10 Configs to Make Your Kafka Producer More Resilient

Bring your Kafka producer to the next level

Xiaoxu Gao Oct 5, 2020 · 7 min read ★

Photo by James Pond on Unsplash

Kafka is well known for its resiliency, fault-tolerance, and high throughput. But its performance doesn't always meet everyone's expectations. In some cases, we can improve it by scaling out or scaling up brokers. While in most cases, we have to *play the game of configurations*.

There are really tons of configurations in Kafka ecosystem. It's nearly impossible to grasp the idea of every single configuration. On one hand, they definitely make the system more flexible, but on the other hand, developers can feel quite confused about what to do with them.

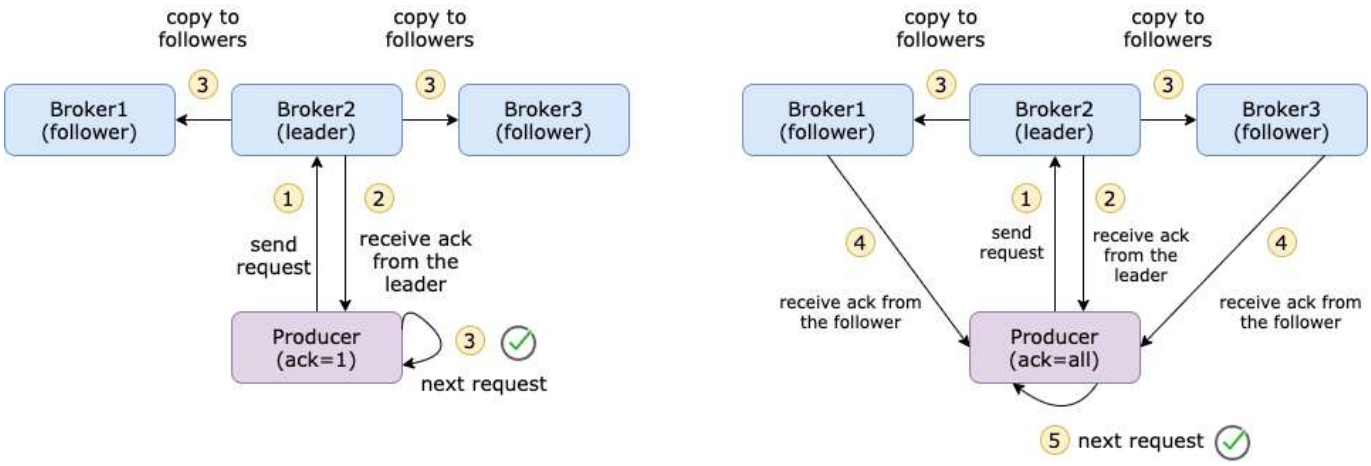
Fortunately, the majority of configurations are already pre-defined in a way that they work well for most situations. For starters, the mandatory configurations they need to know are very limited.

But of course, I assume you are reading this article because you want to bring your Kafka producer to the next level. So in this article, I want to share 10 configs that I think are important to make your producer more resilient.

The configs that will be discussed in this article are `acks`, `replica.lag.time.max.ms`, `min.insync.replicas`, `retries`, `enable.idempotent`, `max.in.flight.requests.per.connection`, `buffer.memory`, `max.block.ms`, `linger.ms`, `batch.size`.

Acks (acknowledgments)

An ack is an acknowledgment that the producer gets from a Kafka broker to ensure that the message has been successfully committed to that broker. The config `acks` is the number of acknowledgments the producer needs to receive before considering a successful commit.



