rAck: Proposta de Configuração para Garantia de Entrega de Mensagens no Apache Kafka

Iago da Cunha Corrêa, Patrícia Pitthan Barcelos
Pós-Graduação em Ciência da Computação (PPGCC)
Universidade Federal de Santa Maria (UFSM)
Santa Maria – RS, Brasil
{icorrea, pitthan}@inf.ufsm.br

***Resumo*—O Kafka é uma plataforma de mensageria e streaming que segue um modelo produtor-consumidor. Visando garantir a entrega de mensagens, o Kafka apresenta um mecanismo de Reconhecimento Positivo (ack). Apesar de existirem três níveis distintos de configuração padrão para ack, os mesmos apresentam restrições de confiabilidade ou desempenho durante transmissões em redes instáveis, obrigando os usuários a priorizar um destes requisitos. Este trabalho propõe o *Reliable Ack* (*rAck*), uma configuração para transmissão de mensagens baseada no monitoramento, identificação e recuperação de mensagens em caso de perda. Experimentos foram conduzidos com o objetivo de comparar a configuração *rAck* com os níveis padrão de ack do Kafka, permitindo a observação de que a configuração proposta é viável para transmissões de mensagens em redes suscetíveis a perdas de pacotes, recuperando mensagens e apresentando desempenho satisfatório.**

***Palavras-chave*—Apache Kafka, Mensageria, Confiabilidade**

I. Introdução

O Apache Kafka [1] é uma plataforma *open source* de processamento distribuído voltada para mensageria e *streaming* de dados, comumente associada à comunicação entre microsserviços. A arquitetura do Kafka apresenta um mecanismo para assegurar confiabilidade na entrega de mensagens, o Reconhecimento Positivo (*Positive Acknowledgement*, ou *ack*). O Kafka possui três níveis de configuração para *ack*, que diferem entre si em relação às garantias de confiabilidade e desempenho. O níveis padrão para *ack* no Kafka são *Fire-and-Forget*, *Leader Confirmation* e *Replicas Confirmation*.

O nível *Fire-and-Forget* é focado em oferecer desempenho, enquanto *Leader Confirmation* e *Replicas Confirmation* centram seus esforços em fornecer confiabilidade. A definição do nível de configuração de *ack* a ser utilizado tende a ser uma tarefa complexa. Isto ocorre pois ao utilizar o nível *Fire-and-Forget*, que prioriza o desempenho, a aplicação está sujeita a perdas de mensagens. A opção por níveis mais confiáveis elimina as chances de perdas, entretanto, impõe degradação de desempenho nas transmissões de mensagens. Neste sentido, aplicações que exigem entrega de mensagem com alta confiabilidade e no menor tempo possível são prejudicadas em transmissões em redes instáveis, já que os níveis padrão de configuração para *ack* no Kafka apresentam dificuldades em conciliar confiabilidade e desempenho [2] [3].

Este trabalho apresenta uma proposta de configuração para o mecanismo de *ack* do Apache Kafka visando aumentar a confiabilidade de transmissões de mensagens sem confirmação de entrega. A configuração proposta, denominada *Reliable Ack* (*rAck*), centra-se na adequação de três módulos à arquitetura do Kafka. Os módulos possuem atribuições distintas durante o fluxo de transmissão, porém atuam de forma cooperativa e coordenada para identificar e recuperar mensagens. O primeiro módulo consiste em um novo nível para *ack* nos produtores Kafka, transformando um produtor convencional do Kafka em um produtor *rAck*. O segundo módulo corresponde a adaptações aos serviços do Kafka para receber produções de mensagens com a configuração *rAck*. Por fim, o terceiro módulo se refere a um monitor responsável por controlar o recebimento de mensagens e efetuar retransmissões quando perdas forem identificadas. Para monitorar o recebimento de mensagens, os módulos da configuração *rAck* exploram os serviços do Apache Zookeeper [4], um agente coordenador de sistemas distribuídos. A fim de comparar a configuração *rAck* com os níveis de configuração padrão no Kafka, o trabalho apresenta ainda uma experimentação que leva em consideração os requisitos de confiabilidade e desempenho.

