Cloudera Streams Messaging Operator 1.2.0

Deploying and Configuring Kafka Replications

Date published: 2024-06-11
Date modified: 2024-12-02

https://docs.cloudera.com/

Legal Notice

Contents

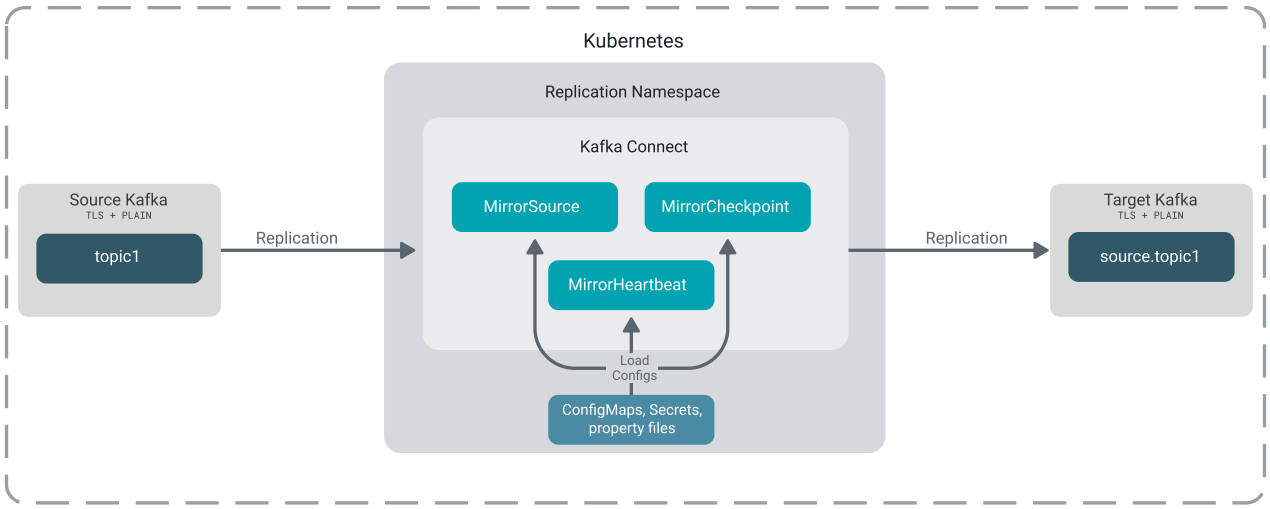
Deploying a replication flow

You deploy a replication flow between two clusters by deploying a Kafka Connect cluster and an instance of each replication connector (MirrorSourceConnector, MirrorCheckpointConnector, and MirrorHeartbeatConnector). Additionally, you create various ConfigMaps and Secrets that store configuration required for replication.

About this task

The following steps walk you through how you can create a replication flow between two secured Kafka clusters. Both Kafka and Kafka Connect are deployed in Kubernetes.

Figure 1: Replication flow between two secured Kafka clusters



The Kafka Connect cluster that you set up must be a new cluster and must be dedicated to the replication flow. Reusing an existing cluster that is running other connectors or using the same cluster for multiple replication flows is not recommended.

Replication of Kafka data as well as other replication-related tasks are carried out by the replication connectors. These are the `MirrorSourceConnector`, `MirrorCheckpointConnector`, and `MirrorHeartbeatConnector`.

Deploying an instance of the `MirrorSourceConnector` and `MirrorChekpointConnector` are mandatory. Deploying `MirrorHeartbeatConnector` is optional.

The connectors load their connection-related configuration from various `Secrets`, `ConfigMaps`, as well as property files.

This example uses the `DefaultReplicationpolicy`, but provides instructions on what connector properties you need to add if you want to use the `IdentityReplicationPolicy` (prefixless replication).

These steps assume that the two Kafka clusters have TLS encryption and PLAIN authentication enabled. Replication can be configured for any other type of security as well, but you will need to change the appropriate security configurations.

For example, assume that one of the clusters does not use PLAIN, but a different authentication method. In a case like this, you must collect and specify the configuration properties appropriate for that authentication method. Configuration related to security is stored in `ConfigMaps` and `Secrets` that you will be setting up.

Tip: These steps use documentation replaceables to refer to the various resources that you need to set up for replication. For example, the namespace you create will be referred to as *[***REPLICATION NS***]*. Pay attention to the replaceables if you are copying examples. You will need to replace many values in the configuration of your resources.

Before you begin

- Strimzi is installed. The Strimzi Cluster Operator is running. See installation Installation.
- You have identified the two Kafka clusters that you want to replicate data between.

  The clusters can be any type of Kafka cluster running on any platform. These steps assume that both Kafka instances are running in Kubernetes and were deployed with Cloudera Streams Messaging - Kubernetes Operator.
- Resource examples in these steps use various features and configurations available in Kafka Connect. Familiarity with the following is recommended.
  - Deploying Kafka Connect clusters
  - Configuration providers
  - Adding external configuration to Kafka Connect worker pods
  - Configuring connectors
  - Replication overview
  - Replication connectors and connector architecture

Procedure

1. Collect the following for both source and target Kafka clusters.
   - Bootstrap servers
   - TLS truststore/crt
   - TLS truststore password
   - PLAIN credentials

   The configurations you collect here will be specified in the `Secrets` and `ConfigMaps` and the `KafkaConnect` resource that you create in the following steps.
2. Create a namespace.

   ```
   kubectl create namespace [***REPLICATION NS***]
   ```

   You deploy all resources required for the replication flow in this namespace.
3. Create a `Secret` containing credentials for the Docker registry where Cloudera Streams Messaging - Kubernetes Operator artifacts are hosted.

   ```
   kubectl create secret docker-registry [***SECRET NAME***] \
     --docker-server [***REGISTRY***] \
     --docker-username [***USERNAME***] \
     --docker-password [***PASSWORD***] \
     --namespace [***REPLICATION NS***]
   ```

   - *[***SECRET NAME***]* must be the same as the name of the `Secret` containing registry credentials that you created during Strimzi installation.
   - Replace *[***REGISTRY***]* with the server location of the Docker registry where Cloudera Streams Messaging - Kubernetes Operator artifacts are hosted. If your Kubernetes cluster has internet access, use cont ainer.repository.cloudera.com. Otherwise, enter the server location of your self-hosted registry.
   - Replace *[***USERNAME***]* and *[***PASSWORD***]* with credentials that provide access to the registry. If you are using container.repository.cloudera.com, use your Cloudera credentials. Otherwise, enter credentials providing access to your self-hosted registry.

4. Create a `ConfigMap` and two `Secrets` for the **target Kafka cluster**.

   These resources store configuration that provides access to the target Kafka cluster.

   a) Create a `ConfigMap` that contains the non-sensitive configuration properties of the target Kafka cluster.

   ```
   kubectl create configmap [***TARGET CONFIGMAP***] \
     --from-literal=alias=[***TARGET CLUSTER ALIAS***] \
     --namespace [***REPLICATION NS***]
   ```

   This `ConfigMap` does not need to include connection related properties like the bootstrap server. These connection properties will be sourced from the Kafka Connect worker's (cluster) property file. Sourcing them from the workers' property file is possible because Kafka Connect will depend on the target Kafka cluster. You can use this `ConfigMap` to store other reusable properties.

   b) Create a `Secret` containing the PLAIN password to use when connecting to the target Kafka cluster.

   ```
   kubectl create secret generic [***TARGET PASSWORD SECRET***] \
     --from-literal=pass=[***PASSWORD***] \
     --namespace [***REPLICATION NS***]
   ```

   c) Create a `Secret` that contains the TLS Certificate Authority (CA) certificate of the target Kafka cluster.

   ```
   kubectl create secret generic [***TARGET CERT SECRET***] \
     --from-file=ca.crt=[***PATH TO CA CERT***] \
     --namespace [***REPLICATION NS***]
   ```

   Tip: If the target Kafka cluster was deployed with Cloudera Streams Messaging - Kubernetes Operator, a `Secret` containing the certificate will already exist in the namespace of the target cluster. The secret containing the certificate is called *[***TARGET KAFKA CLUSTER NAME***]-cluster-ca-cert*. You can extract the certificate from this secret and deploy it in the new namespace.

5. Create a `ConfigMap` and a `Secret` for the source Kafka cluster.

   These resources store configuration that provide access to the source Kafka cluster.

   a) Create a `Secret` that contains the truststore file, the truststore password, and JAAS configuration of the source Kafka cluster.

   ```
   kubectl create secret generic [***SOURCE SECRET***] \
     --from-literal=ssl.truststore.password=[***TRUSTSTORE PASSWORD***] \
     --from-file=truststore.jks=[***TRUSTSTORE FILE***] \
     --from-literal=sasl.jaas.config='org.apache.kafka.common.securi
   ty.plain.PlainLoginModule required username="[***USERNAME***]" passwo
   rd="[***PASSWORD***]";' \
     --namespace [***REPLICATION NS***]
   ```

   b) Create a `ConfigMap` that contains non-sensitive configuration properties of the source Kafka cluster.

   This `ConfigMap` will contain the cluster alias, connection properties, and any other reusable properties.