Difference between ack=1 and ack=all, Created by Xiaoxu Gao

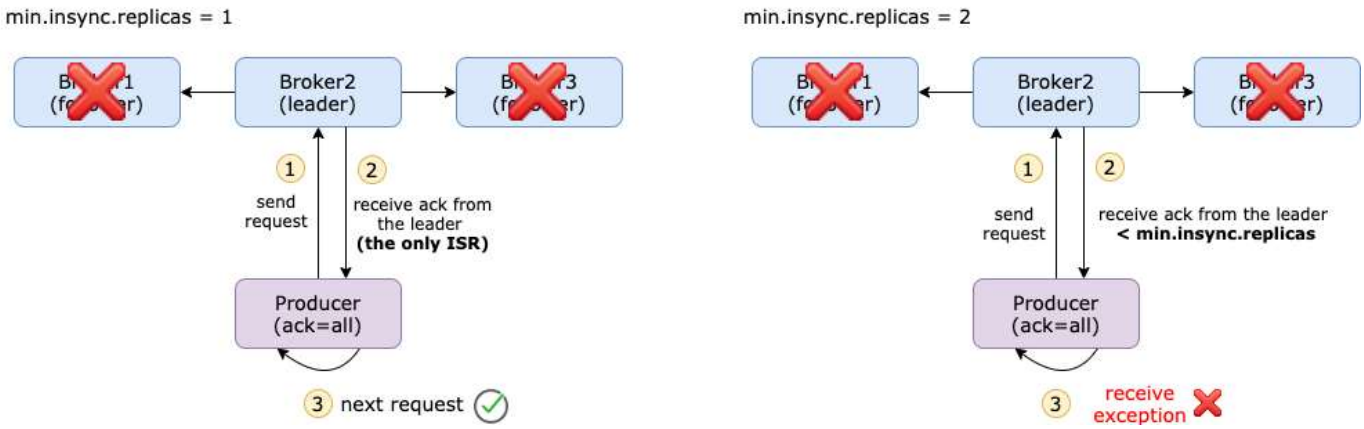
The default value is 1, which means as long as the producer receives an ack from the leader broker of that topic, it would take it as a successful commit and continue with the next message. It's not recommended to set `acks=0`, because then you don't get any guarantee on the commit. `acks=all` would make sure that the producer gets acks from all the *in-sync* replicas of this topic. It gives the strongest message durability, but it also takes long time which results in higher latency. So, you need to decide what is more important for you.

In-sync replicas

`acks=all` would get acknowledgments from all the *in-sync* replicas (ISR), so what is an in-sync replica? When you create a topic, you must define how many replicas you want. A replica is nothing more than a copy of the message in one of the brokers, so the maximum number of replicas is the number of brokers.

Among these replicas, there is a leader and the rest are followers. The leader handles all the read and write requests while the followers passively replicate the leader. **An in-sync replica is a replica that fully catches up with the leader in the last 10 seconds.** The time period can be configured via `replica.lag.time.max.ms`. If a broker goes down or has network issues, then it couldn't follow up with the leader and after 10 seconds, this broker will be removed from ISR.

The default minimum in-sync replica (`min.insync.replicas`) is 1. It means that if all the followers go down, then ISR only consists of the leader. Even if `acks` is set to all, it actually only commits the message to 1 broker (the leader) which makes the message vulnerable.



Difference between different min.insync.replicas Created by Xiaoxu Gao

The config `min.insync.replicas` basically defines how many replicas that the producer must receive before considering a successful commit. This config adds on top of `acks=all` and makes your messages safer. But on the other hand, you have to balance latency and speed.

Retry on failure

Let's say you set `acks=all` and `min.insync.replicas=2`. For some reason, the follower goes done, then the producer identifies a failed commit because it couldn't get acks from `min.insync.replicas` brokers.

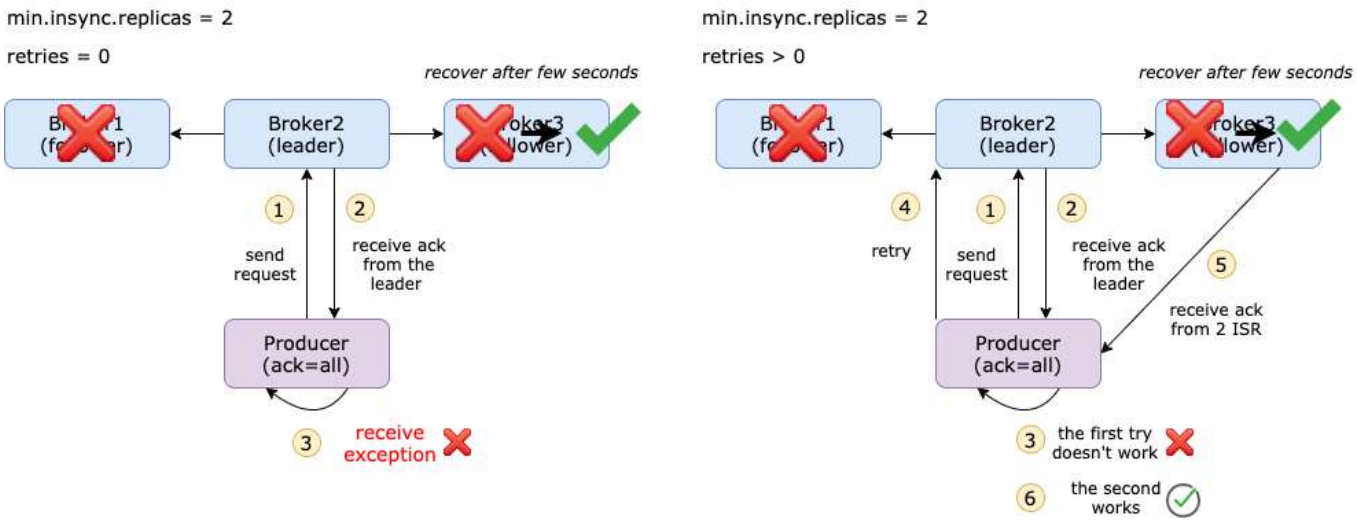
You will get an error message from the producer:

```
KafkaError{code=NOT_ENOUGH_REPLICAS,val=19,str="Broker: Not enough
in-sync replicas"}
```