O artigo está organizado em 6 seções. A Seção II aponta os principais trabalhos identificados como correlatos. A Seção III aborda o Apache Kafka juntamente com seu mecanismo de *ack* e apresenta a relação existente entre o Apache Kafka e o Zookeeper. A Seção IV descreve a configuração *Reliable Ack* (*rAck*). A Seção V apresenta os experimentos realizados, enquanto a Seção VI detalha os resultados obtidos. Por fim, a Seção VII conclui o artigo e direciona os trabalhos futuros.

II. Trabalhos Relacionados

A literatura dispõe de diversos trabalhos que enfocam garantia de entrega durante as transmissões de dados. O trabalho de [5] propõe a ferramenta TRAK (*Testing the Reliability of Apache Kafka*), a qual visa avaliar a confirmação de entrega de mensagens do mecanismo de *ack* do Kafka. Com esta ferramenta, os autores criaram um ambiente de testes com contêineres Docker e introduziram falhas na rede durante a transmissão de mensagens. A análise da confiabilidade ocorre por meio da métrica de taxa de entrega de mensagens. Já o

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

trabalho de [6] apresenta um protocolo orientado à mensageria voltado para transmissões de mensagens *Internet of Things*. O protocolo segue um modelo *store-and-foward* em que o emissor armazena a mensagem até que a mesma seja completamente transmitida. O modelo provê garantia de entrega mesmo em situações onde a rede está sujeita a falhas.

Em [7], os autores realizam uma análise da correlação entre os níveis de QoS (*Quality of Service*) do protocolo *Message Queuing Telemetry Transport* (MQTT) em redes que aceitam perdas e atrasos na entrega de pacotes. Os autores afirmam que, para todos os níveis de QoS do protocolo MQTT, o tamanho do pacote transmitido influencia diretamente no atraso de entrega e na probabilidade de perda. Os autores de [8] apresentam uma análise experimental do comportamento dos protocolos de mensageria *Advanced Message Queuing Protocol* e MQTT em redes móveis e instáveis. A fim de realizar a análise, um produtor de mensagens foi submetido a transmissões em redes sujeitas a atrasos e perdas. Os autores afirmam que os protocolos avaliados são robustos e garantem a entrega de mensagens sem perdas.

Dos trabalhos acima mencionados, foram identificadas possibilidades de transmissões confiáveis de mensagens como, por exemplo, o armazenamento de cópias de mensagens visando recuperação e descarte das mensagens transmitidas com sucesso. Estas possibilidades foram úteis na implementação da configuração *rAck*, visto que a principal contribuição de *rAck* visa identificar e recuperar mensagens perdidas [6] [8]. Os trabalhos também demonstraram estruturas de experimentação com transmissões de mensagens Kafka já consolidadas na literatura. Portanto, o uso de Docker e Pumba para simulação de redes instáveis, assim como as métricas utilizadas em experimentos de trabalhos anteriores facilitaram a avaliação de *rAck* em termos de confiabilidade e desempenho [5] [7].

III. Apache Kafka

O Apache Kafka é uma plataforma *open source* voltada para o processamento de *streams* e mensageria. A arquitetura do Kafka centra-se em um modelo produtor-consumidor, o qual sugere que mensagens sejam enviadas por produtores a um *cluster* Kafka. Os consumidores podem ser aplicações externas que coletam mensagens por meio de uma API ou clientes do próprio Kafka. Um *cluster* Kafka é composto por diversas máquinas que executam cooperativamente instâncias independentes denominadas *brokers* [1].

Quando uma mensagem é enviada a um *cluster*, a mesma é organizada em tópicos de acordo com o contexto em que foi produzida. Os tópicos Kafka são armazenados em disco nos *brokers*, sob o formato de partições. As partições são unidades paralelas cuja principal função é armazenar as mensagens. Um único tópico pode ser constituído por uma ou mais partições. Todas as partições de um mesmo tópico são distribuídas através dos *brokers* para agregar disponibilidade [1].

Por atuarem de forma descentralizada e não cooperativa, os *brokers* necessitam sincronizar os metadados dos tópicos e suas respectivas partições. Para tanto, o Kafka utiliza os serviços do Apache Zookeeper, um agente coordenador de sistemas distribuídos que oferece uma estrutura de dados em árvore para manutenção de metadados. Assim, os *brokers* mantêm a árvore do Zookeeper atualizada com as informações de cada partição e tópico que está armazenando no momento.

