Distributed Message Service for Kafka

Best Practices

Issue 01

Date 2023-09-15

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

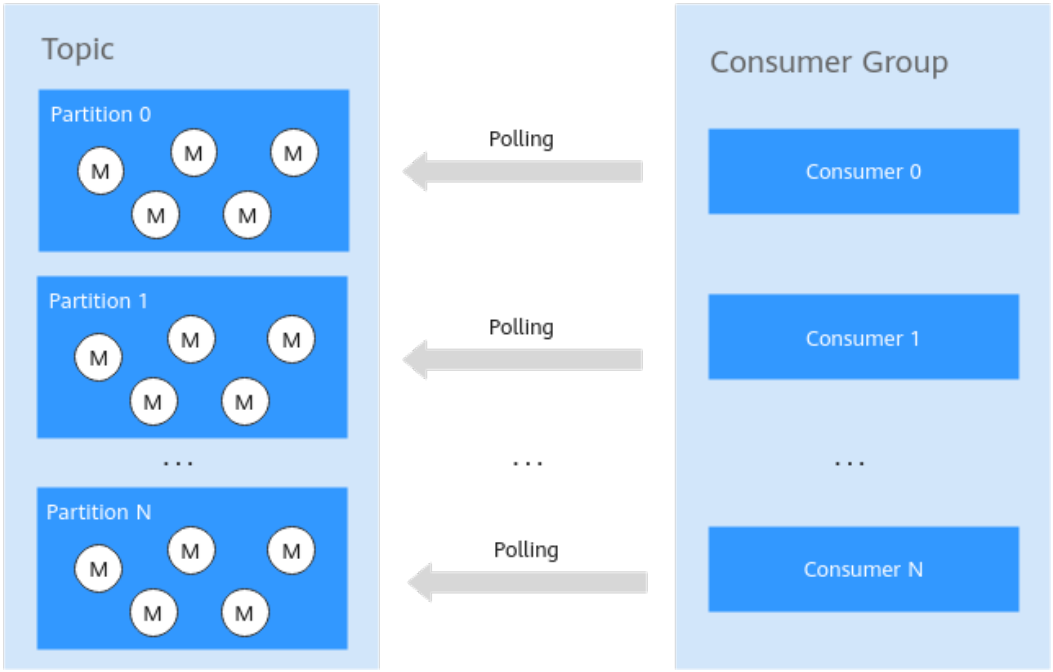
1 Optimizing Message Polling of DMS for Kafka Consumers

Overview

In the native Kafka SDK provided by DMS for Kafka, consumers can customize the duration for pulling messages. To pull messages for a long time, consumers only need to set the parameter of the poll (long) method to a proper value. However, such long connections may cause pressure on the client and the server, especially when the number of partitions is large and multiple threads are enabled for each consumer.

As shown in **Figure 1-1**, the topic contains multiple partitions, and multiple consumers in the consumer group consume the resources at the same time. Each thread is in a persistent connection. When there are few or no messages in the topic, the connection persists, and all consumers pull messages continuously, which causes a waste of resources.

Figure 1-1 Multi-thread consumption of Kafka consumers

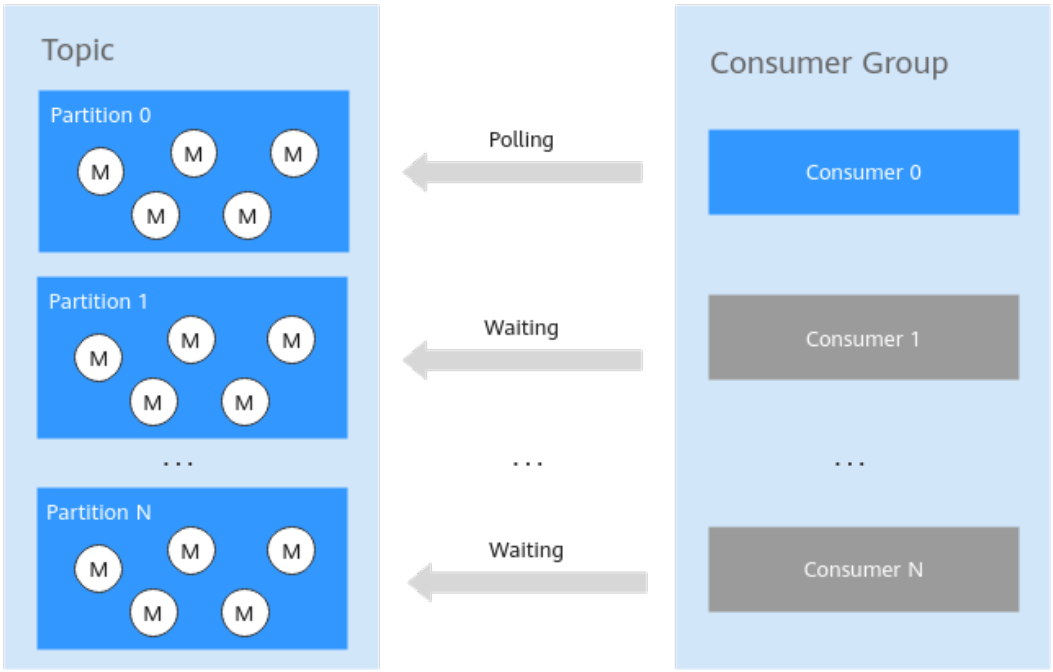


Optimization Solution

When multiple threads are accessed at the same time, if there is no message in the topic, only one thread is required to poll messages in each partition. When a message is found in the polling thread, other threads can be woken up to consume the messages. In this way, the message can be quickly responded, as shown in **Figure 1-2**.

This solution is applicable to scenarios with low requirements on real-time message consumption. If real-time message consumption is required, it is recommended that all consumers be in the active state.

Figure 1-2 Optimized multi-thread consumption solution



NOTE

The number of consumers and the number of partitions are not necessarily the same. The poll (long) method of Kafka helps implement the functions such as message acquisition, partition balancing, and heartbeat detection between consumers and Kafka brokers.

Therefore, in scenarios with low requirements on real-time message consumption and there is a small number of messages, some consumers can be in the wait state.

Sample Code

NOTICE

The following describes only the code related to wake-up and sleep of the consumer thread. To run the entire demo, download the complete **sample code package** and refer to the **Developer Guide** for deploying and running the code.

Sample code for consuming messages:

```
package com.huawei.dms.kafka;

import java.io.IOException;
import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerRebalanceListener;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;
import org.apache.log4j.Logger;
```



```
public class DmsKafkaConsumeDemo
{
    private static Logger logger = Logger.getLogger(DmsKafkaProduceDemo.class);

    public static void WorkerFunc(int workerId, KafkaConsumer<String, String> kafkaConsumer) throws
IOException
    {
        Properties consumerConfig = Config.getConsumerConfig();
        RecordReceiver receiver = new RecordReceiver(workerId, kafkaConsumer,
consumerConfig.getProperty("topic"));
        while (true)
        {
            ConsumerRecords<String, String> records = receiver.receiveMessage();
            Iterator<ConsumerRecord<String, String>> iter = records.iterator();
            while (iter.hasNext())
            {
                ConsumerRecord<String, String> cr = iter.next();
                System.out.println("Thread" + workerId + " recievedrecords" + cr.value());
                logger.info("Thread" + workerId + " recievedrecords" + cr.value());

            }

        }
    }

    public static KafkaConsumer<String, String> getConsumer() throws IOException
    {
        Properties consumerConfig = Config.getConsumerConfig();

        consumerConfig.put("ssl.truststore.location", Config.getTrustStorePath());
        System.setProperty("java.security.auth.login.config", Config.getSaslConfig());

        KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer<>(consumerConfig);
        kafkaConsumer.subscribe(Arrays.asList(consumerConfig.getProperty("topic")),
                new ConsumerRebalanceListener()
                {
                    @Override
                    public void onPartitionsRevoked(Collection<TopicPartition> arg0)
                    {

                    }

                    @Override
                    public void onPartitionsAssigned(Collection<TopicPartition> tps)
                    {

                    }
                });
        return kafkaConsumer;
    }

    public static void main(String[] args) throws IOException
    {

        //Create a consumer for the current consumer group.
        final KafkaConsumer<String, String> consumer1 = getConsumer();
        Thread thread1 = new Thread(new Runnable()
        {
            public void run()
            {
                try
                {
                    WorkerFunc(1, consumer1);
                }
                catch (IOException e)
                {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
```



```
            }
        });
        final KafkaConsumer<String, String> consumer2 = getConsumer();

        Thread thread2 = new Thread(new Runnable()
        {
            public void run()
            {
                try
                {
                    WorkerFunc(2, consumer2);
                }
                catch (IOException e)
                {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        });
        final KafkaConsumer<String, String> consumer3 = getConsumer();

        Thread thread3 = new Thread(new Runnable()
        {
            public void run()
            {
                try
                {
                    WorkerFunc(3, consumer3);
                }
                catch (IOException e)
                {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        });

        //Start threads.
        thread1.start();
        thread2.start();
        thread3.start();

        try
        {
            Thread.sleep(5000);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
        //Add threads.
        try
        {
            thread1.join();
            thread2.join();
            thread3.join();
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

Sample code for consumer thread management:

The sample code provides only simple design ideas. Developers can optimize the thread wake-up and sleep mechanisms based on actual scenarios.


