Cloudera Streams Messaging Operator for Kubernetes 1.6.0

Kafka Replication Overview

Date published: 2024-06-11
Date modified: 2026-01-27

https://docs.cloudera.com/

Legal Notice

Contents

Replication overview....4
Replication flows....5
Replication aliases....5
Replication policies....5
Typical replication architecture....7

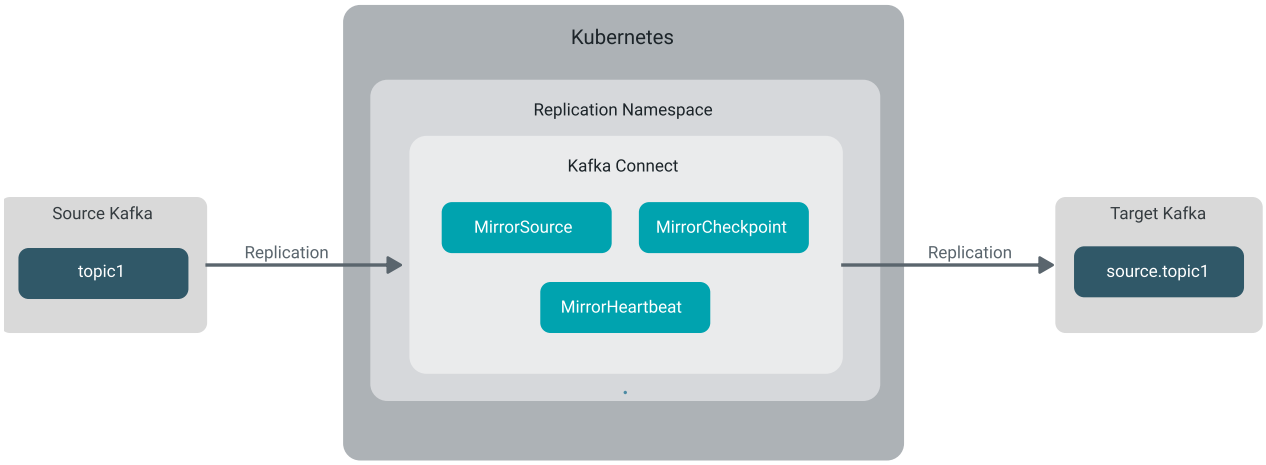
Replication connectors and connector architecture....8
MirrorSourceConnector....9
MirrorCheckpointConnector....10
MirrorHeartbeatConnector....12
Connector task and load balancing....13
Replication connector configurations....14
Recommended configurations for offset syncs and checkpoints topics....15

Replication overview

Learn about data replication across different Kafka clusters using Cloudera Streams Messaging Operator for Kubernetes. Get familiar with the concept of replication flows, replication aliases, and replication policies. Additionally learn the replication architecture recommended by Cloudera.

Cloudera Streams Messaging Operator for Kubernetes does not support MirrorMaker 2 or using the KafkaMirrorMaker2 resource shipped with Strimzi to replicate data between Kafka clusters. Replication between Kafka clusters is instead achieved by manually deploying Kafka Connect clusters and instances of the `MirrorSourceConnector`, `MirrorCheckpointConnector`, and `MirrorHeartbeatConnector`, which are collectively called the **replication connectors**. A replication setup like this is referred to as **Kafka Connect-based** replication.

Figure 1: Replication of data across two Kafka clusters using Kafka Connect-based replication



Using Kafka Connect-based replication offers a complete replication solution that is scalable, robust, and fault tolerant. It supports the same key features as MirrorMaker 2. For example:

- Replication of Kafka topic partitions to have multiple copies of the same data in different Kafka clusters to avoid data loss in case of data center failure.
- Replication of Kafka consumer group offsets to be able to fail over between clusters without losing data.
- Ability to monitor your replication at any time.

In addition, Kafka Connect-based replication has a number of advantages over using MirrorMaker 2, such as:

- Single Messages Transforms (SMTs) can be configured for data replication.
- Manipulating source offsets is possible using the Kafka Connect REST API.
- Some replication architectures, like unidirectional replication, require less resources and Kafka Connect groups when using overrides for heartbeating.

Related Information

Using Single Message Transforms in replication flows
Deploying a replication flow
Checking the state of data replication

Replication flows

Replication involves sending records and consumer group checkpoints from a source cluster to a target cluster. A replication flow (also referred to as replication or flow) specifies a source and target cluster pair, the direction in which data is flowing and the topics that are being replicated.

For example, assume you have two clusters, A and B. You want to replicate data from A to B. To do so you set up an A->B replication flow. If you wanted to replicate from B to A, you set up a B->A replication flow.

