Chapter 6. Reliable Data Delivery

Reliable data delivery is one of the attributes of a system which cannot be left as an afterthought. Like performance, it has to be designed into a system from its very first whiteboard diagram. You cannot bolt reliability on after the fact. More so, reliability is a property of a system, not of a single component - so even when we are talking about the reliability guarantees of Apache Kafka, you will need to keep the entire system and its use-cases in mind. When it comes to reliability, the systems that integrate with Kafka are as important as Kafka itself. And because reliability is a system concern, it cannot be the responsibility of just one person. Everyone, Kafka administrators, Linux administrators, network and storage administrators and the application developers must work together to build a reliable system. In such system, Kafka, with its brokers and clients play a critical but not isolated role.

Apache Kafka is very flexible about reliable data delivery. We understand that Kafka has many use-cases, from tracking clicks in a website to credit card payments. Some of the use-cases require utmost reliability while others prioritize speed and simplicity over reliability. Kafka was written to be configurable enough and its client API flexible enough to allow all kinds of reliability tradeoffs.

Because of its flexibility, it is also easy to accidentally shoot yourself in the foot when using Kafka. Believing that your system is reliable while in fact it is not. In this chapter, we will start by talking about different kinds of reliability and what they mean in the context of Apache Kafka. Then we will talk about Kafka's replication mechanism, how it contribute to the reliability of the system. We will then discuss Kafka's brokers and topics and how they should be configured for different use-cases. Then we will move to discuss the clients, the producer and the consumer, and how they should be used in different reliability scenarios. Last, we will discuss the topic of validating the system reliability - because it is not enough to believe a system is reliable, the assumption must be throughly tested.

Reliability Guarantees

When we talk about reliability, we usually talk in terms of *guarantees*. What kind of behaviors a system is guaranteed to preserve under different circumstances.

Probably the best known reliability guarantee is ACID - the standard reliability guarantee that relational databases universally support. ACID stands for Atomicity, Consistency, Isolation and Durability. When a vendor explains that their database is ACID-complaint, it is short for saying that the database guarantees certain behaviors regarding transaction behavior.

/* * Atomicity - a transaction can consist of multiple actions (insert into one table and delete from another, for example) and the database guarantees that either all actions in the transaction completed successfully or none of them completed at all. * Consistency - following a transaction, the database state will be consistent with all the constraints defined on the database. For example, if a column is defined as non-null, it will never contain nulls after a transaction completes. * Isolation - the database state in the middle of a transaction is not visible to other users, only the before or the after state. For example, if I start a transaction and insert a row to a table and then pause, and you run a "select" statement on the table, your select will either show the table without the row (before state) or your select will wait until my transaction is complete and then show the table with all the rows my transaction added. * Durability - once a transaction has been committed, it will remain so. No matter what happens - power loss, crashes, errors, etc. */

Those guarantees are the reason that people trust relational databases with their most critical applications - they know exactly what the system promises and how it will behave in different conditions. They understand the guarantees and can write safe applications by relying on those guarantees.

So, obviously understanding system guarantees is critical to build reliable applications and to be able to reason about system behavior in different conditions. So, what does Apache Kafka guarantee?

- Kafka provides order guarantee of messages in a partition. If message B was written after message A, using the same producer into the same partition, then Kafka guarantees that the offset of message B will be higher than message A, and that consumers will read message B after message A.
- Produced messages are considered “committed” when they were written to the partition on all its in-sync replicas (but not necessarily flushed to disk). Producers can choose to receive acknowledgement on messages sent when the message was fully committed, when it was written to the leader or when it was sent over the network.
- Messages that are committed, will not be lost as long as at least one replica remains alive.
- Consumers can only read messages that are committed.

These basic guarantees can be used while building a reliable system, but in themselves, they don’t make the system fully reliable. There are trade-offs involved in building a reliable system, and Kafka was built to allow the administrators and developers to decide how much reliability they need by providing configuration parameters that allow controlling these trade-offs. The trade-offs usually involve how important it is to reliably and consistently store messages vs other important considerations such as availability, high throughput, low latency and hardware costs. We next review Kafka’s replication mechanism, to introduce the terminology and give some idea of how reliability is built into Kafka. After that will go over the configuration parameters we just mentioned.

/* So far, we discussed the guarantees in theory. Often it is difficult to make the leap from theoretical guarantees into what it means about the system behavior in real-world scenarios. Often we see even experienced Kafka users surprised by how the system behaves in practice. So, lets take a look at few scenarios and see how Kafka guarantees can be used to understand the system behavior in those scenarios. */

Replication

Kafka’s replication mechanism, with its multiple replicas per partition is at the core of all of Kafka’s reliability guarantees. Having a message written in multiple replicas is how Kafka provides durability of messages in events of crashes.

