

Disaster Recovery for Multi-Datacenter Apache Kafka[®] Deployments

Design, Configuration, Failover, Failback

Yeva Byzek, © 2020 Confluent, Inc.

Table of Contents

Introduction	1
Designs	2
Single Datacenter	2
Multi-Datacenter with Confluent Replicator	3
Centralized Schema Management	8
Key Features	10
Preventing Cyclic Repetition of Messages	10
Timestamp Preservation	12
Consumer Offset Translation	13
Multi-Datacenter Bringup	17
Confluent Replicator	17
Java Consumer Applications	22
Confluent Schema Registry	22
When Disaster Strikes	25
Datacenter Failure	25
Failing Over Client Applications	26
Reset Consumer Offsets	26
Schema Registration	27
Failback	28
Restoring the Kafka Cluster	28
Data Synchronization	30
Client Application Restart	31
Demo	31
Summary	32
Appendix	33

Topic Naming Strategies to Prevent Cyclic Repetition 33

Manually Reset Consumer Offsets 34

Data Synchronization in Active-Passive Design 36

Introduction

Datacenter downtime and data loss can result in businesses losing a vast amount of revenue or entirely halting operations. To minimize the downtime and data loss resulting from a disaster, enterprises can create business continuity plans and disaster recovery strategies.

A disaster recovery plan often requires multi-datacenter Apache Kafka® deployments where datacenters are geographically dispersed. If disaster strikes—catastrophic hardware failure, software failure, power outage, denial of service attack or any other event that causes one datacenter to completely fail—Kafka continues running in another datacenter until service is restored. A multi-datacenter solution with a disaster recover plan ensures that your event streaming applications continue to run even if one datacenter fails.

This white paper provides a general overview of a multi-datacenter solution based on the capabilities of Confluent Platform, the leading distribution of Apache Kafka®. Confluent Platform provides the building blocks for:

- Multi-datacenter designs
- Centralized schema management
- Prevention of cyclic repetition of messages
- Automatic consumer offset translation

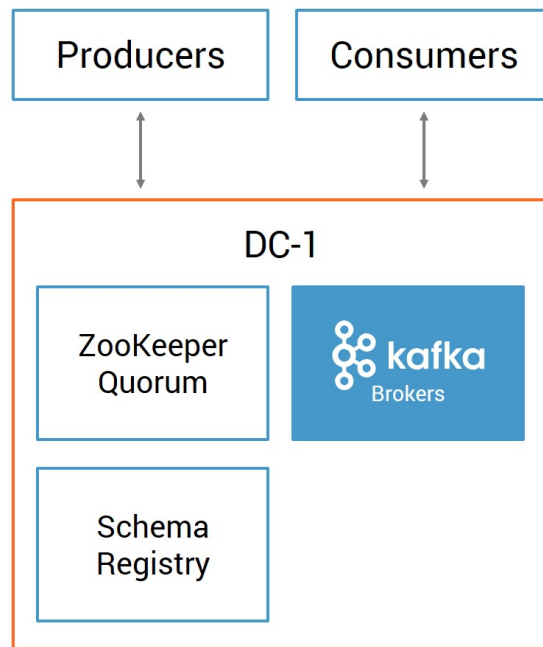
This white paper will use these building blocks to walk through how to configure and bring up a multi-datacenter Kafka deployment, what to do if one datacenter fails and how to failback when the datacenter recovers.

You may be considering an active-passive design (one-way data replication between Kafka clusters), active-active design (two-way data replication between Kafka clusters), client applications that read from just their local cluster or both local and remote clusters, service discovery mechanisms to enable automated failovers, geo locality offerings, etc. Your architecture will vary depending on your business requirements, but you can apply the building blocks from this white paper to strengthen your disaster recovery plan.

Designs

Single Datacenter

First, let us briefly review how a Kafka deployment in a single datacenter can provide message durability. The high-level reference architecture for a single datacenter is shown below.



Within a single datacenter, Kafka intra-cluster data replication is the basic means for achieving message durability. Producers write data to and consumers read data from topic partition leaders. Synchronous data replication from leaders to followers ensures that messages are copied to more than one broker. Kafka producers can set the **acks** configuration parameter to control when a write is considered successful.

Data replication with producer setting **acks=all** provides the strongest available guarantees, because it ensures other brokers in the cluster acknowledge receiving the data before the leader broker responds to the producer. If a leader broker fails, the Kafka cluster recovers when a follower broker is elected leader and thus client applications can continue to write and read messages through the new leader. [KIP-101](#), introduced in Kafka 0.11, hardens the intra-cluster replication protocol and addresses some corner cases to improve fault tolerance.

Additionally, client applications can connect to the Kafka cluster through any set of brokers, called

bootstrap brokers. If a client application is using a certain broker for connectivity to the cluster, and that broker fails, another bootstrap broker can provide connectivity to the cluster.

The ZooKeeper quorum, which provides reliable distributed synchronization, is recommended to be at least three nodes to maintain high availability even if a ZooKeeper node fails.

Finally [Confluent Schema Registry](#), which provides a globally accessible versioned history of all schemas to serve to client applications, can be run with multiple instances. One Schema Registry instance will be elected as a leader that is responsible for registering new schemas, and the remaining will be designated as followers. Followers instances can handle read requests but will forward all write requests to the leader. In the event of a leader failure, one of the followers will be elected as the new leader Schema Registry instance.

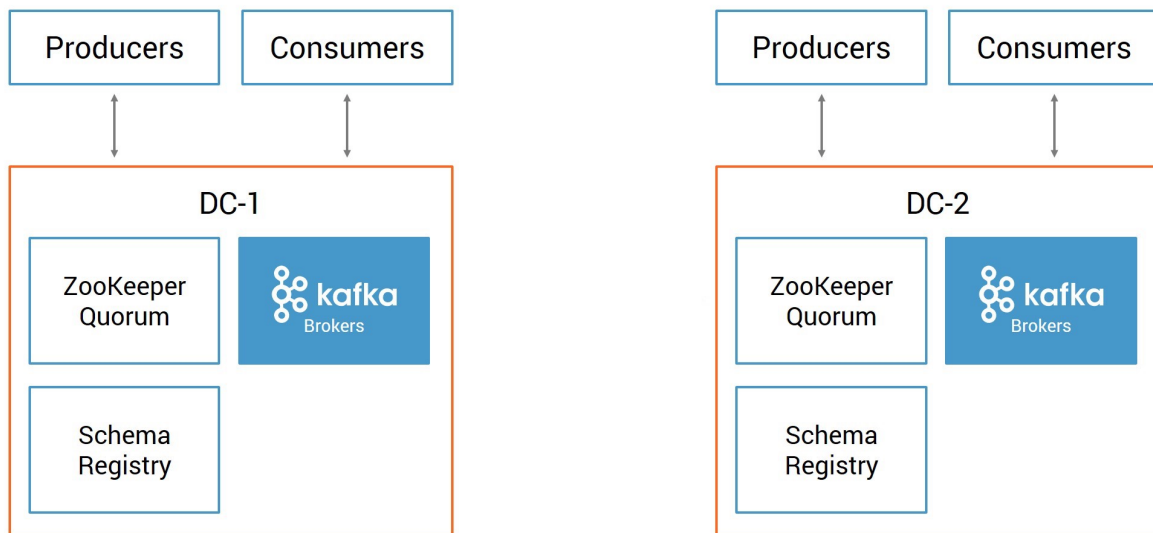
Taken together, the single datacenter design provides robust protection against broker failure. For more information on how to configure and monitor Kafka for message durability and high availability, see the [Optimizing Your Apache Kafka® Deployment](#) white paper.

Multi-Datacenter with Confluent Replicator

In a multi-datacenter design, instead of one datacenter with one Apache Kafka® deployment, there are two or more datacenters with Kafka deployments. Although multi-datacenter Kafka deployments have a variety of [use cases](#), the focus of this white paper is on disaster recovery for two datacenters.

Consider two Kafka deployments, each in a different datacenter in separate geographic locations. One or both of the deployments can be on prem, in [Confluent Cloud](#) or part of a bridge-to-cloud solution. Each datacenter has its own:

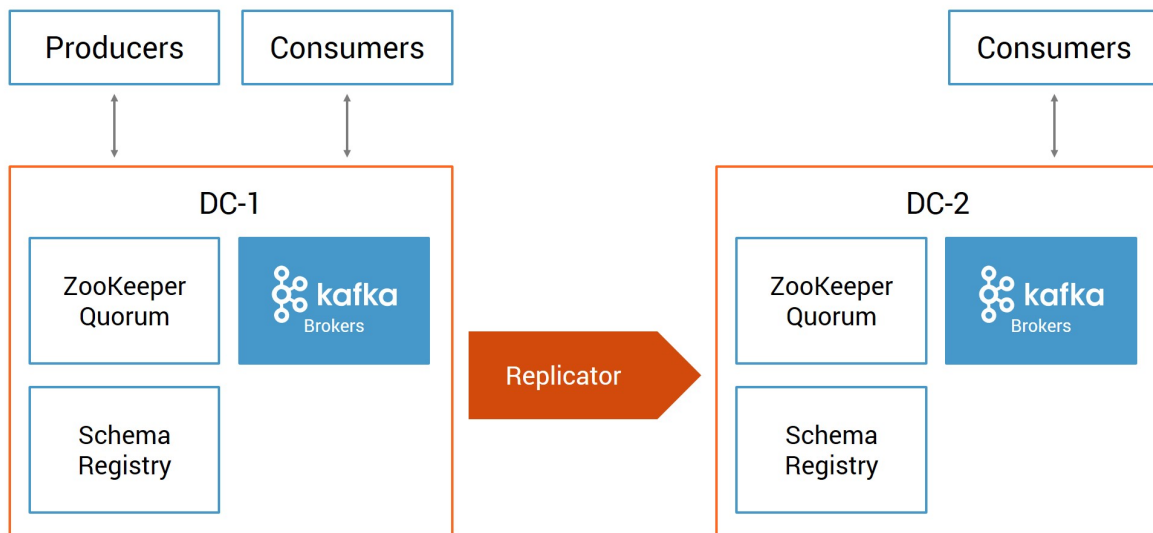
- Kafka cluster, such that the brokers in the local datacenter are grouped together independently from the brokers in the remote datacenter
- ZooKeeper quorum that only serves the local cluster
- Client applications that only connect to the local cluster



In a multi-datacenter design, the goal is to synchronize data across the sites. Confluent Replicator, an advanced feature of Confluent Platform, is the key to this design. Replicator reads data from one cluster and then writes those messages to another cluster, providing a centralized configuration of cross-datacenter replication. New topics are automatically detected and replicated to the destination cluster. As throughput increases, Replicator automatically scales to accommodate the increased load.

