7 mistakes when using Apache Kafka

Michał Matłoka Follow
Jan 22 · 7 min read

Photo by chuttersnap on Unsplash

Apache Kafka is currently very popular. It is being leveraged as a message broker but can be extended by additional tools to become a whole message processing platform. It is often used to communicate microservices in an asynchronous manner but also to process streams of messages. Every tool has its caveats. Small configuration mishaps may lead to a big disaster. In this blog, we'll focus on 7 most common mistakes and how to avoid them.

> *I have written recently a similar blog post about 7 mistakes when using Apache Cassandra. Take a look and learn about best practices!*

Mistake 1 — Let's use the default settings

Sending a message to non-existing Kafka topic, by default results in its creation. Unfortunately the default settings define a single partition and a replication factor of 1. They are totally not acceptable for production usage due to possible data loss and limited scalability. What is more those settings affect various Kafka "special" topics, e.g. intermediate topics used by Kafka Streams or ones leveraged by Kafka Connect in Distributed mode.

How to avoid that?

- First, you may disable the `auto.create.topics.enable` (on broker level), this way every topic will need to be created explicitly, manually.
- Another way is to override `default.replication.factor` and `num.partitions` (on broker level) to change the defaults for auto-created topics.

> *For more configuration settings see the Broker Configs documentation section.*

Mistake 2 — Let’s set 1 partition for now and change that later

What is the right partitions number? That is quite a difficult question everybody asks when they need to define configs for new topics. Unfortunately, there is no single simple answer, only recommendations you need to follow.

First, let’s define the lower bound. The number of partitions defines the maximum number of consumers from a single consumer group. If we define 2 partitions, then 2 consumers from the same group can consume those messages. What is more, if we define too small number, then the partitions may not get located on all possible brokers leading to nonuniform cluster utilization. Basically this means that the maximal consuming speed and producing speed are influenced by lower bound of range of possible partition numbers.

What about higher bound? When do we know that there are too many partitions? Well, big partitions number influences various processes. Publishers buffer batches of messages sent to Kafka per partition. Bigger number of partitions = lower probability that messages with different keys will land on the same partition, meaning lower probability of getting larger batches. Additionally more partitions means more separate buffers = more memory. This means that too big partition number can influence producer throughput and memory usage.

> *Kafka 2.4.0 introduces concept of Sticky Partitioner (KIP-480). Default partitioner, for messages without an explicit key is using Round Robin algorithm. In case of Sticky Partitioner producer aims to fill the batch for a single partition and once it’s full it starts putting messages into another batch for a next partition.*

On broker side more partitions mean bigger number of opened file handles. This isn’t however too worrying. On the other hand, every partition has a leader, which is chosen by Kafka Controller. If simultaneously multiple new leaders need to be chosen (because of e.g. broker failure), then Controller may produce the “decisions" with bigger latency and the partitions may be unavailable for longer time. For example, leader election process for 1000 partitions takes around 5 seconds.

What if you've decided to set some number "temporary" and later resize number of partitions? Yes, that is possible. However, the default partitioner for the messages with the key defined, calculates the hash from the key modulo number of partitions. When the number of partitions changes, then messages with the same key may start being placed to another partition than before. You need to remember about this, because depending on the business use cases such change could lead to messages order change (for given key) around the topic resize time.

So what is the best number? Actually, you should do some performance tests. Simulate the load (let's say predicted for a few years ahead), attach the consumers and test what number of partitions will allow achieving the needed performance. Remember to use the same hardware specs as those which will be available on the production.

> *If you'd like to read more about choosing the right number of partitions, then take a look at Jun Rao blogpost at the Confluent blog.*

Mistake 3 —Let's use the Publisher default configs

In order to make Kafka Producer working it is needed to define actually only 3 configuration keys — bootstrap servers, key and value serializers. However, often it is not enough. Kafka includes a lot of settings that may influence the messaging ordering, performance or probability of data loss. For example:

- `acks` — defines how many of in-sync replicas need to acknowledge the message (by default only the partition leader, what may result in data loss, because leader does not wait for data to be written to a disk, only to filesystem cache)
- `retries` — defines the number of retries during a failed send
- `max.in.flight.requests` — defines max number of not acknowledged requests the client may be processing. In practice may influence the message ordering.

What is more, underneath Producer batches all messages it was ordered to send. It creates buffers per partition. This also means that the message was actually sent when the send callback gets called or appropriate `Future` finishes.

