Chapter 8. Cross-Cluster Data Mirroring

For most of the book we discuss the setup, maintenance and use of a single Kafka cluster. There are, however, few scenarios in which one may design an architecture using more than a single cluster.

In some cases the clusters are completely separated. They belong to different departments or different use-cases and there is no reason to copy data from one cluster to another. Sometimes different SLAs or different workloads make it difficult to tune a single cluster to serve multiple use-cases. Other times there are different security requirements. Those use-cases are fairly easy - managing multiple distinct clusters is the same as running a single cluster multiple times.

In other use-cases, the different clusters are interdependent and the administrators need to continuously copy data between the clusters. In most databases, continuously copying data between database servers is called “replication”. Since we’ve used “replication” to describe movement of data between Kafka nodes that are part of the same cluster, we’ll call copying of data between Kafka clusters “mirroring”. Apache Kafka’s built-in cross-cluster replicator is called “MirrorMaker”

In this chapter we will discuss cross-cluster mirroring or either all or part of the data. We’ll start by discussing some of the common use-cases for cross-cluster mirroring. Then we’ll show few architectures that are used to implement these use cases and discuss the pros and cons of each architecture pattern. We’ll then discuss MirrorMaker itself and how to use it. We’ll share operational tips, including deployment and performance tuning. We’ll finish by comparing few alternatives to MirrorMaker.

Use-Cases of Cross-Cluster Mirroring

- **Regional and central clusters**: In some cases the company has one or more data centers in different geographical regions, different cities or different continents. Each one of those data centers has its own Kafka cluster. Some applications can work just by communicating with the local cluster, but there are at least some applications that require data from multiple data centers (otherwise you wouldn’t be looking at cross data-center replication solutions). There are many cases when this is a requirement, but the classic is a company that modifies prices based on supply and demand. This company can have a data center in each city in which it has presence, collect information about local supply and demand and adjust prices accordingly. All this information will then be mirrored to a central cluster where business analysts can run company-wide reports on its revenue.

- **Redundancy (DR):** All the applications can run on just one Kafka cluster, but you are concerned about a situation where the entire cluster becomes unavailable for some reason. You’d like to have a second Kafka cluster with all the data that exists in the first cluster, so in case of emergency you can direct your applications to the second cluster and continue as usual.

- **Cloud migrations:** Many companies these days run their business in both on-premise data center and using a cloud provider. Often applications run on multiple regions of the cloud provider, for redundancy and sometimes multiple cloud provides are used. In these cases, there is often at least one Kafka cluster in each on-premise data center and each cloud region. Those Kafka clusters are used by applications in each data center and region separately and also to transfer data efficiently between the data centers. For example, if a new application is deployed in the cloud but requires some data that is updated by applications running in the on-premise data

center and stored in an on-premise database, one can use Kafka Connect to capture database changes to the local Kafka cluster and then mirror these changes to the cloud Kafka cluster where the new application can use them. This helps control costs of cross data-center traffic as well as improve governance and security of the traffic.

Multi-Cluster Architectures

Now that we've seen few use-cases that require multiple Kafka clusters, we want to look at few common architectural patterns that we've successfully used when implementing these use-cases. Before we go into the architectures, we'll give a brief overview of the realities of cross data-center communications. The solutions we'll discuss could seem overly complicated without understanding that they represent trade-offs in the face of specific network conditions.

Some realities of cross data-center communication

- **High latencies**: Latency of communication between two Kafka clusters increase as the distance between the clusters increases. In addition to the speed of light being constant, more network hops there are between the two clusters means more delays due to buffering or congestion.
- **Limited bandwidth**: Wide area networks (WANs) typically have far lower available bandwidth than what you'll see inside a single data center and the available bandwidth can vary minute to minute. In addition, higher latencies make it more challenging to utilize all the available bandwith.
- **Higher costs**: Regardless of whether you are running Kafka on-premise or in the cloud, there are higher costs to communication between clusters. Partially because the bandwidth is limited and adding bandwidth can be prohibitively expensive, and because of the prices vendors charge for trasferring data between data centers, regions and clouds.

Apache Kafka's brokers and clients were designed, developed, tested and tuned all within a single data center. We assumed low latency and high bandwidth between brokers and clients. This is apparent in default timeouts and sizing of various buffers. For this reason, it is not recommended (except in specific cases which we'll discuss later) to install some Kafka brokers in one data-center and others in another data-center.

In most cases we'd prefer to avoid producing data to a remote data center, and when you do - you need to account for higher latency and more potential for network partitions (temporary disconnection between the producer and the brokers) by increasing the number of retries (LinkedIn configured their cross data-center mirroring to retry events over 32000 times until they succeed in sending them) and the size of the buffers that hold records between attempts to send them.

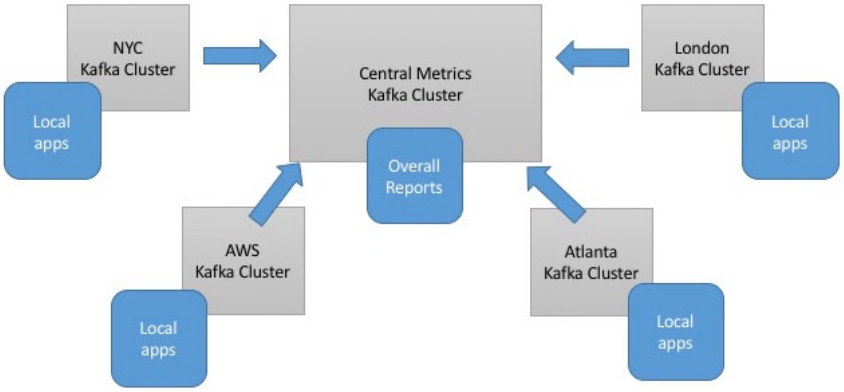
If we are to have any kind of replication between clusters and we ruled out inter-broker communication and producer-broker communication, then we must allow for broker-consumer communication. Indeed, this is the safest form of cross-cluster communication, because in the event of network partition that prevents a consumer from reading data, the records remain safe inside the Kafka brokers until communications resume and consumers can read them. There is no risk of accidental data loss due to network partitions. Still, because bandwidth is limited, if there are multiple applications in one data-center that need to read data from Kafka brokers in another data-center, we'll prefer to install a Kafka cluster in each data-center and mirror the necessary data between them once, rather than have multiple applications consume the same data across the WAN.

