[Aspectos gerais do Kafka 4](#__RefHeading___Toc1378_3884422906)

[Dinâmica básica de funcionamento 4](#__RefHeading___Toc1380_3884422906)

[Tópicos 6](#__RefHeading___Toc1382_3884422906)

[Anatomia de um registro 8](#__RefHeading___Toc1384_3884422906)

[Partições 9](#__RefHeading___Toc1386_3884422906)

[Informações importantes sobre envio de mensagens para partições 9](#__RefHeading___Toc1388_3884422906)

[Garantindo a ordem de entrega 11](#__RefHeading___Toc1390_3884422906)

[Sobre as Keys 12](#__RefHeading___Toc1392_3884422906)

[Partições distribuídas 13](#__RefHeading___Toc1394_3884422906)

[Partition leadership 15](#__RefHeading___Toc1396_3884422906)

[Garantia de entrega de mensagens 16](#__RefHeading___Toc1398_3884422906)

[ACK 0 16](#__RefHeading___Toc1400_3884422906)

[ACK 1 16](#__RefHeading___Toc1402_3884422906)

[ACK – 1 16](#__RefHeading___Toc1404_3884422906)

[Performance vs Garantias 17](#__RefHeading___Toc1406_3884422906)

[Produtor indepotente 18](#__RefHeading___Toc1408_3884422906)

[Consumers e consumer groups 19](#__RefHeading___Toc1410_3884422906)

[Docker compose para utilizar o Kafka (3 brokers) 21](#__RefHeading___Toc1412_3884422906)

[Alguns comandos manuais do Kafka 24](#__RefHeading___Toc1414_3884422906)

[Observação importante 24](#__RefHeading___Toc1416_3884422906)

[Criar tópico 25](#__RefHeading___Toc1418_3884422906)

[Lista tópicos 25](#__RefHeading___Toc1420_3884422906)

[Detalhar tópicos 25](#__RefHeading___Toc1422_3884422906)

[Iniciar um Consumer 25](#__RefHeading___Toc1424_3884422906)

[Configurando o Producer e enviando mensagem 25](#__RefHeading___Toc1426_3884422906)

[Teste com envio de mensagens 26](#__RefHeading___Toc1428_3884422906)

[Testes com Consumer Group 27](#__RefHeading___Toc1430_3884422906)

[Por dentro de um Consumer Group 28](#__RefHeading___Toc1432_3884422906)

[Kafka Connect 30](#__RefHeading___Toc1434_3884422906)

[Dinâmica 30](#__RefHeading___Toc1436_3884422906)

[Standalone Workers 32](#__RefHeading___Toc1438_3884422906)

[Distributed Workers 33](#__RefHeading___Toc1440_3884422906)

[Converters 34](#__RefHeading___Toc1442_3884422906)

[DLQ – Dead Letter Queue 35](#__RefHeading___Toc1444_3884422906)

[Kafka Connect na prática 36](#__RefHeading___Toc1446_3884422906)

[Docker compose 36](#__RefHeading___Toc1448_3884422906)

[Anotações importantes 39](#__RefHeading___Toc1569_3884422906)

[Erro ao utilizar transações - relação entre transações e número de Brokers 39](#__RefHeading___Toc1571_3884422906)

[Particionamento de mensagens 40](#__RefHeading___Toc1573_3884422906)

[Round Robin Partitioner 40](#__RefHeading___Toc1575_3884422906)

[Sticky Partitioner 40](#__RefHeading___Toc1577_3884422906)

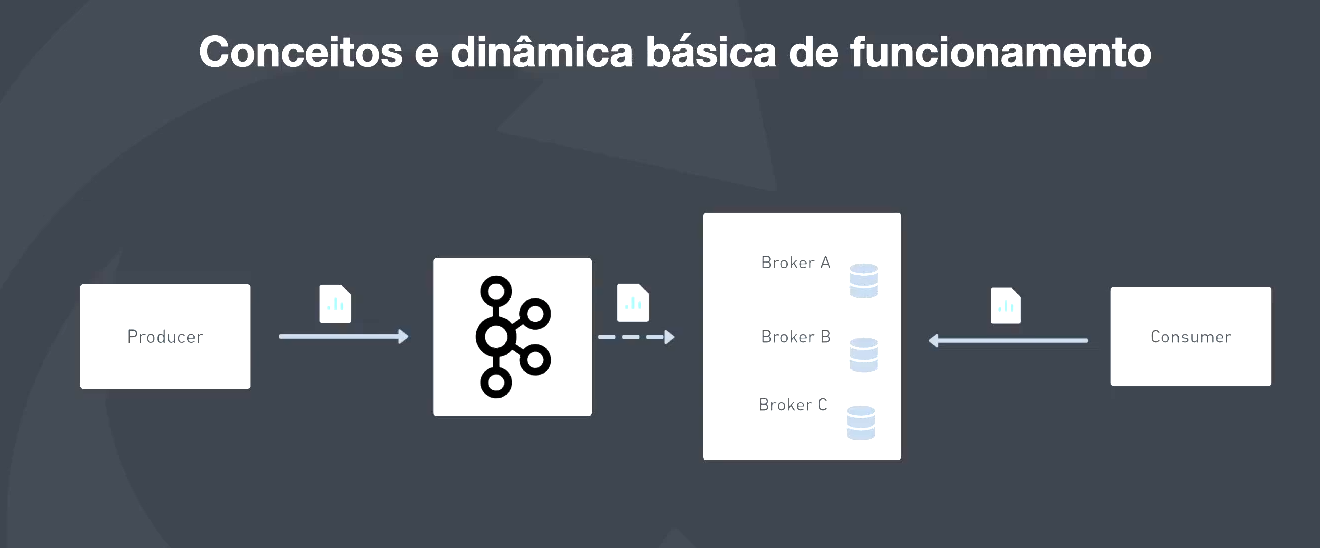
[ACK properties no Spring Kafka 41](#__RefHeading___Toc1293_236501102)

[Réplicas e Retry 42](#__RefHeading___Toc1295_236501102)

# Aspectos gerais do Kafka

* O Kafka tem um alto throughput, ou seja, grande capacidade de receber e processar as requisições.
* Possui uma latência baixa (2ms).
* É escalável.
* Consegue focar em armazenamento. É possível armazenar ou descartar as mensagens.
* Possui alta disponibilidade.
* Se conecta com quase tudo.

## Dinâmica básica de funcionamento



Imagine o seguinte cenário onde um processo de checkout faz um pagamento e deve-se gerar uma nota fiscal. Então, sai um evento do pagamento, esse evento precisa ser guardado em um local, o sistema de nota fiscal irá ficar acessando esse local para verificar todas as compras e, sempre que tiver uma nova compra, uma nota fiscla é emitida.

O Kafka é o sistema intermediário. Ele é um conjunto de máquinas, é um cluster formado de nós. Estes nós do cluster são chamados de Broker.

O producer envia uma mensagem ao Kafka, essa mensagem cai em um broker e fica guardada em um banco de dados. Cada Broker tem o seu banco de dados, cada broker é uma máquina.

O consumer acessa o broker. O Kafka não envia mensagens a ninguém.

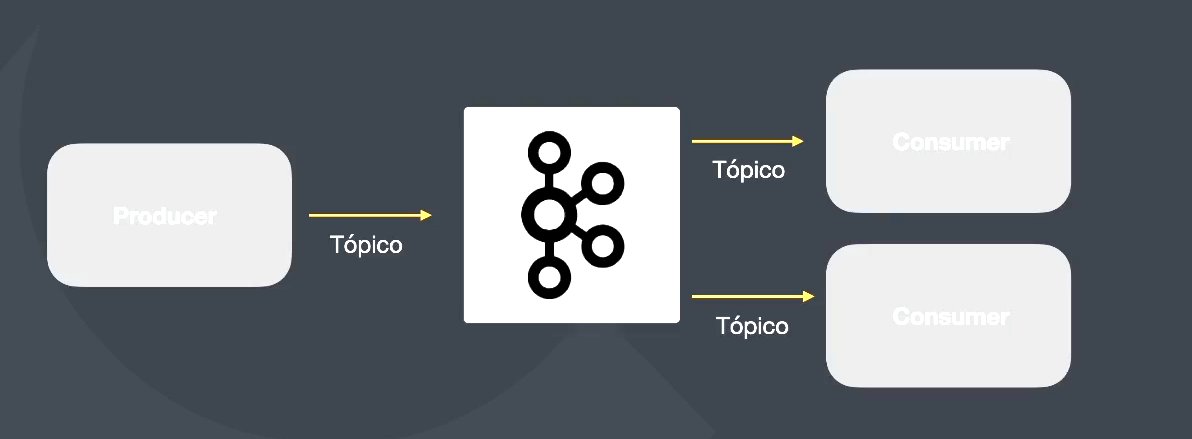
**Importante:**

* Normalmente, para colocar um cluster em produção no Kafka, a recomendação mínima são 3 Brokers.
* Os Brokers se comunicam o tempo inteiro. O Kafka precisa de um sistema para gerenciar essa comunicação. Esse sistema é o zookeeper (O zookeeper está de saída, o Kafka estra sendo atualizado para não precisar mais dele).

# Tópicos

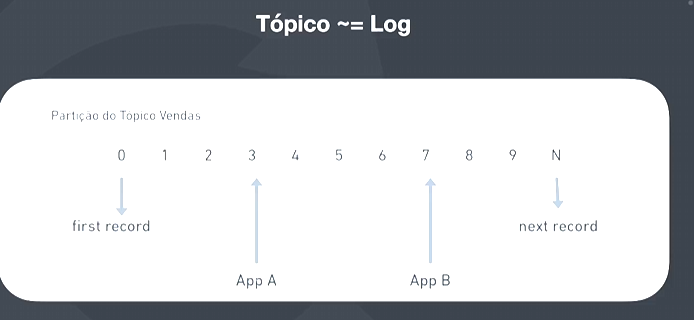
É o canal de comunicação responsável por receber e disponibilizar os dados enviados ao Kafka. Então quando um produtor envia uma mensagem ao Broker, essa mensagem é enviada a um tópico. Quando o consumidor deseja lera mensagem, ele a lê de um tópico.

Tópico não é o mesmo que Fila.



Um tópico pode ser lido por vários consumers. Diferente, por exemplo, do RabbitMQ onde, ao ler uma mensagem de uma fila, a mesma desaparece. Não da para 2 sistemas ler a mesma mensagem na mesma Fila.

O tópico é parecido com um log, ou seja, um local onde é possível colocar uma informação uma atrás da outra.



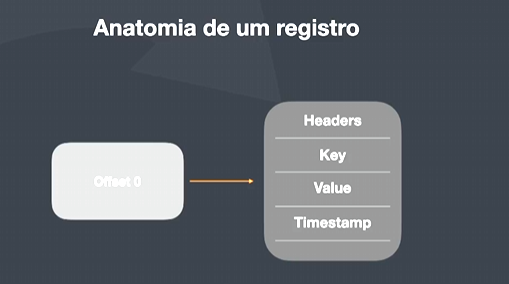
Cada mensagem enviada a um tópico é armazenada uma atrás da outra. Essa mensagem ao ser armazenada ganha uma espécie de ID chamado de offset (posição onde está cada mensagem).

Pode ser que em um momento um consumidor A está lendo a mensagem 3 e o B está lendo a mensagem 7 e no mesmo momento está sendo adicionada uma mensagem. Não há problema em um consumidor estar mais adiantado que o outro na leitura. Pode haver, inclusive, diferença de dias. O Kafka permite inclusive reprocessar as mensagens.

# Anatomia de um registro

A composição de um offset, por exemplo offset 0 é a seguinte:

* Headers: a mensagem pode conter Headers que funcionam como se fossem metadados e que podem ser úteis durante todo o processo.
* Key: conta um contexto do tipo da mensagem, do agrupamento de mensagens que é preciso garantir a ordem na entrega. Será melhor explanado mais a frente.
* Value: é o conteúdo da mensagem em si.
* Timestamp



# Partições

Cada tópico pode ter uma ou mais partições para conseguir garantir a distribuição e resiliência de seus dados.

