Apache Kafka

Felipe Archanjo da Cunha Mendes[1], **Breno Farias da Silva**[1], **João Henrique Gouveia**[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão – PR – Brazil

felipemendes.1999@alunos.utfpr.edu.br,

brenofarias@alunos.utfpr.edu.br,

joaosperandio@alunos.utfpr.edu.br

Abstract. *This article has as it's objective the study and further explanation of kafka fundamentals per the instruction of our Dabase II class professor André Luis Schwerz, such as the event driven communication between modern systems, topics, partitioning, clusters, brokers escalability and open source integrations ecosystem.*

Resumo. *Este artigo tem por objetivo o estudo e a fundamentação de pilares teóricos do kafka, por instruções de nosso professor de Banco de dados II André Luis Schwerz, tais como a comunicação baseada em eventos entre sistemas modernos, tópicos, particionamento, clusters, brokers, escalabilidade e o ecossistema de integrações de código aberto.*

1. Introdução

O Apache Kafka é um sistema Open-Source trabalhado de forma distribuída com o objetivo de trabalhar com o stream de dados. Em outras palavras, há dados disponíveis que podem ser passados de um sistema para o outro, armazenados de alguma forma que esses dados possam ser consultados posteriormente e até mesmo transformados para serem distribuídos para outros sistemas. Nesse sentido, para realizar essas funcionalidades, recomenda-se utilizar o Apache Kafka.

2. Importância

Hoje em dia quase todos os sistemas são orientados a eventos. Imagine, por exemplo, um e-commerce no qual diversas ações são feitas nele diariamente em que há o disparo de eventos, tais como a realização de uma compra, a adição de um item ao carrinho ou uma simples avaliação de produtos. Portanto, ao se analisar com um olhar mais próximo às tecnologias existentes, os eventos estão em toda a parte.

Muitos dos eventos gerados são importantes para a geração de dados históricos ao sistema para que possam ser manipulados posteriormente. Imagine uma situação em que a todo momento são executadas transações bancarias como uma abertura ou fechamento de conta, adição de crédito, debito, criação de cartão e entre outras ações. A todo momento serão disparados eventos que, ao armazená-los no banco de dados, ficará disponível o histórico completo de tudo que aconteceu com esse usuário. Neste sentido, o fato desses dados serem muito importantes e preciosos para a empresa devido à possibilidade de auditoria, estatística, previsões ou agrupamento, torna o Apache Kafka um diferencial

de grande importância. Com isso, ele é responsável não só por manipular, mas também por receber todos os eventos que acontecem no sistema e armazená-los para posterior utilização.

Ademais, em relação a sua capacidade de núcleo, o Apache Kafka consegue processar muita informação com uma latência muito baixa. Não só isso, mas ele escalável, possui grande disponibilidade e é extremamente tolerante a falhas.

Por fim, em relação a seu ecossistema, por padrão, é possível processar o stream de dados. Em outras palavras, é possível receber informações do kafka e processá-las, transformar essas informações e devolver ao kafka novamente ou enviar para outros sistemas de forma totalmente integrada.

3. Como o Kafka funciona?

O apache Kafka funciona na forma de kluster, ou seja, há diversas máquinas rodando ele, sendo que cada uma delas é chamada de broker. Em outras palavras, o Kafka é um kluster com diversos brokers com seus respectivos banco de dados.

De forma geral, há sistemas produtores responsáveis por produzir determinados dados encaminhados ao Apache Kafka que, por sua vez, irá distribuí-los e armazená-los nos brokers. Por outro lado, há sistemas consumidores interessados em adquirir essas informações que foram para o Kafka.

Além disso, há os chamados Zookepers, que são sistemas de Service Discovery com a função de orquestrar ou gerenciar os brokers que o Kafka está rodando, para garantir que tudo no sistema esteja em harmonia.

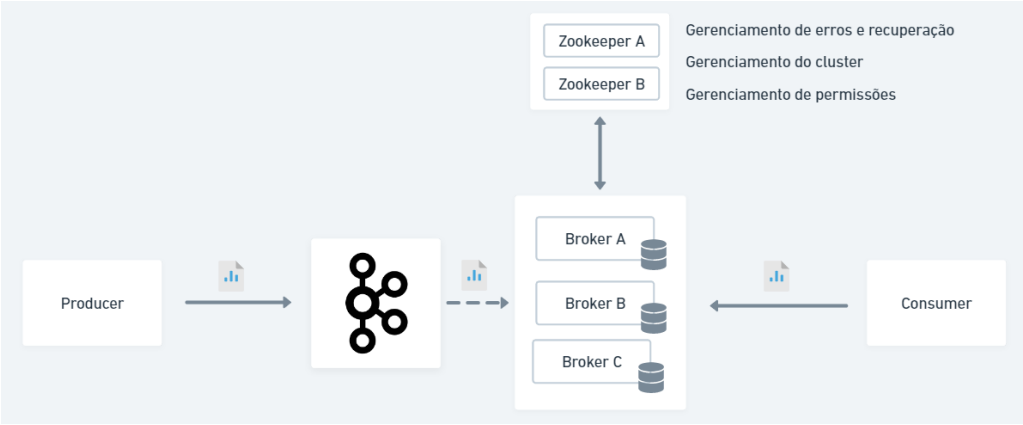

Figura 1. Funcionamento geral do Apache Kafka

4. Tópicos

Os tópicos são como canos ou canais de comunicação (pipes) onde dados são direcionados e armazenados no apache Kafka. Podem ser criados vários tópicos no Apache Kafka, tais como para eventos de produtos vendidos, itens adicionados ao carrinho ou avaliações de produtos. Em outras palavras, quando mensagens são produzidas pelos produtores, elas são enviadas para um tópico que ira gravar essas informações no broker, enquanto que o consumidor vai ficar lendo esse tópico para que ele consiga processar toda mensagem que chegar nele. Em relação a isso, pode haver vários consumidores lendo o mesmo tópico, ou seja, sistemas diferentes buscando o mesmo conjunto de informações no apache Kafka.

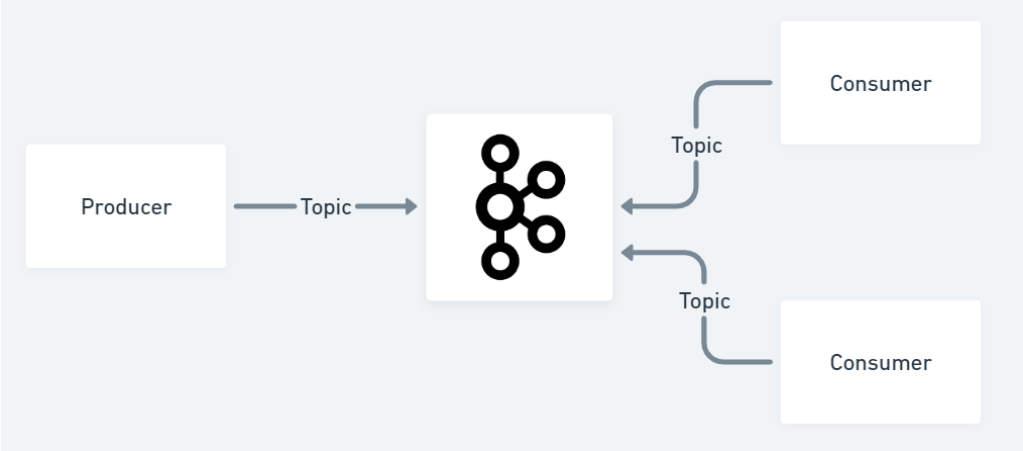

Figura 2. Tópicos

4.1. Partições e Segmentos

Duas outras definições importantes para se entender o funcionamento dos tópicos são as **partições de tópico** e os **segmentos**. Considere um tópico *sales* de vendas, ou seja, todas as vendas que estão sendo feitas pelo e-commerce é entregue a este tópico. O apache Kafka divide o tópico em partições, sendo que cada uma delas recebe parte dos dados. Imagine que determinado usuário comprou um celular, um computador e um video-game, sendo que o celular foi para a partição 1, o computador para a partição 2 e o video-game para a partição 3. Portanto, sempre que for criado um tópico é necessário atribuir um valor para definir com quantas partições se vai trabalhar.

Em relação a isso, quando determinado dado alcança alguma partição, ela cria diversos segmentos com esses dados, como se fossem diversos arquivos ou banco de dados com essas informações.

São nesses segmentos que são definidos os períodos de retenção da mensagem que determina o tempo que determinada mensagem ficará armazenada no apache Kafka. Com isso, expirado o tempo determinado a mensagem será excluída automaticamente. Por padrão, o tempo de retenção do Kafka é de uma semana. No entanto, é possível definir que determinada mensagem fique armazenada pra sempre ou apenas durante alguns minutos.

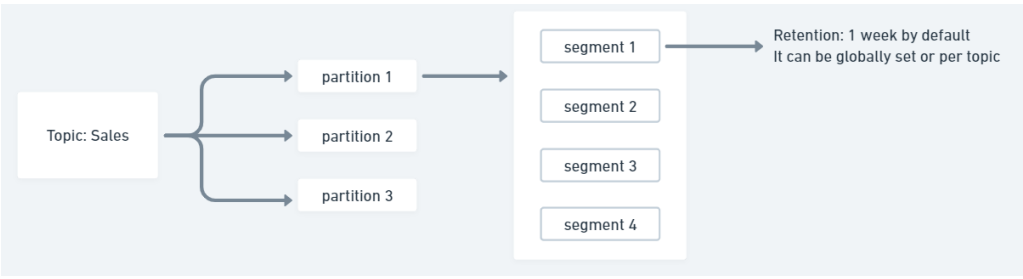

Figura 3. Partições e segmentos

4.2. Logs de Tópico

É preciso notar que o tópico, de forma geral, funciona como se fosse um grande log, no qual as informações são gravadas nele uma atras da outra.

Quando se manda uma mensagem a um tópico e ela é enviada a alguma partição, essa mensagem é armazenada em um bloco chamado *offset*. Cada mensagem fica armazenada em um desses blocos. Portanto, essas mensagens sempre ficam enfileiradas dentro de uma mesma partição.

Imagine que há dois sistemas, sendo que ambos estão lendo a mesma partição, mas em índices diferentes. Caso um desses sistemas saia do ar, o Apache Kafka guarda o índice em que o determinado sistema parou. Isso prova o quão o Apache Kafka é capaz por gerenciar e guardar todas essas informações.