   ```
   kubectl create configmap [***SOURCE CONFIGMAP***] \
     --from-literal=alias=[***SOURCE CLUSTER ALIAS***] \
     --from-literal=bootstrap.servers=[***SOURCE KAFKA
    BOOTSTRAP***].[***SOURCE KAFKA NAMESPACE***]:[***PORT***] \
     --from-literal=security.protocol=SASL_SSL \
     --from-literal=sasl.mechanism=PLAIN \
     --from-literal=ssl.truststore.location=/mnt/[***VOLUME NAME***]/tru
   ststore.jks \
     --namespace [***REPLICATION NS***]
   ```

   You will attach the truststore as a volume in a later step. Note down the value you specify for *[***VOLUME NAME***]*. You will need to provide it in the `KafkaConnect` resource.

6. Create a `ConfigMap` that stores configuration related to replication.

   This `ConfigMap` will store configuration that is shared by the connectors that you will deploy. This map is created to single source configuration that is common across the connectors.

   This example creates a `ConfigMap` that defines a single property, topics, which specifies what topics should be replicated. In this example, all test.* topics are added for replication.

   ```
   kubectl create configmap [***REPLICATION CONFIGMAP***] \
     --from-literal=topics="test.*" \
     --namespace [***REPLICATION NS***]
   ```

   This `ConfigMap` is referred to in the following steps as *[***REPLICATION CONFIGMAP***]*.

   Tip: The replication policy used by the connectors is configured in this `ConfigMap`. If not specified, the `DefaultReplicationPolicy` is used. Add the following property to the `ConfigMap` if you want to use a different replication policy.

   ```
   replication.policy.class=[***POLICY CLASSNAME***]
   ```

   The value of this property is the fully qualified class name of the replication policy. If you want to use the `IdentityReplicationPolicy` (prefixless replication), add `org.apache.kafka.connect.mirror.IdentityReplicationPolicy` as the value.

   If you choose to configure the policy, you will need to reference the property in the configuration of the connectors.

7. Deploy a Kafka Connect cluster.

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaConnect
   metadata:
     name: [***CONNECT CLUSTER NAME***]
     namespace: [***REPLICATION NS***]
     annotations:
       strimzi.io/use-connector-resources: "true"
   spec:
     version: 3.8.0.1.2
     replicas: 3
     bootstrapServers: [***TARGET KAFKA BOOTSTRAP***].[***TARGET KAFKA
    NAMESPACE***]:[***PORT***]
     tls:
       trustedCertificates:
         - secretName: [***TARGET CERT SECRET***]
           certificate: ca.crt
     authentication:
       type: plain
       username: [***USERNAME***]
       passwordSecret:
         secretName: [***TARGET PASSWORD SECRET***]
         password: pass
     template:
       pod:
         volumes:
           - name: [***VOLUME NAME***]
             secret:
               secretName: [***SOURCE SECRET***]
               items:
                 - truststore.jks
       connectContainer:
         volumeMounts:
           - name: [***VOLUME NAME***]
             mountPath: /mnt/[***VOLUME NAME***]
     config:
   ```

```
    group.id: [***CONNECT CLUSTER NAME***]-consumer-group
    offset.storage.topic: [***CONNECT CLUSTER NAME***]-offsets-topic
    config.storage.topic: [***CONNECT CLUSTER NAME***]-config-topic
    status.storage.topic: [***CONNECT CLUSTER NAME***]-status-topic
    config.storage.replication.factor: -1
    offset.storage.replication.factor: -1
    status.storage.replication.factor: -1
    config.providers: cfmap,secret,file
    config.providers.cfmap.class: io.strimzi.kafka.KubernetesConfigMapCon
figProvider
    config.providers.secret.class: io.strimzi.kafka.KubernetesSecretConf
igProvider
    config.providers.file.class: org.apache.kafka.common.config.provider.
FileConfigProvider
```

Notice the following about this resource configuration.

- The names specified in metadata.name, group.id, and *storage.topic follow a consistent naming convention.

  Cloudera recommends adding cluster aliases to these names as well as using prefixes and postfixes. For example, your cluster name can be repl-uswest-useast. Where repl is a prefix, useast and uswest are the aliases. The group ID can be repl-uswest-useast-consumer-group, where repl-uswest-useast is the name of the cluster, -consumer-group is a postfix.

  The prefixes and postfixes like repl, -consumer-group, -offsets-topic, -config-topic, -status-topic are merely suggestions.
- bootstrapServers is set to the target Kafka cluster's bootstrap.

  That is, this Kafka Connect cluster will depend on the target Kafka cluster. This is a must have for correct replication architecture. The *.[***TARGET KAFKA NAMESPACE***]* postfix is only required because this example assumes that the Kafka cluster is running in Kubernetes.
- trustedCertificates references a `Secret` you created in a previous step, which contains the CA certificate of the target cluster.
- template.pod.volumes defines volume which contain the truststore file from a `Secret` you created in a previous step.
- template.connectContainer.volumeMounts mount s the volume which contains the truststore file.
- *.storage.replication.factor properties are set to -1.

  This means that these internal topics are created with the default replication factor configured in the Kafka cluster that this Kafka Connect cluster depends on (the target Kafka cluster).
- The config.providers.* properties enable various configuration providers.

  These are necessary as the connectors you set up in a later step load configuration from various external resources using these configuration providers.

8. Create a `Role` and `RoleBinding`.

The `KubernetesConfigMapConfigProvider` and `KubernetesSecretConfigProvider` configuration providers specified in the `KafkaConnect` resource in the previous step, require additional access rights to access the `ConfigMaps` and `Secrets`, respectively. Creating the below `Role` and `RoleBinding` is required to grant them these privileges.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: connector-configuration-role
  namespace: [***REPLICATION NS***]
rules:
  - apiGroups: [""]
    resources: ["secrets"]
    resourceNames: ["[***SOURCE SECRET***]"]
    verbs: ["get"]
  - apiGroups: [""]
```

```
    resources: ["configmaps"]
    resourceNames: ["[***SOURCE CONFIGMAP***]", "[***TARGET
 CONFIGMAP***]", "[****REPLICATION CONFIGMAP**]"]
    verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: connector-configuration-role-binding
  namespace: [***REPLICATION NS***]
subjects:
  - kind: ServiceAccount
    name: [***CONNECT CLUSTER NAME***]-connect
roleRef:
  kind: Role
  name: connector-configuration-role
  apiGroup: rbac.authorization.k8s.io
```

- The resource names you specify in rules.apiGroups.resourceNames are the names of the `ConfigMap` and `Secret` resources you created for the source and target Kafka clusters in a previous step.
- The `ServiceAccount` name is fixed and follows a pattern.

  The name is the Kafka Connect cluster name postfixed with -connect. This name is fixed because the `ServiceAccount` is generated and named by the Strimzi Cluster Operator. That is, the -connect postfix, is not user defined, ensure that you do not change it.

9. Enable data replication by deploying an instance of `MirrorSourceConnector`.

`MirrorSourceConnector` requires access to both the source and target Kafka clusters. Therefore, it requires access to all configurations you set up in previous steps. Additionally, some extra configuration is required.

Configuration required to connect to the target cluster is sourced from the Kafka Connect worker's property file.

Configuration required to connect to the source cluster is sourced from the `ConfigMap`, `Secret`, and truststore volume you set up for the source cluster in a previous step.

Other configurations such as the target cluster alias is sourced from the `ConfigMap` you set up for the target cluster in a previous step.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: mirror-source-connector
  namespace: [***REPLICATION NS***]
  labels:
    strimzi.io/cluster: [***CONNECT CLUSTER NAME***]
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
  tasksMax: 3
  config:
    key.converter: org.apache.kafka.connect.converters.ByteArrayConverter
    value.converter: org.apache.kafka.connect.converters.ByteArrayConver
ter
    refresh.topics.interval.seconds: 10
    topics: ${cfmap:[***REPLICATION NS***]/[***REPLICATION
 CONFIGMAP***]:topics}

    #replication.policy.class: ${cfmap:[***REPLICATION
 NS***]/[***REPLICATION CONFIGMAP***]:replication.policy.class}

    # Source cluster configurations - sourced from configmap, secret and
 volume
    source.cluster.alias: ${cfmap:[***REPLICATION NS***]/[***SOURCE
 CONFIGMAP***]:alias}
```

```
    source.cluster.bootstrap.servers: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:bootstrap.servers}
    source.cluster.security.protocol: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:security.protocol}
    source.cluster.sasl.mechanism: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:sasl.mechanism}
    source.cluster.sasl.jaas.config: ${secret:[***REPLICATION
 NS***]/[***SOURCE SECRET***]:sasl.jaas.config}
    source.cluster.ssl.truststore.location: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:ssl.truststore.location}
    source.cluster.ssl.truststore.password: ${secret:[***REPLICATION
 NS***]/[***SOURCE SECRET***]:ssl.truststore.password}