Cenário hipotético: Imagine um tópico. Ao enviar uma mensagem para esse tópico, a mesma vai para uma partição 1 (imagine a partição como uma gaveta). Essa partição 1 está no Broker A, por exemplo. Ao mandar uma outra mensagem, a mesma vai para a partição 2. A partição 2 pode estar no Broker A ou no Broker 10, por exemplo.

Ao mandar outra mensagem, a mesma foi para uma partição 3. Então percebe-se que sempre que aumenta a quantidade de partições, as mensagens ficam mais separadas. Se o Broker A cair e as mensagens de um tópico estiverem todas nesse Broker A, ninguém consegue ler nada. Se as mensagens estiverem distribuídas entre mais Brokers, se o Broker A cair, não será possível ler a mensagem deste Broker, mas será possível ler dos outros. O que não é o caso do Kafka, pois ele possui outros mecanismos para garantir a resiliência.

Mas existe uma outra situação. Imagina que há 1 milhão de mensagens para serem lidas. Um computador é utilizado para ficar lendo essas mensagens. (Ou seja, esse computador sozinho terá que ler 1 milhão de msgs.) E se tivesse 2 computadores para ler essas mensagens? Então poderia armazenar, por exemplo, 500 mil msgs em um local para a máquina 1 e as outras em outro local para a máquina 2. Assim é possível ter o dobro da velocidade de leitura. Quanto mais partições e mais máquinas lendo, mais rápido as msgs serão consumidas.

A informação estando distribuída diminui o risco de inacessibilidade e ao mesmo tempo permite-se que se tenham mais consumidores consumindo as mensagens do mesmo tópico.

Desta forma é possível escalar o processo no Kafka.

## Informações importantes sobre envio de mensagens para partições

Nas versões do Kafka a partir da 2.4 houve uma alteração o particionamento default do Kafka, para quando a key é nula.

Agora, através da configuração padrão, as mensagens serão enviadas a uma mesma partição até que o batch esteja cheio ou que o linger.ms tenha decorrido.

Tanto o tamanho do batch quanto o linger.ms podem ser configurados.

Alguns links uteis:

[Producer Default Partitioner & Sticky Partitioner | Learn Apache Kafka](https://learn.conduktor.io/kafka/producer-default-partitioner-and-sticky-partitioner/)

[Client Producer and Consumer Configuration Recommendations for Confluent Platform | Confluent Documentation](https://docs.confluent.io/platform/current/clients/client-configs.html)

[Kafka producer configuration reference | Confluent Documentation](https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html)

# Garantindo a ordem de entrega

Quando se trabalha com partições uma preocupação é como garantir a ordem das mensagens. Para isso as Keys irão ajudar.

Imagine o seguinte cenário:

Há a partição 1 a qual é consumida pelo Consumidor 1 (lento). Há a partição 2 sendo consumida pelo Consumidor 2(mais rápido). Há a partição 3 sendo consumida pelo Consumidor 2 também.

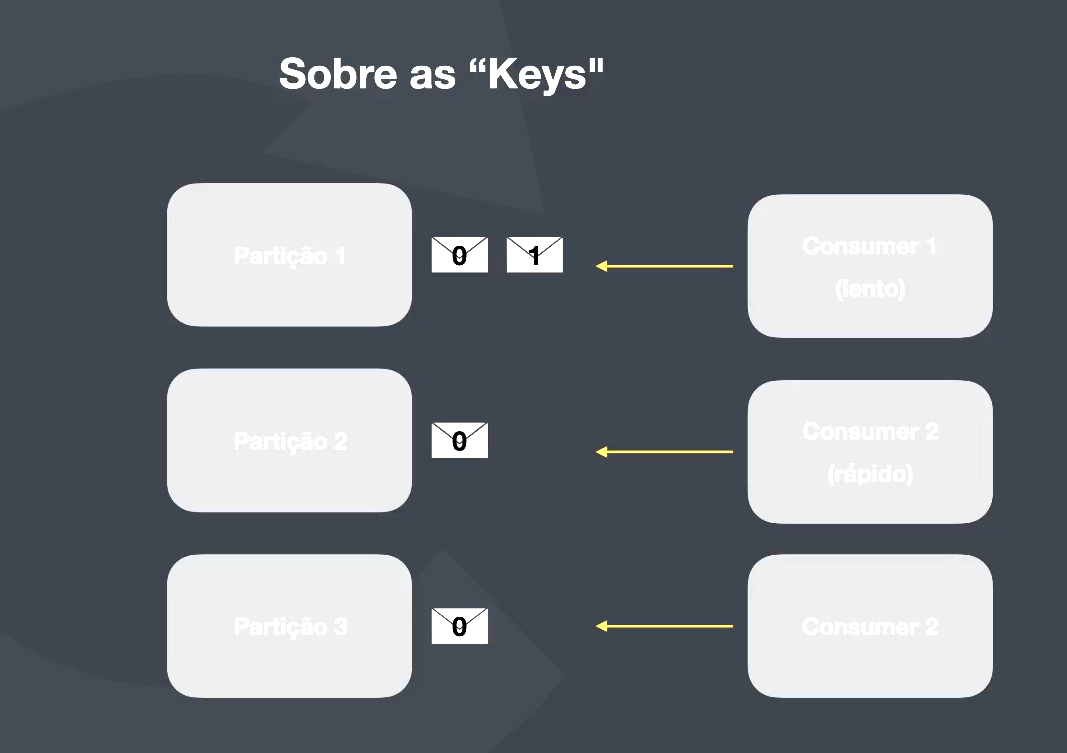
Existem 4 mensagens: A, B ,C e D.

Ao enviar a mensagem A, a mesma cai na partição 1 e vira o offset 0.

Ao enviar a mensagem B, a mesma cai na partição 2 e vira o offset 0.

Ao enviar a mensagem C, a mesma cai na partição 3 e vira o offset 0.

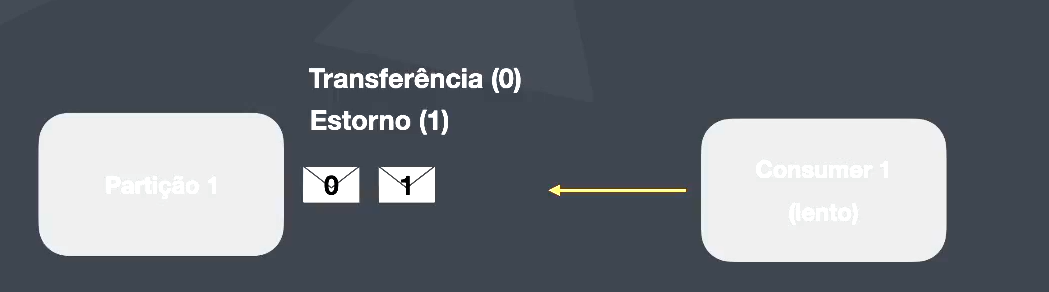
Ao enviar a mensagem D, a mesma cai na partição 1 e vira o offset 1.



Imagine um cenário de uma transferência bancária. A mensagem da partição 1, offset – é a transferência bancária. Suponha que deu um erro de processamento e foi gerada uma mensagem de estorno. Essa é a mensagem da partição 2, offset 0.

O Consumidor 1 está lento enquanto o 2 está rápido. Existe então a chance do Consumidor 2 ler a mensagem do estorno antes da transferência ser consumida. Neste caso, isso é um problema, pois a ordem importa.

**Para garantir a ordem das mensagens, é necessário que elas estejam dentro da mesma partição.** Ou seja, para garantir a ordem é necessário que tanto a mensagem de transferência, quanto a mensagem de Estorno estejam na mesma partição.



Então, como garantir que tanto a transferência quanto o estorno caiam na mesma partição?

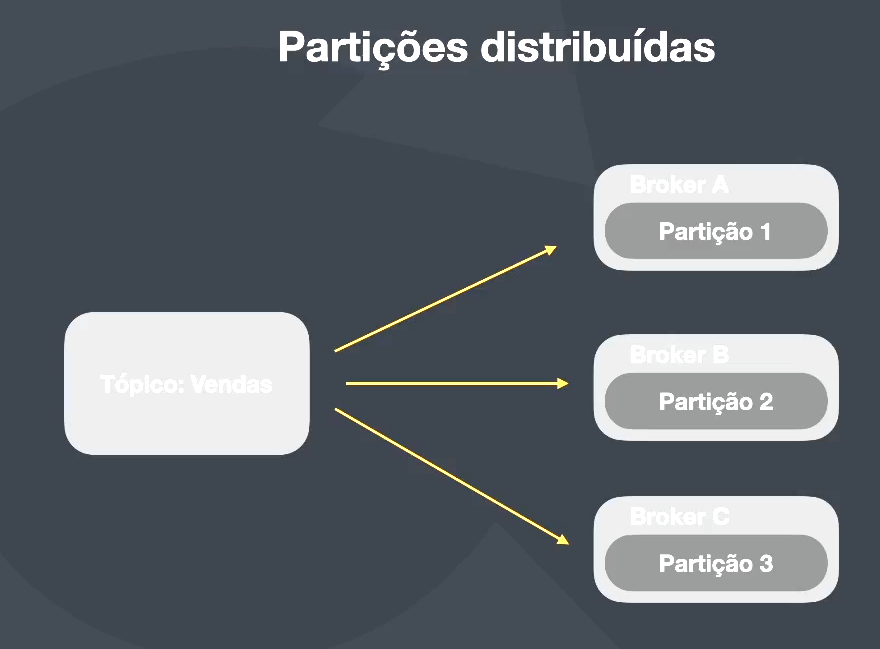
## Sobre as Keys

Para o problema acima, utilizamos as Keys. Ao enviar a mensagem de transferência, deve-se informar, por exemplo, que a key dela é ‘movimentação’. Ao enviar a mensagem de estorno, deve-se informar a mesma key. O Kafka irá enviar as mensagens com a mesma key para a mesma partição.

Em um cenário em que não é necessário ter ordem, a key não se faz necessária.

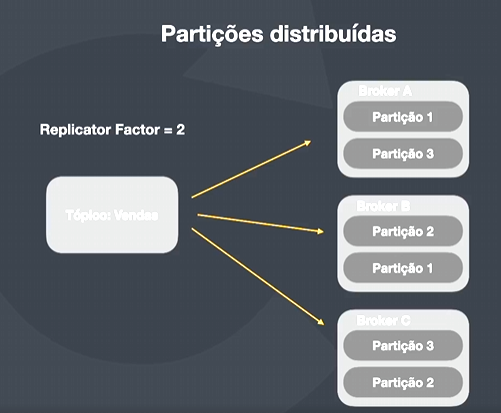
# Partições distribuídas

Imagine o seguinte cenário:



Cada vez que uma mensagem é enviada, cai em uma partição e Broker diferente. Cada Broker é uma máquina diferente.

Desta forma acima, se um Broker cair, os dados de uma partição seriam perdidos. As mensagens seriam perdidas. Para garantir a resiliência, no Kafka é possível escolher algo chamado Replication Factor.

