Playing with Apache Kafka

Kafka Setup

In this article I'm going to use Kafka with Zookeeper instead of KRaft. This is because Zookeeper has been for around a long time and lots of company has been using Zookeeper instead of upgrading it to KRaft.

Zookeeper responsibility

In older version of Kafka, you cannot use Kafka without first installing Zookeeper but this requirements was removed starting with v2.8.0 version of Kafka, it can be run without Zookeeper however, this is not recommended in production.

Zookeeper's responsibility in this distributed system is to coordinate tasks between different Brokers. Remember that Kafka cluster is made up of one or more Kafka Brokers. The brokers are responsible for handling the client's request, for both producer and consumers. The topics are enclosed within the Kafka Brokers and they are also responsible for replicating the partitions within the topics.

A Kafka Broker can have more than one topic enclosed within them, it doesn't just have to be one topic that they are helping to coordinate the writing and consuming from.

Zookeeper's responsibility

1. Controller election: Every Kafka Cluster has a controller broker that is responsible for managing the partitions and replications, basically admin tasks. The Zookeeper will help to pick out one that does the job
2. Cluster membership: Zookeeper keeps the list of functioning brokers in the cluster
3. Topic configuration: Zookeeper also maintains the list of all topics, the number of partition in each topic, the replica partitions, and leader nodes
4. Quotas: Zookeeper can accesses how much data each client is allowed to read/write
5. Access Control List: Zookeeper also can control who and what kind of permission the client can have on each topics.

Basically Zookeeper is used for metadata management, it doesn't really affect producers and consumer.

Offsets

Current offset

Whenever a consumer polls for messages from Kafka, let's assume we have 100 records in a particular partition that the consumer is polling from. The initial current offset will be 0 for the consumer, after we have made our call and we receive 20 messages. The consumer will move the current offset to 20. when we make our next request it will retrieve messages starting at position 20 and then move the offset again forward after receiving the messages.

This "current offset" is just a simple integer that is used by Kafka to maintain the current position of one consumer. That is it. It is just used to maintain the last record that Kafka sent to this particular consumer, so that the consumer doesn't get spammed with the same message twice.

Committed offset

Committed offset is used to confirmed that a consumer has confirmed about the processing of the record after it has received them. The committed offset is a pointer to the last record that any consumer has successfully processed. This offset is used to avoid resending the same record to a new consumer in the event of partition rebalance (This occurs when ownership of a partition changes from one consumer to another at certain events described in section below).

Taking the example of 100 records again let's say 20 records are already processed, and a brand new consumer joins into the group and it wants to help process the message from the partition as well, where should it start? It should process those records that are processed by the previous owner of the partition.

With committed offset you can do auto committing (DEFAULT), or manual committing.

Auto committing you just set a interval say 5 seconds, and everytime your consumer polls for records it will check whether 5 seconds has been and if it has it will commit the offset and poll for records. This option is convenient but it might result in duplicate processing of messages because if a new consumer joins in the group and rebalancing is trigger and the partition goes to a different consumer, as the new owner of the partition it will not see the auto committed offset and thus reprocess the same records that has been processed by the previous owner of the partition.

To solve the previous issue manual commits is used. There is two types of manual commits, synchronous commits and asynchronous commits.

