**Exactly-Once Semantics With Apache Kafka**

Kafka's exactly once semantics was recently introduced with the version which enabled the message being delivered exactly once to the end consumer even if the producer retries to send the messages.

This major release raised many eyebrows in the community as people believed that this was not mathematically possible in distributed systems. Jay Kreps, co-founder of Confluent and co-creator of Apache Kafka, explained its possibility and how is it achieved in Kafka in this [post](https://medium.com/@jaykreps/exactly-once-support-in-apache-kafka-55e1fdd0a35f).

In this blog, we will be discussing how can one take advantage of the exactly once message semantics provided by Kafka.

## **Overview of Different Message Delivery Semantics Provided by Apache Kafka**

**"At most once-messages may be lost but are never redelivered."**

In this case, the producer does not retry to send the message when an ACK times out or returns an error, thus the message might end up not being written to the Kafka topic, and hence not delivered to the consumer.

**"At least once-messages are never lost but may be redelivered."**

In this case, the producer tried to resend the message if the ACK times out or receives an error, assuming that the message was not written to the Kafka topic.

**" Exactly once —  this is what people actually want, each message is delivered once and only once."**

In this case, even if a producer tries to resend a message, it leads to the message being delivered exactly once to the end consumer.

Exactly-once semantics are the most desirable guarantee and require cooperation between the messaging system itself and the application producing and consuming the messages.

For instance, if, after consuming a message successfully, you rewind your Kafka consumer to a previous offset, you will receive all the messages from that offset to the latest one, all over again. This shows why the messaging system and the client application must cooperate to make exactly-once semantics happen.

## **Why Use the Exactly-Once Semantics of Kafka?**

We know that at-least-once guarantees that every message will be persisted at least once, without any data loss, but this may cause duplicates in the stream.

For example, if the broker failed right before it sent the ACK, but after the message was successfully written to the Kafka topic, this retry will lead to the message being written twice and hence delivered more than once to the end consumer.

###### **https://i2.wp.com/blog.knoldus.com/wp-content/uploads/2018/10/kafka-eos1.png?resize=681%2C453&ssl=1**

In the new exactly-once semantics, Kafka's processing semantics guarantee delivery of the message to the end consumer exactly once. This has been strengthened by introducing:

* Idemptotent producers
* Atomic transactions

## **Idempotent Producer**

An **idempotent** operation is an operation that can be performed many times without causing a different effect than if the operation was only performed once.

Now, in Kafka, the producer sends operations that can be made idempotent, so that if an error occurs which causes a producer retry, the same message which is sent by the producer multiple times will only be written once to the logs on the maintained Kafka broker.

Idempotent producers ensure that messages are delivered exactly once to a particular topic partition during the lifetime of a single producer.

To turn on this feature and get exactly-once semantics per partition — meaning no duplicates, no data loss, and in-order semantics — configure your producer with the following property:

enable.idempotence=true

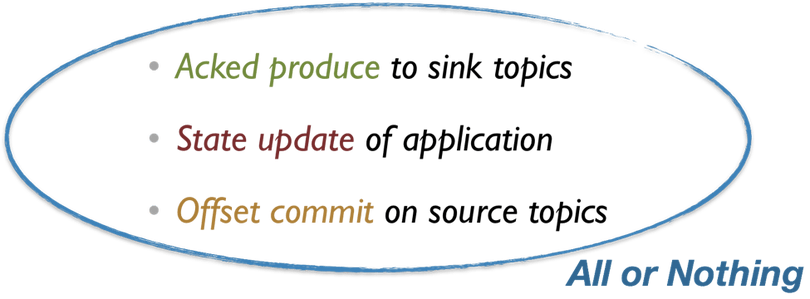
With this feature turned on, each producer gets a unique id (PID), and each message is sent together with a sequence number. When either the broker or the connection fails, and the producer tried to resend the message, it will only be accepted if the sequence number of that message is one more than the one last message.

However, if the producer fails and restarts, it will get a new PID. Hence, the idempotency is guaranteed for only a **single producer session**.

## **Atomic Transactions**

Kafka now supports atomic writes across multiple partitions through the new transactions API. This allows a producer to send a batch of messages to multiple partitions such that either all the messages in the batch are visible to all the consumers or none are ever visible to any consumer.

It allows you to commit your consumer offsets in the same transaction along with the data you have processed, thereby allowing end-to-end exactly-once semantics.



Below is an example snippet that describes how can you send messages atomically to a set of topic partitions using the new Producer API:

{

producer.initTransactions();

try{

producer.beginTransaction();

producer.send(record0);

producer.send(record1);

producer.sendOffsetsToTxn(…);

producer.commitTransaction();

} catch( ProducerFencedException e) {

producer.close();

} catch( KafkaException e ) {

producer.abortTransaction();

}

}

## **Consumers**

To use transactions, you need to configure the Consumer to use the right **isolation.level**and use the new Producer APIs. There are now two new isolation levels in Kafka consumer:

1. **read\_committed**: Read both kinds of messages (those that are not part of a transaction and that are) after the transaction is committed.
2. **read\_uncommitted**: Read all messages in offset order without waiting for transactions to be committed. This option is similar to the current semantics of a Kafka consumer.