We'll talk more about tuning Kafka for cross data-center communication, but the following principals will guide most of the architectures we'll discuss next:

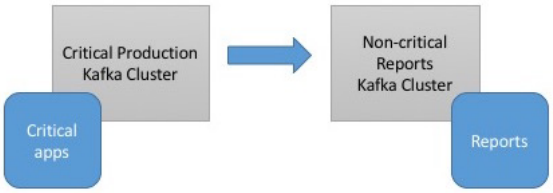
- No less than one cluster per data-center
- Replicate each event exactly once (barring retries due to errors) between each pair of data-centers
- When possible, consume from a remote data-center rather than produce to a remote data-center

Hub and Spokes Architecture

This architecture is intended for the case where there are multiple local Kafka clusters and one central Kafka cluster:



There is also a simpler variation of this architecture with just two clusters - a leader and a follower:



This architecture is used when data is produced in multiple data-centers and some consumers need access to the entire data set. The architecture also allows for applications in each data-center that only process data local to this specific data-center. But it does not give access to the entire data set from each data-center.

The main benefits of this architecture is that data is always produced to the local data-center and that events from each data-center are only mirrored once - to the central data-center. Applications that process data from a single data-center can be located at that data-center. Applications that need to process data from multiple data-centers will be located at the central data-center where all the events are mirrored. Because replication always goes in one direction and because each consumer always reads from the same cluster, this architecture is simple to deploy, configure and monitor.

The main drawbacks of this architecture are direct results of its benefits and simplicity. Processors in one regional data-center can't access data in another. To understand better why this is a limitation, lets look at an example of this architecture:

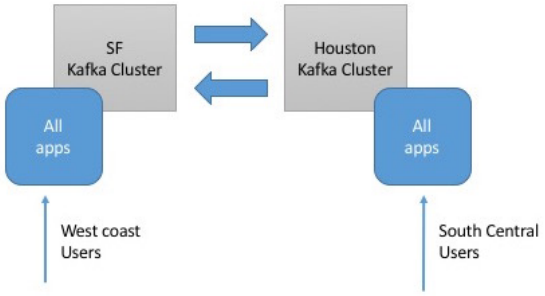
Suppose that we are a large bank and have branches in multiple cities. Lets say that we decided to store the user profiles and their account history in a Kafka cluster in each city. We replicate all this information to a central cluster that is used to run the bank's business analytics. When users connect to the bank website or visit their local branch, they are routed to send events to their local cluster and read events from the same local cluster. However, suppose that a user visits a branch in a different city - because the user information doesn't exist in the city he is visiting, the branch will be forced to interact with a remote cluster (not recommended) or have no way to access the user's information (really embarrassing).

For this reason, use of this pattern is usually limited to only parts of the data set that can be completely separated between regional data-centers.

When implementing this architecture, for each regional data-center you need at least one mirroring process on the central data-center. This process will consume data from each remote regional cluster and produce it to the central cluster. If the same topic exists in multiple data-centers, you can write all the events from this topic to one topic with the same name in the central cluster, or write events from each data-center to a separate topic.

Active-Active Architecture

This architecture is used when 2 or more data-centers share some or all of the data and each data-center is able to both produce and consume events.



The main benefits of this architecture are the ability to serve users from a near-by data-center, which typically has performance benefits, without sacrificing functionality due to limited availability of data (as we've seen happen in the hub-and-spokes architecture). A secondary benefit is redundancy and resilience. Since every data-center has all the functionality, in the event when one data-center is unavailable you can direct users to a remaining data-center. This type of failover only requires network redirects of users, typically the easiest and most transparent type of failover.

The main drawbacks of this architecture is the challenges in avoiding conflicts when data is read and updated asynchronously in multiple locations. These challenges involve technical challenges in mirroring - for example, how do we make sure the same event isn't mirrored back-and-forth endlessly? But more important, they involve challenges in data consistency.

- If a user sends an event to one data-center and reads events from another data-center, it is possible that the event they wrote didn't arrive to the second data-center yet. To the user it will look like he just added a book to his wish list, clicked on the wish list and the book isn't there. For this reason, when this architecture is used, the developers usually find a way to "stick" each user to a specific data-center and make sure they use the same cluster most of the time (unless they connect from remote location or the data-center becomes unavailable).
- An event from one data-center says user ordered book A and an event from more-or-less the same time at a second data-center says that the same user ordered book B. After mirroring, both data-centers have both events and we can say that each data-center has two conflicting events. Applications on both data-centers need to know how to deal with this situation. Do we pick one event as the "correct" one? If so, we need consistent rules on how to pick one so applications on both data centers will arrive at the same conclusion. Do we decide that both are true and simply send the user two books and have another department deal with returns? Amazon used to resolve conflicts that way, but perhaps organizations dealing with stock trades can't. The specific method for minimizing conflicts and handling them when they occur is specific to each use-case. It is important to keep in mind that if you use this architecture you **will** have conflicts and need to deal with them.

If you find ways to handle the challenges of asynchronous reads and writes to the same data set from multiple locations, then this architecture is highly recommended. It is the most scalable, resilient, flexible and cost effective option we are aware of. So it is well worth the effort to figure out solutions for avoiding replication cycles, keeping users mostly in the same data-center and handling conflicts when they occur.

Part of the challenge of active-active mirroring, especially with more than two data centers, is that you will need a mirroring process for each pair of data center and each direction. With 5 data-centers, you need to maintain at least 20 mirroring processes - and more likely 40, since each process needs redundancy for high availability.

