Published in Better Programming

Vladimir Topolev Follow
Oct 12, 2021 · 7 min read ★ · Listen

How to Handle Duplicate Messages and Message Ordering in Kafka

Dealing with the challenges faced when using Apache Kafka

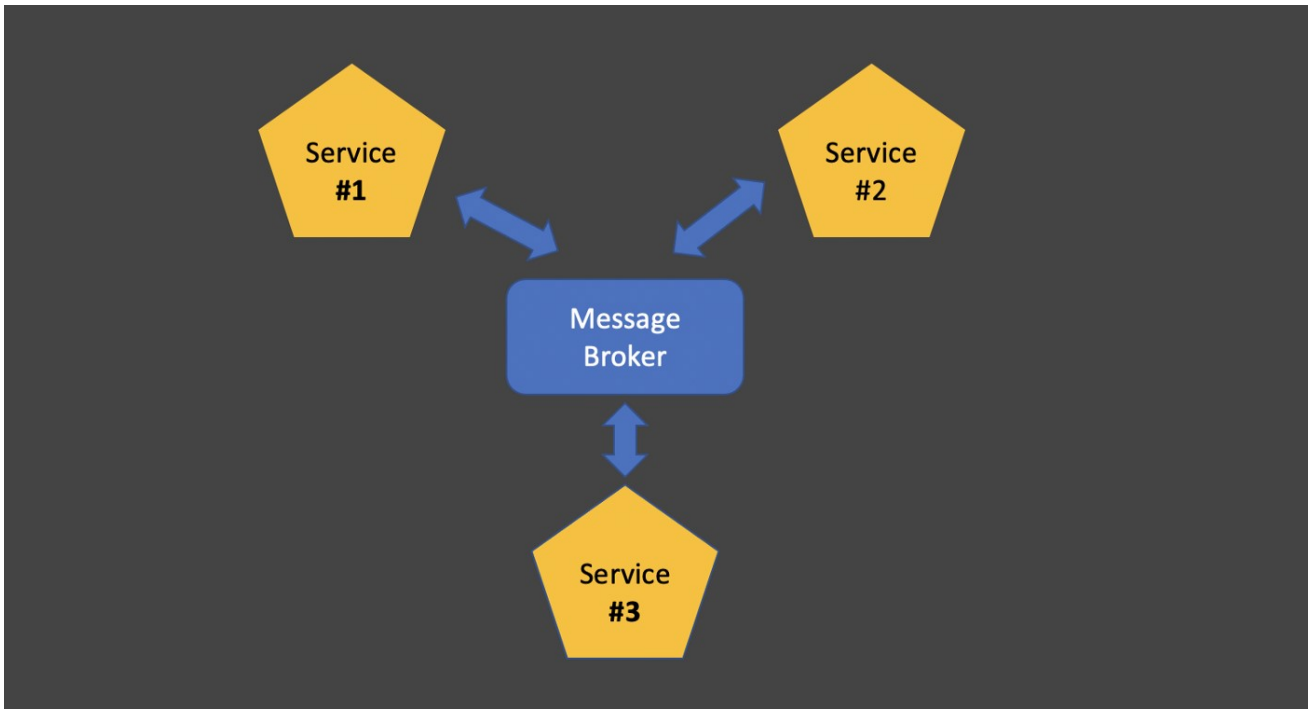

Image by author

Event-driven architecture is one of the ways to implement interprocess communication between some services in a microservice architecture. A message-based application typically uses a message broker which acts as an intermediate layer between the services.

Apache Kafka is one of the tools which allows implementing a broker-based messaging approach. This tool resolves a lot of issues on its own from the box, but as a developer, we do come across challenges that should be taken into account. In the next few sections, we'll consider some of the cand have a look at how to resolve them.

1. Message Ordering

Some applications are pretty sensitive to the order of messages being processed by consumers. Let's imagine that the user creates some `Order`, modifies it, and finally cancels. This means, that the `Producer` sends to Message Broker the following messages with commands: `ORDER_CREATE`, `ORDER_MODIFY`, `ORDER_CANCEL`.

