Strimzi Overview

Table of Contents

Chapter 1. Key features

Strimzi simplifies the process of running Apache Kafka in a Kubernetes cluster.

This guide is intended as a starting point for building an understanding of Strimzi. The guide introduces some of the key concepts behind Kafka, which is central to Strimzi, explaining briefly the purpose of Kafka components. Configuration points are outlined, including options to secure and monitor Kafka. A distribution of Strimzi provides the files to deploy and manage a Kafka cluster, as well as example files for configuration and monitoring of your deployment.

A typical Kafka deployment is described, as well as the tools used to deploy and manage Kafka.

1.1. Kafka capabilities

The underlying data stream-processing capabilities and component architecture of Kafka can deliver:

- Microservices and other applications to share data with extremely high throughput and low latency
- Message ordering guarantees
- Message rewind/replay from data storage to reconstruct an application state
- Message compaction to remove old records when using a key-value log
- Horizontal scalability in a cluster configuration
- Replication of data to control fault tolerance
- Retention of high volumes of data for immediate access

1.2. Kafka use cases

Kafka's capabilities make it suitable for:

- Event-driven architectures
- Event sourcing to capture changes to the state of an application as a log of events
- Message brokering
- Website activity tracking
- Operational monitoring through metrics
- Log collection and aggregation
- Commit logs for distributed systems
- Stream processing so that applications can respond to data in real time

1.3. How Strimzi supports Kafka

Strimzi provides container images and Operators for running Kafka on Kubernetes. Strimzi

Operators are fundamental to the running of Strimzi. The Operators provided with Strimzi are purpose-built with specialist operational knowledge to effectively manage Kafka.

Operators simplify the process of:

- Deploying and running Kafka clusters
- Deploying and running Kafka components
- Configuring access to Kafka
- Securing access to Kafka
- Upgrading Kafka
- Managing brokers
- Creating and managing topics
- Creating and managing users

Chapter 2. About Kafka

Apache Kafka is an open-source distributed publish-subscribe messaging system for fault-tolerant real-time data feeds.

Additional resources

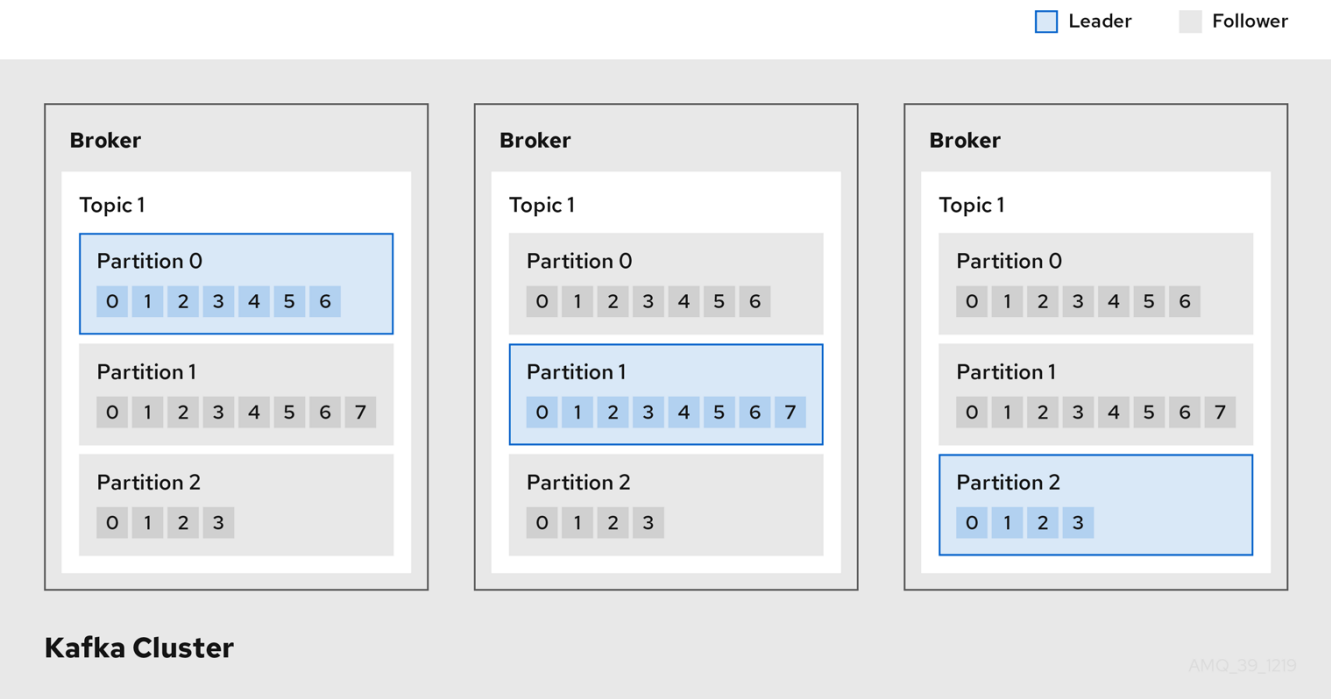
- For more information about Apache Kafka, see the Apache Kafka website.

2.1. Kafka concepts

Knowledge of the key concepts of Kafka is important in understanding how Strimzi works.

A Kafka cluster comprises multiple brokers. Topics are used to receive and store data in a Kafka cluster. Topics are split by partitions, where the data is written. Partitions are replicated across topics for fault tolerance.

Kafka brokers and topics



Broker

A broker, sometimes referred to as a server or node, orchestrates the storage and passing of messages.

Topic

A topic provides a destination for the storage of data. Each topic is split into one or more partitions.

Cluster

A group of broker instances.

Partition

The number of topic partitions is defined by a topic *partition count*.

Partition leader

A partition leader handles all producer requests for a topic.

Partition follower

A partition follower replicates the partition data of a partition leader, optionally handling consumer requests.

Topics use a *replication factor* to configure the number of replicas of each partition within the cluster. A topic comprises at least one partition.

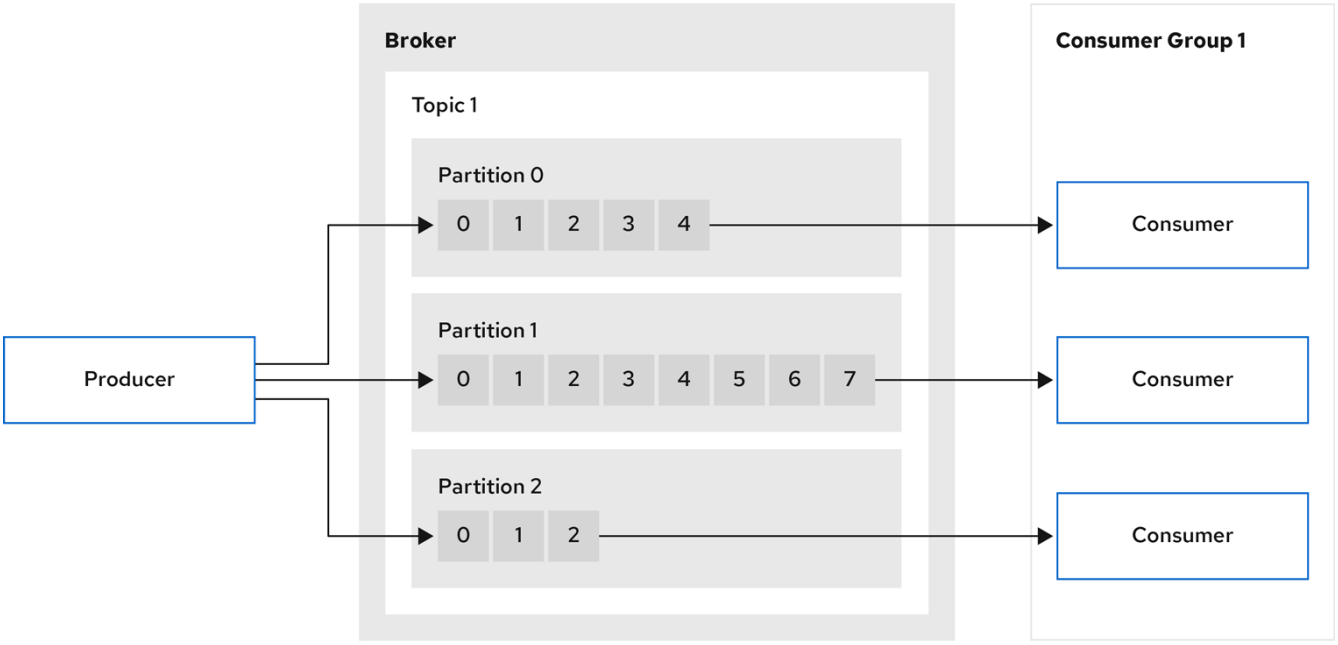
An *in-sync* replica has the same number of messages as the leader. Configuration defines how many replicas must be *in-sync* to be able to produce messages, ensuring that a message is committed only after it has been successfully copied to the replica partition. In this way, if the leader fails the message is not lost.