Each replication flow specifies what topics to replicate by way of topic filters (also referred to as allow and deny lists) using the topics connector property. Therefore, you have full control over what is and what is not replicated

Replication aliases

In any replication flow, the two clusters taking part in the replication must have an alias. The alias is a short name that represents and identifies the cluster.

Aliases are arbitrary, user-defined names. Generally the alias describes your cluster. For example, you can use aliases that are based on the geographic location of the cluster, like us-east or us-west. Alternatively, in a simple, two-cluster setup with a single replication flow you could use aliases like source and target. Aliases are used by default for prefixing replicated topics. Therefore, using descriptive aliases can help when monitoring replication.

Even though you are free to specify any alias you want, you must use the same aliases for your cluster across all replication flows that you deploy. For example, consider that you set up two replication flows; one between clusters A and B and the second between clusters A and C. You must ensure that the alias of cluster A is the same in both replication flows. For example, A->B and A->C. Additionally, if you later on decide to deploy another replication flow between clusters B and C, you must ensure that both B and C clusters have the same aliases in the newly deployed replication flow as well. For example B->C.

Replication policies

In any replication flow, the selected source topics are replicated to replicated topics on the target cluster. The basic rules of how these topics are replicated is defined by the replication policy.

Cloudera Streams Messaging Operator for Kubernetes ships with the following two replication policies. The main difference between the two policies is how they name replicated topics.

Table 1: Replication policies available in Cloudera Streams Messaging Operator for Kubernetes

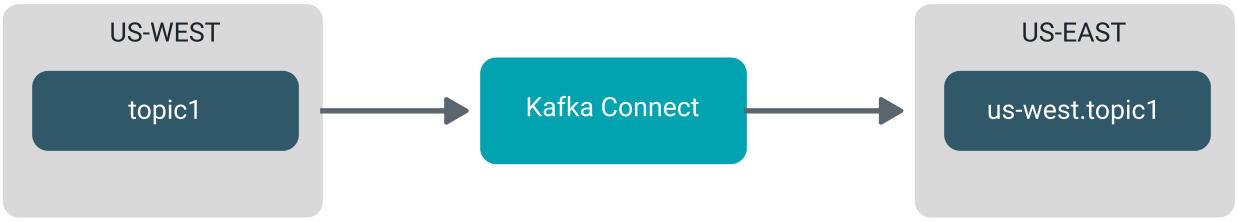
Replication policy	Fully qualified name
`DefaultReplicationPolicy`	`org.apache.kafka.connect.mirror.DefaultReplicationPolicy`
`IdentityReplicationPolicy`	`org.apache.kafka.connect.mirror.IdentityReplicationPolicy`

DefaultReplicationPolicy

The `DefaultReplicationPolicy` is the default and Cloudera-recommended replication policy. This is because the `DefaultReplicationPolicy` is capable of automatically detecting replication loops. This policy prefixes the replicated topic's name with the alias of the source cluster.

For example, the topic1 topic from the us-west source cluster creates the us-west.topic1 topic on the target cluster.

Figure 2: Single-hop replication using the DefaultReplicationPolicy



If a replicated topic is also replicated (there are multiple replication hops in your setup) the replicated topic references all source and target clusters. The prefix in the name will start with the cluster closest to the final target cluster. For example, the topic1 topic replicated from the us-west source cluster to the us-east cluster and then to the eu-west cluster will be named us-east.us-west.topic1.

Figure 3: Two-hop replication using the DefaultReplicationPolicy

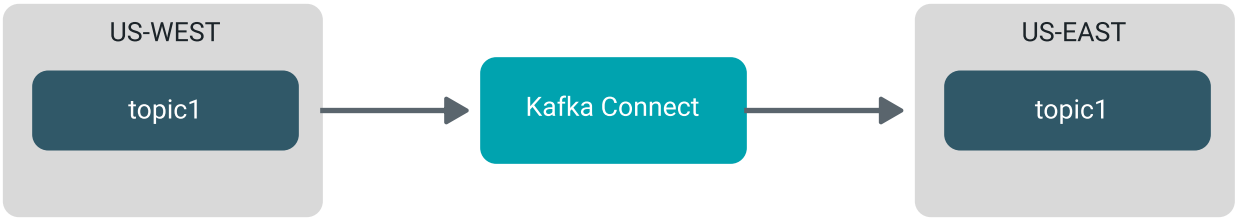
IdentityReplicationPolicy

Important: The `IdentityReplicationPolcy` does not detect replication loops. As a result, if you choose to use the `IdentityReplicationPolcy`, you must ensure that topics are not replicated in a loop between your source and target clusters. You can ensure this by setting up your topic filters in a way that's appropriate for your use case.