In Kafka, we can create a special topic `Order` which will get all messages (commands) that will process any operation related to the orders. If our application has only one `Consumer` and one topic with one partition, you will never come across an ordering issue — since Kafka sends all messages within a topic `Partition` to `Consumer` in the same order which it gets from the `Producer`:

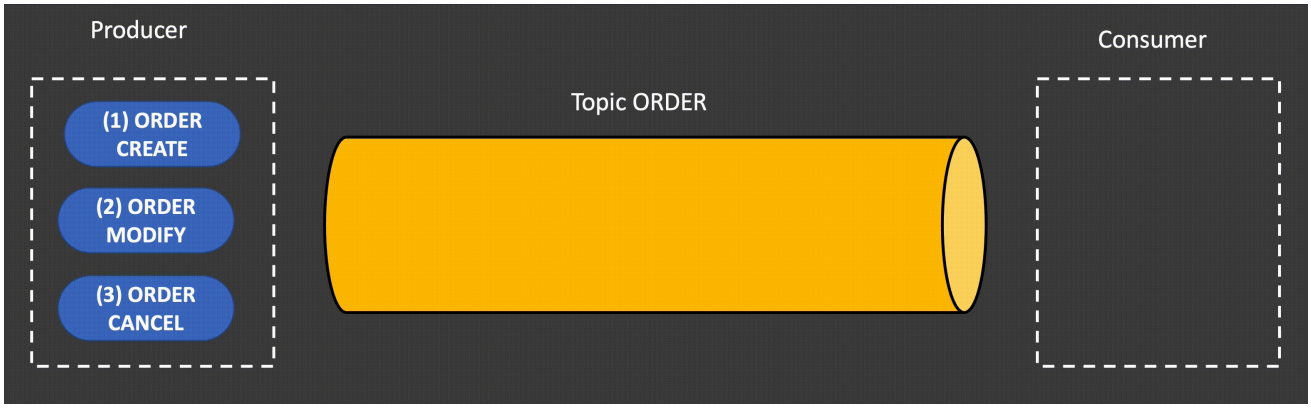

Figure 1 — Message ordering with one consumer and one partition in Kafka Topic

Here, of course, we should pay our attention, since theoretically, all commands related to one `Order` which we mentioned before `ORDER_CREATE`, `ORDER_MODIFY`, `ORDER_CANCEL` might be put in different partitions and will be handled by different `Consumers`. It may lead that for example `ORDER_CANCEL` command getting completed before `ORDER_CREATE` as shown below:

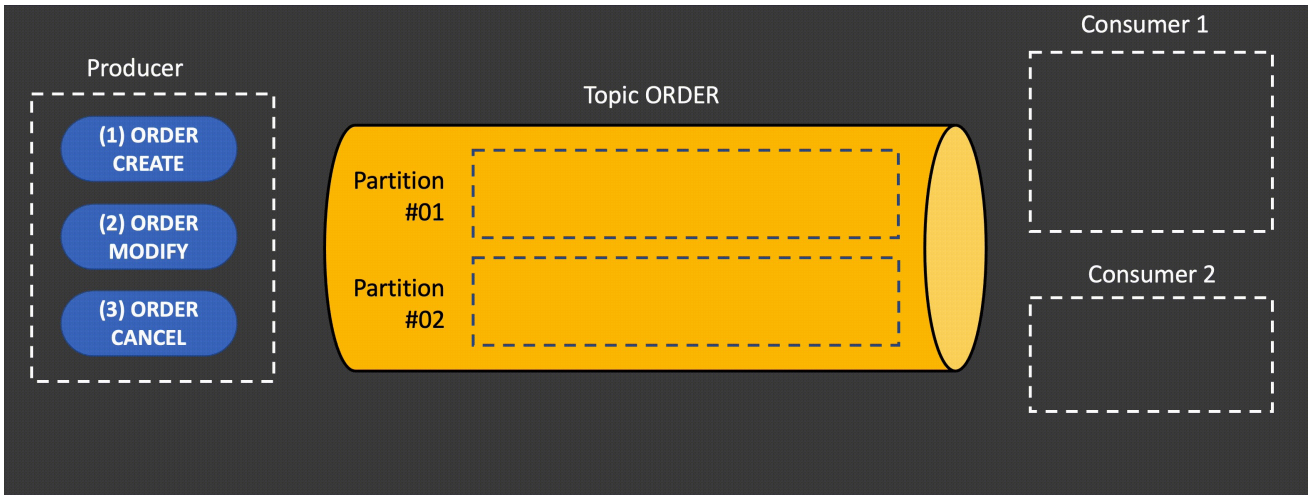

Figure 2 — Unexpected message ordering

But this issue is easy to fix. Here is the main rule:

> *In Kafka, the ordering is guaranteed for all messages in one Partition.*

Therefore, we should only take care that all commands `ORDER_CREATE`, `ORDER_MODIFY`, `ORDER_CANCEL` should be put in one Partition associated with the particular `Order`.