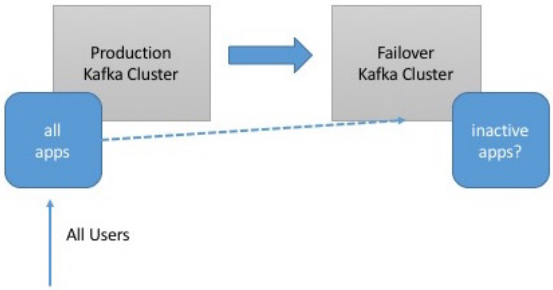
In addition, you will want to avoid loops in which the same event is mirrored back-and-forth endlessly. You can do this by giving each "logical topic" a separate topic for each data center and make sure to avoid replicating topics that originated in remote data centers. For example, logical topic "users" will be topic "SF.users" in one data-center and "NYC.users" in another data-center. The mirroring processes will mirror topic SF.users from SF to NYC, and topic NYC.users from NYC to SF. As a result, each event will

only be mirrored once, but each data center will contain both SF.users and NYC.users - which means each data-center will have information for all the users. Consumers will need to consume events from "*.users" if they wish to consume all user events. Another way to think of this setup is to see it as a separate namespace for each data-center that contains all the topics for this specific data-center. In our example we'll have the NYC and the SF namespaces.

Note that in the near future (and perhaps before you read this book), Apache Kafka will add record headers. This will allow tagging events with their originating data-center and using this header information to avoid endless mirroring loops and also to allow processing events from different data-centers separately. You can still implement this feature by using a structured data format for the record values (Avro is our favorite examples) and use this to include tags and headers in the event itself. This does require extra effort when mirroring, since none of the existing mirroring tools will support your specific header format.

Active-Standby Architecture

In some cases, the only requirement you have for multiple clusters is to support some kind of disaster scenario. Perhaps you have two clusters in the same data-center. You use one cluster all the time for all the applications, but you want a second cluster than contains (almost) all the events in the original cluster that you can use if the original cluster is completely unavailable. Or perhaps you need geographic resiliency. Your entire business is running from a data-center in California, but you need a second data-center in Texas that usually doesn't do much and that you can use in case of an earth quake. The Texas data-center will probably have an inactive ("cold") copy of all the applications that admins can start up in case of emergency and that will use the second cluster. This is often a legal requirement rather than something that the business is actually planning on doing - but you still need to be ready.



The benefits of this setup is simplicity to setup and the fact that it can be used in pretty much any use case. You simply install a second cluster and set up a mirroring process that streams all the events from one cluster to another. No need to worry about access to data, handling conflicts and other architectural complexities.

The disadvantages are waste of a good cluster and the fact that failover between Kafka clusters is, in fact, much harder than it looks.

I almost never recommend a failover cluster to Kafka users, and after reading all of the challenges and limitations - you will have second thoughts as well. The bottom line is that it is currently not possible to perform cluster failover in Kafka without either losing data or having duplicated events. Often both. You can only minimize those, but never eliminate them.

Regarding the waste of a good cluster: It should be obvious that a cluster that does nothing except wait around for a disaster is a waste of resources. Since disasters are (or should be) rare, most of the time we are looking at a cluster of machines that does nothing at all. Some organizations try to fight this issue by having a DR (disaster recovery) cluster that is much smaller than the production cluster. This is a risky decision, because you can't be sure that this minimally-sized cluster will hold up during an emergency. Other organizations prefer to make the cluster useful during non-disasters by shifting some read-only workloads to run on the DR cluster - which means they are really running a small version of hub-and-spoke architecture with a single spoke.

The more serious issue is: How does one failover to a DR cluster in Apache Kafka?

First, it should go without saying that whichever failover method you choose, your SRE team must practice it on a regular basis. A plan that works today may stop working after an upgrade, or perhaps new use-cases makes the existing tooling obsolete. Once a

quarter is usually the bare minimum for failover practices. Strong SRE teams practice far more frequently. Netflix's famous Chaos Monkey, a service that randomly causes disasters is the extreme - any day may become failover practice day.

Now, lets take a look at what is involved in a failover:

DATA LOSS AND INCONSISTENCIES IN UNPLANNED FAILOVER

Because Kafka's various mirroring solutions are all asynchronous (we'll discuss a synchronous solution in the next section), the DR cluster will not have the latest messages from the primary cluster. You should always monitor how far behind the DR cluster is and never let it fall too far behind - but in a busy system you should expect the DR cluster to be few hundreds or even few thousand messages behind the primary. If your Kafka cluster handles 1 million messages a second and there is 5 millisecond lag between the primary and the DR cluster is 5 milliseconds, this mean your DR cluster will be 5000 messages behind the primary in the best case scenario. So prepare for unplanned failover to include some data loss. In planned failover, you can stop the primary cluster and wait for the mirroring process to mirror the remaining messages before failing over applications to the DR cluster and avoid this data loss. When unplanned failover occurs and you lose few thousand messages, note that Kafka currently has no concept of transactions - which means that if some events in multiple topics are related to each other, for example sales and line-items, you can have some events arrive to the DR site in time for the failover and others that don't. Your applications will need to be able to handle a line-item without a corresponding sale after you failover to the DR cluster.

START OFFSET FOR APPLICATIONS AFTER FAILOVER

Perhaps the most challenging part in failing over to another cluster is making sure applications know where to start consuming data. There are several common approaches. Some are simple but can cause additional data loss or duplicate processing, others are more involved but minimize additional data loss and re-processing. Lets take a look at a few:

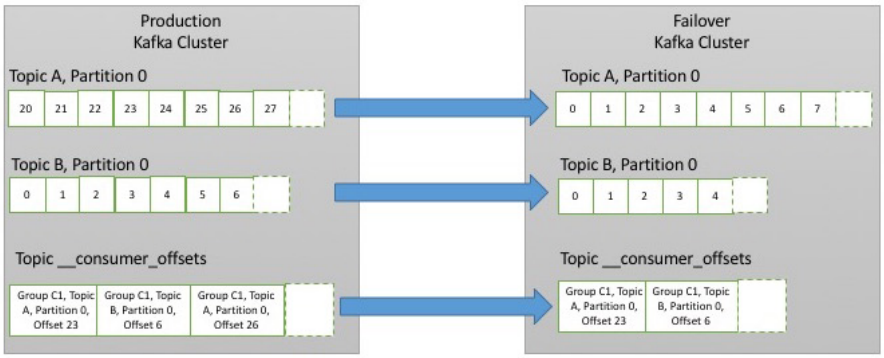
- **Auto offset Reset** - Apache Kafka consumers have a configuration for how to behave when they don't have a previously committed offset. Either start reading from the beginning of the partition or from the end of the partition. If you are using old consumers that are committing offsets to Zookeeper and you are not somehow mirroring these offsets as part of the DR plan, then you need to choose one of these options. Either start reading from the beginning of available data and handle large amounts of duplicates or skip to the end and miss an unknown (and hopefully small) number of events. If your application handles duplicates with no issues or missing some data is no big deal, this option is by far the easiest. Simply skipping to the end of the topic on failover is probably still the most popular failover method.

