck* visa identificar e recuperar mensagens perdidas (BHIMANI; PANCHAL, 2018) (Luzuriaga et al., 2015).

Os trabalhos também demonstraram estruturas de experimentação com transmissões de mensagens Kafka já consolidadas na literatura. Sendo assim, a utilização de Docker e Pumba para simulação de redes instáveis observadas nos trabalhos de Wu, Shang e Wolter (2019) foram de fundamental importância para a execução da fase experimental deste estudo, auxiliando na estruturação de um ambiente de testes. Ainda, as métricas de vazão e taxa de perda de mensagens utilizadas em experimentos de trabalhos anteriores, facilitaram a avaliação da configuração *rAck* em termos de confiabilidade e desempenho (Lee et al., 2013).

Na Tabela 2.1 é possível observar uma associação dos trabalhos identificados como correlatos com o que foi desenvolvido de atividade em cada um deles. Também está presente nesta Tabela a própria configuração *rAck*, o que permite exibir o que há de semelhante entre todos os estudos tabelados.

Inicialmente, Wu, Shang e Wolter (2019) é o único dos trabalhos correlatos focado no mecanismo de *acks* do Kafka. Neste trabalho os autores executam uma análise da confiabilidade de *acks*. Para realizar esta análise, os autores transmitem mensagens à medida que falhas de redes são introduzidas nos canais de transmissão, assim como é feito na experimentação envolvendo a configuração *rAck*. Ainda, ao realizar experimentos com os níveis de configuração *Leader Confirmation* e *Replicas Confirmation*, também há utilização do recurso de *Store-and-Foward* destes níveis. Por sua vez, o recurso de *Store-and-Foward* também está presente no monitor da configuração *rAck*.

Bhimani e Panchal (2018) apresentam apenas um incremento de recursos ao protocolo AMQP, não realizando experimentações com falhas de rede ou apresentando análises de desempenho ou confiabilidade. Entretanto, um dos recursos adicionado pelo autores

Tabela 2.1 – Relação entre trabalhos correlatos e a Configuração rAck

	Trabalho realizado pelos autores				
Referência	Experimentação com falhas de rede	Mecanismo de Store-and-Forward	Análise de Desempenho	Análise de Confiabilidade	Análise de Consumo Energético
Wu, Shang e Wolter (2019)	X	X		X	
Bhimani e Panchal (2018)		X			
Lee et al. (2013)			X	X	
Luzuriaga et al. (2014)	X		X	X	
Luzuriaga et al. (2015)	X		X	X	
Marinov, Nenova e Iliev (2019)	X	X	X		
Sundarasekar et al. (2019)		X	X		X
Configuração rAck	**X**	**X**	**X**	**X**	

Fonte: Autor.

utiliza o princípio de *Store-and-Foward*, também aplicado na configuração *rAck*. No incremento de Bhinami e Panchal, o *broker* segura uma mensagem antes de repassá-la aos clientes. Caso o cliente destino se encontre desconectado, a mensagem é encaminhada para outro cliente inscrito sob o mesmo tópico.

Na análise dos níveis de QoS do protocolo MQTT, efetuada em Lee et al. (2013), os autores avaliam o desempenho e a confiabilidade do protocolo em transmissões reais na internet, sem introduzir falhas de rede intencionalmente. A análise do desempenho ocorre por meio da latência, medindo o tempo necessário para as mensagens chegarem ao destino. Já a confiabilidade é medida por uma taxa de entrega de pacotes. No trabalho envolvendo a configuração *rAck*, também há avaliações de desempenho e confiabilidade. Entretanto, na configuração *rAck* o desempenho é avaliado pela vazão e tempo de execução, enquanto a confiabilidade é medida pela taxa de perda de mensagens.

Assim como ocorre com a experimentação envolvendo *rAck*, em Luzuriaga et al. (2014) os autores realizam uma experimentação com o protocolo AMQP que envolve uma introdução de falha de rede durante transmissões. A falha de rede obriga o transmissor a trocar de canal durante as transmissões. Esta troca de canal permite avaliar a confiabilidade e o desempenho do protocolo. Já em Luzuriaga et al. (2015) os mesmos autores incrementam o estudo inicial, estendendo a experimentação ao protocolo MQTT e comparando ambos em termos de confiabilidade e desempenho.

Em Marinov, Nenova e Iliev (2019) os autores avaliam a eficiência do protocolo implementado por meio de simulações de redes entre carros. As falhas de rede entre os carros é introduzida à medida em que há uma rede de transmissão entre carros que estão em direções opostas, obrigando o protocolo a procurar a melhor rota. Assim como há no monitor *rAck*, o protocolo de VANET também possui o recurso de *Store-and-Foward*, visto que as transmissões ocorrem com base no protocolo TCP, o qual traz este recurso como

padrão.

Por fim, o trabalho de Sundarasekar et al. (2019) associa-se ao trabalho da configuração *rAck* pois, além de também possuir o mecanismo de *Store-and-Foward* entre os sensores acústicos, apresenta uma analise experimental do desempenho da rede construída para transmissão de dados em redes aquáticas. Como diferencial, neste trabalho os autores avaliam o consumo energético dos sensores ao realizar transmissões, visto que manter os sensores operantes pelo maior tempo necessário sem desperdiçar energia em retransmissões desnecessárias é uma preocupação do trabalho.

3 A CONFIGURAÇÃO RACK

A configuração *rAck* atua por meio da coordenação e sincronismo de três módulos visando agregar confiabilidade a transmissões de mensagens sem reconhecimento positivo. O fluxo completo de transmissão, monitoramento e retransmissão de mensagens está dividido em tarefas independentes. Apesar de cada módulo apresentar tarefas específicas, estas se complementam de forma a permitir que, mesmo sem reconhecimento positivo, seja possível garantir a entrega de mensagens. Antes de descrever as funcionalidades específicas de cada módulo, será apresentada uma visão geral da configuração Reliable Ack (*rAck*).

A Figura 3.1 ilustra o fluxo de comunicação entre os três módulos que compõem a configuração *rAck*. Por meio de etapas, caracterizadas pelos números em vermelho, está descrito o caminho que a mensagem percorre a partir do momento em que é transmitida pelo produtor *rAck*. Em um momento prévio as transmissões de mensagens, é atribuído que o produtor Kafka fará transmissões com a configuração *rAck*. Uma vez realizada essa atribuição, o produtor Kafka convencional se torna um produtor *rAck*.

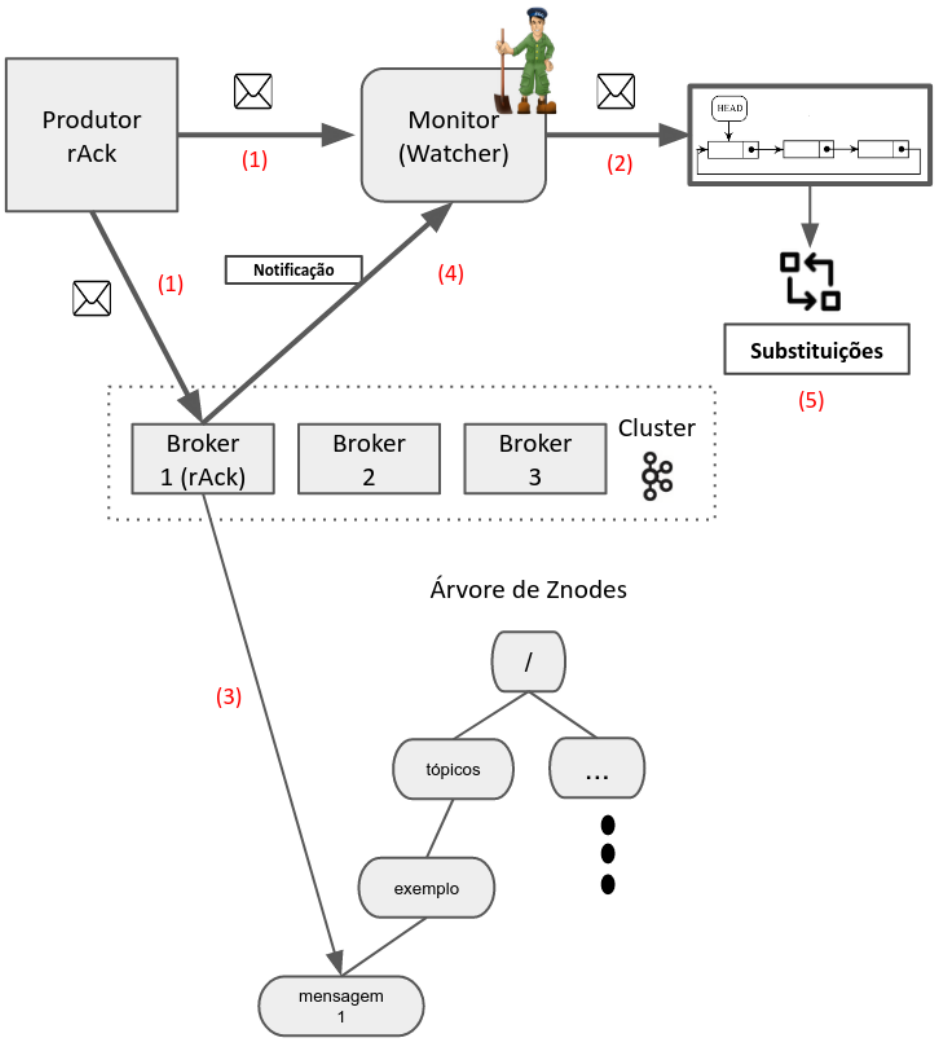
Na sequência, a primeira etapa consiste na transmissão da mensagem aos *brokers rAck* por parte do produtor *rAck*. Paralelamente, uma cópia da mensagem é enviada ao módulo monitor que a armazena internamente em uma lista circular, visando recuperações futuras (etapa 2).

Ao receber uma mensagem, o *broker rAck* (representado na Figura 3.1 pelo *broker 1*) identifica qual o *ack* exigido para a mesma. Caso seja uma transmissão com a configuração *rAck*, o próprio *broker rAck* cria um *Znode* na árvore do Zookeeper, sinalizando que a mensagem foi recebida com sucesso (etapa 3).

Por sua vez, antes de existir qualquer transmissão, monitor registra um *watch* sob o *Znode* que armazena os metadados do tópico que receberá mensagens (representado na figura pelo *Znode* “exemplo”). Assim, a criação de qualquer *Znode* subsequente ao Znode “exemplo” gera uma notificação. Esta notificação é recuperada pelo monitor devido ao *watch* previamente registrado (etapa 4). Após isso, o monitor obtém o conteúdo do *Znode* recém criado, ou seja, o *Znode* “mensagem 1”, identificando qual mensagem foi recebida com sucesso por meio da notificação recebida. Neste exemplo, receber com sucesso a primeira mensagem resultou na criação do Znode nomeado “mensagem 1”. Por fim, ao identificar quais mensagens foram recebidas com sucesso, o monitor recorre à lista circular para registrar que as mensagens cópias armazenadas nos nós foram recebidas e podem ser eventualmente substituídas por novas mensagens cópias (etapa 5).

O detalhamento de cada módulo sintetizado na Figura 3.1 será apresentado no restante deste capítulo. A Seção 3.1 apresenta o funcionamento do módulo Produtor rAck, um produtor Kafka capaz de utilizar a configuração *rAck*. A Seção 3.2 descreve o *broker*

Figura 3.1 – Configuração Reliable Ack (*rAck*).



Fonte: Autor.

rAck, um *broker* Kafka adaptado para viabilizar a produção de mensagens via *rAck*. Por fim, a Seção 3.3 destaca o funcionamento do módulo monitor, o qual é responsável por gerenciar a árvore de *Znodes* do Zookeeper, além de armazenar cópias das mensagens enviadas aos *brokers rAck*.

3.1 PRODUTOR RACK

A ciência de que conforme aumenta-se a confiabilidade do mecanismo de *acks* em produtores Kafka degrada-se o desempenho incitou a implementação de uma nova configuração de *ack* que busque aliar a confiabilidade ao desempenho. Todavia, implementar um novo nível de configuração de *ack* que concilie tais atributos é um desafio, visto que

atualmente o mecanismo de configuração de *acks* do Kafka implica na utilização de confirmações de entrega para as mensagens e tais confirmações prejudicam o desempenho de produtores.

A configuração *rAck* foi estruturada para não fazer uso de configurações de entrega no lado produtor, evitando assim ociosidades durante transmissões. Para utilizar a configuração *rAck* o produtor necessita estar com a variável `acks_config` estipulada como "rAck", assim o mesmo transforma-se no Produtor *rAck* e consequentemente acaba acionando os recursos da configuração *rAck* como um todo, ou seja, o *broker rAck* e o monitor. Após a configuração, as mensagens são enviadas pelo produtor *rAck* de forma idêntica aos demais níveis *ack*, ou seja, através dos métodos da classe que implementa um produtor Kafka. Por meio destas transmissões iniciadas pelo produtor *rAck* que o recurso de *watches* do Zookeeper é acionado no lado monitor.

Além do conteúdo da mensagem propriamente dito, uma mensagem enviada por um produtor *rAck* contém as seguintes informações: origem (identificação do produtor), destino (tópico em que será armazenada a mensagem) e id (identificação da mensagem). Uma vez identificada a transmissão da mensagem com a configuração *rAck*, o produtor cria uma cópia da mensagem para ser enviada ao módulo monitor. O objetivo da criação desta cópia da mensagem é prover recuperação caso o monitor identifique a mesma foi perdida.

O Código 1 exibe o processo de envio de mensagens por parte de um produtor *rAck*. O produtor *rAck* deve ser inicializado (linha 1) e necessita receber configurações fundamentais como o endereço dos *brokers* Kafka (linha 2) e o nível de configuração de *ack* a ser aplicado nas transmissões (linha 3). Além disso, é necessário inicializar um *Socket* que envia cópias das mensagens ao monitor (linha 4). Cada mensagem a ser enviada deve ser inicializada em uma classe do tipo `Record` com os parâmetros `topic`, `id` e `texto` (linha 6). Antes de enviar uma mensagem, o produtor verifica se a configuração para *acks* está definida em *rAck* (linha 7). Confirmando que *ack* está definido em *rAck*, o produtor cria uma cópia da mensagem, parametrizada com a origem (`idProducer`), o destino (`topic`), a identificação (`id`) e o respectivo conteúdo (linha 8). Após inicializada a mensagem, sua cópia é transmitida ao módulo monitor via *Socket* (linha 9). Na sequência, o produtor envia a mensagem aos *brokers* e limpa o *buffer* de mensagens (linhas 10 e 11).

3.2 BROKER RACK

O segundo módulo da configuração *rAck* é composto por um *broker* Kafka cujo código fonte foi modificado visando adequações que permitissem receber mensagens de um produtor *rAck*. Este módulo foi denominado *broker rAck*.