The `IdentityReplicationPolicy` does not change the names of replicated topics. When this policy is in use, topics retain the same name on both source and target clusters. This type of replication is also referred to as prefixless replication.

For example, the topic1 topic from the us-west source cluster creates the topic1 replicated topic on the target cluster.

Figure 4: Single-hop replication using the IdentityReplicationPolicy



Cloudera recommends that you use this replication policy in the following use cases.

- Migrating Kafka data from one cluster to another.
- Aggregating the same topic from multiple clusters to a single target cluster.
- Use cases where MirrorMaker 1 compatible replication is required.

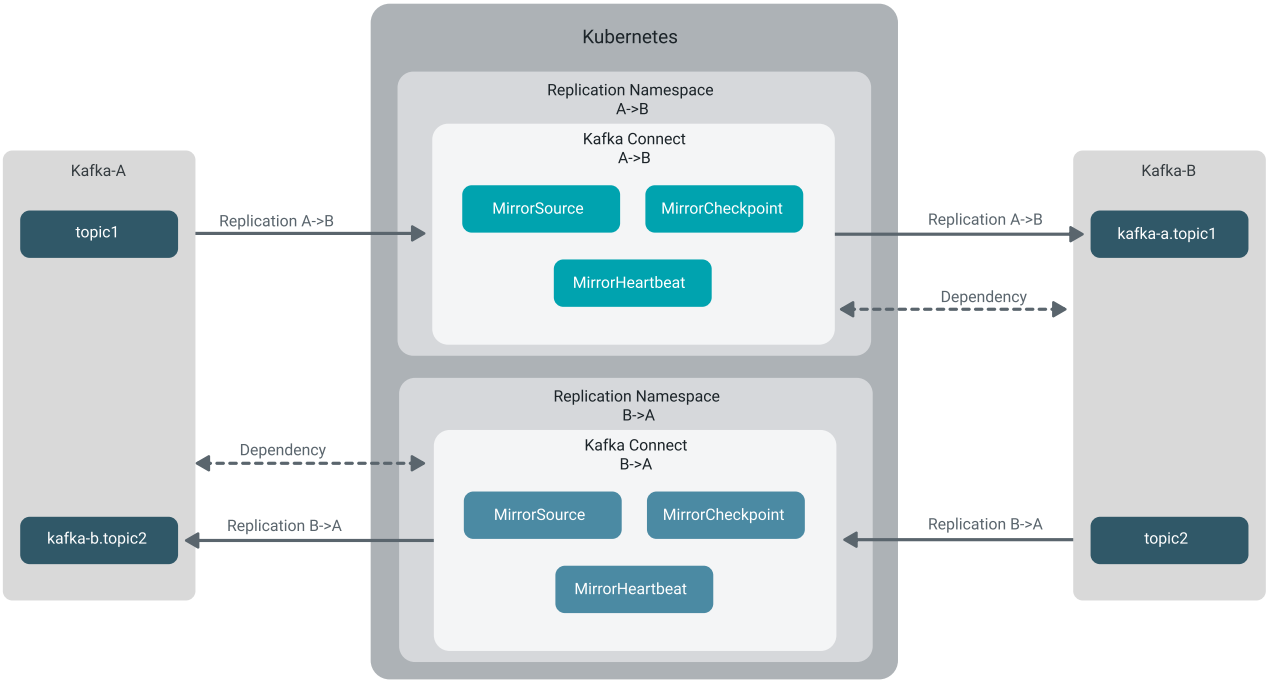
Typical replication architecture

Learn about the typical replication architecture used for replicating Kafka data with Cloudera Streams Messaging Operator for Kubernetes.

When using Kafka Connect-based replication, you set up Kafka Connect clusters and deploy instances of the replication connectors inside the clusters. The Kafka Connect clusters and the connector instances together make up a replication flow.

A typical architecture for a deployment with multiple replication flows is as follows.

Figure 5: Typical replication architecture with two replication flows



Replication flows that you set up in Cloudera Streams Messaging Operator for Kubernetes should follow these architectural principles.

One Kafka Connect cluster for each replication flow

Replication is carried out by the replication connectors. To be able to run these connectors, a Kafka Connect cluster is required where you deploy these connectors. In Cloudera Streams Messaging Operator for Kubernetes, Cloudera recommends that you deploy a Kafka Connect cluster (Kafka Connect group) for each and every replication flow that you want to create.