- **Replicate offsets topic** - If you are using new (0.9 and above) Kafka consumers, the consumers will commit their offsets to a special topic: __consumer_offsets. If you mirror this topic to your DR cluster, then when consumers start consuming from the DR cluster they will be able to pick up their old offsets and continue from where they left off. It is simple, but there is a long list of caveats involved.

  First, there is no guarantee that offsets in the primary cluster will match those in the secondary cluster. Suppose you only store data in the primary cluster for 3 days and you started mirroring a topic a week after it was created. In this case the first offset available in the primary cluster may be offset 57000000 (older events were from the first 4 days and were removed already), but the first offset in the DR cluster will be 0. So a consumer that tries to read offset 57000003 (because thats its next event to read) from the DR cluster will fail to do this.

  Second, even if you started mirroring immediately when the topic was first created and both the primary and the DR topics start with 0, producer retries mean that they could diverge later on. Simply put, there is no existing Kafka mirroring solution that preserves offsets between primary and DR clusters.

  Third, even if the offsets were perfectly preserved, because of the lag between primary and DR clusters and because Kafka currently lacks transactions, an offset committed by a Kafka consumer may arrive ahead or behind the record with this offset. A consumer that fails over may find committed offsets without matching records. Or it may find that the latest committed offset in the DR site is older than the latest committed offset in the primary site.



In these cases, you need to accept some duplicates if the latest committed offset in the DR site is older than the one committed on the primary or if the offsets in the records in the DR site are ahead of the primary due to retries. You will also need to figure out how to handle cases where the latest committed offset in the DR site doesn't have a matching record - do you start processing from the beginning of the topic, or skip to the end?

With all those caveats, this option can be used to reduce duplicates or missing records while still being very simple to implement - just always mirror topics in time to catch offset 0 and always mirror the offsets topic.

- **Time-based failover** - If you are using really new (0.10.0 and above) Kafka consumers, each message includes a timestamp indicating the time the message was sent to Kafka. In really really new Kafka versions (0.10.1.0 and above), the brokers include an index and an API for looking up offsets by the timestamp. So, if you failover to the DR cluster, and you know that your trouble started at 4:05am, you can tell consumers to start processing data from 4:03am. There will be some duplicates from those two minutes, but it is probably better than other alternatives and the behavior is much easier to explain to everyone in the company - "We failed back to 4:03am" sounds better than "We failed back to what may or may not be the latest committed offsets". So this is often a good compromise. The only question is: How do we tell consumers to start processing data from 4:03am?

  One option is to bake it right into your app. Have a user-configurable option to specify start time for the app. If this is configured, the app can use the new APIs to fetch offset by time, seek to that time and start consuming from the right point, committing offsets as usual.

  This option is great - if you wrote all your applications this way in advance. But what if you didn't? It is fairly straight-forward to write a small tool that will take a timestamp, use the new APIs to fetch offsets for this timestamp and then commit these offsets for a list of topics and partitions as a specific consumer group. We hope to add this tool to Kafka in the near future, but it is possible to write one yourself. The consumer group should be stopped while running such tool and started immediately after.

  This option is recommended for those using new versions of Kafka, who like some certainty in their failover and are willing to write a bit of custom tooling around the process.

- **External Offset Mapping**: When discussing the mirroring of the offsets topic, we've seen that one of the biggest challenges with that approach is the fact that offsets in primary and DR clusters can diverge. With this in mind, some organizations choose to use an external datastore, such as Apache Cassandra, to store mapping of offsets from one cluster to another. They built their Kafka mirroring tool in a way that whenever an event is produced to the DR cluster - both offsets are sent to the external datastore. Or they optimize it and only store the mapping whenever the difference between the two offsets changed, so if offset N on primary mapped to offset N-5 on the DR cluster and suddenly due to duplicates offset M is mapped to M-4, they record the new mapping. Then, when failover occurs, instead of mapping timestamps (which are always a bit inaccurate) to offsets, they map primary offsets to DR offsets and use those. They use one of the two techniques listed above to force consumers to start using the new offsets from the mapping. This still has an issue with offset-commits that arrived ahead of the records themselves and offset-commits that didn't get mirrored to the DR on time - but it covers some cases.

  This solution is quite complex and in my opinion almost never worth the extra effort. It dates back to before time indexes existed and could be used for failover. These days I'd opt to upgrade the cluster and use the time-based solution rather than go to the effort of

mapping offsets, which still doesn't cover all failover cases.

AFTER THE FAILOVER

Lets say that failover was successful. Everything is working just fine on the DR cluster. Now we need to do something with the primary cluster. Perhaps turn it into a DR.

It is tempting to simply modify the mirroring processes to reverse their direction and simply start mirroring from the new primary to the old one. However, this leads to two important questions:

- How do we know where to start mirroring? We need to solve the same problem we have for all of our consumers for the mirroring application itself. And remember - all our solutions have cases where they either cause duplicates or miss data - often both.
- In addition, for reasons we discussed above, it is likely that your original primary will have events that the DR cluster does not. If you just start mirroring new data back, the extra history will remain and the two clusters will be inconsistent.

For this reason, the simplest solution is to first scrape the original cluster - delete all the data and committed offsets and then start mirroring from the new primary back to what is now the new DR cluster. This starts you with a clean slate that is identical to the new primary.

FEW WORDS ON CLUSTER DISCOVERY

One of the important points to consider when planning a standby cluster is that in an event of failover, your applications will need to know how to start communicating with the failover cluster. If you hardcoded the hostnames of your primary cluster brokers in the producer and consumer properties, this will be challenging. Most organizations keep it simple and create a DNS name that usually points to the primary brokers, but in case of an emergency can be pointed at a standby. However, some organizations get fancy and use a discovery service like Zookeeper, Etcd or Consul. The discovery service (DNS or other) doesn't need to include all the brokers - Kafka clients only need to access a single broker successfully in order to get metadata about the cluster and discover the other brokers. So including just 3 brokers is usually fine. Regardless of the discovery method, most failover scenarios do require bouncing consumer applications after failover so they can find the new offsets they need to start consuming.