Quando recebe uma mensagem, o *broker rAck* identifica o nível de configuração

Código 1 Envio de mensagem: Produtor rAck

input: brokersAddress, topic, text, monitorAddress, monitorPort
output: Mensagem enviada com ack == rAck

```
1: Producer p = newProducer()
2: p.config(servers, brokersAddress)
3: p.config(ack, "rAck")
4: Socket socketMonitor = newSocket(monitorAddress, monitorPort)
5: for Record r toSend do
6:     Record r = new Record(topic, id, text)
7:     if p.getConfig(ack) == rAck then
8:         Backup b = new Backup(idProducer, topic, id, r.getText())
9:         socketMonitor.send(b)
10:    p.send(r)
11:    p.flush()
```

de *ack* definido para a mensagem. Caso seja identificado que a mensagem foi gerada com a configuração *rAck*, um *Znode* é adicionado à arvore do Zookeeper. Este *Znode* é adicionado como filho do *Znode* que armazena os metadados do tópico destino. O conteúdo deste *Znode* consiste nas informações de origem (identificação do produtor), destino (tópico) e id (identificação da mensagem). Em cada *broker rAck* existe um cliente do Zookeeper o qual permite efetuar modificações na árvore de Znodes. Por meio deste cliente, o *broker rAck* pode remover, adicionar e modificar o conteúdo de cada Znode existente na árvore.

Durante as primeiras observações do funcionamento do *broker rAck* observou-se que os Znodes adicionados à árvore do Zookeeper não eram capazes de expirar após determinado intervalo de tempo. Deste modo, cada Znode criado pela configuração *rAck* apenas poderia ser removido da árvore com a execução de um comando específico para remoção de Znodes. A execução deste comando somente poderia ser efetuada por outro cliente Zookeeper.

Com o desenvolvimento de versões iniciais do *broker rAck*, identificou-se que em transmissões de mais de 100.000 mensagens, o mesmo número de Znodes ficavam permanentemente na árvore. Neste sentido, a permanência dos Znodes na árvore é vista como problemática, visto que cada Znode só é necessário até que o monitor consiga recuperar o seu conteúdo. Ainda, cada um destes Znodes consumia recursos computacionais que eram destinados ao Zookeeper como um todo, fazendo com que uma grande quantidade de Znodes pudesse utilizar muitos recursos das *Java Virtual Machines* (JVM) de cada instância do Zookeeper.

Para evitar utilização indevida de recursos, decidiu-se utilizar a versão 3.6.0 do Zookeeper na implementação da configuração *rAck* pois nessa versão é possível definir um TTL *time to live* aos Znodes. Com o TTL definido, um Znode expirará desde que esteja de acordo com duas restrições. A primeira restrição exige que o Znode com TTL não possa

ter seu conteúdo alterado em um tempo inferior ao tempo de TTL. Já a segunda restrição impõe que o Znode com TTL não possa ter filhos. Uma vez utilizando o recurso de TTLs, cada Znode criado pelo *broker rAck* permanece na árvore apenas pelo tempo necessário para que o monitor possa recuperar o seu conteúdo, permitindo então a identificação da mensagem que foi recebida.

Inicialmente, o tempo de vida de cada Znode foi estipulado como 30 segundos. Entretanto, ao realizar testes da configuração *rAck*, foi possível observar que o tempo era muito pequeno para ser utilizado em transmissões de grandes quantidades de mensagens. Isto ocorre pois as notificações que chegam ao módulo monitor são processadas em ordem de chegada. Assim, dependendo da quantidade de notificações, é possível que ao processar as últimas notificações, os respectivos Znodes já tenham expirado. A expiração precoce impossibilita que o monitor consiga recuperar o conteúdo do Znode e que o mesmo consiga identificar qual a mensagem que chegou com sucesso ao destino. Para solucionar essa questão, o TTL dos Znodes foi aumentado para 600 segundos (10 minutos). Os 600 segundos de TTL evitaram que ao processar uma notificação, o monitor gerasse uma exceção em sua execução por buscar um Znode inexistente.

A partir do momento em que o *broker rAck* recebe uma mensagem, o processamento a ser efetuado tende a variar de acordo com a configuração de *ack* estipulada para a mensagem. Com isso, após receber uma mensagem, a primeira tarefa do *broker rAck* é obter a configuração para *ack* prevista. Essa observação é realizada extraindo os metadados associados a mesma. Caso o *ack* associado seja um nível padrão do Kafka, o *broker rAck* atua como um *broker* Kafka convencional, executando as operações específicas de cada configuração padrão.

Caso a configuração seja *Fire-and-Forget*, o *broker* observa que não é necessário retornar confirmação de entrega ao produtor, iniciando o processo de adicionar a mensagem ao arquivo *append-only* da partição líder. Se for *Leader Confirmation*, antes de adicionar a mensagem ao arquivo da partição, o *broker* retorna uma confirmação de entrega ao produtor e somente após isso, realiza a persistência da mensagem. Por fim, se a configuração for *Replicas confirmation*, o *broker* realiza a persistência da mensagem disparando automaticamente a replicação da mesma entre as partições réplicas. Apenas após concluída a replicação que é retornada uma confirmação de entrega ao produtor.

No que se refere ao processamento específico do *broker rAck* ao nível “rAck”, o comportamento assemelha-se ao nível *Fire-and-Forget*, não sendo necessário retornar confirmação de entrega. Deste modo, antes de adicionar a mensagem ao arquivo da partição, o *broker rAck* recorre ao cliente Zookeeper para criar o Znode específico da mensagem. Após a criação do Znode, a mensagem é incrementada no arquivo da partição que deve recebê-la.

O Código 2 exibe o trecho utilizado pelo *broker rAck* para efetuar a criação de *Znodes* conforme as mensagens são recebidas. Destaca-se que o código apresenta apenas

o processamento específico da adaptação para receber rAck. Conforme mostra o Código 2, a entrada é uma requisição de produção de mensagem (forma como *brokers* Kafka recebem mensagens) e um cliente administrador do Zookeeper (presente por padrão nos serviços do Kafka). A primeira tarefa do *broker rAck* é filtrar do corpo da requisição recebida a abstração que encapsula uma mensagem, transformando o corpo da requisição em uma variável `ProduceRequest` (linha 1).

Código 2 Recepção da mensagem: *broker rAck*

input: request, adminZk
output: Adição de novo *Znode*

1: $ProduceRequest\ p = request.getBody()$
2: **if** $p.acks() == rAck$ **then**
3: $\quad String\ data = p.getOrigem() + \text{“;”} + p.getDestino() + \text{“;”} + p.getId()$
4: $\quad String\ newZnodePath = \text{“}/brokers/topics/\text{”} + p.getDestino() + \text{“}/produce-\text{”}$
5: $\quad adminZk.create(newZnodePath, data, SEQUENTIAL\_WITH\_TTL, 600)$

Após isso, o *broker rAck* verifica o nível de configuração de *ack* previsto na requisição (linha 2). Ao identificar que a mensagem exige configuração em *rAck*, o *broker* gera uma *string* que armazena a informação a ser inserida no *Znode* a ser criado (linha 3). A *string* então recebe a origem, destino e id da mensagem. Na sequência, é criada a *string* que define o caminho na árvore onde o novo *Znode* será criado (linha 4). Essa *string* é estruturada de forma que o novo Znode seja adicionado como filho do Znode do tópico destino.

Por fim, o *Znode* é criado pelo cliente do Zookeeper existente no *broker rAck* Kafka (linha 5). Os parâmetros da criação do Znode são o caminho na árvore, a informação a ser armazenada, a identificação de *Znode* com tempo de vida e o respectivo tempo de vida (TTL). Como mencionado anteriormente, o tempo de vida foi definido como sendo 600 segundos, o que faz o *Znode* expirar após 10 minutos.

3.3 O MONITOR

Visto que a ausência de reconhecimento positivo impede que o produtor *rAck* consiga identificar quando uma mensagem não foi entregue com sucesso, o mesmo não consegue realizar a devida recuperação em caso de perda. Entretanto, a inexistência de reconhecimento positivo também evita degradação no desempenho tal como observada em produções de mensagens com os níveis *Leader Confirmation* e *Replicas Confirmation* (DOBBELAERE; ESMAILI, 2017).

Na configuração *rAck*, a atribuição de rastrear o fluxo de transmissão de mensagens ocorre no módulo monitor, o principal responsável por agregar confiabilidade à configuração. Para tal, o monitor executa externamente ao Kafka e ao Zookeeper ao mesmo tempo

em que mensagens estão sendo transmitidas pelo produtor *rAck*. Na árvore do Zookeeper, há um caminho específico em que são criados Znodes para armazenar os metadados de cada tópico presente no *cluster* e eventualmente é este Znode que será monitorado pelo terceiro módulo da configuração *rAck*.

O papel inicial do monitor é inicializar um *watch* no Znode que armazena metadados do tópico que receberá mensagens do produtor *rAck*. Uma vez registrado o *watch*, sempre que uma modificação na estrutura da árvore for realizada, uma notificação será retornada ao monitor. Como na configuração *rAck* há inserção de *Znodes* na árvore, cada inclusão pressupõe a recepção de uma mensagem nos *brokers rAck*. Assim, por meio das notificações, o monitor consegue observar quais mensagens foram recebidas.

O Código 3 exibe o trecho executado pelo monitor para registrar *watches*. Como entrada, o Código recebe o endereço dos serviços do Zookeeper e um *array* para armazenar o conteúdo das mensagens. Na sequência, o monitor inicializa um cliente do Zookeeper (linha 1), responsável pelo gerenciamento dos serviços desta ferramenta. Após a criação do cliente, o monitor inicializa um *Watcher*, classe responsável por efetivar os registros de *watches* em Znodes (linha 2). Para a boa utilização do *Watcher*, deve-se obrigatoriamente sobrescrever o seu método nomeado `ProcessEvent` (linha 5), que é disparado sempre que qualquer ação incide sob o Znode monitorado. Por fim, o monitor realiza o registro do *watch*, conforme observado na linha 3. Esse registro recebe como parâmetro o caminho a ser monitorado, bem como o *watcher* que deve processar os eventos recebidos.

O método `ProcessEvent` executado pelo *watcher* do monitor recebe como parâmetro uma classe do tipo `WatchedEvent`, responsável pela abstração de eventos em Znodes. Sempre que ocorre um evento, o monitor verifica se o mesmo é do tipo “Criação de Znode” (linha 6). Além disso, o monitor também verifica se a criação do *Znode* contém a *string* “produce” no seu caminho, o que indica que a criação do Znode é consequência de uma produção de mensagem da configuração *rAck*. Caso as verificações sejam verdadeiras, o monitor busca o conteúdo do *Znode* e adiciona o conteúdo em um *array* de mensagens recebidas (linhas 8 e 9).

Código 3 Monitor: registro de watch

input: zookeeperURL, listReceived
output: Registro de watch e processamento de notificações

```
1: Zookeeper zk = new Zookeeper(zookeeperURL)
2: Watcher watcher = new Watcher(ProcessEvent)
3: zk.addWatch("/brokers/topics/topicoDestino", watcher)
4:
5: function PROCESSEVENT(WatchedEvent event)
6:     if event.getType() == Event.EventType.NODECREATED then
7:         if event.getPath().contains("produce") then
8:             String data = zk.getData(event.getPath())
9:             listRecived.add(data)
```

As notificações recebidas pelo monitor facilitam a identificação de mensagens recebidas e de mensagens perdidas. Entretanto, o simples discernimento entre mensagens perdidas e não perdidas torna-se dispensável se não há recuperação. Sendo assim, o monitor possui também a funcionalidade de receber uma cópia das mensagens enviadas aos *brokers*. De posse das cópias de mensagens, as notificações concedem a base para que o monitor consiga retransmitir as mensagens sempre que uma perca é identificada.

Visto que uma das principais tarefas do módulo monitor se baseia na gestão das mensagens *backup*, foram pesquisadas formas de manter tais mensagens próximas ao monitor, agilizando sua manipulação. Inicialmente, a primeira solução encontrada para realizar o armazenamento das mensagens era acoplar um banco de dados de armazenamento em memória, permitindo que o monitor consiga consiga acesso rápido às mensagens.

Para implementar tal acoplamento, foi utilizado o Apache Ignite (FOUNDATION, 2015). O Ignite é um banco de dados distribuído implementado para computações de alto desempenho em memória, permitindo a criação de caches para armazenar entradas de dados no formato chave-valor. Tendo em mente que um dos objetivos do Ignite é o desempenho, integrou-se ao monitor uma instância do Ignite. Esta integração permitiu que o monitor, ao ser inicializado, criasse uma nova cache sob a plataforma Ignite para armazenar as mensagens *backup*.

Muito embora o Ignite fosse uma solução pautada em desempenho, sua operação também era baseada na utilização de máquinas virtuais. Deste modo, para utilizar uma cache Ignite, o monitor necessariamente precisava inicializar uma nova JVM para hospedar a cache. Além disso, por ser uma JVM e apresentar recursos limitados (muitas vezes definidos no momento de sua criação), ainda eram necessárias políticas que permitissem a boa gestão dos recursos da JVM.

Visando implementar uma boa gestão dos recursos da cache Ignite, políticas de descarte das mensagens que já foram recebidas pelos *brokers rAck* foram implementadas. A base para realizar os descartes centrou-se em uma lista de chaves que foram recuperadas pelo monitor em suas consultas aos Znodes criados pelo *broker rAck*. Cada chave presente nesta lista, também era a chave que identificava cada mensagem na cache. Deste modo, para remover a mensagem da cache, apenas era necessário executar uma operação que buscava se determinada chave estava presente na cache, removendo-a caso fosse encontrada.

As políticas de descarte foram divididas em descarte por tempo e descarte por quantidade. Para realizar o descarte por quantidade, o monitor aguardava uma certa quantidade de chaves recebidas dos Znodes para inicializar o processo de descarte. Já para o descarte por tempo, o monitor esperava um período predeterminado para começar a remoção das mensagens da cache.

Com a implementação das políticas de descarte, foi possível observar que, mesmo

que o controle da quantidade de mensagens na cache tenha sido alcançado, muito do processamento do monitor era destinado às operações de descarte. Ainda, a utilização da cache Ignite impossibilitava a implementação de políticas que permitissem a identificação das mensagens perdidas durante o período de transmissão do produtor *rAck*, já que o Ignite não provia formas ágeis de mensurar o tempo exato que cada mensagem estava presente na cache.

Deste modo, as mensagens apenas eram retransmitidas quando o monitor identificava que o produtor *rAck* já havia finalizado todas suas transmissões. Ao identificar que não haveria mais produções de mensagens, o monitor retransmitia todas as mensagens que perduravam na cache, concluindo que, uma vez que não foram descartadas previamente, é porque não foram recebidas pelo *broker rAck*. A retransmissão das mensagens apenas no final da produção das mesmas causava a postergação indefinida de transmissão, sendo essa uma restrição imposta pela decisão de adoção à uma cache Ignite.

Para evitar a postergação indefinida na retransmissão, além de todo o processamento do monitor destinado ao descartes, procurou-se formas de implementar uma solução para o armazenamento de mensagens visando o reaproveitamento de recursos computacionais. A alternativa encontrada foi a implementação de uma lista encadeada circular.

Na adaptação para uma lista encadeada circular, cada nó presente na lista é uma nova uma mensagem *backup* transmitida pelo produtor *rAck*. Inicialmente, a lista comporta 15000 mensagens, entretanto esse limite pode aumentar de acordo com a percentagem de mensagens confirmadas de um total de mensagens a serem enviadas pelo produtor. Assim, caso o monitor espere uma quantidade muito grande de mensagens, as notificações contribuem para que o tamanho da lista aumente gradativamente. Desta forma, evita-se que não exista espaço na lista para cópias de mensagens que estão sendo recebidas.