You will see the following error message from the running broker. It means that even if the broker is running, Kafka will not append the messages to that running broker if the current ISR is not sufficient.

```
ERROR [ReplicaManager broker=0] Error processing append operation on
partition test-2-2-0 (kafka.server.ReplicaManager)
org.apache.kafka.common.errors.NotEnoughReplicasException: The size
of the current ISR Set(0) is insufficient to satisfy the min.isr
requirement of 2 for partition test-2-2-0
```

By default, the producer will not act upon this error, so it will lose messages. This is called **at-most-once semantics**. But you can let the producer resend messages by configuring `retries=n`. This is basically the maximum number of retries the producer would do if the commit fails. The default value is 0.



Difference between retries=0 and retries>0 Created by Xiaoxu Gao

If you set `retries=5`, then the producer will retry maximum 5 times. You will not notice the number of retries from the producer log because it only shows if the commit is successful or not in the end. But you can see `retries+1` log messages on the broker side.

Avoid duplicated messages

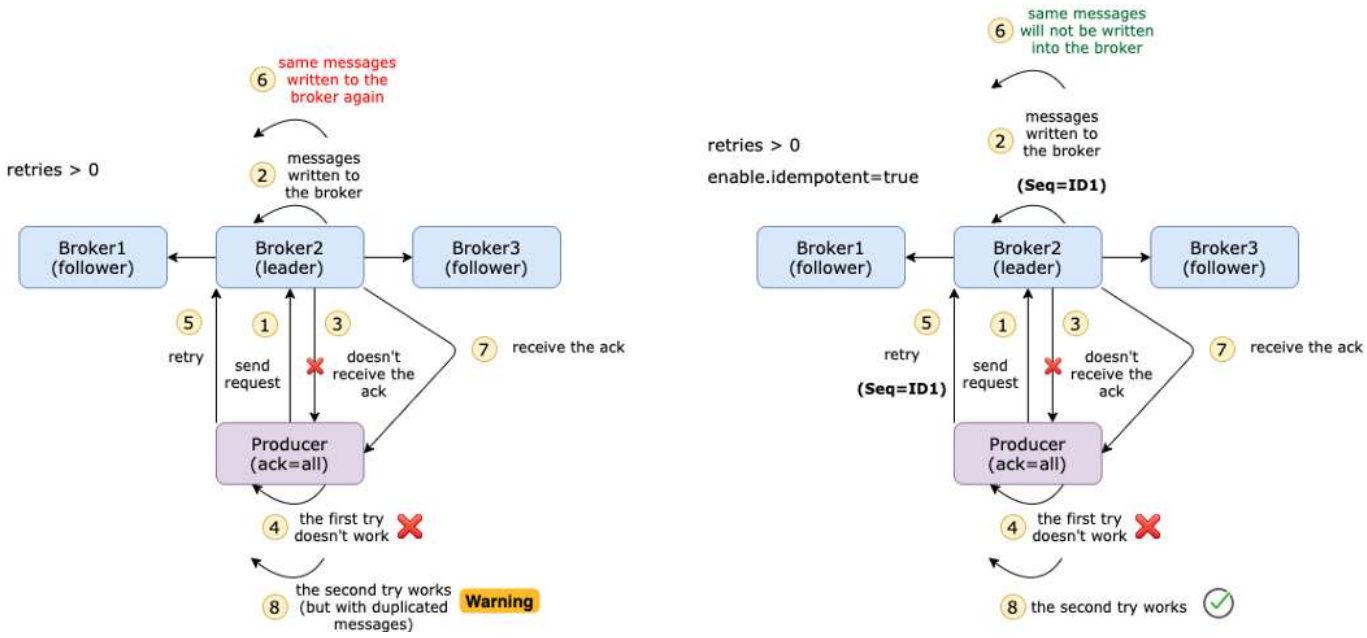
In some situations where a message has actually been committed to all in-sync replicas, but the broker couldn't send an ack back due to a network issue (e.g. only allows one-way communication). Meanwhile, we set `retries=3`, then the producer will resend the messages 3 times. This could lead to duplicated messages in the topic.