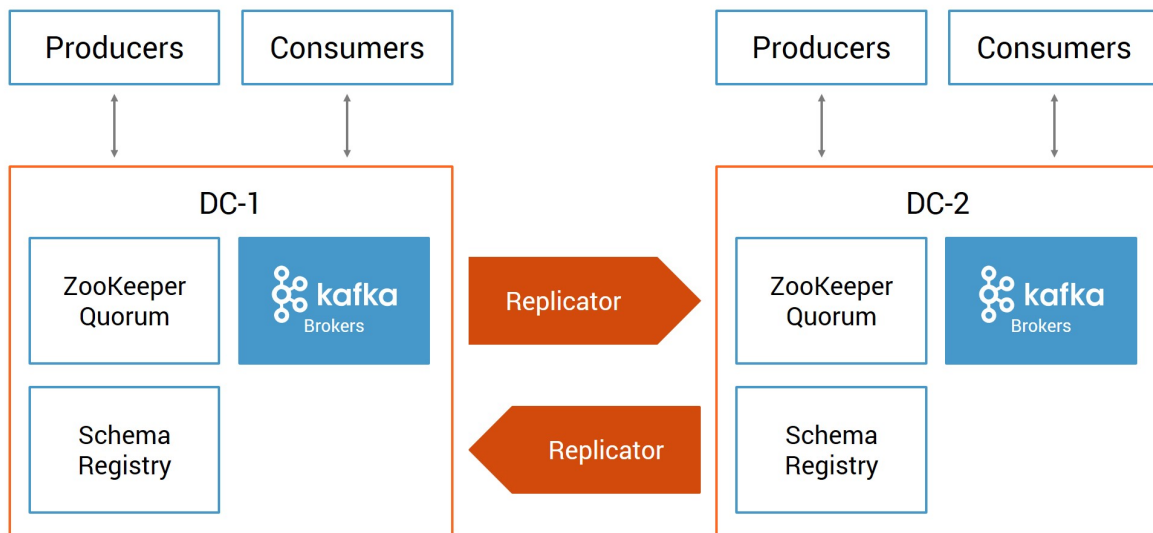
While Replicator is applicable in various use cases, the focus here is disaster recovery for two Kafka clusters. In the event of a partial or complete disaster in one datacenter, applications can failover to a second datacenter.

In the active-passive design shown below, Replicator runs in one direction, copying Kafka data and configurations from the active DC-1 to the passive DC-2.



Producers write data to just the active cluster. Depending on the overall architecture, consumers can read data from just the active cluster, leaving the passive cluster purely for disaster recovery. Consumers can also read from both clusters to create a local cache for geo-locality. In steady state, when both datacenters are running properly, DC-1 is the active cluster and therefore all producers only write to DC-1. This is a viable strategy but an inefficient use of the resources in the passive cluster. DC-1 consumers can read data that originated locally in DC-1, and DC-2 consumers can read data that was replicated from DC-1. If a disaster event causes DC-1 to fail, the business requirements determine how the client applications respond. Client applications can failover to DC-2. When DC-1 recovers, as part of the failback process, all of the latest state in DC-2 needs to be replicated back to the active cluster before processing can return to it.

In the active-active design shown below, one Replicator copies Kafka data and configurations from origin DC-1 to destination DC-2, and another Replicator copies Kafka data and configurations from origin DC-2 to destination DC-1.



Producers write data to both clusters: DC-1 producers write to topics in their local DC-1, and DC-2 producers write to topics in their local DC-2. Depending on how the client applications are written, DC-1 consumers can read data that was produced in DC-1, while reading data that was produced in DC-2 and then replicated to DC-1, and vice versa. Consumers are able to subscribe to multiple topics either by explicitly naming topics or using wildcards. Consequently, the resources in both datacenters are well utilized. In the event of a disaster event that causes DC-1 to fail, existing DC-2 producers and consumers can continue operating, so it essentially remains unaffected. When DC-1 recovers, as part of the failback process, client applications can point back to the active cluster.

Data and Metadata Replication

Data replication within a single Kafka cluster is synchronous, meaning that a producer gets commit acknowledgment after data is replicated to local brokers. Meanwhile, Replicator copies messages between datacenters asynchronously. This means that the client application that produces the messages to the local cluster does not wait for commit acknowledgment from the remote cluster, and Replicator copies the data after it has been committed locally. This asynchronous replication generally minimizes latency by making data available to consumers sooner. Another benefit of asynchronous replication is you are not creating an interdependence between two distinct datacenters. Producer sends to the local cluster will succeed even if connectivity between the clusters failed or you needed to do maintenance on a remote datacenter.

Replicator copies not just topic data but also metadata. As topic metadata or partition count changes in the origin cluster, Replicator makes the same the changes in the destination cluster. To maintain consistency in the Kafka topic administrative preferences between the multiple clusters, topic metadata must be the same in the origin cluster and the destination cluster. This is done automatically by Replicator. It creates initial topic configurations to bootstrap topics, and it synchronizes topic metadata

between clusters if it changes. For example, if you update a topic's configuration property in DC-1 (e.g., `segment.bytes`, which controls the segment file size for the topic's log files), Replicator makes the corresponding configuration update to the replicated topic in DC-2.

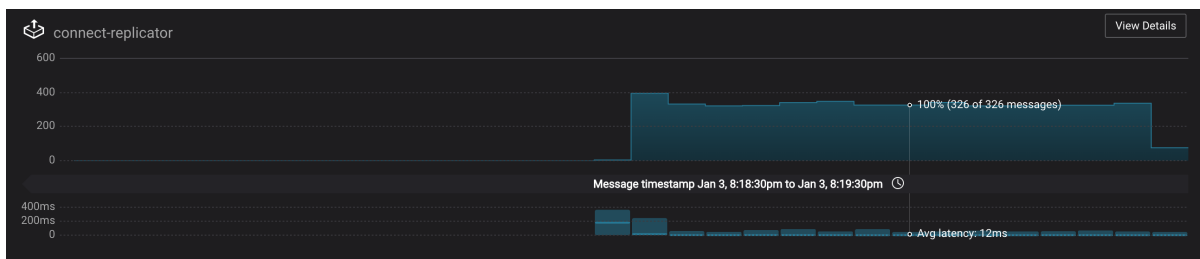
MirrorMaker

You may have heard of a legacy standalone tool called MirrorMaker that copies data between two Kafka clusters. However, MirrorMaker has a long list of shortcomings that make it challenging to build and maintain multi-datacenter deployments, including:

- Cumbersome API for filtering topics to be replicated
- Topics created in the destination cluster may have a configuration that does not match the topics in the origin cluster
- Topic configuration changes in the origin cluster are not detected and propagated to the destination cluster
- Lack of built-in capability to reconfigure the names of topics to prevent cyclic repetition of data
- Inability to scale replication processes as Kafka traffic increases with a single configuration
- Monitoring end-to-end latency across the clusters

Confluent Replicator addresses these shortcomings while providing reliable data replication. Replicator provides better topic data and metadata synchronization across multiple datacenters. Because Replicator integrates with Kafka Connect, it provides superior availability and scalability.

Additionally, Confluent Control Center can manage Replicator and monitor its performance, throughput and latency. To monitor Replicator performance with Control Center, you will need to configure [Confluent Monitoring Interceptors](#).



For a more in-depth comparison between Confluent Replicator and MirrorMaker, read the [documentation](#).

Centralized Schema Management

Schemas need to be available globally so that Kafka messages can be produced and consumed across all clusters. Schema Registry provides central schema management and is designed to be distributed. Multiple Schema Registry instances deployed across datacenters provide resiliency and high availability, and any instance can communicate schemas and schema IDs to Kafka clients. There is a commit log for the database with all the schema information, which gets written to a Kafka topic. In a single-leader architecture, only the leader Schema Registry instance writes to that Kafka topic. The followers forward new schema registration requests to the leader.

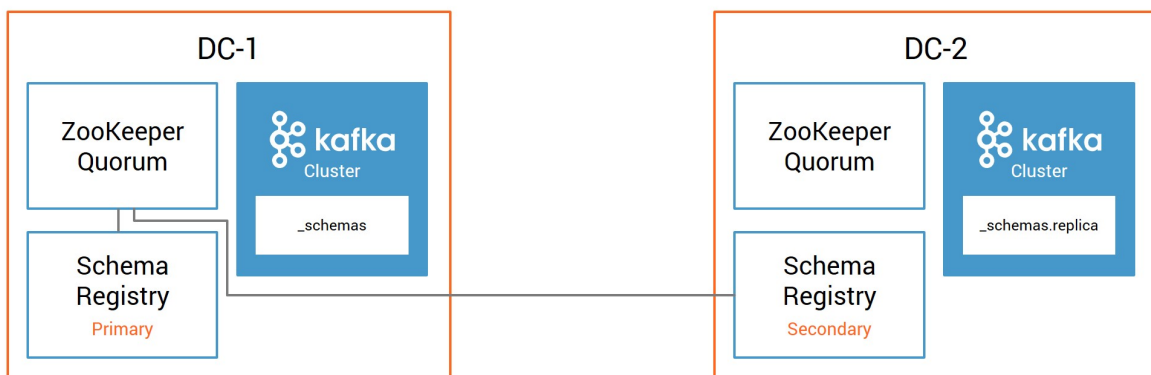
In an multi-datacenter design, all of your Schema Registry instances in both datacenters should have:

Access to the same schema IDs:

Schema information must be available to both datacenters, because a message produced in DC-1 may need to be consumed in DC-2. A producer in DC-1 will register the schema with Schema Registry and insert the schema ID into the message. Then a consumer in DC-2 or any other datacenter can use the message's schema ID to look up the schema from the Schema Registry. Both the producer and consumer should be using the same source of truth (i.e., same Schema Registry cluster) for schema information.