In the *Kafka brokers and topics* diagram, we can see each numbered partition has a leader and two followers in replicated topics.

2.2. Producers and consumers

Producers and consumers send and receive messages (publish and subscribe) through brokers. Messages comprise an optional *key* and a *value* that contains the message data, plus headers and related metadata. The key is used to identify the subject of the message, or a property of the message. Messages are delivered in batches, and batches and records contain headers and metadata that provide details that are useful for filtering and routing by clients, such as the timestamp and offset position for the record.

Producers and consumers



Producer

A producer sends messages to a broker topic to be written to the end offset of a partition. Messages are written to partitions by a producer on a round robin basis, or to a specific partition based on the message key.

Consumer

A consumer subscribes to a topic and reads messages according to topic, partition and offset.

Consumer group

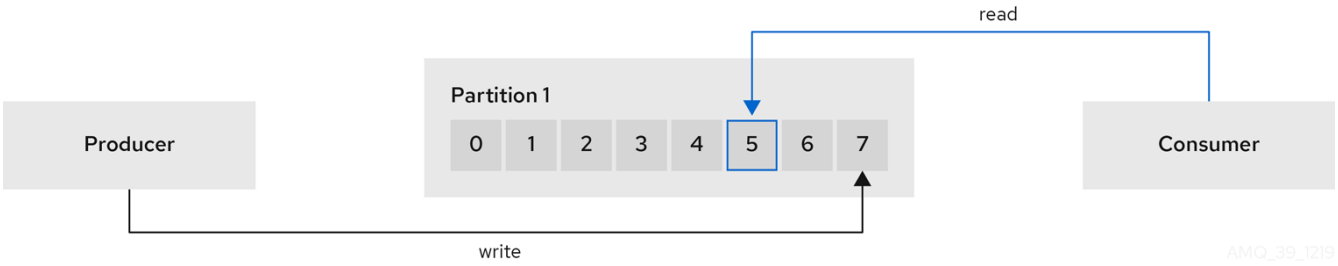
Consumer groups are used to share a typically large data stream generated by multiple producers from a given topic. Consumers are grouped using a `group.id`, allowing messages to be spread across the members. Consumers within a group do not read data from the same partition, but can receive data from one or more partitions.

Offsets

Offsets describe the position of messages within a partition. Each message in a given partition has a unique offset, which helps identify the position of a consumer within the partition to track the number of records that have been consumed.

Committed offsets are written to an offset commit log. A `__consumer_offsets` topic stores information on committed offsets, the position of last and next offset, according to consumer group.

Producing and consuming data


read
Partition 1
0
1
2
3
4
5
6
7
Producer
Consumer
write
AMQ_39_1219

Chapter 3. Strimzi deployment of Kafka

Apache Kafka components are provided for deployment to Kubernetes with the Strimzi distribution. The Kafka components are generally run as clusters for availability.

A typical deployment incorporating Kafka components might include:

- **Kafka** cluster of broker nodes
- **ZooKeeper** cluster of replicated ZooKeeper instances
- **Kafka Connect** cluster for external data connections
- **Kafka MirrorMaker** cluster to mirror the Kafka cluster in a secondary cluster
- **Kafka Exporter** to extract additional Kafka metrics data for monitoring
- **Kafka Bridge** to make HTTP-based requests to the Kafka cluster

Not all of these components are mandatory, though you need Kafka and ZooKeeper as a minimum. Some components can be deployed without Kafka, such as MirrorMaker or Kafka Connect.

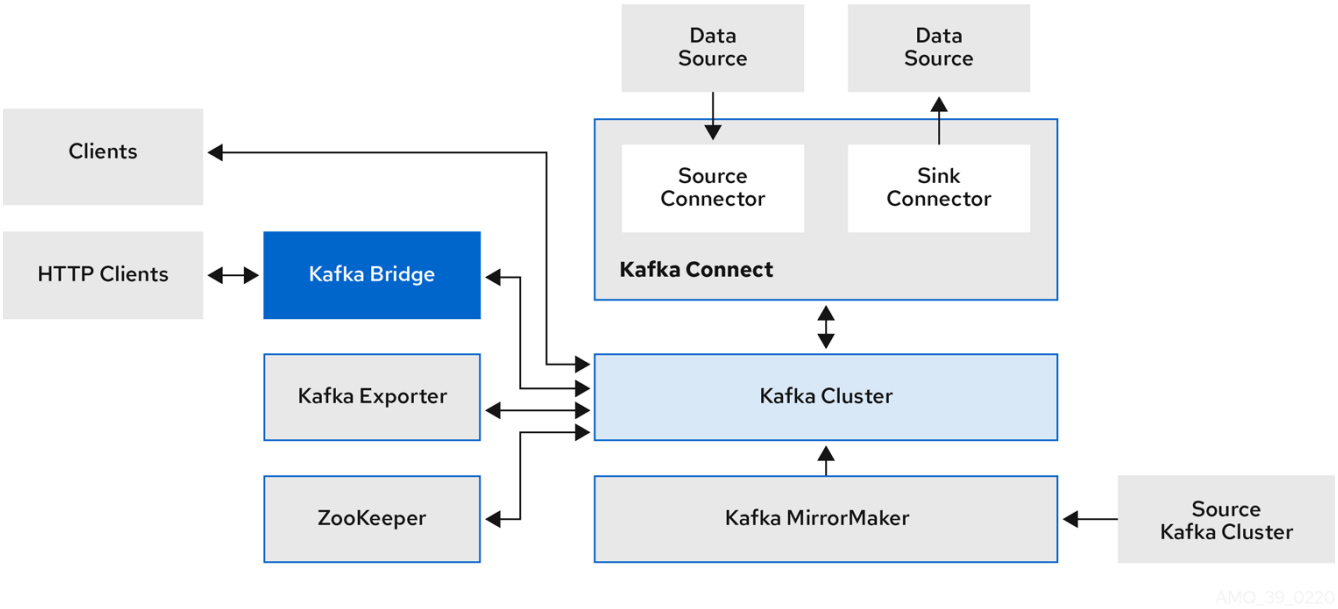
3.1. Kafka component architecture

A cluster of Kafka brokers is main part of the Apache Kafka project responsible for delivering messages.

A broker uses Apache ZooKeeper for storing configuration data and for cluster coordination. Before running Apache Kafka, an Apache ZooKeeper cluster has to be ready.

Each of the other Kafka components interact with the Kafka cluster to perform specific roles.

Kafka component interaction



Apache ZooKeeper

Apache ZooKeeper is a core dependency for Kafka as it provides a cluster coordination service, storing and tracking the status of brokers and consumers. ZooKeeper is also used for leader election of partitions.