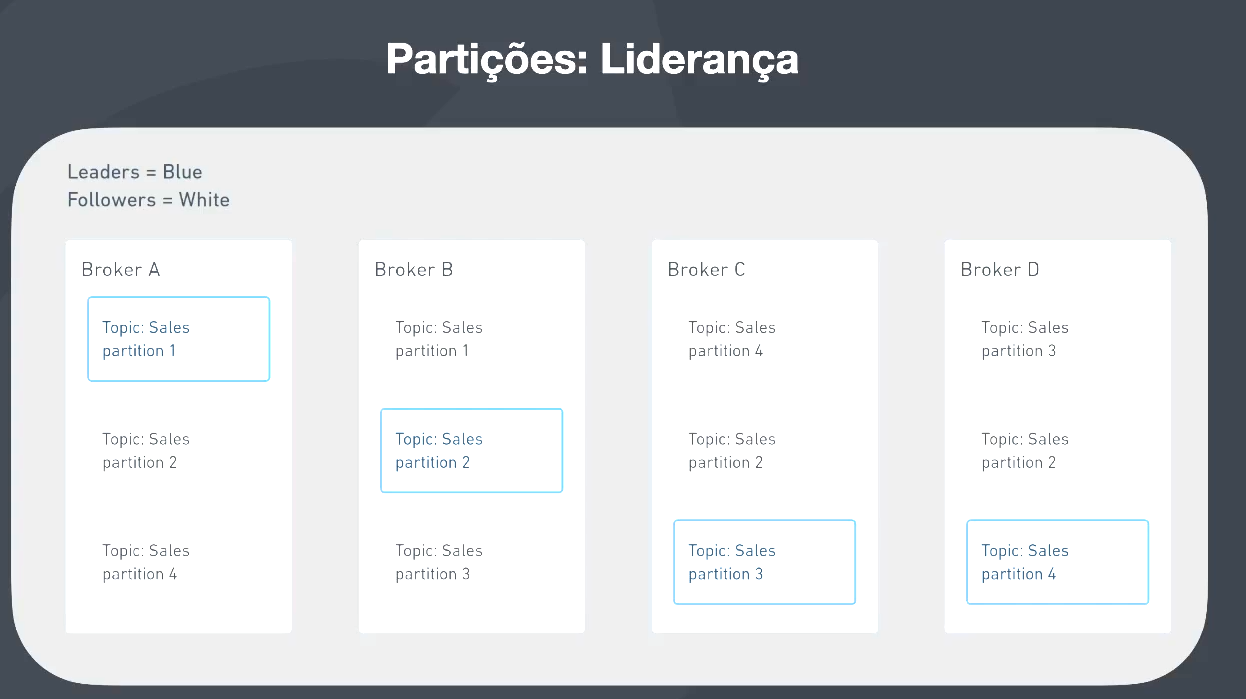


Com isso, é possível ao criar um tópico, dizer por exemplo que esse Tópico terá 3 partições com Replication Factor de 2. Isso significa que cada partição terá pelo menos 2 cópias entre os Brokers. Então se um Broker cair, é garantido um backup daquela partição em outro Broker. Isso ajuda a garantir resiliência.

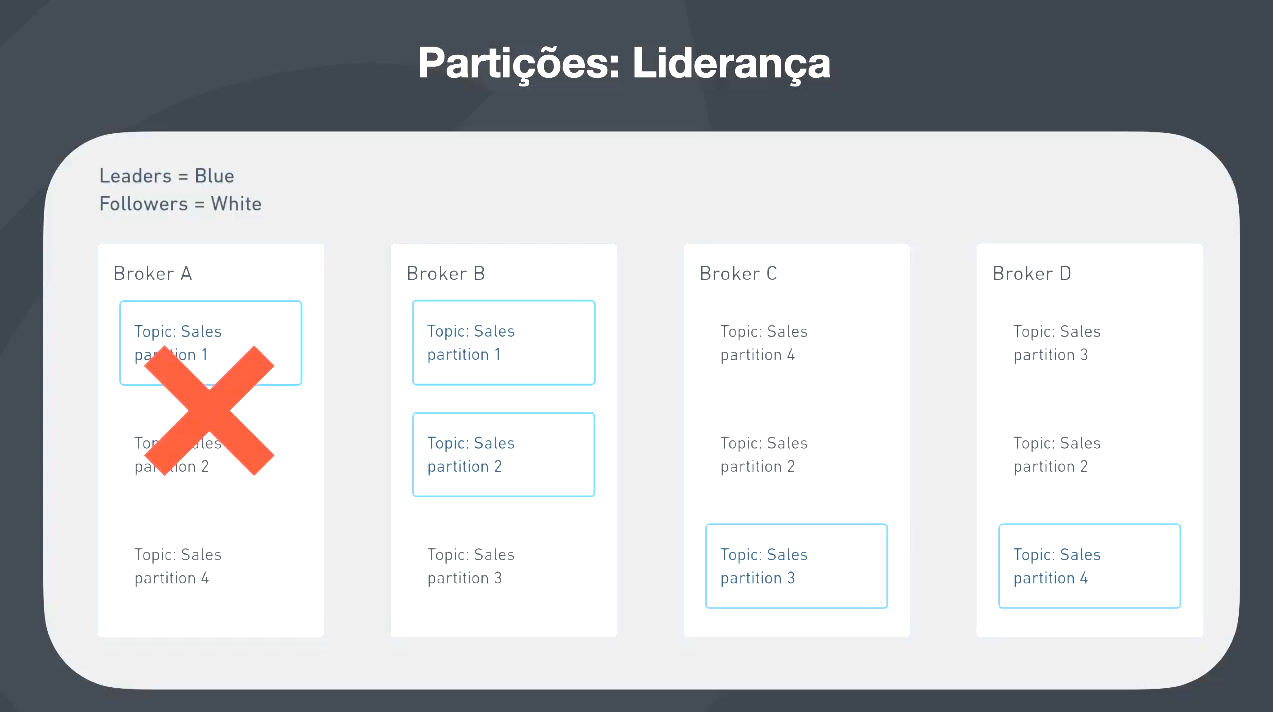
Em contrapartida, quanto maior o Replication Factor, mais espaço em disco vai consumir, mais o Broker será consumido.

Normalmente se utiliza 2 ou 3, sendo 2 o mínimo recomendado. 3 quando é algo muito crítico.

# Partition leadership

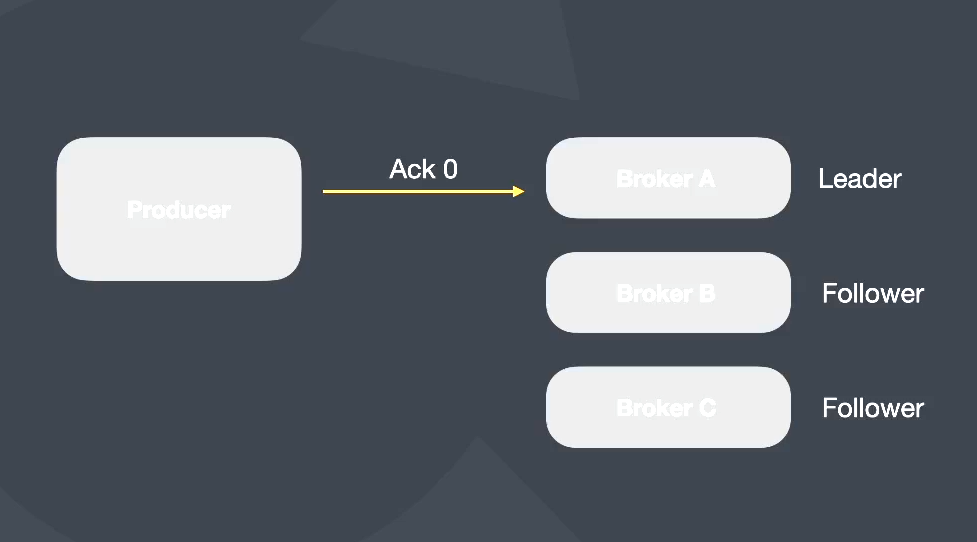


A ilustração acima demonstra 4 Brokers com algumas partições replicadas. As partições em azul são as chamadas partições líderes. Isso significa que, quando um consumidor vai ler um tópico, necessariamente ele irá ler a partição líder. SEMPRE.

Se o Broker A cair, por exemplo, o Kafka irá procurar em qual outro Broker possui uma partição igual ao do líder que estava no Broker A e a transforma em líder.  


As partições reservas, são chamadas de Follower.

# Garantia de entrega de mensagens



A ilustração acima ilustra 3 Brokers, sendo o A com uma partição leader e o B e C, Followers.

O Producer então envia uma mensagem à partição leader, ao enviar a mesagem, ele pode setar um parâmetro chamado Ack.

## ACK 0

O ACK com valor 0 significa que ao enviar uma mensagem não é preciso um retorno informando que a mensagem foi recebida. Sendo assim, o Kafka consegue processar mais mensagens. Então por um lado não se tem certeza do recebimento da mensagem mas por outro se tem um maior poder de processamento do Kafka.

## ACK 1

O ACK com valor 1 significa que ao enviar uma mensagem é preciso um retorno informando que a mensagem foi recebida. Então o leader, ao receber a mensagem, irá salvar a mensagem e enviar uma mensagem de retorno informando que recebeu a mensagem. Após isso, replica nos Followers.

Cenário hipotético: O Producer envia uma mensagem ao Broker A , O Broker A salvou a mensagem e enviou o retorno ao Producer. Logo que isso aconteceu, o Broker A caiu. Não houve tempo de replicar a mensagem recebida nas partições Followers. Neste cenário, houve perda de mensagem e o Producer “acreditou” que essa mensagem foi salva. Percebe-se que ao utilizar ACK 1, a velocidade do processo fica um pouco mais lenta e pode gerar alguns cenários inesperados.

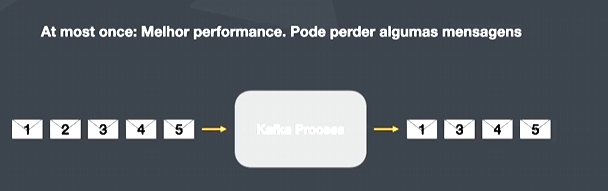
## ACK – 1

Seguindo o exemplo anterior, imagine que não pode nem haver a possibilidade desta perda hipotética. Para isso tem-se o ACK -1. Isso significa que o Producer envia uma mensagem ao Leader, o Leader irá replicar a mensagem nos Followers. Os Followers irão avisar o leader que as mensagens foram salvas e, então, o leader avisa o Producer que a mensagem foi salva. Neste caso se tem uma garantia total que a mensagem foi salva e replicada. O problema é que o processo é bem mais lento.

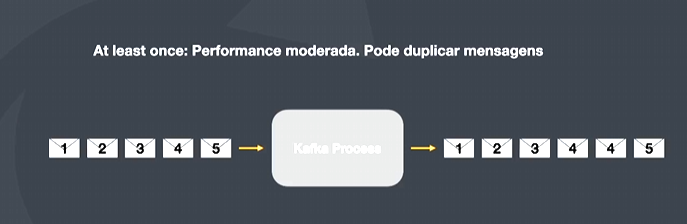
## Performance vs Garantias

Tipos de garantias:

* At most once: Melhor performance. Pode perder algumas mensagens.



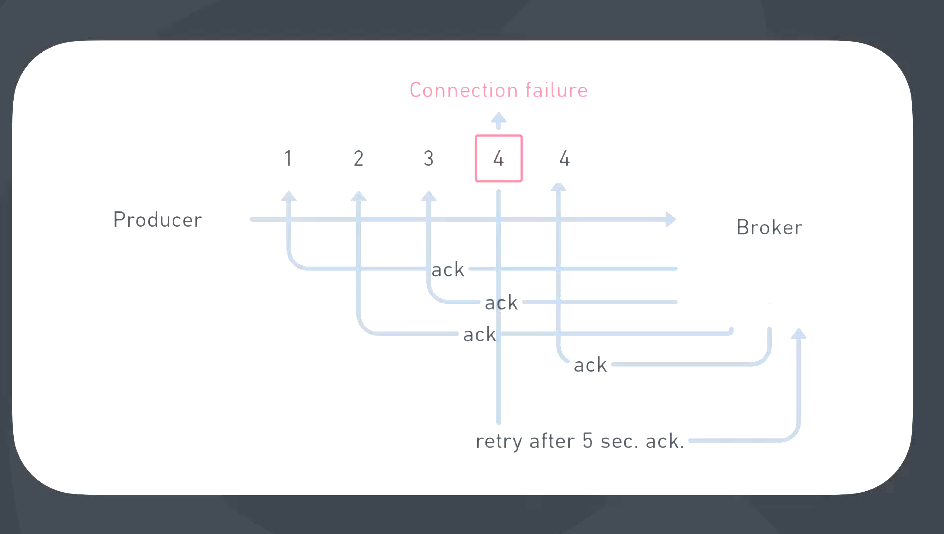
* At least once: Performance moderada. Garante a entrega da mensagem pelo menos uma vez. Pode duplicar mensagens. Neste caso é necessário ter tratamentos no sistema para que o mesmo saiba que determinada mensagem já foi processada e não a processe novamente.