    # Target cluster configurations - mostly sourced from the Connect wor
ker config
    target.cluster.alias: ${cfmap:[***REPLICATION NS***]/[***TARGET
 CONFIGMAP***]:alias}
    target.cluster.bootstrap.servers: ${file:/tmp/strimzi-connect.proper
ties:bootstrap.servers}
    target.cluster.security.protocol: ${file:/tmp/strimzi-connect.proper
ties:security.protocol}
    target.cluster.sasl.mechanism: ${file:/tmp/strimzi-connect.propertie
s:sasl.mechanism}
    target.cluster.sasl.jaas.config: ${file:/tmp/strimzi-connect.propertie
s:sasl.jaas.config}
    target.cluster.ssl.truststore.location: ${file:/tmp/strimzi-connect.
properties:ssl.truststore.location}
    target.cluster.ssl.truststore.password: ${file:/tmp/strimzi-connect.
properties:ssl.truststore.password}
```

- Uncomment the replication.policy.class property if you added this property to *[***REPLICATION CONFIGMAP***]*. This property configures what replication policy is used for replication.

10. Enable consumer group offset synchronization by deploying an instance of `MirrorCheckpointConnector`.

`MirrorCheckpointConnector` requires access to both the source and target clusters. Additionally, it requires the same replication policy configuration, topic filters, and offset synchronization configurations as used by `MirrorSourceConnector`.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: mirror-checkpoint-connector
  namespace: [***REPLICATION NS***]
  labels:
    strimzi.io/cluster: [***CONNECT CLUSTER NAME***]
spec:
  class: org.apache.kafka.connect.mirror.MirrorCheckpointConnector
  tasksMax: 3
  config:
    key.converter: org.apache.kafka.connect.converters.ByteArrayConverter
    value.converter: org.apache.kafka.connect.converters.ByteArrayConver
ter
    refresh.groups.interval.seconds: 10
    sync.group.offsets.enabled: true
    topics: ${cfmap:[***REPLICATION NS***]/[***REPLICATION
 CONFIGMAP***]:topics}
    groups: test.*
    #replication.policy.class: ${cfmap:[***REPLICATION
 NS***]/[***REPLICATION CONFIGMAP***]:replication.policy.class}

    # Source cluster configurations - sourced from configmap, secret and
 volume
```

```
    source.cluster.alias: ${cfmap:[***REPLICATION NS***]/[***SOURCE
 CONFIGMAP***]:alias}
    source.cluster.bootstrap.servers: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:bootstrap.servers}
    source.cluster.security.protocol: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:security.protocol}
    source.cluster.sasl.mechanism: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:sasl.mechanism}
    source.cluster.sasl.jaas.config: ${secret:[***REPLICATION
 NS***]/[***SOURCE SECRET***]:sasl.jaas.config}
    source.cluster.ssl.truststore.location: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:ssl.truststore.location}
    source.cluster.ssl.truststore.password: ${secret:[***REPLICATION
 NS***]/[***SOURCE SECRET***]:ssl.truststore.password}

    # Target cluster configurations - mostly sourced from the Connect wor
ker config
    target.cluster.alias: ${cfmap:[***REPLICATION NS***]/[***TARGET
 CONFIGMAP***]:alias}
    target.cluster.bootstrap.servers: ${file:/tmp/strimzi-connect.proper
ties:bootstrap.servers}
    target.cluster.security.protocol: ${file:/tmp/strimzi-connect.proper
ties:security.protocol}
    target.cluster.sasl.mechanism: ${file:/tmp/strimzi-connect.propertie
s:sasl.mechanism}
    target.cluster.sasl.jaas.config: ${file:/tmp/strimzi-connect.propertie
s:sasl.jaas.config}
    target.cluster.ssl.truststore.location: ${file:/tmp/strimzi-connect.
properties:ssl.truststore.location}
    target.cluster.ssl.truststore.password: ${file:/tmp/strimzi-connect.
properties:ssl.truststore.password}
```

- Uncomment the replication.policy.class property if you added this property to *[***REPLICATION CONFIGMAP***]*. This property configures what replication policy is used for replication.
- The sync.group.offsets.enabled property is true by default. As a result, setting this property explicitly to true is not necessary. The property is explicitly set to true in this example to highlight Cloudera requirements. Using this feature is a must in any replication flow that you set up.

11. Enable heartbeating by deploying an instance of `MirrorHeartbeatConnector`.

`MirrorHeartbeatConnector` is responsible for creating minimal replication traffic in the flow. Because of this, the Connector needs access to the source cluster, but configured as if it was the target cluster. This means that you need to provide the source cluster configurations with the producer.override. and target.cluster. prefixes.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: mirror-heartbeat-connector
  namespace: [***REPLICATION NS***]
  labels:
    strimzi.io/cluster: [***CONNECT CLUSTER NAME***]
spec:
  class: org.apache.kafka.connect.mirror.MirrorHeartbeatConnector
  tasksMax: 1
  config:
    key.converter: org.apache.kafka.connect.converters.ByteArrayConverter
    value.converter: org.apache.kafka.connect.converters.ByteArrayConvert
er

    #replication.policy.class: ${cfmap:[***REPLICATION
 NS***]/[***REPLICATION CONFIGMAP***]:replication.policy.class}

    # Cluster aliases
```

```
    source.cluster.alias: ${cfmap:[***REPLICATION NS***]/[***SOURCE
 CONFIGMAP***]:alias}
    target.cluster.alias: ${cfmap:[***REPLICATION NS***]/[***TARGET
 CONFIGMAP***]:alias}
    # Source cluster configurations configured as target - sourced from
 configmap, secret and volume
    target.cluster.bootstrap.servers: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:bootstrap.servers}
    target.cluster.security.protocol: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:security.protocol}
    target.cluster.sasl.mechanism: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:sasl.mechanism}
    target.cluster.sasl.jaas.config: ${secret:[***REPLICATION
 NS***]/[***SOURCE SECRET***]:sasl.jaas.config}
    target.cluster.ssl.truststore.location: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:ssl.truststore.location}
    target.cluster.ssl.truststore.password: ${secret:[***REPLICATION
 NS***]/[***SOURCE SECRET***]:ssl.truststore.password}

    # Source cluster configurations configured as producer override - sou
rced from configmap, secret and volume
    producer.override.bootstrap.servers: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:bootstrap.servers}
    producer.override.security.protocol: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:security.protocol}
    producer.override.sasl.mechanism: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:sasl.mechanism}
    producer.override.sasl.jaas.config: ${secret:[***REPLICATION
 NS***]/[***SOURCE SECRET***]:sasl.jaas.config}
    producer.override.ssl.truststore.location: ${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:ssl.truststore.location}
    producer.override.ssl.truststore.password: ${secret:[***REPLICATION
 NS***]/[***SOURCE SECRET***]:ssl.truststore.password}
```

- Uncomment the replication.policy.class property if you added this property to *[***REPLICATION CONFIGMAP***]*. This property configures what replication policy is used for replication.

Configuring prefixless replication

By default, replication flows you deploy use the DefaultReplicationPolicy, which prefixes the replicated topic names in the target Kafka cluster. If you want replicated topics to retain their original name, you configure your replication flow to use IdentityReplicationPolicy instead.

You configure replications flows to use the `IdentityReplicationPolicy` with the `replication.policy.class` connector property. This property specifies the class name of the replication policy to use. You add the property to the configuration of the replication connectors that you deploy for each replication flow. That is, you need to add the property to the configuration of `MirrorSourceConnector`, `MirrorHeartBeatConnector`, and `MirrorCheckpointConnector`.

The value of this property must be set to the same replication policy in each connector instance that is deployed for a replication flow.

Instead of hardcoding the replication policy in each connector configuration, Cloudera recommends that you add the value to a `ConfigMap` that stores properties that are common to the connectors, and load the value using the `KubernetesConfigMapConfigProvider`.

```
#...
kind: KafkaConnector
spec:
  class: org.apache.kafka.connect.mirror.MirrorHeartbeatConnector
```

```
  config:
    replication.policy.class: ${cfmap:[***REPLICATION NS***]/[***REPLICATION
CONFIGMAP***]:replication.policy.class}
```

A configuration setup like this enables you to specify the replication policy centrally.

Important: The `KubernetesConfigMapConfigProvider` must be enabled in the Kafka Connect cluster where you deploy your connectors. Additionally, an appropriate `Role` and `RoleBinding` is required for the configuration provider to work.

Related Information

Deploying a replication flow
Replication policies
Configuration providers

Checking the state of data replication

Learn how to check the current state of data replication.

About this task

The `MirrorSourceConnector` keeps track of its progress in the source cluster using the Kafka Connect framework. Kafka Connect allows checking and manipulating the source offsets of the connectors. You can check the current state of data replication by extracting source offsets and comparing them with the end offsets of replicated partitions.

Before you begin

These steps use the connect_shell.sh and kafka_shell.sh Cloudera Streams Messaging - Kubernetes Operator tools. Ensure that these tools are available to you. Running kafka_shell.sh is only necessary if your source Kafka cluster is deployed with Cloudera Streams Messaging - Kubernetes Operator. See Using kafka_shell.sh and Using connect_shell.sh.

Procedure

1. Use connect_shell.sh to exec into a Kafka Connect admin pod of the replicator Kafka Connect cluster.

   ```
   ./connect_shell.sh --namespace=[***REPLICATION NAMESPACE***] --clust
   er=[***CONNECT CLUSTER NAME***]
   ```

2. Use the `GET /connectors/CONNECTOR/offsets endpoint` of the Kafka Connect REST API to extract source offsets.

   ```
   curl -s $CONNECT_REST_URL/connectors/[***CONNECTOR NAME***]/offsets
   ```

   *[***CONNECTOR NAME***]* is the name of the `MirrorSourceConnector` instance.

   Note: The frequency of updates of the offset values returned by this command is controlled by the offset.flush.interval.ms property of the `MirrorSourceConnector`. The interval is 60 seconds by default.

3. In the source cluster, use the kafka-get-offsets.sh Kafka tool to extract the end offsets of the replicated partitions.