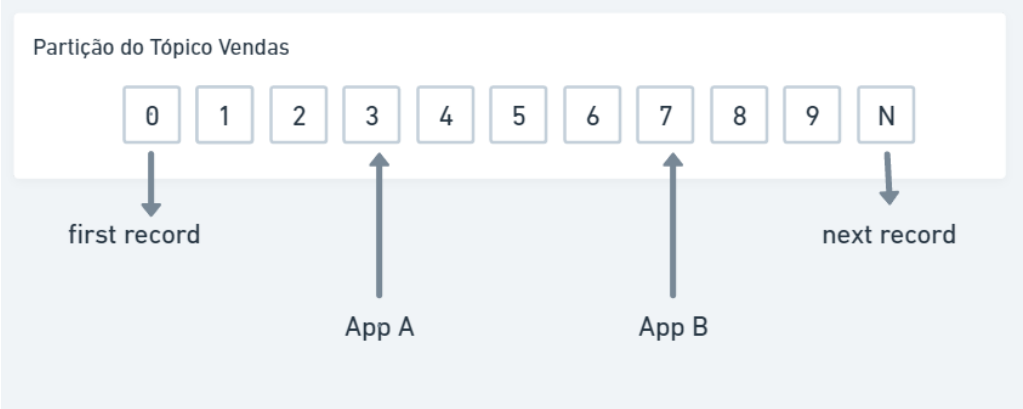

Figura 4. Log de tópico

4.3. Anatomia do offset

Cada offset tem um tipo de anatomia, ou seja, uma estrutura de mensagem que será armazenada no Kafka. Ela é dividida em:

Headers (opcional): Metadados ou informações que vão ajudar o sistema a tomar alguma decisão logica.

Key (opcional): Chave que ajuda a manter o contexto de uma mensagem. Imagine o exemplo do tópico *sales* (de vendas), sendo que sempre que ocorre uma venda, a mensagem é enviada para esse tópico. Nesse sentido, sempre que acontece uma venda há uma mensagem especifica que, ao mandar, vai ser especifica para o sistema de entrega (shipping). Com isso é atribuído *shipping* à key. Quando os sistemas começam a ler as mensagens, eles podem processas as mensagens filtrando suas keys, ou seja, pode haver consumidores que irão processar apenas as mensagens cuja key seja *shipping*. Caso a key seja qualquer outra coisa, a leitura pode ser descartada.

Value: É o conteúdo da mensagem (payload). Nesa área é possível armazenar um json, um arquivo texto, protocol buffers e entre outros.

Timestamp: Toda vez que esse offset for registrado, o Kafka irá gerar um timestamp automaticamente.

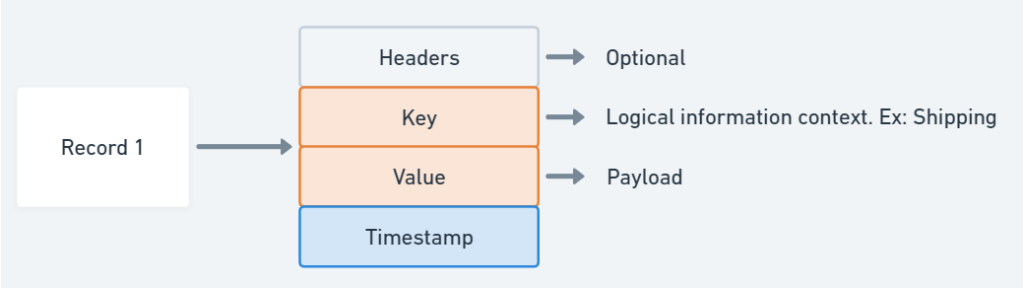

Figura 5. Anatomia do offset

4.4. Tópicos compactos

Imagine que se tem um sistema de meteorologia que registra a mudança de temperatura no tempo e, cada vez que se percebe essa mudança, o dado é enviado ao Kafka. Considere que em Campinas está 27°C e, após 10 min, se encontra em 25°C e, mais tarde, 22ºC. Com isso, o Kafka irá registrar todas essas informações. No entanto, há momentos em que é necessário apenas verificar a temperatura atual, sendo desnecessário ter que ler todas essas mensagens. Para isso, tem-se o log compactado.

O log compactado é capaz de pegar o valor atual que se encontra a sua mensagem. Imagine um tópico com os registros de índice 0-9. Considere as keys **a**,**b** e **c**, sendo que **a** se refere a campinas, **b** a Curitiba e **c** a Campo Mourão. A partir de um determinado tempo, o Kafka irá receber os dados meteorológicos das 3 cidades sempre que houver mudança de temperatura, armazenando-os em um determinado tópico. No entanto, é possível se ter acesso a um log compactado no qual é armazenado os registros mais recentes das mensagens de determinada keys. Com isso é possível ter acesso às temperaturas atuais das 3 cidades sem ter que ler todas as mensagens.

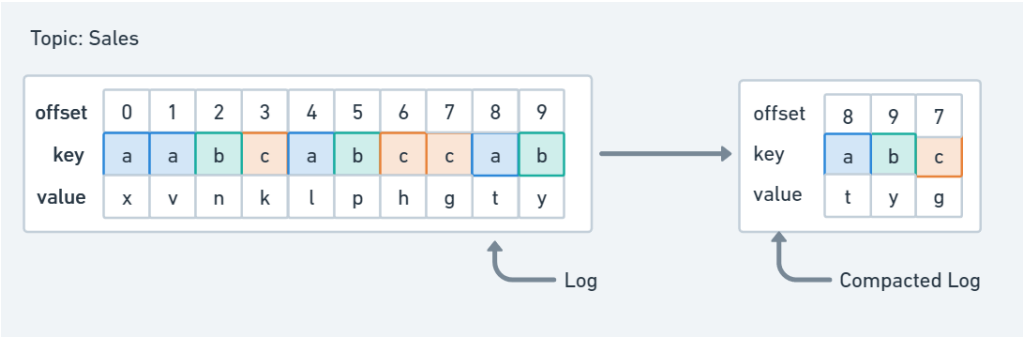

Figura 6. Tópico compacto

4.5. Partições distribuidas

As partições dos tópicos são armazenadas de forma distribuída entre os brokers no apache Kafka. Independetemente dos tópicos, o apache Kafka sempre ira tentar distribuir suas partições entre os brokers.

Considere os tópicos *sales* e *clients*. Neste exemplo a partição 1 do tópico sales está no broker A, a partição 2 no broker B e a partição 3 no broker C. Por outro lado, a partição 1 do tópico clients está no broker C, a partição 2 no broker B e a partição 3 no

broker A. Se em determinado momento, o broker A ficar fora do ar, os dados da partição 3 e 2 do tópico *sales* continuarao no ar, assim como os dados da partição 1 e 2 do tópico *clients*, fazendo com que os consumidores que estavam consumindo o broker A percam apenas os dados contidos nele.

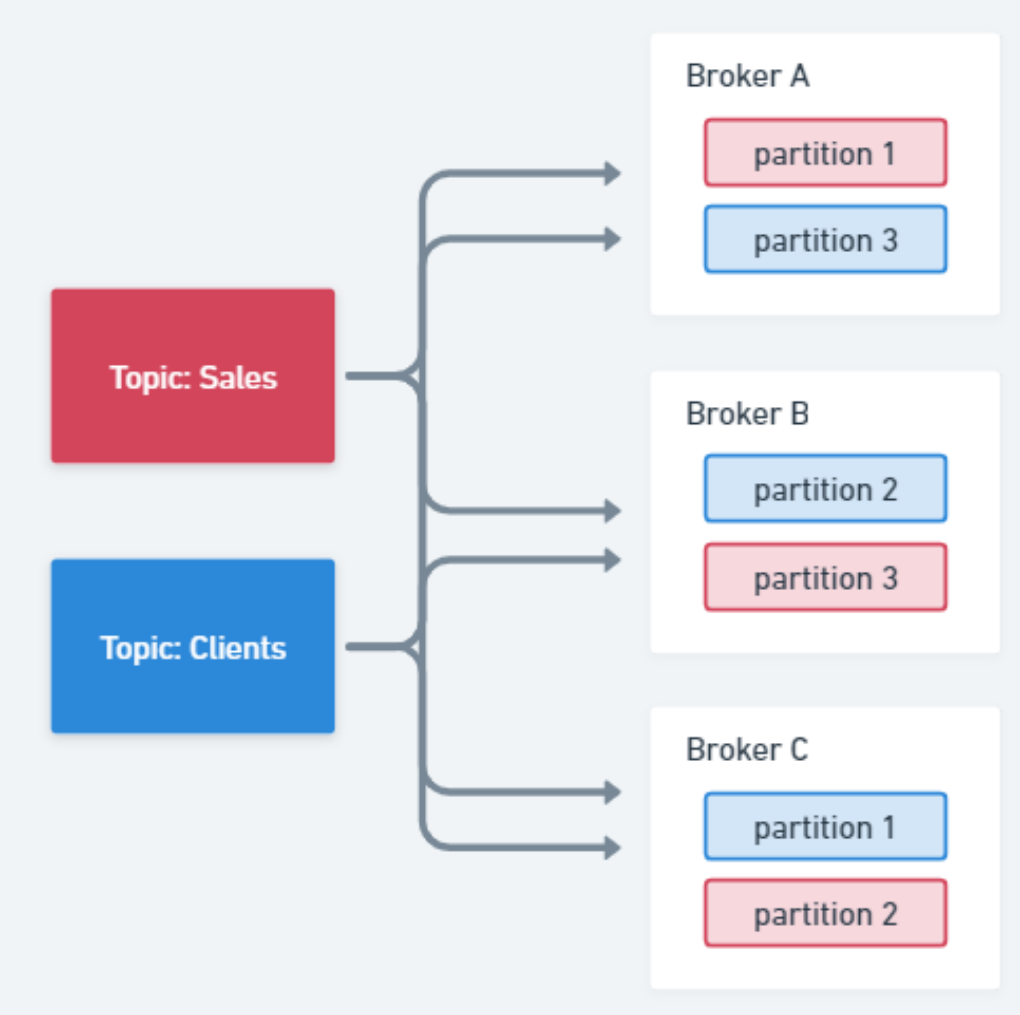

Figura 7. Partições distribuidas

Entretanto, o Kafka deve se manter resiliente exatamente para evitar o problema da perda de dados. Para isso, o Kafka possui um mecanismo chamado **Replication Factor**. Seu objetivo visa garantir que se tenha replicas de uma partição em outros brokers para caso um broker caia, as informações não sejam perdidas.