Kafka Connect

Kafka Connect is an integration toolkit for streaming data between Kafka brokers and other systems using *Connector* plugins. Kafka Connect provides a framework for integrating Kafka with an external data source or target, such as a database, for import or export of data using connectors. Connectors are plugins that provide the connection configuration needed.

- A *source* connector pushes external data into Kafka.
- A *sink* connector extracts data out of Kafka

  External data is translated and transformed into the appropriate format.

  You can deploy Kafka Connect with `build` configuration that automatically builds a container image with the connector plugins you require for your data connections.

Kafka MirrorMaker

Kafka MirrorMaker replicates data between two Kafka clusters, within or across data centers.

MirrorMaker takes messages from a source Kafka cluster and writes them to a target Kafka cluster.

Kafka Bridge

Kafka Bridge provides an API for integrating HTTP-based clients with a Kafka cluster.

Kafka Exporter

Kafka Exporter extracts data for analysis as Prometheus metrics, primarily data relating to offsets, consumer groups, consumer lag and topics. Consumer lag is the delay between the last message written to a partition and the message currently being picked up from that partition by a consumer

3.2. Kafka Bridge interface

The Kafka Bridge provides a RESTful interface that allows HTTP-based clients to interact with a Kafka cluster. It offers the advantages of a web API connection to Strimzi, without the need for client applications to interpret the Kafka protocol.

The API has two main resources — `consumers` and `topics` — that are exposed and made accessible through endpoints to interact with consumers and producers in your Kafka cluster. The resources relate only to the Kafka Bridge, not the consumers and producers connected directly to Kafka.

3.2.1. HTTP requests

The Kafka Bridge supports HTTP requests to a Kafka cluster, with methods to:

- Send messages to a topic.
- Retrieve messages from topics.
- Retrieve a list of partitions for a topic.
- Create and delete consumers.
- Subscribe consumers to topics, so that they start receiving messages from those topics.
- Retrieve a list of topics that a consumer is subscribed to.
- Unsubscribe consumers from topics.
- Assign partitions to consumers.
- Commit a list of consumer offsets.
- Seek on a partition, so that a consumer starts receiving messages from the first or last offset position, or a given offset position.

The methods provide JSON responses and HTTP response code error handling. Messages can be sent in JSON or binary formats.

Clients can produce and consume messages without the requirement to use the native Kafka protocol.

Additional resources

- To view the API documentation, including example requests and responses, see the Kafka Bridge API reference.

3.2.2. Supported clients for the Kafka Bridge

You can use the Kafka Bridge to integrate both *internal* and *external* HTTP client applications with your Kafka cluster.

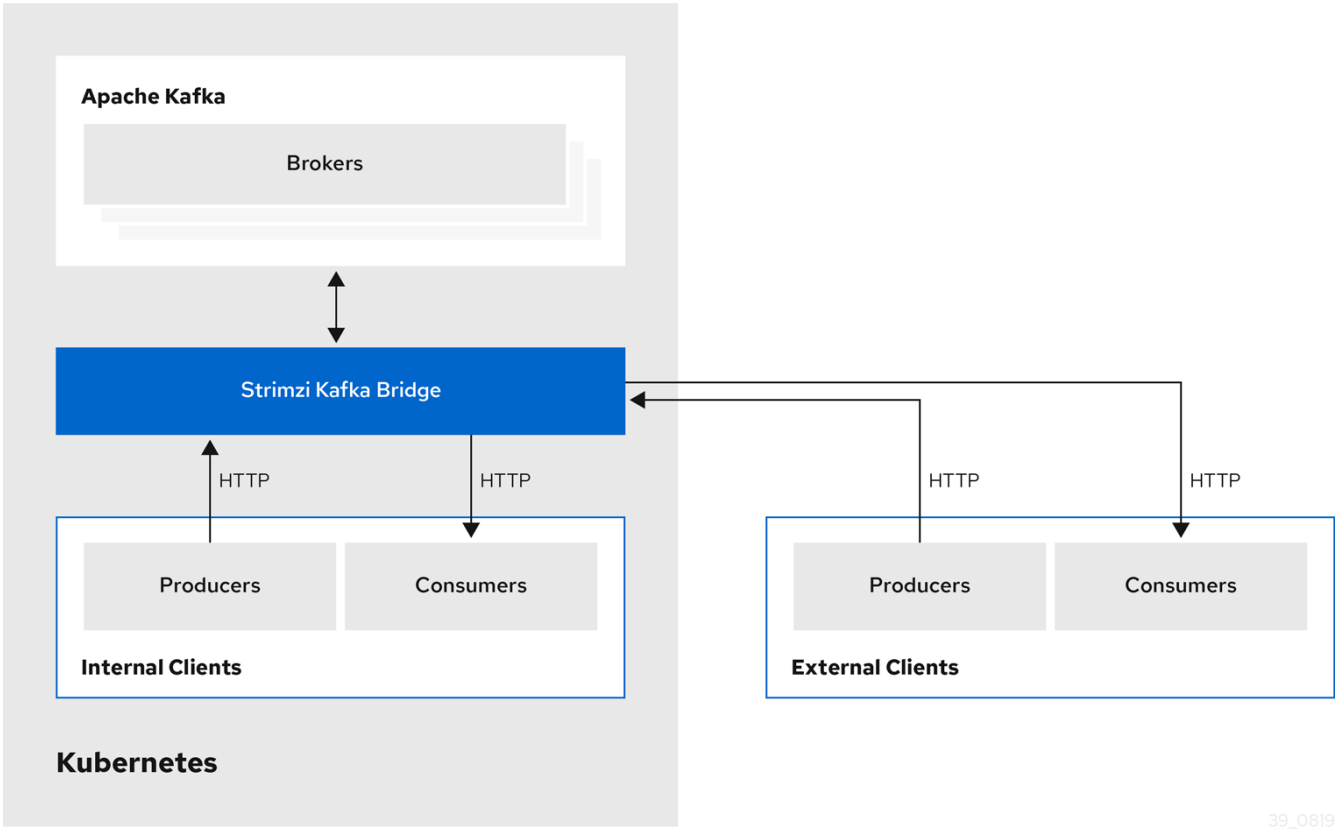
Internal clients

Internal clients are container-based HTTP clients running in *the same* Kubernetes cluster as the Kafka Bridge itself. Internal clients can access the Kafka Bridge on the host and port defined in the `KafkaBridge` custom resource.

External clients

External clients are HTTP clients running *outside* the Kubernetes cluster in which the Kafka Bridge is deployed and running. External clients can access the Kafka Bridge through an OpenShift Route, a loadbalancer service, or using an Ingress.

HTTP internal and external client integration


Apache Kafka
Brokers
Strimzi Kafka Bridge
HTTP
HTTP
HTTP
HTTP
Producers
Consumers
Internal Clients
Producers
Consumers
External Clients
Kubernetes
39_0819

Chapter 4. Strimzi Operators

Strimzi supports Kafka using *Operators* to deploy and manage the components and dependencies of Kafka to Kubernetes.

Operators are a method of packaging, deploying, and managing a Kubernetes application. Strimzi Operators extend Kubernetes functionality, automating common and complex tasks related to a Kafka deployment. By implementing knowledge of Kafka operations in code, Kafka administration tasks are simplified and require less manual intervention.

Operators

Strimzi provides Operators for managing a Kafka cluster running within a Kubernetes cluster.