```
package com.huawei.dms.kafka;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import org.apache.log4j.Logger;

public class RecordReceiver
{
    private static Logger logger = Logger.getLogger(DmsKafkaProduceDemo.class);

        //Interval time of polling
    public static final int WAIT_SECONDS = 10 * 1000;

    protected static final Map<String, Object> sLockObjMap = new HashMap<String, Object>();

    protected static Map<String, Boolean> sPollingMap = new ConcurrentHashMap<String, Boolean>();

    protected Object lockObj;

    protected String topicName;

    protected KafkaConsumer<String, String> kafkaConsumer;

    protected int workerId;

    public RecordReceiver(int id, KafkaConsumer<String, String> kafkaConsumer, String queue)
    {
        this.kafkaConsumer = kafkaConsumer;
        this.topicName = queue;
        this.workerId = id;

        synchronized (sLockObjMap)
        {
            lockObj = sLockObjMap.get(topicName);
            if (lockObj == null)
            {
                lockObj = new Object();
                sLockObjMap.put(topicName, lockObj);
            }
        }
    }

    public boolean setPolling()
    {
        synchronized (lockObj)
        {
            Boolean ret = sPollingMap.get(topicName);
            if (ret == null || !ret)
            {
                sPollingMap.put(topicName, true);
                return true;
            }
            return false;
        }
    }

    //Wake up all threads.
    public void clearPolling()
    {
        synchronized (lockObj)
        {
            sPollingMap.put(topicName, false);
            lockObj.notifyAll();
            System.out.println("Everyone WakeUp and Work!");
```

```
            logger.info("Everyone WakeUp and Work!");
        }
    }

    public ConsumerRecords<String, String> receiveMessage()
    {
        boolean polling = false;
        while (true)
        {
            //Check the poll status of threads and hibernate the threads when necessary.
            synchronized (lockObj)
            {
                Boolean p = sPollingMap.get(topicName);
                if (p != null && p)
                {
                    try
                    {
                        System.out.println("Thread" + workerId + " Have a nice sleep!");
                        logger.info("Thread" + workerId +" Have a nice sleep!");
                        polling = false;
                        lockObj.wait();
                    }
                    catch (InterruptedException e)
                    {
                        System.out.println("MessageReceiver Interrupted! topicName is " + topicName);
                        logger.error("MessageReceiver Interrupted! topicName is "+topicName);

                        return null;
                    }
                }
            }

            //Start to consume and wake up other threads when necessary.
            try
            {
                ConsumerRecords<String, String> Records = null;
                if (!polling)
                {
                    Records = kafkaConsumer.poll(100);
                    if (Records.count() == 0)
                    {
                        polling = true;
                        continue;
                    }
                }
                else
                {
                    if (setPolling())
                    {
                        System.out.println("Thread" + workerId + " Polling!");
                        logger.info("Thread " + workerId + " Polling!");
                    }
                    else
                    {
                        continue;
                    }
                    do
                    {
                        System.out.println("Thread" + workerId + " KEEP Poll records!");
                        logger.info("Thread" + workerId + " KEEP Poll records!");
                        try
                        {
                            Records = kafkaConsumer.poll(WAIT_SECONDS);
                        }
                        catch (Exception e)
                        {
                            System.out.println("Exception Happened when polling records: " + e);
                            logger.error("Exception Happened when polling records: " + e);
```



```
                }
            } while (Records.count()==0);
            clearPolling();
        }
        //Acknowledge message consumption.
        kafkaConsumer.commitSync();
        return Records;
    }
    catch (Exception e)
    {
        System.out.println("Exception Happened when poll records: " + e);
        logger.error("Exception Happened when poll records: " + e);
    }
  }
 }
}
```

NOTE

topicName is the name of the topic.

Running Results of Sample Code

```
[2018-01-25 22:40:51,841] INFO Thread 2 Polling! (com.huawei.dms.kafka.DmsKafkaProduceDemo:119)
[2018-01-25 22:40:51,841] INFO Thread2 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:40:52,122] INFO Everyone WakeUp and Work!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:69)
[2018-01-25 22:40:52,169] INFO Thread2 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:40:52,169] INFO Thread2 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:40:52,216] INFO Thread2 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:40:52,325] INFO Thread 2 Polling! (com.huawei.dms.kafka.DmsKafkaProduceDemo:119)
[2018-01-25 22:40:52,325] INFO Thread2 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:40:54,947] INFO Thread1 Have a nice sleep!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:87)
[2018-01-25 22:40:54,979] INFO Thread3 Have a nice sleep!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:87)
[2018-01-25 22:41:32,347] INFO Thread2 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:41:42,353] INFO Thread2 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:41:47,816] INFO Everyone WakeUp and Work!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:69)
[2018-01-25 22:41:47,847] INFO Thread2 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:47,925] INFO Thread 3 Polling! (com.huawei.dms.kafka.DmsKafkaProduceDemo:119)
[2018-01-25 22:41:47,925] INFO Thread1 Have a nice sleep!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:87)
[2018-01-25 22:41:47,925] INFO Thread3 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:41:47,957] INFO Thread2 Have a nice sleep!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:87)
[2018-01-25 22:41:48,472] INFO Everyone WakeUp and Work!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:69)
[2018-01-25 22:41:48,503] INFO Thread3 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:48,518] INFO Thread1 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:48,550] INFO Thread2 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:48,597] INFO Thread1 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:48,659] INFO Thread 2 Polling! (com.huawei.dms.kafka.DmsKafkaProduceDemo:119)
[2018-01-25 22:41:48,659] INFO Thread2 KEEP Poll records!
```