We explain Kafka’s replication mechanism in-depth in Chapter 5, but lets recap the highlights here:

Each Kafka topic is broken down into *partitions*, which are the basic data building blocks. A partition is stored on a single disk, Kafka guarantees order of events within a partition and a partition can be either on-line (available) or offline (unavailable). Each partition can have multiple replicas, one of which is a designated leader. All events are produced to and consumed from the leader replica. Other replicas just need to stay in sync with the leader and replicate all the recent events on time. In an event the leader becomes unavailable, one of the in-sync replicas becomes the new leader.

A replica is considered in-sync if it is the leader for a partition, or if it is a follower that:

- Has an active session with Zookeeper - meaning, it sent a heartbeat to zookeeper in the last 6 seconds (configurable).
- Fetched messages from the leader in the last 10 seconds (configurable)
- Fetched the most recent messages from the leader in the last 10 seconds. That is - it isn’t enough that the follower is still getting messages from the leader, it must have almost no lag.

If a replica breaks any of these assumptions: It loses connection to Zookeeper, it stops fetching new messages or it falls behind and can’t catch up within 10 seconds, the replica is considered out of sync. An out of sync replica gets back into sync when it connects to Zookeeper again and catches up to the most recent message written to the leader. This usually happens quickly after a temporary network glitch is healed, but can take a while if the broker the replica is stored on was down for a longer period of time.

WARNING

Seeing one or more replicas rapidly flip between in-sync and out of sync status is a sure sign that something is wrong with the cluster - often misconfigured garbage collection on a broker causes long pauses during which the broker loses connectivity to Zookeeper and becomes out of sync. Refer to Chapter 2 for some advice on how to resolve that.

It may be counter-intuitive to some, but note that an in-sync replica that is slightly behind can slow down producers and consumers - since they wait for all the in-sync replicas to get the message before it is "committed". While once a replica falls out of sync, we no longer wait for it to get messages. It is still behind but now have no performance impact. But with fewer in-sync replicas the effective replication factor of the partition is lower (since the out-of-sync replica is missing some committed messages), there is greater risk for the system availability and reliability.

In the next section we will look at what this means in practice.

Broker Configuration

There are 3 configuration parameters in the broker that change Kafka's behavior regarding reliable message storage. Like many broker configuration variables, these can apply at the broker level, controlling configuration for all topics in the system and at the topic level - controlling behavior for a specific topic.

Being able to control reliability trade-offs at the topic level means that the same Kafka cluster can be used to host both reliable and non-reliable topics. For example, at a bank, the administrator will probably want to set very reliable defaults for the entire cluster, but then make an exception to the topic that stores customer complaints coming in through twitter - losing few of those may be ok, while losing a credit transaction is not. In that case, the administrator can decide that saving money on hardware costs (there are many tweets) is more important than not losing events in case of a server malfunction.

Lets look at those configuration parameters one by one and see how they affect reliability of message storage in Kafka and what are the trade-offs involved.

Replication Factor

The topic level configuration is `replication.factor` and at the broker level you control the `default.replication.factor` for automatically created topics.

Until this point, throughout the book, we always assumed that topics have replication factor of 3, meaning that each partition is replicated 3 times on 3 different brokers. This was a reasonable assumption, since this is Kafka's default - but this is also a configuration that users can modify. Even after a topic exist, you can choose to add or remove replicas and thereby modify the replication factor.

Replication factor of N allows you to lose N-1 brokers while still being able to read and write data to the topic reliably. So higher replication factor leads to higher availability, higher reliability and fewer disasters. On the flip-side, for replication factor of N, you will need at least N brokers and you will store N copies of the data, meaning you need N times as much disk space. We are basically trading off availability for hardware.

So how do you determine the right number of replicas for a topic? It is based on how critical a topic is and how much you are willing to pay for higher availability. It also depends a bit on how paranoid you are.

If you are totally ok with a specific topic being unavailable when a single broker is restarted (which is part of normal operations of a cluster), then replication factor of 1 may be enough. Don't forget to make sure your management and users are also ok with this tradeoffs - you are saving on disks or servers, but losing high availability. Replication factor of 2 means you can lose one broker and still be ok, which sound like enough. However, keep in mind that losing one broker can sometimes (mostly on older versions of Kafka) send the cluster into unstable state, forcing you to restart another broker - the Kafka Controller. This means that with replication factor of 2, you may be forced to go into unavailability in order to recover from operational issue. This can be a tough choice.

For those reasons, we recommend replication factor of 3 for any topic where availability is an issue. In rare cases this is considered not safe enough - we've seen banks run critical topics with 5 replicas, just in case.