* Exactly one: Pior performance. Exatamente uma vez.

# Produtor indepotente

Cenário: Producer envia a mensagem 1 ao Broker. Broker retorna ao Producer informando que recebeu a mensagem. O mesmo ocorre nas mensagens 2 e 3. Quando o Producer envia a mensagem 4, ocorre um problema de rede. Então o Producer envia a mensagem 4 novamente e recebe o retorno. Porém, em algum momento houve um retry da mensagem que ficou presa na rede e a mesma acabou indo. Desta forma houve uma duplicidade da mensagem.



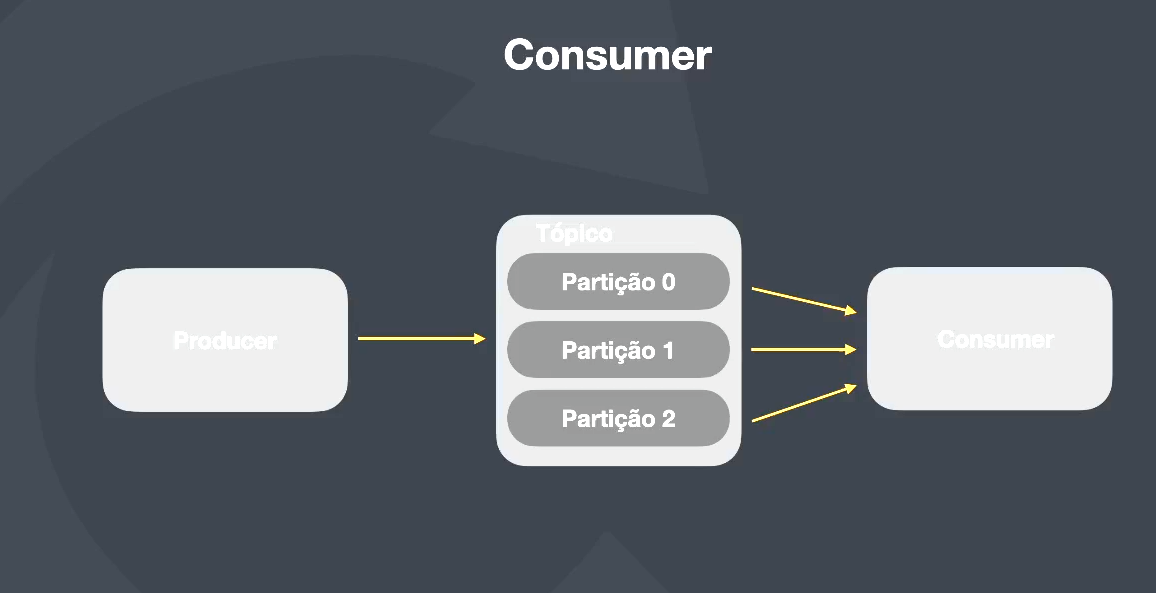
Neste cenário pode-se dizer que a indepotência do Producer é off. Neste caso a mensagem 4 será gravada duplicada e o consumir irá lê-a 2 vezes.

Como evitar esse tipo de problema?

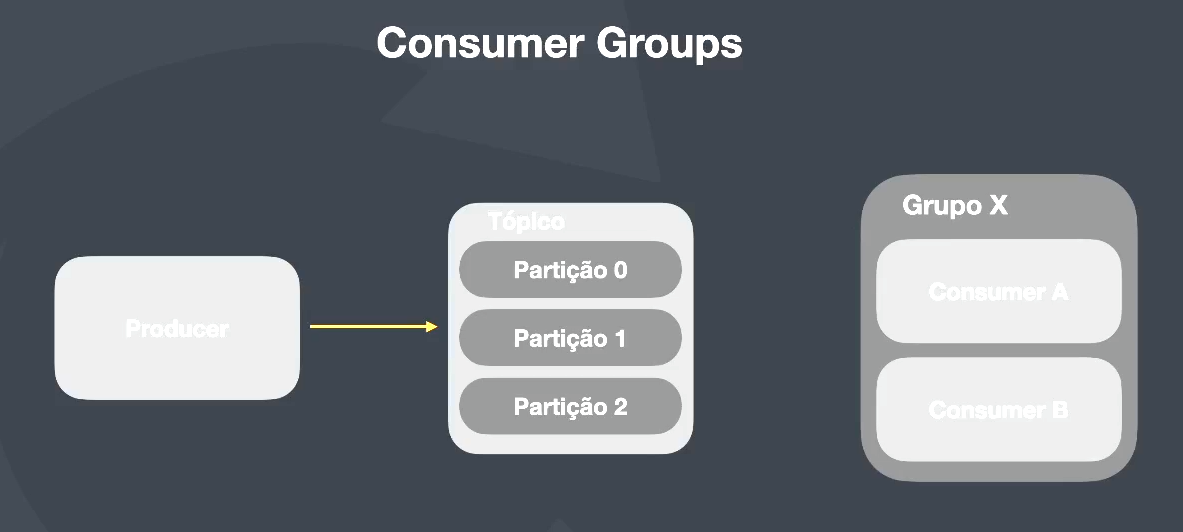
Utilizando produtor indepotente. Quando um produto é indepotente (modo indepotencia ON), o Kafka irá entender que isso aconteceu e irá descartar uma das mensagens.

É algo que gera maior lentidão, porém o Kafka consegue iniciar a mensagem e ao mesmo tempo garantir que a mensagem caia na ordem correto (possui timestamps).

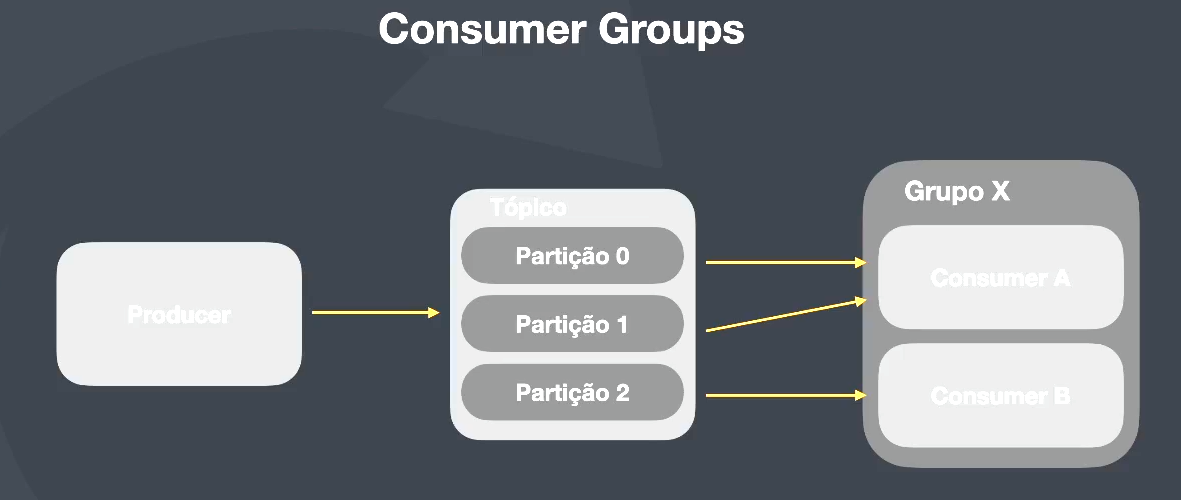
# Consumers e consumer groups



No cenário acima têm-se um Producer enviando uma mensagem a um tópico com 3 partições e apenas um Consumer lendo essas mensagens. Em relação a velocidade e processamento de mensagens, nada muda.



No cenário acima têm-se um Producer enviando uma mensagem a um tópico com 3 partições e agora 2 Consumers pertencentes a um mesmo Group. Consumer A e B podem ser, por exemplo, o mesmo software rodando em máquinas diferentes ou em processos diferentes. Eles fazem a mesma coisa.

Ao colocar os Consumers no mesmo grupo (X), ocorre o seguinte:  


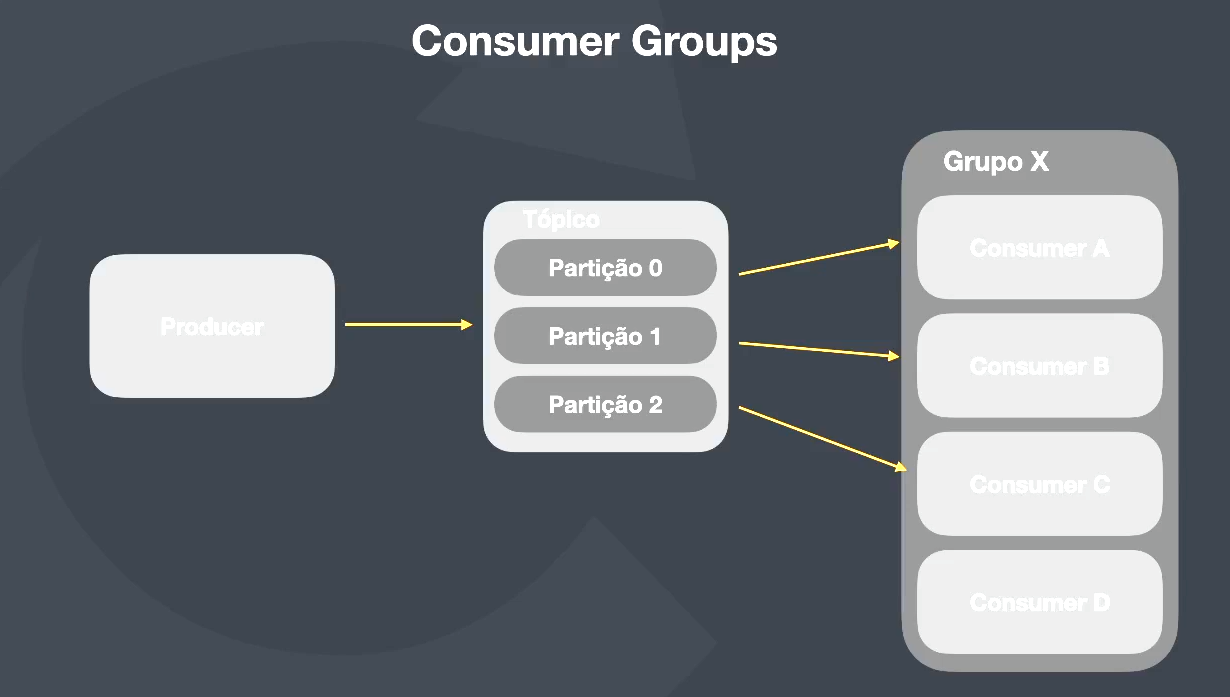
Partições 0 e 1 serão lidas pelo Consumer A , partição 2 pelo Consumer B.

Se por exemplo houvesse também um Consumer Y, que não estivesse no group X, esse Consumer estaria lendo a partição 0, 1 e 2. Ou seja, a leitura entre as partições só está sendo dividida entre os Consumers pois estes estão no mesmo Group. Sem Group o Kafka entende que o Consumer é o próprio grupo.