```
bin/kafka-get-offsets.sh --bootstrap-server [***SOURCE CLUSTER
 HOST***]:[***PORT***] --topic "test.*"
```

   - The kafka-get-offsets.sh tool accepts a regex string as the topic filter, but does not accept a list of regexes. To specify multiple regex expressions in a single command (as a single regex string), chain expressions together with pipes (|).

     ```
     --topic "test.*|abc.*|zxc.*"
     ```

   - If the source Kafka cluster is a Cloudera Streams Messaging - Kubernetes Operator Kafka cluster, use kafk a_shell.sh to run the kafka-get-offsets.sh tool

4. Compare extracted end offsets with the source offsets extracted in Step 2 on page 13.

Configuring data replication offsets

Learn how you can configure and modify what offset the MirrorSourceConnector replicates form.

By default, `MirrorSourceConnector` replicates data from the start of the source topics, and keeps track of the progress by committing source offsets into the Kafka Connect framework.

This behavior can be modified in the following ways.

- Starting data replication from the latest offset for new partitions.
- Manually setting exact offsets for specific source partitions.

Caution: Cloudera advises caution when modifying what offsets replication starts at. Modifying the start offset might affect the guarantees provided for data replication.

Replicating from the latest offset for new partitions

To replicate data from the latest offset, you configure auto.offset.reset property for the source consumer in the MirrorSourceConnector.

```
#...
kind: KafkaConnector
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
  config:
    source.consumer.auto.offset.reset: latest
```

With this configuration, all new partitions (without a committed offset) are replicated from the latest offset. Cloudera recommends applying this configuration under special circumstances only as it violates the at-least-once guarantee of data replication.

This example uses the source.consumer. prefix. That is, auto.offset.reset is specifically set for the source consumer in the connector, which is the consumer connecting to the source cluster.

Related Information

auto.offset.reset | Kafka

Replication connector configurations

Manually setting exact offsets for specific source partitions

In some situations, it might be necessary to rewind the replication and reprocess records, or fast forward and skip some records. To do this, you can manipulate the exact offsets per partition and change the state of the replication.

Before you begin

- The connect_shell.sh tool is available to you. See Using connect_shell.sh.
- Ensure that you are familiar with the process of checking replication state. See Checking the state of data replication .

Procedure

1. Stop the `MirrorSourceConnector`.

   To do this, set the spec.state property to stopped in the `KafkaConnector` resource of the connector.

   ```
   #...
   kind: KafkaConnector
   spec:
     class:org.apache.kafka.connect.mirror.MirrorSourceConnector
     state: stopped
   ```

2. Use connect_shell.sh to get administrative access to the Connect REST API.

   ```
   connect_shell.sh --namespace=[***CONNECT CLUSTER NAMESPACE***] \
     --cluster=[***CONNECT CLUSTER NAME***]
   ```

3. Create a payload to manipulate the source offsets with the offset management endpoints of the Kafka Connect REST API.

   The payload is connector specific. For example, the structure for the `MirrorSourceConnector` is the following.

   ```
   {"offsets":[{"partition":{"cluster":"[***SOURCE CLUSTER ALIAS***]","p
   artition":0,"topic":"[***SOURCE TOPIC NAME***]"},"offset":{"offset
   ":[***OFFSET***]}}]}
   ```

   You can specify multiple partitions in the structure. Additionally, you can set offsets.offset to null to delete the offset for a specific partition. Alternatively, you can also delete all offsets with the `DELETE /connectors/{connector}/offsets` endpoint.

   ```
   curl -X DELETE $CONNECT_REST_URL/connectors/[***CONNECTOR NAME***]/offsets
   ```

4. Submit the payload.

   ```
   curl --data 'PAYLOAD' -H "Content-Type: application/json" -X PATCH $CONN
   ECT_REST_URL/connectors/[***CONNECTOR NAME***]/offsets
   ```

5. Resume the `MirrorSourceConnector`.

   To do this, set the spec.state property to running in the `KafkaConnector` resource of the connector.

   ```
   #...
   kind: KafkaConnector
   spec:
     class:org.apache.kafka.connect.mirror.MirrorSourceConnector
     state: running
   ```

Enabling exactly-once semantics for replication flows

You enable exactly once semantics (EOS) for replication flows by configuring EOS in the KafkaConnect resource. Optionally, Cloudera recommends that you set the source consumer isolation level in your MirrorSourceConnector to read_committed.

About this task

The progress of `MirrorSourceConnector` is tracked by periodically committing the offsets of the processed messages. If the connector fails, uncommitted messages are reprocessed after the connector starts running again.

Using EOS, source connectors are able to handle offset commits and message produces in a single transaction. This either results in a successful operation where messages are produced to the target topic along with offset commits, or a rollback of the whole operation. EOS is enabled in the `KafkaConnect` resource with the exactly.once.source.support property.

If transactional producers are writing messages to the source topic, Cloudera recommends that you filter records from the aborted transactions out from the replicated data. Otherwise, aborted transactions are marked as committed in the target, which results in invalid data. This is configured in your `MirrorSourceConnector` with the isolation.level property. You set the property to read_committed.

Important: Due to the periodic nature of checkpointing, EOS does not apply to failover and failback scenarios. Duplicate messages are expected.

Procedure

1. Enable EOS in your KafkaConnect resource.

   Configuration differs for newly deployed resources and existing resources.

   ### For New resources

   Set exactly.once.source.support to enabled.

   ```
   #...
   kind: KafkaConnect
   spec:
     config:
       exactly.once.source.support: enabled
   ```

   ### For Existing resources

   a. Set exactly.once.source.support to preparing.

   ```
   #...
   kind: KafkaConnect
   spec:
     config:
       exactly.once.source.support: preparing
   ```

   b. Wait until configuration changes are applied and worker pod rolling restart finishes. The restart begins in the next reconciliation loop.

   ```
   kubectl get pods --namespace [***NAMESPACE***] --watch
   ```

   c. Set exactly.once.source.support to enabled.

2. Set isolation.level in your `MirrorSourceConnector`.