Placement of replicas is also very important. By default, Kafka will make sure each replica for a partition is on a separate broker. However, in some cases, this is not safe enough. If all replicas for a partitions are placed on brokers that are on the same rack and the top-of-rack switch misbehaves, you will lose availability of the partition regardless of the replication factor. To protect against rack level misfortune, we recommend placing brokers in multiple racks and using `broker.rack` broker configuration parameter to configure the rack name for each broker. If rack names are configured, Kafka will make sure replicas for a partition are spread across multiple racks, in order to guarantee even higher availability. In Chapter 5 we provided details of how Kafka places replicas on brokers and racks, if you are interested in understanding more.

Unclean Leader Election

This configuration is only available at the broker (and in practice, cluster-wide) level. The parameter name is `unclean.leader.election.enable` and by default it is set to `true`.

As explained earlier, when the leader for a partition is no longer available, one of the in-sync replicas will be chosen as the new leader. This leader election is "clean" in the sense that it guarantees no loss of committed data - by definition, committed data exists on all in-sync replicas.

But what do we do when no in-sync replica exists except for the leader which just became unavailable?

This situation can happen in one of two scenarios: * Partition had 3 replicas, the two followers became unavailable (lets say two brokers crashed). In this situation, as producers continue writing to the leader, all the messages are acknowledged and committed (since the leader is the one and only in-sync replica). Now lets say that the leader becomes unavailable (oops, another broker crash). In this scenario, if one of replicas that used to be followers starts first - we have an out of sync replica as the only available replica for the partition. * Partition had 3 replicas and due to network issues the two followers fell behind, so that even though they are up and replicating, they are no longer in sync. The leader keeps accepting messages as the only in-sync replica. Now if the leader becomes unavailable, the two available replicas are no longer in sync.

In both these scenarios we need to make a difficult decision:

- If we don't allow the out of sync replica to become the new leader, the partition will remain offline until we bring the old leader (and the last in-sync replica) back online. In some cases (for example, memory chip needs replacement), this can take many hours.
- If we do allow the out of sync replica to become the new leader, we are going the lose all messages that were written to the old leader while that replica was out of sync and also cause some inconsistencies in consumers. Why? Imagine that while replicas 0 and 1 were not available, we wrote messages with offsets 100-200 to replica 2 (then the leader). Now replica 3 became unavailable and replica 0 is back online. Replica 0 only has messages 0-100 but not 100-200. If we allow replica 0 to become the new leader, it will allow producers to write new messages and allow consumers to read them. So, now the new leader has completely new messages 100-200. First, lets note that some consumers may have read the old messages 100-200, some consumers got the new 100-200 and some got a mix of both. This can lead to pretty bad consequences when looking at things like downstream reports. In addition, replica 2 will come back online and become a follower of the new leader. At that point, it will delete any messages it got that are ahead of the current leader. Those messages will not be available to any consumer in the future.

To summarize the decision: If we allow out of sync replicas to become leaders, we risk data loss and data inconsistencies. If we don't allow them to become leaders, we face lower availability as we must wait for the original leader to become available before the partition is back online.

Setting `unclean.leader.election.enable` to true, means we allow out of sync replicas to become leaders (knowns as "unclean election"), knowing that we will lose messages when this occurs. If we set it to false, we choose to wait for the original leader to come back online, resulting in lower availability. We typically see unclean leader election disabled (configuration set to false) in systems where data quality and consistency are critical - banking systems are a good example, most banks would rather be unable to process credit card payments for few minutes or even hours than risk processing a payment incorrectly. In systems where availability is more important, such as real-time clickstream analysis, unclean leader election is often enabled.

Minimum In-Sync Replicas

Both the topic and the broker level configuration are called `min.insync.replicas`.

As we've seen, there are cases where even though we configured a topic to have 3 replicas, we may be left with a single in-sync replica. If this replica becomes unavailable, we may have to choose between availability and consistency - Never an easy choice. Note that part of the problem is that per Kafka reliability guarantees, data is considered committed when it is written to all in-sync repicas. Even when "all" means just one and the data could be lost if that replica is unavailable.

If you would like to be sure that committed data is written to more than one replica, you need to set the minimum number of in-sync replicas to a higher value. On topic with 3 replicas, setting `min.insync.replicas` to 2 means that you can only write to a partition if at least 2 of the 3 replicas are in-sync.

When all three replicas are in-sync, everything proceeds normally. Same if one of the replicas becomes unavailable. However, if 2 out of 3 replicas are not available, the brokers will no longer accept produce requests. Instead producers that attempt to send data will receive `NotEnoughReplicasException`. Consumers can continue reading existing data. In effect, with this configuation, a single in-sync replica becomes read-only. This prevents the undesirable situation where data is produced and consumed, only to disappear when unclean election occurs. In order to recover from this read-only situation, we must make one of the two unavailable partitions available again (maybe restart the broker) and wait for it to catch up and become in-sync.