Outro cenário é o seguinte:

Imagine que há 3 partições dentro do tópico e que há 4 Consumers dentro do mesmo Group. Neste caso, um dos Consumers ficará ocioso. Não tem como 2 Consumers dentro do mesmo Group lerem a mesma partição. Se houver mais Consumers que partição, os excedentes ficarão parados.

O melhor cenário existente é ter a mesa quantidade de partições e Consumers.



# Docker compose para utilizar o Kafka (3 brokers)

version: "3"

services:

zookeeper:

image: confluentinc/cp-zookeeper:7.8.0

environment:

ZOOKEEPER\_CLIENT\_PORT: 2181

extra\_hosts:

- "host.docker.internal:172.17.0.1"

kafka1:

image: confluentinc/cp-kafka:7.8.0

depends\_on:

- zookeeper

ports:

- "9092:9092"

- "9094:9094"

environment:

KAFKA\_BROKER\_ID: 1

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 3

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: INTERNAL

KAFKA\_LISTENERS: INTERNAL://:9092,OUTSIDE://:9094

KAFKA\_ADVERTISED\_LISTENERS: INTERNAL://kafka1:9092,OUTSIDE://host.docker.internal:9094

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: INTERNAL:PLAINTEXT,OUTSIDE:PLAINTEXT

#transaction.state.log.replication.factor: 3 e transaction.state.log.min.isr: 2: - Padrão na criação

#Essas configurações indicam que o Kafka espera que o log de estado de transação seja replicado em três

#brokers, com pelo menos dois deles em sincronia. Isso requer um cluster com pelo menos três brokers para

#funcionar corretamente.

#Timeout expired while initializing transactional state in 60000ms.

#This means your cluster doesn’t have enough brokers to support transactions.

#You probably just have one broker because you’re developing locally.

#You have to update the broker configuration parameters:

#Resumindo, se tiver apenas um broker, configurar as variaveis abaixo com 1

# KAFKA\_TRANSACTION\_STATE\_LOG\_REPLICATION\_FACTOR: 1

# KAFKA\_TRANSACTION\_STATE\_LOG\_MIN\_ISR: 1

extra\_hosts:

- "host.docker.internal:172.17.0.1"

kafka2:

image: confluentinc/cp-kafka:7.8.0

depends\_on:

- zookeeper

ports:

- "9093:9093"

- "9095:9095"

environment:

KAFKA\_BROKER\_ID: 2

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 3

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: INTERNAL

KAFKA\_LISTENERS: INTERNAL://:9093,OUTSIDE://:9095

KAFKA\_ADVERTISED\_LISTENERS: INTERNAL://kafka2:9093,OUTSIDE://host.docker.internal:9095

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: INTERNAL:PLAINTEXT,OUTSIDE:PLAINTEXT

#transaction.state.log.replication.factor: 3 e transaction.state.log.min.isr: 2: - Padrão na criação

#Essas configurações indicam que o Kafka espera que o log de estado de transação seja replicado em três

#brokers, com pelo menos dois deles em sincronia. Isso requer um cluster com pelo menos três brokers para

#funcionar corretamente.

#Timeout expired while initializing transactional state in 60000ms.

#This means your cluster doesn’t have enough brokers to support transactions.

#You probably just have one broker because you’re developing locally.

#You have to update the broker configuration parameters:

#Resumindo, se tiver apenas um broker, configurar as variaveis abaixo com 1

# KAFKA\_TRANSACTION\_STATE\_LOG\_REPLICATION\_FACTOR: 1

# KAFKA\_TRANSACTION\_STATE\_LOG\_MIN\_ISR: 1

extra\_hosts:

- "host.docker.internal:172.17.0.1"

kafka3:

image: confluentinc/cp-kafka:7.8.0

depends\_on:

- zookeeper

ports:

- "9096:9096"

- "9097:9097"

environment:

KAFKA\_BROKER\_ID: 3

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 3

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: INTERNAL

KAFKA\_LISTENERS: INTERNAL://:9096,OUTSIDE://:9097

KAFKA\_ADVERTISED\_LISTENERS: INTERNAL://kafka3:9096,OUTSIDE://host.docker.internal:9097

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: INTERNAL:PLAINTEXT,OUTSIDE:PLAINTEXT

#transaction.state.log.replication.factor: 3 e transaction.state.log.min.isr: 2: - Padrão na criação

#Essas configurações indicam que o Kafka espera que o log de estado de transação seja replicado em três

#brokers, com pelo menos dois deles em sincronia. Isso requer um cluster com pelo menos três brokers para

#funcionar corretamente.

#Timeout expired while initializing transactional state in 60000ms.

#This means your cluster doesn’t have enough brokers to support transactions.

#You probably just have one broker because you’re developing locally.

#You have to update the broker configuration parameters:

#Resumindo, se tiver apenas um broker, configurar as variaveis abaixo com 1

# KAFKA\_TRANSACTION\_STATE\_LOG\_REPLICATION\_FACTOR: 1

# KAFKA\_TRANSACTION\_STATE\_LOG\_MIN\_ISR: 1

extra\_hosts:

- "host.docker.internal:172.17.0.1"

control-center:

image: confluentinc/cp-enterprise-control-center:7.8.0

hostname: control-center

depends\_on:

- kafka1

- kafka2

- kafka3

ports:

- "9021:9021"

environment:

CONTROL\_CENTER\_BOOTSTRAP\_SERVERS: 'kafka1:9092,kafka2:9093,kafka3:9096'

CONTROL\_CENTER\_REPLICATION\_FACTOR: 3

CONTROL\_CENTER\_CONNECT\_CLUSTER: http://app-connect:8083

PORT: 9021

extra\_hosts:

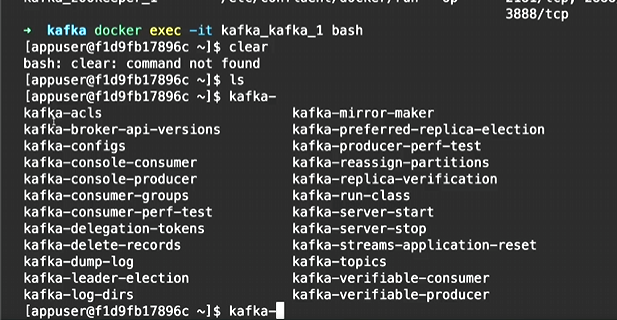
- "host.docker.internal:172.17.0.1"

Algumas explicações para o uso do Docker compose acima:

* Devem haver as seguintes configurações no /etc/hosts
  + host.docker.internal 127.0.0.1
  + kafka1 kafka2 kafka3 127.0.0.1
* Para acessar o container:
  + docker-compose os -> para descobrir o nome do serviço
  + docker exec -it npme-do-serviço bash -> para entrar no container

# Alguns comandos manuais do Kafka

Ao digitar Kafka- serão listados diversos comandos do Kafka.



## Observação importante

Em alguns casos, ou talvez algumas versões do Kafka, pode-se receber uma mensagem de “command not found”. Se for o caso seguem algumas sugestões:

1. **Verifique o PATH**: Certifique-se de que o diretório onde o Kafka está instalado está incluído no PATH. Você pode verificar o PATH atual executando echo $PATH no terminal do container.
2. **Localize o Kafka**: Tente localizar o diretório onde o Kafka está instalado. Normalmente, os binários do Kafka estão em um diretório como /opt/kafka/bin ou /usr/local/kafka/bin. Você pode usar o comando find / -name kafka-\* para procurar os binários do Kafka no container.
3. **Execute o comando completo**: Se você encontrar o diretório onde o Kafka está instalado, tente executar o comando completo, por exemplo, /opt/kafka/bin/kafka-topics.sh em vez de apenas kafka-.
4. **Verifique a instalação do Kafka**: Certifique-se de que o Kafka está realmente instalado no container. Se não estiver, você pode precisar modificar o Dockerfile ou o script de inicialização para incluir a instalação do Kafka.
5. **Verifique a imagem Docker**: Se você estiver usando uma imagem Docker personalizada, verifique se o Kafka foi instalado corretamente durante o build da imagem.

## Criar tópico

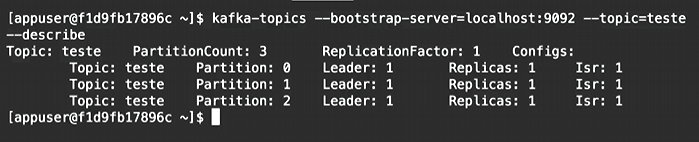
* kafka-topics --create–topic=teste --bootstrap-server=localhost:9092 --partitions=3

## Lista tópicos

* kafka-topic --list --bootstrap-server=localhost:9092
  + Alguns tópicos são criados pelo próprio kafka e outras ferramentas
  + O \_\_consumer\_offset é onde é guardado qual offset determinado Consumer está no momento da leitura dos tópicos.