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 implementados pelo Kafka são o Fator de Replicação e o Reconhecimento Positivo (*Positive Acknowledgement*) ou simplesmente *ack*. O fator de replicação é uma técnica de tolerância a falhas baseada em redundância que tem por objetivo criar cópias de dados para aumentar a disponibilidade. No Kafka, as partições são replicadas e distribuídas entre os *brokers*. Dentre todas as réplicas de uma mesma partição, uma é definida como líder, sendo responsável por receber requisições de leitura e escrita. Por ser o objeto central deste trabalho, o mecanismo de *ack* é abordado em detalhe na Seção III-A, juntamente com os níveis de configuração padrão. Já a Seção III-B apresenta o Zookeeper, visto que recursos deste *framework* foram aplicados na configuração *rAck*.

A. Reconhecimento Positivo no Apache Kafka

Embora as transmissões de mensagens no Kafka sejam sob o protocolo TCP, implicando em retransmissões de pacotes, o Kafka adiciona uma camada superior para a garantia de entrega, entretanto a nível de mensagem. O mecanismo de tolerância a falhas que trata da confiabilidade empregada durante a transmissão de mensagens recebe o nome de Reconhecimento Positivo ou *ack*. Um *ack* no Kafka é uma confirmação de entrega que é retornada ao produtor da mensagem sempre que a mesma foi entregue com sucesso ao tópico destino.

O Kafka possui três níveis de configuração para *ack*, a saber: *Fire-and-Forget*, *Leader Confirmation* e *Replicas Confirmation*. Os níveis de configuração estipulam o comportamento do produtor a cada mensagem transmitida. Este comportamento está associado à confiabilidade que deve ser empregada durante a transmissão. A definição de qual nível de configuração utilizar é realizada durante a implementação do produtor.

Fire-and-Forget é o nível de configuração que tem o desempenho como objetivo. Com esta configuração, os produtores consideram uma mensagem entregue assim que é transmitida. Após a transmissão da mensagem, o produtor inicia a transmissão de uma nova mensagem sem aguardar a confirmação de entrega da anterior. O produtor que utiliza *Fire-and-Forget* não consegue identificar perdas. Além disso, as perdas de mensagens são permanentes. Entretanto, a ausência de confirmação de entrega beneficia o desempenho de produtores.

O nível de configuração *Leader Confirmation* visa alcançar confiabilidade. O produtor recebe uma confirmação de entrega sempre que a mensagem enviada foi recebida pela partição líder do tópico destino. Caso a partição líder deixe de retornar uma confirmação de entrega ao produtor, a mensagem é retransmitida pelo mesmo. Com essa configuração, produtores conseguem identificar perdas de mensagens devido à ausência de confirmação de entrega. Contudo, perdas de mensagens

ainda podem ser observadas quando a partição líder apresenta uma falha antes de replicar as mensagens em outras partições.

No nível de configuração *Replicas Confirmation*, 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. Esse nível de configuração apresenta maior confiabilidade, pois mesmo que a partição líder apresente uma falha, uma nova partição contendo a mensagem persistida é eleita líder.

B. Relação entre Kafka e Zookeeper

Como os *brokers* do Kafka atuam de forma independente, faz-se necessário recorrer ao Apache Zookeeper para realizar a coordenação e a manutenção do *cluster*. É através do Zookeeper que ocorre a eleição do *broker* Kafka controlador, responsável pela manutenção das relações de liderança entre todas as partições de todos os tópicos. Essa gestão de liderança ocorre em dois níveis: o Zookeeper é responsável pela eleição do *broker* controlador, e este, por sua vez, estabelece as relações de liderança entre as partições réplicas dos tópicos.

Além disso, o Kafka utiliza a estrutura de dados do Zookeeper na gestão dos metadados de todos os tópicos. O Zookeeper mantém uma estrutura de árvore onde cada nó é denominado *Znode*. Cada *Znode* é representado por um nome, o qual consiste no seu caminho na árvore, podendo armazenar ou não algum conteúdo. Sendo assim, o Kafka adiciona *Znodes* à árvore para armazenar os metadados de cada tópico.

Outro recurso importante do Zookeeper utilizado pelo Kafka são os *watches*, que são ações de gatilho único que notificam clientes sempre que um determinado evento ocorreu sob um *Znode* [4]. Para receber uma notificação de evento em um *Znode* específico, um cliente Zookeeper deve registrar um *watch* sob o caminho do *Znode* alvo. O Kafka utiliza *watches* para efetuar notificações aos *brokers* sempre que a formação do *cluster* for modificada.