Using Producers in Reliable System

Even if we configure the brokers in the most reliable configuration possible, the system as a whole can still accidentally lose data if we don't configure the producers to be reliable as well.

Here are two example scenarios to demonstrate this:

- We configured the brokers with 3 replicas and unclean leader election is disabled. So we should never lose a single message that was committed to the Kafka cluster. However, we configured the producer to send messages with acks=1 (i.e. message is considered written to Kafka successfully after the leader got it). We send a message from the producer and it was written to the leader, but not yet to the in-sync replicas. The leader sent back a response to the producer saying "Message was written successfully" and immediately crashes before the data was replicated to the other replicas. The other replicas are still considered in-sync (remember that it takes a while before we declare a replica out of sync) and one of them will become the leader. Since the message was not written to the replicas, it will be lost. But the producing application thinks it was written successfully. The system is consistent since no consumer saw the message (it was never committed since the replicas never got it), but from the producer perspective - a message was lost.

- We configured the brokers with 3 replicas and unclean leader election is disabled. We learned from our mistakes and started producing messages with acks=*all*. Suppose that we are attempting to write a message to Kafka, but the leader for the partition we are writing to just crashed and a new one is still getting elected. Kafka will respond with "Leader not Available". At this point, if the producer doesn't handle the error correctly and doesn't retry until the write is successful - the message may be lost. Once again, this is not a broker reliability issue, the broker never got the message and it is not a consistency issue, the consumers never got the message either. But if producers don't handle errors correctly, they may cause message loss.

So, how do we avoid those tragic results? As the examples show, there are two important things everyone who write applications that produce to Kafka must pay attention to: * Use the correct "acks" configuration to match reliability requirements * Handle errors correctly both in configuration and in code

We discussed producer modes in-depth in Chapter 3, but lets go over the important points again:

Send Acknowledgements

Producers can choose between 3 different acknowledgement modes:

`acks = 0` means that a message is considered to be written successfully to Kafka if the producer managed to send it over the network. You will still get errors if the object your are sending cannot be serialized or if the network card failed, but you won't get any error if partition is offline or if the entire Kafka cluster decided to take a long vacation. This means that even in the expected case of a clean leader election, your producer will lose messages - because it won't know that the leader is unavailable while a new leader is being elected. Running with acks=0 is very fast (which is why you see a lot of benchmarks with this configuration), you can get amazing throughput and utilize most of your bandwidth, but you are guaranteed to lose some messages if you choose this route.

`acks = 1` means that the leader will send either an acknowledgment or an error the moment it got the message and wrote it to the partition data file (but not necessarily synced to disk). This means that under normal circumstances of leader election, your producer will get `LeaderNotAvailableException` while a leader is getting elected, and if the producer handles this error correctly (see next section), it will retry sending the message and it will arrive safely to the new leader. You can still lose data in the scenario where the message was successfully written to the leader, but then the leader crashed before the message was propagated to the replicas.

`acks = all` means that the leader will wait until all in-sync replicas got the message before sending back an acknowledgement or an error. In conjunction with min.insync.replica configuration on the broker, this lets you control how many replicas got the message before it is acknowledged. This is the safest option - the producer won't stop trying to send the message before it is fully committed. This is also the slowest option - the producer waits for all replicas to get all the messages before it can mark the message batch as "done" and carries on. The effects can be mitigated by using async-mode for the producer and by sending larger batches, but this option will typically get you lower throughput.

Configuring Producer Retries

There are two parts to handling errors in the producer: The errors that the producers handle automatically for you and the errors you, as the developer using the producer library must handle.

The producer can handle for you *retriable* errors that are returned by the broker. When the producer sends messages to a broker, the broker can return either success or an error code. Those error codes belong to two categories - errors that can be resolved after retrying and errors that won't be. For example, if the broker returns the error code for LEADER_NOT_AVAILABLE error, the producer can try sending the error again - maybe in the meanwhile a new broker was elected and the second attempt will succeed. This means that LEADER_NOT_AVAILABLE is a *retriable* error. On the other hand, if a broker returns INVALID_CONFIG exception, trying the same message again will not change the configuration, so there is simply no point in retrying. This is an example of non-retriable error.