Deploying a unique Kafka Connect cluster for each replication flow makes it easier to manage your different replication flows. This results in easier monitoring, troubleshooting and reduced rebalance times.

Kafka Connect clusters depend on the target Kafka cluster

Any Kafka Connect cluster that you deploy requires a Kafka cluster as a dependency. Kafka Connect uses the Kafka cluster to store information about its state in internal topics.

For Kafka Connect clusters that you deploy for replication, the cluster must always depend on the target Kafka cluster of replication flow. The dependency is configured in your `KafkaConnect` resource with spec.bootstrapServers.

This dependency makes configuring the connectors that make up the replication flow easier. Properties required to connect to the target cluster can be sourced from the property file of the Kafka Connect workers.

Group IDs and internal topic names follow a consistent naming convention

In a production environment, it is highly likely that you will create multiple replication flows and therefore deploy multiple Kafka Connect clusters. Ensure that the group IDs and internal topic names are explicitly configured for each Kafka Connect cluster. These are configured in the spec of the `KafkaConnect` resource using the following properties:

```
#...
kind: KafkaConnect
spec:
  groupId: [***GROUP ID***]
  configStorageTopic: [***CONFIG TOPIC NAME***]
  offsetStorageTopic: [***OFFSET TOPIC NAME***]
  statusStorageTopic: [***STATUS TOPIC NAME***]
```

By default both the group ID and internal topic names are hardcoded. If you do not set them explicitly in your `KafkaConnect` resource, the IDs and names will clash across your clusters.

Cloudera recommends that you use a consistent naming convention in environments with multiple Kafka Connect clusters. A consistent naming convention can help in avoiding confusion in your configurations down the line.

Replication policy is consistent across connectors and replication flows

The replication policy configured with replication.policy.class connector property must be identical in all connectors instances that make up a replication flow. This is because the replication policy influences the behavior of the connectors.

Additionally, if you are deploying multiple replication flows where a replication flow replicates replicated topics (you have multiple replication hops), you must ensure that all replication flows use the same replication policy.

For example, consider that you are replicating a topic from cluster A to cluster B, and then from cluster B to cluster C. This setup requires two replication flows, A->B and B->C. Both replication flows must use the same replication policy.

Topic filters are consistent across all connectors in a replication flow

The topic filters configured with the topics connector property must be an exact match in the `MirrorSourceConnector` and `MirrorCheckpointConnector` instances that are part of the same replication flow. This is a must have to ensure that both data and offsets are replicated properly.

Cluster aliases are consistent across all replication flows

Cluster aliases configured with the source.cluster.alias and target.cluster.alias connector properties must be configured in each connector instance to use the same alias for the same cluster. This must be true across all replication flows that you deploy.

Replication connectors and connector architecture

Replication of data across different Kafka clusters is carried out by the replication connectors that you deploy in a Kafka Connect cluster dedicated to a replication flow. Get familiar with these connectors, learn about their architecture and configuration properties.

There are three different connectors that you deploy to create a replication. Each has its own purpose and carries out a different task related to replication. The replication connectors are as follows.

- `MirrorSourceConector` – Replicates topics between source and target clusters.

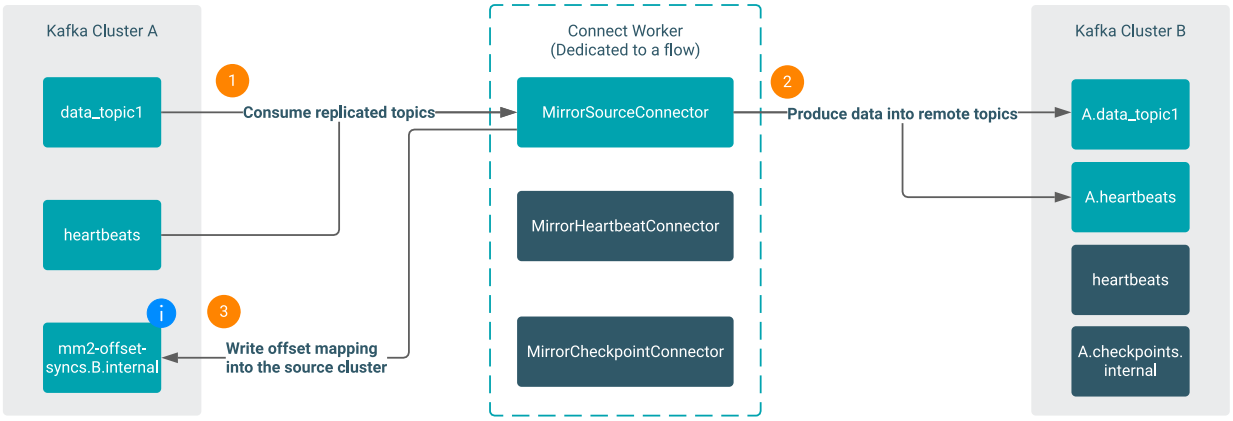
- `MirrorCheckpointConnector` – Replicates the committed group offsets between the source and target clusters.
- `MirrorHeartbeatconnector` – Creates a heartbeats topic in a chosen cluster and periodically produces heartbeats into the heartbeats topic.