IV. Configuração Reliable Ack (*rAck*)

A configuração *rAck* foi estruturada para agregar confiabilidade em transmissões de mensagens sem necessidade de confirmação de entrega. O *rAck* é composto por três módulos que cooperam entre si para monitorar as transmissões de mensagens, identificando e recuperando mensagens perdidas. O módulos são: produtor *rAck*, *broker rAck* e monitor. O produtor *rAck* consiste em um produtor configurado com *"rAck"*. O *broker rAck* corresponde às adaptações realizadas no *broker* Kafka de modo a receber transmissões de mensagens com o novo nível de configuração. Já o monitor é o agente responsável pelo armazenamento e recuperação de mensagens.

A Figura 1 ilustra o fluxo de comunicação entre os três módulos da configuração *rAck*. Primeiramente, um produtor é inicializado com o nível de *ack* estipulado em *"rAck"*. O produtor *rAck* transmite uma mensagem aos *brokers* para ser armazenada em tópicos e transmite também uma cópia da mesma ao monitor (etapa 1), que a armazena internamente (etapa 2) visando a recuperação futura por parte do monitor. O *broker rAck*, ao receber mensagens, altera a árvore de *Znodes* do Zookeeper (etapa 3), adicionando um *Znode* ao caminho em que o monitor registrou um *watch*. Essa alteração gera uma notificação que é retornada ao monitor (etapa 4). A partir desta notificação, o monitor descarta as cópias de mensagens, trocando-as por novas que estão sendo recebidas (etapa 5).

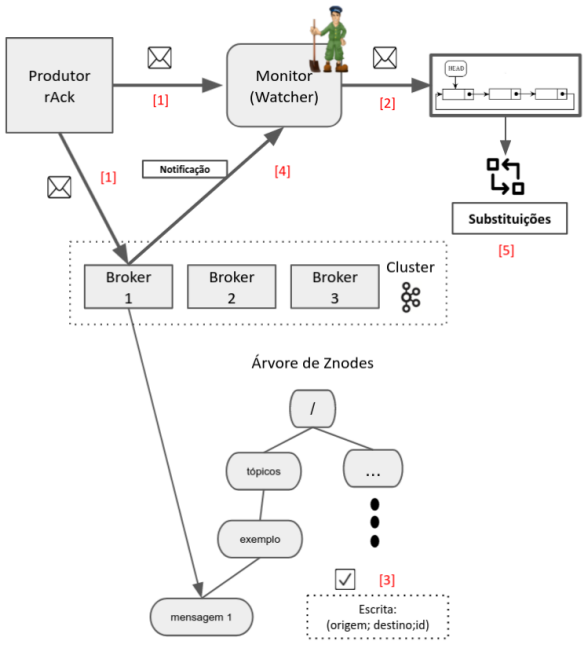

Figura 1. Cooperação entre os módulos na configuração *rAck*.

Quando não há uma notificação de recebimento para uma determinada mensagem, o monitor assume que a mesma não foi recebida com sucesso pelos *brokers*, iniciando assim a retransmissão. Por meio da coordenação entre o produtor *rAck*, o *broker rAck* e o monitor, é possível aumentar a confiabilidade em transmissões de mensagens sem reconhecimento positivo. Para utilizar a configuração *rAck*, durante a implementação do produtor que transmitirá as mensagens é necessário configurar a variável de ambiente que define o nível de configuração a ser aplicado. Na sequência, a cada transmissão de mensagem, o produtor aciona o *broker rAck* e o monitor.

Durante a operação, o produtor *rAck*, além de transmitir a mensagem para o *broker rAck*, transmite uma cópia da mesma para o monitor, que a armazena possibilitando uma eventual recuperação. A mensagem carrega as informações necessárias para que o *broker rAck* registre o recebimento da mensagem na árvore do Zookeeper. Este registro é necessário para a geração de notificações que auxiliem o monitor a discernir quais cópias foram recebidas. As informações são: origem, destino e identificação da mensagem.