```
#...
kind: KafkaConector
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
  config:
    source.consumer.isolation.level: read_committed
```

This example uses the source.consumer. prefix. That is, isolation.level is specifically set for the source consumer in the connector, which is the consumer connecting to the source cluster.

Note:

Setting the isolation.level comes with caveats. If the connector reaches a message written by an uncommitted transaction, it stops reading until the transaction is either committed or rolled back. This can cause significant lag in replication. You can limit this by applying an appropriate application timeout, however, the timeout you set will depend on the application and use case.

Related Information

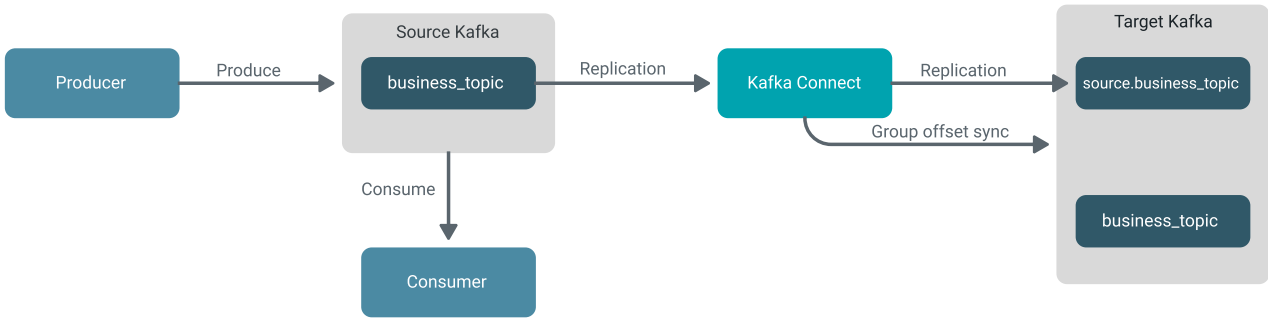
Replication connector configurations

Performing a failover or failback

Performing a failover or failback

Learn about failover and failback operations that you can perform between two Kafka clusters that have data replication enabled. Performing a failback or failover operation enables you to migrate consumer and producer applications between Kafka clusters. These operations are typically performed after a disaster event or in migration scenarios.

Figure 2: Failover/failback setup



The producer and consumer applications both connect to the source cluster, while a Kafka Connect cluster is configured to replicate the business topics and synchronize the group offsets into the target cluster. Note that the busi ness_topic in the target cluster is not created by replication. Instead you create this topic in preparation for the failover or failback scenario.

There are multiple types of failover and failback operations that you can carry out. Which one you perform depends on your scenario and use case. The failover and failback types are as follows.

Continuous and controlled failover

A continuous and controlled failover is carried out when all applications and services are working as expected, but you want to move workloads from one cluster to another. This type of failover is

continuous because applications are moved continuously to the target without a cutoff. This failover can be performed rapidly and comes with minimal service disruptions.

This failover type works with `DefaultReplicationPolicy` only.

Controlled failover with a cutoff

A controlled failover with a cutoff is carried out when all applications and services are working as expected. The cutoff means that producers are stopped for the duration of the failover and consumer traffic is exhausted in the source cluster.

Compared to a continuous failover, this failover is more complex, but does not rely on group offset syncing, and can also guarantee message ordering for consumers even across the failover.

This failover type works with both the `DefaultReplicationPolicy` and `IdentityReplicationPolicy`.

Failover on disaster

A failover on disaster is carried out when you encounter a disaster scenario where your source cluster becomes unavailable. A failover on a disaster simply consists of reconfiguring and restarting your client applications to use the target Kafka cluster.

Controlled failback

A controlled failback is the same as a failover operation but in a reverse order. That is, you move clients back to their original cluster. A failback operation assumes that you already performed a failover operation.

Performing a continuous and controlled failover

Learn how to perform a continuous and controlled failover between Kafka clusters that have data replication enabled.

About this task

A continuous and controlled failover is carried out when all applications and services are working as expected. That is, there is no disaster scenario. Instead you make an executive decision to move your workload from the source cluster to the target cluster so that you can stop the source cluster, either temporarily or permanently, without disrupting applications.

The failover is continuous because applications can be continuously moved to the target cluster without a strict cutoff. Because of this, the failover can be performed rapidly with minimal service disruptions.

Throughout this process, replication of Kafka data is not stopped, ensuring that no data is lost.

Important: This failover type works with `DefaultReplicationPolicy` only.

Before you begin

Ensure that you are familiar with the process of checking replication state. See Checking the state of data replication .

Procedure

1. Fail over consumers.
   a) Gracefully stop consumers.

      This allows the consumers to commit their offsets to the source Kafka cluster of their latest state.
   b) Wait for the replication to successfully synchronize the latest offsets.

      Calculate wait time based on the intervals configured in the emit.checkpoints.interval.seconds (default 60 seconds) and synch.group.offsets.interval.seconds (default 60 seconds) properties of the `MirrorCheckpointConnector`. The wait time is the sum of these properties multiplied by two.

      ```
      wait time = 2 * (emit.checkpoints.interval.seconds + sync.group.offsets.
      interval.seconds)
      ```
   c) Configure consumers to connect to the target cluster and to consume from both the replicated and the active (prefixless) topic in the target cluster.

      There is a possibility that consumers still did not process all messages from the source cluster. To pick up the remaining data, they need to consume from the prefixed replica topics as well as from the active (prefixless) topic so that they also see the new data produced to the target cluster when the producers are failed over.

      Important: This also means that message ordering is not guaranteed, as consumers now consume from two separate topics. Replicated messages might get mixed with messages produced into the target cluster.
   d) Start consumers.
2. Fail over producers.
   a) Gracefully stop producers.
   b) Configure producers to connect to the target cluster.

      Producers can safely produce to the exact same topics without any name changes as the replicated data is stored in a prefixed topic.

      Note: Producers must never be configured to produce to the replicated (prefixed) topics.
   c) Start producers.
3. Wait for the replication fo finish replicating all data that was produced to the cluster.

   At this point, it is still possible that not all records are migrated to the target cluster. Check the state of the replication to ensure that all records are fully replicated.
4. Stop the source cluster.

Performing a controlled failover with a cutoff

Learn how to perform a controlled failover with a cutoff between Kafka clusters that have data replication enabled.

About this task

A controlled failover with a cutoff is carried out when all applications and services are working as expected. That is, there is no disaster scenario. Instead you make an executive decision to stop the source cluster, either temporarily or permanently, and move your workload from the source to the target cluster.

The failover has a cutoff because producers are stopped for the duration of the failover. Additionally, all consumer traffic is exhausted in the source cluster. This results in a longer disruption in client applications.

A controlled failover with a cutoff is a complex process, but does not rely on group offset syncing, and can also guarantee message ordering for consumers even across the failover.

Important: This failover type works with both the `DefaultReplicationPolicy` and `IdentityReplicationPolicy`.

Before you begin

Ensure that you are familiar with the process of checking replication state. See Checking the state of data replication .

Procedure

1. Gracefully stop producers.

   This stops the ingress traffic, allowing all consumers to fully read all data.
2. Monitor the consumers and the replication, and wait until all data is read.
   - To monitor the consumer applications, use the kafka-consumer-groups.sh Kafka tool with the --describe option. Wait until the lag becomes 0.

     Note: There might be consumer groups of old or inactive applications for which the lag will never become 0. You will have to decide whether to follow up on those cases or ignore them for the cutoff.
   - To check replication state, compare source offsets and the offsets of the `MirrorSourceConnector`. Wait until replication fully catches up with business data.
3. Gracefully stop consumers.
4. Gracefully stop replication.
5. If using the `IdentityReplicationPolicy`: Reset the offsets of all consumer groups to the latest offset in the target cluster.

   This ensures that old data is not consumed after the failover. Steps 2 on page 20 and 3 on page 20 already ensure that all old data has been successfully consumed.
6. Configure the producers to connect to the target cluster.

   Producers can safely produce to the exact same topics without any name changes.

   Note: Producers must never be configured to produce to the replicated (prefixed) topics.
7. Start producers.
8. Configure consumers to connect to the target cluster.

   Consumers can safely consume from the exact same topics without any name changes.
   - If `DefaultReplicationPolicy` and topic prefixing is used, the replicated data is separated into the prefixed topic. This only affects new consumers, as old consumers were previously allowed to completely consume old data from the source cluster.
   - If `IdentityReplicationPolicy` is used, all old data was written into the topic already, since Step 2 on page 20 and 3 on page 20 ensure that there will be no more old data coming into the topic. Only newly produced data is written into it after the failover.
9. Start consumers.
10. Stop the source cluster.

Performing a failover on disaster

Learn how to perform a failover operation in a disaster scenario between Kafka clusters that have data replication enabled.

In a disaster scenario where your source cluster becomes unavailable, you cannot perform a failover in a controlled manner. In a case like this, a failover operation simply involves reconfiguring and restarting all client applications to use the target Kafka cluster.

In a failover on disaster, the data and the group offsets replicated up until the failure can be used to continue processing.

In a disaster scenario with an uncontrolled stop and crash event, some messages that were successfully accepted in the source cluster might **not** be replicated to the target cluster. This means that some messages will not be accessible

for consumers, even though they were successfully produced into the source cluster. This is due to the fact that replication is asynchronous and may lag behind the source data. This is also true when exactly-once semantics (EOS) is enabled for data replication.

Performing a controlled failback

Learn about performing failback operations between Kafka clusters that have data replication enabled.

A controlled failback operation is the same as a failover operation, but in reverse order. That is, you move your clients back to their original Kafka cluster. Typically this means moving from the target cluster of the replication to the source cluster of the replication some time after a failover operation was performed.

To complete a failback operation, follow the steps for any of the failover operations, but in reverse order. However, take note of the following caveats.

- A failback assumes a bidirectional replication, as data produced into the target Kafka is not present in source, so the data needs replication.
- You cannot perform a failback operation if the `IdentityReplicationPolicy` is in use.

  This is because the `IdentityReplicationPolicy` does not allow bidirectional replication over the same topics as the topic names are not altered during replication. A bidirectional replication setup with `IdentityReplicationPolicy` would result in a replication loop where topics are infinitely replicated between source and target clusters. If using the `IdentityReplicationPolicy`, after a failover you must stop and remove your previous replication setup and reconfigure it again in the reverse direction before you can be ready to failback.
- The `MirrorCheckpointConnector` and group offset synchronization only function in the context of a single replication flow. Mapping offsets back to the original topic is not supported.

  This means that any progress made by consumers in the target Kafka cluster over the replicated (prefixed) topics, aka the old data, is lost. There is a high likelihood that consumers will reprocess old data after the failback. You can avoid a scenario like this if the initial failover operation that you carry out is a controlled failover with a cutoff. A failover with a cutoff guarantees that all old data was already consumed.

Using Single Message Transforms in replication flows

In Cloudera Streams Messaging - Kubernetes Operator you can apply Single Message Transforms (SMT) in the connectors that make up a replication flow. Configuring an SMT chain enables you to transform the Kafka records during replication. This collection of examples demonstrates how you can transform keys and values as well as metadata in Kafka records during replication.

The following examples on key and value transformation are simple examples that are meant to demonstrate the use of the SMT framework in data replication. They might not be directly applicable or appropriate for all use cases in a production environment. Specifically, these examples use the `JsonConverter` with schemaless data which is handled as a Map by the Kafka Connect framework. You can replace the `JsonConverter` for any other converters to handle data with schema depending on your data formats present in your use case.

While it is possible to modify the topic name of a record using the SMT framework, these types of transformations should not be used in replication flows. Modifying the topic name can block replication policy and data replication as a whole.

Transforming the key or value of the Kafka records in replication flows

When the `MirrorSourceTask` provides Kafka records for the Kafka Connect framework, it provides them with keys and values as only bytes that have the BYTES schema. This is true even if your data inside the blob is structured data, for example JSON.

The result of this is that you can not directly manipulate the data, because most SMT plugins rely on the Kafka Connect internal data format and its schema. In this context, the `BYTES` schema is meaningless. You can use the `ConvertFromBytes` plugin with an appropriate converter to be able to run manipulations on structured data.

The following example converts each replicated message value into JSON format with the `ConvertFromBytes` plugin. This example assumes that the message values contain JSON data.

```
#...
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
  config:
    transforms: ConvertFromBytes
    transforms.ConvertFromBytes.type: com.cloudera.dim.kafka.connect.transf
orms.ConvertFromBytes$Value
    transforms.ConvertFromBytes.converter: org.apache.kafka.connect.json.Jso
nConverter
    transforms.ConvertFromBytes.converter.schemas.enable: false
```

Adding additional transformations

You can put any transformation after the `ConvertFromBytes` plugin. The following example replaces two fields in the record values with the `ReplaceField` plugin.