MirrorSourceConnector

The MirrorSourceConnector is responsible for replicating topics between the source and the target cluster.

The topics that should be replicated are specified with topic filters (also referred to as allow and deny lists) specified in topics connector property. This gives you full control over what is and what is not replicated. In addition to replicating user specified topics, the connector automatically replicates all heartbeats topics. which are created by the `MirrorHeartbeatConnector`.

Figure 6: MirrorSourceConnector



Important: The `MirrorSourceConnector` guarantees at-least-once delivery of messages. This means that duplicates are possible, but data will not be lost as long as the source cluster is accessible and the replication is not obstructed. The `MirrorSourceConnector` can also use the exactly-once semantic to replicate messages transactionally, avoiding duplicates in the target topic. For more information, see, *Enabling exactly-once semantics for replication flows*.

MirrorSourceTask

The `MirrorSourceTask` is created by the `MirrorSourceConnector`. It is responsible for executing data replication. It uses a producer for writing replicated data to the target cluster. This producer is managed by the Kafka Connect framework, all the other clients are managed by the task itself.

Each task receives its assignment from the `MirrorSourceConnector` as a list of topic partitions. These are assigned to the consumer. The fetched records are then forwarded to the producer. The target topic name is generated based on what replication policy is configured.

Note: Since the `MirrorSourceTask` instances share the load over topic partitions, there is no point setting the task sMax property of the connector to higher than the number of topic partitions that need to be replicated.

Offset sync

In addition to replicating data, the MirrorSourceConnector also manages an offset mapping between the source and target cluster for each replicated topic partition. This offset mapping is called offset sync and it is used by the MirrorCheckpointConnector for replicating consumer group offsets.

By default the offset sync is stored in an internal Kafka topic in the source Kafka cluster. The topic is named mm2-offset-syncs.*[***TARGET CLUSTER ALIAS***]*.internal.

The offset sync is a compact topic, which means that at least the latest mapping for each replicated topic partition is kept in the topic, but some old values with older offsets can also be present in the topic until rotation and cleanup.

With the offset.lag.max property you can influence how often a new offset sync should be created. If you create it often, your mapping will be more accurate, but if your consumer groups can lag behind, it increases the chance that offset translation will be unsuccessful. For more information, see MirrorCheckpointConnector on page 10.

MirrorSourceConnector example

The following is an example `KafkaConnector` resource that represents an instance of the `MirrorSourceConnector.`