Imagine a definição de um tópico sales divido em três partições, 3 brokers e um replication factor de valor 2. Isso significa que haverá, pelo menos, duas copias das partições em brokers distintos.

Nesse exemplo, tem-se a partição 1 no broker A e outra cópia dela no broker C. Da mesma forma, percebe-se que no broker B há a partição 2, ao mesmo tempo que se tem uma cópia dessa partição no broker C. Por último, no broker A tem a partição 3, assim como também no broker B. Isso significa que sempre haverá uma réplica de uma partição em outro broker.

Diferente do exemplo sem o replication factor, caso o broker A saia do ar, o consumidor que estava recebendo os dados da partição 1 ou 3 não ficará sem os dados. O kafka, automaticamente, vai perceber que o broker A esta fora do ar e vai direcionar os consumidores a utilizarem a partição 1 do broker C e a partição 3 do broker B. Mas no momento que o broker A voltar, o Kafka vai, automaticamente, redistribuir essas partições.

Portanto, utilizando o replication factor, caso um broker caia, dados nunca serão perdidos sempre haverá uma replica desses dados em outro broker, sendo que a quantidade de replicas é definido ao se atribuir um valor ao replication factor.

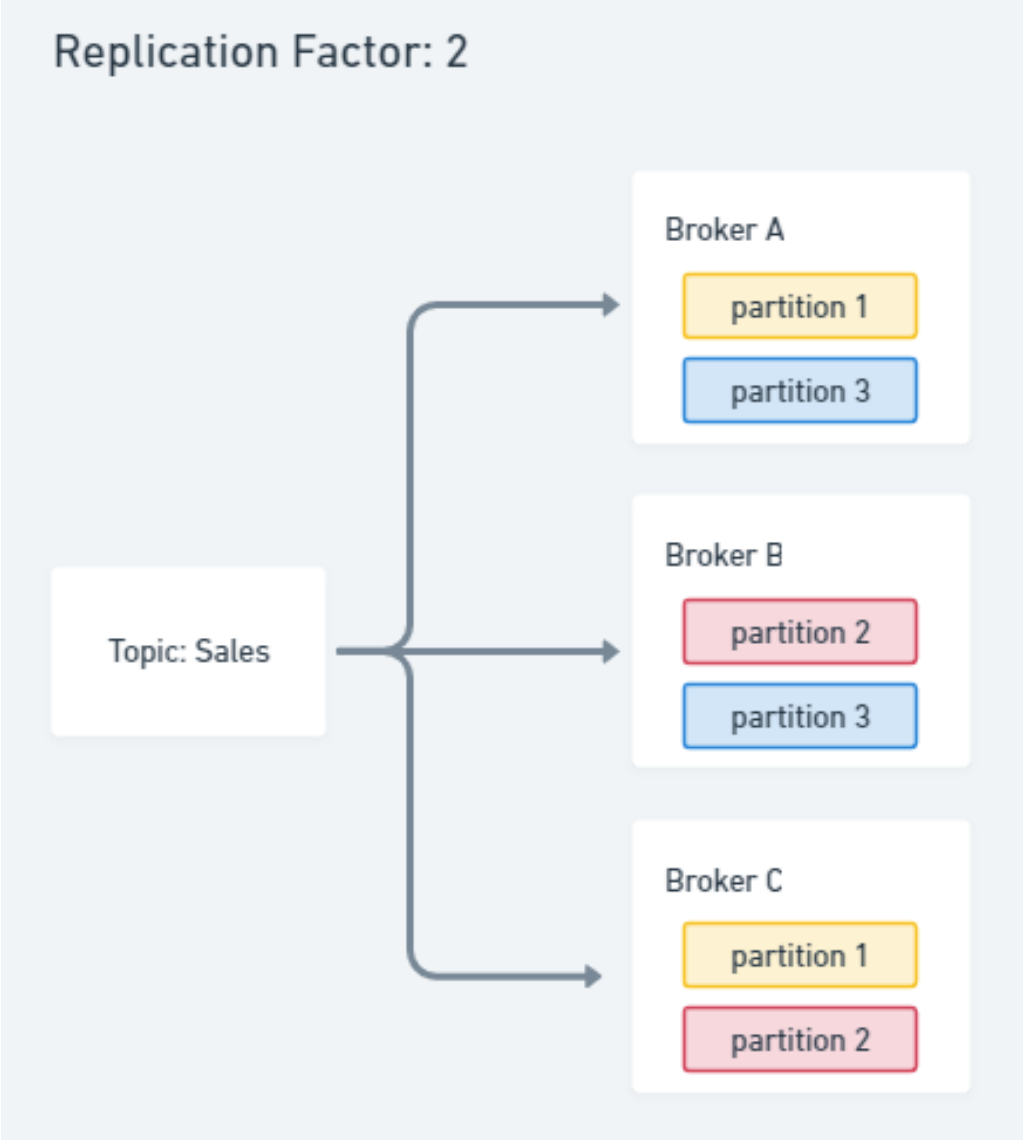

Figura 8. Replication factor = 2

4.6. Entrega

O Kafka não gera uma regra principal para entregar as mensagens em cada partição.

Imagine a produção de uma mensagem sem key que é enviada a partição 1 do tópico sales. A proxima mensagem será enviada a partição 2, a próxima na partição

3, a próxima na partição 1 e assim por diante, respeitando o algoritmo Round Robin, mandando as mensagens da forma mais distribuída possível.

Cada vez que for enviada uma mensagem, ela será inserida em partições diferentes caso não possua uma key. Por outro lado, se uma mensagem tiver uma key, ela sempre estará na mesma partição.

Imagine a criação de uma mensagem com a key *vendas-campinas* que foi adicionada na partição 1. Após o envio de várias outras mensagens sem utilizar essa mesma chave, quando uma nova mensagem utilizar a key *vendas-campinas* ela será adicionada na partição 1, uma vez que o Kafka esta armazenando todas na partição 1 que possuem essa mesma key.

Como o Kafka trabalha com partições, não é possível garantir a ordem de leitura das mensagens. No entanto, existem casos que seria de extrema importância ler as mensagens de forma ordenada. Nesse caso, é possível manter a ordem das mensagens quando se trabalha com a mesma key, uma vez que sempre que se quiser ler uma mensagem com a key *vendas-campinas*, todas essas mensagens estarão de forma ordenada na partição 1.

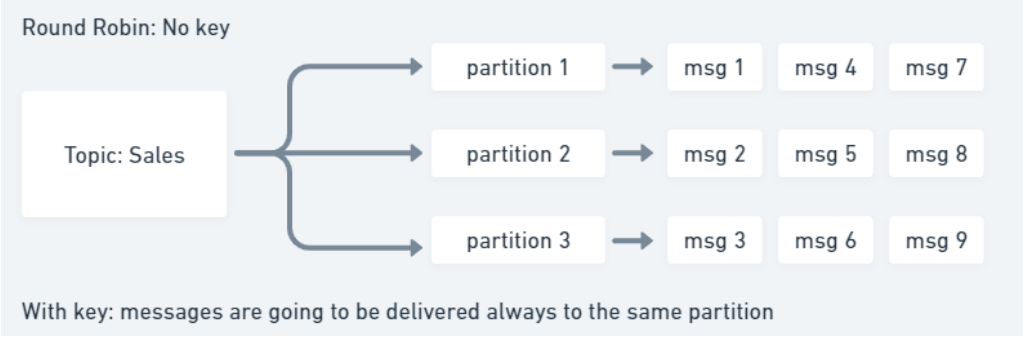

Figura 9. Entrega de mensagens

4.7. Partição líder

Quando foi discutido, anteriormente, o uso do replication factor, foi explicado a geração de cópias das partições em diversos brokers. Uma das cópias sempre será a líder e as outras chamadas de *followers*. Toda vez que alguém estiver consumindo uma mensagem, ela sempre vai consumir da partição líder, enquanto que as followers ficarão inativas, apenas sincronizando informações.

Caso o broker A, que possui a partition 1 líder, caia, o kafka irá transformar uma das partitions 1 followers em líder temporario, para que o consumidor que estava lendo a partition 1 nao perca os dados. Caso o broker A volte, o Kafka pode reestabeler o status de líder novamente ao partition 1 desse broker.

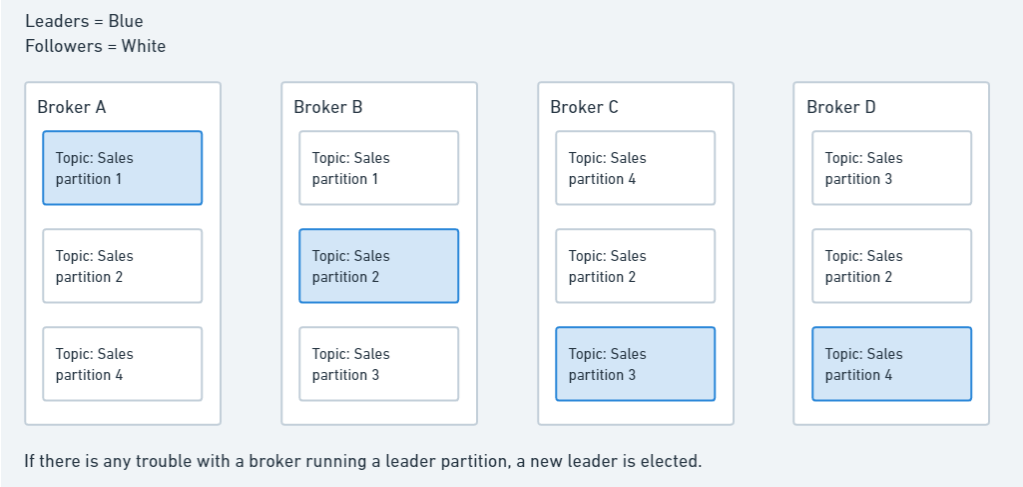

Figura 10. Partições lider

5. Produtores

5.1. Processo de entrega

Uma mensagem é criada, sendo especificado o tópico, valor e chave correspondentes. Logo após, na linguagem que for utilizada, java, python, etc, usa-se o comando ***send()***, o qual irá tentar realizar o envio da mensagem. Em seguida, dentro do apache kafka, há um elemento chamado *serializer* ou serializador, o qual é um processo que converte objetos em bytes. O serializador apenas trata de padronizar a mensagem, pois ela pode ser, por exemplo, em formato JSON, *plain text*, *protocol buffers*, assim ele converte-as de modo a serem armazenas em um formato padrão no Apache Kafka. Ainda mais, acontecerá a seleção sobre em qual partição a mensagem será gravada, por meio do ***partitioner*** ou particionador, por fim, a gravação definitiva dos dados.