In general, if your goal is to never lose a message - your best approach is to configure the producer to keep trying to send the messages when it encounters a retriable error. Why? Because things like lack of leader or network connectivity issues ofter take few seconds to resolve - and if you just let the producer keep trying until it succeeds, it means you don't need to handle these issues in any other way. I frequently get asked "how many times should configure the producer to retry?" and the answer really depends on what you are planning on doing after the producer throws an exception that it retried N times and gave up. If your answer is "I'll catch the exception and retry some more" - you definitely need to set the number of retries higher and let the producer continue trying. You want to stop retrying when the answer is either "I'll just drop the message - after N retries it is too late to try delivering it anywhere" or "I'll just write it somewhere else and handle it later". Note that Kafka's cross-DC replication tool (MirrorMaker, which we'll discuss in Chapter 8) is configured by default to retry endlessly (i.e retries = MAX_INT) - because as a highly reliable replication tool, it should never just drop messages.

Note that retrying to send a failed message often includes a small risk that both messages were successfully written to the broker - leading to duplicates. For example, if network issues prevented the broker acknowledgement from reaching the producer, but the message was successfully written and replicated, the producer will treat the lack of acknowledgement as a temporary network issue and will retry sending the message (since it can't know that it was received). In that case, the broker will end up having the same message twice. Retries and careful error handling can guarantee each message will be stored *at least once*, but in the current version of Apache Kafka (0.10.0) we can't guarantee it will be stored *exactly once*. Many real-world applications add a unique identifier to each message to allow detecting duplicates and cleaning them when consuming the messages. Other applications make the messages *idempotent* - meaning that even if the same message is sent twice, it has no negative impact on correctness. For example, the message "Account value is 110$" is idempotent, since sending it several time makes no different to the result. While the message "Add 10$ to the account" is not.

Additional Error Handling

Using the built-in producer retries is an easy way to correctly handle a large variety of errors without loss of messages, but as a developer, you must still be able to handle other types of errors. These include: * Non-retriable broker errors such as errors regarding message size, authorization errors, etc. * Errors that occur before the message was sent to the broker - for example, serialization errors. * Errors that occur when the producer exhausted all retry attempts or when the available memory used by the producer is filled to the limit due to using all of it to store messages while retrying.

In Chapter 3 we discussed how to write error handlers for both sync and async message sending methods. The contents of these error handlers is specific to the application and its goals - do you throw away "bad messages"? Log errors? Store these messages in a directory on the local disk? trigger a callback to another application? these decisions are specific to your architecture. Just note that if all your error handler is doing is retrying to send the message, you are better off relying on the producer's retry functionality.

Using Consumers in Reliable System

Now that we learned how to produce data while taking Kafka's reliability guarantees into account, it is time to see how to consume data.

As we've seen in the first part of this chapter, data is only available to consumer after is has been committed to Kafka - meaning it was written to all in-sync replicas. This means that consumers already get data that is guaranteed to be consistent. The only thing consumers are left to do is make sure they keep track of which messages they've read and which messages they didn't. This is key to not losing messages while consuming them.

When reading data from a partition, a consumer is fetching a batch of events, checking the last offset in the batch and then requests another batch of events starting from the last offset received. This guarantees that Kafka consumer will always get new data in correct order without missing any events while it is running.

When a consumer stops, another consumer needs to know where to pick up the work - what was the last offset that the previous consumer processed before it stopped? The "other" consumer can even be the original one after a restart. It doesn't really matter - some consumer is going to pick up consuming from that partition and it needs to know in which offset to start. This is why consumers need to "commit" their offsets. They store their current location in the partition so they or another consumer in the group will be able to pick up the work if they stop. The main way consumers can lose messages is when committing offsets for events they've read, but didn't complete processing yet. This way, when another consumer picks up the work, it will skip those events and they will never get processed. This is why paying careful attention to when and how offsets get committed is critical.

```
 Note that this is different from a "committed message" which, as discussed previously, is a message that was written t
```

In Chapter 4 we discussed the consumer API in detail and covered the many methods for committing offsets and many details of using them. Here we will cover some important considerations and choices, but refer you back to Chapter 4 for the details of using the APIs.

Important Consumer Configuration for Reliable Processing

There are four consumer configuration properties that are important to understand in order to configure your consumer to a desired reliability behavior.

The first is `group.id` - this was explained in great detail in Chapter 4, but the basic idea is that if two consumers have the same group id and subscribe to the same topic, each will be assigned a subset of the partitions in the topic and will therefore only read a subset of the messages individually (but all the messages will be read by the group as a whole). If you need a consumer to see, on its own, every single message in the topics it subscribed to - it will need a unique group.id.

The second relevant configuration is `auto.offset.reset` - this parameter controls what the consumer will do when no offsets were committed (for example, when the consumer first starts) or when the consumer asks for offsets that don't exist in the broker (Chapter 4 explains how this can happen). There are only two options here. If you choose `earliest`, the consumer will start from the beginning of the partition whenever it doesn't have a valid offset. This can lead to the consumer processing a lot of messages twice, but guarantees to minimize data loss. If you choose `latest`, the consumer will start at the end of the partition. This minimizes duplicate processing by the consumer, but almost certainly leads to some messages getting missed by the consumer.