Cluster Operator

Deploys and manages Apache Kafka clusters, Kafka Connect, Kafka MirrorMaker, Kafka Bridge, Kafka Exporter, and the Entity Operator

Entity Operator

Comprises the Topic Operator and User Operator

Topic Operator

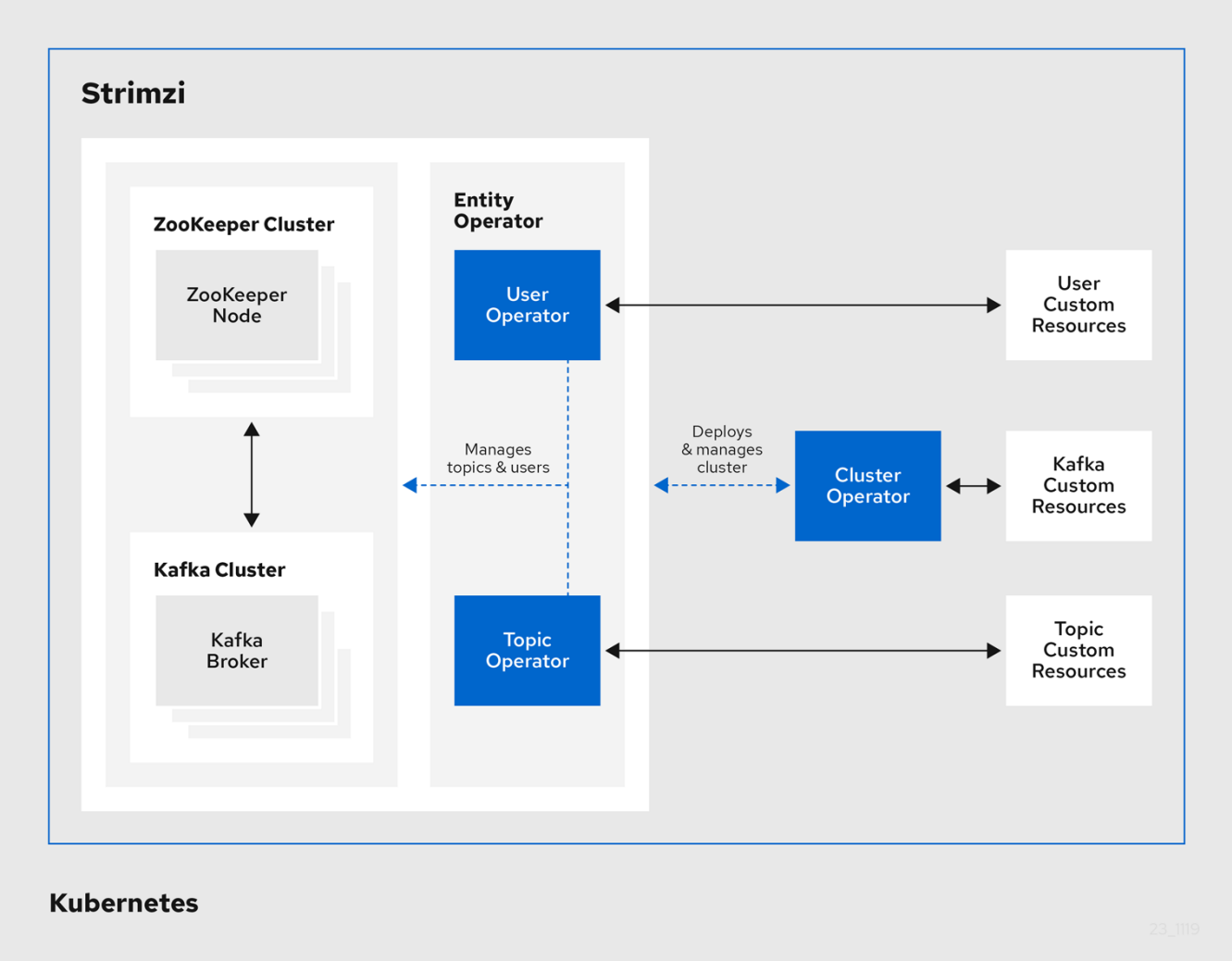
Manages Kafka topics

User Operator

Manages Kafka users

The Cluster Operator can deploy the Topic Operator and User Operator as part of an **Entity Operator** configuration at the same time as a Kafka cluster.

Operators within the Strimzi architecture



4.1. Cluster Operator

Strimzi uses the Cluster Operator to deploy and manage clusters for:

- Kafka (including ZooKeeper, Entity Operator, Kafka Exporter, and Cruise Control)
- Kafka Connect
- Kafka MirrorMaker
- Kafka Bridge

Custom resources are used to deploy the clusters.

For example, to deploy a Kafka cluster:

- A `Kafka` resource with the cluster configuration is created within the Kubernetes cluster.
- The Cluster Operator deploys a corresponding Kafka cluster, based on what is declared in the `Kafka` resource.

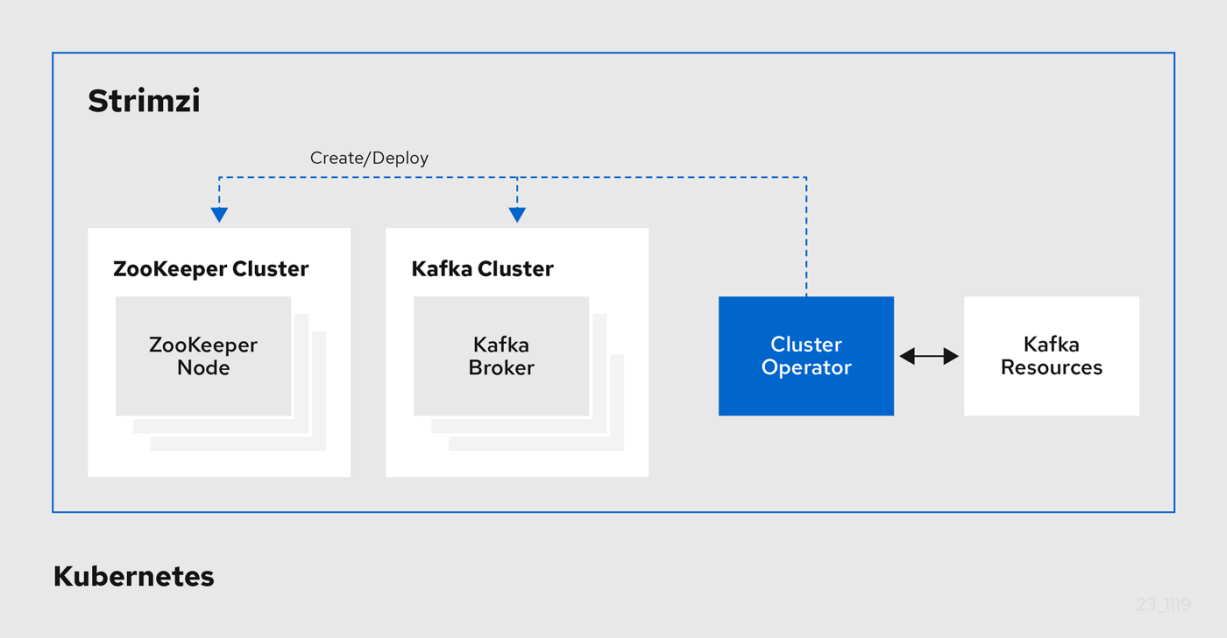
The Cluster Operator can also deploy (through configuration of the `Kafka` resource):

- A Topic Operator to provide operator-style topic management through `KafkaTopic` custom resources

- A User Operator to provide operator-style user management through `KafkaUser` custom resources

The Topic Operator and User Operator function within the Entity Operator on deployment.