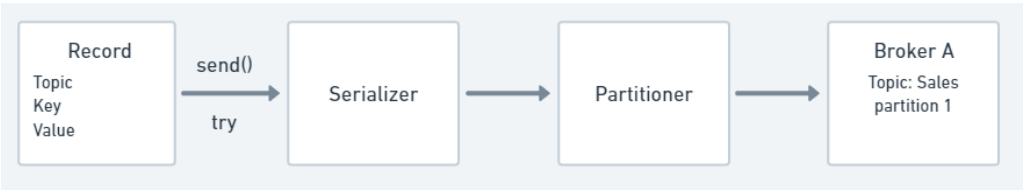

Figura 11. Processo de entrega

5.2. Garantia de entrega

Um processo produtor irá ler dados de alguma fonte, a qual pode ou não ser local, depois irá gravar no sistema de mensagens pela rede. O sistema de mensagem deve ser capaz de persistir a mensagem e, no caso do Kafka, seguindo o atributo *Replication factor* o qual, por padrão estará definido como 1, pode ou não ter várias cópias distribuídas, gerando redundância. Eventualmente haverá consumidores que irão pesquisar no sistema de mensageiria e receber mensagens, as quais irão aplicar algum tipo de ação nelas. Dessa forma, é possível dividir o sistema de garantia de entrega em 4 partes visto que, como qualquer:

Sem garantia - Como o próprio nome diz, nada é garantido, o que implica que a mensagem pode ser processada nenhuma, uma ou várias vezes.

No máximo uma vez - De forma a tentar evitar *overhead* em processar a mesma mensagem várias vezes, adota-se essa prática como uma ”lei do menor esforço”. Sendo assim, a mensagem poderia não ser processada, mas caso isso ocorra, iria acontecer no máximo uma vez.

Exatamente uma vez - Nesse caso, é garantido que os consumidores irão processar todas as mensagens, pois nenhuma mensagem não será processada, além de que, quando forem, não será mais de uma vez, evitando *overhead*.

Pelo menos uma vez - Os produtores irão receber e processar todas as mensagens, todavia o *overhead* ainda não é evitado.

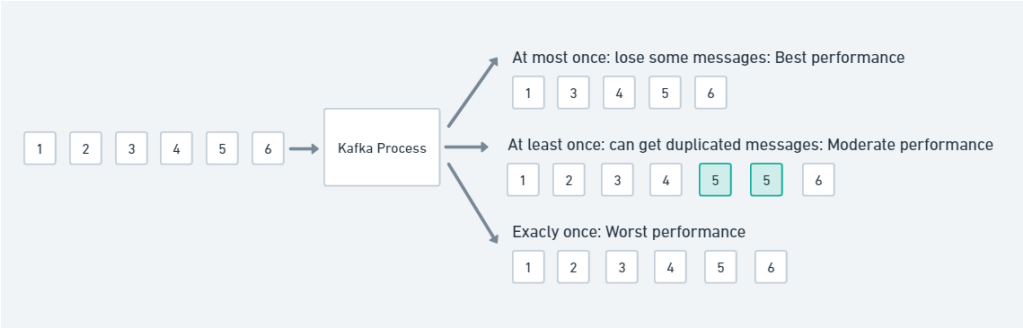

Figura 12. Garantia de entrega pt1

O motivo da divergência entre as formas do sistema de garantia de entrega se deve pelo fato de aplicações grandes irão, eventualmente, sofrer com algum tipo de falha, sendo ela catastrófica ou não. Assim sendo, a implementação desse sistema de forma alguma é dada como trivial. Outro fator relevante no sistema de garantias do Apache Kafka é o ACK, ou *acknowledged*. O ack nada mais é do que uma mensagem de confirmação, a qual pode ser definida das seguintes formas:

Ack 0 - Nesse caso, o produtor não espera o Kafka enviar uma mensagem de confirmação se o Kafka gravou a mensagem com sucesso. Isso implica em menos *overhead* pois o Kafka não precisa parar de processar mensagens para enviar um sinal de confirmação. O uso do ack 0 resulta em mais poder de processamento de mensagens.

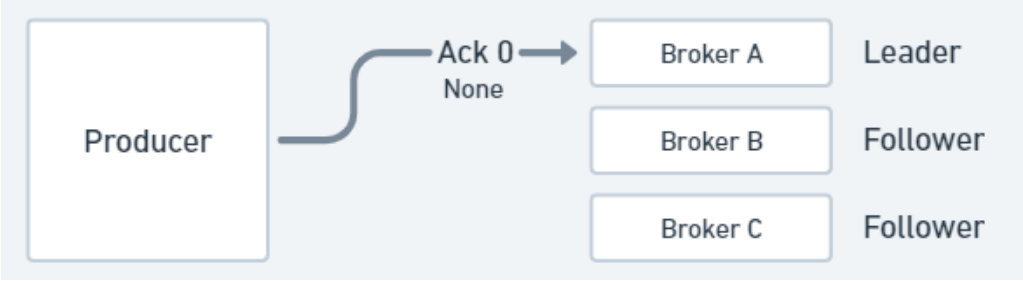

Figura 13. Ack 0

Ack 1 - Nesse caso, o líder de uma partição, ao receber uma mensagem, envia uma mensagem de confirmação para o produtor, implicando em uma garantia na entrega da mensagem. Ainda que o produtor tenha recebido uma confirmação de que a mensagem foi gravada em um *broker* x, contudo, assim que o Kafka retornou sobre a confirmação de

gravação dos dados, o mesmo *broker* sofreu com alguma falha. Este fato implica que o a mensagem não foi replicada para garantir a propriedade de durabilidade do ACID. O uso do ack 1 implica em uma velocidade moderada, porém com algum nível de segurança sobre os dados.

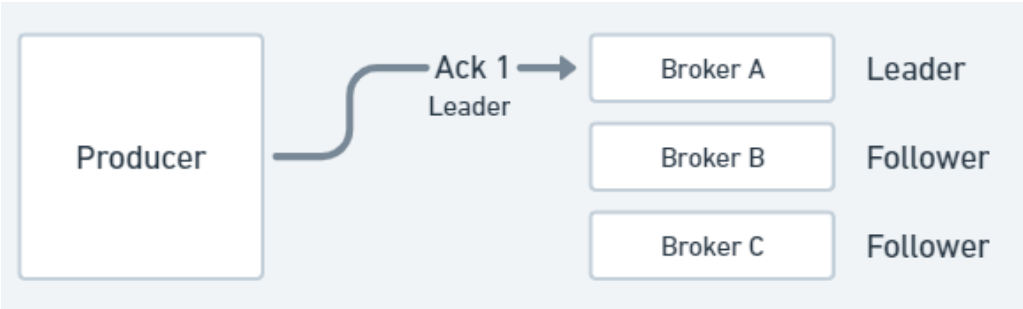

Figura 14. Ack 1

Ack -1 ou ALL - Nesse caso, replicamos o caso do ack 1, todavia a mensagem de confirmação só é enviada após a mensagem ter sido replicada em diferentes partições, o que resulta na garantia da propriedade de durabilidade. O uso do ack 1 implica no menor poder de processamento de mensagens.

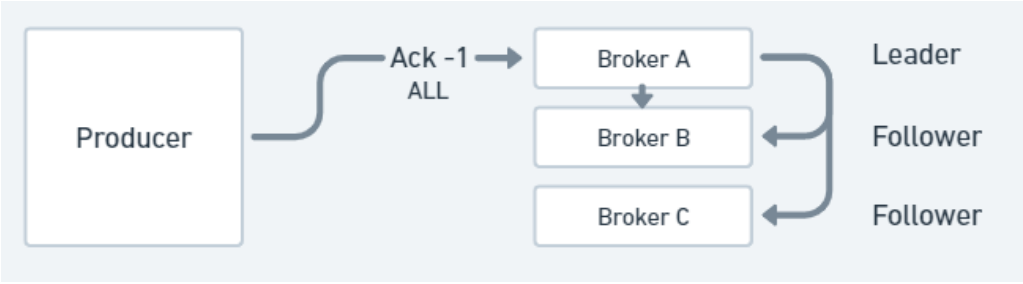

Figura 15. Ack -1 ou ALL

5.3. Produtores idempotentes

Idempotência nada mais é do que a possibilidade de executar a mesma operação várias vezes sem alterar o resultado. O uso de produtores idempotentes aumenta o nível de confiabilidade no produtor, pois garante que não haverá duplicatas em nível de partição. O Apache Kafka permite configurar essa propriedade por meio da instrução de ativação da idempotência ***enable.idempotence=true***, configurar o modo de uso do ack ***acks=all*** e determinar o número máximo de tentativas de envio da mensagem de ack para o produtor. ***max.in.flight.requests.per.connection= 3***

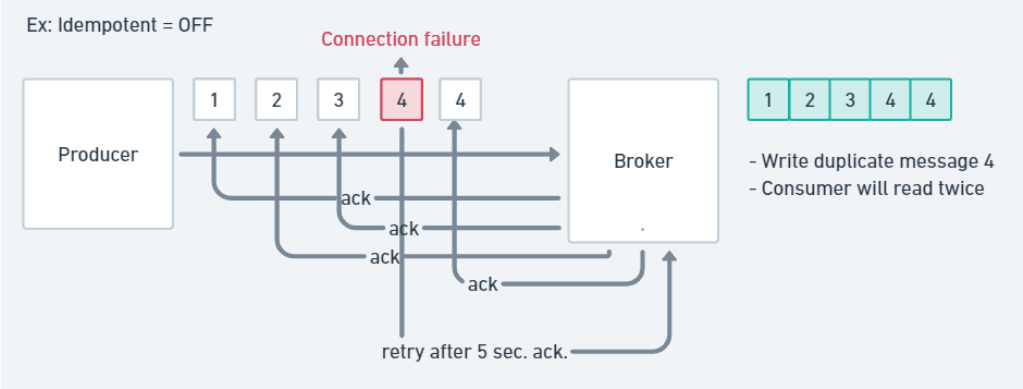

Figura 16. Produtores idempotentes OFF

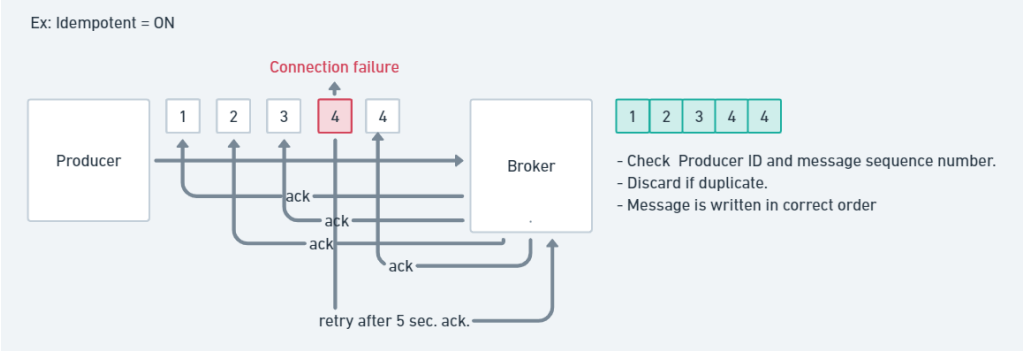

Figura 17. Produtores idempotentes ON

6. Consumidores

De maneira geral, os consumidores são processos responsáveis por ler as mensagens do Kafka, cujo processo de leitura sobre os tópicos, por padrão, é baseado em uma espera ocupada.