Kafka actually has several ways to define to which partition sends a message. One of the ways is to assign a special key to the message. Kafka distributes them according to key hash: `hashCode(key) % N`; where N — is a count of partitions.

This means if we assign the same key for all commands `ORDER_CREATE`, `ORDER_MODIFY`, and `ORDER_CANCEL`, all of them will be sent to one partition and all of them will be sent to one `Consumer` in the same order. In our case, I believe that the better choice for the key will be `orderId` which is unique and stable and all commands across one order will be guaranty completed in one `Consumer` in an expected order.

Another way to distribute messages across `Partitions` is to explicitly define to which partitions send the message. For example, if you have 2 partitions and each message belongs to a particular client, you may divide all messages that all clients whose surname starts from letter belongs to range A-M should go to the first partition and for clients whose surname starts from N-Z should go to the second partition.

2. Duplicate Messages

There're different strategies for how messages should be delivered to a `Consumer`.

Ideally, the message broker should send each message only once, but different failures may occur in Consumers during processing those messages. For example, `Consumer` gets a message but shut down before making all necessary operations against DB, it leads that we leave some messages unhandled.

Instead, most message brokers promise to deliver a message at least once. It means that if some failure happens during message processing, the message broker may deliver the same message several times.

How does it work in Kafka? Kafka under the hood has a special cursor (offset). All messages behind offset are considered as handled and not delivered anymore to the particular consumer group.

Kafka has a special configuration that defines how the offset should be changed. If you set `enable.auto.commit` to `true`, it means that Kafka shifts offset as soon as it sends batched messages to `Consumer` — and doesn't take care of whether Consumers handled messages or not. In this case, Kafka guarantees that each message sends only once. But it's not what we're expecting to develop.

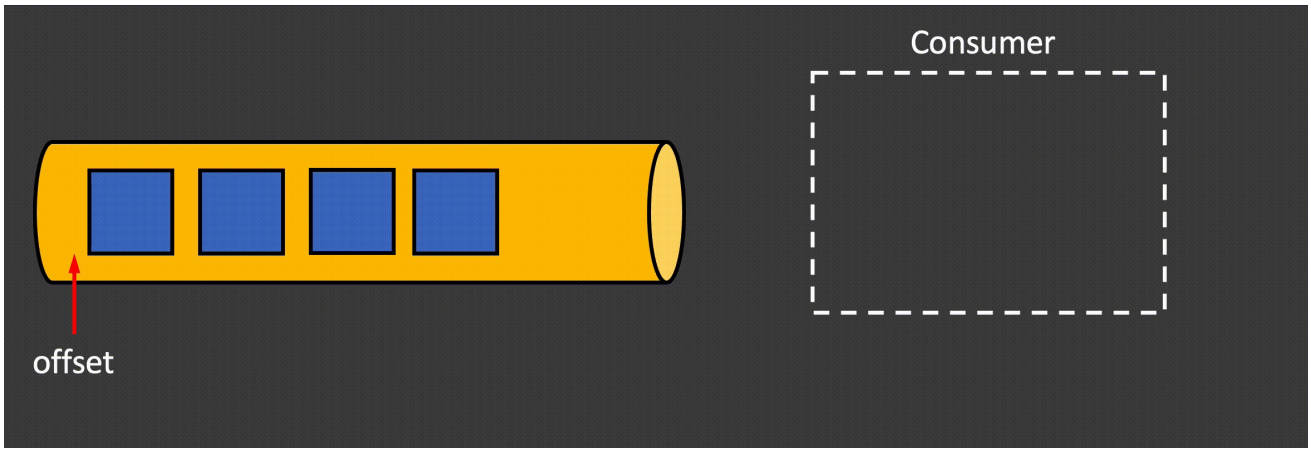

Figure 3 — Kafka may be set up with auto-commit, as soon as it sends messages it automatically move offset. If a

If we set `enable.auto.commit` into false, Kafka after sending a batch of messages to Consumer will shift offset only after getting explicit acknowledge message from Consumer that all messages in the batch have been handled.