This connector example replicates the partitions of the test topic from a Kafka cluster that is aliased as target. The topics property (topic filter) must match the topics property in the `MirrorCheckpointConnector` that is part of the same replication flow.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
  tasksMax: 2
  config:
    topics: test
    source.cluster.alias: us-west
    source.cluster.bootstrap.servers: source-cluster-kafka-bootstrap.kafka:9
092
    target.cluster.alias: us-east
    target.cluster.bootstrap.servers: target-cluster-kafka-bootstrap.kafka
:9092
    refresh.topics.interval.seconds: 10
    key.converter: org.apache.kafka.connect.converters.ByteArrayConverter
    value.converter: org.apache.kafka.connect.converters.ByteArrayConverter
```

Related Information

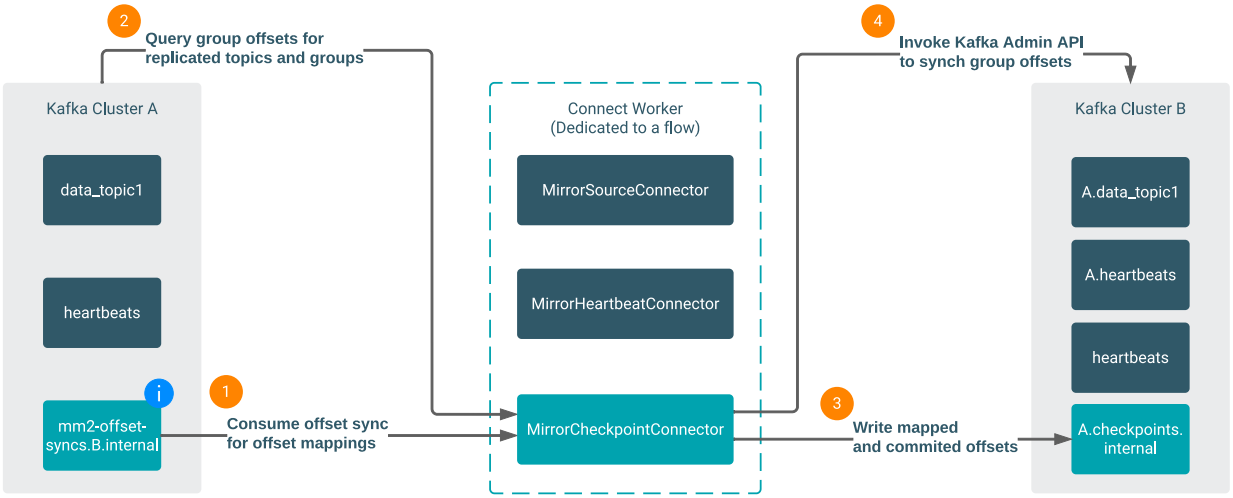
Enabling exatcly-once semantics for replication flows

Replication connector configurations

MirrorCheckpointConnector

The MirrorCheckpointConnector is responsible for replicating the committed group offsets between the source and target clusters. The offsets are translated based on the offset sync topic managed by the MirrorSourceConnector. The translated offsets are periodically applied to the consumer group offsets in the target cluster by the MirrorCheckpointConnector.

Figure 7: MirrorCheckpointConnector



Important: The `MirrorCheckpointConnector` provides at-least-once guarantee for consumers. The guarantee stays at-least-once even when exactly-once semantics (EOS) is enabled for the data replication. Additionally, the MirrorCheckpointConnector only works in the context of a single replication flow. If there are other messages being produced into the replicated topics in the target cluster, the checkpointing cannot account for those messages. The checkpoints do not guarantee anything for the messages produced by other clients or replicated in different flows (for example, in an aggregator architecture).

MirrorCheckpointTask

The `MirrorCheckpointTask` is created by the `MirrorCheckpointConnector`. It is responsible for executing consumer group offset synchronization. It uses a producer for writing translated offsets to the target cluster. This producer is managed by the Kafka Connect framework, all the other clients are managed by the task itself.

Each task receives its assignment from the `MirrorCheckpointConnector` as a list of consumer groups. The offsets of the assigned consumer groups are periodically queried for the replicated topic partitions through an admin client, get translated based on the offset syncs topic and are synchronized in the target cluster consumer offsets.

Note: Since the `MirrorCheckpointTask` instances share the load over consumer groups, there is no point setting the tasksMax property higher than the number of consumer groups that need to be replicated.

Automatically apply consumer offsets in target cluster

Since sync.group.offsets.enabled is set to true by default, the offsets are periodically applied to the consumer groups in the target cluster automatically, no additional connector configuration is needed in order to make it work. The frequency of this process is controlled by the sync.group.offsets.interval.seconds property of the `MirrorCheckpointConnector`, which defaults to 60 seconds. Having this feature enabled is a must in any replication flow that you set up.

Checkpoints

The consumer groups can only be updated in the target cluster if there are no active members in the group at that time. To make sure the consumer offset information is always replicated to the target cluster, checkpoints are also created in the target cluster in an internal topic called *[***SOURCE CLUSTER ALIAS***]*.checkpoints.internal. This topic contains the information about each replicated consumer group where they left consuming in the source cluster.

Guarantees

Checkpointing guarantees that replicated group checkpoints are monotonic. This is true as long as the upstream committed offset of the group is monotonic. This means that checkpointing prioritizes monotonicity over emitting new checkpoint records.

The difficulty in performing checkpointing consistently is the offset translation. Checkpointing relies on the offset syncs to perform offset translation from upstream to downstream offsets. The offset syncs are backed by a compact topic which ensures that the last offset sync of a partition is always kept in the topic, but it is also possible that older offset syncs are also present.

Checkpointing utilizes these older offset syncs to perform offset translation on a wide range of upstream offsets. The width of this range solely depends on the number and age of the offset sync records that are present in the backing topic. In a best-case scenario, offset translation is performed on a wide range of offsets if the offset sync history is present. In a worst-case scenario, offset translation can only happen based on the last offset sync record.

Any consumer groups which lag behind the translatable range are not checkpointed. To fine-tune the worst-case guarantees, configure the offset.lag.max property for the `MirrorSourceConnector`. Configuring this property influences how often a new offset sync should be created for each partition.

MirrorCheckpointConnector example

The following is an example `KafkaConnector` resource that represents an instance of the `MirrorCheckpointConnector`. This connector example replicates the offsets of the testgroup consumer group related to partitions of the test topic. The topics property (topics filter) must match the topics property in the `MirrorSourceConnector` that is part of the same replication flow.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-checkpoint-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.mirror.MirrorCheckpointConnector
  tasksMax: 2
  config:
    topics: test
    groups: testgroup
    source.cluster.alias: us-east
    source.cluster.bootstrap.servers: source-cluster-kafka-bootstrap.kafka
:9092
    target.cluster.alias: us-west
    target.cluster.bootstrap.servers: target-cluster-kafka-bootstrap.kafka:9
092
    refresh.groups.interval.seconds: 10
    emit.checkpoints.interval.seconds: 10
    key.converter: org.apache.kafka.connect.converters.ByteArrayConverter
    value.converter: org.apache.kafka.connect.converters.ByteArrayConverter
```