Also, the **transactional.id**property must be set to a unique ID in the producer config. This unique ID is needed to provide continuity of transactional state across application restarts.

## References

* [Confluent’s blog on exactly once semantics](https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/)
* [Transactions in Apache Kafka](https://www.confluent.io/blog/transactions-apache-kafka/)
* [Image source](https://hevodata.com/blog/wp-content/uploads/2017/09/IDEM.png) for comparison between favorable and gone cases of at least once semantics

# What does Kafka's exactly-once processing really mean?

Kafka’s [0.11 release](https://kafka.apache.org/downloads) brings a new major feature: exactly-once semantics. If you haven’t heard about it yet, Neha Narkhede, co-creator of Kafka, [wrote a post](https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/) which introduces the new features, and gives some background.

This announcement caused a stir in the community, with some claiming that exactly-once is not mathematically possible. Jay Kreps wrote a [follow-up post](https://medium.com/@jaykreps/exactly-once-support-in-apache-kafka-55e1fdd0a35f) with more technical details. Plus, if you’re really curious, there’s also a [detailed design document](https://docs.google.com/document/d/11Jqy_GjUGtdXJK94XGsEIK7CP1SnQGdp2eF0wSw9ra8/edit) available.

However, as there’s still some confusion as to what *exactly-once* means in Kafka’s context, I’d like to analyse how you can construct an exactly-once pipeline in Kafka, with an emphasis on where the new features come into play, what kind of guarantees you get, and more importantly, what guarantees you **don’t get**.

Some of the discussions focused on whether Kafka guarantees exactly-once *processing* or *delivery*. I’m not sure if there are precise definitions of either; but, to avoid ambiguity, I would say that Kafka provides an **observably exactly-once** guarantee, if we take into account only Kafka-related side-effects.

Using the features of 0.11, it is possible to create a pipeline where, at each stage, the **result of processing** of each message will be observed exactly-once, as far as Kafka is concerned. This includes the producer (through which the data enters the Kafka pipeline), through possibly many intermediate Kafka-streams-based steps, to the consumer (where the data leaves the Kafka pipeline).

The features which make the above possible are:

* idempotent producers (introduced in 0.11)
* transactions across partitions (introduced in 0.11)
* Kafka-based offset storage (introduced in 0.8.1.1)

Let’s see which of these features are useful at which stage of an exactly-once processing pipeline.

Producer

On the **producer** side, the crucial feature is **idempotency**. To prevent a message from being processed multiple times, we first need to make sure that it is persisted to the Kafka topic only once. With idempotency turned on, each producer gets a unique id (the PID), and each message is sent together with a sequence number. When either the broker or the connection fails, and the producer retries the message send, it will only be accepted if the sequence number of that message is 1 more than the one last seen.

Note, however, that if the producer fails and restarts, it will get a new Pid(or the same one, but with a new epoch number, when a TransactionalIdis specified in the config). Hence, the idempotency guarantees only span a **single producer session**. We might still get duplicates, depending on where the producer gets the data from. If it’s e.g. an HTTP endpoint accessed by a mobile client, in case of failure the mobile client will retry sending, and Kafka won’t prevent the duplicate from being persisted. Or, if we are transferring data from another system to Kafka, we might get duplicates, depending on how we determine the “starting point” from which to read the data from the source system.