Sempre que recebe uma cópia de mensagem e a lista ainda não atingiu sua capacidade limite, o monitor a adiciona ao final da lista. Ao receber uma notificação, o monitor percorre a lista a procura do nó que armazena a mensagem correspondente à notificação, marcando-o como mensagem recebida. Caso a capacidade da lista esteja esgotada, o monitor busca um nó cuja mensagem já tenha sido previamente recebida, substituindo-a.

Cada nó na lista circular apresenta um atributo denominado “idade". A idade de um nó indica quantas vezes o monitor passou por aquele nó em busca de espaço para uma cópia de mensagem. Sempre que, ao percorrer a lista em busca de espaço, o monitor passe por um nó que não tenha sido confirmado, a idade do mesmo é incrementada. Ao passar pela segunda vez em um nó não confirmado, a respectiva mensagem é retransmitida, cedendo espaço para o novo *backup*. O tempo necessário para passar duas vezes em um mesmo nó indica que não há uma notificação para a mensagem contida naquele nó. Dessa forma, entende-se que a mensagem presente no referido nó foi perdida.

Com a política de retransmissão baseada na idade do nó adotada no armazenamento com lista, substitui-se a retransmissão tardia imposta pelo armazenamento com o

cache Ignite. Essa nova política de retransmissão possibilita que mensagens não sejam retransmitidas em um tempo muito superior ao tempo inicial de sua transmissão inicialmente realizada pelo produtor *rAck*. Além disso, todo o processamento envolvido em descartes de mensagens é desnecessário.

O Código 4 exibe o trecho da inserção de novas mensagens na lista circular. A entrada do Código consiste na porta a ser utilizada pelo *Socket* que receberá as mensagens e na quantidade máxima de itens que a lista permite. O Código começa com as inicializações do *Socket* que receberá as mensagens (linha 1) e da variável contadora de mensagens na lista (linha 2). Após isso, os nós da lista circular necessários para percorrer a lista são inicializados (linhas 3 a 5). O laço responsável por receber as mensagens é introduzido na linha 6. Cada mensagem recebida pelo *Socket* é transformada em uma classe genérica do tipo `Object` (linha 8). Após a recepção, a instância de objeto que abstrai a mensagem é convertida em uma classe do tipo `Record` (linha 9). A classe `Record` é responsável pela abstração de uma mensagem propriamente dita. Após a conversão para `Record`, verifica-se a lotação da lista (linha 10). Caso a lista não esteja esgotada, chama-se um método para a inserção de nó ao final da lista (linha 11). Caso contrário, chama-se um método que tem por objetivo buscar nós que contenham mensagens já confirmadas.

O método `insert()` (linha 15) do Código 4 consiste em uma inserção clássica de nó a uma lista encadeada circular. Se `head` for nulo, a lista ainda está vazia. Assim, cria-se um novo nó com a mensagem (linha 17) e tanto `head` quanto `tail` recebem este nó como referência. Caso `head` não seja nulo (início do `else` na linha 19), primeiramente cria-se um novo nó temporário. Após isso a próxima instrução faz com que `tail` aponte para o nó temporário (linha 21). Como o nó temporário deve ser o novo `tail`, faz-se então esse novo nó apontar para `head` (linha 22) e o mesmo passa a ser `tail` (linha 23). Ao final do método, incrementa-se a quantidade de mensagens na lista (linha 24).

Diferentemente do método `insert()`, o método `insertAfterFull()` (linha 26) necessita percorrer a lista inteira em busca de um nó passível de substituição, visto que neste caso ao lista encontra-se esgotada. Na linha 30, tem-se início um laço para percorrer a lista. Este laço inicia a busca na lista a partir do nó `current` (atual), o qual recebe o nó `startPointer` como referência. O nó `startPointer` representa o ponto de parada da última busca efetuada na lista, logo, se for a primeira busca, `startPointer` é nulo e deve receber `head` (linha 28).

Se o nó `current` já tiver sido lido, ou seja, a mensagem contida nele já tiver sido confirmada, a mesma é substituída, finalizando a inserção (linhas 31 a 33). Caso contrário, o laço continua com uma verificação se o nó atual já foi visitado anteriormente, ou seja, se é a segunda vez que este nó está sendo verificado para eventuais substituições (linha 35). Em caso afirmativo, a mensagem contida no nó verificado é retransmitida e substituída pela nova mensagem, finalizando o laço (linhas 36 a 38). Se o nó nunca foi visitado, então a idade é incrementada (linha 39). Ao final da inserção, o nó *startPointer* é atualizado com

Código 4 Monitor: recebimento de mensagens *backup*

input: port, SIZEMAX

output: Armazenamento de mensagem *backup*

```
1:  SocketServer server = new SocketServer(port)
2:  int count = 0
3:  circularListNode head = null
4:  circularListNode tail = null
5:  circularListNode startPointer = null
6:  while True do
7:      server.accept()
8:      Object o = new Object(server.getInputStream())
9:      Record r = (Record) o.readObject()
10:     if count ! = SIZEMAX then
11:         insert(r)
12:     else
13:         insertAfterFull(r)
14:
15: function INSERT(Record r, Int count)
16:     if head == null then
17:         head = new circularListNode(r)
18:         tail = head
19:     else
20:         circularListNode tmp = new circularListNode(r)
21:         tail.setNext(tmp)
22:         tmp.setNext(head)
23:         tail = tmp
24:     count + +
25:
26: function INSERTAFTERFULL(Record r)
27:     if startPointer == null then
28:         startPointer = head
29:     circularListNode current = startPointer
30:     while True do
31:         if current.getRead() == true then
32:             current.replace(r)
33:             break
34:         else
35:             if current.getAge() > 0 then
36:                 resend(current)
37:                 current.replace(r)
38:                 break
39:             current.incrementAge()
40:         current = current.getNext()
41:     startPointer = current.getNext()
```

próximo nó a ser visitado em uma inserção futura (linha 41).

Destaca-se, que a estrutura do código foi definido para que sempre exista uma limitação que define a quantidade exata de mensagens que a lista suporta. Deste modo, sempre que a lista está completa, é necessário investigar se há posições em que o nó armazene uma mensagem previamente confirmada. Com este processo, as substituições na lista acabam forçando o reaproveitamento dos nós sem interferir na estrutura base da lista, bem como no seu tamanho máximo, fazendo com que os recursos destinados a ela sejam reaproveitados.

O nó `startPointer`, presente no Código 4, tem por objetivo otimizar as substituições na lista. Ao manter este ponteiro, sempre que uma nova mensagem é recebida e a lista encontra-se lotada, o monitor recorre ao último nó não verificado em buscas anteriores. Entende-se que neste nó há uma maior probabilidade de a mensagem já ter sido confirmada. Caso isso ocorra, a mensagem é substituída, evitando percorrer toda a lista em busca de espaço para uma nova mensagem.

Muito embora o método `insert()` (linha 15) seja um procedimento padrão de adição de nós a listas, o método `insertAfterFull()` (linha 26) trata de um inserção adaptada às demandas do módulo monitor. A definição de uma lista com lotação bem como a necessidade de haver um meio de identificar perdas demandam mais complexidade no momento inserção de mais nós a lista. Isto ocorre pois, assim como não se pode substituir uma mensagem que ainda não foi retransmitida, deve-se identificar e retransmitir uma mensagem perdida, sendo que essas tarefas devem ocorrer quase que simultaneamente.

O Código 5 exibe o trecho em que o monitor percorre a lista circular para marcar as mensagens presentes nos nós como recebidas. A entrada do Código é um *array* que contém as notificações recebidas (`listReceived`) e um ponteiro que faz referência ao último nó marcado como (`listUnmarked`). Inicialmente, verifica-se se o ponteiro que faz referência ao primeiro nó da lista é `null` (linha 1). Isso indica que ainda não existem elementos na lista, não sendo possível percorrê-la. Na sequência, inicializa-se o ponteiro `lastUnmarked` caso o mesmo ainda seja `null` (linhas 3 e 4). A seguir, dois laços aninhados têm início. O laço externo (linha 5) percorre cada item do *array* que armazenas as notificações. Já o laço interno (linha 7) percorre a lista circular a partir do ponteiro armazenado por *listUnmarked*. Para cada nó presente na lista circular, verifica-se se a notificação recebida corresponde à notificação esperada pelo nó (linha 8). Se a verificação for verdadeira, o nó é marcado como recebido (linha 9), atualiza-se `lastUnmarked` com o próximo item (linha 10) e finaliza-se o laço interno para a notificação em questão (linha 11).

As retransmissões de mensagens efetuadas pelo monitor ocorrem com o nível de *acks* com confiabilidade máxima, ou seja, o nível *Replicas Confirmation*. Para este nível, o Kafka implementa um sistema próprio de retransmissão. Estas tentativas de retransmissões podem ser um fator crucial para prejudicar o desempenho de produtores alocados em redes que sofrem diversas interferências.

Código 5 Monitor: marcando mensagens como recebida

input: listReceived, listUnmarked
output: Mensagens marcadas como lidas na lista cirtular

```
1: if head == null then
2:     return
3: if head != null & lastUnmarked == null then
4:     lastMarked = head
5: for Notification n in listReceived do
6:     circularListNode current = lastUnmarked
7:     while True do
8:         if current.getNotification() == n then
9:             current.setRead(true)
10:            lastUnmarked = current.getNext()
11:            break
12:        current = current.getNext()
```

Está presente nas configurações de produtor uma variável de ambiente que define a quantidade máxima de vezes que um produtor tentará transmitir uma única mensagem. Esta variável de ambiente, denominada `retries`, é do tipo inteiro e tem por padrão o valor 2147483647. Visto que em *Fire-and-Forget* não há um mecanismo que auxilie o produtor a identificar perdas, tal variável torna-se dispensável. Porém, tanto para *Leader Confirmation* quanto para *Replicas Confirmation*, esta variável tende a obrigar que o produtor sempre assegure que uma mensagem chegue com sucesso ao tópico, empregando o tempo em retransmissões que for necessário.

Como o cliente produtor presente no monitor transmite mensagens com o nível *Replicas Confirmation*, ele tende a ser impactado pelas variável `retries`. Para evitar que o monitor possa empregar muito tempo para a transmissão de uma única mensagem, a variável de ambiente `retries` do produtor presente no monitor é inicializada com o valor 0. Ainda assim, para que o monitor assegure que as mensagens retransmitidas cheguem com sucesso ao destino, um *array* separado da lista circular foi preparado para armazenar temporariamente as mensagens a serem retransmitidas.

Antes de retransmitir uma mensagem removida da lista, a mesma é armazenada em um *array* separado da lista circular principal. Após separá-la, executa-se o método do produtor que realiza a transmissão da mensagem. Este método recebe como parâmetro a mensagem propriamente dita e uma instância da classe denominada `Callback`. É por meio desta instância de `Callback` que produtores identificam se a transmissão de mensagem ocorreu com sucesso. Desta forma, este `Callback` somente é funcional quando há transmissão com os níveis *Leader Confirmation* e *Replicas Confirmation*.

Após o monitor identificar que na instância de `Callback` não apresenta nenhuma exceção, ou seja, não apresentou erros, a mensagem é removida do *array*. Caso contrário, a mensagem é mantida para retransmissão futura. Assim, evita-se períodos em que a rede

de transmissão está sofrendo grandes intervalos de interferência. É função do monitor verificar, a cada 3 minutos, a existência de itens no *array* de retransmissões. Caso existam mensagens neste *array*, as mesmas são retransmitidas.

4 EXPERIMENTAÇÃO

Este capítulo descreve a metodologia aplicada na fase experimental do trabalho. Os experimentos conduzidos visam validar a operacionalidade da configuração *rAck*, identificando se a mesma atende à sua proposta. Ainda, as experimentações permitiram comparar, em termos de confiabilidade e desempenho, a configuração *rAck* com os três níveis de configuração de *ack* padrão do Apache Kafka.

Para propiciar a execução dos experimentos, um *cluster* Kafka foi estruturado utilizando a plataforma Docker[1]. Docker é uma tecnologia de virtualização *open-source* que permite alocar os recursos computacionais de uma máquina a entidades encapsuladas denominadas “contêineres". Deste modo, cada contêiner Docker é uma instância isolada. Todos os contêineres operantes em um sistema são gerenciados por uma Docker *engine*. Apesar de cada contêiner possuir um sistema operacional próprio, os contêineres da plataforma Docker podem usufruir de rotinas de biblioteca e partes do kernel Linux do sistema responsável por hospedar os contêineres.

A opção pela utilização de contêineres e não outras tecnologias clássicas de virtualização (como máquinas virtuais) baseou-se na observação de que a plataforma Docker tem se demonstrado uma alterativa viável para implementação de microsserviços em contêineres que se comunicam através de mensagens (BROGI; NERI; SOLDANI, 2017). Além disso, o Docker se mostra versátil e prático na estruturação de ambientes computacionais direcionados para experimentação científica (HIGGINS; HOLMES; VENTERS, 2015).

O trabalho de Wu, Shang e Wolter (2019) apresenta o Docker como uma ferramenta propícia para testar a confiabilidade de transmissões de mensagens no Kafka. Isto ocorre pois a utilização de contêineres permite a construção de um ambiente controlado, possibilitando a introdução de falhas na rede de comunicação entre cada contêiner e a ativação dos mecanismos de garantia de entrega do Kafka. Deste modo, neste trabalho também tentou-se reproduzir um *cluster* Kafka por meio de contêineres. Com o *cluster* sob o Docker, foi possível a introdução de falhas na rede de comunicação entre os contêineres, permitindo uma observação prática e controlada de como a configuração *rAck* e os níveis padrão de *ack* se comportavam durante transmissões de mensagens em redes instáveis.

Após a definição de que a experimentação seria executada sob a tecnologia Docker, foram definidos os parâmetros de execução. Esses parâmetros são a quantidade de mensagens a serem transmitidas, o tamanho das mensagens (em bytes), a quantidade de *brokers*, bem como partições e, também, o fator de replicação aplicado no tópico destino das mensagens transmitidas pelo produtor.