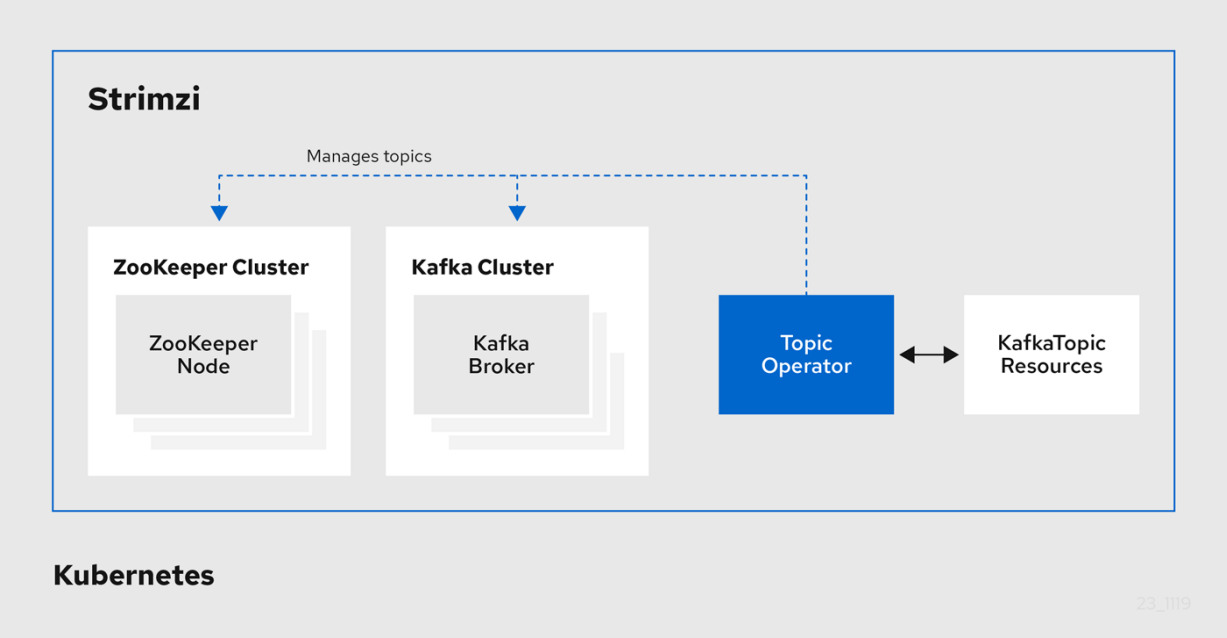
Example architecture for the Cluster Operator



4.2. Topic Operator

The Topic Operator provides a way of managing topics in a Kafka cluster through Kubernetes resources.

Example architecture for the Topic Operator



The role of the Topic Operator is to keep a set of `KafkaTopic` Kubernetes resources describing Kafka topics in-sync with corresponding Kafka topics.

Specifically, if a `KafkaTopic` is:

- Created, the Topic Operator creates the topic
- Deleted, the Topic Operator deletes the topic
- Changed, the Topic Operator updates the topic

Working in the other direction, if a topic is:

- Created within the Kafka cluster, the Operator creates a `KafkaTopic`
- Deleted from the Kafka cluster, the Operator deletes the `KafkaTopic`
- Changed in the Kafka cluster, the Operator updates the `KafkaTopic`

This allows you to declare a `KafkaTopic` as part of your application's deployment and the Topic Operator will take care of creating the topic for you. Your application just needs to deal with producing or consuming from the necessary topics.

The Topic Operator maintains information about each topic in a *topic store*, which is continually synchronized with updates from Kafka topics or Kubernetes `KafkaTopic` custom resources. Updates from operations applied to a local in-memory topic store are persisted to a backup topic store on disk. If a topic is reconfigured or reassigned to other brokers, the `KafkaTopic` will always be up to date.

4.3. User Operator

The User Operator manages Kafka users for a Kafka cluster by watching for `KafkaUser` resources that describe Kafka users, and ensuring that they are configured properly in the Kafka cluster.

For example, if a `KafkaUser` is:

- Created, the User Operator creates the user it describes
- Deleted, the User Operator deletes the user it describes
- Changed, the User Operator updates the user it describes

Unlike the Topic Operator, the User Operator does not sync any changes from the Kafka cluster with the Kubernetes resources. Kafka topics can be created by applications directly in Kafka, but it is not expected that the users will be managed directly in the Kafka cluster in parallel with the User Operator.

The User Operator allows you to declare a `KafkaUser` resource as part of your application's deployment. You can specify the authentication and authorization mechanism for the user. You can also configure *user quotas* that control usage of Kafka resources to ensure, for example, that a user does not monopolize access to a broker.

When the user is created, the user credentials are created in a `Secret`. Your application needs to use the user and its credentials for authentication and to produce or consume messages.

In addition to managing credentials for authentication, the User Operator also manages authorization rules by including a description of the user's access rights in the `KafkaUser`

declaration.

4.4. Feature gates in Strimzi Operators

You can enable and disable some features of operators using *feature gates*.

Feature gates are set in the operator configuration and have three stages of maturity: alpha, beta, or General Availability (GA).

For more information, see Feature gates.

Chapter 5. Kafka configuration

A deployment of Kafka components to a Kubernetes cluster using Strimzi is highly configurable through the application of custom resources. Custom resources are created as instances of APIs added by Custom resource definitions (CRDs) to extend Kubernetes resources.

CRDs act as configuration instructions to describe the custom resources in a Kubernetes cluster, and are provided with Strimzi for each Kafka component used in a deployment, as well as users and topics. CRDs and custom resources are defined as YAML files. Example YAML files are provided with the Strimzi distribution.

CRDs also allow Strimzi resources to benefit from native Kubernetes features like CLI accessibility and configuration validation.

In this chapter we look at how Kafka components are configured through custom resources, starting with common configuration points and then important configuration considerations specific to components.

5.1. Custom resources

After a new custom resource type is added to your cluster by installing a CRD, you can create instances of the resource based on its specification.

The custom resources for Strimzi components have common configuration properties, which are defined under `spec`.

In this fragment from a Kafka topic custom resource, the `apiVersion` and `kind` properties identify the associated CRD. The `spec` property shows configuration that defines the number of partitions and replicas for the topic.

Kafka topic custom resource

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 1
  # ...
```

There are many additional configuration options that can be incorporated into a YAML definition, some common and some specific to a particular component.

5.2. Common configuration

Some of the configuration options common to resources are described here. Security and metrics collection might also be adopted where applicable.

Bootstrap servers

Bootstrap servers are used for host/port connection to a Kafka cluster for:

- Kafka Connect
- Kafka Bridge
- Kafka MirrorMaker producers and consumers

CPU and memory resources

You request CPU and memory resources for components. Limits specify the maximum resources that can be consumed by a given container.

Resource requests and limits for the Topic Operator and User Operator are set in the `Kafka` resource.

Logging

You define the logging level for the component. Logging can be defined directly (inline) or externally using a config map.

Healthchecks

Healthcheck configuration introduces *liveness* and *readiness* probes to know when to restart a container (liveness) and when a container can accept traffic (readiness).

JVM options

JVM options provide maximum and minimum memory allocation to optimize the performance of the component according to the platform it is running on.

Pod scheduling

Pod schedules use *affinity/anti-affinity* rules to determine under what circumstances a pod is scheduled onto a node.

Example YAML showing common configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  bootstrapServers: my-cluster-kafka-bootstrap:9092
  resources:
    requests:
      cpu: 12
      memory: 64Gi
    limits:
      cpu: 12
      memory: 64Gi
  logging:
    type: inline
    loggers:
      connect.root.logger.level: "INFO"
  readinessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  jvmOptions:
    "-Xmx": "2g"
    "-Xms": "2g"
  template:
    pod:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: node-type
                    operator: In
                    values:
                      - fast-network
  # ...
```

5.3. Kafka cluster configuration

A kafka cluster comprises one or more brokers. For producers and consumers to be able to access topics within the brokers, Kafka configuration must define how data is stored in the cluster, and how the data is accessed. You can configure a Kafka cluster to run with multiple broker nodes across *racks*.

Storage

Kafka and ZooKeeper store data on disks.