```
#...
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
  config:
    transforms: ConvertFromBytes,ReplaceField
    transforms.ConvertFromBytes.type: com.cloudera.dim.kafka.connect.trans
forms.ConvertFromBytes$Value
    transforms.ConvertFromBytes.converter: org.apache.kafka.connect.json.Js
onConverter
    transforms.ConvertFromBytes.converter.schemas.enable: false
    transforms.ReplaceField.type: org.apache.kafka.connect.transforms.Replac
eField$Value
    transforms.ReplaceField.renames: name:replaced_name,age:replaced_age
```

After applying your transformation, you have to consider how to create bytes from your structured JSON. There is a required converter in the connector configuration which is applied on the records just before providing them for the Kafka connect framework's producer.

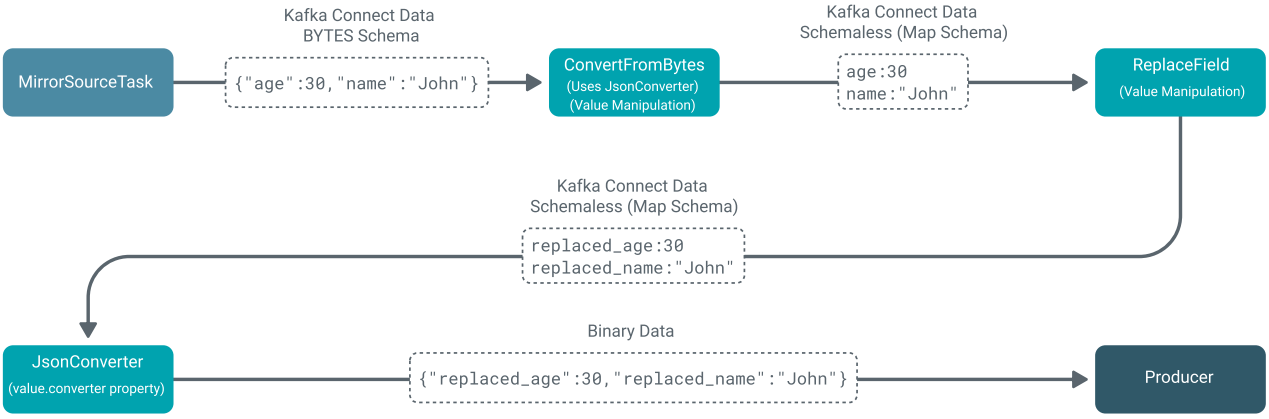
This conversion happens after the data goes through your SMT chain. In this example, you can simply use `JsonConverter` as value converter, you do not need additional SMT steps to convert values back.

```
#...
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
```

```
  config:
    value.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter.schemas.enable: false
    key.converter: org.apache.kafka.connect.converters.ByteArrayConverter
```

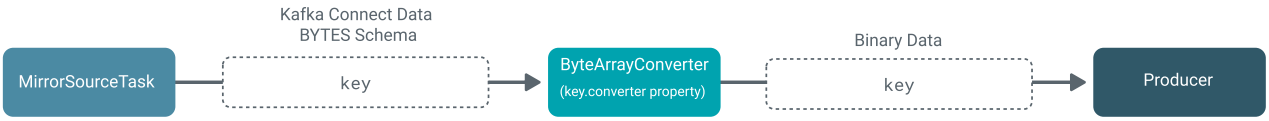
Once both the SMT chain and the converters in the connector configuration are applied, you will get a value transformation chain like the following.

Figure 3: Value conversion using ConvertFromBytes and ReplaceField



The keys were not converted to JSON, so you can use `ByteArrayConverter` on them, only the values need to be converted from JSON to byte array. The key transformation chain is as follows.

Figure 4: Key conversion using ByteArrayConverter



Filtering data using SMTs

If your replication flow replicates topics with different data formats, a transformation chain like the one in the examples above will fail when trying converting data of the wrong type.

A typical example of that happens when your replication flow uses a `MirrorHeartbeatConnector`. The heartbeats topic contains records that can not be converted into JSON. Since heartbeat records are automatically replicated by the `MirrorSourceConnector`, you will encounter exceptions during data conversion

In cases like this, you must use predicates to filter heartbeat records from the transformation chain.

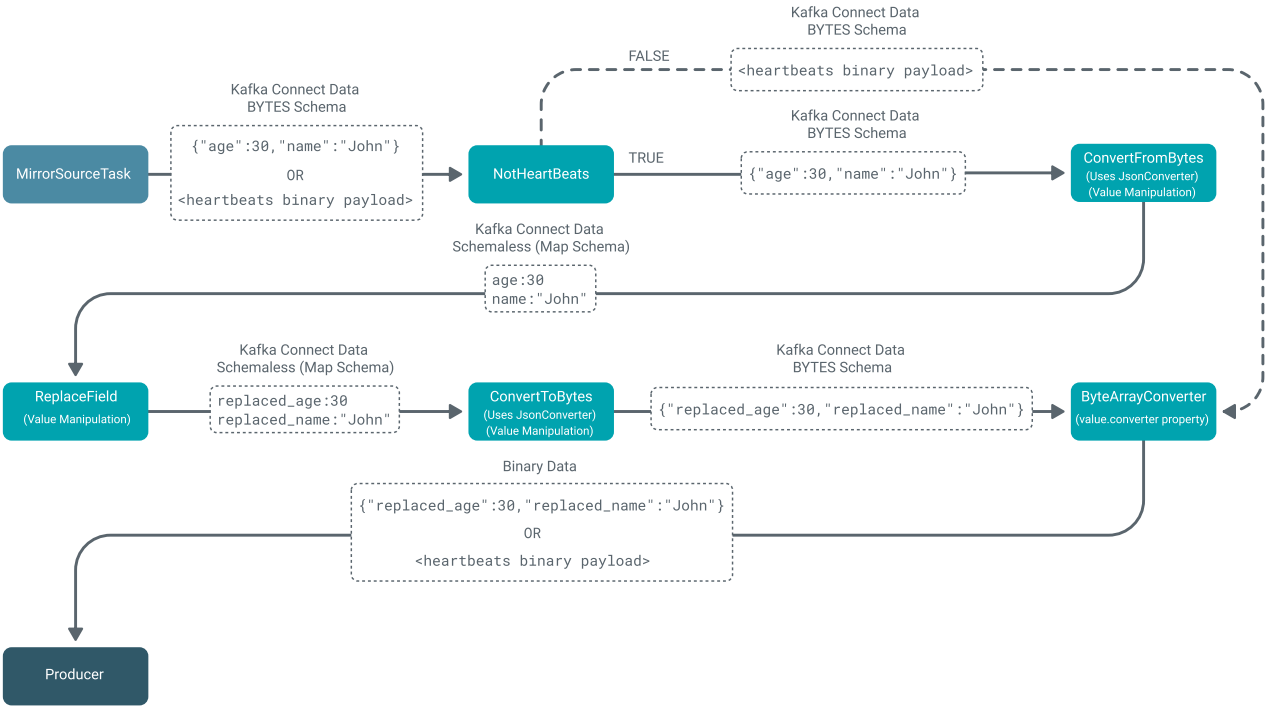
```
#...
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
  config:
    transforms: ConvertFromBytes,ReplaceField,ConvertToBytes
    transforms.ConvertFromBytes.type: com.cloudera.dim.kafka.connect.transf
orms.ConvertFromBytes$Value
```

```
    transforms.ConvertFromBytes.converter: org.apache.kafka.connect.json.Jso
nConverter
    transforms.ConvertFromBytes.converter.schemas.enable: false
    transforms.ReplaceField.type: org.apache.kafka.connect.transforms.Rep
laceField$Value
    transforms.ReplaceField.renames: name:replaced_name,age:replaced_age
    transforms.ConvertToBytes.type: com.cloudera.dim.kafka.connect.transfor
ms.ConvertToBytes$Value
    transforms.ConvertToBytes.converter: org.apache.kafka.connect.json.JsonC
onverter
    transforms.ConvertToBytes.converter.schemas.enable: false
    predicates: NotHeartbeats
    predicates.NotHeartbeats.type: org.apache.kafka.connect.transforms.pred
icates.TopicNameMatches
    predicates.NotHeartbeats.pattern: ^(?!(.+\.)?heartbeats).*$
    transforms.ConvertFromBytes.predicate: NotHeartbeats
    transforms.ReplaceField.predicate: NotHeartbeats
    transforms.ConvertToBytes.predicate: NotHeartbeats
    key.converter: org.apache.kafka.connect.converters.ByteArrayConverter
    value.converter: org.apache.kafka.connect.converters.ByteArrayConverter
```

Since heartbeats records are not converted into JSON, they remain byte arrays. All the other record values, however, will be converted to JSON.

To unify the data format of the record values, you have to convert your non heartbeat record values back to byte arrays, using `ConvertToBytes`. After applying your configuration, all record values become byte arrays, so you can use `ByteArrayConverter`as the final converter. Key conversion in this case is the same as in the previous example.

Figure 5: Value conversion with a predicate that filters heartbeat records



Transforming metadata of Kafka records in replication flows

Unlike transformation of keys or values, you can transform the metadata (headers, timestamps and so on) in Kafka records without any preliminary conversion. That is, you do not need to create a chain with multiple transforms or predicates. You can simply use a single plugin like `InsertHeader`.

The following transformation chain example adds smt-header-key=smt-header-value as a fixed header to all of the replicated records using the `InsertHeader` plugin.