Os parâmetros, assim como o equipamento que executou a experimentação são descritos na Seção 4.1. Na Seção 4.2 são descritos e discutidos os resultados obtidos com

[1] https://www.docker.com/resources/what-container

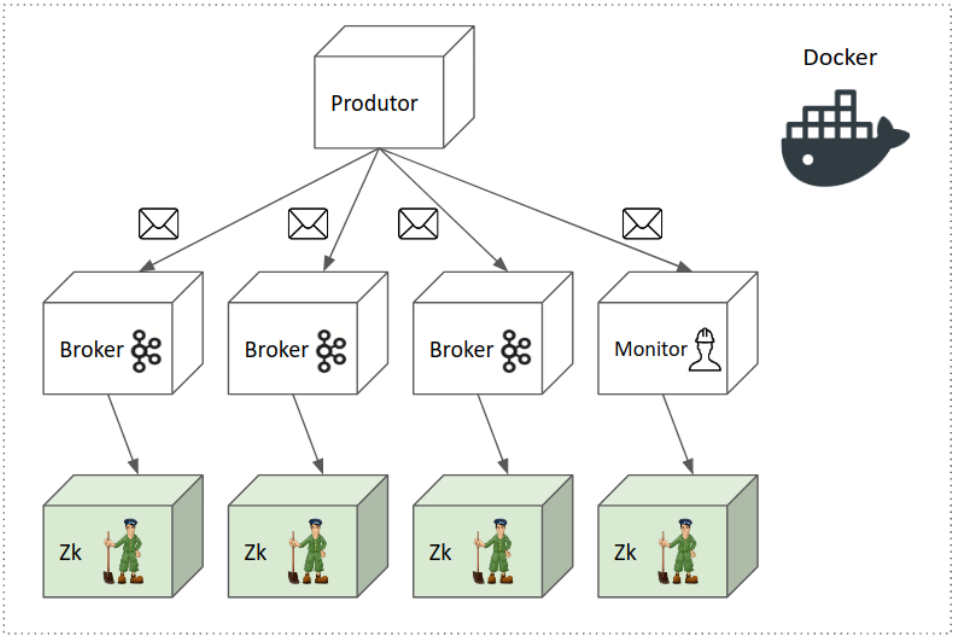
os cenários de testes com perdas de pacotes. Já a Seção 4.3 apresenta uma discussão a respeito dos resultados obtidos nos cenários com atraso na entrega de pacotes.

4.1 AMBIENTE E PARÂMETROS DE EXECUÇÃO

Os experimentos foram conduzidos em um equipamento do Departamento de Linguagens e Sistemas de Computação da UFSM. O equipamento é uma máquina Dell PowerEdge T420 com 4 memórias DIMM DDR3 Síncronas 1600 MHz de 4 GB, 8 memórias DIMM DDR3 Síncronas de 1333 MHz de 8 GB, processador Intel Xeon E5-2420 de 1.90GHz e disco rígido de 1TB. Utilizou-se Docker na versão 20.10.2, Zookeeper versão 3.6.0, Kafka versão 2.3.1 e sistema operacional Ubuntu Server 18.04.

O ambiente foi estruturado com um *cluster* com 9 contêineres, a saber: 1 contêiner contendo um produtor de mensagens, 1 contêiner responsável por executar o monitor, 3 contêineres *brokers* Kafka e 4 contêineres com instâncias do Zookeeper. O ambiente estruturado sobre a plataforma Docker pode ser observado na Figura 4.1.

Figura 4.1 – Estrutura do *cluster* em contêineres para experimentação sob a plataforma Docker.



Fonte: Autor.

De acordo com a Figura 4.1, estão distribuídos no *cluster* quatro clientes Zookeeper que realizam constantes requisições às instâncias do Zookeeper. Os clientes estão presentes nos *brokers* Kafka e no monitor. O cliente posicionado no monitor é responsável pelos registros de *watches*. Para evitar sobrecarga em apenas uma instância do

Zookeeper, foram utilizados quatro contêineres com instâncias do mesmo. Desta forma, garante-se que cada cliente sempre tenha uma instância do Zookeeper disponível para atender sua requisição. Os contêineres são inicializados por meio da ferramenta *docker compose*, a qual possibilita orquestrar contêineres com base em um arquitvo YAML.

Cada teste do experimento consiste na execução de um produtor que transmite mensagens a serem armazenadas em um tópico distribuído entre os três *brokers rAck*. Caso o produtor seja *rAck*, o *broker rAck* realiza o processamento específico da configuração proposta. Se o produtor tiver *ack* configurado em algum nível padrão, o *broker rAck* mantém o processamento específico de cada nível.

O tópico que receberá mensagens é composto por três partições e utiliza fator de replicação três. Ao todo, o tópico é constituído por nove partições, dentre as quais três são eleitas como líderes e as demais são réplicas. A estrutura deste tópico foi definida de forma que nenhum *broker rAck* possa apresentar mais de uma réplica por partição, incluindo as partições líderes. Assim, em cada *broker rAck* há apenas uma réplica de cada partição.

Para definir a quantidade de mensagens a serem enviadas pelo produtor, buscou-se na literatura trabalhos que tenham realizado experimentações com transmissões de mensagens em produtores Kafka. Neste meio, analisando os trabalhos de (Bang et al., 2018) e (Wu; Shang; Wolter, 2019), identificou-se que comumente se aplica a quantidade de 500000 mensagens em cada execução. Portanto, este trabalho adotou este mesmo valor.

No que se refere ao tamanho das mensagens, destaca-se que no trabalho de Bang et al. (2018) o tamanho das mensagens utilizado em experimentos foi de 30 bytes. Em (DOBBELAERE; ESMAILI, 2017), a fase experimental do trabalho utilizou mensagens com tamanhos de 500, 1000, 1500 e 2000 bytes. Em (ZHANG, 2015) as mensagens tinham tamanhos de 200 bytes. Em (Wu; Shang; Wolter, 2019), as mensagens apresentavam tamanhos de 100 e 500 bytes.

Para o presente trabalho, o tamanho das mensagens foi fixado em 500 bytes. Ressalta-se que este tamanho em bytes traduz o seu conteúdo da mensagem propriamente dito. Ao ser encapsulada em pacotes para transmissão na rede, o peso da mensagem é adicionado ao corpo do pacote, ou seja, aos dados de cabeçalho do pacote, fazendo com que o peso do pacote seja superior a 500 bytes.

Após a estruturação do ambiente de testes e a definição dos parâmetros de produtores, mensagens e tópicos, recorreu-se à literatura para identificar métricas direcionadas a avaliar a confiabilidade e desempenho. Em se tratando de produtores Kafka, diversas métricas podem ser aplicadas para avaliar desempenho (Le Noac'h; Costan; Bougé, 2017). Para este trabalho, a métrica aplicada foi a vazão.

Vazão descreve o número de mensagens enviadas por um produtor em um determinado intervalo de tempo (GOODHOPE et al., 2012). Além da vazão, foi coletou-se também, o tempo de execução, o qual mede o tempo necessário para realizar a transmissão de to-

das as 500000 mensagens. Deste modo, o tempo de execução é complementar à vazão pois quanto maior for a vazão, menor é o tempo de execução empregado na transmissão da mesma quantidade de mensagens.

No que tange à confiabilidade da entrega de mensagens, a métrica utilizada foi a taxa de perda de mensagens. Essa taxa indica a percentagem de mensagens perdidas em função de uma probabilidade de perda de pacote ou de atraso de rede, em milissegundos (Wu; Shang; Wolter, 2019). A taxa de perda de mensagens é dada por $1-(M_r \,/\, M_e)$, onde M_r e M_e equivalem, respectivamente, ao número de mensagens recebidas pelo *broker rAck* e ao número de mensagens enviadas pelo produtor.

Visto que em um ambiente real as redes apresentam problemas (Wu; Shang; Wolter, 2019), dois grupos de experimentos foram elaborados. Em cada grupo, uma falha de rede é introduzida no contêiner responsável por executar o produtor. O primeiro grupo representa uma rede que está suscetível a perdas de pacotes. Já no segundo a rede apresenta atrasos na entrega de pacotes. Tais grupos foram definidos com o objetivo de identificar como a configuração *rAck* se comporta em situações onde a rede está propensa a falhas que resultam em interferências na entrega. Ainda, os grupos permitem a comparação do desempenho e da confiabilidade entre a configuração proposta e os níveis *ack* padrão do Kafka.

As falhas foram introduzidas nos contêineres através da ferramenta Pumba (LEDENEV, 2020), que possibilita emular perda e atraso de entrega nas transmissões de pacotes em contêineres Docker. A ferramenta Pumba provê uma interface que utiliza o *Network Emulation* (*netem*) como *backend* (FOUNDATION, 2021). Por meio do *netem*, a ferramenta pumba consegue introduzir falhas de comunicação diretamente na interface de rede responsável pela comunicação entre os contêineres.

4.2 EXPERIMENTOS COM PERDA DE PACOTES

Da mesma forma como a perda de uma mensagem aciona os mecanismos de recuperação dos níveis *Leader Confirmation* e *Replicas confirmation*, perder encadeia a identificação e recuperação exercida pela configuração *rAck*. Deste modo, uma das formas de identificar se o mecanismo de recuperação da configuração *rAck* demonstra-se funcional é induzindo a perdas de pacotes até que ocorra uma perda efetiva de mensagem.

Ao perder uma mensagem, o monitor consegue observar na sua lista interna que a mesma carece de uma confirmação de entrega, iniciando o seu processo de retransmissão. Sendo assim, ao forçar a perda de mensagens pode-se observar se a recuperação exercida pelo monitor está ocorrendo de acordo com o esperado. Além de validar a funcionalidade de recuperação da configuração *rAck*, os experimentos com perda de pacotes permitem observar e comparar o desempenho de *rAck* em relação aos níveis de configu-

ração padrão de *ack* do Kafka.

Foram elaborados quatro cenários para este grupo de experimentos em que cada cenário apresentava uma probabilidade distinta de perder pacotes durante transmissões de mensagens. As probabilidades de perda de pacotes aplicadas nos cenários foram de 0% (sem falhas), 3%, 6% e 9% (com falhas). Essa definição permitiu a observação de como produtores Kafka se comportam com diferentes probabilidades de perda de pacotes. Mesmo com opção por valores distintos para as probabilidades, ainda houve uma preocupação em não ultrapassar o valor máximo de probabilidade de perder pacote definida no trabalho de Wu, Shang e Wolter (2019) (probabilidade de 10%), pois assim mantém-se os valores aplicados na experimentação deste trabalho dentro do que já foi utilizado em outros trabalhos presentes na literatura.

A apresentação e discussão dos resultados obtidos nos experimentos com perdas de pacotes estão dividias em Subseções. Na Subseção 4.2.1 será destinada a análise de desempenho focada na métrica de vazão. Na Subseção 4.2.2 analisa-se o desempenho por meio da métrica de tempo de execução. Por fim, a Subseção 4.2.3 destina-se a análise dos resultados da confiabilidade alcançada pelos níveis de *ack* padrões e a configuração *rAck*.

4.2.1 Desempenho - Vazão

Na Tabela 4.1 estão presentes a média e o desvio padrão da vazão obtida por meio das 20 execuções do produtor para cada configuração de *ack*. Tendo em mente que vazão mede a quantidade de mensagens transmitidas em um intervalo de tempo, quanto maior for a vazão, melhor o desempenho. Na Tabela também observa-se que as médias e os desvios padrões estão organizados por probabilidade, exibindo o cenário sem falhas (chance de perda em 0%) e os cenários com falhas (chances de perda em 3%, 6% e 9%).

Tabela 4.1 – Vazão dos níveis de configuração de ack (em msg/seg)

Nível de Configuração de Acks	Percentual de perda de pacote a cada transmissão							
	0%		3%		6%		9%	
	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão
Fire-and-Forget	7293,885	249,083	4527,804	233,746	2422,109	160,892	1252,759	154,742
Reliable Ack	**212,727**	**26,905**	**190,551**	**12,290**	**180,103**	**12,638**	**181,127**	**11,517**
Leader Confirmation	1367,173	89,469	71,550	6,291	35,118	2,450	23,496	1,886
Replicas Confirmation	744,945	31,589	24,338	1,423	11,835	0,979	7,228	0,938

Fonte: Autor.

Nestes experimentos, as comparações dos resultados obtidos serão realizadas sob dois aspectos. Primeiramente será analisado o cenário sem falha, comparando o desempenho de cada um dos níveis de *ack* do Kafka com a configuração *rAck*. Essa comparação permite observar a diferença, em termos de desempenho, dos níveis padrões com a confi-

guração *rAck* em um cenário ideal de transmissão de mensagens. Após isso, será avaliado o comportamento que cada nível obteve nos cenários com falhas, analisando cada um dos níveis conforme é aumentada a probabilidade de perda de pacote. Dessa forma, é possível avaliar, de forma isolada, o real impacto que as perdas pacote tiveram no desempenho dos níveis testados.

No cenário sem falhas, observa-se que a configuração *rAck* apresenta a menor vazão média se comparada com os níveis de configuração padrão do Kafka. Ao analisar a degradação, em termos de proporção, da média obtida em *Fire-and-Forget* para com a configuração *rAck*, observa-se que há uma degradação de 97,083% nos valores observados. Tal percentual de degradação foi obtido através do cálculo $(212,727 \times 100 / 7293,885) - 100$.

Ainda no cenário sem falhas, a degradação de desempenho observada em *Leader-Confirmation* com a média da vazão da em 1367,173 mensagens por segundo para as 212,727 mensagens por segundo em *rAck* é de 84,44%. Este percentual é obtido por meio de $(212,727 \times 100 \ / \ 1367,173) - 100$.

Por fim, *Replicas-Confirmation* no cenário sem falhas demonstra uma média de 744,945 mensagens por segundo que comparando com a média de 212,727 mensagens por segundo de *rAck*, em termos de proporção, exibe uma degradação de 71,443%. Percentagem obtida por de $(212,727 \times 100) \ / \ 744,945) - 100$.

O menor desempenho observado da configuração *rAck* em todas as execuções do cenário sem falha se deve à necessidade de *rAck* transmitir a mesma mensagem duas vezes, sendo uma para o *broker rAck* e outra para o módulo monitor. Enquanto isso, os níveis de configuração padrões do Kafka no cenário sem falhas se aproximam do caso ótimo, uma vez que não acionam mecanismos de recuperação.

Ao analisar apenas os cenários com falhas, ou seja, execuções que apresentam chances de perda de pacote em 3%, 6% e 9%, observa-se que os níveis de *ack* padrões são prejudicados com as inserções de falha na rede. Da probabilidade 3% para 9%, a configuração *Fire-and-Forget* parte de uma vazão de 4527,804 para 1252,759 mensagens por segundo, representando uma degradação de 72,331%. Esse percentual é alcançado com o cálculo $(1252,759 \times 100 \ / \ 4527,804) - 100$.

Leader Confirmation diminui sua vazão média de 71,550 para 23,496 mensagens por segundo, o que corresponde a 67,161% de degradação. Percentual obtido por meio de $(23,496 \times 100 \ / \ 71,550) - 100$. A vazão da configuração *Replicas Confirmation* diminui de 24,338 para 7,228 mensagens por segundo, indicando uma degradação de 70,301% e este percentual obtém-se por $(7,228 \times 100 \ / \ 24,338) - 100$.

Enquanto os níveis padrão apresentam degradação de, no mínimo, 67,161% em sua vazão, a configuração *rAck* vai de uma vazão de 190,551 mensagens por segundo com 3% de chance de perda para 181,127 mensagens por segundo com 9% de chance de perda, representando uma degradação na sua vazão de apenas 4,945%. A degradação de 4,945% é obtida com $(181,127 \times 100 \ / \ 190,551) - 100$

Tratando a respeito do impacto que a probabilidade de perda de pacotes têm sobre os níveis de *ack* testados, vale ressaltar a vantagem que a configuração proposta apresenta sob os níveis padrões. Estes níveis são prejudicados por perdas de pacotes pois apresentam degradações em suas vazões médias conforme aumenta-se a probabilidade de perder pacotes. Já a configuração *rAck*, nesta mesma situação consegue manter um desempenho regular devido as suas restrições de implementação. Na prática, os níveis padrões do Kafka utilizam apenas um canal de comunicação entre o produtor e o *broker*. Quando um pacote é perdido, o próprio protocolo TCP identifica a perda do pacote devido à falta de uma confirmação de entrega e, após isso, efetua assim a sua retransmissão. O processo de aguardar uma confirmação de entrega para cada pacote perdido faz com que o canal de transmissão fique em espera até que a camada de transporte identifique uma perda de pacote concreta. Esta verificação ocorre por meio de *timeouts* e, após eles chagarem ao seu limite, o pacote é então retransmitido pelo transmissor.