Coordinated elections for leader Schema Registry instance:

Regardless of whether your multi-datacenter design is active-active or active-passive, designate one Kafka cluster as primary for Schema Registry. That cluster coordinates leader election among the Schema Registry instances. Since Confluent Platform version 4.0, either the Kafka group protocol or ZooKeeper can coordinate the leader election. Use the Kafka group protocol if connecting to Confluent Cloud or if access to ZooKeeper is otherwise unavailable. In versions prior to 4.0, only ZooKeeper can coordinate election. The remainder of this white paper shows the Kafka group protocol to coordinate leader election.



Replicator copies the Kafka topic that backs up the schema information from the primary cluster to the other cluster, as shown in the diagram above. However, all follower Schema Registry instances in both datacenters subscribe directly to the schema topic in the designated primary cluster. Always ensure that the primary Kafka cluster and leader-eligible Schema Registry instances are globally accessible by all instances in both datacenters.

Key Features

Preventing Cyclic Repetition of Messages

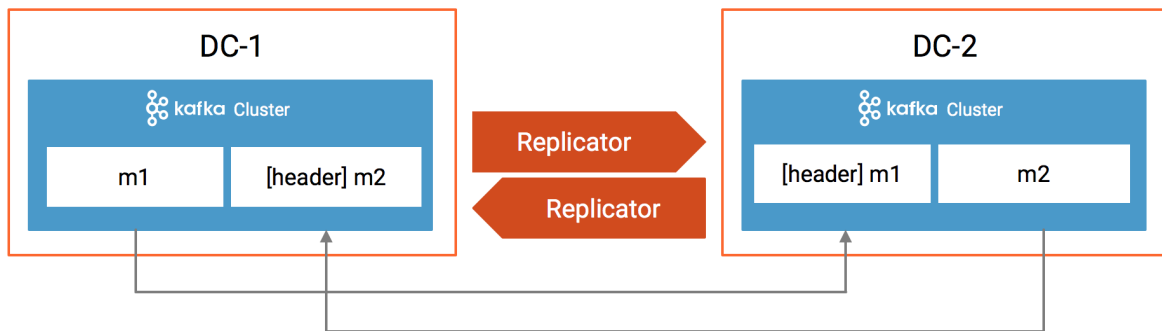
In active-active multi-datacenter designs where topics are replicated bidirectionally across Kafka clusters, it is important to prevent cyclic repetition of messages. What you don't want to happen is an infinite loop copying a topic's messages from DC-1 to DC-2, and then copying those same messages in DC-2 back to DC-1, and then again from DC-1 to DC-2.

You can use a new feature introduced in Confluent Replicator version 5.0.1 that prevents this cyclic replication of messages without the constraint of unique topic names. If this feature is enabled, Replicator tracks provenance information—cluster and topic origin—on a per-message basis. Replicator uses Kafka headers, a new capability provided by [KIP-82](#), to track provenance information and ensure that messages are not copied to the destination cluster if they originated there. Kafka headers are supported on brokers running Kafka version 0.11 or higher. The broker configuration parameter to set is called `log.message.format.version`, and its default is already the desired setting `2.0`, so leave it as is.

To enable this Replicator feature, configure `provenance.header.enable=true`. Replicator then puts the provenance information in the message header after replication. The provenance information includes the following:

- ID of the origin cluster where this message was first produced
- Name of the topic to which this message was first produced
- Timestamp when Replicator first copied the record

By default, Replicator will not replicate a message to a destination cluster if the cluster ID of the destination cluster matches the origin cluster ID in the provenance header, **and** the destination topic name matches the origin topic name in the provenance header. Consider the diagram below. For a given topic of the same name that exists in both the origin and destination cluster, message **m1** was originally produced to DC-1, and message **m2** was originally produced to DC-2.



For a Replicator instance copying messages from topic **topic1** in DC-1 to topic **topic1** in DC-2:

- **m1** is copied to DC-2 because the message header in DC-1 does not have any provenance information in its header
- **m2** is not copied to DC-2 because the message in DC-1 already indicates DC-2 and **topic1** as its origin

The inverse is true for Replicator instances copying messages from DC-2 to DC-1:

- **m1** is not copied to DC-1 because the message in DC-2 already indicates DC-1 and **topic1** as its origin
- **m2** is copied to DC-1 because the message header in DC-2 does not have any provenance information in its header

Consequently, applications in different datacenters can access topics with exactly the same name while Replicator automatically avoids cyclic repetition of messages.

Client applications should be designed to take into consideration the effect of having the same topic name span datacenters. Producers do not wait for commit acknowledgment from the remote cluster, and Replicator asynchronously copies the data between datacenters after it has been committed locally. If there are producers in each datacenter writing to topics of the same name, there is no "global ordering," which means there are no message ordering guarantees for data that originated from producers in different datacenters. Also, if there are consumer groups in each datacenter with the same group ID reading from topics of the same name, in steady state they will be reprocessing the same messages in each datacenter.

In some cases, you may not want to use the same topic name in each datacenter. This may be due to any of the following reasons:

- Replicator is running a version less than 5.0.1
- Kafka brokers are running a version prior to Kafka 0.11 that does not yet support message headers

- Kafka brokers are running Kafka version 0.11 or later but have less than the minimum required `log.message.format.version=2.0` for using headers
- Client applications are not designed to handle topics with the same name across datacenters

For these situations, please see the section in the Appendix [Topic Naming Strategies to Prevent Cyclic Repetition](#).

Timestamp Preservation

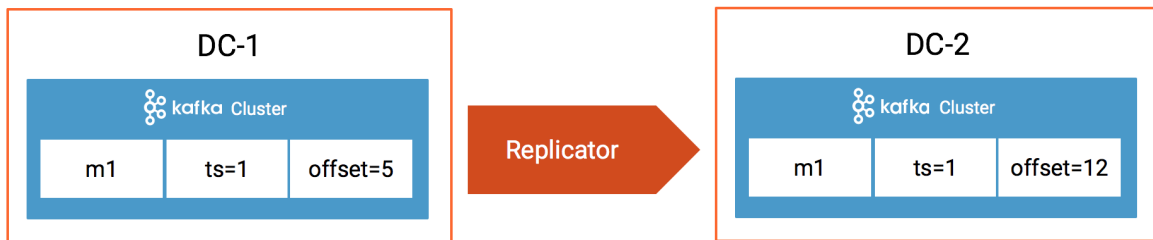
Within a given cluster, Kafka consumers track their consumption of messages. They can identify the offset of the next message to be read so that if they stop and have to continue later, they can start where they left off. These consumer offsets are stored in a Kafka topic called `__consumer_offsets`.

With multi-datacenter, if a consumer stops in one cluster due to a disaster event, it may need to restart in another cluster. Ideally the new consumers start where old consumers left off. It may be tempting to use Replicator to copy the consumer offsets topic. However, this will not work because there are situations where an offset for a message in one datacenter may not be the same as the offset of the same message in a different datacenter. These situations include the following:

- The origin cluster may have garbage collected some messages due to the retention policy or compaction, before they could be replicated. This may occur if Replicator had been started long after data was written into the origin cluster, in which case offsets will never match.
- There may be transient errors in copying data from the origin to destination Kafka cluster. In the event of these transient errors, Replicator will resend the data, though there is a chance for duplication. As a result of possible duplicate messages, the same offset value may no longer correspond to the same message.
- Replication of the data topic may lag in replication of the consumer offsets topic. This issue may arise because the consumer offsets topic is replicated independently from the data topics. So when a consumer is restarted in the new cluster, it may try to read from an offset for which topic data has not yet been replicated.

Because of these possible situations, applications cannot use consumer offsets as the basis for identifying the same message in two different clusters. In fact, the `__consumer_offsets` topic should not be replicated between datacenters. Instead, Replicator can use the message creation timestamp, which is the original producer's create time of the message. While copying the data, Replicator preserves the timestamps of the messages across clusters. [KIP-33](#) added a time-based index file to correlate timestamps to offsets.

The diagram below shows a message `m1` replicated from DC-1 to DC-2. The message offset differs between the clusters but the timestamp `t1` is preserved.



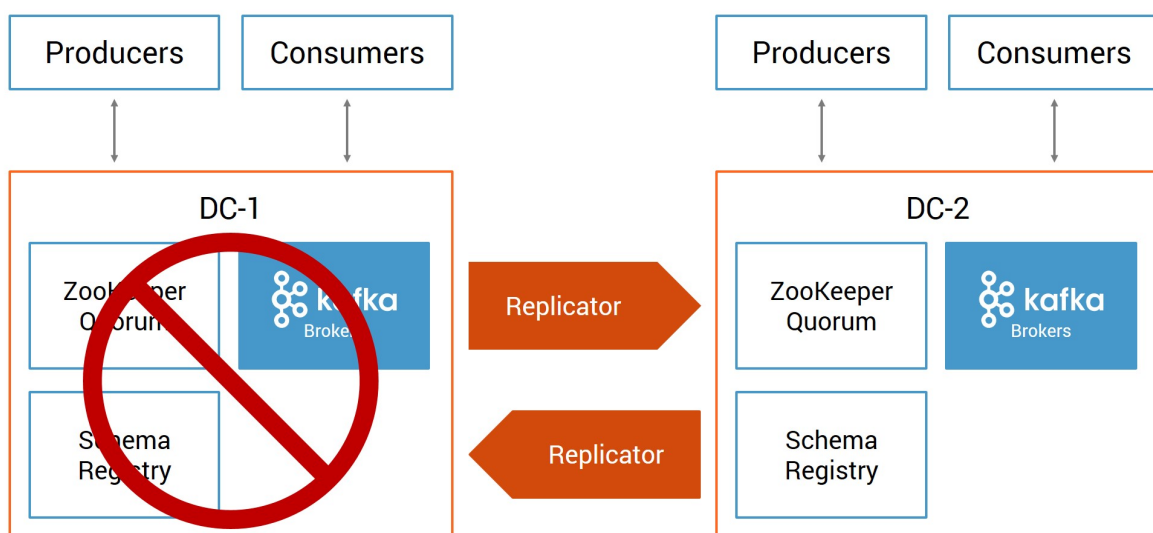
Kafka brokers should be configured to preserve message timestamps based on when producers create messages. The configuration parameter is called `log.message.timestamp.type` and its default is already the desired setting `CreateTime`, so leave it as is. For compressed messages, the create time of the wrapper message will be the create time of the first compressed message.