Ao receber uma mensagem, o *broker rAck* verifica a configuração de *ack* definida para o processamento da mesma. Se for detectada uma configuração padrão, o *broker rAck* prossegue com o processamento convencional de cada nível. Sempre que identifica que a mensagem recebida está configurada com *rAck*, o *broker rAck* receptor, por meio do um cliente Zookeeper, cria um *Znode* e adiciona a mensagem

ao final do arquivo da partição líder do tópico que deve receber as mensagens. O *Znode* criado refere-se à mensagem recebida. O *Znode* tem o caminho na árvore que permite ser adicionado como filho do *Znode* controlado pelo monitor por meio de *watches*. Sendo assim, sempre que o novo *Znode* é criado, gera-se uma notificação que é recuperada pelo monitor. Para evitar sobrecarga, cada *Znode* criado pelo *broker rAck* apresenta um tempo de vida, o que implica em expiração após atingir o tempo predeterminado. O tempo de vida foi definido em 10 minutos, sendo este o tempo mais do que necessário para que seja gerada uma notificação para o monitor e o mesmo consiga recuperar o conteúdo deste *Znode*.

O monitor visa controlar, através de *watches*, a árvore do Zookeeper, aguardando inserções de *Znodes*. Neste contexto, uma das funcionalidades do monitor é registrar um *watch* sob o *Znode* responsável por armazenar os metadados do tópico que receberá mensagens. Ao registrar o *watch*, o monitor é notificado sempre que um novo *Znode* for adicionado como filho do *Znode* registrado nos *watches*.

Embora o monitor consiga identificar as mensagens que foram recebidas pelos *brokers rAck*, a distinção de quais mensagens foram perdidas torna-se dispensável se não há como recuperá-las. Sendo assim, o monitor possui também a funcionalidade de receber uma cópia das mensagens enviadas aos *brokers rAck*. Com essas cópias, é possível que o monitor retransmita as mensagens observadas como perdidas. As cópias das mensagens são armazenadas em uma lista circular. Inicialmente, a lista é capaz de comportar até 15000 mensagens, entretanto, esse limite pode aumentar, de acordo com a porcentagem de mensagens confirmadas de um total de mensagens enviadas. Assim, caso o monitor espere um número muito grande de mensagens, as notificações contribuem para que o tamanho da lista aumente gradativamente, possibilitando armazenar o maior número de mensagens possível.

Ao receber uma cópia da mensagem, o monitor verifica se a lista não está completa. Caso exista espaço na lista, a mensagem é adicionada ao final (etapa 2 da Figura 1). Quando recebe uma notificação do Zookeeper sobre alterações na árvore efetuadas pelo *broker rAck*, o monitor percorre a lista em busca do nó que armazena a mensagem correspondente à notificação. Ao encontrar a mensagem, realiza-se uma confirmação que define que naquele nó está presente uma mensagem recebida. Caso a lista esteja completa, o monitor busca um nó previamente confirmado, realizando a substituição da mensagem presente no nó.

O controle de qual mensagem na lista deve ser retransmitida ocorre por meio do atributo "idade" presente em cada um dos nós. A idade de um nó indica quantas vezes o monitor já passou por aquele nó em busca de um espaço para uma mensagem *backup*. Sempre que, ao percorrer a lista em busca de espaço, o monitor atravessa por um nó que não tenha sido confirmado, a idade do nó é incrementada. Ao passar por um nó cuja idade já tenha sido incrementada, o monitor inicia a retransmissão da mensagem contida no nó e, após isso, a mesma é substituída pela nova mensagem cópia recebida.

Uma vez que perdas de mensagens são consequências de redes instáveis, as retransmissões efetuadas pelo monitor apresentam *Replicas Confirmation* em *ack*. Deste modo, ao realizar retransmissões de mensagem com alta confiabilidade, evita-se que as mensagens sejam novamente perdidas. O nível *Replicas Confirmation* realiza automaticamente retransmissões configuradas pela variável de ambiente "retries", que estipula o número de tentativas para cada transmissão. Essa variável está desabilitada no monitor, pois várias tentativas em um período de instabilidade tendem a prejudicar o desempenho.

Antes retransmitir uma mensagem, o produtor armazena a mesma em uma lista separada. Caso seja recebida uma confirmação de entrega, a mensagem é removida dessa lista. Não havendo confirmação de entrega, a mensagem permanece na lista, a fim de ser retransmitida em um período futuro. Assim, é possível evitar intervalos de instabilidade na rede. As novas tentativas de retransmissão são efetuadas sempre que o monitor necessita adicionar uma nova mensagem à lista.