Strimzi requires block storage provisioned through `StorageClass`. The file system format for storage must be *XFS* or *EXT4*. Three types of data storage are supported:

Ephemeral (Recommended for development only)

Ephemeral storage stores data for the lifetime of an instance. Data is lost when the instance is restarted.

Persistent

Persistent storage relates to long-term data storage independent of the lifecycle of the instance.

JBOD (Just a Bunch of Disks, suitable for Kafka only)

JBOD allows you to use multiple disks to store commit logs in each broker.

The disk capacity used by an existing Kafka cluster can be increased if supported by the infrastructure.

Listeners

Listeners configure how clients connect to a Kafka cluster.

By specifying a unique name and port for each listener within a Kafka cluster, you can configure multiple listeners.

The following types of listener are supported:

- **Internal listeners** for access within Kubernetes
- **External listeners** for access outside of Kubernetes

You can enable TLS encryption for listeners, and configure authentication.

Internal listeners are specified using an `internal` type.

External listeners expose Kafka by specifying an external `type`:

- `route` to use OpenShift routes and the default HAProxy router
- `loadbalancer` to use loadbalancer services
- `nodeport` to use ports on Kubernetes nodes
- `ingress` to use Kubernetes *Ingress* and the NGINX Ingress Controller for Kubernetes

If you are using OAuth 2.0 for token-based authentication, you can configure listeners to use the authorization server.

Rack awareness

Rack awareness is a configuration feature that distributes Kafka broker pods and topic replicas across *racks*, which represent data centers or racks in data centers, or availability zones.

Example YAML showing Kafka configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
      - name: external1
        port: 9094
        type: route
        tls: true
        authentication:
          type: tls
    # ...
    storage:
      type: persistent-claim
      size: 10000Gi
    # ...
    rack:
      topologyKey: topology.kubernetes.io/zone
    # ...
```

5.4. Kafka MirrorMaker configuration

To set up MirrorMaker, a source and target (destination) Kafka cluster must be running.

You can use Strimzi with MirrorMaker 2.0, although the earlier version of MirrorMaker continues to be supported.

MirrorMaker 2.0

MirrorMaker 2.0 is based on the Kafka Connect framework, *connectors* managing the transfer of data between clusters.

MirrorMaker 2.0 uses:

- Source cluster configuration to consume data from the source cluster
- Target cluster configuration to output data to the target cluster

Cluster configuration

You can use MirrorMaker 2.0 in *active/passive* or *active/active* cluster configurations.

- In an *active/active* configuration, both clusters are active and provide the same data simultaneously, which is useful if you want to make the same data available locally in different geographical locations.
- In an *active/passive* configuration, the data from an active cluster is replicated in a passive cluster, which remains on standby, for example, for data recovery in the event of system failure.

You configure a `KafkaMirrorMaker2` custom resource to define the Kafka Connect deployment, including the connection details of the source and target clusters, and then run a set of MirrorMaker 2.0 connectors to make the connection.

Topic configuration is automatically synchronized between the source and target clusters according to the topics defined in the `KafkaMirrorMaker2` custom resource. Configuration changes are propagated to remote topics so that new topics and partitions are detected and created. Topic replication is defined using regular expression patterns to include or exclude topics.

The following MirrorMaker 2.0 connectors and related internal topics help manage the transfer and synchronization of data between the clusters.

MirrorSourceConnector

A *MirrorSourceConnector* creates remote topics from the source cluster.

MirrorCheckpointConnector

A *MirrorCheckpointConnector* tracks and maps offsets for specified consumer groups using an *offset sync* topic and *checkpoint* topic. The offset sync topic maps the source and target offsets for replicated topic partitions from record metadata. A checkpoint is emitted from each source cluster and replicated in the target cluster through the checkpoint topic. The checkpoint topic maps the last committed offset in the source and target cluster for replicated topic partitions in each consumer group.

MirrorHeartbeatConnector

A *MirrorHeartbeatConnector* periodically checks connectivity between clusters. A heartbeat is produced every second by the MirrorHeartbeatConnector into a *heartbeat* topic that is created on the local cluster. If you have MirrorMaker 2.0 at both the remote and local locations, the heartbeat emitted at the remote location by the MirrorHeartbeatConnector is treated like any remote topic and mirrored by the MirrorSourceConnector at the local cluster. The heartbeat topic makes it easy to check that the remote cluster is available and the clusters are connected. If things go wrong, the heartbeat topic offset positions and time stamps can help with recovery and diagnosis.

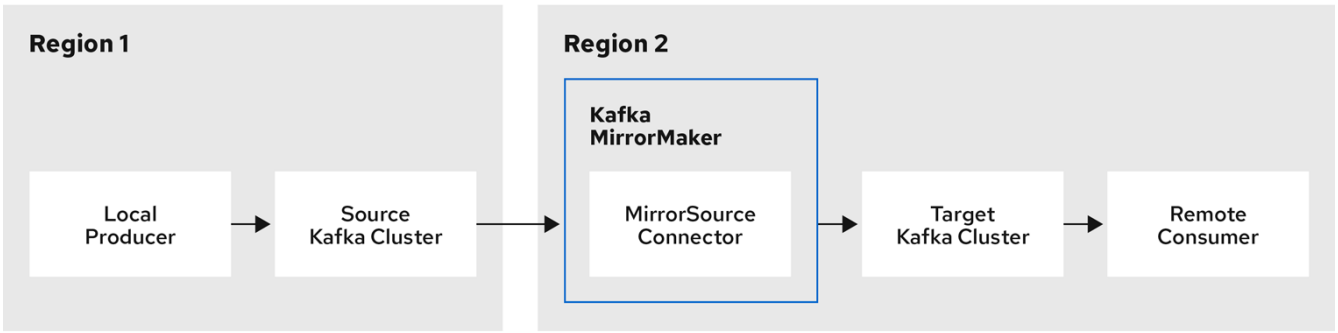


Figure 1. Replication across two clusters

Bidirectional replication across two clusters

The MirrorMaker 2.0 architecture supports bidirectional replication in an *active/active* cluster configuration, so both clusters are active and provide the same data simultaneously. A MirrorMaker 2.0 cluster is required at each target destination.

Remote topics are distinguished by automatic renaming that prepends the name of cluster to the name of the topic. This is useful if you want to make the same data available locally in different geographical locations.

However, if you want to backup or migrate data in an active/passive cluster configuration, you might want to keep the original names of the topics. If so, you can configure MirrorMaker 2.0 to turn off automatic renaming.

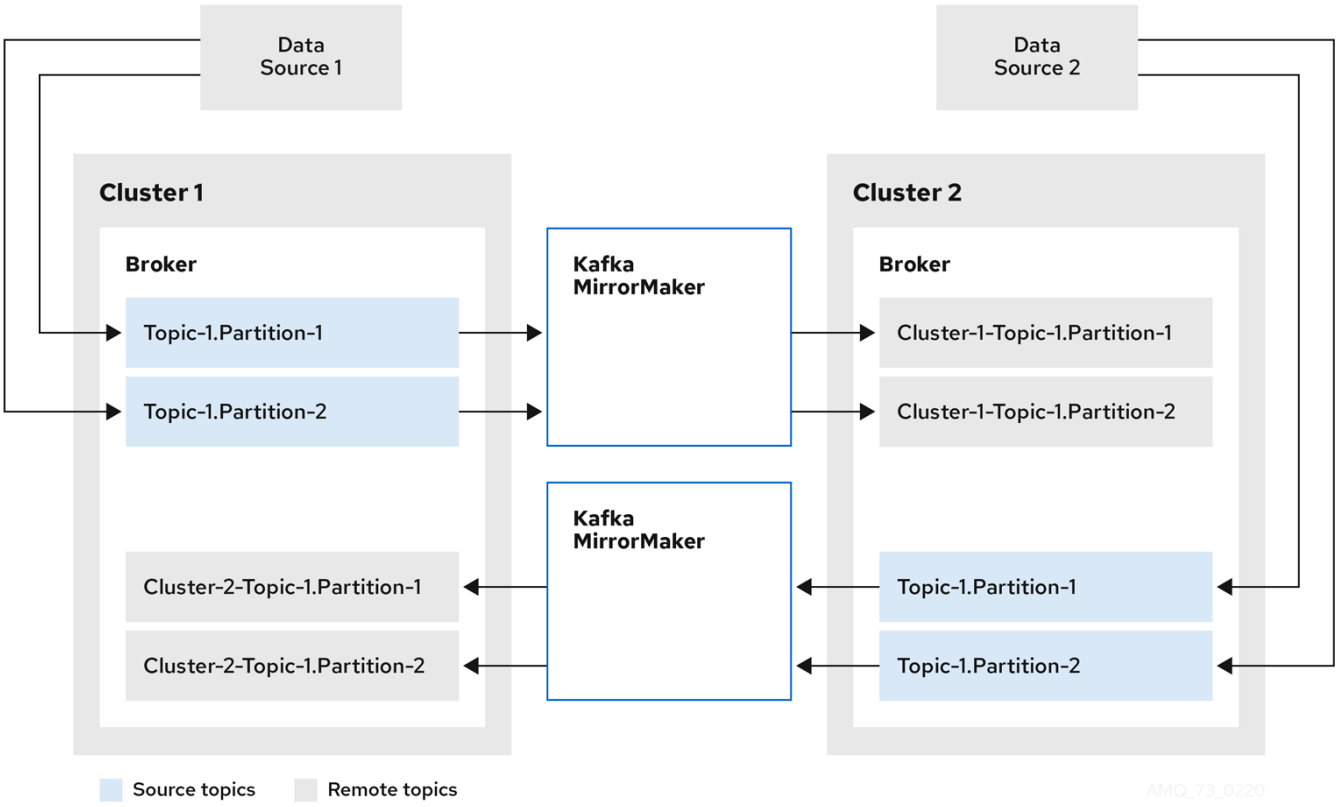


Figure 2. Bidirectional replication

Example YAML showing MirrorMaker 2.0 configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
  spec:
    version: 2.8.0
    connectCluster: "my-cluster-target"
    clusters:
    - alias: "my-cluster-source"
      bootstrapServers: my-cluster-source-kafka-bootstrap:9092
    - alias: "my-cluster-target"
      bootstrapServers: my-cluster-target-kafka-bootstrap:9092
    mirrors:
    - sourceCluster: "my-cluster-source"
      targetCluster: "my-cluster-target"
      sourceConnector: {}
    topicsPattern: ".*"
    groupsPattern: "group1|group2|group3"
```