Stretch Clusters

Active-standby architectures are used to protect the business against the failure of a Kafka cluster, by moving applications to communicate with another cluster in case of cluster failure. Stretch clusters are intended to protect the Kafka cluster from failure in the event an entire data-center failed. They do this by installing a single Kafka cluster across multiple data-centers.

Stretch clusters are fundamentally different from other multi data-center scenarios. To start with, they are not multi-cluster - it is just one cluster. As a result, we don't need a mirroring process to keep two clusters in sync. Kafka's normal replication mechanism is used, as usual, to keep all brokers in the cluster in sync. This setup can include synchronous replication - you can configure the stretch cluster so that producers receive acknowledgement about successfully sending a message after the message was written to multiple data-center. This involves using rack definitions to make sure each partition has replicas in multiple data-centers and use of min.isr and acks=all to make sure every write is acknowledged from at least two data-centers.

The advantages of this architecture are in the synchronous replication - some types of business simply require that their DR site will always be 100% synchronized with the primary site. This is often a legal requirement and is applied to any data store across the company. Kafka included. The other advantage is that both data-centers and all brokers in the cluster are used. There is no waste of the type we saw in active-standby architectures.

The limitations of this architecture is in the limited type of disasters it protects against. It only protects from data-center failures, not any kind of application or Kafka failures. The other limitation is in the operational complexity. This architecture demands physical infrastructure of a type that not all companies can provide.

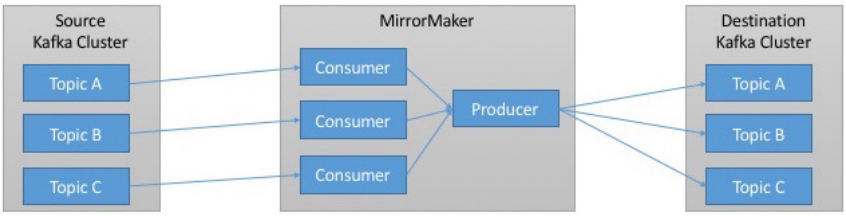
This architecture is feasible if you can install Kafka (and Zookeeper) in at least 3 data-centers with high-bandwidth and low-latency line between them. This can be done if your company owns 3 buildings on the same street. Or more common, by using 3 availability zones inside one region of your cloud provider.

The reason 3 data-centers are important is due to Zookeeper. Zookeeper requires uneven number of nodes in a cluster and will remain available if a majority of the nodes are available. With two data-centers and uneven number of nodes, one data-center will always contain a majority, which means that if this data-center is unavailable, Zookeeper is unavailable, and Kafka is unavailable. With 3 data-centers, you can easily allocate nodes so no single data-center has a majority. So if one data-center is unavailable, a majority of nodes exist in the other two data-centers and the Zookeeper cluster will remain available. Therefore so will the Kafka cluster.

There is a theoretical possibility to run Zookeeper and Kafka in two data-centers, using Zookeeper groups configuration that allows for manual failover between two data-centers. In practice, this setup is still uncommon.

Apache Kafka's MirrorMaker

Apache Kafka contains a simple tool for mirroring data between two data-centers. It is called MirrorMaker and at its core, it is a collection of consumers (called "streams" in MirrorMaker documentation, for historical reasons), all part of the same consumer group and reading data from the set of topics you chose to replicate. Each MirrorMaker process has a single producer. The workflow is pretty simple: MirrorMaker runs a thread for each consumer. Each consumer consumes events from the topics and partitions it was assigned on the source cluster and uses the shared producer to send those events to the target cluster. Every 60 seconds (by default), the consumers will flush the producer, to make sure all events arrived to the target Kafka and then commit the offsets for those events for the source Kafka.



MirrorMaker sounds very simple, but because we are trying to be very efficient and get very close to exactly once delivery, it turned out to be tricky to implement correctly. By release 0.10.0.0 of Apache Kafka, MirrorMaker was rewritten 4 times. Additional writes may happen in the future as well. The description here and the details in the following sections apply to MirrorMaker as it existed from release 0.9.0.0 to release 0.10.2.0.

How to configure

MirrorMaker is highly configurable. First, it uses a producer and consumers, so every configuration property of producers and consumers can be used when configuring MirrorMaker. In addition, MirrorMaker itself has a sizable list of configuration options, sometimes with complex dependencies between them. We will show few usage examples here and highlight few important configuration options, but exhaustive documentation of MirrorMaker is outside our scope.

With that in mind, lets take a look at a MirrorMaker example:

```
bin/kafka-mirror-maker --consumer.config etc/kafka/consumer.properties --producer.config etc/kafka/producer.properties
```

Lets look at MirrorMaker's basic command line arguments one by one:

- `consumer.config` - this is the configuration for all the consumers that will be fetching data from the source cluster. They all share one configuration file, which means you can only have one source cluster and one group.id. So all the consumers will be part of the same consumer group - which is exactly what we want. The mandatory configurations in the file are the `bootstrap.servers` (for

the source cluster) and the `group.id`. But you can use any additional configuration you want for the consumers. The one configuration you don't want to touch is `auto.commit.enable=false`. MirrorMaker depends on its ability to commit its own offsets after they safely arrived to the target Kafka cluster. Changing this setting can result in data loss. A configuration you do want to change is `auto.offset.reset`. This defaults to `latest`, which means MirrorMaker will only mirror events that arrived to the source cluster after MirrorMaker started. If you want to mirror existing data as well, change this to `earliest`. We will discuss additional configuration properties in the tuning section.

- `producer.config` - the configuration for the producer used by MirrorMaker to write to the target cluster. The only mandatory configuration is `bootstrap.servers` (for the target cluster). We will discuss additional configuration properties in the tuning section.
- `new.consumer` - MirrorMaker can use the only 0.8 consumer or the new 0.9 consumer. We recommend the 0.9 consumer as more stable at this point.
- `num.streams` - as we explained above, each stream is another consumer reading from the source cluster. Remember that all consumers in the same MirrorMaker process share the same producers. It will take multiple streams to saturate a producer. If you need additional throughput after this point, you'll need another MirrorMaker process.
- `whitelist` - a regular expression for the topic names that will be mirrored. All topic names that match the regular expression will be mirrored. In this example we chose to replicate every topic, but it is often a good practice to use something like "prod.*" and avoid replicating test topics. Or in active-active architecture, MirrorMaker replicating from NYC data-center to SF data-center will configure `whitelist="NYC.\*"` and avoid replicating back topics that originated in SF.