V. Experimentação

Para validar a proposta, elaborou-se um cenário experimental para testar a configuração *rAck* durante a transmissões. Os experimentos foram conduzidos visando comparar *rAck* aos níveis de configuração padrão de *ack* do Kafka, considerando os critérios de confiabilidade e desempenho. A experimentação ocorreu sob o Docker[1], simulando um ambiente em que contêineres receberam atribuições de um *cluster* Kafka. O ambiente foi composto por três contêineres *brokers* Kafka, três *contêineres* executando instâncias do Zookeeper, um contêiner produtor e um contêiner executando o módulo monitor.

Cada teste do experimento consistiu na execução de um produtor que enviou mensagens para serem armazenadas em um tópico distribuído entre os três *brokers* Kafka. O tópico que recebeu as mensagens era composto por três partições e utilizava fator de replicação três. Ao todo, o tópico era constituído por nove partições, dentre as quais três eram líderes e as demais eram réplicas. Em cada execução de produtor, foram transmitidas 500.000 mensagens com 500 bytes cada. Como métrica de avaliação do desempenho do produtor utilizou-se a vazão [9]. Para analisar a confiabilidade da entrega de mensagens, a métrica utilizada foi a taxa de perda de mensagens. Essa taxa indica a porcentagem de mensagens perdidas em função de uma probabilidade de perdas de pacotes ou de atraso de rede. 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 instabilidade, durante a execução do produtor foram inseridas falhas de rede. A falha escolhida foi a perda de pacotes, a qual foi introduzida no contêiner produtor por meio da ferramenta Pumba[2], que por sua vez provê uma interface que utiliza Netem[3] como *backend* – uma ferramenta que possibilita a geração de perda de pacotes em interfaces de rede.

[1] docker.com

[2] https://github.com/alexei-led/pumba

[3] https://wiki.linuxfoundation.org/networking/netem

Desta forma, adiciona-se a propriedade de perda de pacote na interface de rede do contêiner produtor, fazendo com que sua comunicação com os demais contêineres seja afetada. Com a introdução da falha, sempre que o produtor transmitir um pacote, existe uma probabilidade desse pacote ser perdido. Assim, ao forçar a perda de pacotes, também são acionadas as retransmissões de pacotes do protocolo TCP, bem como o mecanismo de *ack* do próprio Kafka. A perda de pacotes permite a comparação da confiabilidade das configurações padrão de *ack* do Kafka com *rAck*, além de identificar o impacto da perda nas transmissões.

Para cada nível de configuração de *ack* foram realizadas 20 execuções do produtor. Ao final, foram coletadas a vazão e a quantidade de mensagens recebidas com sucesso nos *brokers*. Os testes foram executados com as seguintes probabilidades de perda: 0%, 3%, 6% e 9%. Os experimentos foram conduzidos em uma máquina Dell PowerEdge T420 com 4 memórias DIMM DDR3 síncronas de 1600MHz de 4GB, 8 memórias DIMM DDR3 síncronas de 1333 MHz de 8GB, 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.

VI. Resultados e discussão

Durante a realização dos experimentos, a métrica de vazão foi coletada por meio dos *logs* gerados em cada uma das execuções de produtor. A Tabela I apresenta as médias e desvios padrões da vazão obtida a partir das 20 execuções para cada nível de configuração de *ack*. Quanto maior a vazão, melhor o desempenho para o produtor. De acordo com a Tabela I, a vazão média tende a diminuir para todas as configurações de *ack* conforme se aumenta a probabilidade de perda de pacote. Esta é uma consequência comum aos níveis de configuração testados pois ao perder pacotes, mais tempo é empregado na retransmissão dos mesmos.

A Figura 2 exibe graficamente os dados apresentados na Tabela I. Por meio do gráfico, é possível observar que a vazão média obtida com o nível *Fire-and-Forget* é maior que a obtida nas configurações para todos os percentuais de perda de pacotes. Por ser focada em desempenho, a configuração *Fire-and-Forget* não exige confirmação de entrega de mensagens, justificando sua vazão média superior se comparada as demais configurações de *ack*. O gráfico mostra ainda que a vazão observada em *rAck* é a menor dentre todos os níveis de configuração para o percentual de perda de pacote de 0%, ou seja, cenário sem falhas. A maior diferença está de *Fire-and-Forget* para *rAck*, com uma degradação de 97%. A degradação de desempenho de *Leader Confirmation* para *rAck* é de 84%, enquanto de *Replicas Confirmation* para *rAck* há uma degradação de aproximadamente 71%. A menor vazão de *rAck* no cenário sem falhas se deve à sobrecarga causada pela necessidade de transmitir duas vezes a mesma mensagem, uma para o *broker rAck* e uma cópia para o módulo monitor.