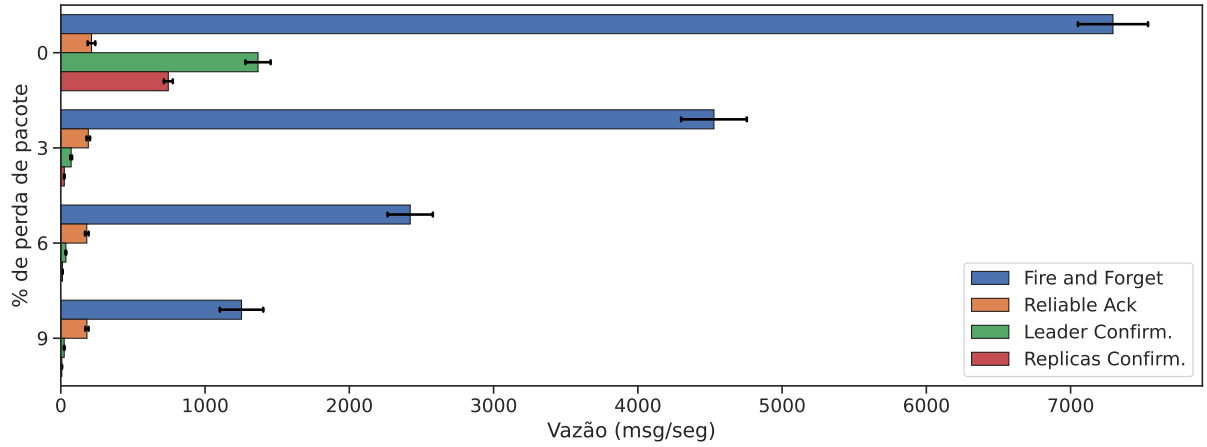
O processo de perda de pacote, quando ocorre, consequentemente tendem a atrasar ainda mais as transmissões de mensagens, sobretudo quando perdas sucessivas de pacotes resultam em perdas definitivas de mensagens. Concluindo, a perda definitiva de uma mensagem no Kafka acontece quando o protocolo TCP desiste de fato da transmissão do pacote que carrega a mensagem. Tal situação pode ocorrer após sucessivas falhas na transmissão do mesmo pacote. Nesse caso, descarta-se a entrega de uma mensagem e essa perda é identificada pelo produtor no mecanismo de *ack* específico do Kafka pois o *broker* não retornará nenhuma confirmação de entrega.

Diferentemente dos níveis padrões de *ack*, a configuração *rAck* mostra-se pouco afetada pelas probabilidades de perda de pacotes devido a dois aspectos. Inicialmente, o fato produtor *rAck* não necessitar de confirmação de entrega garante que, mesmo quando sucessivas falhas na entrega resultem na perda de uma mensagem, não será empregado tempo na retransmissão da mesma. Logo, abdicar de uma confirmação de entrega beneficia o desempenho do produtor *rAck*, assim como ocorre com o nível *Fire-and-Forget*. O outro aspecto, diz respeito ao fato de haver dois canais de transmissão, um com o *broker* *rAck* e outro com o monitor. Apresentar dois canais de transmissão obriga o produtor a intercalar a transmissão entre os mesmos, contribuindo para que o produtor não fique ocioso aguardando as recuperações de pacotes impostas pelo protocolo TCP em um único canal.

A Figura 4.2 exibe graficamente os valores constantes na Tabela 4.1. Através do gráfico, é possível observar o impacto da perda de pacotes na vazão de cada um dos níveis de configuração padrão para *ack*. Em se tratando das configurações focadas em confiabilidade, *Leader Confirmation* e *Replicas Confirmation* passam de um desempenho superior a *rAck* no cenário sem falhas a um desempenho inferior quando se introduz chances de perda de pacotes. Isto ocorre pois produtores que utilizam a configuração *rAck* não interrompem o fluxo de transmissão de mensagens para tratar retransmissões, uma vez que esta é uma tarefa do módulo monitor. Assim, a perda de mensagens pouco interfere

nas transmissões com *rAck*. Já as configurações *Leader Confirmation* e *Replicas Confirmation* são severamente prejudicados pelas perdas. Além de esperar uma confirmação de entrega para cada transmissão, eventuais perdas forçam o produtor a aguardar até 30 segundos para efetuar novamente uma retransmissão de mensagem, sendo este o pior caso.

Figura 4.2 – Gráfico da Vazão dos níveis de configuração de ack.



Fonte: Autor.

Ressalta-se que, por aguardar confirmações de entrega dos *brokers*, as configurações *Leader Confirmation* e *Replicas Confirmation* são duplamente impactadas pela chance de perda de pacotes. Nestas configurações, tanto o pacote que leva a mensagem quanto o pacote que retorna a confirmação de entrega podem ser perdidos. Diferentemente, *Fire-and-Forget* e *rAck* podem apenas perder o pacote que realiza a transmissão da mensagem.

4.2.2 Desempenho - Tempo de Execução

Além da vazão, também estavam presentes nos *logs* de cada produtor os seus respectivos tempos de execução. O tempo de execução compreende o tempo total que o produtor demandou para efetuar a transmissão das 500.000 mensagens. A Tabela 4.2 apresenta os tempos médios das 20 execuções do produtor para cada nível de configuração de *ack*. Existe uma relação de proporção inversa entre vazão e tempo de execução visto que quanto maior a vazão menor é o tempo empregado pelo produtor para enviar as mensagens. Na Tabela 4.2 são exibidas as médias dos tempos de execução para os níveis padrão de *acks* do Kafka e *rAck* tanto no cenário sem falhas (0% de perda) e com falhas (chance de perda de 3, 6 e 9%).

Tabela 4.2 – Tempos de execução dos níveis de configuração de ack (em seg)

Nível de Configuração para Acks	Percentual de perda pacote a cada transmissão							
	0%		3%		6%		9%	
	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão
Fire-and-Forget	75,269	2,083	115,254	5,130	213,270	7,934	393,200	23,579
Reliable Ack	**2468,563**	**60,825**	**2670,293**	**88,533**	**2777,545**	**62,695**	**2771,233**	**72,881**
Leader Confirmation	378,897	23,613	7065,054	52,269	13931,890	87,033	21424,540	88,912
Replicas Confirmation	689,108	26,060	20623,950	95,337	42264,926	218,355	68080,152	324,898

Fonte: Autor.

De acordo com a Tabela 4.2, a configuração *rAck* apresenta o maior tempo de execução para o cenário sem falhas. Neste cenário, o aumento no tempo de execução ocorrido em *rAck* em relação a *Fire-and-Forget* é de, aproximadamente 3179,654%. Esta percentagem se obtém por meio do cálculo $(2468,563 \times 100 \ / \ 75,269) - 100$. O tempo médio obtido em *Leader Confirmation* é de 378,897 segundos, que para o tempo de 2468,563 segundos de *rAck* representa um aumento de 551,512%, percentual obtido por $(2468,563 \times 100 / 378,897) - 100$. Já o tempo médio de *Replicas Confirmation* é de 689,108 segundos, que para o tempo de médio da configuração *rAck* demonstra um aumento de 258,225%, percentual obtido por $(2468,563 \times 100 \ / \ 689,108) - 100$.

A sobrecarga causada pelo envio de mensagens tanto para o *broker rAck* quanto para o monitor afeta negativamente o produtor *rAck*, enquanto isso os níveis padrão do Kafka se aproximam do caso ótimo, uma vez que não despendem tempo em retransmissões.

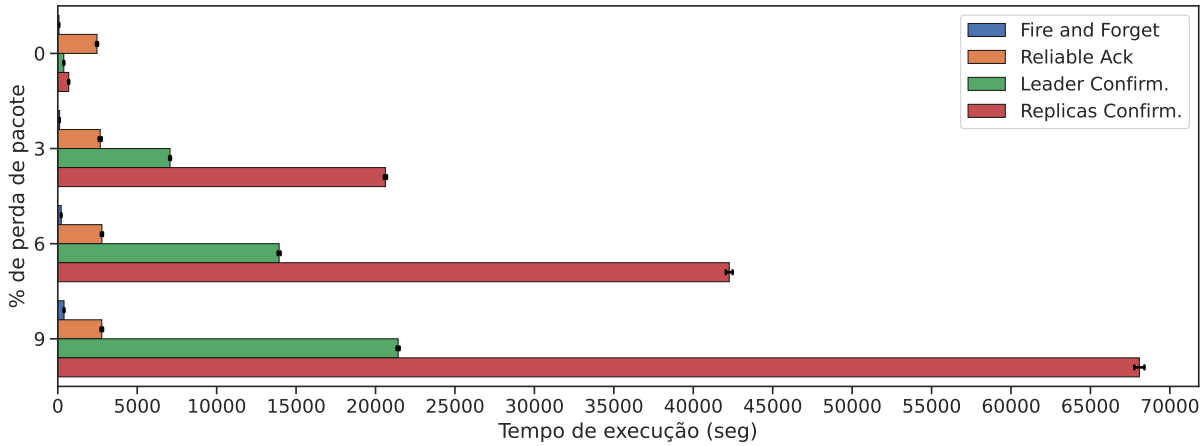
Dentre as médias observadas nos cenários com falhas, a configuração *rAck* é a menos impactada. *Fire-and-Forget* apresenta tempo médio de 115,254 segundos em chande de 3% e de 393,200 segundos em 9%, correspondendo a um aumento de 241,159%, percentual obtido por $(393,200 \times 100 \ / \ 115,254) - 100$. *Leader Confirmation* tem seu tempo aumentando de 7065,054 para 21424,540 segundos, representando um aumento de 203,246%. Este percentual pode ser obtido por $(21424,540 \times 100 \ / \ 7065,054) - 100$. *Replicas Confirmation* aumentou seu tempo de 20623,950 segundos para 68080,152, aumento de 230,102%, percentual de aumento obtido por $(68080,152 \times 100 / 20623,950) - 100$.

Enquanto os níveis padrões apresentam aumentos de no mínimo 200%, a configuração *rAck* apresenta tempo de 2670,293 segundos para percentual de perda de 3% e 2771,233 segundos para percentual de 9%. Sendo assim, a configuração proposta teve um aumento de apenas 3,780% em seu tempo de execução médio. Esse percentual de aumento é alcançado com o cálculo $(2771,233 \times 100 \ / \ 2670,293) - 100$. Os valores de aumento no tempo de execução estão de acordo com o que ocorre na vazão, visto que no cenário com falhas, os níveis padrão apresentam degradação significativa em sua respectiva vazão, reverberando no aumento de seus respectivos tempos.

É possível observar graficamente os dados constantes na Tabela 4.2 por meio da Figura 4.3. Focando no cenário sem falhas, identifica-se que a configuração *rAck* apre-

senta o maior tempo de execução entre as configurações testadas. Entretanto, ao observar os cenários com falhas, identifica-se que tanto a configuração *Leader Confirmation* quanto *Replicas Confirmation* passam a ter tempos de execução maiores que a configuração *rAck*.

Figura 4.3 – Gráfico do Tempo de execução dos níveis de configuração de ack.



Fonte: Autor.

Visto que a reprodução de uma rede perfeita para transmissão de mensagens é custosa, normalmente redes comuns estão sujeitas a instabilidades que podem resultar em perdas de pacotes, por exemplo. Nesses casos, por meio dos resultados observados no Gráfico 4.3, foi possível identificar que a configuração *rAck* consegue se sobressair aos níveis de *ack* do Kafka que se destinam a oferecer certo grau de confiabilidade. Assim como *Leader Confirmation* e *Replicas Confirmation*, a configuração *rAck* demonstra confiabilidade na entrega mensagens. Entretanto, devido a forma como foi implementada a configuração apresentada neste trabalho, demonstra pouca interferência em seu desempenho mesmo quando há perda de pacotes.

4.2.3 Confiabilidade

Após a avaliação do desempenho da configuração *rAck*, foi realizada a análise da confiabilidade de *rAck* isoladamente, buscando observar se as recuperações impostas pelo módulo monitor estavam de acordo com o que foi idealizado. Ao verificar a quantidade de mensagens presentes no tópico destino, observou-se que *Leader Confirmation*, *Replicas Confirmation* e a configuração *rAck* foram capazes de entregar as 500.000 mensagens em todas as execuções. Por outro lado, com o nível *Fire-and-Forget* houveram perdas de mensagens nas probabilidades de 6% e 9%. Em 6%, a média de mensagens perdidas foi de 191,250 mensagens por execução, sendo uma perda de 0,038% das 500.000 enviadas.

Já em 9% a média foi de 1031,350 mensagens perdidas, sendo uma percentagem de 0,206% do total.

Analisando a quantidade de mensagens perdidas de acordo com cada probabilidade, nota-se que a probabilidade até 3% ainda é insuficiente para gerar perdas de mensagens no nível *Fire-and-Forget*. Nessa probabilidade, as recuperações de pacotes efetuadas pelo protocolo TCP são capazes de evitar que mensagens não sejam entregues ao tópico. Entretanto, da probabilidade de 6% a 9%, perdas de mensagens foram observadas. Isso demonstra que, a partir da perda de pacote de 6%, a agressividade da falha é suficiente para resultar em perdas de mensagens.

Na Tabela 4.3 estão exibidos os resultados obtidos por meio da análise da confiabilidade do nível *Fire-and-Forget*. Além da média, está presente na tabela o desvio padrão, e as quantidades mínima e máxima de mensagens perdidas nas execuções do produtor. Para o percentual de 6%, identifica-se que já um desvio padrão de 214,637 mensagens perdidas, o valor mínimo de mensagens perdidas foi de 0 (não houve perda) e o máximo observado foi de 729 mensagens. Já para 9%, temos um desvio padrão de 560,802 mensagens perdidas, a execução com menos mensagens perdidas foi 223 e a máxima foi de 2549 mensagens perdidas.

Tabela 4.3 – Quantidade média de mensagens perdidas nas probabilidades de 6% e 9%

	Percentual de perda de pacote a cada envio							
Nível de Configuração para Acks	6%				9%			
	Média	Desvio Padrão	Min	Max	Média	Desvio Padrão	Min	Max
Fire-and-Forget	191,250	214,637	0	729	1031,350	560,802	223	2549

Fonte: Autor.

Com os dados da Tabela 4.3, prova-se que o aumento na probabilidade de perda de pacote também reflete na quantidade de mensagens perdidas, sendo o nível *Fire-and-Forget* o nível de *ack* mais suscetível a perdas. Muito embora não existam perdas de mensagens nos níveis *Leader Confirmation*, *Replicas Confirmation* e na configuração *rAck*, o aumento das probabilidades de perda também refletiu na quantidade de vezes que tais níveis tiveram que retransmitir mensagens. Nesse contexto, em *Leader* e *Replicas Confirmation* o aumento na quantidade de retransmissões afeta obrigatoriamente o seu desempenho. Já a configuração *rAck* não é afetada pelas retransmissões pois as mesmas são feitas pelo módulo monitor e não pelo produtor.