The third relevant configuration is `enable.auto.commit` - this is a big decision that you need to make - are you going to let the consumer commit offsets for you based on schedule, or are you planning on committing offsets manually in your code. The main benefit of automatic offset commits is that its one less thing to worry about when implementing your consumer. If you do all the processing of consumed records within the consumer poll loop, then the automatic offset commit guarantees you will never commit an offset that you didn't process (If you are not sure what is the consumer poll loop, refer back to Chapter 4). The main drawbacks of automatic offset commits is that you have no control over the number of duplicate records you many need to process (because your consumer stopped after processing some records but before the automated commit kicked in) and if you do anything fancy like pass records to another thread to process in the background, the automatic commit may commit offsets for records the consumer has read but perhaps did not process yet.

The fourth relevant configuration is tied to the third, it is `auto.commit.interval.ms` - if you choose to commit offsets automatically, you can configure how frequently they will be committed. The default is every 5 seconds. In general, committing more frequently adds some overhead, but reduces the number of duplicates that can occur when a consumer stops.

Explicitly Committing Offsets in Consumer

If you chose to go with the automatic offset commits, you don't need to worry about explicitly committing offsets. You do need to think about how you will commit offsets if you decided you need more control over the timing of offset commits - either in order to minimize duplicates or because you are doing event processing outside the main consumer poll loop.

I will not go over the mechanics and APIs involved in committing offsets here, since they are covered in great depth in Chapter 4.

Instead, we will review important considerations you need to take into account when developing a consumer to make sure it will handle data reliably. We'll start with the simple and perhaps obvious points and move to discuss more complex patterns.

ALWAYS COMMIT OFFSETS AFTER EVENTS WERE PROCESSED

If you do all the processing within the poll loop and don't maintain state between poll loops (for example, for aggregation), this should be easy. You can use the auto-commit configuration or commit events at the end of the poll loop.

COMMIT FREQUENCY IS A TRADE-OFF BETWEEN PERFORMANCE AND NUMBER OF DUPLICATES IN THE EVENT OF A CRASH

Even in the simplest case where you do all the processing within the poll loop and don't maintain state between poll loops, you can choose to commit multiple times within a loop (perhaps even after every event) or choose to only commit every several loops. Committing has some performance overhead (similar to produce with acks=all), so it all depends on the trade-offs that work for you.

MAKE SURE YOU KNOW EXACTLY WHAT OFFSETS YOU ARE COMMITTING

A common pitfall when committing in the middle of the poll loop is to accidentally commit the last offset read when polling and not the last offset processed. Remember that it is critical to always commit offsets for messages after they were processed - committing offsets for messages read but not processed can lead to the consumer missing messages. Chapter 4 has examples showing how to do just that.

REBALANCES

When designing your application, remember that consumer rebalances will happen and you need to handle them properly. Chapter 4 contains few examples, but this usually involves committing offsets before partitions are revoked and cleaning any state you maintain when you are assigned new partitions.

CONSUMERS MAY NEED TO RETRY

In some cases, after calling poll and processing records, some records were not fully processed and you may want to retry processing them later. For example, you may try to write records from Kafka to a database, but the database is not available at that moment and you may wish to retry later. Note that unlike traditional pub-sub messaging systems, you commit offsets and not "ack" individual messages. This means that if you failed to process record #30 and succeeded in processing record #31, you should not commit record #31 - this will result in committing all the records up to #31 including #30, which is usually not what you want. Instead, try following one of these two patterns:

One option, when you encounter a retriable error, is to commit the last record you processed successfully. Then store the records that still need to be processed in a buffer (so the next poll won't override them), use the consumer `pause()` method to ensure that additional polls won't return data (so you wouldn't need to worry about your buffer running too full while retrying), keep polling (refer to Chapter 4 on why you should never stop polling) and in-between polling also keep retrying. If you succeed, or if you retried enough times and decided to give up, log an error and throw the records away, you can call `resume()` to un-pause the consumer and the next poll will return new records to process.

A second option is, when encountering a retriable error is to write it to a separate topic and continue. A separate consumer group can be used to handle retries from the retry topic, or one consumer can subscribe to both the main topic and to the retry topic, but pause the retry topic between retries. This pattern is similar to the dead-letter-queue system used in many messaging systems.