Imagine que um tópico sales com N partições recebe uma mensagem de um produtor qualquer. Essas mensagens serão distribuídas entre as diferentes partições em diversos brokers. Dessa forma, considere que um consumidor quer receber todas mensagens do tópico sales. Com isso, ele ficará lendo de todas as partições desse tópico.

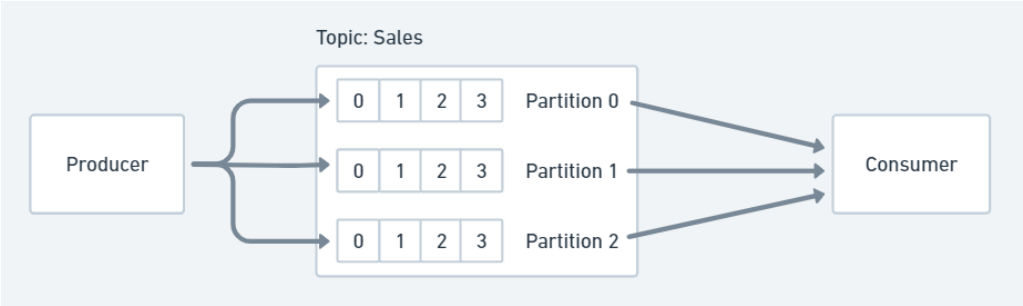

Figura 18. Consumidor básico

6.1. Grupo de consumidores

Da mesma forma que o exemplo anterior, pode-se imaginar um produtor que envia uma mensagem a um tópico que irá distribui-la entre suas partições. No entanto, é possível ter dois consumidores do mesmo grupo lendo essa mensagem. Nesse sentido, o objetivo do grupo de consumidores é deixar a leitura de forma mais paralelizada.

Imaginando que se tem 3 partições de um tópico sales e dois consumidores em determinado grupo, ao fazer com que um deles leia duas partições e o outro uma partição, o processo de leitura ficará muito mais rápido do que sendo feito por apenas um consumidor, apesar de um dos consumidores ficar mais sobrecarregado que o outro (lendo uma partição a mais que o outro).

Figura 19. Grupo com 2 consumidores

Imaginando a mesma situação anterior, porém com três consumidores, o processo de leitura terá um desempenho superior uma vez que cada um dos consumidores ficará responsável por fazer a leitura de apenas uma partição.

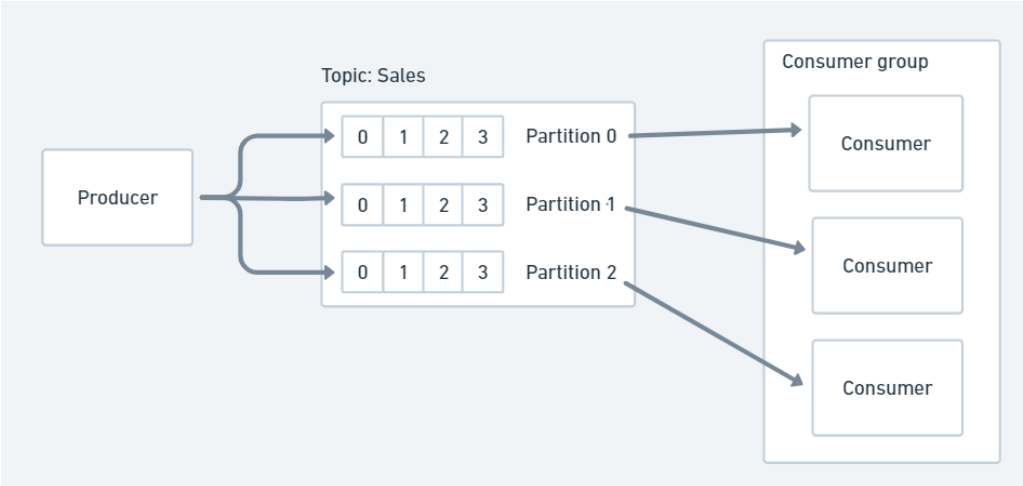

Figura 20. Grupo com 3 consumidores

No entanto, quanto mais consumidores não necessariamente significa que o processo de leitura sempre será mais rápido. Caso seja 4 consumidores para 3 partições, um desses consumidores ficará inativo, uma vez que cada partição pode ser lida por apenas um consumidor do mesmo grupo. Portanto, para haver um máximo desempenho no processo de leitura, o número de partições deve ser a mesma que o número de consumidores.

Um consumidor de determinado grupo pode ler partições de tópicos diferentes em simultâneo. Imagina que o tópico sales é divido em 3 partições e um tópico product dividido em 2 partições. Um dos consumidores de determinado grupo consegue, por exemplo, ler o conteúdo da partição 3 do tópico sales e outro conteúdo da partição 1 do tópico products. Apesar desse fato ser uma possibilidade, para haver um paralelismo mais completo, o mais correto seria haver mais um consumidor para que a distribuição de leitura seja total.

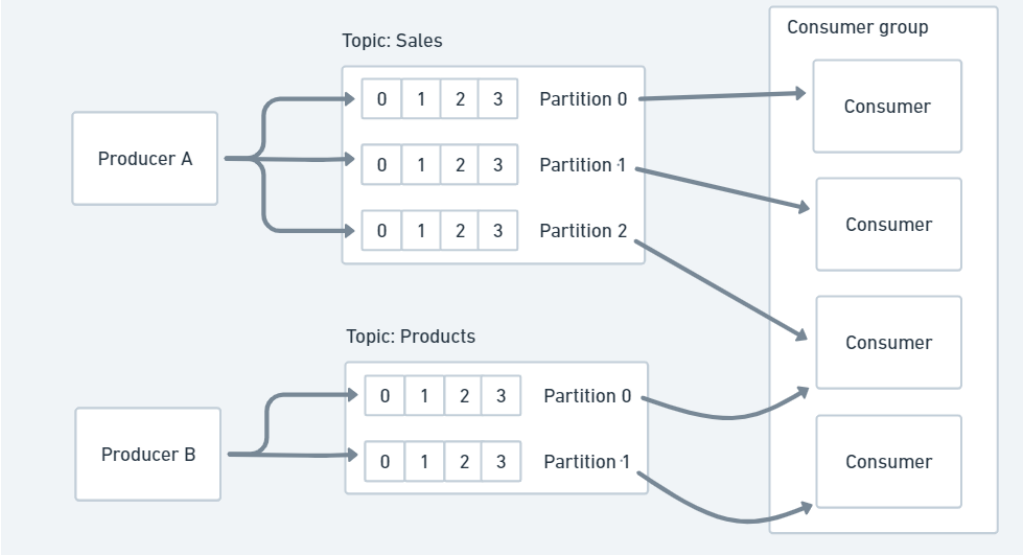

Figura 21. Grupo com 4 consumidores

6.2. Reequilíbrio dos consumidores

Continuando com o exemplo anterior, considere dois tópicos divididos em suas partições e um grupo de consumidores distribuídos para leitura do conteúdo dessas partições, sendo que um deles esteja responsável por ler uma partição do primeiro e do segundo tópico. Caso esse consumidor saia do ar, o Kafka percebe, automaticamente, a ausência desse consumidor e rebalanceia o sistema, fazendo com que as partições que estavam sendo lidas por esse consumidor que caiu sejam lidas por outro.

Portanto, toda vez que é alterado o número de consumidores, o Kafka irá fazer uma redistribuição de tarefas entre eles.

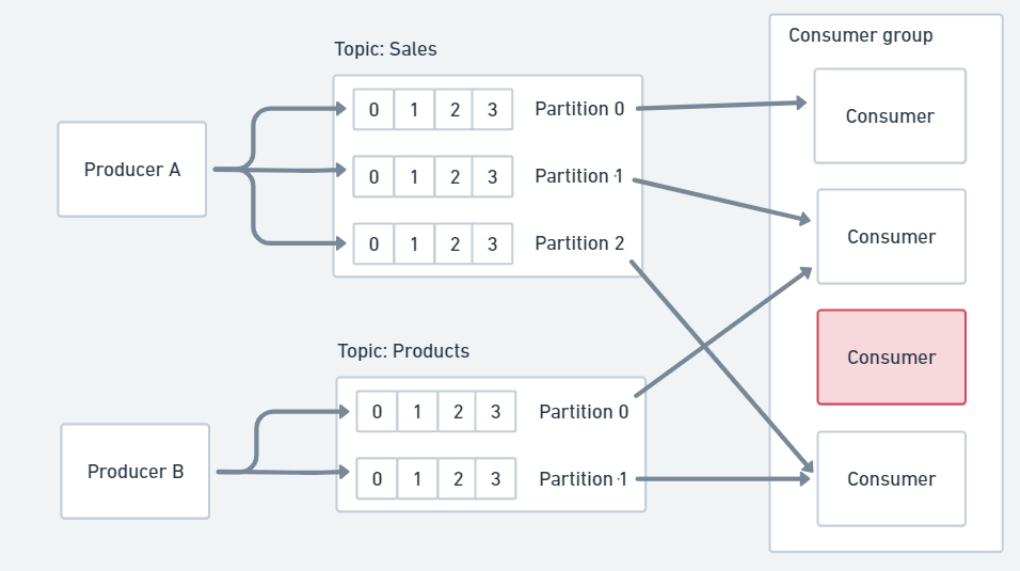

Figura 22. Grupo com 1 consumidor inativo

7. Segurança

O Kafka provê diversos mecanismos de segurança dos dados, tais como a criptografia de dados em trânsito e a autenticação/autorização para controle de acesso.