MirrorMaker

The earlier version of MirrorMaker uses producers and consumers to replicate data across clusters.

MirrorMaker uses:

- Consumer configuration to consume data from the source cluster
- Producer configuration to output data to the target cluster

Consumer and producer configuration includes any authentication and encryption settings.

The `include` field defines the topics to mirror from a source to a target cluster.

Key Consumer configuration

Consumer group identifier

The consumer group ID for a MirrorMaker consumer so that messages consumed are assigned to a consumer group.

Number of consumer streams

A value to determine the number of consumers in a consumer group that consume a message in parallel.

Offset commit interval

An offset commit interval to set the time between consuming and committing a message.

Key Producer configuration

Cancel option for send failure

You can define whether a message send failure is ignored or MirrorMaker is terminated and recreated.

Example YAML showing MirrorMaker configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    bootstrapServers: my-source-cluster-kafka-bootstrap:9092
    groupId: "my-group"
    numStreams: 2
    offsetCommitInterval: 120000
    # ...
  producer:
    # ...
    abortOnSendFailure: false
    # ...
  include: "my-topic|other-topic"
  # ...
```

5.5. Kafka Connect configuration

A basic Kafka Connect configuration requires a bootstrap address to connect to a Kafka cluster, and encryption and authentication details.

Kafka Connect instances are configured by default with the same:

- Group ID for the Kafka Connect cluster
- Kafka topic to store the connector offsets
- Kafka topic to store connector and task status configurations
- Kafka topic to store connector and task status updates

If multiple different Kafka Connect instances are used, these settings must reflect each instance.

Example YAML showing Kafka Connect configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: my-connect-cluster
    offset.storage.topic: my-connect-cluster-offsets
    config.storage.topic: my-connect-cluster-configs
    status.storage.topic: my-connect-cluster-status
  # ...
```

Connectors

Connectors are configured separately from Kafka Connect. The configuration describes the source input data and target output data to feed into and out of Kafka Connect. The external source data must reference specific topics that will store the messages.

Kafka provides two built-in connectors:

- `FileStreamSourceConnector` streams data from an external system to Kafka, reading lines from an input source and sending each line to a Kafka topic.
- `FileStreamSinkConnector` streams data from Kafka to an external system, reading messages from a Kafka topic and creating a line for each in an output file.

You can add other connectors using connector plugins, which are a set of JAR files or TGZ archives that define the implementation required to connect to certain types of external system.

You create a custom Kafka Connect image that uses new connector plugins.

To create the image, you can use:

- Kafka Connect configuration so that Strimzi creates the new image automatically.
- A Kafka container image on Container Registry as a base image.
- OpenShift builds and the Source-to-Image (S2I) framework to create new container images.

For Strimzi to create the new image automatically, a `build` configuration requires `output` properties to reference a container registry that stores the container image, and `plugins` properties to list the connector plugins and their artifacts to add to the image.

The `output` properties describe the type and name of the image, and optionally the name of the Secret containing the credentials needed to access the container registry. The `plugins` properties describe the type of artifact and the URL from which the artifact is downloaded. Additionally, you can specify a SHA-512 checksum to verify the artifact before unpacking it.

Example Kafka Connect configuration to create a new image automatically

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  # ...
  build:
    output:
      type: docker
      image: my-registry.io/my-org/my-connect-cluster:latest
      pushSecret: my-registry-credentials
    plugins:
      - name: debezium-postgres-connector
        artifacts:
          - type: tgz
            url: https://ARTIFACT-ADDRESS.tgz
            sha512sum: HASH-NUMBER-TO-VERIFY-ARTIFACT
      # ...
  #...
```

Managing connectors

You can use the KafkaConnector resource or the Kafka Connect REST API to create and manage connector instances in a Kafka Connect cluster. The KafkaConnector resource offers a Kubernetes-native approach, and is managed by the Cluster Operator.

The `spec` for the KafkaConnector resource specifies the connector class and configuration settings, as well as the maximum number of connector *tasks* to handle the data.

Example YAML showing KafkaConnector configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector
  tasksMax: 2
  config:
    file: "/opt/kafka/LICENSE"
    topic: my-topic
    # ...
```

You enable KafkaConnectors by adding an annotation to the `KafkaConnect` resource. KafkaConnector

resources must be deployed to the same namespace as the Kafka Connect cluster they link to.