CONSUMERS MAY NEED TO MAINTAIN STATE

When you want to maintain state across multiple calls to poll, for example, if you try to calculate moving average: You'll want to be able to calculate average after calling poll the first time, but then update the average in subsequent calls. If your process is restarted, you will need to not just start consuming from last offset, but you'll also need to recover the matching moving average. One way to do it is to write the latest accumulated value to a "results" topic at the same time you are committing offset. This means that when a thread is starting up, it can pick up the latest accumulated value when it starts and pick up right where it left off. This doesn't completely solve the problem, as Kafka does not offer transactions yet. You could crash after you wrote the latest result and before you committed offsets, or vice versa. In general, this is a rather complex problem to solve, and rather than solving it on your own, we recommend looking at a library like KafkaStreams that provides high level DSL-like API for aggregation, joins, windows and other complex analytics.

HANDLING LONG PROCESSING TIMES

When you know that processing records could take a long time: Maybe you are fetching information from an external system that can block, writing to an external system or doing a very complex calculation. Remember that you can't stop polling for more than few seconds. Even if you don't want to grab additional records, you must continue polling so the client can send heartbeats to the broker. A common pattern in those cases is to hand off the data to process to a thread-pool, when possible with multiple threads so they can speed things up a bit by processing in parallel. After handing off the records to the worker threads, you can pause the consumer, and keep polling without actually fetching additional data until the worker-threads finished. Once they are done, you can resume the consumer. Because the consumer never stops polling, the heartbeat will be sent as planned and rebalancing will not be triggered.

EXACTLY ONCE DELIVERY

Some applications require not just at-least-once semantics (meaning no data loss), but also exactly-once semantics. While Kafka does not provide full exactly-once support at this time, consumers have few tricks available that allow them to guarantee that each message in Kafka will be written to an external system exactly once (not that this doesn't handle duplications that may have occured while the data was produced into Kafka).

The easiest, and probably most common way to do exactly-once is by writing results to a system that has some support for unique keys. This includes all key-value stores, all relational databases, Elastic search and probably many more data stores. In those cases, either the record itself contains a unique key (this is fairly common), or you can create a unique key using the topic, partition and offset combination - which uniquely identifies a Kafka record. If you write the record as a value with a unique key, and later you accidentally consume the same record again, you will just write the exact same key and value. The data-store will override the existing one, and you will get the same result as you would without the accidental duplicate. This pattern is called **idempotent writes** and is very common and very useful.

Another option is available when writing to a system that has transactions. Relational databases are the easiest example, but HDFS has atomic renames that are often used for the same purpose. The idea is to write the records and their offsets in the same transaction, so they will be in sync. When starting up, retrieve the offsets of the latest records written to the external store and then use `consumer.seek()` to start consuming again from those offsets. Chapter 4 contains an example of how this can be done.

Validating System Reliability

Once you went through all the process of figuring out your reliability requirements, configuring the brokers, configuring the clients, using the APIs in the best way for your use-case… now you can just relax and run everything in production confident that no event will ever be missed, right?

You could do that, but we recommend doing some validation first. We suggest three layers of validation - validate then configuration, validate the application, and monitor the application in production. Lets look at each of these steps and see what you would want to validate and how.

Validating Configuration

It is easy to test the broker and client configuration in isolation from the application logic, and it is recommended to do so for two reasons: * It helps to test if the configuration you've chosen can meet your requirements * It is good exercise to reason through the expected behavior of the system, test to see what the behavior actually is, and if there are any gaps you can improve your understanding of Kafka's basic principals and how they apply in different scenarios. This chapter was a bit theoretical, so checking your understanding of how the theory applies in practice is important.

Kafka includes two important tools to help with this validation. The `org.apache.kafka.tools` package includes VerifiableProducer and VerifiableConsumer classes. These can run as command line tools, or be embedded in automated testing framework.

The idea is that the verifiable producer produces a sequence of messages containing numbers from 1 to a value you choose. You can configure it the same way you will configure your own producer, setting the right number of acks, retries and rate at which the messages will be produced. When you run it, it will print success or error for each message sent to the broker, based on acks received. The verifiable consumer performs the complementary check - it consumes events (usually those produced by the verifiable producer) and prints out the events it consumed in order. It also prints information regarding commits and rebalances.

You will usually want to think about some tests you want to run. For example:

- Leader election - what happens if I kill the leader? How long does it take the producer and consumer to start working as usual again?
- Controller election - how long does it take the system to resume after a restart of the controller?
- Rolling restart - can I restart the brokers one by one without losing any messages?
- Unclean leader election test - what happens when we kill all the replicas for a partition one by one (to make sure each goes out of sync) and then start a broker that was out of sync? what needs to happen in order to resume operations? is this acceptable?

Then you pick a scenario, start the verifiable producer, start the verifiable consumer and run through the scenario - for example, kill the leader of the partition you are producing data into. If you expected a short pause and then everything to resume normally with no message loss, make sure the number of messages produced by the producer and the number of messages consumed by the consumer match.