Os dados enviados a partir de um producer podem ser criptografados de modo a proteger a integridade e confiabilidade do dado no trajeto até ao broker, impedindo que atores maliciosos interceptem a informação, porém, esse dado é descriptografado e armazenado em plain text por padrão no cluster do Kafka, sendo criptografado novamente apenas após a saída em direção aos consumers, desta forma, é importante também ter controle sob quem tem acesso ao cluster, caso contrario, um ator malicioso poderia ler os dados em plain text disponíveis nos brokers sem a utilização de uma chave de criptografia.

Para isso, também são disponibilizadas ferramentas de AAA (Authentication Authorization and Accounting), pelas quais é possível autenticar consumers (aplicações) autorizadas a escrever e ler naquele cluster, e com base nisso, também custodiar direito de acesso para diferentes clientes à tópicos específicos, garantindo que eles tenham acesso apenas às informações necessárias e nada mais, também conhecido como o conceito de *least priviledge* da tríade CIA (Confidentiality, Integrity and availability).

Por fim, existem também os audit logs, utilizados para identificar os responsáveis por extensivas ações realizadas no cluster, desde simples postagens a um tópico, até ações administrativas. Os audit logs registram o usuário utilizado e o timestamp de ocorrência das ações, que podem ter sido autorizadas ou não, ou seja, tentativas falhas de acesso à um recurso não autorizado serão armazenadas e passiveis de processamento em um SIEM (Security Information and Event Management) para o reconhecimento de atividades maliciosas, na ocorrência de falhas de segurança e de vazamento de dados, esses logs também podem ser utilizados para rastrear o pedaço de software comprometido.

8. Kafka Connect

O Kafka connect é uma forma de se pegar dados de um determinado sistema e transferir para outro. Imagina uma situação cotidiana para um programador em que ele deseja

receber os dados de algum ambiente, tais como o twitter ou algum banco de dados como o MySQL ou MongoDB. Ao em vez de codificar esse processo de leitura de busca de dados, pode ser utilizado o Kafka connect, pois o processo será feito de forma automática.

Nesse sentido, o Kafka é provido de diversos conectores que servem para escutar determinados sistemas em busca de alguma alteração nos dados. Todas às vezes que houver uma alteração nesses sistemas, como no MySQL, o Kafka connect pega essa alteração e publica em determinado tópico.

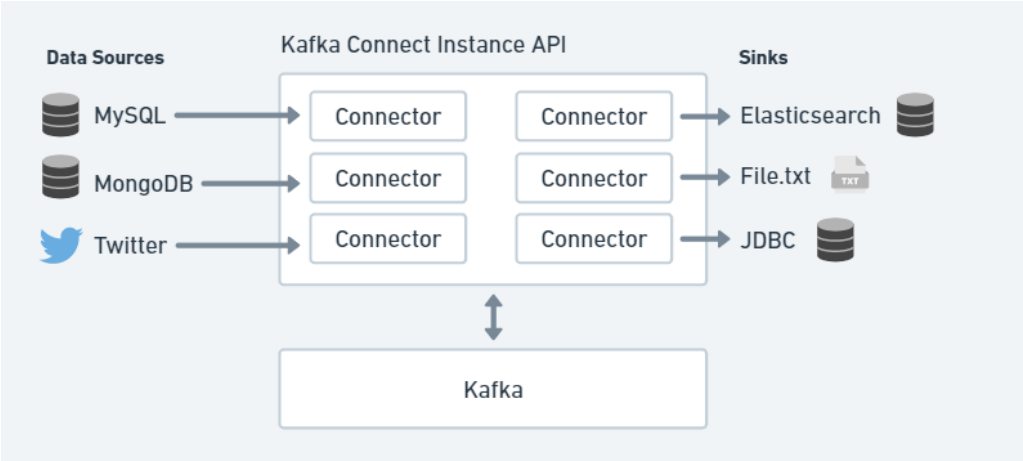

Figura 23. Kafka Connect

Não só isso, mas o Kafka connect não trabalha apenas com data sources, mas também trabalha com sinks. As sinks funcionam da forma inversa explicada anteriormente. Diferentemente dos data sources em que dados são recebidos de terminados sistemas e enviados ao Kafka, é possível, com elas, receber dados de determinado tópico e enviar a diferentes fontes de dados externos.

Dessa forma, é possível receber dados de um MySQL e enviá-los ao Kafka, por um conector, processar esses dados e enviá-los a outras fontes de dados como um simples arquivo texto, através de outro conector.

Imagine que se queira pegar os dados do twitter e armazená-los em um elasticsearch para posterior análise. Fazer todo o processo de leitura do twitter e envio a determinado banco de dados de forma manual não seria vantajoso para nenhum programador. Para resolver esse problema utiliza-se o Kafka connect, uma vez que todo esse processo é feito de forma automática.

8.1. Workers e Tasks

Assim como o apache Kafka é um cluster com várias maquinas, o Kafka connect também possui essa mesma estrutura.

Para que o Kafka connect tenha alto desempenho em relação ao recebimento de dados, é necessario que sua estrutura trabalhe de forma paralela. Dessa forma, o Kafka divide um conector do Kafka connect, responsavel pela transferência de dados, em tarefas (ou tasks) que serão atribuídas aos diferentes workers.

Quando se deseja receber dados do twitter, o conector A, responsável por esse recebimento, será dividido, por exemplo, em 3 tarefas. Da mesma forma, quando for

criado o conector B para o envio desses dados a um banco de dados externo, esse novo conector será divido também em 3 tasks. Dessa forma os dados serão transferidos de um lado para o outro de forma paralela, de forma mais rápida.

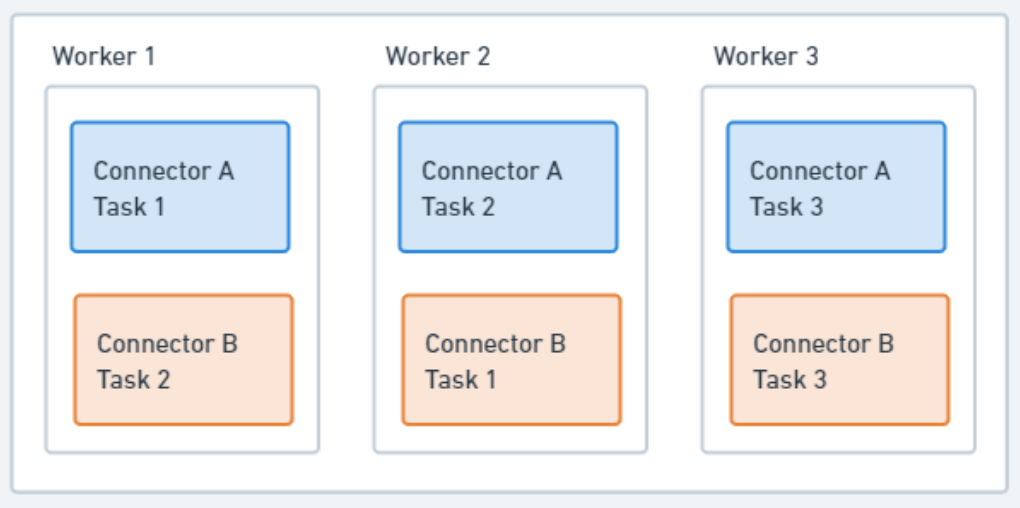

Figura 24. Workers e tasks

9. Kafka REST Proxy

O Apache Kafka fornece um serviço para o usuário, o qual não tem pretenção de conectar-se ao Kafka através do driver padrão. Dessa forma, a aplicação exerce uma requisição HTTP para a API do Rest Proxy do Kafka, o qual vai entrar em uma fila e o Rest Proxy redireciona a mensagem para o Kafka. Por fim, o Kafka irá executar todo o fluxo do processo de entrega explicado anteriormente na secção 5.1.

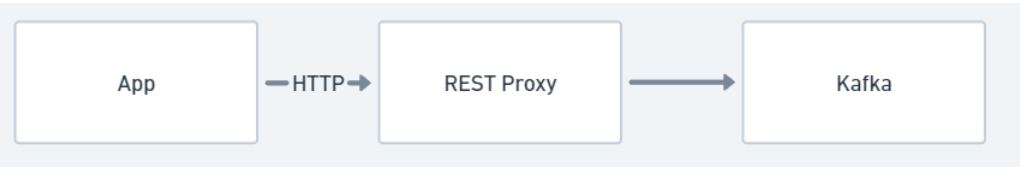

Figura 25. Kafka REST Proxy

10. Compatibilidade de Dados

Normalmente, quando se trata de qualquer sistema de mensageria, como o Kafka, mensagens são transportadas de um lado para o outro o todo tempo. Em relação a isso, para o Kafka conseguir receber esses dados, eles precisam estar formatados com um determinado padrão e, além disso, os consumidores que estarão lendo esses dados devem saber qual padrão se encontra determinada mensagem, para realização de forma correta da leitura.

O Kafka consegue garantir que os consumidores entendam o padrão das mensagens, evitando qualquer tipo de dificuldade de leitura, por meio do Schema Registry, sendo ele um sistema no qual será registrado o padrão da mensagem que será envada. Em relação a isso, quando o produtor envia uma mensagem ao kafka, ele irá definir qual

padrão ele quer utilizar. O Kafka, através do schema registry irá carregar esse schema e verificar qual que é o padrão que essa mensagem será enviada. Uma vez que se verifica que o padrão está correto, essa mensagem será serializada no serializer e enviada diretamente ao tópico.

Como cada schema carregada tem um id próprio para reconhecimento de determinado padrão de mensagem, quando um consumidor ler uma mensagem ele irá verificar o schema id dessa mensagem e fazer a consulta do padrão utilizada por ela no schema registry.

Portanto, com o schema registry é possível registrar o padrão com o qual as mensagens são enviadas ao Kafka, de modo a garantir coesão.