Deploying MirrorMaker in production

In the example above, we ran MirrorMaker as a command-line utility. Usually when running MirrorMaker in production environment, you will want to run MirrorMaker as a service - running in the background, with `nohup` and redirecting its console output to a log file. Technically, the tool has `-daemon` as a command-line option that should do all of the above for you, but in practice, this hasn't worked as expected in recent releases.

Most companies that use MirrorMaker have their own startup scripts that also include the configuration parameters they use. Production deployment systems like Ansible, Puppet, Chef and Salt are often used to automate the deployment and manage the many configuration options and files.

A more advanced deployment option that is becoming very popular is to run MirrorMaker inside a Docker container. MirrorMaker is completely stateless and doesn't require any disk storage (all the data and state is stored in Kafka itself). Wrapping MirrorMaker in Docker also allows running multiple instances on a single machine. Since a single MirrorMaker instance is limited to the throughput of a single producer, this is often important to launch multiple instances of MirrorMaker and Docker makes it much easier. It also makes it easier to scale up and down - spin additional container when more throughput is needed at peak time and spin them down when there is less traffic. If you are running MirrorMaker in a cloud environment, you can even spin up additional servers to run the containers on - all based on throughput and demand.

If at all possible, run MirrorMaker at the destination data-center. So if you are sending data from NYC to SF, MirrorMaker should run in SF and consume data across the US from NYC. The reason for this is that long distance networks can be a bit less reliable than typical inside a data-center. If there is a network partition and you lose connectivity between the data-centers, having a consumer that is unable to connect to a cluster is much safer than a producer that can't connect. If the consumer can't connect, it simply won't be able to read events - but the events are still stored in the source Kafka cluster and can remain there for a long time. There is no risk to losing events. On the other hand, if the events were already consumed and MirrorMaker can't produce them due to network partition, there is always a risk that these events will accidentally get lost by MirrorMaker. So remote consuming is safer than remote producing.

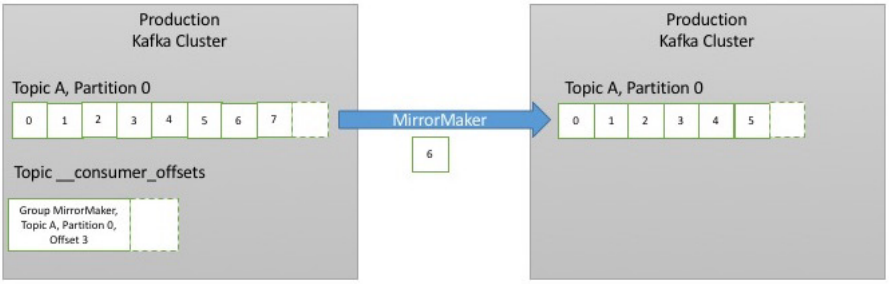
When do you have to consume locally and produce remotely? If you need to encrypt the data while it is transferred between the data-centers but you don't need to encrypt the data inside the data-center. Consumers take a significant performance hit from connecting to Kafka with SSL encryption. Much more so than producers. And this performance hit also affects the Kafka brokers themselves. If your cross data-center traffic requires encryption, you are better off placing MirrorMaker at the source data-center, have it consume

un-encrypted data locally and then produce it to the remote data-center through an SSL encrypted connection. This way the SSL connection is with the producer which is far less noticeable. If you use this configuration, make sure MirrorMaker is configured to never commit offsets before the target brokers acknowledged them from sufficient replicas and to fail instantly if retries are exhausted or the producer buffer is completely full.

If having very low lag between the source and target clusters is important, you will probably want to run at least two MirrorMaker instances on two different servers and have both use the same consumer group. This means that if one server is stopped for whatever reason, the MirrorMaker instance can continue mirroring the data.

When deploying MirrorMaker in production, it is important to remember to monitor it. Here are some monitoring we recommend for MirrorMaker:

- **Lag monitoring**: You will definitely want to know if the destination cluster is falling behind the source. The lag is the difference in offsets between the latest message in the source Kafka and the latest message in the destination.



In this diagram, the last offset in the source cluster is 7 and the last offset in the destination is 5 - meaning there is a lag of 2 messages.

There are two ways to track this lag, neither is perfect:

- Check the latest offset committed by MirrorMaker to the source Kafka cluster. You can use kafka-consumer-groups tool to check for each partition MirrorMaker is reading - what is the latest offset in the partition, what is the latest MirrorMaker committed and what is the. This indicator is not 100% accurate because MirrorMaker doesn't commit offsets all the time, it commits offsets every minute by default. So you will see the lag grow for a minute and then suddenly drop. In the diagram, the real lag is 2, but kafka-consumer-groups tool will report a lag of 4 because MirrorMaker haven't committed offsets for more recent messages yet. LinkedIn's burrow monitors the same information but has a more sophisticated method to determine whether the lag represents a real problem, so you won't get false alerts.

- Check the latest offset read by MirrorMaker (even if it isn't committed). The consumers embedded in MirrorMaker publish key metrics in JMX. One of them is the consumer maximum lag (over all the partitions it is consuming). This lag is also not 100% accurate, because it is updated based on what the consumer read, but doesn't take into account whether the producer managed to send those messages to the destination Kafka cluster and whether they were acknowledged successfully. In this example, the MirrorMaker consumer will report a lag of 1 message rather than 2, because it already read message 6 - even though the message wasn't produced to the destination yet.

  Note that if MirrorMaker skips or drops messages, neither method will detect an issue because they just tracks the latest offset. Confluent's Control Center monitor message counts and checksums and closes this monitoring gap.