Realizou-se uma análise do tempo de retransmissão de mensagem efetuada pelo módulo monitor. Para tal, coletou-se o tempo total de retransmissão de todas as transmissões de mensagens realizadas pelo módulo monitor para todos os percentuais de perda de pacote. De posse dos tempos, identificou-se que, conforme aumenta o percentual de perda, aumenta o número de retransmissões efetuadas pelo monitor, ocasionando maior

tempo total de retransmissão. Para chance de 3% há uma média de 69,602 segundos empregues em retransmissão. Para 6% a média é de 315,490 segundos e para 9% a média é de 1077,893 segundos. O aumento da chance de 3% para 9% foi de aproximadamente 1448,652% no tempo de retransmissão total, percentual obtido com o cálculo $(1077,893 \times 100 \ / \ 69,602) - 100$. Esse aumento ocorre pois, além de haver mais mensagens a serem retransmitidas, as retransmissões são efetuadas com alta confiabilidade, utilizando a configuração *Replicas Confirmation*.

4.3 EXPERIMENTOS COM ATRASO NA ENTREGA DE PACOTES

No segundo cenário de experimentos simula-se um *cluster* Kafka que apresenta atraso na entrega de pacotes transmitidos através da rede. Este cenário de testes tem por objetivo aproximar as transmissões de mensagens o máximo possível de um ambiente real de utilização do Kafka, onde a rede está sujeita problemas que reverberam em atrasos na entrega de mensagens.

O cenário de experimentos com atraso na entrega de pacotes foi criado para observar como o *rAck* atua em uma rede que não consegue entregar pacotes precisamente, enfrentando atrasos. Além disso, *brokers* de um *cluster* Kafka podem estar alocados geograficamente distantes, permitindo que ocasionalmente pacotes sejam entregues com atrasos. Sendo assim, este cenário aproxima-se também de um ambiente real de utilização de produtores e *brokers* Kafka.

Outro motivo da estruturação do cenário com atraso na entrega de pacotes é identificar o impacto da existência de confirmações de entrega na transmissão de pacotes. Esta observação é possível neste cenário pois os níveis de configuração de *ack* que apresentam confirmações de entrega são especialmente afetados pelos atrasos, já que é necessário aguardar confirmação antes de transmitir uma nova mensagem. Tais confirmações de entrega também são afetadas por atrasos, forçando produtores a aguardar mais tempo entre transmissões de mensagens se comparados com os níveis de configuração de *acks* sem confirmação de entrega (*rAck* e *Fire-and-Forget*).

As falhas de atrasos são introduzidas no ambiente Docker aproximadamente na metade da transmissão das 500.000 mensagens e possuem duração de 60 segundos. Assim, todo pacote transmitido pelo produtor Kafka durante esse intervalo de tempo é entregue com atraso.

No grupo de experimentos com falhas de atraso de entrega foram executados quatro cenários com tempos de atraso distintos. No primeiro cenário, o tempo de atraso foi de 0 milissegundos (sem falha). Nos outros três foram aplicados os atrasos de 100, 200 e 300 milissegundos, caracterizando cenários com falhas. Os valores de atraso utilizados nesse cenário são baseados no trabalho de Wu, Shang e Wolter (2019).

Nesta Seção, também subdividiu-se a apresentação e discussão dos resultados de acordo com a métrica avaliada. Na Subseção 4.3.1 estaremos apresentando e discutindo os resultados obtidos sobre a métrica de vazão. Na Subseção 4.3.2 serão discutidos os resultados sobre a métrica de tempo de execução. Por fim, na Subseção 4.3.3 serão apresentados os resultados de confiabilidade.

4.3.1 Desempenho - Vazão

Na experimentação com atraso de entrega de pacotes, buscou-se observar nos *logs* do execução de produtor o impacto da entrega tardia de pacotes durante transmissões de mensagens. Na Tabela 4.4 são exibidas as médias da vazão observadas em cada uma das 20 execuções do produtor.

Tabela 4.4 – Vazão dos níveis de configuração de ack (em msg/seg)

Nível de Configuração para Acks	Atraso na Entrega de Pacotes (em millisegundos)							
	sem atraso		100 ms		200 ms		300 ms	
	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão
Fire-and-Forget	7293,885	249,083	4501,229	603,123	4427,962	313,696	4612,900	656,560
Reliable Ack	**212,727**	**26,905**	**200,144**	**13,050**	**193,143**	**34,901**	**212,159**	**17,860**
Leader Confirmation	1367,173	89,469	1388,594	120,143	1352,162	180,794	1362,342	99,440
Replicas Confirmation	744,945	31,589	742,637	57,018	701,611	55,707	739,161	49,040

Fonte: Autor.

Por meio da Tabela 4.4, é possível observar que, diferentemente da experimentação com perda de pacotes, a introdução da falha para o cenário com atraso não tende a impactar severamente os níveis *Leader Confirmation*, *Replicas Confirmation* e a configuração *rAck*. Em contrapartida, do cenário sem atraso para o cenário com atraso de 100 milissegundos, *Fire-and-Forget* exibe uma degradação de 38,287% na sua vazão quando diminui de 7293,885 mensagens por segundo para 4051,229 mensagens por segundo, demonstrando ser o único nível realmente impactado. O percentual de 38,287% foi obtido com o cálculo de $(4501,229 \times 100 \,/\, 7293,885) - 100$.

A introdução da falha de atraso de entrega de pacotes pouco afetou os níveis *Leader* e *Replicas Confirmation* devido ao tempo de duração da falha de apenas 60 segundos. Como nestes níveis a cada transmissão o produtor aguarda por uma confirmação de entrega, poucas mensagens foram diretamente afetadas pela falha. Diferentemente, *Fire-and-Forget* realiza a transmissão de muitas mensagens em um curto intervalo de tempo. Logo, ao colocar várias mensagens para serem transmitidas no canal de transmissão, no momento que a falha tem início, o protocolo TCP impacta a entrega de todas essas em processo de transmissão, afetando a vazão do produtor configurado com *Fire-and-Forget* mais do que os outros níveis.

Apesar de funcionar igualmente ao nível *Fire-and-Forget* no que se refere a aguardar confirmações de entrega, a configuração *rAck* demonstrou-se pouco afetada pela falha de atraso. Isso é devido ao fato de que *rAck* já apresenta uma vazão baixa pois realiza a transmissão de uma quantidade menor de mensagens por intervalo de tempo em decorrência de ter que realizar duas vezes a transmissão da mesma mensagem. Ao intercalar o processamento das transmissões de mensagens, a configuração *rAck* torna-se menos suscetível a falha de atraso, o que não ocorre com o nível *Fire-and-Forget*. Devido a isso, poucas mensagens são diretamente afetadas quando há a introdução da falha de atraso no canal de transmissão.

Focando apenas nos cenários com falhas, ou seja, atrasos de 100, 200 e 300 milissegundos, identifica-se que há pouca ou nenhuma interferência no desempenho de todos os níveis conforme a falha progride. A falha de atraso na entrega de pacote progride conforme a severidade da falha aumenta, nesse sentido, há uma progressão na falha quando aumenta o tempo de atraso de 100 milissegundos para 200, assim como também há progressão de 200 para 300 milissegundos.

Do atraso de 100 para 300 milissegundos, *Fire-and-Forget* melhora sua vazão em aproximadamente 2,480%, visto que aumenta de 4501,229 para 4612,900 mensagens por segundo. Este percentual de 2,480% é obtido por $(4612,900 \times 100 \ / \ 4501,229) - 100$. *Leader Confirmation* apresenta uma redução de 1388,594 para 1362,342 mensagens por segundo, uma degradação de 1,890%. Percentual de 1,890% obtido por $(1362,342 \times 100 \ / \ 1388,594) - 100$. *Replicas Confirmation* demonstra uma redução de 742,637 mensagens por segundo para 739,61, degradação de 0,468%. Percentual de degradação obtido pelo cálculo de $(739,161 \times 100 \ / \ 742,637) - 100$. Por fim, do atraso de 100 para 300 milissegundos, a configuração *rAck* demonstra um aumento na vazão de 200,144 mensagens por segundo para 212,159, o que representa um melhora de 6,003% na sua vazão. Percentual de 6,003% obtido por $(212,159 \times 100 \ / \ 200,144) - 100$.

A pouca interferência das falhas de atraso faz com que todos os níveis de configuração de *acks* demonstrem vazão com pouca variação de desempenho. Além disso, por haver pouca diferença na vazão observada tanto no cenário sem atraso quanto nos cenários com atraso, a configuração *rAck* demonstra a menor vazão em todos os cenários de testes.

O principal motivo pelo qual os níveis *Leader Confirmation* e *Replicas Confirmation* tiveram suas respectivas vazões pouco afetadas durante os experimentos com perdas de pacotes se deve às recuperações de mensagens. A falha de perda de pacote executada no primeiro grupo de experimentos, quando ocorre sucessivamente, é capaz de forçar perdas de mensagens, deixando todo o canal de transmissão ocioso. Diferentemente, a falha de atraso testada não tende a gerar perdas, apenas atrasa a comunicação entre transmissores e receptores por um tempo predeterminado em *Leader Confirmation* e *Replicas Confirmation*. Nesse contexto, a falha de atraso de entrega de pacote com tempo fixo apli-

cada nos cenários definidos não é suficiente para acionar os mecanismos de recuperação dos níveis padrões focados em confiabilidade, o que resulta em um desempenho constante em todos os cenários com atraso.

Mesmo no cenário com atraso de 300 milissegundos, de *Fire-and-Forget* para *rAck* há degradação na vazão de 95,400% ao calcular a redução em percentual existente quando diminuímos de 4612,900 mensagens por segundo para 212,159. Obtém-se este percentual de redução com $(212,159 \times 100 \ / \ 4612,900) - 100$. De *Leader Confirmation* para a configuração *rAck* há uma degradação de 84,426% observando observando uma redução de 1362,342 mensagens por segundo para 212,159. Calcula-se este percentual de redução por $(212,159 \times 100 \ / \ 1362,342) - 100$. Por fim, de *Replicas Confirmation* para a configuração *rAck* vemos uma comparação de 739,161 para 212,159 mensagens por segundo, degradação de 71,297%, e este percentual obtém-se por $(212,159 \times 100 \ / \ 739,161) - 100$.

Visto que a falha pouco interfere no desempenho dos níveis padrões *ack*, observa-se que a sobrecarga resultante da transmissão das mensagens tanto para o monitor quanto para o *broker rAck* faz com que a configuração *rAck* mantenha um desempenho inferior se comparada aos demais níveis de configuração de *ack*.

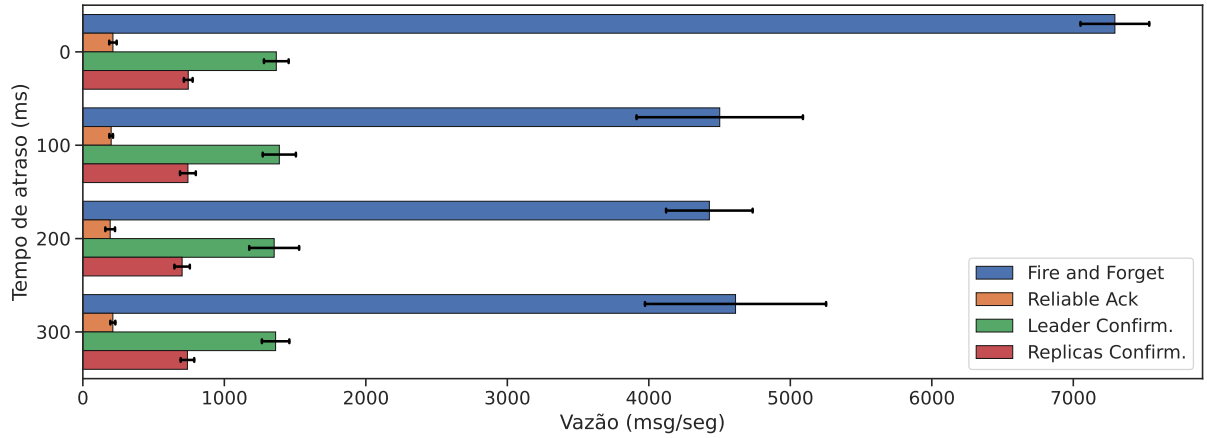
Mesmo sendo o cenário de falha de 300 milissegundos, o que apresenta o tempo de atraso maior, observou-se que a diferença de desempenho entre os níveis de *ack* padrão e a configuração *rAck* se manteve a mesma. Com isso, foi possível concluir que, apesar de haver um canal de transmissão de mensagem a mais, o *overhead* causado por dupla transmissão da mensagem não tem seu desempenho afetado com maior gravidade que os níveis padrão.

A Figura 4.4 exibe graficamente os dados apresentados na Tabela 4.4. Na Figura é possível observar que a única variação significativa no desempenho ocorre em *Fire-and-Forget* na transição do cenário sem falhas para o cenário com falhas de atraso de 100 milissegundos. Levando em consideração o desvio padrão, pode-se concluir que o desempenho de *Leader Confirmation*, *Replicas Confirmation* e a configuração *rAck* não sofre mudanças significativas.

4.3.2 Desempenho - Tempo de Execução

No cenário com atraso de entrega de pacotes também foram coletados os tempos de execução de cada produtor. Sendo assim, a média dos tempos das 20 execuções de cada nível de configuração de *ack* estão exibidas na Tabela 4.5. Por meio desses tempos, é possível concluir que a pouca variação na vazão média observada anteriormente na Tabela 4.4 reverbera na baixa alteração dos tempos de execução tanto no cenário sem falhas (sem atraso) quanto no cenário com falhas. Assim como ocorre com a vazão, apenas o nível *Fire-and-Forget* demonstra-se inicialmente impactado quando há uma transição do cená-

Figura 4.4 – Gráfico da vazão dos níveis de configuração de ack.



Fonte: Autor.

rio sem atraso para o cenário com atraso de 100 milissegundos. Nesta transição, o tempo de execução de *Fire-and-Forget* aumenta de aproximadamente 75,269 para 145,915 segundos, aumento de 93,858%. Este aumento é calculado por $(145,915 \times 100/75,269) - 100$.

Visto que o comportamento do nível *Fire-and-Forget* é destinado a oferecer apenas desempenho, quando a falha de atraso é aplicada no canal de transmissão, uma quantidade maior de mensagens é entregue com atraso se comparada com os outros níveis. Como entregas tardias afetam diretamente vazão, o tempo de execução do produtor *Fire-and-Forget* também tende a aumentar. Diferentemente, *Leader Confirmation*, *Replicas Confirmation* e *rAck* apresentam um número diminuto de mensagens afetadas pela falha de atraso de 100 milissegundos, reverberando em mudanças irrisórias em seus respectivos tempos de execução médio quando se compara o cenário sem falha com o cenário de atraso de 100 milissegundos.