When the Kafka brokers preserve timestamps in their messages, consumers can reset message consumption to some previous point in time. In the next section, this white paper will discuss how consumers can use these preserved timestamps to reset their offsets.

Consumer Offset Translation

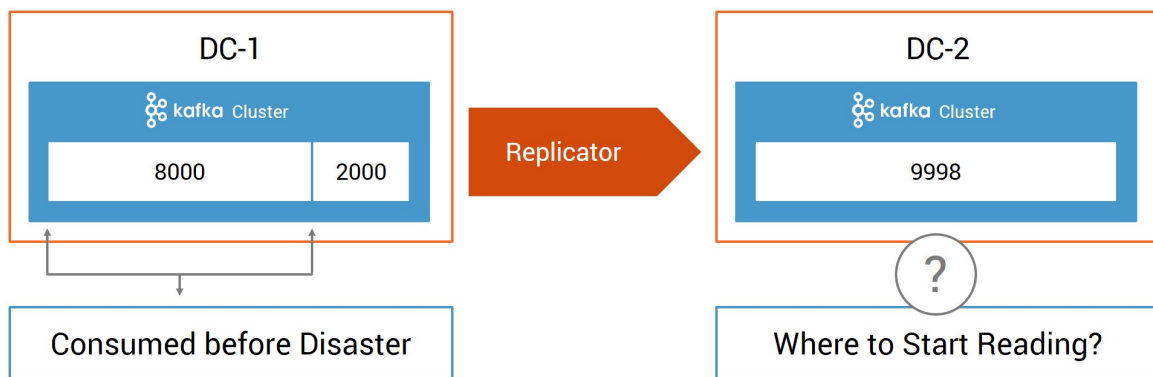
Where to Resume After Failover

If there is a disaster, consumers must restart to connect to a new datacenter. Whereas before the disaster they consumed messages from topics in one datacenter, now they consume messages from topics in the other datacenter.



How do the consumers that failed over to the other datacenter determine from where in the topic they should start reading? They could start at the earliest message of each topic, at the latest message or at the last known good time before the disaster event, which is close to the last time a message may have been consumed.

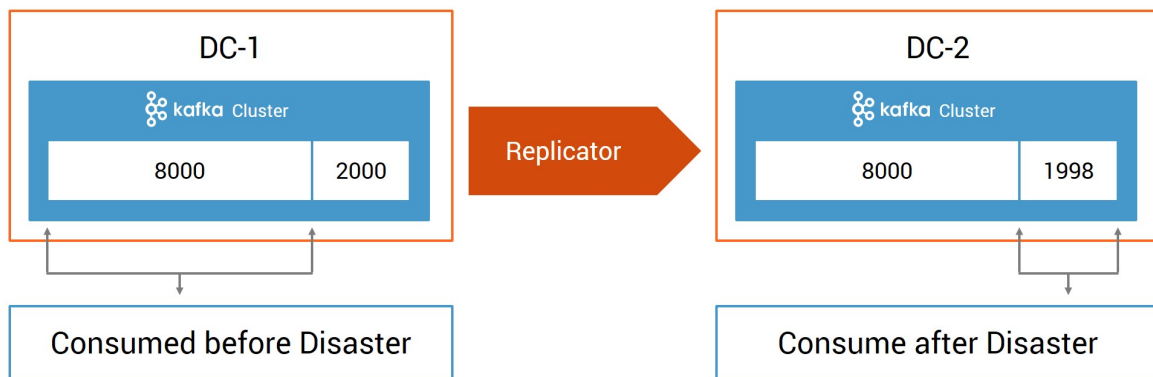
Consider a producer that may have written 10000 messages to a topic in DC-1. Replicator copies those messages to DC-2. Given replication lag, it copied 9998 messages when the disaster occurred. It could be that the original consumer in DC-1 read only 8000 of those messages before the disaster event, leaving 2000 messages unread. After failover to DC-2, the consumer sees a topic with 9998 messages, the last 1998 of which were unread. So where should the consumer start reading messages?



By default, when a consumer is created in DC-2, the configuration parameter `auto.offset.reset` could be set to either `latest` or `earliest`:

Value of <code>auto.offset.reset</code>	Effect on the consumer that failed over to DC-2
<code>latest</code> (default)	Starts consuming from the latest message, and thus loses the last 1998 messages
<code>earliest</code>	Starts consuming from the first message, and thus reprocesses the first 8000 messages

For some applications, it may be acceptable to start at either the latest message (e.g., for clickstream analytics or log analytics) or earliest message (e.g., for anything idempotent or anything that can be duplicated). However, for other applications, neither of these two options may be acceptable. The desired behavior may be that the consumer starts at message 8000 and consumes just the unread 1,998 messages, as shown below.



To do this, the consumer needs to reset its consumer offsets to something meaningful in the new datacenter. As discussed in the section [Timestamp Preservation](#), consumers cannot reset their consumption by exclusively relying on offsets to determine where to start because the offsets may differ between clusters. Offsets may vary between datacenters, but timestamps will not. With timestamp preservation in the messages, it is the timestamp that has similar meaning between the clusters, and a consumer can start consumption at an offset that is derived from a timestamp.

Confluent Platform version 5.0 introduces a new feature that automatically translates offsets using timestamps so that consumers can failover to a different datacenter and start consuming data in the destination cluster where they left off in the origin cluster. To use this capability, configure Java consumer applications with an interceptor called [Consumer Timestamps Interceptor](#), which preserves metadata of consumed messages including:

- Consumer group ID
- Topic name
- Partition
- Committed offset
- Timestamp

This consumer timestamp information is preserved in a Kafka topic called `__consumer_timestamps` located in the origin cluster. Replicator does not replicate this topic since it has only local cluster significance.

As Confluent Replicator is copying data from one datacenter to another, it is concurrently:

- Reading the consumer offset and timestamp information from the `__consumer_timestamps` topic in the origin cluster to understand a consumer group's progress
- Translating the committed offsets in the origin datacenter to the corresponding offsets in the destination datacenter

- Writing the translated offsets to the `__consumer_offsets` topic in the destination cluster, as long as no consumers in that group are connected to the destination cluster

When Replicator writes the translated offsets to the `__consumer_offsets` topic in the destination cluster, it applies the offsets to any topic renames configured with `topic.rename.format`.

If there were already consumers in that consumer group connected to the destination cluster, Replicator will not write offsets to the `__consumer_offsets` topic since those consumers will be committing their own offsets. Either way, when a consumer application fails over to the backup cluster, it looks for and will find previously committed offsets using the normal mechanism.

Accuracy Of Calculated Offset

Using the Consumer Timestamps Interceptor described in [\[Where To Resume After Failover\]](#), a consumer group that fails over to the new datacenter can resume consuming messages from where it left off in the origin cluster. Consumers will not miss any messages, but depending on the factors that affect the accuracy of the calculated offset, consumers may consume duplicate messages right after a failover.

The accuracy of the calculated offset is affected by:

- Replication lag
- Periodic nature of offset commits
- Fidelity of timestamps (e.g., the number of records with same timestamp)

Multi-Datacenter Bringup

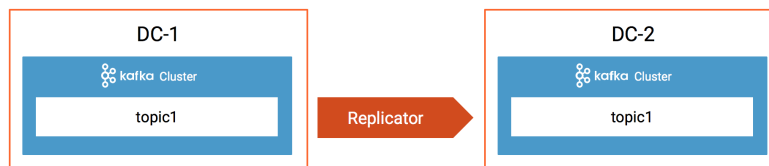
Note: This section assumes that you understand the basic procedure of bringing up a Kafka deployment with its own ZooKeeper quorum and cluster of Kafka brokers, or that you are using Confluent Cloud. If you need guidance on configuring these core Kafka services, see the [quick start](#).

Confluent Replicator

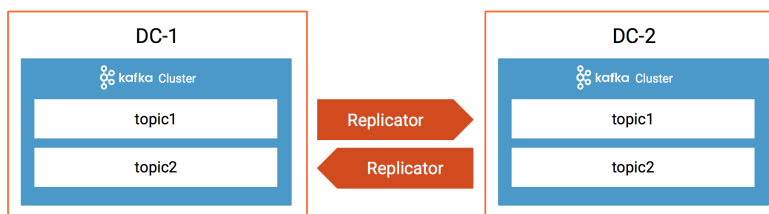
Configuring Replicator

Under the hood, Confluent Replicator is a Kafka connector and therefore runs within the Kafka Connect framework. Replicator inherits all the benefits of the Kafka Connect API, including scalability, performance and fault tolerance. More specifically, Confluent Replicator consumes messages from the origin cluster and then Kafka Connect workers handle the producer functionality of writing messages to the destination cluster. These Kafka Connect workers should be located in the same datacenter as the destination Kafka cluster. This is because Replicator is designed for the embedded consumer to read for the origin cluster over the higher latency inter-datacenter connection.

For an active-passive multi-datacenter design, the diagram below shows the Replicator connector running in DC-2, copying data one-way from DC-1 to DC-2.



For an active-active multi-datacenter design where data is replicated bi-directionally, one Replicator connector copies data from DC-1 to DC-2, and another Replicator connector copies data from DC-2 to DC-1. The diagram below shows two-way Replicator, with each connector running in DC-1 and DC-2.