```
#...
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
  config:
    transforms: InsertHeader
    transforms.InsertHeader.header: smt-header-key
    transforms.InsertHeader.type: org.apache.kafka.connect.transforms.Insert
Header
    transforms.InsertHeader.value.literal: smt-header-value
```

Related Information

Single Message Transforms

MirrorHeartbeatConnector

Transformations | Kafka

Replication monitoring and diagnostics

If you already installed Prometheus and Grafana, you can monitor your replication flows. When configuring Kafka cluster replication, replication connectors provide some additional metrics which are worth monitoring besides the underlying Kafka Connect cluster metrics.

For the complete list of replication connector related metrics, *Monitoring Geo-Replication* in the Apache Kafka documentation. In order to be able to access these metrics, you must configure the Connect JMX metrics exporter.

You can use the included kafka-connect-replication-metrics.yaml example file to create a Kafka Connect cluster which exports the necessary metrics. This example exports both replication related metrics as well as metrics about the underlying Kafka Connect cluster, which can be useful when monitoring replication flows.

Before applying the example file, you need to modify spec.bootstrapServers which should point to your target Kafka cluster. After deploying the replication connectors into this Kafka Connect cluster, the metrics will be available with the kafka_connect_mirror_ prefix. You can change the prefix by specifying different renaming rules in the JMX exporter configuration.

The following are some metrics that can be of interest when monitoring a replication:

- kafka_connect_mirror_mirrorsourceconnector_byte_rate – Measures the Bytes/sec in replicated records through the source connector.
- kafka_connect_mirror_mirrorsourceconnector_record_age_ms – Time duration between record timestamp in the source topic and the time when the source connector handles the record.
- kafka_connect_mirror_mirrorsourceconnector_replication_latency_ms – Time duration it takes records to propagate from source to target. The difference between record timestamp in the source topic and the time when the producer receives ack from the target cluster that the record was written successfully.
- kafka_connect_source_task_source_record_active_count – The number of records that this task has consumed from the source but not yet produced to the target.

- kafka_connect_connector_task_offset_commit_avg_time_ms – Time duration that this task takes to commit its offsets to the target.
- kafka_consumer_fetch_manager_records_lag – Consumer lag which in the context of the replication indicates whether the consumer in the source connector can keep up with the rate records are produced in the source.

A sample Grafana dashboard is provided in strimzi-kafka-connect-replication.json among the examples which configures visualizations of the above metrics. It can serve as a basis for monitoring replication flows. You can even use it for multiple replication flows, as you can choose the namespace and connect cluster which you want to monitor. You might want to tailor it to your specific needs by modifying or extending this dashboard.

The prometheus-rules.yaml contains some replication related alerting rules under the replication group. You might want to configure the exact thresholds based on your specific needs or define your own rules. It is also recommended to configure the alerting rules for Kafka Connect (connect group).

Related Information

Monitoring Geo-Replication

Cloudera Archive

Advanced replication use cases and examples

A collection of advanced replication flow examples for different use cases leveraging the configuration options available with Kafka Connect-based replication.

When using Kafka Connect-based replication, you have full control over the configuration and deployment of the replication connectors (`MirrorSourceConnector`, `MirrorCheckpointConnector`, and `MirrorHeartbeatConnector`). As a result, you can fine-tune your replication set up and deploy the following types of replication flows.

- A replication flow that uses multiple `MirrorSourceConnector` instances

  This type of replication flow can be used if you want to replicate different topics, located in the same source cluster, using different replication configurations.
- A replication flow that is deployed on multiple Kafka Connect clusters

  This type of replication flow can be used to separate replication work across available resources. It enables you to have dedicated resources for specific replication workloads.
- A bidirectional and prefixless replication flow

  This type of replication flow can be used if your business requires a bidirectional replication setup that also uses prefixless replication. That is, you require a deployment that has topics that are actively produced, consumed, and bidirectionally replicated at the same time with the topic names remaining unchanged during replication.

The following provides instructions and examples that demonstrate how you can set up each of these replication flows.

Tip: The following instructions assume that you are familiar with the procedure of deploying a replication flow. Review Deploying a replication flow before continuing.

Deploying a replication flow that uses multiple MirrorSourceConnector instances

Learn how to deploy a replication flow that uses multiple MirrorSourceConnector instances to carry out replication of a single source cluster. Using multiple MirrorSourceConnectors enables you to replicate different topics using different replication configurations.

About this task

Using Kafka Connect-based replication, you can deploy replication flows that use multiple instances of the `MirrorSourceConnector`. Using multiple connector instances, in turn, enables you to fine tune how data is replicated and apply different replication configurations to different topics that are in the same source Kafka cluster. For example you can configure the following.

- Different compression type per connector
- Different replication factor per connector
- Different single message transforms per connector
- Different offset sync frequency

The `MirrorSourceConnector` instances are deployed in a single Kafka Connect cluster. Additionally, all connector instances connect to the same source and target clusters. All other configurations related to replication can be customized in each connector instance.

The following example demonstrates a configuration where a header is added to a subset of replicated topics. This is done by creating two `MirrorSourceConnector` instances and configuring one of them to use Single Message Transforms (SMT).

Steps

To deploy replication flow with multiple `MirrorSourceConnector` instances, follow the steps in the Deploying a replication flow with the following changes.

1. Create multiple `ConfigMaps` that store configuration related to each replication.

   In this example, a total of three `ConfigMaps` are created. One that stores configuration properties common to both `MirrorSourceConnector` instances, and an additional two that store configuration properties specific to each `MirrorSourceConnector` instance.

   Important: Ensure that there is no overlap in the topics that you configure as the value of the topics property in your `ConfigMaps`. The topics that you add to the topics property are the topics that will be replicated by your `MirrorSourceConnector` instances. A single topic must be only replicated by a single `MirrorSourceConnector` instance.

```
kubectl create configmap [***COMMON REPLICATION CONFIGMAP***] \
  --from-literal=replication.policy.class="org.apache.kafka.connect.mirror
.DefaultReplicationPolicy" \
  --namespace [***REPLICATION NS***]
```

```
kubectl create configmap [***FIRST REPLICATION CONFIGMAP***] \
  --from-literal=topics="test.*" \
  --namespace [***REPLICATION NS***]
```

```
kubectl create configmap [***SECOND REPLICATION CONFIGMAP***] \
  --from-literal=topics="prod.*" \
  --from-literal=transformHeaderKey="my_header_key"
  --from-literal=transformHeaderValue="my_header_value"
  --namespace [***REPLICATION NS***]
```

2. Deploy multiple `MirrorSourceConnector` instances.

   Configure each connector instance as needed. Reference appropriate `ConfigMaps`.

   In this example, two connector instances are deployed. Both instances reference *[***COMMON REPLICATION CONFIGMAP***]* created in the previous step. Additionally, each connector instance also pulls configuration from their respective `ConfigMaps`.

   The second connector instance includes an SMT chain that applies a transformation on the replicated records. The SMT chain adds a header to each record replicated by the second connector instance.

   For MirrorSourceConnector 1

   ```
   #...
   kind: KafkaConnector
   metadata:
     name: first-mirror-source-connector
   spec:
     class: org.apache.kafka.connect.mirror.MirrorSourceConnector
     config:
       topics: ${cfmap:[***REPLICATION NS***]/[***FIRST REPLICATION
    CONFIGMAP***]:topics}

       # use common replication policy in both connectors
       replication.policy.class: ${cfmap:[***REPLICATION NS***]/[***COMMON
    REPLICATION CONFIGMAP***]:replication.policy.class}
   ```

   For MirrorSourceConnector 2

   ```
   #...
   kind: KafkaConnector
   metadata:
     name: second-mirror-source-connector
   spec:
     class: org.apache.kafka.connect.mirror.MirrorSourceConnector
     config:
       topics: ${cfmap:[***REPLICATION NS***]/[***SECOND REPLICATION
    CONFIGMAP***]:topics}

       transforms: insertHeader
       transforms.insertHeader.type: org.apache.kafka.connect.transforms
   .InsertHeader
       transforms.insertHeader.header: ${cfmap:[***REPLICATION
    NS***]/[***SECOND REPLICATION CONFIGMAP***]:transformHeaderKey}
       transforms.insertHeader.value.literal: ${cfmap:[***REPLICATION
    NS***]/[***SECOND REPLICATION CONFIGMAP***]:transformHeaderValue}

       # use common replication policy in both connectors
       replication.policy.class: ${cfmap:[***REPLICATION NS***]/[***COMMON
    REPLICATION CONFIGMAP***]:replication.policy.class}
   ```

3. Configure the topics property of your `MirrorCheckpointConnector` instances so that they include all topics that are replicated.

   ```
   # ...
   kind: KafkaConnector
   metadata:
     name: mirror-checkpoint-connector
   spec:
     class: org.apache.kafka.connect.mirror.MirrorCheckpointConnector
     tasksMax: 3
     config:
   ```