Example YAML showing annotation to enable KafkaConnector

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
  annotations:
    strimzi.io/use-connector-resources: "true"
  # ...
```

5.6. Kafka Bridge configuration

A Kafka Bridge configuration requires a bootstrap server specification for the Kafka cluster it connects to, as well as any encryption and authentication options required.

Kafka Bridge consumer and producer configuration is standard, as described in the Apache Kafka configuration documentation for consumers and Apache Kafka configuration documentation for producers.

HTTP-related configuration options set the port connection which the server listens on.

CORS

The Kafka Bridge supports the use of Cross-Origin Resource Sharing (CORS). CORS is a HTTP mechanism that allows browser access to selected resources from more than one origin, for example, resources on different domains. If you choose to use CORS, you can define a list of allowed resource origins and HTTP methods for interaction with the Kafka cluster through the Kafka Bridge. The lists are defined in the `http` specification of the Kafka Bridge configuration.

CORS allows for *simple* and *preflighted* requests between origin sources on different domains.

- A simple request is a HTTP request that must have an allowed origin defined in its header.
- A preflighted request sends an initial OPTIONS HTTP request before the actual request to check that the origin and the method are allowed.

Example YAML showing Kafka Bridge configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  bootstrapServers: my-cluster-kafka:9092
  http:
    port: 8080
    cors:
      allowedOrigins: "https://strimzi.io"
      allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
  consumer:
    config:
      auto.offset.reset: earliest
  producer:
    config:
      delivery.timeout.ms: 300000
  # ...
```

Additional resources

- Fetch CORS specification

Chapter 6. Securing Kafka

A secure deployment of Strimzi can encompass:

- Encryption for data exchange
- Authentication to prove identity
- Authorization to allow or decline actions executed by users

6.1. Encryption

Strimzi supports Transport Layer Security (TLS), a protocol for encrypted communication.

Communication is always encrypted for communication between:

- Kafka brokers
- ZooKeeper nodes
- Operators and Kafka brokers
- Operators and ZooKeeper nodes
- Kafka Exporter

You can also configure TLS between Kafka brokers and clients by applying TLS encryption to the listeners of the Kafka broker. TLS is specified for external clients when configuring an external listener.

Strimzi components and Kafka clients use digital certificates for encryption. The Cluster Operator sets up certificates to enable encryption within the Kafka cluster. You can provide your own server certificates, referred to as *Kafka listener certificates*, for communication between Kafka clients and Kafka brokers, and inter-cluster communication.

Strimzi uses *Secrets* to store the certificates and private keys required for TLS in PEM and PKCS #12 format.

A TLS Certificate Authority (CA) issues certificates to authenticate the identity of a component. Strimzi verifies the certificates for the components against the CA certificate.

- Strimzi components are verified against the *cluster CA* Certificate Authority (CA)
- Kafka clients are verified against the *clients CA* Certificate Authority (CA)

6.2. Authentication

Kafka listeners use authentication to ensure a secure client connection to the Kafka cluster.

Supported authentication mechanisms:

- Mutual TLS client authentication (on listeners with TLS enabled encryption)
- SASL SCRAM-SHA-512

- OAuth 2.0 token based authentication

The User Operator manages user credentials for TLS and SCRAM authentication, but not OAuth 2.0. For example, through the User Operator you can create a user representing a client that requires access to the Kafka cluster, and specify TLS as the authentication type.

Using OAuth 2.0 token-based authentication, application clients can access Kafka brokers without exposing account credentials. An authorization server handles the granting of access and inquiries about access.

6.3. Authorization

Kafka clusters use authorization to control the operations that are permitted on Kafka brokers by specific clients or users. If applied to a Kafka cluster, authorization is enabled for all listeners used for client connection.

If a user is added to a list of *super users* in a Kafka broker configuration, the user is allowed unlimited access to the cluster regardless of any authorization constraints implemented through authorization mechanisms.

Supported authorization mechanisms:

- Simple authorization
- OAuth 2.0 authorization (if you are using OAuth 2.0 token-based authentication)
- Open Policy Agent (OPA) authorization
- Custom authorization

Simple authorization uses `AclAuthorizer`, the default Kafka authorization plugin. `AclAuthorizer` uses Access Control Lists (ACLs) to define which users have access to which resources. For custom authorization, you configure your own `Authorizer` plugin to enforce ACL rules.

OAuth 2.0 and OPA provide policy-based control from an authorization server. Security policies and permissions used to grant access to resources on Kafka brokers are defined in the authorization server.

URLs are used to connect to the authorization server and verify that an operation requested by a client or user is allowed or denied. Users and clients are matched against the policies created in the authorization server that permit access to perform specific actions on Kafka brokers.

Chapter 7. Monitoring

Monitoring data allows you to monitor the performance and health of Strimzi. You can configure your deployment to capture metrics data for analysis and notifications.

Metrics data is useful when investigating issues with connectivity and data delivery. For example, metrics data can identify under-replicated partitions or the rate at which messages are consumed. Alerting rules can provide time-critical notifications on such metrics through a specified communications channel. Monitoring visualizations present real-time metrics data to help determine when and how to update the configuration of your deployment. Example metrics configuration files are provided with Strimzi.

Distributed tracing complements the gathering of metrics data by providing a facility for end-to-end tracking of messages through Strimzi.

Cruise Control provides support for rebalancing of Kafka clusters, based on workload data.

Metrics and monitoring tools

Strimzi can employ the following tools for metrics and monitoring:

- **Prometheus** pulls metrics from Kafka, ZooKeeper and Kafka Connect clusters. The Prometheus **Alertmanager** plugin handles alerts and routes them to a notification service.
- **Kafka Exporter** adds additional Prometheus metrics
- **Grafana** provides dashboard visualizations of Prometheus metrics
- **Jaeger** provides distributed tracing support to track transactions between applications
- **Cruise Control** balances data across a Kafka cluster

Additional resources

- Prometheus
- Kafka Exporter
- Grafana Labs
- Jaeger
- Cruise Control Wiki

7.1. Prometheus

Prometheus can extract metrics data from Kafka components and the Strimzi Operators.

To use Prometheus to obtain metrics data and provide alerts, Prometheus and the Prometheus Alertmanager plugin must be deployed. Kafka resources must also be deployed or redeployed with metrics configuration to expose the metrics data.

Prometheus scrapes the exposed metrics data for monitoring. Alertmanager issues alerts when conditions indicate potential problems, based on pre-defined alerting rules.

Sample metrics and alerting rules configuration files are provided with Strimzi. The sample alerting mechanism provided with Strimzi is configured to send notifications to a Slack channel.

7.2. Grafana

Grafana uses the metrics data exposed by Prometheus to present dashboard visualizations for monitoring.

A deployment of Grafana is required, with Prometheus added as a data source. Example dashboards, supplied with Strimzi as JSON files, are imported through the Grafana interface to present monitoring data.

7.3. Kafka Exporter

Kafka Exporter is an open source project to enhance monitoring of Apache Kafka brokers and clients. Kafka Exporter is deployed with a Kafka cluster to extract additional Prometheus metrics data from Kafka brokers related to offsets, consumer groups, consumer lag, and topics. You can use the Grafana dashboard provided to visualize the data collected by Prometheus from Kafka Exporter.

A sample configuration file, alerting rules and Grafana dashboard for Kafka Exporter are provided with Strimzi.

7.4. Distributed tracing

Within a Kafka deployment, distributed tracing using Jaeger is supported for:

- MirrorMaker to trace messages from a source cluster to a target cluster
- Kafka Connect to trace messages consumed and produced by Kafka Connect
- Kafka Bridge to trace messages consumed and produced by Kafka Bridge, and HTTP requests from client applications

Template configuration properties are set for the Kafka resources, which describe tracing environment variables.

Tracing for Kafka clients

Client applications, such as Kafka producers and consumers, can also be set up so that transactions are monitored. Clients are configured with a tracing profile, and a tracer is initialized for the client application to use.

7.5. Cruise Control

Cruise Control is an open source project for simplifying the monitoring and balancing of data across a Kafka cluster. Cruise Control is deployed alongside a Kafka cluster to monitor its traffic, propose more balanced partition assignments, and trigger partition reassignments based on those proposals.

Cruise Control collects resource utilization information to model and analyze the workload of the Kafka cluster. Based on *optimization goals* that have been defined, Cruise Control generates *optimization proposals* outlining how the cluster can be effectively rebalanced. When an *optimization proposal* is approved, Cruise Control applies the rebalancing outlined in the proposal.

Prometheus can extract Cruise Control metrics data, including data related to optimization proposals and rebalancing operations. A sample configuration file and Grafana dashboard for Cruise Control are provided with Strimzi.