```
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
[2018-01-25 22:41:48,675] INFO Thread3 recievedrecordshello, dms kafka.
(com.huawei.dms.kafka.DmsKafkaProduceDemo:32)
[2018-01-25 22:41:48,675] INFO Everyone WakeUp and Work!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:69)
[2018-01-25 22:41:48,706] INFO Thread 1 Polling! (com.huawei.dms.kafka.DmsKafkaProduceDemo:119)
[2018-01-25 22:41:48,706] INFO Thread1 KEEP Poll records!
(com.huawei.dms.kafka.DmsKafkaProduceDemo:128)
```


2 Improving Message Processing Efficiency

During message sending and consumption, DMS for Kafka, producers, and consumers collaborate to ensure service reliability. In addition, developers must use DMS for Kafka topics properly to improve the efficiency and accuracy of message sending and consumption.

Best practices for message producers and consumers are as follows:

Acknowledging Message Production and Consumption

Message production (sending)

The producer decides whether to re-send a message based on the DMS for Kafka response.

The producer waits for the sending result or asynchronous callback function to determine if the message is successfully sent. If an exception occurs when sending the message, the producer will not receive a success response and must decide whether to re-send the message. If a success response is received, it indicates that the message has been stored in DMS for Kafka.

Message consumption

The consumer acknowledges successful message consumption.

The produced messages are sequentially stored in DMS for Kafka. During consumption, messages stored in DMS for Kafka are obtained in sequence. Consumers obtain messages, consume them, and record the status (successful or failed). The status is then submitted to DMS for Kafka.

During this process, the message consumption status may not be successfully submitted due to an exception. In this case, the corresponding messages will be re-obtained by the consumer in the next message consumption request.

Idempotent Transferring of Message Production and Consumption

To guarantee lossless messaging, DMS for Kafka implements a series of reliability measures. For example, the message synchronization storage mechanism is used to prevent the system and server from being abnormally restarted or powered off. The ACK mechanism is used to deal with exceptions that occur during message transmission.

Considering extreme conditions such as network exceptions, you can use DMS for Kafka to design message sending and consumption in addition to acknowledging message production and consumption.

- If message sending cannot be acknowledged, the producer needs to re-send the message.
- After consuming a message that has been processed, the consumer needs to notify DMS for Kafka that consumption is successful and ensure that the message is not processed repeatedly.

Producing and Consuming Messages in Batches

It is recommended that messages be sent and consumed in batches to improve efficiency.

Figure 2-1 Messages being produced (sent) and consumed in batches

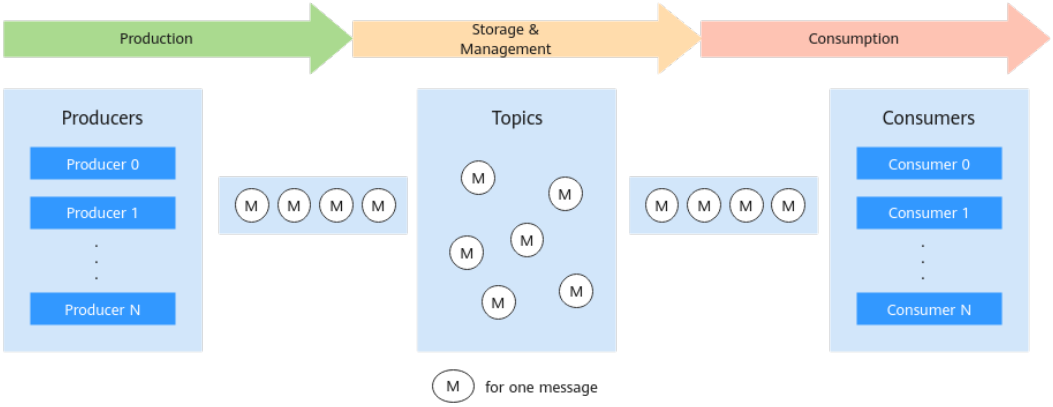
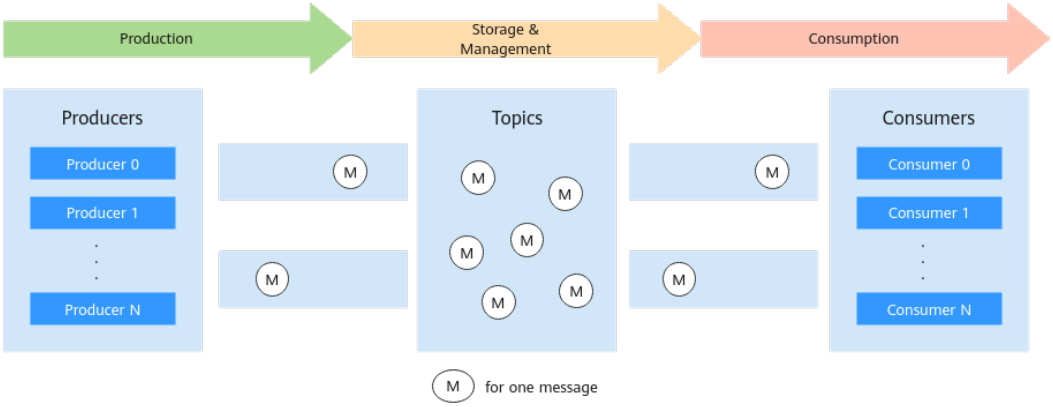


Figure 2-2 Messages being produced (sent) and consumed one by one



When consuming messages in batches, consumers need to process and acknowledge messages in the sequence of receiving messages. Therefore, when a message in the batch fails to be consumed, the consumer does not need to consume the remaining messages, and directly submit consumption acknowledgment of the successfully consumed messages.

Using Consumer Groups to Facilitate O&M

You can use DMS for Kafka as a message management system. Reading message content from topics is helpful to fault locating and service debugging.

When problems occur during message production and consumption, you can create different consumer groups to locate and analyze problems or debug services for interconnecting with other services. To ensure that other services can continue to process messages in topics, you can create a new consumer group to consume and analyze the messages.

3 Migrating Kafka Services

Scenario

You can migrate Kafka services to connect message producers and consumers to a new Kafka instance and can even migrate persisted message data to the new Kafka instance. Kafka services can be migrated in the following two scenarios:

- Migrating services to the cloud without downtime

  Services that have high requirements on continuity must be smoothly migrated to the cloud because they cannot afford a long downtime.
- Re-deploying services in the cloud

  A Kafka instance deployed within an AZ is not capable of cross-AZ disaster recovery. For higher reliability, you can re-deploy services to an instance that is deployed across AZs.

Preparation

1. Configure the network environment.

   A Kafka instance can be accessed within a VPC or over a public network. For public network access, the producer and consumer must have public access permissions, and the following security group rules must be configured.

   Table 3-1 Security group rules

Direction	Protocol	Port	Source	Description
Inbound	TCP	9094	0.0.0.0/0	Access Kafka through the public network (without SSL encryption).
Inbound	TCP	9095	0.0.0.0/0	Access Kafka through the public network (with SSL encryption).

2. Create a Kafka instance.

   The specifications of the new instance cannot be lower than the original specifications. For details, see **Buying an Instance**.

3. Create a topic.

   Create a topic with the same configurations as the original Kafka instance, including the topic name, number of replicas, number of partitions, message aging time, and whether to enable synchronous replication and flushing. For details, see **Creating a Topic**.

Migration Scheme 1: Migrating the Production First

Migrate the message production service to the new Kafka instance. After migration, the original Kafka instance will no longer produce messages. After all messages of the original Kafka instance are consumed, migrate the message consumption service to the new Kafka instance to consume messages of this instance.

Step 1 Change the Kafka connection address of the producer to that of the new Kafka instance.

Step 2 Restart the production service so that the producer can send new messages to the new Kafka instance.

Step 3 Check the consumption progress of each consumer group in the original Kafka instance until all data in the original Kafka instance is consumed.

Step 4 Change the Kafka connection addresses of the consumers to that of the new Kafka instance.

Step 5 Restart the consumption service so that consumers can consume messages from the new Kafka instance.

Step 6 Check whether consumers consume messages properly from the new Kafka instance.

Step 7 The migration is completed.

----End

This is a common migration scheme. It is simple and easy to control on the service side. During the migration, the message sequence is ensured, so this scheme is **suitable for scenarios with strict requirements on the message sequence**. However, latency may occur because there is a period when you have to wait for all data to be consumed.

Migration Scheme 2: Migrating the Production Later

Use multiple consumers for the consumption service. Some consume messages from the original Kafka instance, and others consume messages from the new Kafka instances. Then, migrate the production service to the new Kafka instance so that all messages can be consumed in time.

Step 1 Start new consumer clients, set the Kafka connection addresses to that of the new Kafka instance, and consume data from the new Kafka instance.

NOTE

Original consumer clients must continue running. Messages are consumed from both the original and new Kafka instances.

Step 2 Change the Kafka connection address of the producer to that of the new Kafka instance.

Step 3 Restart the producer client to migrate the production service to the new Kafka instance.

Step 4 After the production service is migrated, check whether the consumption service connected to the new Kafka instance is normal.

Step 5 After all data in the original Kafka is consumed, close the original consumption clients.

Step 6 The migration is completed.

----End

In this scheme, the migration process is controlled by services. For a certain period of time, the consumption service consumes messages from both the original and new Kafka instances. Before the migration, message consumption from the new Kafka instance has already started, so there is no latency. However, early on in the migration, data is consumed from both the original and new Kafka instances, so the messages may not be consumed in the order that they are produced. This scheme is **suitable for services that require low latency but do not require strict message sequence**.

How Do I Migrate Persisted Data Along with Services?

You can migrate consumed data from the original instance to a new instance by using the open-source tool **MirrorMaker**. This tool mirrors the original Kafka producer and consumer into new ones and migrates data to the new Kafka instance. For details, see **Using MirrorMaker to Synchronize Data Across Clusters**.

Note that each HUAWEI CLOUD Kafka instance stores data in three replicas. Therefore, the storage space of the new instance should be three times that of the original single-replica message storage.

4 Using MirrorMaker to Synchronize Data Across Clusters

Scenario

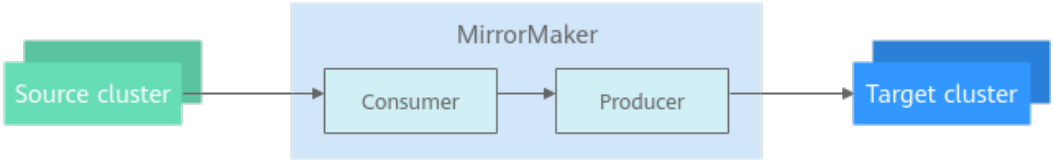
In the following scenarios, MirrorMaker can be used to synchronize data between different Kafka clusters to ensure the availability and reliability of the clusters:

- Backup and disaster recovery: An enterprise has multiple data centers. To prevent service unavailability caused by a fault in one data center, cluster data is synchronously backed up in multiple data centers.
- Cluster migration: As enterprises migrate services to the cloud, data in on-premises clusters must be synchronized with that in cloud clusters to ensure service continuity.

Solution Architecture

MirrorMaker can be used to mirror data from the source cluster to the target cluster. As shown in **Figure 4-1**, in essence, MirrorMaker first consumes data from the source cluster and then produces the consumed data to the target cluster. For more information about MirrorMaker, see **Mirroring data between clusters**.

Figure 4-1 How MirrorMaker works



Restrictions

- The IP addresses and port numbers of the nodes in the source cluster cannot be the same as those of the nodes in the target cluster. Otherwise, data will be replicated infinitely in a topic.
- Use MirrorMaker to synchronize data between at least two clusters. If there is only one cluster, data will be replicated infinitely in a topic.

Procedure

Step 1 Buy an ECS that can communicate with the source and target clusters. For details, see the **ECS documentation**.

Step 2 Log in to the ECS, install JDK, and add the following contents to **.bash_profile** in the home directory to configure the environment variables **JAVA_HOME** and **PATH**. In this command, **/opt/java/jdk1.8.0_151** is the JDK installation path. Change it to the path where you install JDK.

```
export JAVA_HOME=/opt/java/jdk1.8.0_151
export PATH=$JAVA_HOME/bin:$PATH
```

Run the **source .bash_profile** command for the modification to take effect.

NOTE

Use Oracle JDK instead of ECS's default JDK (for example, OpenJDK), because ECS's default JDK may not be suitable. Obtain Oracle JDK 1.8.111 or later from **Oracle's official website**.

Step 3 Download the binary software package of Kafka 3.3.1.

```
wget https://archive.apache.org/dist/kafka/3.3.1/kafka_2.12-3.3.1.tgz
```

Step 4 Decompress the binary software package.

```
tar -zxvf kafka_2.12-3.3.1.tgz
```

Step 5 Go to the binary software package directory and specify the IP addresses and ports of the source and target clusters and other parameters in the **connect-mirror-maker.properties** configuration file in the **config** directory.

```
# Specify two clusters.
clusters = A, B
A.bootstrap.servers = A_host1:A_port, A_host2:A_port, A_host3:A_port
B.bootstrap.servers = B_host1:B_port, B_host2:B_port, B_host3:B_port

# Specify the data synchronization direction. The data can be synchronized unidirectionally or bidirectionally.
A->B.enabled = true

# Specify the topics to be synchronized. Regular expressions are supported. By default, all topics are replicated, for example, foo-.*.
A->B.topics = .*

# If the following two configurations are enabled, clusters A and B replicate data with each other.
#B->A.enabled = true
#B->A.topics = .*

# Specify the number of replicas. If multiple topics need to be synchronized and their replica quantities are different, create topics with the same name and replica quantity before starting MirrorMaker.
replication.factor=3

# Specify the consumer offset synchronization direction (unidirectionally or bidirectionally).
A->B.sync.group.offsets.enabled=true

############################## Internal Topic Settings  ##############################
# The replication factor for mm2 internal topics "heartbeats", "B.checkpoints.internal" and
# "mm2-offset-syncs.B.internal"
# In the test environment, the value can be 1. In the production environment, it is recommended that the value be greater than 1, for example, 3.
checkpoints.topic.replication.factor=3
heartbeats.topic.replication.factor=3
offset-syncs.topic.replication.factor=3

# The replication factor for connect internal topics "mm2-configs.B.internal", "mm2-offsets.B.internal" and
# "mm2-status.B.internal"
# In the test environment, the value can be 1. In the production environment, it is recommended that the value be greater than 1, for example, 3.
```



```
offset.storage.replication.factor=3
status.storage.replication.factor=3
config.storage.replication.factor=3

# customize as needed
# replication.policy.separator = _
# sync.topic.acls.enabled = false
# emit.heartbeats.interval.seconds = 5
```

Step 6 In the binary software package directory, start MirrorMaker to synchronize data.