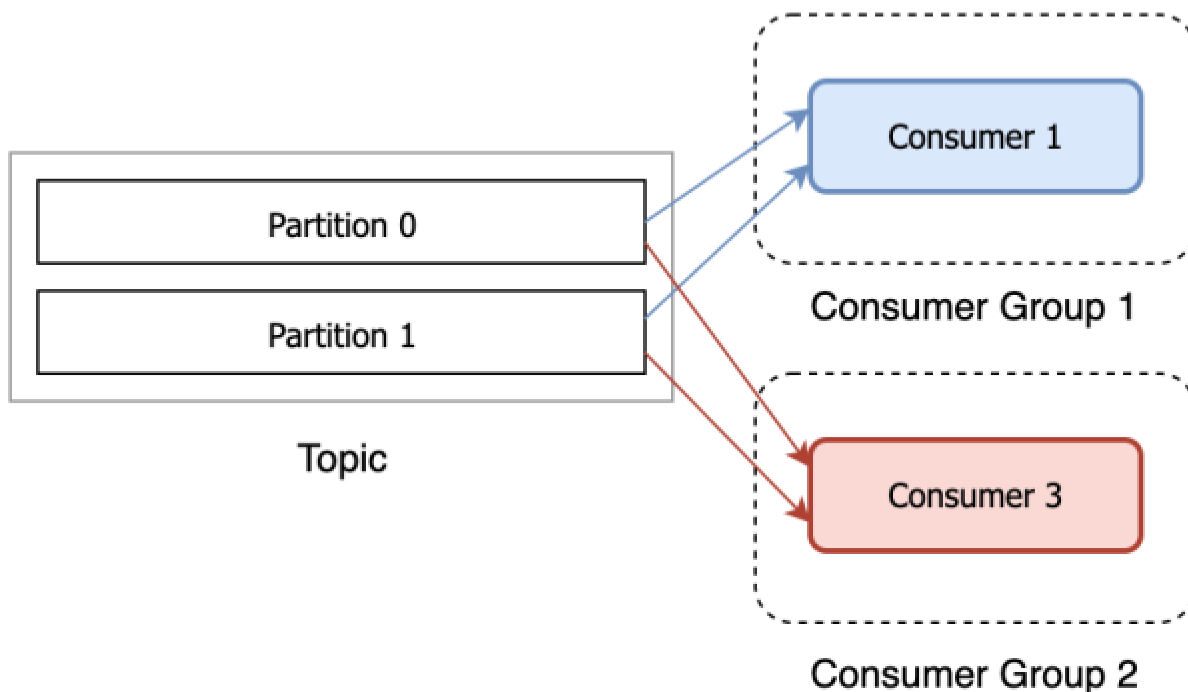
Synchronous commits: It is blocking, it will block until the commits succeeds

Asynchronous commits: It is non-blocking, if it fails it will not be retried. It is purposely designed not to retry otherwise it the ordering problem will occur. If commit to 75 failed but commit to 100 succeed, but 75 is retried and succeed it will cause problem. Therefore, Kafka avoided this problem just by not retrying async commits.

Consumer groups

A consumer group is a group of consumers that shared the same **group id**. Consumers from the consumer group will then be assigned a partition (if available, because again if there are more consumer than the partition, the extra consumers will just sit idling).

This means that if you want to consume the same record (from same partition) from multiple consumers, they MUST be under different consumer groups. Otherwise, the consumer will process the same records.



How does consumers keep track of committed and current offset

Let's start with current offset because this is easy. When a consumer reads messages it will increment the current offset so that IT will not get duplicate messages. This is kept track of by the consumer internally. This is more for the consumer itself, that "oh I got messages and next message I need to get starts at position 20".

Committed offset, this offset is taken care by the group coordinator by producing a message to an internal **__consumer_offsets** topic. The offsets are stored for each `(consumer group, topic, partition)` tuple, **so that you can distinguish different commit offsets for each topic, each partition, and each consumer group.** (Source)

What the hell is `auto.offset.reset`

This property is used to define the behavior of the consumer when there is no committed position.

For example, when the consumer group is first initialized, or when an offset is out of range. They must decide from which point to start to poll the messages.

You can choose to set it to earliest or the latest (default) offset.