Let's assume, that Kafka sends a batch of the following messages to Consumer: `ORDER_CREATE`, `ORDER_MODIFY`, `ORDER_CANCEL`. Consumer completed only one command `ORDER_CREATE` and crushes down and doesn't send acknowledge message to Kafka to shift offset. As soon as `Consumer` spins up again, Kafka starts to send all 3 messages again. It means that a `Consumer` may get one message twice and there may be an issue as shown below.

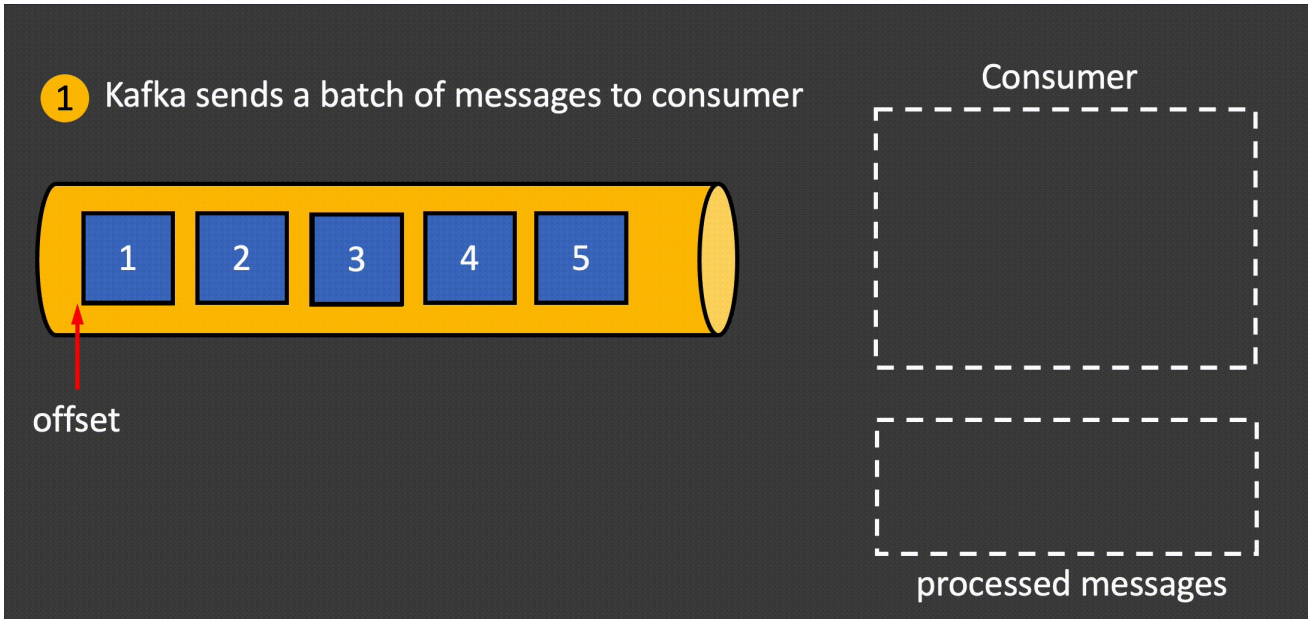

Figure 4 — Consumer explicitly requests Kafka to move offset as soon as all messages are processed. But it may handle some messages in batch and crash before sending acknowledge message to move offset, in this case, Kafka will resend all messages again and the consumer may handle one message twice

Therefore, we should keep in mind during the development that a `Consumer` may accept multiple times the same message. There're a couple of ways to handle duplicate messages:

- write idempotent message handler
- track all received messages and discard duplicates

2.1 Write idempotent message handler

It's the easiest way to have a deal with duplicate messages. The message handler is idempotent if calling it multiple times with the same payload has no additional effect. For example, modify an already modified `Order` with the same payload should give the same result. It's like a `PUT` operation in RESTful API, which we should develop in an idempotent manner.

Unfortunately, it's not applicable for all cases. Let's assume, besides `ORDER_CREATE`, `ORDER_MODIFY`, and `ORDER_CANCEL`, we have a command `ORDER_PAYMENT`. In this message handler, we should reduce a customer credit balance and this operation couldn't be completed in an idempotent manner by its nature. We need to come up with another way which is considered below.