```
./bin/connect-mirror-maker.sh config/connect-mirror-maker.properties
```

Step 7 (Optional) If a topic is created in the source cluster after MirrorMaker has been started, and the topic data needs to be synchronized, restart MirrorMaker. For details about how to restart MirrorMaker, see **Step 6**. You can also add configurations listed in **Table 4-1** to periodically synchronize new topics without restarting MirrorMaker. **refresh.topics.interval.seconds** is mandatory. Other parameters are optional.

Table 4-1 MirrorMaker configurations

Parameter	Default Value	Description
sync.topic.configs.enabled	true	Whether to monitor the source cluster for configuration changes.
sync.topic.acls.enabled	true	Whether to monitor the source cluster for ACL changes.
emit.heartbeats.enabled	true	Whether to let the connector send heartbeats periodically.
emit.heartbeats.interval.seconds	5 seconds	Heartbeat frequency.
emit.checkpoints.enabled	true	Whether to let the connector periodically send the consumer offset information.
emit.checkpoints.interval.seconds	5 seconds	Checkpoint frequency.
refresh.topics.enabled	true	Whether to let the connector periodically check for new topics.
refresh.topics.interval.seconds	5 seconds	Frequency of checking for new topics in the source cluster.
refresh.groups.enabled	true	Whether to let the connector periodically check for new consumer groups.
refresh.groups.interval.seconds	5 seconds	Frequency of checking for new consumer groups in the source cluster.

Parameter	Default Value	Description
replication.policy.class	org.apache.kafka.connect.mirror.DefaultReplicationPolicy	Use LegacyReplicationPolicy to imitate MirrorMaker of an earlier version.
heartbeats.topic.retention.ms	One day	Used when heartbeat topics are created for the first time.
checkpoints.topic.retention.ms	One day	Used when checkpoint topics are created for the first time.
offset.syncs.topic.retention.ms	max long	Used when offset sync topics are created for the first time.

----End

Verifying Data Synchronization

Step 1 View the topic list in the target cluster to check whether there are source topics.

NOTE

Topic names in the target cluster have a prefix (for example, **A.**) added to the source topic name. This is a MirrorMaker 2 configuration for preventing cyclic topic backup.

Step 2 Produce and consume messages in the source cluster, view the consumption progress in the target cluster, and check whether data has been synchronized from the source cluster to the target cluster.

If the target cluster is a Huawei Cloud Kafka instance, view the consumption progress on the **Consumer Groups** page.

----End

5 Setting Parameters for Kafka Clients

This section provides recommendations on configuring common parameters for Kafka producers and consumers. For details about other parameters, see the **Kafka official website**.

Table 5-1 Producer parameters

Parameter	Default Value	Recommended Value	Description
acks	1	**all** or **–1** (if high reliability mode is selected) **1** (if high throughput mode is selected)	Number of acknowledgments the producer requires the server to return before considering a request complete. This controls the durability of records that are sent. Options: **0**: The producer will not wait for any acknowledgment from the server at all. The record will be immediately added to the socket buffer and considered sent. No guarantee can be made that the server has received the record, and the retries configuration will not take effect (as the client generally does not know of any failures). The offset given back for each record will always be set to –1. **1**: The leader will write the record to its local log but will respond without waiting until receiving full acknowledgement from all followers. If the leader fails immediately after acknowledging the record but before the followers have replicated it, the record will be lost. **all** or **–1**: The leader will wait for the full set of replicas to acknowledge the record. This is the strongest available guarantee because the record will not be lost even if there is just one replica that works. **min.insync.replicas** specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful.
retries	0	Set as required.	Number of times that the client resends a message. Setting this parameter to a value greater than zero will cause the client to resend any record that failed to be sent. Note that this retry is no different than if the client resent the record upon receiving the error. Allowing retries will potentially change the ordering of records because if two batches are sent to the same partition, and the first fails and is retried but the second succeeds, then the records in the second batch may appear first. You are advised to configure producers so that they can be able to retry in case of network disconnections. Set **retries** to **3** and the retry interval **retry.backoff.ms** to **1000**.

Parameter	Default Value	Recommended Value	Description
request.timeout.ms	30000	Set as required.	Maximum amount of time (in ms) the client will wait for the response of a request. If the response is not received before the timeout elapses, the client will throw a timeout exception. Setting this parameter to a large value, for example, **127000** (127s), can prevent records from failing to be sent in high-concurrency scenarios.
block.on.buffer.full	TRUE	TRUE	Setting this parameter to **TRUE** indicates that when buffer memory is exhausted, the producer must stop receiving new message records or throw an exception. By default, this parameter is set to **TRUE**. However, in some cases, non-blocking usage is desired and it is better to throw an exception immediately. Setting this parameter to **FALSE** will cause the producer to instead throw "BufferExhaustedException" when buffer memory is exhausted.
batch.size	16384	262144	Default maximum number of bytes of messages that can be processed at a time. The producer will attempt to batch records together into fewer requests whenever multiple records are being sent to the same partition. This improves performance of both the client and the server. No attempt will be made to batch records larger than this size. Requests sent to brokers will contain multiple batches, one for each partition with data available to be sent. A smaller batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). A larger batch size may use more memory as a buffer of the specified batch size will always be allocated in anticipation of additional records.

Parameter	Default Value	Recommended Value	Description
buffer.memory	33554432	67108864	Total bytes of memory the producer can use to buffer records waiting to be sent to the server. If data is generated faster than it is sent to the broker, the producer blocks or throw a "block.on.buffer.full" exception. This setting should correspond roughly to the total memory the producer will use, but is not a rigid bound since not all memory the producer uses is used for buffering. Some additional memory will be used for compression (if compression is enabled) as well as for maintaining in-flight requests.

Table 5-2 Consumer parameters

Parameter	Default Value	Recommended Value	Description
auto.commit.enable	TRUE	FALSE	If this parameter is set to **TRUE**, the offset of messages already fetched by the consumer will be periodically committed to ZooKeeper. This committed offset will be used when the process fails as the position from which the new consumer will begin. Constraints: If this parameter is set to **FALSE**, to avoid message loss, an offset must be committed to ZooKeeper after the messages are successfully consumed.