Ao observar os cenários com falhas, nota-se que os níveis padrão de *ack* do Kafka são progressivamente prejudicados. Da probabilidade de perda de 3% para 9%, o nível *Fire-and-Forget* vai de uma vazão de cerca de 4527 para 1252 mensagens por segundo, uma degradação de 72% no seu desempenho. *Leader Confirmation* diminui sua vazão de cerca de 71 para 23 mensagens por segundo, degradação de 67%, já *Replicas confirmation* tem sua vazão reduzida de cerca de 24 para 7 mensagens por segundo, representando degradação de desempenho de aproximadamente 70%. Enquanto os níveis padrão para *ack* no Kafka apresentam diminuições de, no mínimo, 67% em suas respectivas vazões, a configuração *rAck* vai de uma vazão de cerca de 190 mensagens por segundo em chance de perda de 3% para 181 mensagens por segundo em 9%, representando uma degradação de apenas 4%.

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 desempenho inferior quando se introduz chances de perda de pacotes. Isto porque produtores que utilizam a configuração *rAck* não interrompem o fluxo de envio 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 prejudicadas 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.

Ao coletar a quantidade de mensagens presente nos tópicos, identificou-se que em todas as 20 execuções, *Leader Confirmation*, *Replicas Confirmation* e a configuração *rAck* foram capazes de entregar todas as 500.000 mensagens. Diferentemente, no nível *Fire-and-Forget* houve perdas nos percentuais de 6% e 9%. Em 6% a média foi de 191 mensagens perdidas por execução, o que representa 0,04% das mensagens enviadas. Já em 9% a média foi de 1031 mensagens perdidas, sendo um percentual de 0,2% do total de mensagens enviadas.

Apesar de recuperar todas as mensagens, observou-se que

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

Nível de Configuração de ack	Percentual de perda de pacote a cada envio							
	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

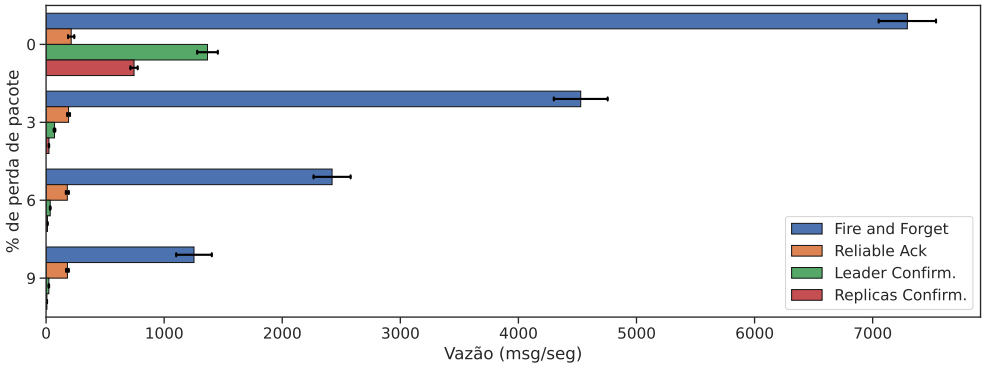

Figura 2. Gráfico da Vazão dos níveis de configuração de ack.

a recuperação exercida pelo monitor não é capaz de garantir a ordem na entrega de mensagem. Isto se deve ao fato de que o produtor não interrompe sua transmissão sempre que há perda, já que não consegue identificá-la. Assim, tanto a identificação de perdas quanto a retransmissão são efetuadas pelo monitor paralelamente à transmissão do produtor. Dessa forma, as mensagens podem chegar aos *brokers rAck* em ordem diferente da qual inicialmente foram enviadas pelo produtor *rAck*. Isso não ocorre com os níveis *Leader Confirmation* e *Replicas Confirmation*, uma vez que nestes níveis o produtor aguarda a confirmação de entrega antes de enviar a próxima mensagem.