Mistake 4 —Let’s use basic Java Consumer

Kafka Java client is quite powerful, however, does not present the best API. Using the Java Consumer is quite painful. First, the `KafkaConsumer` class can be used only by a single thread. Then, it is required to define an "infinite" while loop, which will `poll` broker for messages. However, the most important is how the timeouts and heartbeats work.

Heartbeats are handled by an additional thread, which periodically sends a message to the broker, to show that it is working. Kafka requires one more thing. `max.poll.interval.ms` (default 5 minutes) defines the maximum time between poll invocations. If it's not met, then the consumer will leave the consumer group. This is ultra important! Let's say that you consume messages and send them to some external HTTP API. In case of failure, you may leverage exponential backoff. If it goes over 5 minutes, then… your consumer will leave the group and message will get delivered to another instance. Scary and difficult to maintain. The simplest way to avoid such scenario is to limit number of records fetched in a single `poll` invocation.

What are other choices instead of the plain Java client? There are a few:

- Spring for Apache Kafka
- Alpakka Kafka — which offers both Java and Scala APIs. Did you know that it originates at our company — SoftwareMill?
- FS2 Kafka — Scala lib for Kafka integration with FS2

- Micronaut Kafka
- Quarkus Kafka
- and others

Most of the libraries automatically manage the requirement for the `poll` intervals, by explicit consumption pausing. Before choosing your lib, check if it is for sure handled correctly.

Mistake 5 — Our business requires, of course, the exactly-once semantics

Usually, when you ask your client what deliver/processing semantics are acceptable for the business you'll hear the answer: exactly once. Then, you do some Googling and you see wonderful headlines saying that Kafka supports exactly once! That's partially true, from a theoretical perspective there is no such thing as exactly-once delivery… because, usually it's an at least once with de-duplication, what effectively is exactly-once.

In Kafka's case, there are limitations around the exactly-once. You need to enable special settings on Producer side (e.g. `enable.idempotence` which requires specific values for `max.in.flight.requests.per.connection` , `retries` and `acks`). On Consumer side it is more difficult. Due to failures you still can process messages more than once, then the solution is to deduplicate them, e.g. doing database upserts.

Another solution is to use Kafka Streams, which explicitly defines a setting “exactly once”, but in practice this means that you get the exactly once only by going from Kafka to Kafka.

> *You can find more details about exactly-once in Adam Warski blogpost: What does Kafka’s exactly once processing really mean?*

Mistake 6 — Who cares about monitoring when the system works?

Kafka Cluster is a distributed system. In such architectures a lot of things can actually go wrong. Monitoring is a necessity to know if everything works as it should. It is important to observe systems and define alerts.

I have worked once for a customer, who actually had a pretty nice dashboard with Kafka metrics. However, it was just red, because nobody fully understood the values which were there. One of the found issues was a problem with under replicated partition, where hundreds of them were just not in sync. In the worst case scenario this could lead to service unavailability or even data loss.

If you really don't want to lose data or loose the availability you should actually leverage the Kafka metrics. There are various tools which allow to export them quite easily (take a look at JMX or Kafka Exporters). Metrics may allow you to prevent the incoming disaster.

Mistake 7 —Let's upgrade project dependencies without looking at release notes

Since Kafka 0.11 clients are generally forward and backward compatible with Kafka brokers. When new versions are released the upgrade process is quite simple. However, a single version bump may lead to great problems. As an example let's take a look at the 2.1.0 release. That's the version that introduced the KIP-91 Intuitive Producer Timeout influencing the default value of the Producer `retry` parameter. It was changed from `0` to `Integer.MAX_VALUE` . Additionally, new producer timeout was added. This change hasn't effectively introduced the infinite retries, it just enabled the retries by default, what with addition of `max.in.flight.request.per.connection` >1 (default 5) can lead to message reorder!

> *More on this change you can read in Kamil Charłampowicz blogpost — Does Kafka really guarantee the order of messages?*

Conclusions

Kafka is very powerful, however, you need to use it wisely. Avoiding simple mistakes allows to avoid unexpected problems in the future. Check carefully what settings are used for Brokers, Topics, Producers and Consumers. After deployment observe and define alerts around most important Kafka and business metrics. This way you can make sure that the system works as you intended.

. . .

```
Like this post and interested in learning more?

Follow us on Medium!

Need help with your Cassandra, Kafka or Scala projects?

Just contact us here.
```

Thanks to Jarosław Kijanowski.

Big Data   Lessons Learned   Microservices   Kafka   Apache Kafka