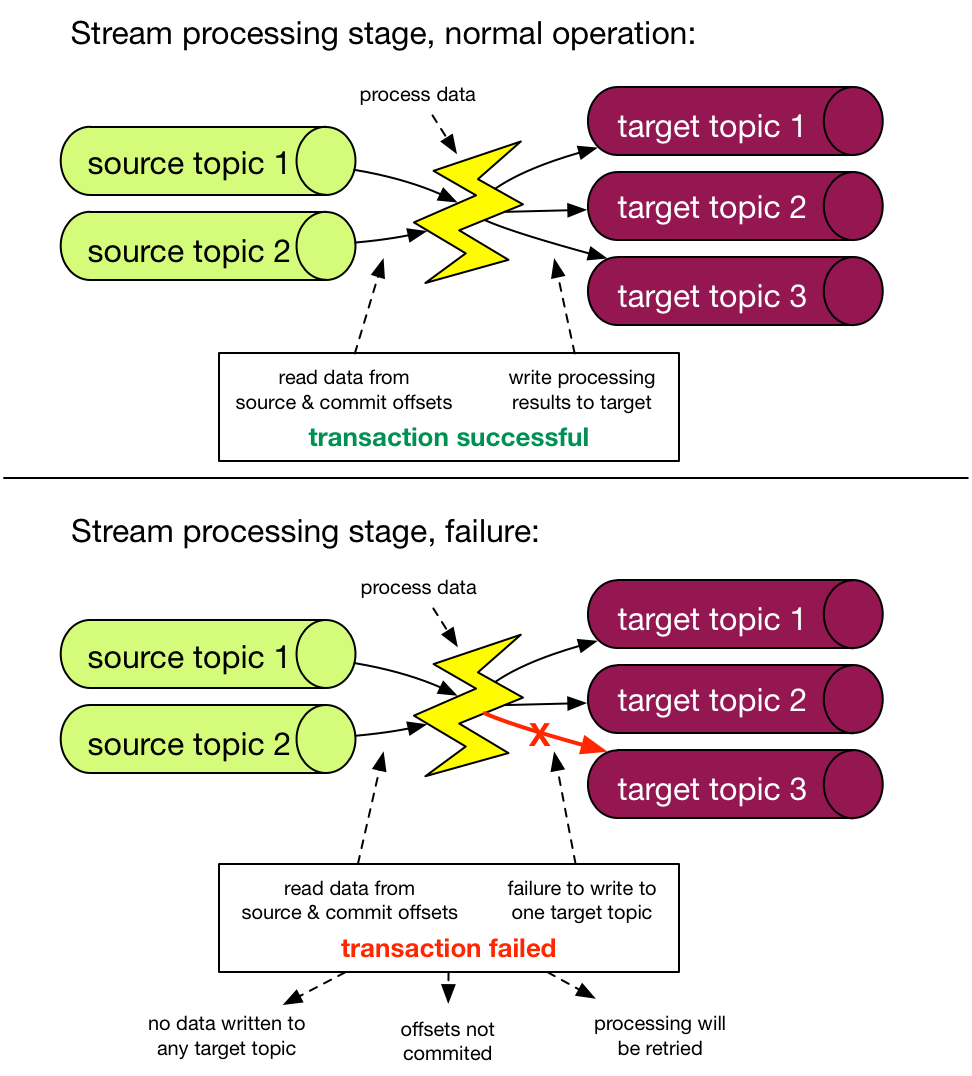
Hence, in some cases, we might need an additional deduplication component. In others, for example when transferring data from another storage system, [Kafka Connect](https://www.confluent.io/product/connectors/) might be worth looking at: it provides a lot of connectors out-of-the-box.

Pipeline stages

Now that we have the data in Kafka, what about processing it? There’s a lot that we can do with data without leaving Kafka, thanks to [Kafka Streams](http://docs.confluent.io/current/streams/index.html). Apart from simple mapping & filtering, we can also aggregate, compute [queryable](http://docs.confluent.io/current/streams/developer-guide.html" \l "interactive-queries) projections, window the data based on event or processing time, and so on. In the process, the data goes through multiple Kafka topics, and multiple processing stages.

So, how to make sure that in each stage, we observe each message as being processed exactly once?

Here the new **transactions** feature comes in. Using it, it’s possible to atomically write data to multiple topics and partitions along with offsets of consumed messages. If we take a closer look at what a single processing step does, it reads data from one or more source topics, performs a computation, and writes the data back to one or more target topics. And we can capture this as an atomic Kafka transaction unit: writing to target topics, and storing the offsets in source topics.



When the exactly-once processing guarantee configuration is set on a Kafka streams application, it will use the transactions transparently behind the scenes; there are no changes in how you use the API to create a data processing pipeline.

We all know that transactions are hard, especially distributed ones. So, how come they work in a distributed system such as Kafka? The key insight here is that we are working within a **closed system** - that is the transaction spans only Kafka topics/partitions.

Consumer

Finally, we will probably need to get the data out of Kafka. How to make sure this is done exactly-once? Here it’s possible provided that the consumer is transactional, i.e. if we can store the result of processing of a given message, along with its offset, together as an atomic unit in the target system. Again, [Kafka Connect](https://www.confluent.io/product/connectors/) might be useful here.

Alternatively, this will also work if the sink is idempotent. In fact, if our processing stages are idempotent, we don’t really need any of the additional exactly-once features: at-least-once is good enough.

Side-effects

If a failure occurs at any of the above described steps, a message might be processed many times - here the at-least-once guarantee is preserved. Because of that, if any of the stages or the consumer has side-effects, they might be executed multiple times. For example, if you have a simple println in your consumer, or streams stage, you might see some messages processed twice. The same applies to sending e-mails, or calling any kind of http endpoints.

However, the messages will only be processed multiple times **internally**. If there are no extra side-effects, the **observable** effect - which in the case of Kafka Streams is what gets written to the target topics of each stage - will be as if each message was processed exactly once.

Summary

If we take the meaning of exactly-once delivery/processing literally, Kafka gives neither: messages might be *delivered* to each processing stage/consumer multiple times, as well as *processed* by a stream’s stage multiple (at-least-once) times. But when using idempotent sends and transactions, we can make sure that **observably** we achieve exactly-once: the result of processing each message will end up in the target stream only once. All that with a single configuration change.