Foi realizada uma análise dos tempos de retransmissão de mensagens efetuados pelo monitor. Para tanto, coletou-se o tempo total de retransmissão a cada execução do monitor para todos os percentuais de perda de pacote. Observou-se que, conforme progride o percentual de perda, o número de retransmissões efetuadas pelo monitor aumenta. Para a chance de 3% foram demandados, em média, 69 segundos em retransmissão. Para 6% a média foi de 315 segundos e para 9% a média foi de 1077 segundos. O aumento da chance de 3% para 9% foi de aproximadamente 1448% no tempo de retransmissão total. Esse aumento ocorre pois, além de existirem mais mensagens a serem retransmitidas, as retransmissões são efetuadas em alta confiabilidade com *Replicas Confirmation*.

VII. Considerações Finais

O presente artigo apresentou uma configuração de *ack* no Apache Kafka, o *rAck*. A configuração *rAck* baseia-se na cooperação de três módulos: o produtor *rAck*, o *broker rAck* e o monitor. A coordenação destes módulos visa agregar confiabilidade na transmissão de mensagens sem reconhecimento positivo. O objetivo da configuração *rAck* é contribuir com o cenário de transmissões de mensagens em redes instáveis. Neste cenário, os níveis de *ack* padrões forçam o usuário a priorizar desempenho e assumir perdas de mensagens ou optar por confiabilidade, aceitando degradação no desempenho.

A configuração *rAck* foi submetida a uma experimentação para coletar métricas de desempenho e confiabilidade, objetivando comparações do nível proposto com as configurações padrão do Kafka. Na análise dos resultados obtidos, *rAck* demonstrou ser uma alternativa viável para a transmissão de mensagens em redes sujeitas a perda de pacotes, apresentado pouca variação em sua vazão e tempo de execução conforme aumenta a instabilidade na rede.

Para trabalhos futuros, pretende-se estender o cenário de testes para experimentos com atrasos na entrega de pacotes, visando observar o comportamento da configuração *rAck*. Como as transmissões com *rAck* não garantem a ordem de entrega, pretende-se implementar essa garantia através de *buffers* de preordenação de mensagens no *broker rAck*. Por fim, identificou-se que o monitor utiliza recursos computacionais que em níveis padrões do Kafka estariam livres para serem alocados para mais instâncias de *brokers*. Ainda, o monitor apenas é capaz de atender as demandas exclusivas de um único produtor *rAck*. Deste modo, trabalhos futuros também compreendem comparações da utilização de recursos computacionais entre os níveis padrões de *ack* e a configuração *rAck*, além de buscar formas de prover escalabilidade ao monitor.

Referências

[1] N. Narkhede, G. Shapira, and T. Palino, *Kafka: The Definitive Guide*, 1st ed. Cambridge: O'Reilly Media, 2017.

[2] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, G. Morgan, and R. Ranjan, *A study on the evaluation of HPC microservices in containerized environment*. Wiley Online Library, 2019.

[3] A. B. A. Alaasam, G. Radchenko, and A. Tchernykh, "Stateful stream processing for digital twins: Microservice-based kafka stream dsl," in *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*, 2019, pp. 0804–0809.

[4] F. Junqueira and B. Reed, *Zookeeper: Distributed Process Cordination*, 1st ed. Cambridge: O'Reilly Media, 2013.

[5] H. Wu, Z. Shang, and K. Wolter, "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)*, 2019, pp. 394–397.

[6] P. Bhimani and G. Panchal, "Message delivery guarantee and status update of clients based on iot-amqp," in *Intelligent Communication and Computational Technologies*. Singapore: Springer, 2018, pp. 15–22.

[7] S. Lee, H. Kim, D. Hong, and H. Ju, "Correlation analysis of mqtt loss and delay according to qos level," in *The International Conference on Information Networking 2013 (ICOIN)*, 2013, pp. 714–717.

[8] J. E. Luzuriaga, M. Perez, P. Boronat, J. C. Cano, C. Calafate, and P. Manzoni, "A comparative evaluation of amqp and mqtt protocols over unstable and mobile networks," in *12th IEEE Consumer Communications and Networking Conference (CCNC)*, 2015, pp. 931–936.

[9] P. Le Noac'h, A. Costan, and L. Bougé, "A performance evaluation of apache kafka in support of big data streaming applications," in *2017 IEEE Int. Conference on Big Data (Big Data)*, 2017, pp. 4803–4806.