- Metrics monitoring: MirrorMaker contains and producer and a consumer. Both have many available metrics and we recommend collecting and tracking them. Kafka documentation (http://kafka.apache.org/documentation.html#selector_monitoring) lists all the available metrics. Here are few metrics that proved useful in tuning MirrorMaker performance:

- Consumer: fetch-size-avg, fetch-size-max, fetch-rate, fetch-throttle-time-avg and fetch-throttle-time-max
- Producer: batch-size-avg, batch-size-max, requests-in-flight and record-retry-rate
- Both: io-ratio and io-wait-ratio
- Canary: If you monitor everything else, a canary isn't strictly necessary, but we like to add it in for multiple layers of monitoring. Have a process that every minute sends an event to a special topic in the source cluster and tries to read this event from the destination cluster. Alert if the event takes more than an acceptable amount of time to arrive. This can mean MirrorMaker is lagging or that it isn't around at all.

Tuning MirrorMaker

Sizing of the MirrorMaker cluster depends on the throughput you need and the lag you can tolerate. If you can't tolerate any lag, you have to size MirrorMaker with enough capacity to keep up with your top throughput. If you can tolerate some lag, you can size MirrorMaker to be 75%-80% utilized 95-99% of the time. Then, expect some lag to develop when you are at peak throughput and because MirrorMaker has spare capacity most of the time, it will catch up once the peak is over.

Then you want to measure the throughput you get from MirrorMaker with different number of consumer threads - configured with `num.streams` parameter. We can give you some ballpark numbers (LinkedIn gets 6MB/s with 8 consumer threads and 12MB/s with 16), but since this depends a lot of your hardware, data-center or cloud provider, you will want to run your own tests. Kafka ships with kafka-performance-producer tool. Use it to generate load on a source cluster and then connect MirrorMaker and start mirroring this load. Test MirrorMaker with 1, 2, 4, 8, 16, 24 and 32 consumer threads. Watch where performance tapers off and set `num.streams` just below this point. If you are consuming or producing compressed events (recommended, since bandwidth is the main bottleneck for cross data-center mirroring), MirrorMaker will have to decompress and recompress the events. This uses a lot of CPU, so keep an eye on CPU utilization as you increase the number of threads. Using this process, you will find the maximum throughput you can get with a single MirrorMaker instance. If it is not enough, you will want to experiment with additional instances and after that, additional servers.

In addition, you may want to separate sensitive topics, those that absolutely require low latency and where the mirror must be as close to the source as possible, to a separate MirrorMaker cluster with its own consumer group. This will prevent a bloated topic or an out of control producer from slowing down your most sensitive data pipeline.

This is pretty much all the tuning you can do to MirrorMaker itself. However, you can still increase the throughput of each consumer thread and each MirrorMaker instance.

If you are running MirrorMaker across data-centers, you want to optimize the network configuration in Linux:

- Increase TCP buffer size (`net.core.rmem_default`, `net.core.rmem_max`, `net.core.wmem_default`, `net.core.wmem_max`, `net.core.optmem_max`)
- Enable automatic window scaling (sysctl –w net.ipv4.tcp_window_scaling=1 or add net.ipv4.tcp_window_scaling=1 to /etc/sysctl.conf)
- Reduce the TCP slow start time (set /proc/sys/net/ipv4/tcp_slow_start_after_idle to 0)

Note that tuning the Linux network is a large and complex topic. To understand more about these parameters and others, we recommend reading a network tuning guide. For example, *Performance tuning for Linux servers* by Sandra K. Johnson, et al. (IBM Press).

In addition, you may want to tune the producers and consumers that are running in MirrorMaker. First, you will want decide whether the producer or the consumer is the bottleneck - is the producer waiting for the consumer to bring more data or the other way around? One way to decide is to look at the producer and consumer metrics you are monitoring - if one process is idle while the other is fully utilized, you know which one requires tuning. Another method is to do several thread dumps (using jstack) and see if the MirrorMaker threads are spending most of the time in poll or in send - more time spent polling usually means the consumer is the bottleneck, while more time spent sending shifts suspicion to the producer.

If you need to tune the producer, the following configuration settings can be useful:

- `max.in.flight.requests.per.connection` - by default, MirrorMaker only allows one in-flight request. This means every request that was sent by the producer has to be acknowledged by the destination cluster before the next message will be sent. This can limit throughput, especially if there is significant latency before the brokers acknowledge the messages. The reason MirrorMaker limits the number of in-flight requests is because this is the only way to guarantee that Kafka will preserve message order in the event that some messages will require multiple retries before they are successfully acknowledged. If message order is not critical for your use-case, increasing `max.in.flight.requests.per.connection` can significantly increase your throughput.
- `linger.ms` and `batch.size` - if your monitoring shows that the producer consistently sends partially empty batches (i.e. `batch-size-avg` and `batch-size-max` metrics are lower than configured `batch.size`), you can increase throughput by introducing a bit of latency. Increase `latency.ms` and the producer will wait for few milliseconds for the batches to fill up before sending them. If you are sending full batches and have memory to spare, you can increase `batch.size` and send larger batches.

The following consumer configurations can increase throughput for the consumer:

- The partition assignment strategy in MirrorMaker (i.e. the algorithm that decides which consumer is assigned which partitions) defaults to `range`. There are benefits to `range` strategy, which is why it is the normal default for consumers - but it can lead to uneven assignment of partitions to consumers. For MirrorMaker, it is usually better to change the strategy to `round robin`, especially when mirroring large number of topics and partitions. You set this by adding `partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor` to the consumer properties file.
- `fetch.max.bytes` - if the metrics you are collecting show that `fetch-size-avg` and `fetch-size-max` are close to the `fetch.max.bytes` configuration, it means that the consumer is reading as much data from the broker as it is allowed. If you have available memory, try increasing `fetch.max.bytes` to allow the consumer to read more data in each request.
- `fetch.min.bytes` and `fetch.max.wait` - if you see in the consumer metrics that `fetch-rate` is high, it means that the consumer is sending too many requests to the brokers and not receiving enough data in each request. Try increasing both `fetch.min.bytes` and `fetch.max.wait`, so the consumer will receive more data in each request and the broker will wait until enough data is available before responding to the consumer request.