Let's say we have a producer that sends 1M messages to the topic and the broker fails after the messages have been committed but before the producer receives all the acks. In this case, we will probably end up with more than 1M messages on the topic. This is also called **at-lease-once semantics**.

The most ideal situation is **exactly-once semantics** where even if the producer resends the message, the consumer should receive the same message only once.

What we need is an **Idempotent Producer.** Idempotent means that applying an operation once or applying it multiple times have the same effect. It's very easy to turn this feature on with config `enable.idempotent=true`.

How does it work? Messages are sent in batches, and each batch has a sequence number. On the broker side, it keeps track of the largest sequence number for each partition. If a batch with a smaller or equal sequence number comes in, the broker will not write that batch into the topic. In this way, it also ensures the order of the batches.



Difference between disabling idempotent and enabling idempotent Created by Xiaoxu Gao

Send messages in order

Another important config to ensure the order is `max.in.flight.requests.per.connection`, and the default value is 5. This represents the number of unacknowledged requests that can be buffered on the producer side. If the `retries` is greater than 1 and the first request fails, but the second request succeeds, then the first request will be resent and messages will be in the wrong order.

According to the documentation:

> *Note that if this setting is set to be greater than 1 and there are failed sends, there is a risk of message re-ordering due to retries (i.e., if retries are enabled).*

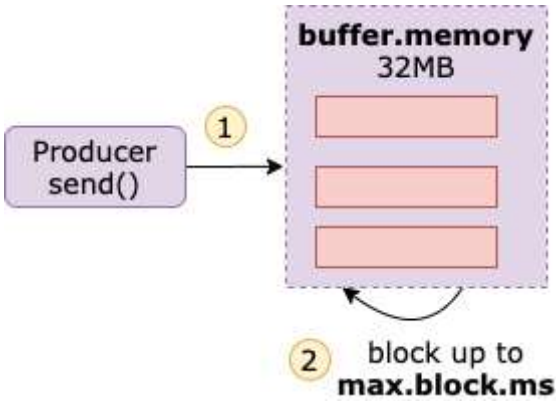
If you don't enable idempotent, but still want to keep messages in order, then you should config this setting to 1.

But if you've already enabled idempotent, then you don't need to explicitly define this config. Kafka will choose suitable values, as stated here.

> *If these values are not explicitly set by the user, suitable values will be chosen. If incompatible values are set, a* `ConfigException` *will be thrown.*

Sending messages too fast

When the producer calls `send()`, the messages will not be immediately sent but added to an internal buffer. The default `buffer.memory` is 32MB. If the producer sends messages faster than they can be transmitted to the broker or there is a network issue, it will exceeds `buffer.memory` then the `send()` call will be blocked up to `max.block.ms` (default 1 minute).



buffer.memory and max.block.ms Created by Xiaoxu Gao

This problem can be mitigated by increasing both values.

Another 2 configs that you can play around with are `linger.ms` and `batch.size`. `linger.ms` is the delay time before the batches are ready to be sent. The default value is 0 which means batches will be immediately sent even if there is only 1 message in the batch. Sometimes, people increase `linger.ms` to reduce the number of requests and

improve throughput. But this will lead to more messages kept in memory. So, make sure that you take care of both sides.

There is an equivalent configuration as `linger.ms`, which is `batch.size`. This is the maximum size of a single batch. Batches will be sent when any of these 2 requirements are fulfilled.

Conclusion

Here are the 10 configurations that I think can make your Kafka producer more resilient. Of course, they are not the only configs you need to take care, but it can be a good starting point. Make sure you read the official Apache Kafka documentation for a reliable reference.

Feel free to leave your comments below if you have any other ideas. Take care!

Reference

What does In-Sync Replicas in Apache Kafka Really Mean? - CloudKarafka, Apache Kafka Message...

Kafka considers that a record is committed when all replicas in the In-Sync Replica set (ISR) have confirmed that they...

www.cloudkarafka.com

Exactly-once Semantics is Possible: Here's How Apache Kafka Does it

I'm thrilled that we have hit an exciting milestone the Apache Kafkaand ® community has long been waiting for: we have...

www.confluent.io