Parameter	Default Value	Recommended Value	Description
auto.offset.reset	latest	earliest	Indicates what to do when there is no initial offset in ZooKeeper or if the current offset has been deleted. Options: • **earliest**: Automatically reset to the smallest offset. • **latest**: Automatically reset to the largest offset. • **none**: The system throws an exception to the consumer if no offset is available. • **anything else**: The system throws an exception to the consumer. **NOTE** If this parameter is set to **latest**, the producer may start to send messages to new partitions (if any) before the consumer resets to the initial offset. As a result, some messages will be lost.
connections.max.idle.ms	600000	30000	Timeout interval (in ms) for an idle connection. The server closes the idle connection after this period of time ends. Setting this parameter to **30000** can reduce the server response failures when the network condition is poor.

6 Using Kafka Clients

Consumers

1. Ensure that the owner thread does not exit abnormally. Otherwise, the client may fail to initiate consumption requests and the consumption will be blocked.
2. Commit messages only after they have been processed. Otherwise, the messages may fail to be processed and cannot be polled again.
3. If there is a large number of OFFSET_COMMIT requests, CPU usage will be high. For example, if a consumption request pulls 1000 messages and each message is committed separately, the commit TPS will be 1000 times that of the consumption request. The smaller the message body, the larger the difference. Therefore, you are not advised to commit every message separately. You can commit a specific number of messages in batches or enable **enable.auto.commit**. However, if the client is faulty, some cached consumption offset may be lost, resulting in repeated consumption. Therefore, you are advised to commit messages in batches based on service requirements.
4. A consumer cannot frequently join or leave a group. Otherwise, the consumer will frequently perform rebalancing, which blocks consumption.
5. The number of consumers cannot be greater than the number of partitions in the topic. Otherwise, some consumers may fail to poll for messages.
6. Ensure that the consumer polls at regular intervals to keep sending heartbeats to the server. If the consumer stops sending heartbeats for long enough, the consumer session will time out and the consumer will be considered to have stopped. This will also block consumption.
7. Ensure that there is a limitation on the size of messages buffered locally to avoid an out-of-memory (OOM) situation.
8. Set the timeout for the consumer session to 30 seconds: session.timeout.ms=30000.
9. Kafka supports exactly-once delivery. Therefore, ensure the idempotency of processing messages for services.
10. Always close the consumer before exiting. Otherwise, consumers in the same group may be blocked within the timeout set by **session.timeout.ms**.

11. Do not start a consumer group name with a special character, such as a number sign (#). Otherwise, monitoring data of the consumer group cannot be displayed.

Producers

1. Synchronous replication: Set **acks** to **all**.
2. Retry message sending: Set **retries** to **3**.
3. Optimize message sending: For latency-sensitive messages, set **linger.ms** to **0**. For latency-insensitive messages, set **linger.ms** to a value ranging from **100** to **1000**.
4. Ensure that the producer has sufficient JVM memory to avoid blockages.
5. Set the timestamp to the local time. Messages will fail to age if the timestamp is a future time.

Topics

Recommended topic configurations: Use 3 replicas, enable synchronous replication, and set the minimum number of in-sync replicas to 2. The number of in-sync replicas cannot be the same as the number of replicas of the topic. Otherwise, if one replica is unavailable, messages cannot be produced.

You can enable or disable automatic topic creation. If it is enabled, a topic will be automatically created with 3 partitions and 3 replicas when a message is produced in or consumed from a topic that does not exist.

The recommended maximum number of partitions for a topic is 100.

Each topic can have 3 replicas (the number of replicas cannot be modified once configured).

Other Suggestions

Maximum number of connections: 3000

Maximum size of a message: 10 MB

Access Kafka using SASL_SSL. Ensure that your DNS service is capable of resolving an IP address to a domain name. Alternatively, map all Kafka broker IP addresses to host names in the **hosts** file. Prevent Kafka clients from performing reverse resolution. Otherwise, connections may fail to be established.

Apply for a disk space size that is more than twice the size of service data multiplied by the number of replicas. In other words, keep 50% of the disk space idle.

Avoid frequent full GC in JVM. Otherwise, message production and consumption will be blocked.

7 Configuring an Alarm Rule for Accumulated Messages

Scenario

Configure alarm rules so that you will be notified when the number of accumulated messages in a consumer group exceeds the threshold.

The procedure described in this section can also be applied to setting alarm rules for **other metrics**.

Prerequisites

You have **purchased a Kafka instance**, **created a topic**, and there are available messages.

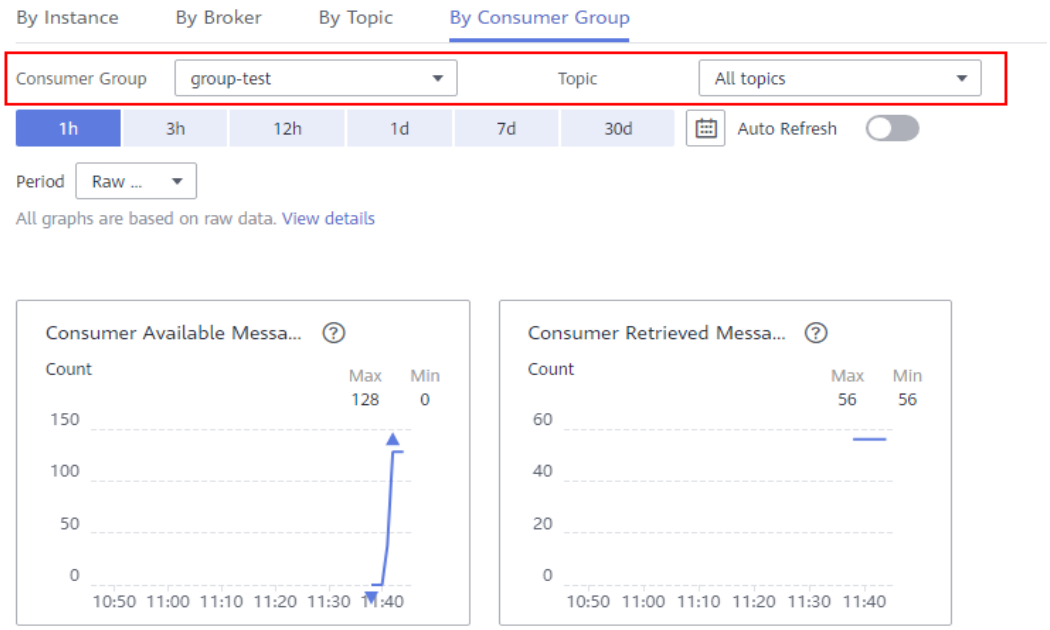
Procedure

Step 1 Log in to the console of DMS for Kafka. Click the instance to be configured with an alarm rule.

Step 2 In the left navigation pane, choose **Monitoring**.

Step 3 On the **By Consumer Group** tab page, select the consumer group for which you want to create an alarm rule.

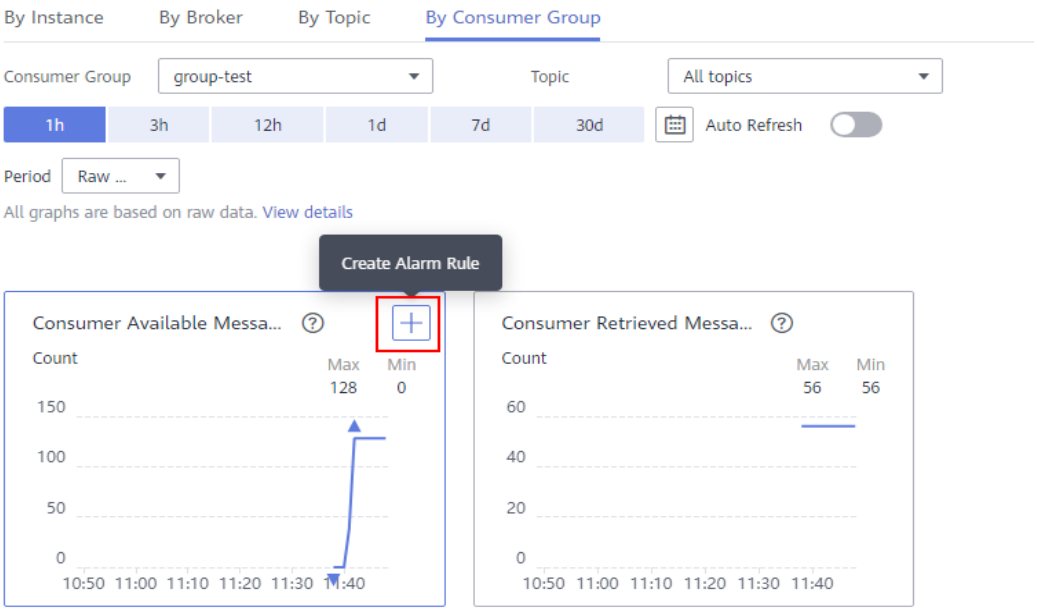
Figure 7-1 Selecting a consumer group



- **Consumer Group**: Select the consumer group for which you want to create an alarm rule.
- **Topic**: Select **All topics**.

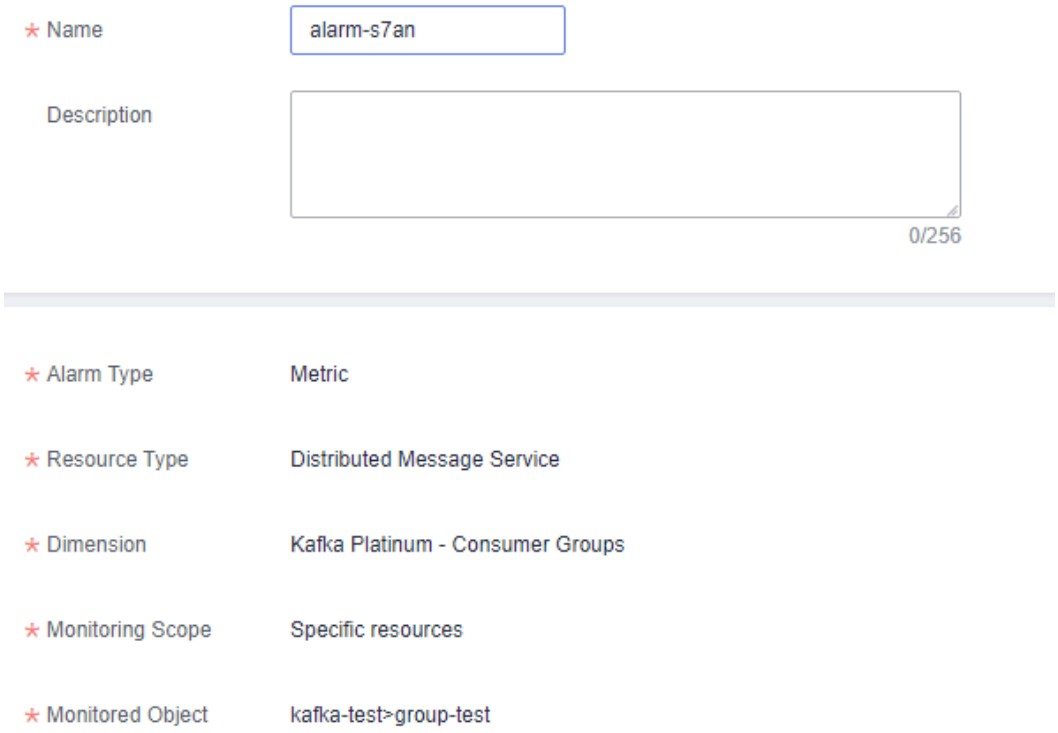
Step 4 Hover the mouse pointer over **Consumer Available Messages** and click  .

Figure 7-2 Consumer available messages chart



Step 5 On the **Create Alarm Rule** page, configure the basic information of the alarm rule.

Figure 7-3 Configuring the basic information of the alarm rule



- **Name**: name of the alarm rule
- (Optional) **Description**: description of the alarm rule

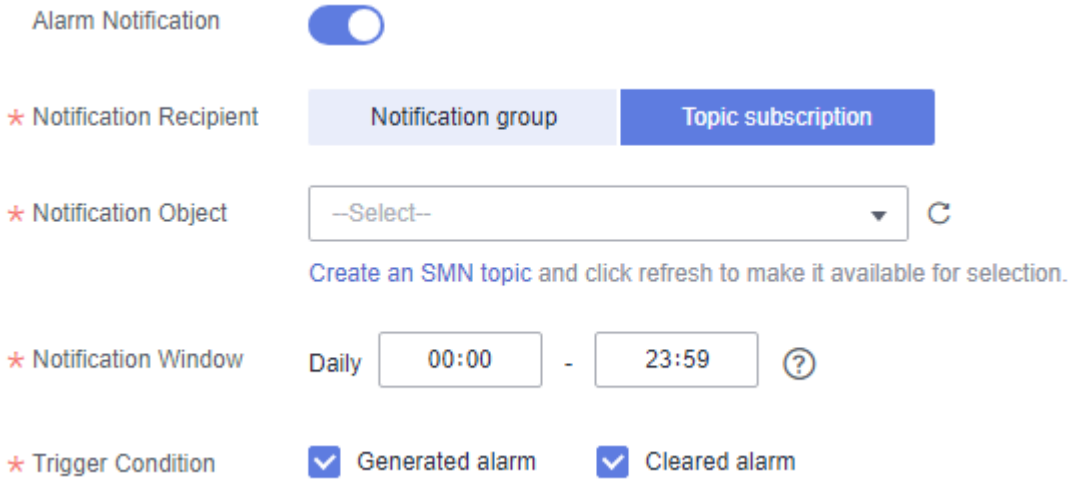
Step 6 Configure the alarm policy.

Figure 7-4 Configuring the alarm policy

- **Method**: Select **Configure manually**.
- **Alarm Policy**: Specify the conditions for triggering an alarm. An alarm will be triggered if the metric data in the specified number of consecutive periods reaches the specified threshold.
- **Alarm Severity**: Select an alarm severity as required.

Step 7 Configure the alarm notification.

Figure 7-5 Configuring the alarm notification



- **Alarm Notification**: Enable this option.
- **Notification Recipient**: Select **Topic subscription**.
- **Notification Object**: Select a cloud account contact or a created alarm notification topic. An alarm notification topic contains the mobile number or email address receiving the notification.