```
   topics: ${cfmap:[***REPLICATION NS***]/[***FIRST REPLICATION
CONFIGMAP***]:topics},${cfmap:[***REPLICATION NS***]/[***SECOND
REPLICATION CONFIGMAP***]:topics}
   groups: [***CONSUMER GROUP NAME***]
```

Deploying a replication flow on multiple Kafka Connect clusters

Learn how to set up a replication flow that is deployed on two or more Kafka Connect clusters. A replication set up like this enables you to spread replication tasks across available clusters allowing you to have dedicated resources for critical workloads.

About this task

Using Kafka Connect-based replication, you can set up a replication flow that is spread across multiple Kafka Connect clusters. This is done by deploying multiple Kafka Connect clusters and an instance of each replication connector on each Kafka Connect cluster. Afterward, you configure your replication connectors to replicate data between the same source and target Kafka cluster pair.

In a setup like this, replication between a source and a target cluster is carried out by multiple sets of replication connectors. In a standard deployment, replication would be carried out by a single set of the replication connectors.

A replication set up that uses multiple Kafka connect clusters makes it possible for you to designate Kafka Connect clusters to handle specific types of workloads. For example, you can dedicate a cluster to handle replication of business critical topics, while a different cluster can handle the replication of other topics.

Steps

To deploy a replication flow with multiple Kafka Connect clusters, follow the steps in the Deploying a replication flow with the following changes.

1. Deploy multiple Kafka Connect clusters with differing configurations in separate namespaces.

   When deploying multiple Kafka Connect clusters, use a different namespace for each cluster. This way it is possible to separate the dedicated configurations and `Secrets` into the appropriate namespace. It also makes it possible to use namespaced limitations, such as `ResourceQuota`, or to apply different namespaced access control policies to each Kafka Connect cluster.

   For example, you can set up two Kafka Connect clusters. One to handle small workloads, which has fewer replicas and fewer resources allocated. Additionally, one for larger workloads, which has more replicas, higher resource allocation, as well as limits.

   For Small workload

```
#...
kind: KafkaConnect
metadata:
  name: small-workload
  namespace: [***SMALL WORKLOAD NS***]
spec:
  replicas: 3
  config:
    # Custom configuration
  resources:
    requests:
      memory: "1Gi"
      cpu: "1"
    limits:
      memory: "2Gi"
```

```
      cpu: "2"
```

For High workload

```
#...
kind: KafkaConnect
metadata:
  name: high-workload
  namespace: [***HIGH WORKLOAD NS***]
spec:
  replicas: 6
  config:
    # Custom configuration for higher workloads
  resources:
    requests:
      memory: "6Gi"
      cpu: "2"
    limits:
      memory: "12Gi"
      cpu: "4"
```

2. Create `ConfigMaps` that contain replication related properties in each of your namespaces.

   Assuming that you are deploying two Kafka Connect clusters, one for small and one for high workloads, you would create two `ConfigMaps` that contain the topic filter (topics property) configuration for your `MirrorSourceConnector` instances.

   For example, the `MirrorSourceConnector` instance in the high workload Kafka Connect cluster can replicate business critical topics with high expected volume. The connector running in the small workload cluster can handle the replication of less critical topics with less expected volume.

   Important: Ensure that there is no overlap in the topics that you configure as the value of the topics property in your `ConfigMaps`. The topics that you add to the topics property are the topics that will be replicated by your `MirrorSourceConnector` instances. A single topic must be only replicated by a single `MirrorSourceConnector` instance.

```
kubectl create configmap [***SMALL WORKLOAD CONFIGMAP***] \
  --from-literal=topics="low.*" \
  --namespace [***SMALL WORKLOAD NS***]
```

```
kubectl create configmap [***HIGH WORKLOAD CONFIGMAP***] \
  --from-literal=topics="high.*" \
  --namespace [***HIGH WORKLOAD NS***]
```

   Note: In addition to the dedicated `ConfigMaps`, creating another `ConfigMap` that contains common connector properties, like replication.policy, is recommended.

3. Deploy and configure your `MirrorSourceConnector` instances to replicate appropriate topics.

   For Small workload

```
#...
kind: KafkaConnector
metadata:
  name: mirror-source-connector
  namespace: [***SMALL WORKLOAD NS***]
  labels:
    strimzi.io/cluster: small-workload
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
  tasksMax: 3
```

```
  config:
    topics: ${cfmap:[***SMALL WORKLOAD NS***]/[***SMALL WORKLOAD
 CONFIGMAP***]:topics}
```

For High workload

```
#...
kind: KafkaConnector
metadata:
  name: mirror-source-connector
  namespace: [***HIGH WORKLOAD NS***]
  labels:
    strimzi.io/cluster: high-workload
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
  tasksMax: 3
  config:
    topics: ${cfmap:[***HIGH WORKLOAD NS***]/[***HIGH WORKLOAD
 CONFIGMAP***]:topics}
```

4. Configure the topics property of your `MirrorCheckpointConnector` instances so that they include all topics that are replicated.

For Small workload

```
# ...
kind: KafkaConnector
metadata:
  name: mirror-checkpoint-connector
  namespace: [***SMALL WORKLOAD NS***]
spec:
  class: org.apache.kafka.connect.mirror.MirrorCheckpointConnector
  tasksMax: 3
  config:
    topics: ${cfmap:[***SMALL WORKLOAD NS***]/[***SMALL WORKLOAD
 CONFIGMAP***]:topics}
    groups: [***CONSUMER GROUP NAME***]
```

For High workload

```
# ...
kind: KafkaConnector
metadata:
  name: mirror-checkpoint-connector
  namespace: [***HIGH WORKLOAD NS***]
spec:
  class: org.apache.kafka.connect.mirror.MirrorCheckpointConnector
  tasksMax: 3
  config:
    topics: ${cfmap:[***HIGH WORKLOAD NS***]/[***HIGH WORKLOAD
 CONFIGMAP***]:topics}
    groups: [***CONSUMER GROUP NAME***]
```

Deploying bidirectional and prefixless replication flows

Learn how to set up bidirectional and prefixless replication flows. A replication set up like is achieved with the use Single Message Transforms (SMT).

About this task

Using out of the box configurations and behavior, deploying a bidirectional flow that uses the prefixless replication policy (`IdentityReplicationPolicy`)) is not recommended. This is because the prefixless replication policy does not support replication loop detection. By default, a setup like this results in records being replicated infinitely between your source and target clusters.

However, using such a setup can make failover and failback scenarios easy. This is because bidirectional and prefixless setup requires minimal reconfiguration of Kafka clients when you failover or failback between Kafka clusters. You only need to reroute the client to connect to a different cluster. Reconfiguring clients to consume from or produce to differently named (prefixed) topics is not necessary.

A bidirectional and prefixless replication setup can be achieved when using Kafka Connect-based replication with the use of an SMT chain. By deploying an SMT chain on top of your replication flow, you can effectively filter replication loops while still having at-least-once guarantees.

Important: Even though a replication setup like this is possible without replication loops, checkpointing and consumer group offset synchronization do not work. This means that when your clients switch to a new cluster, it can happen that a considerable amount of duplicates are processed as consumers restart consumption from the beginning of the topic.

Steps

To configure a bidirectional and prefixless replication flow, follow the steps in Deploying a replication flow with the following changes.

1. Deploy two replication flows.
   - The replication setup must be bidirectional. One replication replicates data from cluster A to cluster B, the second replicates data from B to A.
   - The replication flows are configured to use prefixless replication. That is, replication.policy.class property of the `MirrorSourceConnector` and `MirrorHeartbeatConnector` instances are set to `org.apache.kafka.connect.mirror.IdentityReplicationPolicy`.
   - Skip the creation of `MirrorCheckpointConnector` instances. `MirrorCheckpointConnector` instances enable consumer group offset synchronization, which is not supported in a bidirectional setup. Creating these connectors is not necessary.
2. Configure your `MirrorSourceConnector` instances with an SMT chain that filters replication loops.

   The following is an example `MirrorSourceConnector` instance that uses the `Filter` and `InsertHeader` transforms as well as the `HasHeaderKey` predicate. When used in combination, these plugins provide a way to filter replication loops.

```
#...
kind: KafkaConnector
spec:
  class: org.apache.kafka.connect.mirror.MirrorSourceConnector
  tasksMax: 2
  config:
   transforms: FilterReplicatedFromTarget,InsertHeader
   transforms.FilterReplicatedFromTarget.type: org.apache.kafka.connect.tr
ansforms.Filter
   transforms.FilterReplicatedFromTarget.predicate: ReplicatedFromTarget
   transforms.InsertHeader.type: org.apache.kafka.connect.transforms.In
sertHeader
   transforms.InsertHeader.header: replicated-from-${cfmap:[***REPLICATION
 NS***]/[***SOURCE CONFIGMAP***]:alias}
   transforms.InsertHeader.value.literal: true
   predicates: ReplicatedFromTarget
   predicates.ReplicatedFromTarget.type: org.apache.kafka.connect.transfo
rms.predicates.HasHeaderKey
   predicates.ReplicatedFromTarget.name: replicated-from-${c
fmap:[***REPLICATION NS***]/[***TARGET CONFIGMAP***]:alias}
```

```
    emit.offset-syncs.enabled: false
```

Notice the following about this example.

- The `InsertHeader` transformation adds a new header for each replicated record. The header marks each record. This way the record include information on which cluster it came from.
- The `ReplicatedFromTarget` predicate returns true if a record already has the configured target cluster related replication header. In other words, it returns true if the record came from the target cluster earlier.
- The `FilterReplicatedFromTarget` transformation excludes records from replication for which the `ReplicatedFromTarget` predicate returns true. This filters replication loops because a record is never replicated back to a cluster where it was replicated from. This does not mean that some records are not consumed from source. All records are consumed. The records that would cause a replication loop are dropped.
- emit.offset-syncs.enabled is set to false to disable creation of the offset syncs internal topic. This is done because checkpointing is not supported in this set up. `MirrorCheckpointConnector` instances are not created. Creating this internal topic is unnecessary.

Note: Ensure that cluster aliases are consistent across your replication flows. For more information, see Replication aliases.