The configuration parameters that will differ between these Replicator instances are as follows, with example configuration settings.

Configuration Parameter	DC-1 to DC-2	DC-2 to DC-1
<code>name</code>	<code>replicator-dc1-to-dc2</code>	<code>replicator-dc2-to-dc1</code>
<code>src.kafka.bootstrap.servers</code>	<code>dc1-broker1:9092</code>	<code>dc2-broker2:9092</code>
<code>dest.kafka.bootstrap.servers</code>	<code>dc2-broker2:9092</code>	<code>dc1-broker1:9092</code>
<code>src.consumer.group.id</code>	<code>replicator-dc1-to-dc2</code>	<code>replicator-dc2-to-dc1</code>

Note that Confluent Replicator itself has an embedded consumer and, by default, commits its own consumer timestamps to the origin cluster. Its offsets are translated to the destination cluster. This is used in active-passive designs to restore service during failback when a new Replicator starts to replicate back to DC-1 only the new messages produced in DC-2 during the downtime, as described in the section [Data Synchronization in Active-Passive Design](#). However, in active-active designs, this is not needed because there are two separate Replicator instances running in each direction. So to simplify the work in an active-active design, disable Replicator from committing its own consumer timestamps by configuring `offset.timestamps.commit=false`.

For active-active design, here is the minimal sample Replicator configuration for DC-1 to DC-2. This assumes a topic whitelist filter on `topic1`, the topic that is configured for replication between datacenters. If you are also using Confluent Schema Registry, the topic filter should also include the topic `_schemas`, but only in one direction (more on that in the section [Confluent Schema Registry](#)). This configuration also enables Replicator to use header information to prevent cyclic replication of topics.

```
"name" : "replicator-dc1-to-dc2",
"config" :
{
  "connector.class" :
  "io.confluent.connect.replicator.ReplicatorSourceConnector",
  "topic.whitelist" : "topic1,_schemas",
  "key.converter" : "io.confluent.connect.replicator.util.ByteArrayConverter",
  "value.converter" :
  "io.confluent.connect.replicator.util.ByteArrayConverter",
  "header.converter":
  "io.confluent.connect.replicator.util.ByteArrayConverter",
  "src.kafka.bootstrap.servers" : "dc1-broker1:9092",
  "dest.kafka.bootstrap.servers" : "dc2-broker2:9092",
  "provenance.header.enable": "true",
  "src.consumer.group.id": "replicator-dc1-to-dc2",
  "offset.timestamps.commit" : "false",
  "tasks.max" : "4",
  "topic.create.backoff.ms" : "10000"
}
```

Here is the minimal sample Replicator configuration for DC-2 to DC-1. It is similar to the above, except it has a different name and consumer group ID, as well as swapped values for `src.kafka.bootstrap.servers` and `dest.kafka.bootstrap.servers`.

```

"name" : "replicator-dc2-to-dc1",
"config" :
{
  "connector.class" :
"io.confluent.connect.replicator.ReplicatorSourceConnector",
  "topic.whitelist" : "topic1",
  "key.converter" : "io.confluent.connect.replicator.util.ByteArrayConverter",
  "value.converter" :
"io.confluent.connect.replicator.util.ByteArrayConverter",
  "header.converter":
"io.confluent.connect.replicator.util.ByteArrayConverter",
  "src.kafka.bootstrap.servers" : "dc2-broker2:9092",
  "dest.kafka.bootstrap.servers" : "dc1-broker1:9092",
  "provenance.header.enable": "true",
  "src.consumer.group.id": "replicator-dc2-to-dc1",
  "offset.timestamps.commit" : "false",
  "tasks.max" : "4",
  "topic.create.backoff.ms" : "10000"
}

```

Confluent Replicator version 4.1 or higher removes the requirement that Replicator communicates to ZooKeeper in the origin or destination clusters. Use these later versions if connecting to Confluent Cloud or if access to ZooKeeper is otherwise unavailable. Consult the [documentation](#) to configure Replicator for earlier versions.

Running Replicator

This section describes how to run Replicator as a distinct connector within a Kafka Connect cluster. (Alternatively, you may deploy a [Replicator Executable](#), which bundles a distributed Kafka Connect worker and a Replicator connector in a single application.) To run Replicator with a Kafka Connect cluster, you first have to initialize Kafka Connect, which in production should always use [distributed mode](#) for scalability and fault tolerance. You can use [Confluent Control Center](#) for central management of all Kafka connectors.

```
# Start Kafka Connect in DC-1
$ connect-distributed /etc/kafka/connect-distributed.properties

# Start Kafka Connect in DC-2
$ connect-distributed /etc/kafka/connect-distributed.properties
```

Next, add the Replicator connector to Kafka Connect. The Replicator instance copying data from DC-1 to DC-2 should run in DC-2, and the Replicator instance copying data from DC-2 to DC-1 should run in DC-1.

```
# Add the DC-1 to DC-2 Replicator connector to the DC-2 Connect workers
$ curl -X POST -H "Content-Type: application/json" --data @replicator-dc1-to-dc2.properties.json http://dc2-connect2:8083/connectors

# Add the DC-2 to DC-1 Replicator connector to the DC-1 Connect workers
$ curl -X POST -H "Content-Type: application/json" --data @replicator-dc2-to-dc1.properties.json http://dc1-connect1:8083/connectors
```

You can interact with Kafka Connect via a REST API to [manage and verify the connector](#):

```
# On DC-1
$ curl http://dc1-connect1:8083/connectors/replicator-dc2-to-dc1

# On DC-2
$ curl http://dc2-connect2:8083/connectors/replicator-dc1-to-dc2
```

Once running, Replicator will copy topics per the following filters:

- 1) Topic-level filters: whitelist, blacklist or regex defined in the Replicator properties file.
- 2) Message-level filters: automatically prevent messages from being copied in a continuous loop between datacenters, using provenance information in message headers, as described in the section [Preventing Cyclic Repetition of Messages](#).

While Replicator is easy to run off the shelf, be sure to follow the [installation and configuration](#)

[procedures](#), and [tune Replicator](#) to optimize it for your environment. Replicator also supports secure communication over SSL for both the [origin](#) and destination clusters. For general configuration options, see the [Confluent Replicator documentation](#).

Java Consumer Applications

After a disaster event, you may move client applications from the failed datacenter to another datacenter. Java consumer applications should ideally resume consumption in the new datacenter where they had left off in the failed datacenter. How close to resuming to the last consumed message depends on a few factors, as described in the section [\[Accuracy of Calculated Offset\]](#).

Confluent Replicator helps these consumers automatically resume consumption at the right offset without requiring developers to manually set it as long as:

- Replicator is version 5.0 or later
- Consumer application is Java
- Consumer application is using the Consumer Timestamps Interceptor
- Consumer application is using the same group IDs in the origin and destination clusters

To use this feature, configure your Java consumer application with the Consumer Timestamps Interceptor. No additional configuration changes are required for Replicator. This interceptor preserves the required timestamp information, which is later used by Confluent Replicator as described in the section [Consumer Offset Translation](#).

```
interceptor.classes=io.confluent.connect.replicator.offsets.ConsumerTimestampsInterceptor
```

This interceptor is located in the Confluent Replicator JAR, `kafka-connect-replicator-<release>.jar`, and the Confluent Replicator JAR must be available in the `CLASSPATH` of the consumer JVM.

Confluent Schema Registry

Confluent Schema Registry provides a wide set of configuration parameters, and this white paper describes the most important ones. First, configure `host.name` to be unique per Schema Registry instance. It is important that you change this parameter from the default value `localhost`. This is because other Schema Registry instances must be able to resolve this hostname and you need to be

able to track which instance is the leader.

The producer and consumer clients need a common source of truth for schema IDs, which is a Kafka topic in the primary datacenter with a topic name specified by `kafkastore.topic`. This schema topic is automatically created when you start the first Schema Registry instance, and it has high durability settings. It is a compacted topic with a replication factor of three.

Configure all the Schema Registry instances to use the DC-1 Kafka cluster, defined by `kafkastore.bootstrap.servers`. Replicate the schemas topic to DC-2 as a backup so that in case something happens in DC-1 that renders the topic unrecoverable, you can still deserialize messages.

Finally, configure the Schema Registry instances in the primary datacenter to be eligible to become leader, which would enable them to register new schemas, and configure the Schema Registry instances in the secondary datacenter to not be eligible, which would disable them from registering new schemas. Leader eligibility can be set with the configuration parameter `leader.eligibility` (prior to Schema Registry version 6.0, `leader.eligibility` was called `master.eligibility`). Setting `leader.eligibility=true` across multiple Schema Registry instances in the primary datacenter does not mean there will be multiple leaders; it is just a flag to identify whether that instance can become leader.