## Detalhar tópicos

* kafka-topics --bootstrap-server=localhost:9092 --topic=teste –describe

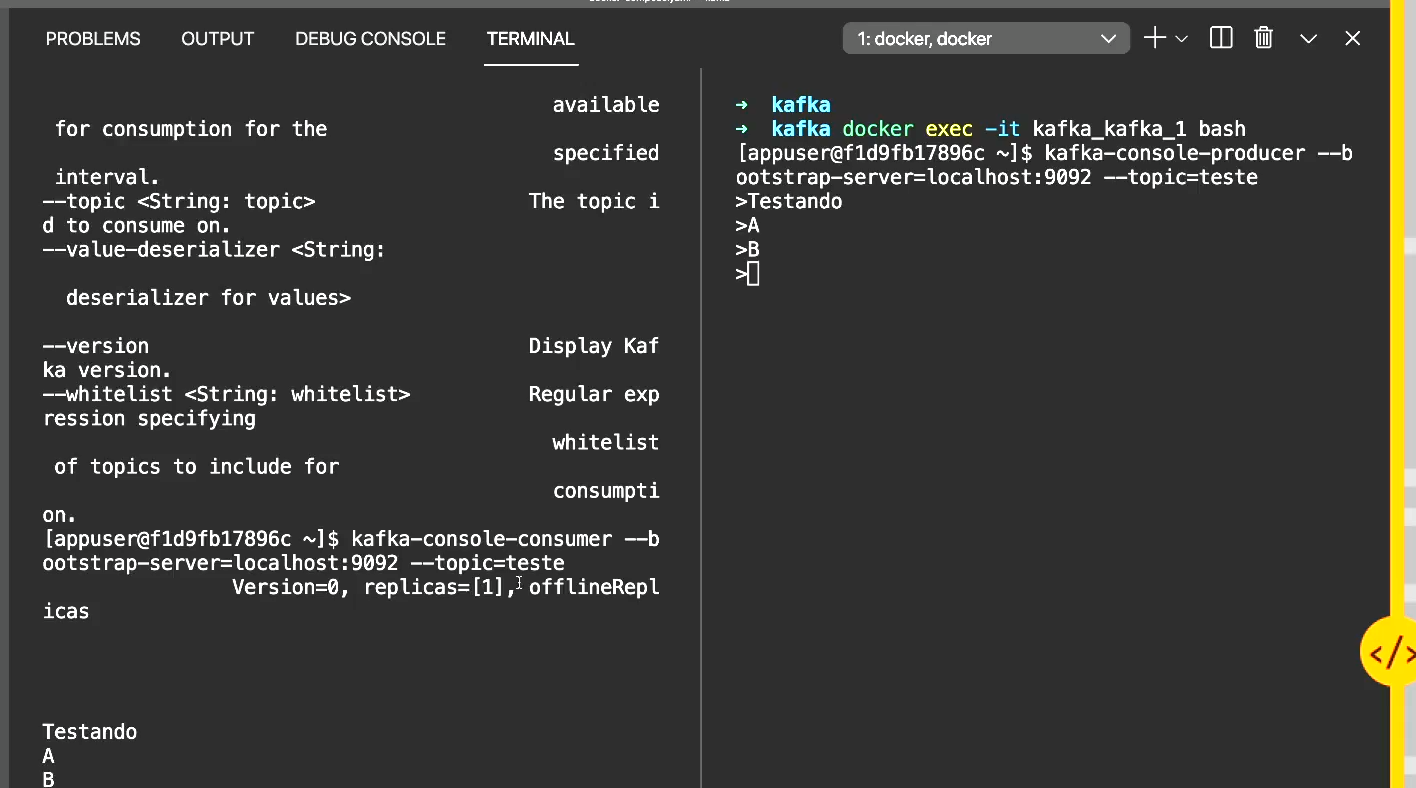


## Iniciar um Consumer

* kafka-console-consumer --bootstrap-server=localhost:9092 --topic=teste

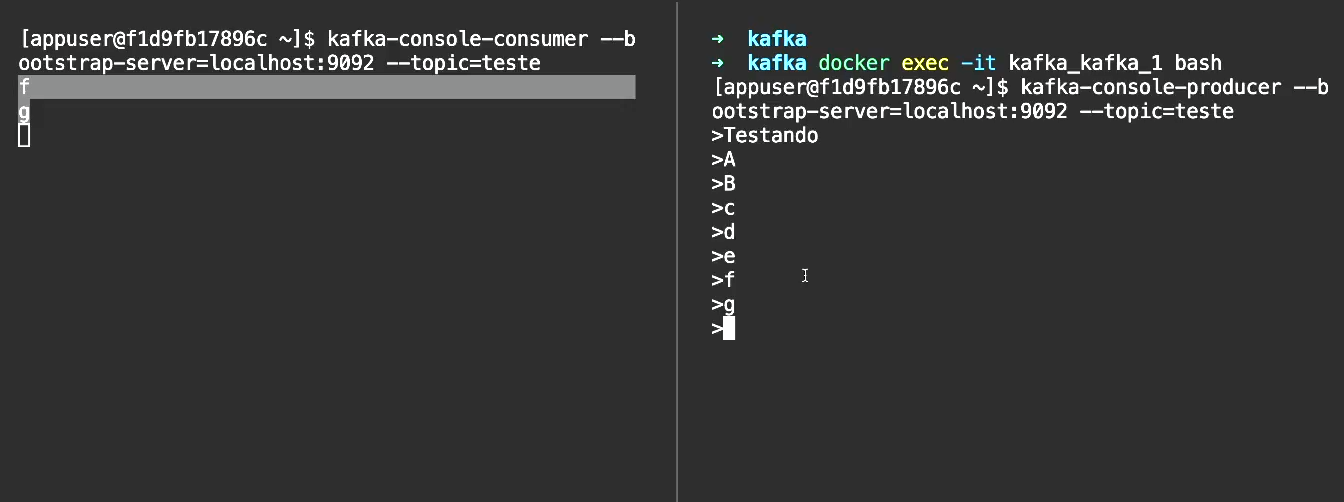
## Configurando o Producer e enviando mensagem

* kafka-console-producer --bootstrap-server=localhost:9092 --topic=teste
  + Irá solicitar que a mensagem seja escrita e enviada- escreva a mensagem. (Na imagem abaixo o lado esquerdo é o terminal do Consumer e lado esquerdo do Producer)



### Teste com envio de mensagens

* Derrube o Consumer
* Envie as mensagens c,d, e
* Acesse o Consumer de novo
* Envie as mensagens f, g



O Consumer irá ler a partir das mensagens enviadas no momento em que o mesmo foi iniciado novamente. Isso não significa que as mensagens foram perdidas.

Para ler as mensagens desde o começo precisa rodar o seguinte comando:

* kafka-console-consumer --bootstrap-server=localhost:9092 --topic=teste --from-beginning



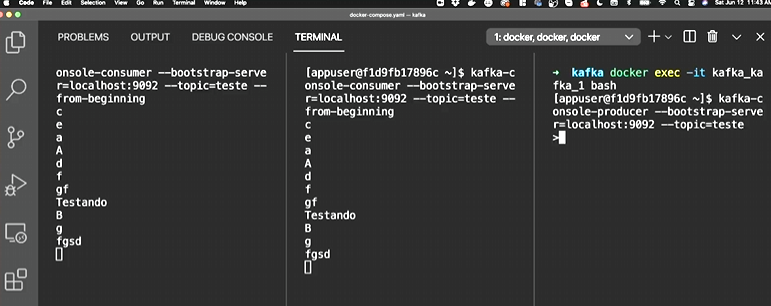
Todas as mensagens foram lidas, porém fora de ordem, pois cada mensagem foi para uma partição diferente (atenção para o que foi explicado na sessão ”Informações importantes sobre envio de mensagens para partições”)

. Para garantir a ordem das mensagens é necessário da key.

## Testes com Consumer Group

Inicie 2 Consumers e 1 Producer com os comandos:

* kafka-console-consumer --bootstrap-server=localhost:9092 --topic=teste --from-beginning
* kafka-console-producer --bootstrap-server=localhost:9092 --topic=teste



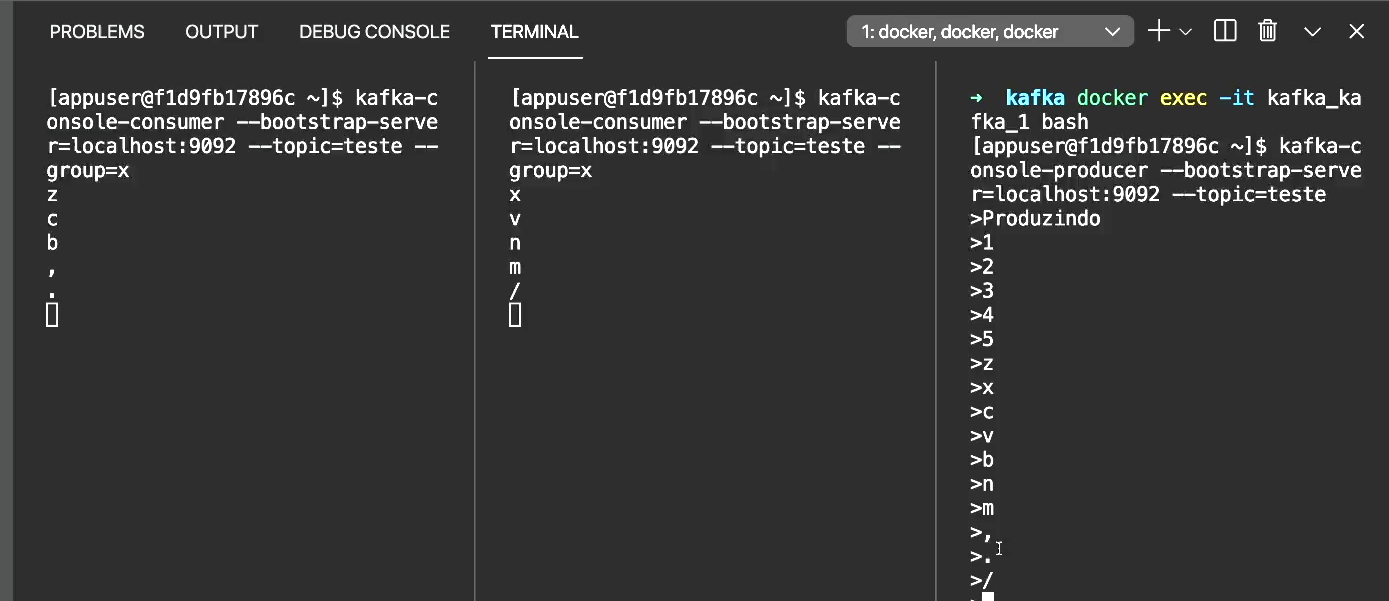
Envie mensagens e repare que os 2 Consumers estão recebendo as mesmas. Isto pois são consumidores independentes, não estão no mesmo Group.



Agora mate os Consumers e suba novamente (sem --from-beginning para facilitar visualização) adicionando um grupo a eles:

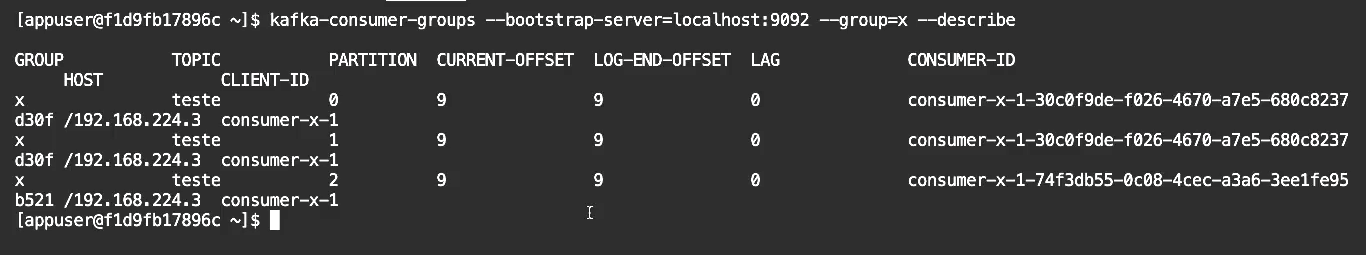
* kafka-console-consumer --bootstrap-server=localhost:9092 --topic=teste –group=x

Envie novas mensagens e repare que as mensagens serão lidas entre os 2 Consumers, sem se repetirem em 2 Consumers.



## Por dentro de um Consumer Group

* kafka-consumer-groups --bootstrap-server=localhost:9092 -- group=x --describe



Os 3 primeiros atributos mostrados são intuitivos.

O CURRENT-OFFSET – é o offset sendo lido atualmente.

O LOG-END-OFFSET – é o último offset.

LAG – É a quantidade de mensagens não lidas.

No exemplo acima é possível ver também que o Consumer com ID final d30f está lendo as partições 0 e 1. Já o Consumer com ID final b521 está lendo a partição 2.

# Kafka Connect

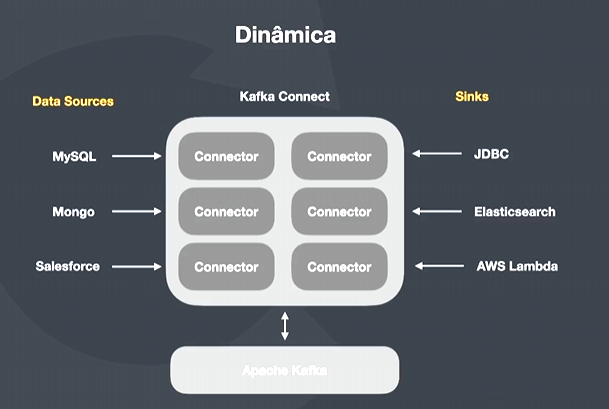
Kafka Connect é um componente gratuito e open-source do Apachae Kafka que trabalha como um hub de dados centralizado para integração simples entre banco de dados, key-value stores, search indexes e file systems.

## Dinâmica

Cenário: Um CRM rodando no Salesforce. As informações desse CRM precisam ser migradas para um outro sistema com um banco Mysql. Então é possível conectar o Kafka connect no Salesforce, pegar os dados do Salesforce e jogar no Kafka (ficará gravado no tópico do Kafka), e então pegar os dados gravados neste tópico e jogar no Mysql.

O Kafka Connect fica o tempo todo em contato com o Kafka. Todos os dados que o Kafka Connect for persistir será persistido no Kafka. No Connect existe algo chamado Connector. Podem haver diversos tipos de Connectors (Mysql, Mongo, ElasticSearch, Salesforce, entre outros).