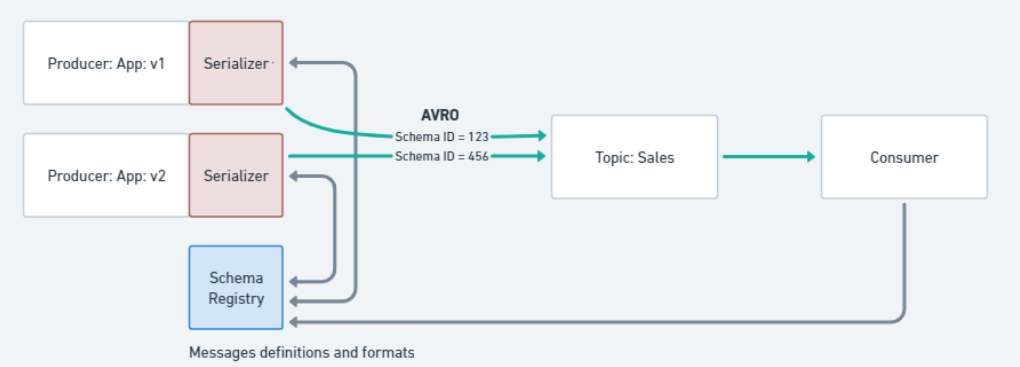

Figura 26. Compatibilidade de dados

10.1. Linguagem de descrição de interface (LDI)

O IDL (Interface Description Language) é um dos padrões de descrição de mensagens do Kafka, sendo que um dos padrões utilizados por ele para o tratamento de mensagens no Kafka é o Apache AVRO. Basicamente esse padrão é um JSON com um namespace, tipo, nome e os campos da mensagem a ser enviada.

```
{
  "namespace": "pnda.entity",
  "type": "record",
  "name": "event",
  "fields": [
     {"name": "timestamp", "type": "long"},
     {"name": "src",       "type": "string"},
     {"name": "host_ip",   "type": "string"},
     {"name": "rawdata",   "type": "bytes"}
  ]
}
```

Figura 27. Padrão Apache AVRO

11. Kafka Streams

O Kafka Streams é uma biblioteca feita em java que trabalha em real time com o objetivo de processar e transformar dados. Em outras palavras, ele consegue pegar os dados recebidos no Kafka, transformar essas informações e entregar a algum tópico do Kafka.

Imagine que um cluster com Kafka connect está recebendo informações do MongoDB através de seus conectores. Nesse sentido o Kafka connect irá persistir esses dados a um tópico no Kafka. A partir daí será conectado um sistema em java utilizando o Kafka streams para leitura de mensagens desse tópico a fim de transformar ou modificar esses dados. Com isso, esses dados modificados podem ser lidos por mais alguns streams ou diretamente enviados ao Kafka para ser distribuídos a algum outro sistema de armazenamento de dados (sink).

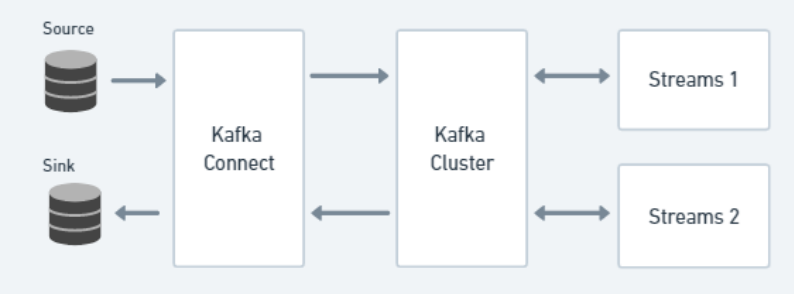

Figura 28. Kafka Streams

Em um outro exemplo, uma mensagem pode ser produzida por um sistema produtor e enviada ao cluster onde será tratada pelo kafka streams e enviada a algum sistema consumidor.

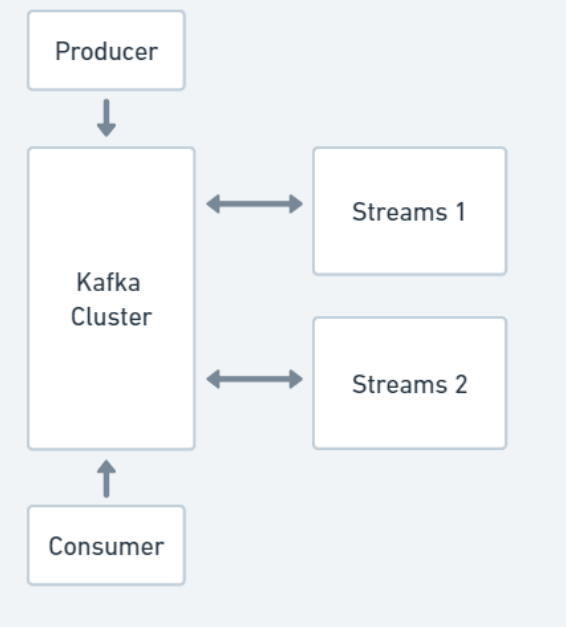

Figura 29. Kafka Streams

12. Confluent ksqlDB

O confluent ksqlDB é um banco de dados desenvolvido para processamento de dados em tempo real integrado com a infraestrutura Kafka, de modo que pode ser adicionado à instalação existentes sem muita dificuldade.

Ele permite que os dados recebidos em tempo real nos tópicos sejam processados através de simples queries SQL, facilitando a transformação de dados como operações matemáticas, selects, joins, counts e views, além disso, é possível disponibilizar todas essas informações em uma REST API, que pode ser utilizada por terceiros para integração com o sistema, sem acesso direto, mas ainda assim se aproveitando do processamento em tempo real.

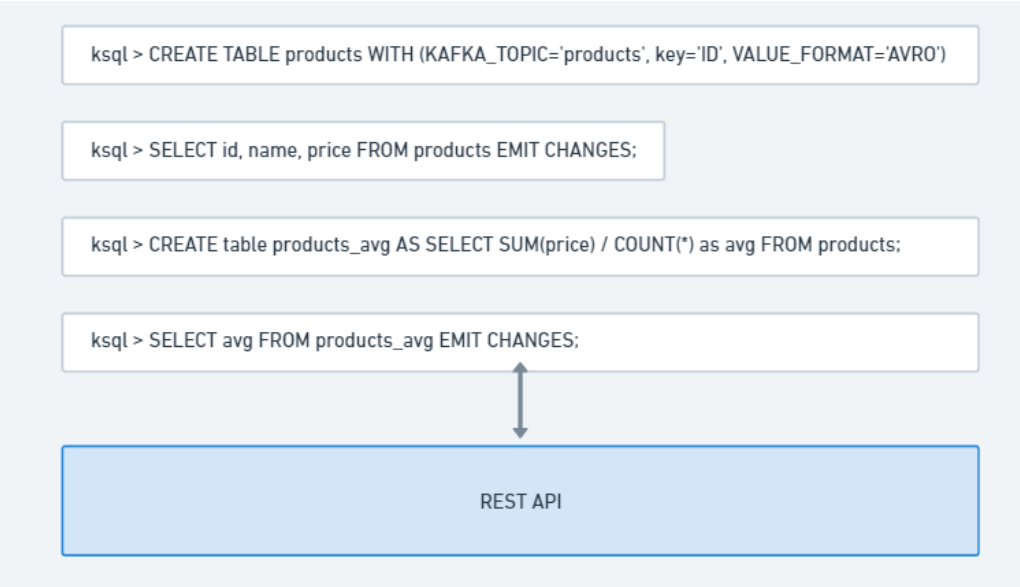

Figura 30. Confluent ksqlDB

13. Apache Kafka na prática

Acessar: aqui

Antes de mais nada, é necessário realizar a configuração básica para uso do Apache Kafka. Para isso, é exige-se que acesse o github da confluent inc e realizar o download (por meio de um git clone, por exemplo) da imagem do kafka cluster para uso no docker. Após isso, abra o terminal e navegue até o diretório da imagem. Logo após, execute o comando ***docker-compose up -d***. O Docker Compose é usado para executar vários contêineres como um único serviço e, no caso especificado, irá subir os zookeepper e kafka dentro do arquivo ***docker-compose.yml***. Em seguida, mas não obrigatório, apenas para confirmar que tudo ocorreu com sucesso, basta executar o comando ***docker-compose ps*** e espera-se que apareça 6 items, 3 referentes ao apache kafka e 3 referente ao zookeeper. Além disso, para entrar no terminal dentro do container, basta executar ***docker exec -it NOMEDOCONTAINER bash***, esse seria o básico para começar a trabalhar com o Apache Kafka.

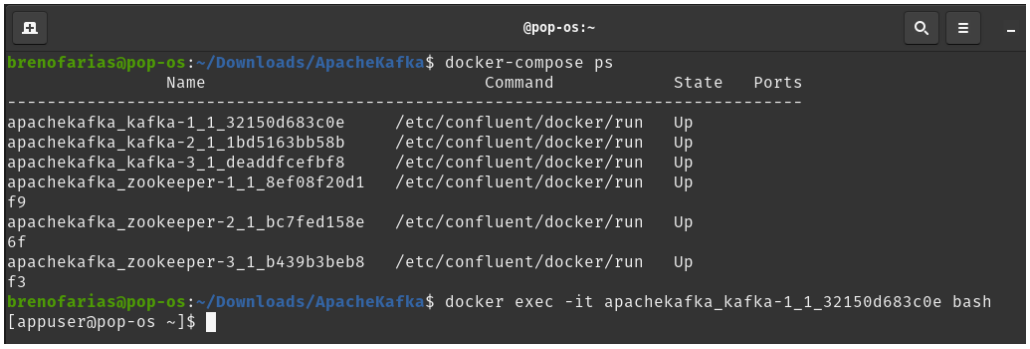

Figura 31. Setup

kafka-console-producer –broker-list localhost:19092 –topic topicoDeExemplo kafka-console-consumer –bootstrap-server localhost:19092 –topic topicoDeExemplo –from-beginning –group primeiroGrupo kafka-topics –describe –bootstrap-server localhost:19092 –topic topicoDeExemplo kafka-consumer-groups –group primeiroGrupo –bootstrap-server localhost:19092 –describe ***kafka-console-producer –broker-list localhost:19092 –topic topicoDeExemplo kafka-console-consumer –bootstrap-server localhost:19092 –topic topicoDeExemplo –from-beginning –group primeiroGrupo***