Apache Kafka source repository includes an extensive test suite. Many of the tests in the suite are based on the same principal - use the verifiable producer and consumer to make sure rolling upgrades work for example.

Validating Applications

Once you are sure your broker and client configuration are up to your requirements, it is time to test whether your application provides the guarantees you need. This will check things like your custom error handling code, offset commits, rebalance listeners and similar places where your application logic interacts with Kafka's client libraries.

Naturally, because it is your application, we can provide less guidance on how to test it. Hopefully you have integration tests for your application as part of your development process. However you validate your application, we recommend to run the tests under variety of failure conditions:

- Clients lose connectivity to the server (your system administrator can assist you in simulating network failures)
- Leader election
- Rolling restart of brokers
- Rolling restart of consumers
- Rolling restart of producers

For each scenario you will have "expected behavior", which is what you planned on seeing when you developed your application, and then you can run the test to see that reality matches your plans. For example, when planning for "rolling restart of consumers", you may plan for "short pause as consumers rebalance and then continued consumption with no more than 1000 duplicate values". Your test will show whether or not the way the application commits offsets and handles rebalances actually provides this behavior.

Monitoring Reliability in Production

Testing the application is important, but it does not replace the need to continuously monitor your production systems to make sure data is flowing as expected. Chapter 9 will contain detailed suggestions on how to monitor the Kafka cluster, but in addition to monitoring the health of the cluster, it is important to also monitor the clients and the flow of data through the system.

First, Kafka's Java clients include JMX metrics that allow monitoring client-side status and events. On the producers, the two metrics most important for reliability are error-rate and retry-rate per record (aggregated). Keep an eye of those since error rate or retry rate going up can indicate an issue with the system. Also monitor the producer logs - errors sending events are logged at `WARN` level, and say something along the lines of "Got error produce response with correlation id 5689 on topic-partition [topic-1,3], retrying (2 attempts left). Error: …". If you see events with 0 attempts left, this indicates the producer is running out of retries. Based on the discussion in "Producer Configuration" section of this chapter, you may want to increase the number of retries - or solve the problem that caused the errors in first place.

On the consumer side, the most important metric is consumer-lag. This metric indicates how far the consumer is from the latest message committed to the partition on the brokers. Ideally, the lag would always be zero and the consumer will always read the latest message. In practice, because calling `poll()` returns multiple messages and then the consumer spends time processing them before fetching more messages, the lag will always fluctuate a bit. What is important is to make sure consumers do eventually catch up rather than fall farther and farther behind. Because of the expected fluctuation in consumer lag, setting traditional alerts on the metric can be challenging. Burrow is a consumer lag checker by LinkedIn and can make this easier.

Monitoring the flow of data also means making sure all produced data is consumed in a timely manner (your requirements will dictate what "timely manner" means). In order to make sure data is consumed in a timely manner, you need to know when the data was produced. Kafka assists in this: since starting with version 0.10.0 all messages include a timestamp which indicates when the event was produced. If you are running clients with an earlier version, we recommend recording timestamp, name of the app producing the message and hostname where the message was created, in each event. This will help tracking down sources of issues later on.

In order to make sure all produced messages were consumed within reasonable amounts of time, you will need the application producing the code to record the number events produced (usually as "events per second"), and then the consumers need to both record numbers of events consumed (also "events per second") and also the gap from the timestamp on the event (time the event was produced) to the current time (in which the event was consumed). Then you will need a system to reconcile the "events per second" numbers from both the producer and the consumer (to make sure no messages were lost on the way) and to make sure the time gaps between the time events were produced and time there were consumed do not exceed the requirements. For even better monitoring, you can add a "monitoring consumer" on critical topics that will just count events and compare them to the "events produced" counts, so you will get accurate monitoring of producers even if no one is consuming the events at a given point in time. These type of end-to-end monitoring systems can be challenging and time-consuming to implement. To the best of our knowledge there was no open source implementation of such system at the time the book was written. Confluent provides a commercial implementation as part of the Confluent Control Center (http://www.confluent.io/product/control-center).

Final Notes

As we said in the beginning of the chapter - Reliability is not just a matter of specific Kafka features. You need to build an entire reliable system, including your application architecture, the way your application uses the producer and consumer APIs, producer and consumer configuration, topic configuration and broker configuration. Making the system more reliable always has trade-offs, in application complexity, performance, availability or disk-space usage. By understanding all the options and common patterns and understanding requirements for your use-case, you can make informed decisions regarding how reliable your application and Kafka deployment needs to be and which trade-offs makes sense for you.

PREV
5. Kafka Internals

NEXT
7. Building Data Pipelines