Configuration Parameter	DC-1 (primary) Value	DC-2 (secondary) Value
<code>kafkastore.bootstrap.servers</code>	PLAINTEXT://dc1-broker1:9092	PLAINTEXT://dc1-broker1:9092
<code>leader.eligibility</code>	true	false

You should also make one change to the Replicator configuration. Since DC-1 is the primary datacenter from a Schema Registry point of view, DC-1 will need to replicate the topic defined by `kafkastore.topic` to DC-2 for backup. The `kafkastore.topic` topic in DC-2 will never be used for reading or writing since all Schema Registry instances in DC-2 will point back to DC-1, but you may want it for backup. In the Replicator JSON file that is copying data from DC-1 to DC-2, if you have configured any topic filters, make sure it will replicate whatever name is defined by `kafkastore.topic` (by default, this value is `_schemas`). For example, if you are already using the `topic.regex` parameter to filter topics, modify it to include `_schemas`:

```
"topic.regex" : "dc1.*|_schemas"
```

Once you have Schema Registry running in both datacenters, check the Schema Registry logs (the location of this Schema Registry log file is defined in `/etc/schema-registry/log4j.properties`):

1) Verify how each of the local Schema Registry instances is configured for leader eligibility:

```
$ grep leader.eligibility /usr/logs/schema-registry.log
leader.eligibility = true
```

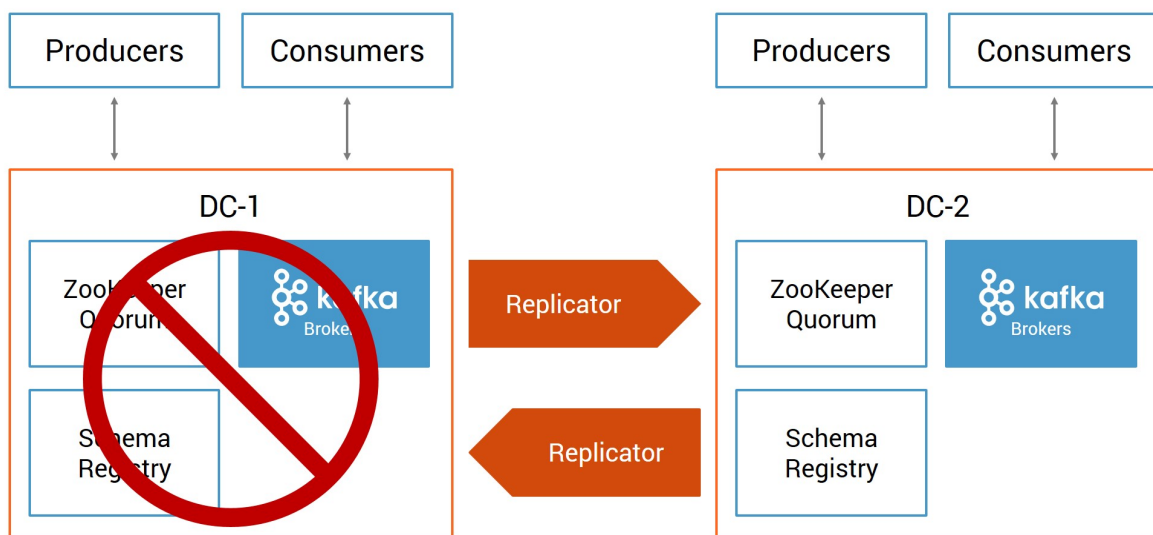
2) Verify which Schema Registry instance was elected leader. Below shows `dc1-schema-registry` is leader. Alternatively, if the Schema Registry instances were started with open JMX ports, you may also check the MBean `kafka.schema.registry:type=master-slave-role`.

```
$ grep "leader election" /usr/logs/schema-registry.log
[2020-08-24 16:16:16,434] INFO Finished rebalance with leader election
result: Assignment{version=1, error=0, leader='sr-1-64384506-e67a-410a-9179-
7849a488c20f', leaderIdentity=SchemaRegistryIdentity{version=1,host=schema-
registry-dc1,port=8081,scheme=http,leaderEligibility=true},
nodes=[SchemaRegistryIdentity{version=1,host=schema-registry-
dc2,port=8082,scheme=http,leaderEligibility=false},
SchemaRegistryIdentity{version=1,host=schema-registry-
dc1,port=8081,scheme=http,leaderEligibility=true}]}
(io.confluent.kafka.schemaregistry.ledelector.kafka.KafkaGroupLeaderElecto
r)
```

When Disaster Strikes

Datcenter Failure

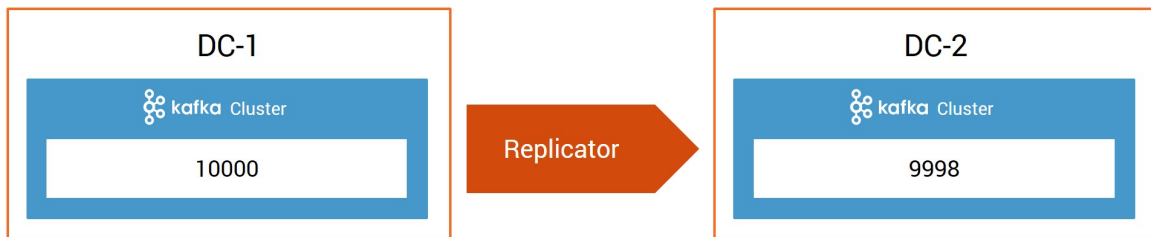
Disasters that cause data loss and downtime in a partial or complete failure of a datacenter can be due to natural disasters, flooding, fire, power loss, external connectivity loss, physical facility compromise, etc. When these disasters strike, they can cripple an entire datacenter for an indeterminate amount of time. In the context of Kafka, what happens in the event of such a disaster?



Consider the failover workflow if a disaster event causes DC-1 to fail.

First, client applications that were connected to DC-1 time out or fail completely when DC-1 fails, whereas client applications that were connected to DC-2 continue to produce and consume messages from DC-2. Replicator in DC-2 that was copying data from DC-1 to DC-2 is still running, but it's no longer able to consume data from DC-1 since it has failed.

Due to replication lag, there may be some messages that were committed to the origin cluster but have not yet arrived to the destination cluster. Consider a producer that may have written 10000 messages to a topic in DC-1. Replicator was copying messages asynchronously to DC-2, but when the disaster event occurred, some of the last messages were not copied yet to DC-2. In the example below, 9998 messages of the original 10000 messages were replicated before the disaster event. This results in a loss of some messages.



As a result, it is very important to [monitor the replication lag](#). Some replication lag is normal depending on the network latency between the two datacenters. However, if it keeps growing, it may indicate that data production rate in the origin cluster exceeds Replicator throughput rate and that you need to address its performance.

Failing Over Client Applications

Client applications that were originally connecting to DC-2 continue to work, and they may be already consuming from topics written by clients in DC-2 and topics replicated from DC-1. No new data will be written to the topics that originated in DC-1 while DC-1 is down, but consumers will happily continue processing.

Meanwhile, client applications that were originally connecting to DC-1 no longer work. Depending on your business requirements, this may be acceptable, and you can leave them down in DC-1 until the datacenter recovers.

On the other hand, for business continuity you may want to failover client applications from the failed datacenter to the still-working datacenter. Client applications will need to reinitialize and refresh the metadata for the "new" cluster, to use DC-2 instead of DC-1. This involves reconfiguring their bootstrap servers and related cluster connection settings to DC-2. How manual or automated the failover workflow is depends on the recovery time objective (RTO), which is the point in time after the disaster event when the failover completes. Ideally it is as soon after disaster time as possible to minimize downtime. The more aggressive the RTO is, the more you want an automated workflow, which depends on the service discovery mechanism and failover logic developed into the applications.

Reset Consumer Offsets

As discussed earlier, starting consumers at the right point in the topic in the backup cluster cannot exclusively rely on offsets because they may differ between the clusters. In contrast to offsets, the preserved message timestamps do have a similar meaning between the clusters, so a consumer can resume consumption at an offset derived from a timestamp. The recovery point objective (RPO) is the last point in time before the disaster event that the application can recover to with known good data.

Usually it is as close before disaster time as possible to minimize data loss.

While developers still need to manage when and how to move client applications between datacenters, it is quite easy for consumers to determine where to start processing. A consumer application can automatically determine where to resume consumption in the new datacenter based on where it left off in the original datacenter if:

- Replicator is version 5.0 or later
- Consumer application is Java
- Consumer application is using the Consumer Timestamps Interceptor
- Consumer application is using the same group IDs in the origin and destination clusters

If Java consumer applications are using the Consumer Timestamps Interceptor, then Replicator uses message timestamp information to commit its offsets in the destination cluster. As a result, the consumer can look up its offsets in the `__consumer_offsets` topic and automatically start at the right offset, without manual intervention to preset the offset. The offset calculation is based on timestamps, so the DC-2 offset matches the earliest record with that same timestamp at the DC-1 offset.

In some cases, you may want to manually preset consumer offsets derived from a timestamp before the disaster event. This may be because of any of the following reasons:

- Replicator is running a version less than 5.0
- Consumer application is not Java
- Consumer application is Java but is not using the Consumer Timestamps Interceptors
- Consumer application is not using the same group IDs in the origin and destination clusters
- A developer wants to decide the reset time manually outside the consumer application

For those situations, please refer to [Manually Reset Consumer Offsets](#) in the Appendix.

Schema Registration

If you are using Schema Registry, the leader instances in DC-1 are now down. The follower Schema Registry instances in DC-2 will continue to provide existing schemas and schema IDs to client applications, though they cannot register new schemas. They are also ineligible to become leader instances because they were configured with `leader.eligibility=false`.

During the disaster period, we recommend that you do not allow producers to send new messages that require registration of a new schema, and do not restart the instances in DC-2. This is for two reasons:

1. In active-passive designs, allowing new schemas to be written to DC-2 will complicate the recovery process of DC-1 and failback workflow.
2. Due to replication lag, there is a very small window of time during which a schema may have been committed to the backing Kafka topic in DC-1, but was not yet replicated to DC-2. Allowing Schema Registry instances in DC-2 to become leaders may result in losing those schemas and losing the ability to read messages that use those schemas.

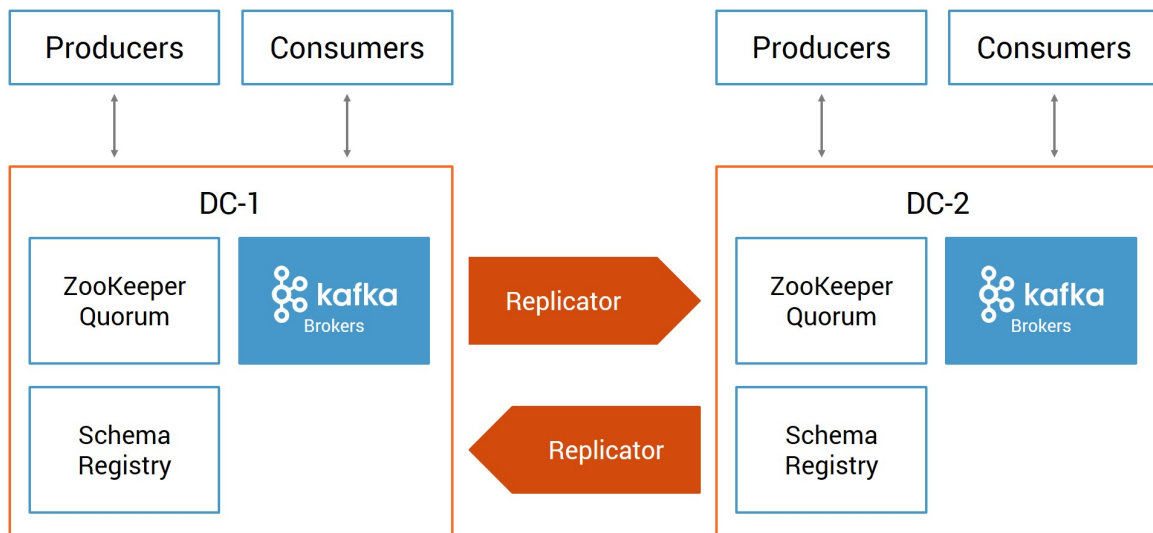
However, if registering new schemas is a requirement, then you need to take extra steps during failover and failback to reconfigure the Schema Registry instances in DC-2. Before the disaster event, the DC-2 Schema Registry instances read new schemas from DC-1. Now, they need to be redirected to the DC-2 cluster. See the table below for required changes to the Schema Registry configuration. You will need to restart Schema Registry for these changes to take effect.