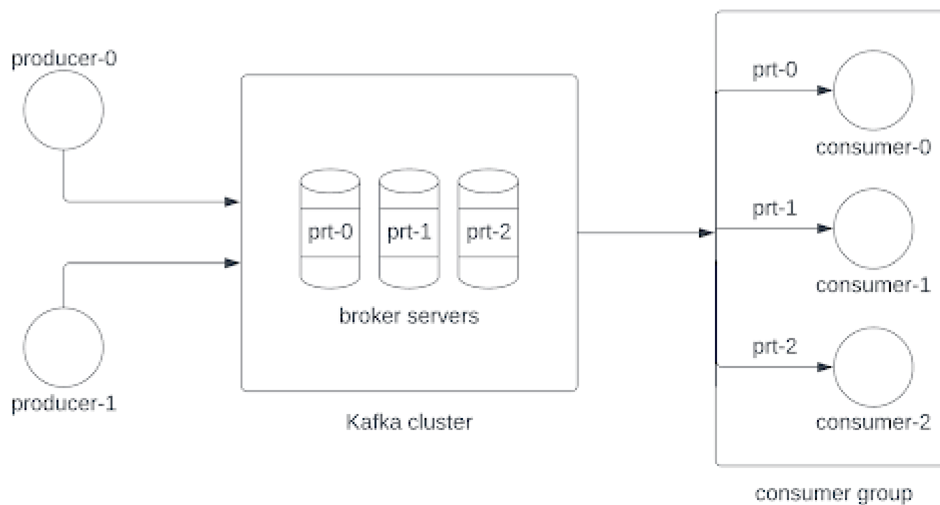
Earliest: Read messages from the beginning of the partition, processing every message present in the partition.

Latest: Only read new messages written to the topic once the consumer has begun listening (so ignoring all the existing messages)

Partition rebalancing

The process of changing partition ownership across a consumer group is called the partition rebalancing.

Let's say for example, each partition has it's consumer in a consumer group laid out like below:

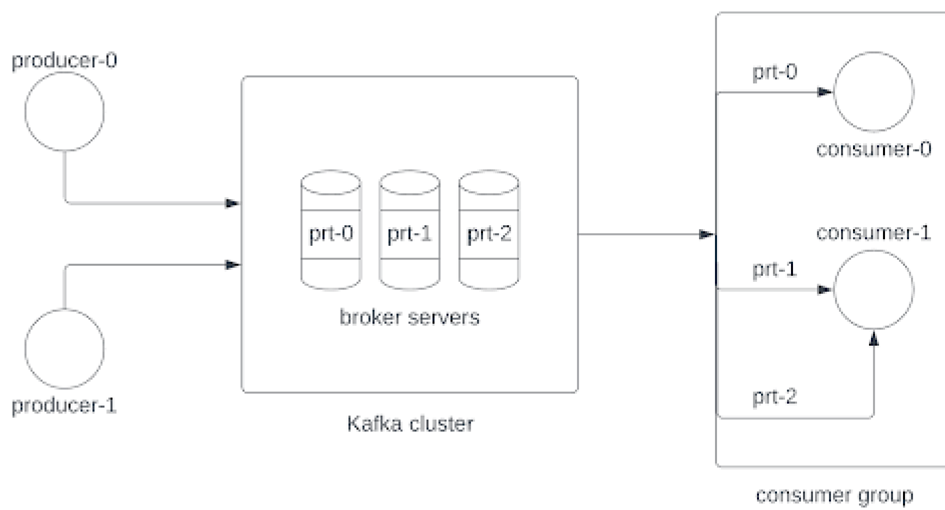


If one of the consumer in the consumer group goes bad due to variety of reasons:

1. Consumer unsubscribed from the topic
2. Consumer doesn't respond to heart beats
3. Consumer hasn't polled the topic for a while
4. Topic is modified (new partition is added)

They will be removed from the consumer group and a partition rebalancing will occur, where the partition will be redistributed with the remaining healthy consumers. Note that this doesn't affect producers because they can still produce to the topic without any issue, only the consumers will be blocked until the partition rebalancing is completed.

The result of the rebalancing will may result in:



> Note that in the case of more partition than consumers, the consumer can be assigned multiple partitions.

Partition rebalancing is needed to make sure the consumer group is high availability and is scalable.

Revision #3
Created 2023-08-06 13:09:29 UTC by Tamarine
Updated 2023-08-06 19:34:29 UTC by Tamarine