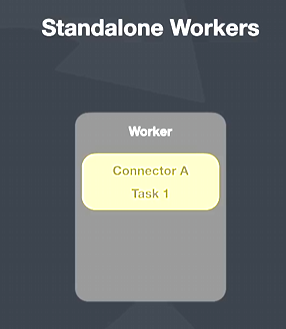
De um lado existem Connectors focados em Data sources, ou seja, Connectors que pegam dados de algum lugar e jogam no Apache Kafka.



De outro lado existem os Connectors chamados de Sinks, que basicamente é onde a informação será jogada. Então pode-se, por exemplo, obter o dado do MySQL e jogar para o AWS lambda.

# Standalone Workers

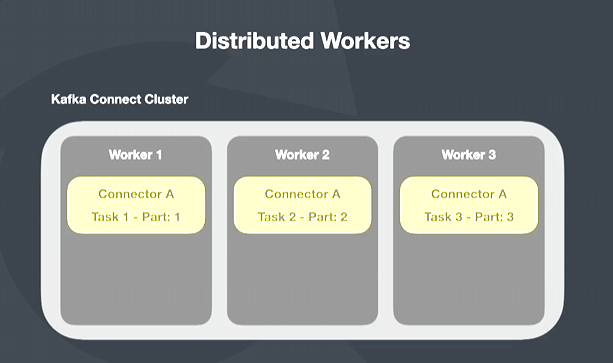
O Kafka Connect é formado por Workers, que nada mais é que máquinas. Existe um tipo de Worker chamado standalone(é apenas 1 worker). Ele é responsável por executar as tarefas que precisamos rodar. Como exemplo, imagine um Connector 1 que fica rodando uma tarefa 1 para obter os dados do Mysql e jogar no Kafka. A tarefa é o processo que fica rodando no Worker para ficar usando o Connector para pegar os dados e jogar no Kafka ou vice versa.

   
A tarefa é algo de extrema importância para se ter conhecimento, pois pode haver algum erro ou então parar.

É possível que haja N tarefas dentro do Worker. Então é possível, por exemplo, haver um Connector B com uma Tarefa 2, rodando em paralelo com a tarefa 1.

# Distributed Workers

Neste caso, o Worker trabalha em um cluster. Imagina que tenham 3 Workers e que é necessário que eles se ajudem a processar as tarefas existentes. Quando se tem mais de um Worker dentro de um mesmo group eles irão “dividir” essas tarefas para que seja possível processar as informações mais rapidamente.



No exemplo acima se tem um Connector A que possui a tarefa 1 que lê a partição 1.

Têm-se o mesmo Connector A com a Tarefa 2 que vai ler a partição 2. Têm-se o mesmo Connector A com a Tarefa 3 que vai ler a partição 3. Suponha que haja um tópico com muita informação e seja necessário processar essas informações. Desta forma essas informações serão processadas mais rapidamente. Ao trabalhar desta forma, o Kafka Connect sabe que esses Workers estão no mesmo group formando um cluster e eles se organizam para decidir que lê qual partição e trabalham juntos.

# Converters

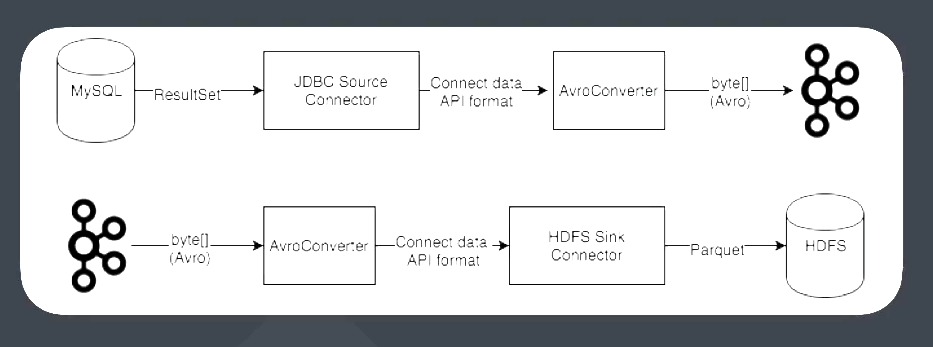
As tasks utilizam os Converters para mudar o formato dos dados para leitura ou escrita no Kafka. Por exemplo, há uma informação em um determinado formato que precisa ser gravada no Kafka. Para isso o Kafka Connect precisa ler e entender este formato para conseguir gravar.

Para isso existem diversos formatos de dados como:

* Avro: Um dos mais utilizados.
* Protobuf
* JsonSchema
* Json
* String
* ByteArray

Suponha que haja um Mysql em que é necessário obter as informações do Mysql e jogar esses dados no Connector JDBC. Essa informação irá trabalhar com o formato do Avro. Então a informação será convertida em Avro e armazenada no Kafka.

Do outro lado, a informação será obtida em Avro, será reconvertida e jogada em outro tipo de disco/fonte de dados.



# DLQ – Dead Letter Queue

Existem Converters e existem vários passos até o dado chegar e ser gravado no Kafka, ou para ser lido ou gravado em outro lugar, Muitas vezes esse dado pode não estar no formato desejado ou pode ocorrer algum erro de processamento. Quando esses erros ocorrem, é necessário saber o que fazer. Para isso se tem o DLQ.

DLQ é um local onde são jogadas essas informações que deram erros para serem posteriormente analisadas. Quando há um registro inválido, independente da razão, o erro pode ser tratado nas configurações do conector através da propriedade erros.tolerance. Esse tipo de configuração pode ser realizado apenas para contectores do tipo Sink. Ou seja, os dados já estão no Kafka, e o desejo e obter esses dados e jogá-los em algum lugar.

Tipos de tolerância (configurados no errors.tolerance):

* none: Faz a tarefa falhar imediatamente. Quando a tarefa falha, neste caso, ela para a task. Ou seja, quando há uma falha em uma tarefa do WOrker configurada como sem tolerância, a tarefa vai parar. Será necessário olhar o log para entender os motivos.
* all: Erros são ignorados e o processo continua normalmente. O problema é que não será possível saber que esse erro aconteceu. Se olhar no log do Kafka Connect, do respectivo worker, o erro não aparecerá lá.
* erros.deadletterqueue.topic.name=<nome do tópico> : Para que essa opção seja ativada, a tolerância de erros deve ser configurada como all. Sempre que houver um erro, a mensagem em que o erro ocorreu será jogada no tópico configurado. Para saber o motivo do erro, deverá consultar o cabeçalho dessas mensagens jogadas neste tópico.

No Confluent Hub pode-se encontrar diversos Connectors.

# Kafka Connect na prática

## Docker compose

version: "3"

services:

mysql:

image: wesleywillians/mysql-kafka-connect:latest

command: --innodb-use-native-aio=0

tty: true

restart: always

ports:

- "33600:3306"

environment:

- MYSQL\_DATABASE=fullcycle

- MYSQL\_ROOT\_PASSWORD=root

- MYSQL\_USER=root

extra\_hosts:

- "host.docker.internal:172.17.0.1"

zookeeper:

image: confluentinc/cp-zookeeper:latest

environment:

ZOOKEEPER\_CLIENT\_PORT: 2181

kafka:

image: confluentinc/cp-kafka:latest

depends\_on:

- zookeeper

ports:

- "9092:9092"

- "9094:9094"

environment:

KAFKA\_BROKER\_ID: 1

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: INTERNAL

KAFKA\_LISTENERS: INTERNAL://:9092,OUTSIDE://:9094

KAFKA\_ADVERTISED\_LISTENERS: INTERNAL://kafka:9092,OUTSIDE://host.docker.internal:9094

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: INTERNAL:PLAINTEXT,OUTSIDE:PLAINTEXT

extra\_hosts:

- "host.docker.internal:172.17.0.1"

control-center:

image: confluentinc/cp-enterprise-control-center:latest

hostname: control-center

depends\_on:

- kafka

ports:

- "9021:9021"

environment:

CONTROL\_CENTER\_BOOTSTRAP\_SERVERS: 'kafka:9092'

CONTROL\_CENTER\_REPLICATION\_FACTOR: 1

CONTROL\_CENTER\_CONNECT\_CLUSTER: http://kafka-connect:8083

PORT: 9021

extra\_hosts:

- "host.docker.internal:172.17.0.1"

kafka-connect:

image: confluentinc/cp-kafka-connect-base:6.0.0

container\_name: kafka-connect

depends\_on:

- zookeeper

- kafka

ports:

- 8083:8083

environment:

CONNECT\_BOOTSTRAP\_SERVERS: "kafka:9092"

CONNECT\_REST\_PORT: 8083

CONNECT\_GROUP\_ID: kafka-connect

CONNECT\_CONFIG\_STORAGE\_TOPIC: \_connect-configs

CONNECT\_OFFSET\_STORAGE\_TOPIC: \_connect-offsets

CONNECT\_STATUS\_STORAGE\_TOPIC: \_connect-status

CONNECT\_KEY\_CONVERTER: org.apache.kafka.connect.storage.StringConverter

CONNECT\_VALUE\_CONVERTER: org.apache.kafka.connect.json.JsonConverter

CONNECT\_INTERNAL\_KEY\_CONVERTER: "org.apache.kafka.connect.json.JsonConverter"

CONNECT\_INTERNAL\_VALUE\_CONVERTER: "org.apache.kafka.connect.json.JsonConverter"

CONNECT\_REST\_ADVERTISED\_HOST\_NAME: "kafka-connect"

CONNECT\_LOG4J\_ROOT\_LOGLEVEL: "INFO"

CONNECT\_LOG4J\_LOGGERS: "org.apache.kafka.connect.runtime.rest=WARN,org.reflections=ERROR"

CONNECT\_LOG4J\_APPENDER\_STDOUT\_LAYOUT\_CONVERSIONPATTERN: "[%d] %p %X{connector.context}%m (%c:%L)%n"

CONNECT\_CONFIG\_STORAGE\_REPLICATION\_FACTOR: "1"

CONNECT\_OFFSET\_STORAGE\_REPLICATION\_FACTOR: "1"

CONNECT\_STATUS\_STORAGE\_REPLICATION\_FACTOR: "1"

# # Optional settings to include to support Confluent Control Center

# CONNECT\_PRODUCER\_INTERCEPTOR\_CLASSES: "io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor"

# CONNECT\_CONSUMER\_INTERCEPTOR\_CLASSES: "io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor"

# ---------------

CONNECT\_PLUGIN\_PATH: /usr/share/java,/usr/share/confluent-hub-components,/data/connect-jars

# If you want to use the Confluent Hub installer to d/l component, but make them available

# when running this offline, spin up the stack once and then run :

# docker cp kafka-connect:/usr/share/confluent-hub-components ./data/connect-jars

volumes:

- $PWD/data:/data

# In the command section, $ are replaced with $$ to avoid the error 'Invalid interpolation format for "command" option'

command:

- bash

- -c

- |

echo "Installing Connector"

confluent-hub install --no-prompt debezium/debezium-connector-mysql:1.2.2

confluent-hub install --no-prompt mongodb/kafka-connect-mongodb:1.5.0

#

echo "Launching Kafka Connect worker"

/etc/confluent/docker/run &

#

sleep infinity

extra\_hosts:

- "host.docker.internal:172.17.0.1"

mongodb:

image: mongo:4.4.4

restart: always

volumes:

- ./.docker/dbdata:/data/db

- ./.docker/mongo:/docker-entrypoint-initdb.d

environment:

- MONGO\_INITDB\_ROOT\_USERNAME=root

- MONGO\_INITDB\_ROOT\_PASSWORD=root

- MONGO\_INITDB\_DATABASE=fullcycle

extra\_hosts:

- "host.docker.internal:172.17.0.1"

mongo-express:

image: mongo-express

restart: always

ports:

- 8085:8081

environment:

- ME\_CONFIG\_MONGODB\_SERVER=mongodb

- ME\_CONFIG\_MONGODB\_AUTH\_USERNAME=root

- ME\_CONFIG\_MONGODB\_AUTH\_PASSWORD=root

- ME\_CONFIG\_MONGODB\_ADMINUSERNAME=root

- ME\_CONFIG\_MONGODB\_ADMINPASSWORD=root

- ME\_CONFIG\_MONGODB\_ENABLE\_ADMIN=true

- ME\_CONFIG\_BASICAUTH=true

- ME\_CONFIG\_BASICAUTH\_PASSWORD=root

- ME\_CONFIG\_BASICAUTH\_USERNAME=root

depends\_on:

- mongodb

O Mysql da imagem que está sendo utilizada já está configurado com logs no formato binário. Isso é necessário para que seja possível pegar as modificações.