Tabela 4.5 – Tempo de execução dos níveis de configuração de ack (segundos)

Nível de Configuração para Acks	Atraso na Entrega de Pacotes (em millisegundos)							
	sem atraso		100 ms		200 ms		300 ms	
	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão
Fire-and-Forget	75,269	2,083	145,915	3,744	146,761	3,281	146,259	4,712
Reliable Ack	**2468,563**	**60,825**	**2610,553**	**111,082**	**2517,025**	**108,204**	**2548,934**	**107,186**
Leader Confirmation	378,897	23,613	427,259	17,040	441,375	33,899	434,849	21,671
Replicas Confirmation	689,108	26,060	702,465	32,110	733,780	47,534	719,404	37,296

Fonte: Autor.

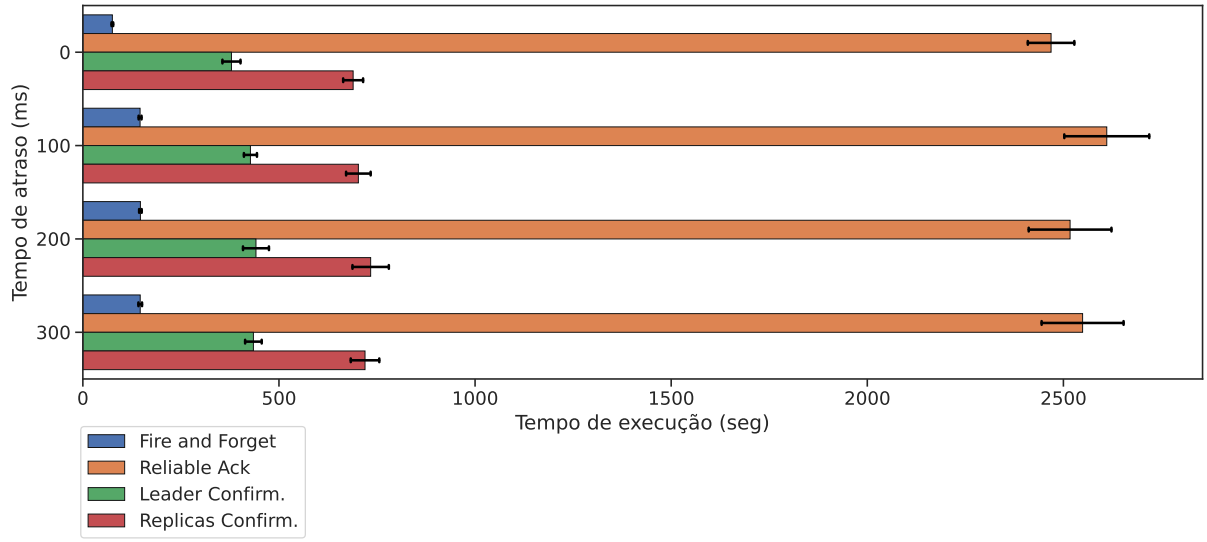
Analisando os cenários com falhas, o aumento do tempo de atraso aplicado em cada transmissão de pacote não tende a impactar severamente os tempos médios de execução. Do cenário de atraso de 100 para 300 milissegundos, *Fire-and-Forget* aumenta seu tempo de execução de 145,915 segundos para 146,259, aumento de apenas 0,235%, percentual obtido por $(146,259 \times 100\,/\,145,915) - 100$. *Leader-Confirmation* tem seu tempo

de execução progredindo de 427,259 segundos para uma média de 434,849, aumento de 1,776% e esse percetual é otido por $(434,849 \times 100 \ / \ 427,259) - 100$. *Replicas Confirmation* aumenta seu tempo de execução de 702,465 segundos para 719,404, aumento de 2,411% obtido por $(719,404 \times 100 \ / \ 702,465) - 100$. Dos tempos de execução mensurados, apenas o tempo de execução do nível configuração *rAck* diminui. Dos tempos observados de 100 para 300 milissegundos, a configuração *rAck* diminui seu tempo de execução de 2610,553 segundos para 2548,934 segundos, uma redução no tempo de apenas 2,360%. O percentual de redução no tempo de execução da configuração *rAck* é obtido por $(2548,934 \times 100 \ / \ 2610,553) - 100$.

O gráfico da Figura 4.5 possibilita observar que a configuração *rAck* apresenta o maior tempo de execução tanto no cenário sem falhas quanto nos cenários com falhas de atraso, mantendo o padrão observado com a vazão. Conforme descrito na Seção 4.1, a vazão tende a influenciar diretamente o tempo de execução pois, conforme aumenta-se ou diminui-se a vazão, mais ou menos tempo é empregado na transmissão de uma quantidade de mensagens.

Conforme se observa no gráfico da Figura 4.5, as vazões de todos os níveis tendem a ser constantes nos quatro cenários, com exceção do nível *Fire-and-Forget* que apresenta uma redução na sua vazão do cenário sem falha para o cenário com falha de atraso de 100 milissegundos. Com isso, o gráfico da Figura 4.5 permite observar que os resultados obtidos com os tempos médios de execução de cada cenário estão de acordo com a vazão.

Figura 4.5 – Gráfico do Tempo de execução dos níveis de configuração de ack.



Fonte: Autor.

Focando apenas na falha de atraso de 300 milissegundos, de *Fire-and-Forget* para a configuração *rAck* há um aumento no tempo de execução de 146,259 para 2548,934 segundos, o que representa um aumento de 1642,753%. Percentual de 1642,753% ob-

tido por $(2548,934 \times 100 \ / \ 146,259) - 100$. De *Leader Confirmation* para *rAck* o aumento no tempo é de 434,849 segundos para os 2548,934, representando um aumento de 486,165%, percentual obtido por $(2548,934 \times 100 \ / \ 434,849) - 100$. Por fim, o aumento de *Replicas Confirmation* para a configuração *rAck* vai de 719,404 segundos para os mesmos 2548,934, indicando um aumento de 254,311%. O percentual de 254,311% obtém-se por $(2548,934 \times 100 \ / \ 719,404) - 100$.

Visto que a duração da falha de atraso impacta pouco na vazão de todos os níveis de configuração padrão de *ack* e em *rAck*, também não se observa mudanças expressivas nos tempos de execução mesmo com o aumento no tempo de atraso dos pacotes. Uma consequência da baixa interferência das falhas de atraso no desempenho de todos os níveis testados é que o tempo de execução médio de *rAck* sempre é superior aos demais em, no mínimo, 254,311%, sendo esse percentual o aumento que há da configuração *rAck* para *Replicas Confirmation*. Sendo assim, ao contrário do que se observa no cenário de perda de pacotes, durante o cenário de atraso de entrega, a configuração *rAck* não se mostrou a opção mais viável em termos de desempenho.

Ao analisar que a diferença de desempenho entre os níveis de *ack* padrões e *rAck* se mantém regular conforme aumenta-se o tempo de atraso aplicado, é possível concluir que o tempo que um pacote demora a ser entregue tem pouco ou nenhum efeito direto no desempenho de produtores. O desempenho semelhante dos níveis em cada cenário indica que a mesma quantidade de mensagens foi impactada durante o momento de atuação da falha, e isso independe se o atraso de transmissão aplicado foi de 100, 200 e 300 milissegundos. Como o tempo de aplicação das falhas de atraso foi o mesmo em todos os cenários (60 segundos), acredita-se que essa seja uma variável que possa prejudicar o desempenho de produtores Kafka quando há atraso de entrega de pacotes. Uma forma de validar tal questão, seria aumentar o tempo de duração da falha ou aumentar a sua intermitência, fazendo com que a falha de atraso seja aplicada com certa frequência.

4.3.3 Confiabilidade

Buscando realizar uma análise da confiabilidade dos níveis padrões e de configuração *rAck* durante os experimentos com atraso, também coletou-se a quantidade de mensagens presentes no tópico ao final de cada execução. Com os resultados obtidos, foi possível identificar que, com exceção da configuração *rAck*, todos os níveis foram impactados com as falhas de atraso, resultando em perdas ou duplicações de mensagens.

Os níveis padrões foram impactados de duas formas, com duplicações ou com perda de mensagens. As duas formas de impacto ocorreram devido ao mecanismo de *ack* do Kafka ser suscetível à instabilidade durante o processo de entrega. O nível *Fire-and-Forget*, por não exigir confirmação de entrega e não realizar retransmissões, pode ser

suscetível a perdas de mensagens. Isso ocorre quando um pacote é perdido devido a um período muito longo de espera por um reconhecimento, ocasionando na desistência de transferência de um pacote. Já *Leader Confirmation* e *Replicas Confirmation* tendem a sofrer duplicações de mensagens quando o produtor retransmite uma mensagem erroneamente devido à esperas muitos longas por reconhecimento positivo.

Tratando especificamente a respeito do nível *Fire-and-Forget*, observou-se que as falhas de atraso ocasionaram perda de mensagens em algumas execuções de produtor. Para a falha de atraso de 100 milissegundos, uma única execução de produtor apresentou perdas, perdendo uma quantidade de 172 mensagens das 500.000 transmitidas, um percentual de 0,0344%. Este percentual é obtido por $(172 \ / \ 500000) \times 100$. Para a falha de 200 milissegundos, houveram perdas em 5 das 20 execuções de produtor, apresentando uma média de 107,6 mensagens perdidas, o que representa um percentual de perda de 0,0215% que pode ser obtido com o resultado de $(107,6 \ / \ 500000) \times 100$. Já para falhas de atraso de 300 milissegundos, houve perda em apenas uma execução, totalizando 101 mensagens perdidas, sendo um percentual de 0,0202% que é obtido por $(101 \ / \ 500000) \times 100$.

Quando as falhas de atraso são introduzidas na redes de transmissão alguns pacotes podem ser desconsiderados quando o receptor atinge o seu limite de espera. Uma vez que o pacote é desconsiderado pelo receptor, a mensagem contida nele pode ser perdida. Como o nível *Fire-and-Forget* não apresenta confirmação de entrega, as mensagens não entregues em decorrência dos atrasos são perdidas permanentemente.

Por sua vez, uma duplicação de mensagem pode ocorrer quando a camada de retransmissão do Kafka, ou seja, seu mecanismo de *ack*, acredita que é necessário retransmitir uma mensagem que ainda não chegou ao destino. Uma das formas de ocorrer essa falta de sincronismo é quando o tempo que o produtor aguarda por uma confirmação de entrega do *broker* ultrapassa o tempo necessário para o pacote que carrega a mensagem chegar até ao seu destino.

Na Tabela 4.6 são exibidas as médias, desvios padrões e quantidades máximas e mínimas de mensagens duplicadas nas execuções de produtor durante os cenários com atraso de entrega de pacote. Na Tabela estão presentes apenas os valores de *Leader Confirmation* e *Replicas Confirmation*, pois foram os únicos níveis a exibirem duplicações. Como é possível observar, para ambos os cenários, em todos os tempos de atraso foram observadas duplicações. A quantidade de duplicações mínima e máxima ficou entre 0 e 4 mensagens duplicadas em cada execução de produtor.

Diferentemente dos níveis padrões de *acks*, a configuração *rAck* não apresentou duplicação e nem perda de mensagens em nenhuma das execuções para todos os tempos de atraso. Entretanto, ao checar os *logs* do monitor, observou-se que houveram retransmissões sendo efetuadas nas execuções de todos os cenários de atraso. Isto sugere que o monitor foi eficaz na identificação e recuperação de mensagens. No que se refere ao tempo de retransmissão do monitor, para o tempo de atraso de 100 milissegundos, a

Tabela 4.6 – Quantidade média de mensagens duplicadas nos cenários com atraso de entrega

Nível de Configuração para Acks	Atraso na Entrega de Pacotes (em millisegundos)											
	100 ms				200 ms				300 ms			
	Média	Desvio Padrão	Min	Max	Média	Desvio Padrão	Min	Max	Média	Desvio Padrão	Min	Max
Leader Confirmation	2,3	0,865	0	3	1,95	0,865	1	3	1,65	0,745	0	3
Replicas Confirmation	1,8	0,768	0	3	2,35	0,988	1	4	1,25	0,7164	0	2

Fonte: Autor.

média foi de 31,869 segundos empregados em retransmissão. Para o tempo de 200 milissegundos, a média observada é 42,715 segundos. Já para o tempo de 300 milissegundos, a média baixa para 25,781 segundos.

A respeito do tempo de retransmissão do monitor, ressalta-se que as retransmissões ocorrem quando o monitor observa que a mensagem que precisa ser transmitida já está há muito tempo em sua lista circular. Nesse sentido, as retransmissões são efetuadas paralelamente às transmissões do produtor *rAck*, não interferindo na vazão e, consequentemente, no tempo de execução.

Diferentemente, toda retransmissão que for efetuada por produtores configurados por *Leader Confirmation* e *Replicas confirmation* afeta diretamente a vazão do produtor. Tal impacto é devido aos dois níveis sempre aguardarem confirmações de entrega antes de realizar retransmissões e novas transmissões. Já na configuração *rAck*, sendo as retransmissões paralelas do módulo monitor transmissões do produtor *rAck*, permite-se que o produtor se dedique unicamente as suas transmissões.

Apesar de não demonstrar um desempenho superior aos níveis padrão de *acks*, em termos de confiabilidade para os cenários de atraso na entrega de pacotes, a configuração *rAck* mostrou ser confiável. Com os resultados obtidos, conclui-se que a configuração *rAck* eliminou a possibilidade de duplicação de mensagens, além de ser capaz de recuperar as mensagens perdidas.

5 CONCLUSÃO

A adoção de microsserviços para suprir a demanda por aplicações modulares e escaláveis resultou na problemática inerente ao paradigma de microsserviços que trata especificamente da comunicação entre cada serviço. A comunicação confiável entre os microsserviços é de fundamental importância para que exista uma perfeita coordenação e sincronia entre cada serviço, possibilitando a boa execução de cada funcionalidade de uma aplicação.

O Apache Kafka atualmente tem se demonstrado uma solução recorrente para viabilizar a comunicação confiável entre microsserviços por meio da mensageria. Para implementar a garantia de entrega de mensagens, está presente na arquitetura do Kafka o mecanismo de *ack*, o qual define o nível de confiabilidade que será empregado durante as transmissões de mensagens. Por padrão, existem três níveis para configuração de *acks*. *Fire-and-Forget* é o nível focado em desempenho, enquanto *Leader Confirmation* e *Replicas Confirmation* são níveis focados em confiabilidade.

Entretanto, apesar de apresentar três níveis para *acks*, em situações de transmissões em redes instáveis, o Kafka apresenta restrições de implementação que o impedem de conciliar desempenho e confiabilidade, exigindo que os usuários tenham que priorizar um destes requisitos. Desta forma, os usuários são obrigados a escolher entre desempenho, aceitando a possibilidade de perder mensagens, ou confiabilidade, aceitando a degradação de desempenho imposta pelas garantias de entrega e persistência das mensagens aos *brokers*.

Este trabalho apresentou a configuração *rAck*. Esta configuração baseia-se na coordenação e cooperação de três módulos que foram anexados à arquitetura do Apache Kafka, com o objetivo de aumentar a confiabilidade de transmissões de mensagens sem reconhecimento positivo. Os módulos são: o produtor *rAck*, o *broker rAck* e o monitor.