Other Cross-Cluster Mirroring Solution

We looked in depth at MirrorMaker because this mirroring software arrives as part of Apache Kafka. However, MirrorMaker also have some limitations when used in practice. It is worth-while looking at some of the alternatives to MirrorMaker that are available and the ways they address MirrorMaker limitations and complexities:

Uber uReplicator

Uber ran MirrorMaker at very large scale, and as the number of topics and partitions grew and the cluster throughput increased, they started running into the following problems:

- **Rebalancing delays**: MirrorMaker consumers are just consumers. Adding MirrorMaker threads, add MirrorMaker instances, bounce a MirrorMaker instance or even add new topics that match the regular expression used in the whitelist - all those activities cause the consumers to *rebalance*. As we've seen in Chapter 4, rebalancing stops all the consumers until new partitions can be assigned to each consumer. With very large number of topics and partitions, this can take a while. This is especially true when using the old consumers like Uber did. In some cases this caused 5-10 minutes of inactivity, which causes mirroring to fall behind and accumulate large backlog of events to mirror, which can take a long time to recover from. This caused very high latency for consumers reading events from the destination cluster.
- **Difficulty adding topics**: Using a regular expression as the topic whitelist means that MirrorMaker will rebalance every time someone adds a matching topic to the source cluster. We've already seen that rebalances were particularly painful for Uber. To avoid this, they decided to simply list every topic they need to mirror and avoid the surprise rebalances. But this does mean that they need to manually add new topics that they want to mirror to the whitelist on all MirrorMaker instances and bounce the instances - which

leads to rebalances. At least those rebalances happen on scheduled maintenance and not every time someone adds a topic, but it is still lots of maintenance. This also meant that if the maintenance wasn't done correctly and different instances had a different topic lists, MirrorMaker would start and endlessly rebalance as the consumers wouldn't be able to agree on the topics they subscribe to.

Given these issues, Uber decided to write their own MirrorMaker clone, called uReplicator, that will address those. They decided to use Apache Helix as a central (but highly available) controller that will manage the topic list and the partitions assigned to each uReplicator instance. Administrators uses a REST API to add new topics to the list in Helix and uReplicator is responsible for assigning partitions to the different consumers. To achieve this, Uber replaced the Kafka Consumers used in MirrorMaker with a Kafka Consumer they wrote themselves called Helix Consumer. This consumer takes its partition assignment from the Apache Helix controller rather than as a result of an agreement between the consumers (see Chapter 4 for details on how this is done in Kafka), as a result the Helix Consumer can avoid rebalances and instead listen to changes in the assigned partitions that arrive from Helix.

Uber wrote a blog post describing the architecture in more detail and showing the improvements they experienced. At this point, we are not aware of any company outside Uber that uses the uReplicator. Possibly because most companies don't operate at Uber's scale and don't run into the same issues, or perhaps because the dependency on Apache Helix introduces a completely new component to learn and manage, which adds complexity to the entire project.

Confluent's Replicator

At the same time Uber developed their uReplicator, Confluent independently developed their Replicator. Despite the similarities in names, the projects have almost nothing in common - they are different solutions to two different sets of MirrorMaker problems. Confluent's Replicator was developed to address issues their enterprise customers encountered when using MirrorMaker to manage their multi-cluster deployments.

- **Diverging cluster configuration**: While MirrorMaker keeps data in sync between source and destination, this is the only thing it keeps in sync. Topics can end up with different numbers of partitions, different replication factors, and different topic-level settings. Increasing topic retention from 1 week to 3 weeks on the source cluster and forgetting about the DR cluster can lead to a rather nasty surprise when you failover to the second cluster and discover few weeks of data are now missing. Trying to manually keep all these settings in sync is error prone and can lead downstream applications, or even replication itself, to fail if the systems fall out of sync.

- **Cluster management challenges**: We've already seen that MirrorMaker is typically deployed as a cluster of multiple instances. This means yet another cluster to figure out how to deploy, monitor and manage configuration for. With two configuration files and large number of parameters, configuration management for MirrorMaker itself can be a challenge. This increases if there are more than two clusters and one-direction replication. If you have 3 active-active clusters, you have 6 MirrorMaker clusters to deploy, monitor and configure, and each of those likely has at least 3 instances. With 5 active-active clusters, the number of MirrorMaker clusters increases to 20.

With the goal of minimizing administration overhead for busy enterprise IT departments, Confluent decided to implement Replicator as a Source Connector for the Kafka Connect framework, that just happened to read data from another Kafka cluster, rather than from a database. If you'll recall Kafka Connect architecture from Chapter 7, you'll remember that each connector divides the work between a configurable number of tasks. In case of Replicator, each task is a consumer and a producer pair. Connect framework assigns those tasks to different Connect Worker nodes as needed - so you may have multiple tasks on one server or have the tasks spread out to multiple servers. This replaces the manual work of figuring our how many MirrorMaker streams should run per instance and how many instances per machine. Connect also has a REST API to centrally manage the configuration for the connectors and tasks. If we assume that most Kafka deployments include Kafka Connect for other reasons (sending database change events into Kafka is a very popular use-case), then by running Replicator inside Connect, we can cut down on the number of clusters we need to manage. The other significant improvement is that the Replicator connector, in addition to replicating data from a list of Kafka topics it also replicates the configuration for those topics from Zookeeper.

Summary

We started the chapter by describing the reasons you may need to manage more than a single Kafka cluster and then proceeded to describe several common multi-cluster architectures, ranging from the simple to the very complex. We went into the details of implementing failover architecture for Kafka and compared the different options currently available. Then we proceeded to discuss

the available tools. Starting with Apache Kafka's MirrorMaker, we went into many details of using it in production. We finished by reviewing two alternative options that solve some of the issues you may encounter with MirrorMaker.

Whichever architecture and tools you end up using - remember that multi-cluster configuration and mirroring pipelines should be monitored and tested just like everything else you take into production. Because multi-cluster management in Kafka can be easier than it is with relational databases, some organizations treat it as an afterthought and neglect to apply proper design, planning, testing, deployment automation, monitoring and maintenance. By taking multi-cluster management seriously, preferably as part of a holistic disaster plan or geo-diversity plan for the entire organization that involves mutliple applications and data-stores, you will greatly increase the chances of successfully managing multiple Kafka clusters.

PREV
7. Building Data Pipelines

NEXT
9. Administering Kafka