Related Information

Replication connector configurations

MirrorHeartbeatConnector

The MirrorHeartbeatConnector is responsible for creating a heartbeats topic in a chosen cluster and to periodically produce heartbeats into the heartbeats topic.

The purpose of this is to always have at least a single topic that can be replicated. To achieve this, Cloudera recommends configuring the connector to create the heartbeats topic in the source cluster and let the `MirrorSourceConnector` to replicate it.

This functions as a reliable smoke test for the replication flow. This can be also helpful in edge cases where a `MirrorClient` is used that requires having heartbeats to discover the replication flows and upstream clusters. Configuring this connector is not required to deploy replication flow, but it is recommended.

MirrorHeartbeatTask

The `MirrorHeartbeatTask` is created by the `MirrorHeartbeatConnector`. It is responsible for producing heartbeats into the configured cluster's heartbeats topic. It uses a producer for writing heartbeats to the heartbeats topic. This producer is managed by the Kafka Connect framework, all the other clients are managed by the task itself. There is always a single `MirrorHeartbeatTask` instance created by a `MirrorHeartbeatConnector`.

Creating heartbeats topic in source cluster

The heartbeats topic is created in the cluster specified in the target.cluster.* properties of the `MirrorHeartbeatConnector`. If you choose to use this connector you must ensure that the `target.cluster.*` properties refer to the source cluster in the replication flow.

With a setup like this, you will have a topic that is automatically replicated and acts as a reliable smoke test for your replication flow. To configure the connector to create the heartbeats in the source cluster, you override the producer client managed by Kafka Connect to connect and produce to the source Kafka cluster.

Cloudera also recommends configuring target.cluster.bootstrap.servers to point to the source cluster. In this context, the target means where to produce heartbeats, not the replication flow's target. This property is required by other internal connector clients other than the producer.

MirrorHeartbeatConnector example

The following is an example `KafkaConnector` resource that represents an instance of the `MirrorHeartbeatConnector`.