Para dar início ao uso próprio da ferramenta, é primordial e que o usuário crie um tópico. O processo de criação é extremamente importante pois irá influenciar todo o sistema. A criação deve ser feita de modo a usar os argumentos para referenciar qual máquina será usado (neste exemplo, será a máquina local, ***localhost:numeroDaPortaDoKafka***. O valor da porta pode ser encontrado dentro do arquivo em formato .yml adquirido, dentro de kafka-1: usando o valor usado em ***KAFKA_ADVERTISED _LISTENERS***, além de explicitar o ***–replication-factor valorDoReplicationFactor***, o número de partições ***–partitions numeroDePartições*** e o nome do tópico ***–topic nomeDoTopico***. Dessa forma, o comando para criação do tópico dentro do Apacha Kafka será algo como: ***kafka-topics –create –bootstrap-server localhost:19092 –replication-factor 3 –partitions 3 –topic topicoDeExemplo***. Para mais informações sobre a criação de um tópico, basta executar o comando ***karka-topics***.

```
kafka-1:
  image: confluentinc/cp-kafka:latest
  network_mode: host
  depends_on:
    - zookeeper-1
    - zookeeper-2
    - zookeeper-3
  environment:
    KAFKA_BROKER_ID: 1
    KAFKA_ZOOKEEPER_CONNECT: localhost:22181,localhost:32181,localhost:42181
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:19092
  extra_hosts:
    - "moby:127.0.0.1"
```

Figura 32. Variável *KAFKA_ADVERTISED _LISTENERS*

Além disso, é possível observar quais tópicos há ao usar o comando ***karka-topics –list –bootstrap-server localhost: karka-topics –create –bootstrap-server numeroDaPortaDoKafka***. Não é o intuito deste projeto programar em uma linguagem de programação,

mas sim mostrar o básico para que o leitor seja capaz de compreender o funcionamento da ferramenta apresentada. Sendo assim, será utilizado o ***kafka-console-producer –broker-list localhost:numeroDaPortaDoKafka –topic nomeDoTopico*** para fazer a conexão com o tópico criado e tornar possível o envio de mensagem para o tópico referenciado. Dessa forma, tudo que for escrito no terminal após o uso deste comando, será enviado para o tópico criado. Todavía, para ler a informação do tópico, é necessário abrir outro terminal e seguir o comando kafka-console-consumer –bootstrap-server localhost:numeroDaPortaDoKafka –topic nomeDoTopico, após isso, qualquer informação que for enviada no console do produtor irá aparecer no console do consumidor. Porém, uma vez que uma mensagem é lida, ela não será lida novamente, pois apenas são lidas as mensagens a partir do momento em que elas começaram a ser consumidas. Todavía, o Kafka armazena em disco essas mensagens que foram mandadas desde o início da criação desse tópico para que possam ser reprocessadas. Dessa forma, para poder usufruir deste recurso, é necessário adicionar, no fim do comando de execução do consumidor, o parâmetro ***–from-beginning***, o que resultará em ***kafka-console-consumer –bootstrap-server localhost:19092 –topic topicoDeExemplo –from-beginning***.

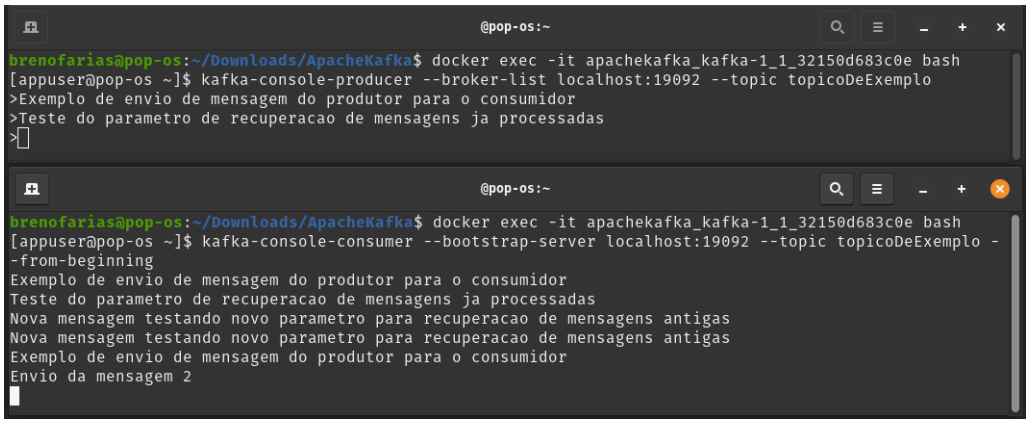

Figura 33. Produtor / Consumidor usando o parâmetro –from-beginning

A partir do momento em que um consumidor é aberto, ele vira um cliente e, tendo isso em mente, é importante lembrar que esse tópico pode ter várias partições. Dessa forma, seria possível ter 3 clientes, lendo essas 3 partições diferentes, o que implica que seria possível processar essas informações em paralelo. Para fazer com que cada cliente leia uma partição diferente do mesmo tópico, é preciso criar um grupo de consumo, e o kafka irá, sem esforço por parte do programador, organizar esse grupos de modo que cada cliente consuma mensagens de partições diferentes. Para tal, é necessário usar o parâmetro textbf–*group nomeDoGrupo*. No entanto, no exemplo dado, havia apenas um consumidor. Para simular o uso de vários clientes, basta seguir os passos de execução de um consumidor e adcionar o parâmetro para referenciar a qual grupo ele pertence. Note que, ao fazer isso, tendo vários clientes lendo um tópico, ao enviar uma mensagem para um determinado grupo em um dado tópico, os consumidores não irão ambos mostrar essa mensagem, pois cada um estará lendo uma partição diferente.

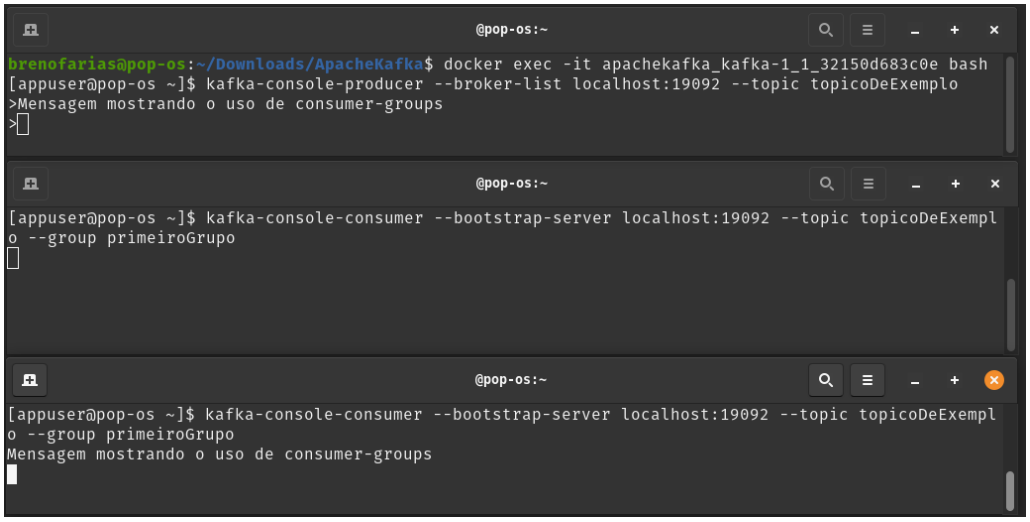

Figura 34. Apresentando o uso de *consumer-groups*

Para ver como o tópico criado está dividido, basta dar um ***kafka-topics –describe –bootstrap-server localhost:numeroDaPortaDoKafka –topic nomeDoTopico*** no tópico criado.

- O mesmo vale para verificar a configuração de um determinado grupo ***kafka-consumer-groups –group nomeDoGrupo –bootstrap-server localhost:numeroDaPortaDoKafka –topic nomeDoTopico***. Note que, são oferecidas informações, após a execução de ambos os comandos como, nome do ***TOPIC***, **Partition**, ***Partition Leader***, o ***PartitionCount***, ***ReplicationFactor***, etc. A maioria são bem intuitivas, então será explicado apenas os mais incomuns.
- ***ISR*** é simplesmente todas as réplicas de uma partição que estão ***”in-sync”*** com o líder.
- ***Current-Offset*** - Cada partição tem um offset, ou seja, um número associado para gravação de uma mensagem em uma partição.
- ***Log-End-Offset*** - Esse valor representa qual foi o último offset gravado, pois pode acontecer de um consumidor estar processando uma partição, mas ele ainda não chegou no último offset que foi gerado.
- ***LAG*** - Esse número representa quantas mensagens estão pendentes, ou seja, estão em um estado pronto para serem processadas.

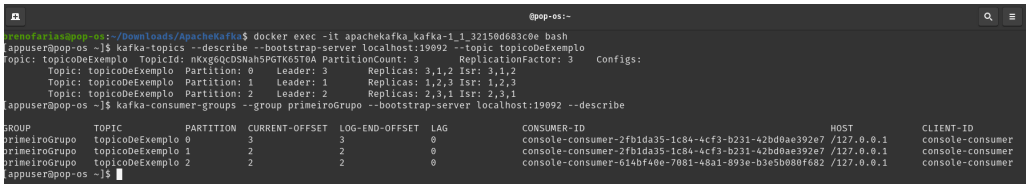

Figura 35. Uso de *describe* para tópicos e grupos

14. References

Audit Log Concepts — Confluent Documentation. Confluent.io. Disponível em: ¡https://docs.confluent.io/platform/current/security/audit-logs/audit-logs-concepts.html¿. Acesso em: 29 maio 2022.

Apache Kafka Documentation. Apache Kafka. Disponível em: ¡https://kafka.apache.org/documentation/¿. Acesso em: 29 maio 2022.

Apache Kafka Security. Apache Kafka. Disponível em: ¡https://kafka.apache.org/10/documentation/streams/developer-guide/security¿. Acesso em: 29 maio 2022.

Security fundamentals. Networking Academy. Disponível em: ¡https://www.netacad.com/pt-br/courses/all-courses¿. Acesso em: 29 maio 2022.

ksqlDB: The database purpose-built for stream processing applications. Ksqldb.io. Disponível em: ¡https://ksqldb.io/quickstart.html¿. Acesso em: 29 maio 2022.

Kafka Streams Overview — Confluent Documentation. Confluent.io. Disponível em: ¡https://docs.confluent.io/platform/current/streams/index.html¿. Acesso em: 29 maio 2022.

Full Cycle. Full Cycle. Disponível em: ¡https://fullcycle.com.br/¿. Acesso em: 29 maio 2022.