Configuration Parameter	DC-2 Original Value	DC-2 Updated Value
<code>kafkastore.bootstrap.servers</code>	PLAINTEXT://dc1-broker1:9092	PLAINTEXT://dc2-broker2:9092
<code>leader.eligibility</code>	false	true

Failback

Restoring the Kafka Cluster

When the original failed datacenter recovers from the disaster event, you must restore the multi-datacenter setup, synchronize data between the Kafka clusters and restart client applications appropriately. This failback workflow may seem like just a footnote in old school frameworks because those frameworks need to only consider new data. However, failback in Kafka requires further consideration, because Kafka needs to replicate data that has already been consumed as well as data that is yet to be consumed.



When the original datacenter recovers from the disaster event, ensure that both datacenters have fully functioning ZooKeeper and Kafka brokers.

You need to restore Schema Registry to the original design, and its failback workflow depends on what you did during the failover. If you decided to leave the DC-2 Schema Registry instances running as follower instances and did not allow registration of new schemas during the disaster period, then no Schema Registry changes are required for failback. However, if as part of failover you made the configuration changes that allowed new [Schema Registration](#), then all the new schema data needs to be replicated from DC-2 to DC-1 when it recovers.

In an active-active design, assuming Replicator was configured to copy the `_schemas` topic bi-directionally, Replicator will automatically copy the new schemas from DC-2 to DC-1. Replication will continue where it left off before the disaster event, because Replicator tracks the offsets of messages it has already processed. Assuming Replicator has enabled provenance information in message headers, there will be no cyclic repetition of schemas.

On the other hand, in an active-passive design, assuming Replicator was configured to copy the `_schemas` topic only from DC-1 to DC-2, and if some producers in DC-2 registered new schemas, then further steps are required to synchronize schemas between the datacenters. Please refer to [Data Synchronization in Active-Passive Design](#) for procedures to synchronize the schema data.

Finally, reset the Schema Registry configurations to values originally set and restart all Schema Registry instances.

Data Synchronization

In an active-active design where Replicator already copies topics bi-directionally, data synchronization on failback is automatic. When DC-1 is fully restored, Replicator will automatically copy any new data from DC-2 to DC-1.

Because Replicator tracks the offsets of messages it has already processed, replication will continue where it left off before the disaster event. Replicator as a source connector uses the topic defined by `offset.storage.topic` (by default, `offset.storage.topic=connect-offsets`) to track offsets of messages consumed from the origin cluster. (This is similar to how typical consumers use the `__consumer_offsets` topic to track offsets of their messages consumed.) You can monitor Replicator progress in this topic to determine how far Replicator is in synchronization of messages from DC-2 to DC-1.

```
# Verify offsets tracked for this replicator
$ kafka-console-consumer --topic connect-offsets --bootstrap-server dc1-
broker1:9092 --consumer-property enable.auto.commit=false --from-beginning
--property print.key=true
["replicator-dc2-to-dc1",{"topic":"topic1","partition":0}] {"offset":3}
["replicator-dc2-to-dc1",{"topic":"topic1","partition":0}] {"offset":5}
```

In an active-passive design in which there was not already Replicator copying data from DC-2 to DC-1, you may need to first synchronize DC-1 to the latest state in DC-2. Please refer to [Data Synchronization in Active-Passive Design](#) for procedures on synchronizing all Kafka data.

Message Ordering

After data synchronization, it is possible that the message order may differ between DC-1 and DC-2 for a given topic partition. This is because, at the moment of the disaster event, there might have been data in DC-1 that was not yet replicated to DC-2 due to replication lag. To illustrate this point, earlier examples in the section [Consumer Offset Translation](#) had shown DC-1 with 10000 messages in a topic, and DC-2 with 9998 of those messages replicated. After failback, as the original datacenter recovers and Replicator starts again, it examines the `connect-offsets` topic to determine where it left off. In this example, Replicator replicates the last two of the 10000 messages because they had not been replicated before the disaster event. However, these two messages are copied to the topic in DC-2 *after* producers may have written more recent messages to this same topic, while DC-1 was down. This may result in out of order messages for a given topic partition. To address this, consider developing consumer applications or Kafka streaming applications.

Client Application Restart

Once the primary data center recovers and data is synchronized, point client applications back to DC-1. Client applications will need to reinitialize and refresh the metadata for the original cluster. This involves reconfiguring their bootstrap servers and related cluster connection settings to DC-1.

Demo

Learn more by running a [multi-datacenter demo](#) in GitHub. In one command, this demo environment brings up an active-active multi-datacenter environment with Confluent Replicator copying data bidirectionally.

The demo includes a playbook that walks through several scenarios showing how to monitor Replicator. The provided sample Java client application lets you see how data consumption can resume in the new datacenter based on where it left off in the original datacenter. It also shows you how to derive which producers are writing to which topics, and which consumers are reading from which topics, which is especially useful in a more complex multi-datacenter Kafka environment.

This [multi-datacenter demo](#) is useful for testing purposes, and you can adapt the configurations to be more representative of your deployment and run your client applications against it. You can also just use this demo as a configuration reference for multi-datacenter deployments. Disclaimer: This is just for testing—do not take the demo's Docker setup into production!

Summary

This paper discussed the building blocks for designing, configuring and following failover and failback workflows. There are various use cases for [multi-datacenter designs](#), but this paper focused on disaster recovery. Your architecture will vary depending on your business requirements. Whether it includes on prem and Confluent Cloud, you can apply the appropriate building blocks to strengthen your disaster recovery plan.

The Confluent disaster recovery solution, which includes [Confluent Replicator](#), [Confluent Schema Registry](#), timestamp preservation and consumer offset reset features, drives a business continuity plan for mission-critical applications. Confluent Replicator is the key for these multi-datacenter deployments. Replicator reads data from one cluster and then writes those messages to another cluster. Replicator also uses provenance information in message headers to prevent cyclic repetition of messages between clusters. When disaster strikes, Replicator can automatically reset offsets for consumers so that they can start reading messages where they left off. As topic metadata or partition count changes in the origin cluster, Replicator replicates the changes in the destination cluster. New topics are automatically detected and replicated to the destination cluster, and as throughput increases, Replicator automatically scales to accommodate the higher load.

Confluent Platform is the leading distribution of Apache Kafka®, containing all of Kafka's capabilities and enhancing it with integrated, tested and packaged features that make architecting and managing large-scale streaming pipelines easier and more reliable.

Visit www.confluent.io/download to download the latest version of Confluent Platform.

Version 2.0 November 2018

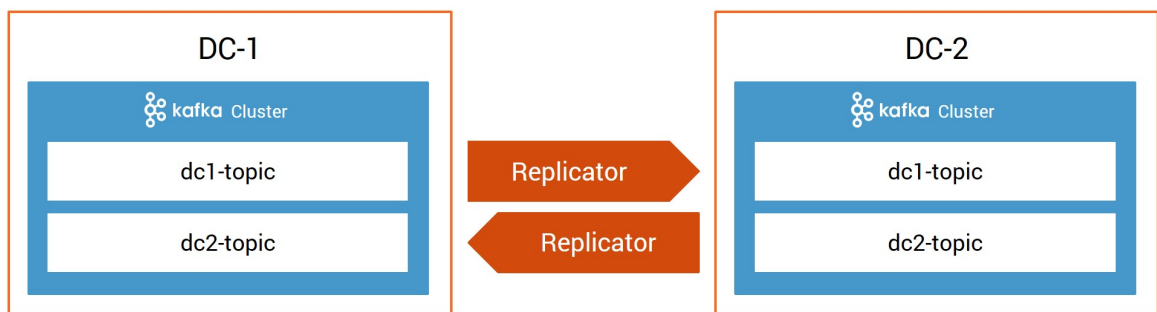
Appendix

Topic Naming Strategies to Prevent Cyclic Repetition

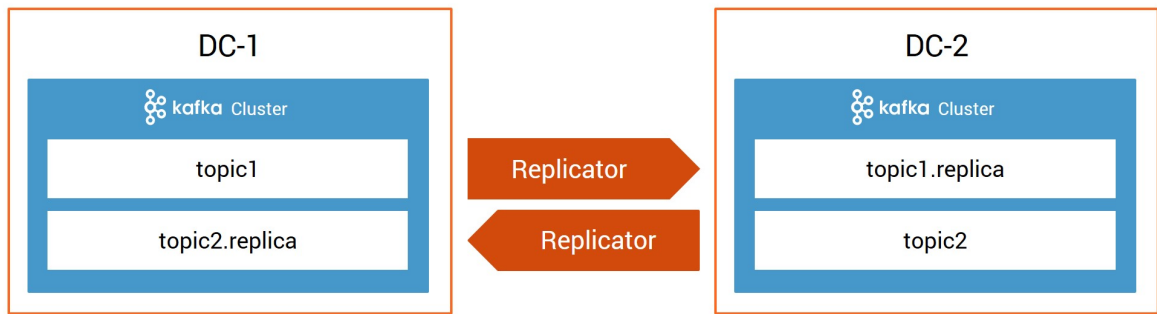
Since Confluent Replicator version 5.0.1, active-active multi-datacenter designs with bi-directional Replicator can automatically prevent data from being copied in a continuous loop between datacenters. This capability using provenance headers is described in the section [Preventing Cyclic Repetition of Messages](#).