This configuration example contains the client overrides and other settings that configure the connector to produce heartbeats to the source cluster.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-heartbeat-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.mirror.MirrorHeartbeatConnector
  tasksMax: 2
  config:
    source.cluster.alias: us-west
    target.cluster.alias: us-east
    target.cluster.bootstrap.servers: source-cluster-kafka-bootstrap.kafk
a:9092
    producer.override.bootstrap.servers: source-cluster-kafka-bootstrap.k
afka:9092
    key.converter: org.apache.kafka.connect.converters.ByteArrayConverter
    value.converter: org.apache.kafka.connect.converters.ByteArrayConverter
```

Related Information

Replication connector configurations

Connector task and load balancing

Learn how tasks are distributed and how load is balanced in replication flows.

A typical production Kafka Connect cluster consists of multiple workers. Whenever a replication flow is configured, the replication connectors that make up a replication flow create their own tasks.

If you choose to deploy all three replication connectors, then the connectors will create one or more `MirrorSourceTasks`, one or more `MirrorCheckpointTasks`, as well as a single `MirrorHeartbeatTask`.

The connectors and tasks are assigned to the Kafka Connect workers in a round robin fashion.

When the Kafka Connect workers already have their assigned connectors and tasks, there can be changes that result in triggering a rebalance which means tasks and connectors should be reassigned. These changes can be the following.

- A worker joins or leaves the group (membership change)
- The filter for replicated topics or groups changes and the number of tasks changes because of this

The reassignment of connectors and tasks are done in a cooperative and incremental manner. This allows for the majority of the work to continue without interruption. Based on Kafka Connect group membership changes, the tasks can also be moved between workers.

Replication connector configurations

Learn what configuration properties and prefixes are available for replication connectors.

Connector properties

The replication connectors support various properties. Supported properties of the connectors can be categorized into groups. There are a number of common properties that are accepted by all three connectors. Additionally, each connector has a unique set of properties that it supports.

The following table lists each property group and provides a link to the relevant reference section of the Apache Kafka documentation.

Table 2: Replication connector properties

Property group	Reference in Apache Kafka documentation
Common source connector properties	Source Connector Configs | Kafka
Common replication connector properties	MirrorMaker Common Configs | Kafka
`MirrorSourceConnector` properties	MirrorMaker Source Configs | Kafka
`MirrorCheckpointConnector` properties	MirrorMaker Checkpoint Configs | Kafka
`MirrorHeartbeatConnector` properties	MirrorMaker Heartbeat Configs | Kafka

Important: The sync.group.offsets.enabled property of the `MirrorCheckpointConnector` is set to false by default in Apache Kafka. However, in Cloudera Streams Messaging Operator for Kubernetes this property is set to true by default to ensure that consumer offsets are automatically synchronized whenever possible. This is done using the Kafka Admin API. Automatic consumer offset synchronization is only possible if the consumer group is empty at that moment in the target

Configuration prefixes

All three replication connectors use multiple Kafka clients (producer, consumer, admin client) internally. These clients are created by the connector itself and are not managed by the Kafka Connect framework. You can provide common client configurations to these internal clients on different levels by using configuration prefixes.

There can be two types of variables in the prefix:

- *[***CLUSTER TYPE***]* – This variable specifies the type of the cluster. This variable is either source or target.
- *[***CLIENT TYPE***]* – This variable specifies the type of client. This variable can be *PRODUCER*, *CONSUMER*, or *ADMIN*.

[CLUSTER TYPE***].cluster prefix**

Properties that use the *[***CLUSTER TYPE***]*.cluster prefix are applied to all clients used for connecting to the cluster type specified in the prefix.

For example, the following configuration ensures that all internal clients that interact with the source cluster will use the same bootstrap server.

```
source.cluster.bootstrap.servers=localhost:9092
```

[CLIENT TYPE***]* prefix**

Properties that use the *[***CLIENT TYPE***]* prefix are applied to all clients of the type specified in the prefix regardless of what type of cluster (source or target) they connect to. This prefix has a higher precedence than the *[***CLUSTER TYPE***]*.cluster prefix.

For example, the following configuration ensures that all clients that admin clients use the same bootstrap server.

```
admin.bootstrap.servers=localhost:9092
```

[CLUSTER TYPE***].[***CLIENT TYPE***]* prefix**

Properties that use the *[***CLUSTER TYPE***].[***CLIENT TYPE***]* prefix are applied to all client types specified in the prefix that connect to the cluster type specified in the prefix. This prefix has a higher precedence than the *[***CLIENT TYPE***]* prefix.

For example, the following configuration ensures that all producers that connect to the target cluster use the same bootstrap server.

```
target.producer.bootstrap.servers=localhost:9093
```

Recommended configurations for offset syncs and checkpoints topics

Cloudera recommends the following configuration for the offset synchs and checkpoints topics.

Offset syncs

When the checkpoint connector starts and restarts, it reads the offset syncs topic from the beginning. To ensure that offset translations remain accurate, you need to determine how long you must keep offset sync records uncompacted. Cloudera recommends setting min.compaction.lag.ms equal to the maximum tolerable consumer lag (in milliseconds).

Checkpoints

Similarly to the offset syncs topic, the checkpoints topic is also read from the beginning during connector initialization. With this topic, the goal is to keep at least one checkpoint message uncompressed for each subscription. You achieve this by calculating an appropriate value for segment.bytes. This is because records in the active segment are not compacted. Cloudera recommends the following formula for calculation.

```
segment.bytes = max(REPLICATED_GROUPS*SUBSCRIBED_PARTITIONS*CHECKPOINT_MSG_S
IZE, 64 MB)
```

Calculate CHECKPOINT_MSG_SIZE with the following formula.

```
avarage topic name length + avarage group id length + 4 bytes for partition
id
```

When evaluating configuration, Cloudera used 16 B topic names and 16 B group IDs, which comes out to a 36 B total.

The formula defines 64Mb minimum value to avoid too frequent segment rolls. You can adjust this value as required. Additionally, if you want to have more frequent compaction, you can experiment with setting min.cleanable.dirty. ratio to a lower value. The default is 0.5.