# Anotações importantes

## Erro ao utilizar transações - relação entre transações e número de Brokers

Idealmente, um servidor Kafka deve ser composto por um Cluster (recomendado que se tenha ao menos 3 Brokers). Algumas vezes, em ambiente de desenvolvimento por exemplo, pode-se criar apenas um Broker. Isso pode gerar erro ao tentar trabalhar com transações: ***Timeout expired while initializing transactional state in 60000ms***

Isso significa que o Cluster não possui brokers suficiente para suportar transações. Um Cluster é criado, por padrão, configurado para suportar transações que requerem múltiplos brokers.

Os valores padrão das configurações transaction.state.log.replication.factor e transaction.state.log.min.isr são 3 e 2. Essas configurações indicam que o Kafka espera que o log de estado de transação seja replicado em três brokers, com pelo menos dois deles em sincronia. Isso requer um cluster com pelo menos três brokers para funcionar corretamente.

Sendo assim, deve-se configurar essas propriedades com o valor 1 para que esse erro não ocorra quando se tem apenas 1 Broker:  
transaction.state.log.replication.factor=1  
transaction.state.log.min.isr=1

Em caso de utilização do Docker, Isso pode ser feito ao configurar o Broker, da seguinte forma:

kafka1:

image: confluentinc/cp-kafka:7.8.0

depends\_on:

- zookeeper

ports:

- "9092:9092"

- "9094:9094"

environment:

KAFKA\_BROKER\_ID: 1

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 3

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: INTERNAL

KAFKA\_LISTENERS: INTERNAL://:9092,OUTSIDE://:9094

KAFKA\_ADVERTISED\_LISTENERS: INTERNAL://kafka1:9092,OUTSIDE://host.docker.internal:9094

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: INTERNAL:PLAINTEXT,OUTSIDE:PLAINTEXT

**KAFKA\_TRANSACTION\_STATE\_LOG\_REPLICATION\_FACTOR: 1**

**KAFKA\_TRANSACTION\_STATE\_LOG\_MIN\_ISR: 1**

extra\_hosts:

- "host.docker.internal:172.17.0.1"

## Particionamento de mensagens

Particionamento é o processo que irá determinar para qual partição a mensagem será atribuída.

Até a versão 2.3 do Kafka, o particionamento padrão era o ROUND ROBIN, da 2.4 em diante passou a ser o STICKY PARTITIONER. O Sticky partitioner melhora a performance do produtor, especialmente os que possuem alto tráfego.

### Round Robin Partitioner

O particionamento Round Robin era o padrão da versão 2.3 e anterior, quando não havia partição e chave especiados. Isto resultava em mais lotes de mensagens (um lote por partição) e lotes menores. Isto era um problema pois lotes menores levam a mais requisições assim como maior latência.

### 

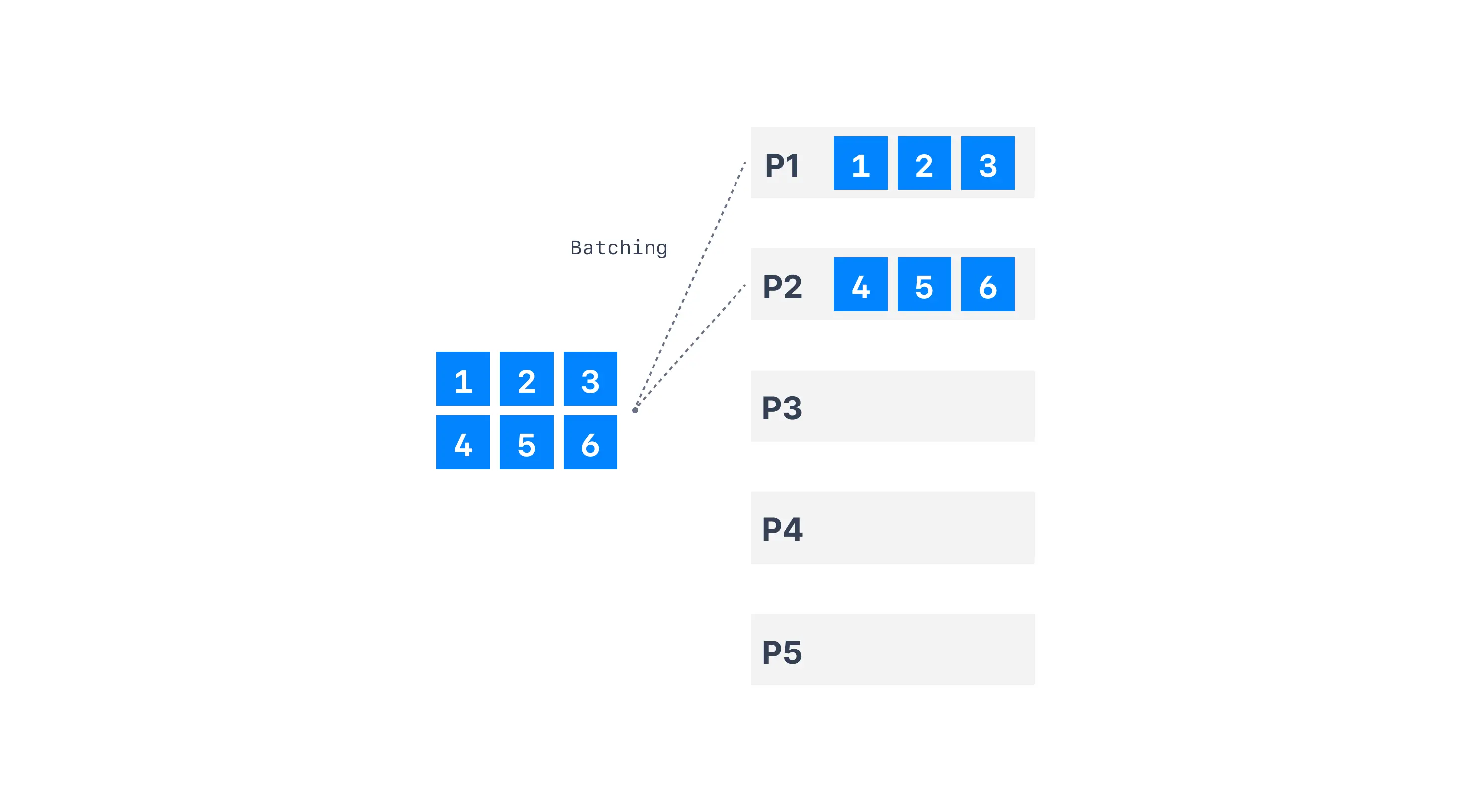
### Sticky Partitioner

É um objetivo de performance ter todos os registros enviados para uma única partição, e não para múultiplpas, para melhorar o batching.

O produtor que utiliza o stcky partitioner irá:

* manter numa partição até que o lote esteja cheio (**batch.size**) ou que o tempo de espera(**linger.ms**) tenha passado
* Após enviar o lote, altera a partição

Isso levará a lotes maiores e a uma latência reduzida (porque temos solicitações maiores e é mais provável que o tamanho do lote seja atingido). Ao longo do tempo, os registos continuam a ser distribuídos uniformemente pelas partições, então o equilíbrio do cluster não é afetado.



## ACK properties no Spring Kafka

* spring.kafka.producers.acks=all
  + Espera por um acknowledgement de todos os brokers
* spring.kafka.producers.acks=1
  + Espera por um acknowledgement somente do broker líder
* spring.kafka.producers.acks=0
  + Não espera por acknowledgement

Para a configuração all deve-se atentar ao seguinte:

Um tópico possui um replication factor. Esse replication factor corresponde a quantas réplicas de uma mensagem aquele tópico terá. Supondo que sejam 3 réplicas. Sendo assim, o acknowledgement só será retornado quando todas as 3 réplicas forem criadas e reconhecidas. (3 réplicas = 1 mensagem do líder + 2 dos Followers)

Porém existe uma configuração min.insync.replicas que diz respeito a quantas réplicas são necessárias para retornar o acknowledgement. Caso esse valor fosse 2, por exemplo, ainda que 3 réplicas sejam criadas, o acknowledgement seria retornado quando 2 réplicas fossem criadas e reconhecidas e não 3.

## Réplicas e Retry

Imagine uma configuração de 3 réplicas. Ao enviar uma mensagem ao líder, um dos Followers cai não sendo possível criar estas 3 réplicas.

O comportamento padrão do Kafka é que neste caso a mensagem será enviada novamente por um grande número de vezes até que dê certo a operação ou até que um tempo determinado seja atingido (padrão 2 minutos).

Quando um Producer envia uma mensagem a um Broker pode:

* Não haver resposta – produtor configurado com acks 0
* Receber uma resposta e na resposta conter uma confirmação de armazenamento bem sucedido (ACK)
* Non-retriable Error – Significa que houve um problema permanente que provavelmente não será solucionado com retry. Por exemplo, uma mensagem muito grande que excede o limite máximo de tamanho.
* Retriable Error – Um erro temporário que pode ser resolvido com retry. Por exemplo um erro de rede ou indisponibilidade de um Broker.

Quem toma a decisão se o erro retriable ou não é o próprio kafka Producer. Se for retriable, o Producer irá realizar tentativas de acordo com as configurações.

No Spring:

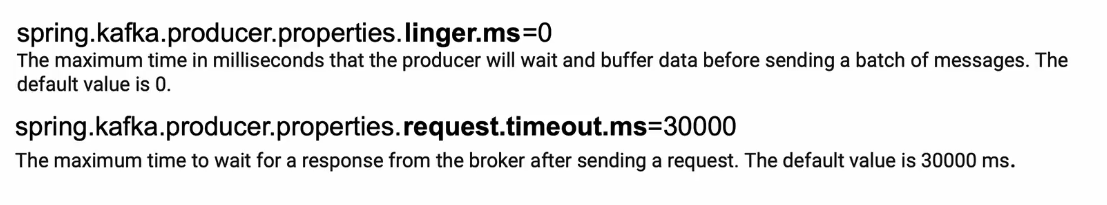
**spring.kafka.producer.retries**=10 : Quantas vezees o Producer irá tentar enviar as mensagens antes de falhar. O padrão é um valor bem alto.

**spring.kafka.producer.properties.retry.backoff.ms**=1000 : Tempo em que o produtor irá esperar antes de tentar enviar a request que falhou. O padrão é 100ms.

As 2 configurações acima são muito úteis porém a documentação do kafka encoraja a controlar o comportamento do retry com uma propriedade diferente.

**spring.kafka.producer.properties.delivery.timeout.ms :** Tempo máximo que um producer irá gastar tentando enviar uma mensagem. Isso inclui o tempo de envio da solicitação, tempo em que espera pelo ACK de todas as in sync replicas e o tempo em que gasta tentando novamente a operação. (Valor padrão – 2 minutos)

Caso decida alterar o valor padrão, deve-se atentar para que o novo valor seja igual ou maior que a soma das seguintes propriedades:

A diferença entre **spring.kafka.producer.properties.delivery.timeout.ms** e **spring.kafka.producer.properties.request.timeout.ms**

é que a primeira especifica o tempo máximo que o produtor esperará por toda a operação de envio, já a segunda é o tempo máximo que um Producer irá esperar a resposta de um Broker após o envioa da requisição.