O produtor *rAck* é um produtor capaz de transmitir mensagens tanto para o *broker rAck* quanto ao módulo monitor. A mensagem para o *broker rAck* é armazenada no tópico destino, já a mensagem transmitida ao monitor trata-se de uma cópia a ser utilizada em caso de retransmissão. O *broker rAck* é um *broker* Kafka adaptado para receber transmissões de mensagens do produtor *rAck*. Além disso, o *broker rAck* apresenta uma integração com o Apache Zookeeker, o que permite realizar alterações na árvore de *Znodes* desta plataforma. As alterações são realizadas para indicar a recepção correta de uma mensagem. Por fim, o monitor é o módulo responsável por gerenciar a árvore do Zookeeper, observando alterações resultantes das ações do *broker rAck*. Além disso, o monitor também é responsável por realizar a retransmissão de mensagens identificadas como perdidas.

Para validar a operacionalidade da configuração *rAck* e compará-la com os níveis

de configuração padrão de *ack* no Kafka, executou-se uma experimentação que subdividiu-se em dois tipos de experimentos. Em ambos os experimentos, um *cluster* Kafka foi estruturado sob um ambiente Docker para que produtores realizassem transmissões de mensagens na medida em que eram coletadas métricas de desempenho e confiabilidade. Além disso, foram introduzidas falhas de rede que permitiram observar como os produtores Kafka convencionais e o produtor *rAck* reagiam a transmissões de mensagens em redes instáveis.

No primeiro grupo de experimentos, durante as transmissões de mensagens foi introduzida na rede de transmissão de mensagem uma falha de probabilidade de perda de pacote, ou seja, para cada pacote transmitido na rede existia uma probabilidade de perdê-lo, podendo resultar em perdas de mensagem. Ainda no primeiro grupo de experimento, quatro cenários de testes foram idealizados para realizar execuções de produtor, um cenário sem falha, onde não existia falha a ser introduzida, e três cenários com falhas. Os três cenários com falhas apresentavam a probabilidade de perda de pacote em 3%, 6% e 9%.

No segundo grupo de experimentos, a falha introduzida na rede era de atraso na entrega de pacotes, logo todo pacote transmitido na rede era entregue com um atraso de milissegundos pré-determinado. Novamente, mais quatro cenários de testes foram definidos, sendo um cenário sem falha, não havendo atraso de entrega, e três cenários com falhas. Os cenários com falhas apresentavam atraso de entrega de 100, 200 e 300 milissegundos respectivamente.

A experimentação do primeiro grupo demonstrou que a configuração *rAck* torna-se uma opção viável frente à *Leader Confirmation* e *Replicas Confirmation* quando há perdas de pacotes. Quando diversas perdas de pacotes são suficientes para resultarem na perda de uma mensagem, *Leader Confirmation* e *Replicas Confirmation* empregam muito tempo em recuperação. Este tempo empregado, afeta diretamente na métrica de vazão, a qual mede o desempenho de produtores.

No que se refere aos experimentos com atraso na entrega de pacotes, os níveis padrão de *ack* não foram severamente afetados pela falha, não apresentando mudanças significativas em seus desempenhos individuais. Sendo assim, para todos os cenários de atrasos a configuração *rAck* apresentou as piores médias de vazão e tempo de execução. Entretanto, no que tange a confiabilidade, as falhas de atrasos sempre resultaram em perdas ou duplicações de mensagens nos níveis padrão de *ack*, situação que não ocorreu com a configuração *rAck* em nenhuma execução de produtor em nenhum cenário com falha de atraso.

Também identificou-se que durante a experimentação com atrasos na entrega de pacotes, os níveis *Leader Confirmation* e *Replicas Confirmation* foram beneficiados pelo pouco tempo de atuação da falha (60 segundos), visto que os mesmos aguardam durante 30 segundos por confirmações de entrega. Com isso, durante metade da atuação da falha, os níveis ao invés de estarem transmitindo mensagens, estão ociosos, fato que não ocorre

com *Fire-and-Forget* e a configuração *rAck*

Durante a fase de implementação da configuração *rAck*, observou-se que as escolhas de implementação do monitor o limitavam a garantir a entrega de pacotes de um único produtor, visto que o mesmo somente consegue monitorar a árvore de *Znodes* esperando mensagens de um produtor específico. Ainda, por se tratar de uma instância separada, o monitor acaba utilizando recursos computacionais que estariam livres para serem alocados a outras instâncias do Kafka. Ademais, a atual implementação do *broker rAck* não consegue assegurar a ordem de entrega de mensagens, pois retransmissões, quando são necessárias, ocorrem pelo monitor em paralelo ao produtor.

Levando em consideração os pontos observados a respeito das restrições de implementação de *rAck* e do que foi observado a partir dos resultados obtidos em experimentação, os trabalhos futuros são direcionados tanto a otimizações na implementação da configuração *rAck* e alterações nos parâmetros aplicados durante os experimentos. No que tange a implementação, serão buscadas formas de aumentar a escalabilidade do monitor, permitindo que o mesmo consiga atender a demanda de vários produtores *rAck*. Também serão alcançadas formas da realizar a garantia de ordem de entrega de mensagens nos *brokers rAck* por meio de um *buffer* de pré-ordenação de mensagens.

Para a fase experimental, serão estruturados experimentos que visem considerar a utilização de recursos de cada nível de configuração de *ack* e da configuração *rAck*. Ainda, o cenário com atraso na entrega de pacotes será estruturado de forma que a falha não beneficie nenhum nível de *ack* específico. Para isso, além de se diminuir o tempo de espera por confirmações dos níveis *Leader* e *Replicas Confirmation*, também serão introduzidos atrasos com *jitter*, ou seja, variações estatísticas no tempo de atraso de entrega.

Ao fim, havendo diversos casos de aplicação do Kafka, identificou-se que a configuração *rAck* beneficia aplicações que, primeiramente não permitam perdas de mensagens, e que não exigem que a ordem de entrega das mesmas não seja um requisito. Nesse contexto, aplicações de microsserviços focadas em agregações de *logs* e *tracking* de eventos podem-se aproveitar da configuração *rAck* para a produções de mensagens. Nesse tipo de aplicações, a entrega dos dados é fundamental para permitir um processamento posterior que pode propiciar a tomada de decisão. Ainda, a ordem como cada *logs* ou evento chega ao destino não é um requisito obrigatório, e isto ocorre pois cada *log* ou evento apresenta o tempo que o dado foi gerado como uma parte intrínseca da mensagem que estará sendo transportada.

Ressalta-se que a configuração *rAck* demonstrou-se eficiente em situações em que as redes de transmissões estejam sujeitas a falhas. Em redes apropriadas para realizar transmissões de pacotes, ou seja, redes que não ocorrem perdas ou atrasos de entrega de pacotes, a configuração *rAck* exibiu desempenho inferior aos níveis padrões de *acks*. Com isso, engloba-se também nas perspectivas de trabalhos futuros, otimizações da configuração em cenários sem falhas.

Diferentemente, aplicações voltadas a processamento de estados de aplicações, tendem a priorizar a ordem de entrega das mensagens pois a cronologia de recebimento de mensagens indica o estado atual da aplicação. Nesses casos de uso, a falta de uma garantia de ordem de entrega de mensagens na configuração *rAck* atual faz com que este nível proposto não seja recomendável.

Para fins de conclusão de mestrado, este trabalho foi publicado no XI Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC) (CORRêA; BARCELOS, 2021). Além disso, os códigos implementados durante o desenvolvimento do produtor *rAck*[1], *broker rAck*[2] e do monitor[3] estão disponíveis publicamente no Github.

[1] https://github.com/icunhacorrea/producer-rack
[2] https://github.com/icunhacorrea/broker-rack
[3] https://github.com/icunhacorrea/monitor-rack

REFERÊNCIAS BIBLIOGRÁFICAS

Alaasam, A. B. A.; Radchenko, G.; Tchernykh, A. Stateful stream processing for digital twins: Microservice-based kafka stream dsl. In: **2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)**. [S.l.: s.n.], 2019. p. 0804–0809. Acesso em: 8 de out. 2020.

Bang, J. et al. Design and implementation of a load shedding engine for solving starvation problems in apache kafka. In: **NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium**. [s.n.], 2018. p. 1–4. Acesso em 24 abr. 2020. Disponível em: <https://ieeexplore.ieee.org/document/8406306>.

BHIMANI, P.; PANCHAL, G. Message delivery guarantee and status update of clients based on iot-amqp. In: HU, Y.-C. et al. (Ed.). **Intelligent Communication and Computational Technologies**. Singapore: Springer Singapore, 2018. p. 15–22. ISBN 978-981-10-5523-2. Acesso em: 6 de jan. 2021. Disponível em: <https://link.springer.com/chapter/10.1007/978-981-10-5523-2_2>.

BROGI, A.; NERI, D.; SOLDANI, J. A microservice-based architecture for (customisable) analyses of docker images. **Software: Practice and Experience**, v. 48, 09 2017.

CORRêA, I.; BARCELOS, P. rack: Proposta de configuração para garantia de entrega de mensagens no apache kafka. SBC, Porto Alegre, RS, Brasil, p. 80–85, 2021. ISSN 2763-9002. Disponível em: <https://sol.sbc.org.br/index.php/sbesc_estendido/article/view/18497>.

DOBBELAERE, P.; ESMAILI, K. S. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In: **Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems**. New York, NY, USA: Association for Computing Machinery, 2017. (DEBS '17), p. 227–238. ISBN 9781450350655. Acesso em: 20 de out. 2020. Disponível em: <https://doi.org/10.1145/3093742.3093908>.

DRAGONI, N. et al. Microservices: Yesterday, today, and tomorrow. In: ____. **Present and Ulterior Software Engineering**. Cham: Springer International Publishing, 2017. p. 195–216. ISBN 978-3-319-67425-4. Acesso em: 8 de jan. 2021. Disponível em: <https://doi.org/10.1007/978-3-319-67425-4_12>.

FOUNDATION, A. S. **Apache Kafka is a distributed streaming platform. What exactly does that mean?**: Introduction. Apache Software Foundation, 2011. Acesso em: 22 jul. 2020. Disponível em: <https://kafka.apache.org/intro>.

____. **What is Apache Ignite?**: Apache ignite documentation. Apache Software Foundation, 2015. Acesso em 11 nov. 2021. Disponível em: <https://apacheignite.readme.io/docs>.

FOUNDATION, L. **Netem**: Network emulation. Linux Foundation, 2021. Acesso em: 8 de out. 2020. Disponível em: <https://wiki.linuxfoundation.org/networking/netem>.

GOODHOPE, K. et al. Building linkedin's real-time activity data pipeline. **IEEE Data Eng. Bull.**, v. 35, p. 33–45, 2012. Acesso em: 13 de nov. 2020. Disponível em: <http://sites.computer.org/debull/A12june/p33.pdf>.

HIGGINS, J.; HOLMES, V.; VENTERS, C. Orchestrating docker containers in the hpc environment. In: KUNKEL, J. M.; LUDWIG, T. (Ed.). **High Performance Computing**. Cham: Springer International Publishing, 2015. p. 506–513. ISBN 978-3-319-20119-1.

Acesso em: 12 de nov. 2020. Disponível em: <https://link.springer.com/chapter/10.1007/978-3-319-20119-1_36>.

JHA, D. N. et al. A study on the evaluation of hpc microservices in containerized environment. In: ____. **Concurrency and Computation: Pratice and Experience**. Wiley Online Library, 2019. Acesso em: 15 de jan. 2021. Disponível em: <https://onlinelibrary.wiley.com/doi/epdf/10.1002/cpe.5323>.

JUNQUEIRA, F.; REED, B. **Zookeeper: Distributed Process Cordination**. 1th. ed. Cambridge: O'Reilly Media, 2013. 238 p.

KUL, S. et al. Event-based microservices with apache kafka streams: A real-time vehicle detection system based on type, color, and speed attributes. **IEEE Access**, v. 9, p. 83137–83148, 2021. Acesso em: 8 de out. 2020.

Le Noac'h, P.; Costan, A.; Bougé, L. A performance evaluation of apache kafka in support of big data streaming applications. In: **2017 IEEE International Conference on Big Data (Big Data)**. [s.n.], 2017. p. 4803–4806. Acesso em: 12 de nov. 2020. Disponível em: <https://ieeexplore.ieee.org/document/8258548>.

LEDENEV, A. **Pumba: Chaos Testing Tool for Docker**: Chaos testing, network emulation and stress testing tool for containers. Alexei Ledenev, 2020. Acesso em: 1 de out. 2021. Disponível em: <https://github.com/alexei-led/pumba>.

Lee, S. et al. Correlation analysis of mqtt loss and delay according to qos level. In: **The International Conference on Information Networking 2013 (ICOIN)**. [s.n.], 2013. p. 714–717. Acesso em: 6 de jan. 2021. Disponível em: <https://ieeexplore.ieee.org/document/6496715>.

LUZURIAGA, J. E. et al. Testing amqp protocol on unstable and mobile networks. In: FORTINO, G. et al. (Ed.). **Internet and Distributed Computing Systems**. Cham: Springer International Publishing, 2014. p. 250–260. ISBN 978-3-319-11692-1. Acesso em: 7 de jan. 2021. Disponível em: <https://link.springer.com/chapter/10.1007/978-3-319-11692-1_22>.

Luzuriaga, J. E. et al. A comparative evaluation of amqp and mqtt protocols over unstable and mobile networks. In: **2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)**. [s.n.], 2015. p. 931–936. Acesso em: 6 de jan. 2021. Disponível em: <https://ieeexplore.ieee.org/document/7158101>.

Marinov, T.; Nenova, M.; Iliev, G. Message transmission protocol in vanet. In: **2019 X National Conference with International Participation (ELECTRONICA)**. [s.n.], 2019. p. 1–4. Acesso em: 7 de jan. 2021. Disponível em: <https://ieeexplore.ieee.org/document/8825644>.

NARKHEDE, N.; SHAPIRA, G.; PALINO, T. **Kafka: The Definitive Guide**. 1th. ed. Cambridge: O'Reilly Media, 2017. 322 p.

PIVOTAL. **RabbitMQ: Messaging that just works**: Home. Pivotal Software, 2015. Acesso em: 24 abr. 2020. Disponível em: <http://wrf-model.org/>.

Sundarasekar, R. et al. Adaptive energy aware quality of service for reliable data transfer in under water acoustic sensor networks. **IEEE Access**, v. 7, p. 80093–80103, 2019. Acesso em: 7 de jan. 2021. Disponível em: <https://ieeexplore.ieee.org/document/8733845>.

TANENBAUM, A.; WETHERALL, D. **Redes de Computadores**. 5th. ed. Belo Horizonte, Minas Gerais: Pearson, 2011. 600 p.

Wu, H.; Shang, Z.; Wolter, K. Trak: A testing tool for studying the reliability of data delivery in apache kafka. In: **2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)**. [s.n.], 2019. p. 394–397. Acesso em 24 abr. 2020. Disponível em: <https://ieeexplore.ieee.org/document/8990246>.

ZHANG, T. Reliable event messaging in big data enterprises: Looking for the balance between producers and consumers. In: **Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems**. New York, NY, USA: Association for Computing Machinery, 2015. (DEBS '15), p. 226–233. ISBN 9781450332866. Acesso em 24 abr. 2020. Disponível em: <https://doi.org/10.1145/2675743.2771878>.