2.2 Track all received messages and discard duplicated

As we already highlighted, that some of the commands couldn't be completed in an idempotent manner. An example maybe `ORDER_PAYMENT`. We should avoid any attempt to reduce the customer credit balance twice or more.

If our application is using Relational DB with transaction support, it's easy to implement to introduce a new table, for example, `PROCESSED_MESSAGE` where we would store all message ids which is already processed. Therefore any DB update will be wrap in one transaction with an insert record in the `PROCESSED_MESSAGE` table as shown below:

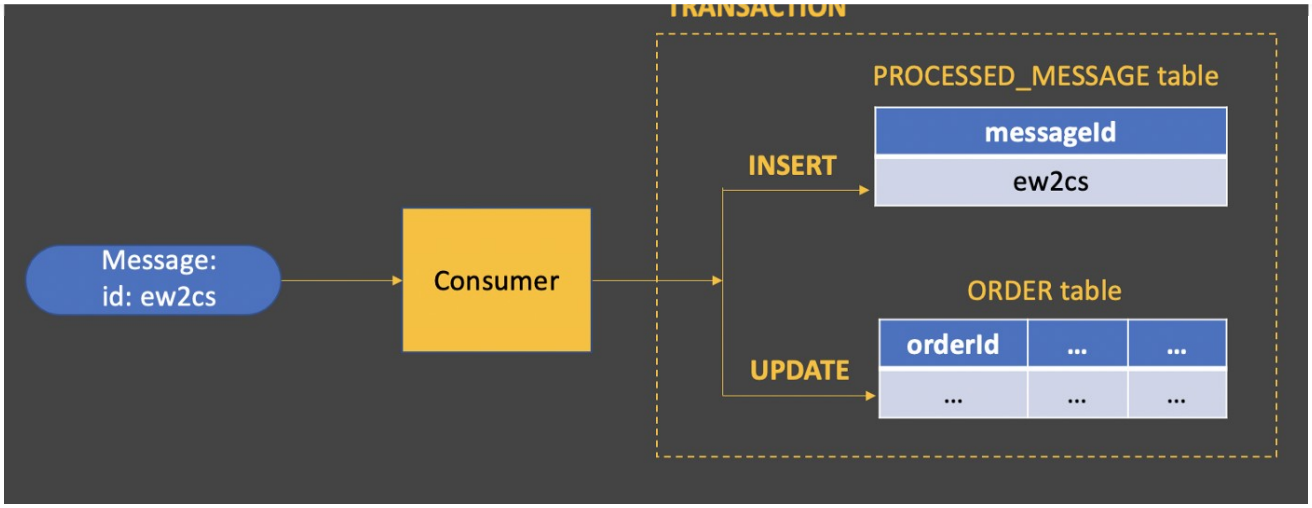

Figure 5— Tracking all received messaged from Kafka and discard duplicate ones

In this example, a Consumer inserts a row containing `messageId` into the `PROCESSED_MESSAGE` table. If a message is a duplicate, the `INSERT` operation will fail and Consumer discards this message without updating the `Order` entity.

Conclusion

1. Kafka guarantees the message ordering for all messages within one partition.
2. The developer should take into account how to distribute messages between Topic partitions if the message ordering matters.
3. Kafka may distribute messages by a key associated with each message, if a key is the same for some messages, all of them will be put in the same partition.
4. Message brokers may offer some strategies to deliver messages: “only once” or “at least once”.
5. Delivering with “only once” strategy leads that some messages may be left unhandled
6. Delivering with an “at least once” strategy supposes that the developer should have a deal with the duplicate message.
7. Duplicate messages may be handled in several ways: implement an idempotent message handler or tracking all received messages and discard duplicate ones.

Related sources:

Kafka Overview with pictures
Right now event-driven architecture is getting more popular in modern development. The one reason is that it allows to...
medium.com

Saga Pattern with Kafka and NodeJS: simple implementation
Whenever you’ve decided to rebuild your application from monolithic to microservices architecture you come across with...
medium.com

570 6 •••
Using Saga Pattern in Microservices
How to Prevent Anomalies in Distributed Transactions with Saga Pattern
enlear.academy