If you do not want to or cannot use this feature, remember that preventing cyclic repetition of data is not automatic. You need to decide on a topic naming strategy with unique topic names per datacenter to prevent the loop, and use that strategy in your client applications. There are essentially two main strategies:

1. Name original topics to include the datacenter name. For example, DC-1 could have a topic called **dc1-topic**, and DC-2 could have a topic called **dc2-topic**. You can configure Replicator to use a regular expression to only copy topics with names that match the topic's origin datacenter, thereby breaking the cyclic repetition. The regular expression filter would not copy **dc1-topic** from DC-2 back to DC-1, nor copy **dc2-topic** from DC-1 back to DC-2.



2. Replicate original topics to the destination cluster with new names to indicate that they are copies. For example, DC-1 could have a topic called **topic1**, and when it is replicated to DC-2, Replicator would be configured to rename it to **topic1.replica** by setting the **topic.rename.format** configuration parameter. You configure Replicator to only copy topics with names that do not have ***.replica**, and to rename the topic as it is copied to the destination cluster, thereby breaking the cyclic repetition. The regular expression filter would not copy **topic1.replica** from DC-2 back to DC-1, nor copy **topic2.replica** from DC-1 back to DC-2.



Which strategy to use depends on how you want to write your client applications. We recommend the first strategy because it uses an easy regular expression filter in Replicator, and consumers can subscribe to `*-topic` (e.g., `dc1-topic` and `dc2-topic`) to receive topic data regardless of which datacenter they are connected to. The remainder of this white paper shows examples with this strategy, whereby the datacenter identifier is prepended to the topic name.

Manually Reset Consumer Offsets

Since Confluent Platform version 5.0, Java consumer applications can automatically restart consuming data where they left off after a disaster event. This capability is described in the section [Reset Consumer Offsets](#).

Alternatively, you may want to manually reset consumer offsets to a reset time derived from a timestamp before the disaster event. If the original consumers in DC-1 were lagging even before the disaster event, there will be even more messages that were not consumed, so the reset time could be an even earlier point in time. To address this, you can monitor the consumer lag metric [records-lag-max](#), which is the lag in terms of number of records for any partition in a time window that the consumer is behind in the log. Record this metric at regular intervals, and if there is a disaster event, look up the lag to determine how far back to roll.

You can manually reset consumer offsets in two ways:

1. Inside the Java client application using the Kafka consumer API
2. Outside the client application using the Kafka command line tool

If you want to manually reset the consumer offsets from within the consumer application itself, you can use a combination of API calls: `offsetsForTimes()` and `seek()`.

The `offsetsForTimes(Map<TopicPartition, Long> timestampsToSearch)` API described in [KIP-79](#) was introduced in Kafka version 0.10.1, which is the minimum version both brokers and client applications should run. As explained in the [Javadocs](#), this method looks up the offsets for the given

partitions by timestamp. The returned offset for each partition is the earliest offset whose timestamp is greater than or equal to the given timestamp. Once the offsets are returned, the consumer can move to those offsets in the logs using the `seek (TopicPartition partition, long offset)` method. As explained in the [Javadocs](#), this method overrides the fetch offsets that the consumer uses on the next `poll()`.

Using `offsetsForTimes()` and `seek()` together, the following sample code resets consumer offsets for a list of partitions assigned to the consumer. So in case of failover, the developer will need to identify the timestamp to rewind to (i.e., `RESET_TIME`), and pass it into a failover function in the client application.

```
// Get set of partitions assigned to consumer
final Set<TopicPartition> partitionSet = consumer.assignment();

// Initialize HashMap used for partition/timestamp mapping
final Map<TopicPartition, Long> timestampsToSearch = new HashMap<>();

// Add each partition and timestamp to the HashMap
for (TopicPartition topicPartition : partitionSet) {
    timestampsToSearch.put(topicPartition, RESET_TIME);
}

// Get the offsets for each partition
Map<TopicPartition, OffsetAndTimestamp> result = consumer.offsetsForTimes
(timestampsToSearch);

// Seek to the specified offset for each partition
for (Map.Entry<TopicPartition, OffsetAndTimestamp> entry : result.entrySet())
{
    consumer.seek(entry.getKey(), entry.getValue().offset());
}
```

If you want to manually reset the consumer offsets from outside the application, for example, if the application is not Java, then you can use a command line tool to reset consumer offsets. Described in [KIP-122](#) and introduced in Kafka version 0.11.0, the `--reset-offset` argument was added to the command line `kafka-consumer-groups`. It resets consumer offsets, operating on one consumer group at a time. Effectively, resetting the offset sends an offset update to the `__consumer_offsets` topic for this [consumer group, topic, partition] tuple. The consumer group should be inactive at the time of offset reset.

[KIP-122](#) describes various methods to rewind the offset. The example usage below shows how to reset to a specific time which should logically correspond to a time right before the disaster recovery event. In the scenario described above, this specific time (**2017-07-05T16:34:33.236** in this example) is equivalent to offset 8000 in the log, so the new offset gets reset to 8000.

```
# Reset to a specific datetime before the disaster event
$ kafka-consumer-groups --new-consumer --bootstrap-server dc2-
broker2:9092 --reset-offsets --topic dc1-topic --group my_consumer --execute
--to-datetime 2017-07-05T16:34:33.236

...
TOPIC      PARTITION  NEW-OFFSET
dc1-topic  0          8000
```

Describing the consumer group will now show the new updated offset at 8000, which is where the consumption ended in DC-1 right before the disaster recovery event.

```
$ kafka-consumer-groups --new-consumer --bootstrap-server dc2-broker2:9092
--describe --group my_consumer
Note: This will only show information about consumers that use the Java
consumer API (non-ZooKeeper-based consumers).
```

Consumer group my_consumer has no active members.

TOPIC	PARTITION	CURRENT-OFFSET	LOG-END-OFFSET	LAG	CONSUMER-ID
HOST	CLIENT-ID				
dc1-topic	0	8000	9998	1998	

Once this offset is reset, the next fetch from this consumer group **my_consumer** for this topic **dc1-topic** in DC-2 will return records starting from the updated offset 8000.

Data Synchronization in Active-Passive Design

To recover from a disaster event, Confluent Replicator and active-active multi-datacenter designs allow for easy synchronization between datacenters. Replicator automatically continues where it left off and prevents cyclic repetition of topics using provenance information in message headers.

In contrast, in active-passive designs, it is simplest to not allow producers to send new messages to the

backup datacenter while the primary is down. However, if producer applications were redirected to the backup datacenter, then there may be new Kafka data in the backup cluster that needs to be synchronized to the primary when it comes back online.

As part of bringing DC-1 back online, if all the data in the Kafka topics in the original cluster was restored, then only data that was produced to DC-2 after the disaster event needs to be copied over to DC-1. Configure a new Replicator instance to copy data from DC-2 to DC-1 using the same consumer group ID, and it will be able to replicate back to DC-1 only the new messages produced in DC-2 during the downtime. Replicator will know where to resume copying because it has an embedded consumer and, by default, Replicator commits its own consumer timestamps to the origin cluster. Its offsets are translated to the destination cluster, as described in the section [Consumer Offset Translation](#).

However, if all the data in the Kafka topics in the original cluster could not be restored, then you need to restore the original cluster DC-1 with all the data in DC-2. Before running Replicator, delete any remaining data in DC-1. While this may seem like an aggressive step, it is the most reliable way to ensure no out-of-order messages. Some of that data may have been deleted anyway depending on how long the primary cluster was down and how that downtime compares to the Kafka data retention period, as defined by the parameter `log.retention.hours`.

Next, use Replicator to copy existing data in DC-2 to restore DC-1. Recall that the backup cluster, DC-2, may have a mix of data:

- Data that was originally produced to DC-1 before the disaster event, replicated with added provenance information in message headers
- Data that was originally produced to DC-2 after the disaster event that has never been replicated and has no provenance headers

If you were to run Replicator with its default filtering behavior, then data that was originally produced to DC-1 before the disaster event would not be replicated back. This is because the provenance information indicates that if a message originated in DC-1, was replicated to DC-2 before the disaster event and was now being copied back to the same topic name in DC-1, then it should be filtered out and not replicated to prevent cyclic repetition. This default filtering behavior, which is desirable during normal operations, would prevent full restoration of the data in DC-1.

To override this default behavior and copy all the messages from DC-2 to DC-1, including messages with and without provenance information in message headers, run a separate Replicator instance and configure it for `provenance.header.filter.overrides`. The format of `provenance.header.filter.overrides` is a semi-colon-delimited list of tuples with three attributes:

1. Regular expression for the cluster ID to match that in the provenance header
2. Regular expression for the topic name to match that in the provenance header
3. Range of timestamps

If there is an existing Replicator instance copying data from DC-2 to DC-1, stop the instance. Reconfigure it or create a temporary Replicator instance to use for failback. Configure explicit overrides for the replication rules to force the data that you want synchronized. The following example overrides the filter behavior for the `_schemas` topic such that it copies all messages from DC-2 to DC-1, and for the `topic1` topic such that it copies messages from DC-2 to DC-1 for messages that were replicated before timestamp 1539634429528. Messages that do not match the filter overrides are subject to normal replication filtering rules that prevent cyclic replication of messages.

```
provenance.header.enable=true
provenance.header.filter.overrides=DC-1,_schemas,-;DC-1,topic1,0-1539634429528
```

Monitor Replicator progress and wait for replication lag to go to zero to determine when DC-1 is caught up to DC-2. All headers in the original messages will still remain, including provenance information in message headers so they will not be replicated again. Then stop this temporary Replicator instance and resume normal operations.