  If no alarm notification topics are available, click **Create an SMN topic**. On the SMN console, **create a topic** and **add a subscription**. After the alarm notification topic is created, go back to the **Create Alarm Rule** page, click to make the created topic available for selection.

  NOTE

  After the subscription is added, the corresponding subscription endpoint will receive a subscription notification. You need to confirm the subscription so that the endpoint can receive alarm notifications.

Figure 7-6 Creating an alarm notification topic

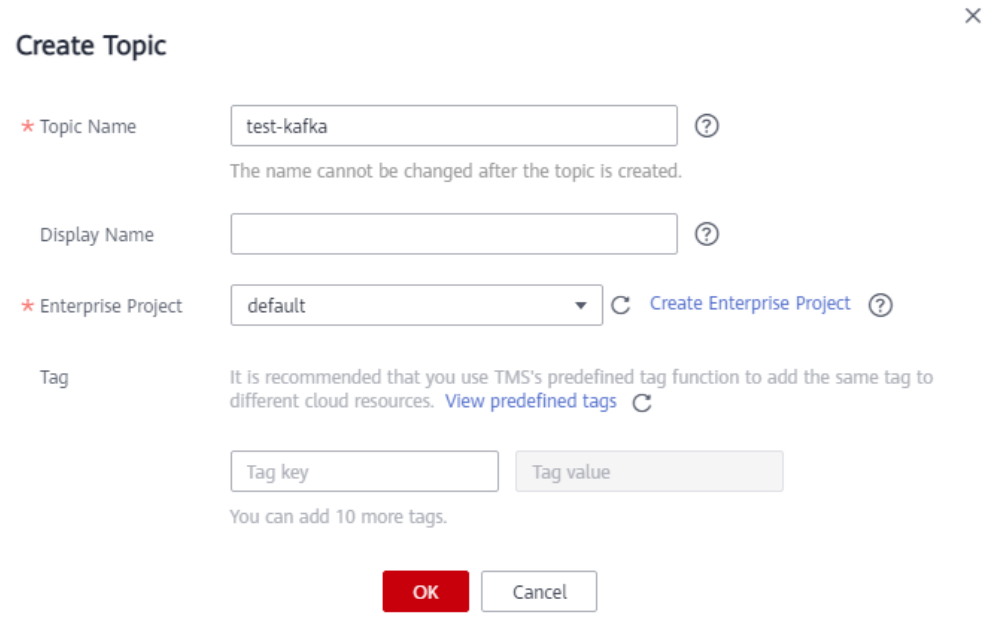
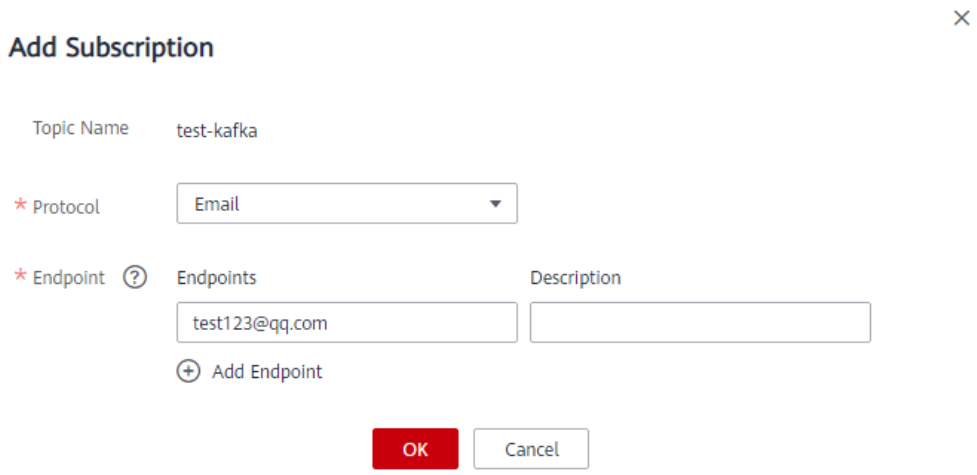


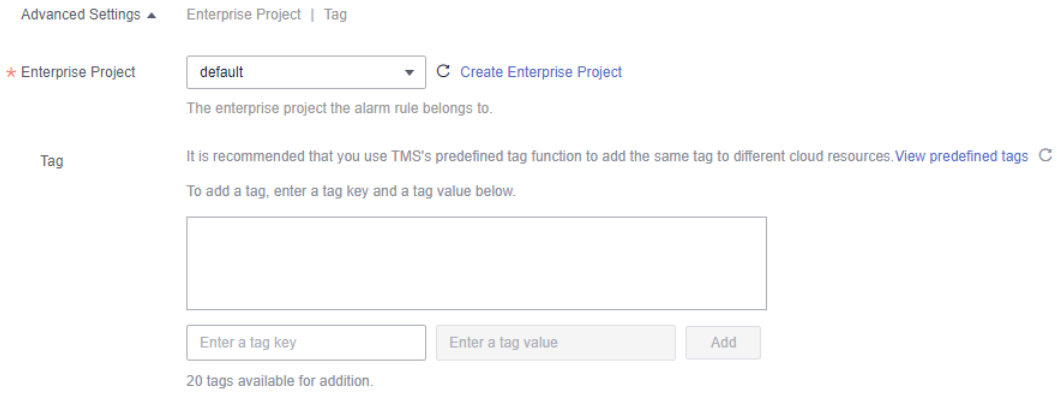
Figure 7-7 Adding a subscription



- **Validity Period**: Cloud Eye sends notifications only within the validity period specified in the alarm rule.
- **Trigger Condition**: condition for triggering an alarm notification.

Step 8 Configure the enterprise project and tag.

Figure 7-8 Configuring the enterprise project and tag.

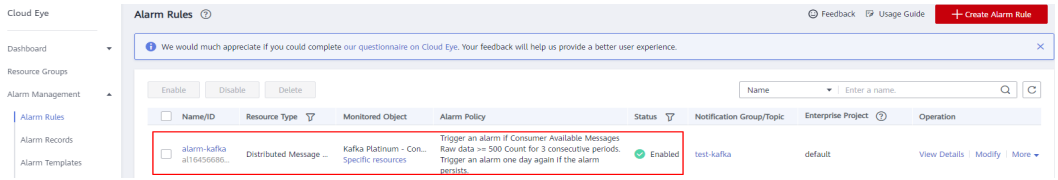


- **Enterprise Project**: enterprise project with which the alarm rule is associated. Only users who have the permissions of the enterprise project can view and manage the alarm rule.
- **Tag**: tags are used to identify cloud resources. When you have many cloud resources of the same type, you can use tags to classify cloud resources by dimension (for example, usage, owner, or environment).

Step 9 Click **Create**.

After the alarm rule is created, you can view it on the **Alarm Management** > **Alarm Rules** page.

Figure 7-9 Viewing the new alarm rule



----End

8 Interconnecting Logstash with Kafka

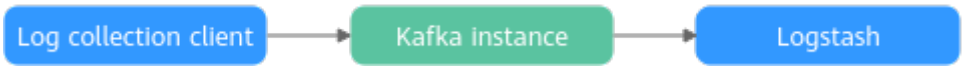
Scenario

Logstash is a free and open server-side data processing pipeline that integrates data from multiple sources, converts it, and then sends it to the specified storage. Kafka is a high-throughput distributed message pub/sub system. It is one of the input and output sources of Logstash. The following describes how to interconnect Logstash with a Kafka instance.

Solution Architecture

- The following figure shows Kafka as an input source of Logstash.

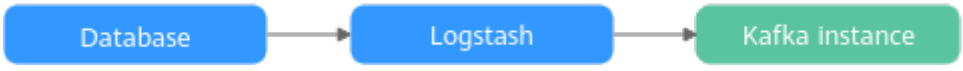
  Figure 8-1 Kafka as an input source of Logstash

  The log collection client sends data to the Kafka instance. Logstash pulls data from the Kafka instance based on its performance. Using a Kafka instance as the Logstash input source can prevent the impact of burst traffic on Logstash, and decouple the log collection client from Logstash to ensure system stability.

- The following figure shows Kafka as an output source of Logstash.

  Figure 8-2 Kafka as an output source of Logstash

  Logstash collects data from the database and sends the data to the Kafka instance for storage. Using a Kafka instance as the Logstash output source can store a large amount of data thanks to the high throughput of Kafka.

Restrictions

Logstash 7.5 and later versions support Kafka Integration Plugin which includes the Kafka input plugin and Kafka output plugin. Kafka input plugin reads data from topics of Kafka instances, and Kafka output plugin writes data to topics of

Kafka instances. **Table 8-1** lists the version mapping between Logstash, Kafka Integration Plugin, and Kafka clients. **Ensure that the Kafka client version is later than or equal to the Kafka instance version.**

Table 8-1 Version mapping

Logstash Version	Kafka Integration Plugin Version	Kafka Client Version
8.3–8.8	10.12.0	2.8.1
8.0–8.2	10.9.0–10.10.0	2.5.1
7.12–7.17	10.7.4–10.9.0	2.5.1
7.8–7.11	10.2.0–10.7.1	2.4
7.6–7.7	10.0.1	2.3.0
7.5	10.0.0	2.1.0

Prerequisites

Make the following preparation before implementation.

- **Download Logstash**.
- Prepare a Windows host, install **JDK v1.8.111 or later** and Git Bash on the host, and configure related environment variables.
- **Create a Kafka instance** and **a topic**, and obtain the instance information.

  If both public access and SASL authentication are disabled for the Kafka instance, obtain the information listed in **Table 8-2**.

  Table 8-2 Kafka instance information (public access and SASL authentication disabled)

Parameter	How to Obtain
Instance address (private network)	View it in the **Connection** area on the instance details page.
Topic name	On the Kafka console, click your instance. In the left navigation pane, choose **Topics** to view the topic name. The following uses **topic-logstash** as an example.

  If public access is disabled and SASL authentication is enabled for the Kafka instance, obtain the information listed in **Table 8-3**.

Table 8-3 Kafka instance information (public access disabled and SASL authentication enabled)

Parameter	How to Obtain
Instance address (private network)	View it in the **Connection** area on the instance details page.
SASL mechanism	View it in the **Connection** area on the instance details page.
Security protocol	View it in the **Connection** area on the instance details page.
Certificate	Click **Download** next to **SSL Certificate** in the **Connection** area on the instance details page. Download and decompress the package to obtain the client certificate file **client.truststore.jks**.
SASL username and password	On the Kafka console, click your instance. In the left navigation pane, choose **Users** to view the username. If you have forgotten the password, click **Reset Password**.
Topic name	On the Kafka console, click your instance. In the left navigation pane, choose **Topics** to view the topic name. The following uses **topic-logstash** as an example.

If public access is enabled and SASL authentication is disabled for the Kafka instance, obtain the information listed in **Table 8-4**.

Table 8-4 Kafka instance information (public access enabled and SASL authentication disabled)

Parameter	How to Obtain
Instance address (public network)	View it in the **Connection** area on the instance details page.
Topic name	On the Kafka console, click your instance. In the left navigation pane, choose **Topics** to view the topic name. The following uses **topic-logstash** as an example.

If both public access and SASL authentication are enabled for the Kafka instance, obtain the information listed in **Table 8-5**.

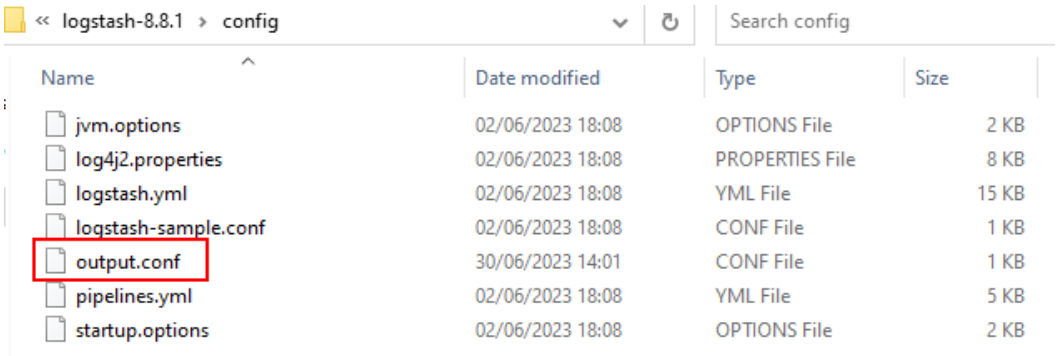
Table 8-5 Kafka instance information (public access and SASL authentication enabled)

Parameter	How to Obtain
Instance address (public network)	View it in the **Connection** area on the instance details page.
SASL mechanism	View it in the **Connection** area on the instance details page.
Security protocol	View it in the **Connection** area on the instance details page.
Certificate	Click **Download** next to **SSL Certificate** in the **Connection** area on the instance details page. Download and decompress the package to obtain the client certificate file **client.truststore.jks**.
SASL username and password	On the Kafka console, click your instance. In the left navigation pane, choose **Users** to view the username. If you have forgotten the password, click **Reset Password**.
Topic name	On the Kafka console, click your instance. In the left navigation pane, choose **Topics** to view the topic name. The following uses **topic-logstash** as an example.

Procedure (Kafka Instance as the Logstash Output Source)

Step 1 On the Windows host, decompress the Logstash package, go to the **config** folder, and create the **output.conf** configuration file.

Figure 8-3 Creating the output.conf configuration file



Step 2 Add the following content to the **output.conf** file:

```
input {
    stdin {}
}
```



```
output {
 kafka {
    bootstrap_servers => "ip1:port1,ip2:port2,ip3:port3"
    topic_id => "topic-logstash"

   # If SASL authentication is disabled, comment out the following options:
    # If the SASL mechanism is PLAIN, configure as follows:
    sasl_mechanism => "PLAIN"
    sasl_jaas_config => "org.apache.kafka.common.security.plain.PlainLoginModule required
username='username' password='password';"

    # If the SASL mechanism is SCRAM-SHA-512, configure as follows:
    sasl_mechanism => "SCRAM-SHA-512"
    sasl_jaas_config => "org.apache.kafka.common.security.scram.ScramLoginModule required
username='username' password='password';"

    # If the security protocol is SASL_SSL, configure as follows:
    security_protocol => "SASL_SSL"
    ssl_truststore_location => "C:\\Users\\Desktop\\logstash-8.8.1\\config\\client.jks"
    ssl_truststore_password => "dms@kafka"
    ssl_endpoint_identification_algorithm => ""

   # If the security protocol is SASL_PLAINTEXT, configure as follows:
    security_protocol => "SASL_PLAINTEXT"
    }
}
```

Description:

- **bootstrap_servers**: private network connection address or public network connection address of the Kafka instance.
- **topics**: topic name.
- **sasl_mechanism**: SASL authentication mechanism.
- **sasl_jaas_config**: SASL JAAS configuration file. Change the SASL username and password as required.
- **security_protocol**: security protocol used by the Kafka instance.
- **ssl.truststore.location**: location where the SSL certificate is stored.
- **ssl_truststore_password**: server certificate password, which must be set to **dms@kafka** and cannot be changed.
- **ssl_endpoint_identification_algorithm**: Indicates whether to verify the certificate domain name. If this option is left blank, the certificate domain name is not verified. **In this example, leave it blank**.

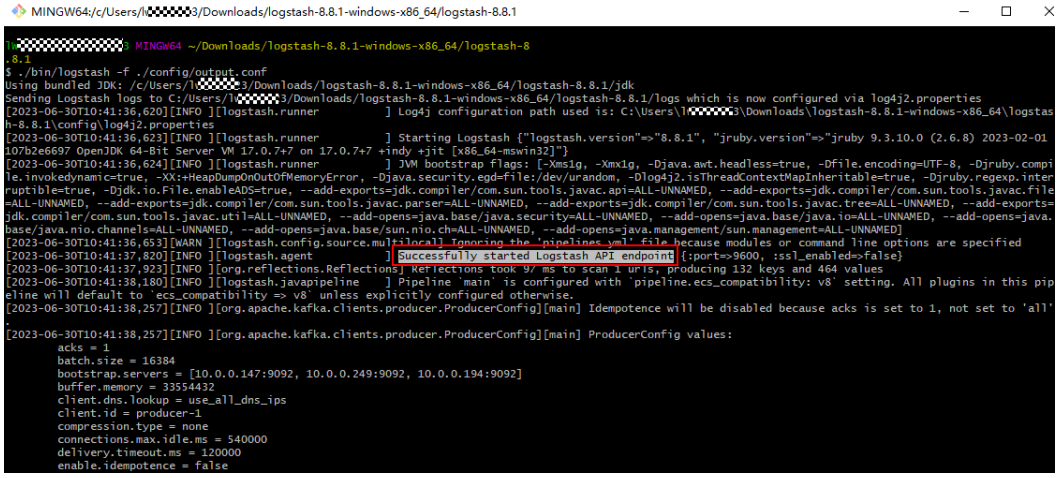
For more information about Kafka output plugin options, see **Kafka output plugin**.

Step 3 Open Git Bash in the **root** directory of the Logstash folder and run the following command to start Logstash:

```
./bin/logstash -f ./config/output.conf
```

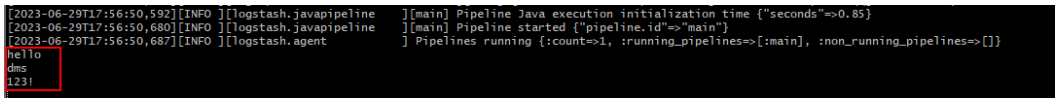
If the message "Successfully started Logstash API endpoint" is displayed, Logstash has been started.

Figure 8-4 Starting Logstash



Step 4 In Logstash, produce messages, as shown in the following figure.

Figure 8-5 Producing messages

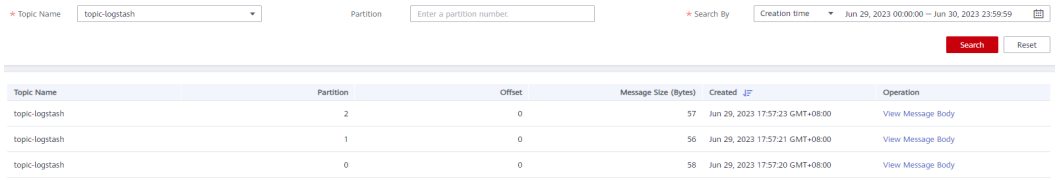


Step 5 Go to the Kafka console and click your instance.

Step 6 In the left navigation pane, choose **Message Query**.

Step 7 Select **topic-logstash** from the **Topic Name** drop-down list box and click **Search** to query messages.

Figure 8-6 Querying messages



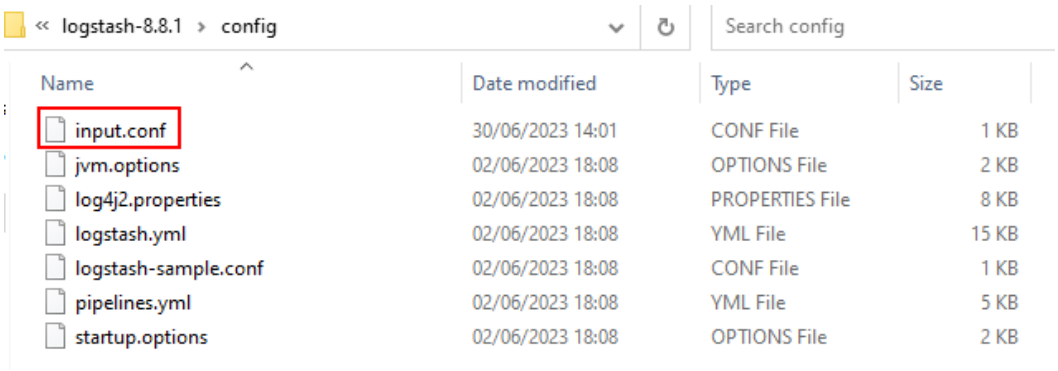
As shown in **Figure 8-6**, the Kafka output plugin of Logstash has written data to **topic-logstash** of the Kafka instance.

----End

Procedure (Kafka Instance as the Logstash Input Source)

Step 1 On the Windows host, decompress the Logstash package, go to the **config** folder, and create the **input.conf** configuration file.

Figure 8-7 Creating the input.conf configuration file



Step 2 Add the following content to the **input.conf** file to connect to the Kafka instance:

```
input {
 kafka {
    bootstrap_servers => "ip1:port1,ip2:port2,ip3:port3"
    group_id => "logstash_group"
    topic_id => "topic-logstash"
    auto_offset_reset => "earliest"

   # If SASL authentication is disabled, comment out the following options:
    #If the SASL mechanism is PLAIN, configure as follows:
    sasl_mechanism => "PLAIN"
    sasl_jaas_config => "org.apache.kafka.common.security.plain.PlainLoginModule required
username='username' password='password';"

    # If the SASL mechanism is SCRAM-SHA-512, configure as follows:
    sasl_mechanism => "SCRAM-SHA-512"
    sasl_jaas_config => "org.apache.kafka.common.security.scram.ScramLoginModule required
username='username' password='password';"

    # If the security protocol is SASL_SSL, configure as follows:
    security_protocol => "SASL_SSL"
    ssl_truststore_location => "C:\\Users\\Desktop\\logstash-8.8.1\\config\\client.jks"
    ssl_truststore_password => "dms@kafka"
    ssl_endpoint_identification_algorithm => ""

   # If the security protocol is SASL_PLAINTEXT, configure as follows:
    security_protocol => "SASL_PLAINTEXT"
    }
}
output {
 stdout{codec=>rubydebug}
}
```

Description:

- **bootstrap_servers**: private network connection address or public network connection address of the Kafka instance.
- **group_id**: consumer group name.
- **topics**: topic name.
- **auto_offset_reset**: consumers' consumption policy. This example uses **earliest**.
- **sasl_mechanism**: SASL authentication mechanism.
- **sasl_jaas_config**: SASL JAAS configuration file. Change the SASL username and password as required.
- **security_protocol**: security protocol used by the Kafka instance.

- **ssl.truststore.location**: location where the SSL certificate is stored.
- **ssl_truststore_password**: server certificate password, which must be set to **dms@kafka** and cannot be changed.
- **ssl_endpoint_identification_algorithm**: Indicates whether to verify the certificate domain name. If this option is left blank, the certificate domain name is not verified. **In this example, leave it blank**.

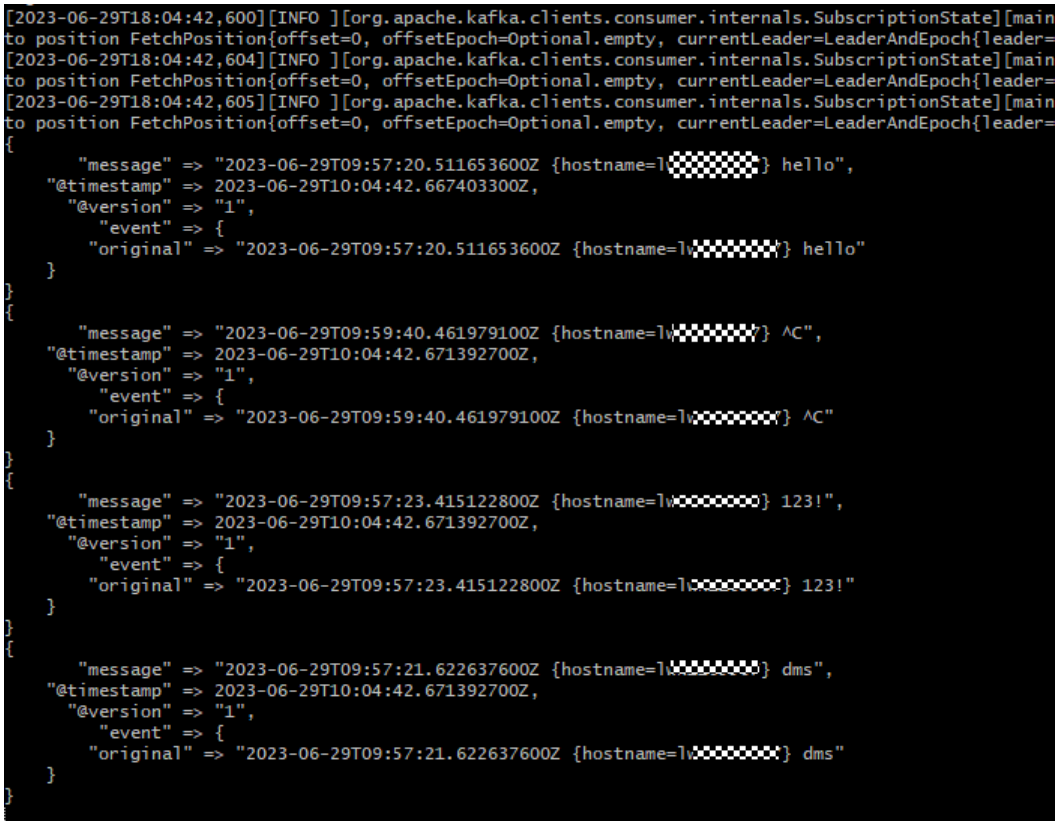
For more information about Kafka input plugin options, see **Kafka input plugin**.

Step 3 Open Git Bash in the **root** directory of the Logstash folder and run the following command to start Logstash:

```
./bin/logstash -f ./config/input.conf
```

After Logstash is started successfully, the Kafka input plugin automatically reads data from **topic-logstash** of the Kafka instance, as shown in the following figure.

Figure 8-8 Logstash reading data from topic-logstash

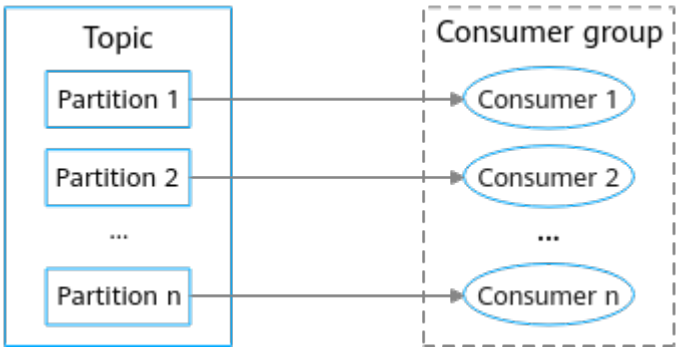


----End

9 Avoiding Message Accumulation

Introduction

Kafka divides each topic into multiple partitions for distributed message storage. Within the same consumer group, each consumer can consume multiple partitions at the same time, but each partition can be consumed by only one consumer at a time.



Unprocessed messages accumulate if the client's consumption is slower than the server's sending. Accumulated messages cannot be consumed in time.

Causes of accumulation

The following are some main causes:

- Producers produce messages too fast for consumers to keep up.
- Incapable consumers (low concurrency and long processing) cause lower efficiency of consumption than production.
- Abnormal consumers (faulty and network error) cannot consume messages.
- Improper topic partitions, or no consumption in new partitions.
- Frequent topic rebalancing reduces consumption efficiency.

Solution

Accumulation can be avoided by the consumer, producer, and server.

- **Consumer**

  - Add consumers (consumption concurrency) based on actual needs. The number of consumers should be accorded with the number of partitions.
  - Speed up consumption by optimizing the consumer processing logic (less complicated computing, API invoking, and database reading).
  - Increase the number of messages in each poll: Polling/Processing speed should be equal to or higher than the production speed.

- **Producer**

  Attach a random suffix to each message key so that messages can be evenly distributed in partitions.

  NOTE

  In actual scenarios, attaching a random suffix to each message key compromises global message sequence. Decide whether a suffix is required by your service.

- **Server**
  - Set the number of topic partitions properly. Add partitions without affecting processing efficiency.
  - Stop production when messages are accumulating or forward them to other topics.

10 Handling Service Overload

Introduction

High CPU usage and full disks indicate overloaded Kafka services.

- High CPU usage leads to low system performance and high risk of hardware damage.
- If a disk is full, the Kafka log content stored on it goes offline. Then, the disk's partition replicas cannot be read or written, reducing partition availability and fault tolerance. The leader partition switches to another broker, adding load to the broker.

Causes of high CPU usage

- There are too many data operation threads: **num.io.threads**, **num.network.threads**, and **num.replica.fetchers**.
- Improper partitions. One broker carries all production and consumption services.

Causes of full disk

- Current disk space no longer meets the needs of the rapidly increasing service data volume.
- Unbalanced broker disk usage. The produced messages are all in one partition, taking up the partition's disk.
- The time to live (TTL) set for a topic is too long. Old data takes too much disk space.

Solution

Handling high CPU usage:

- Optimize the parameters configuration for threads **num.io.threads**, **num.network.threads**, and **num.replica.fetchers**.
  - Set the number of **num.io.threads** and the number of **num.network.threads** threads to multiples of the disk quantity. Do not exceed the number of CPU cores
  - Set the number of **num.replica.fetchers** threads to smaller than or equal to 5.

- Set topic partitions properly. Set the number of partitions to multiples of the number of brokers.
- Attach a random suffix to each message key so that messages can be evenly distributed in partitions.

  NOTE

  In actual scenarios, attaching a random suffix to each message key compromises global message sequence. Decide whether a suffix is required by your service.

Handling full disk:

- Increase the disk space.
- Migrate partitions from the full disk to other disks on the broker.
- Set a proper TTL for topics to decrease the of old data.
- If CPU resources are sufficient, compress the data with compression algorithms.

  Common compression algorithms include ZIP, gzip, Sappy, and LZ4. You need to consider the data compression rate and duration when selecting compression algorithms. Generally, an algorithm with a higher compression rate consumes more time. For systems with high performance requirements, select algorithms with quick compression, such as LZ4. For systems with high compression rate requirements, select algorithms with high compression rate, such as gzip.

  Configurethe **compression.type** parameter on producers to specify a compression algorithm.

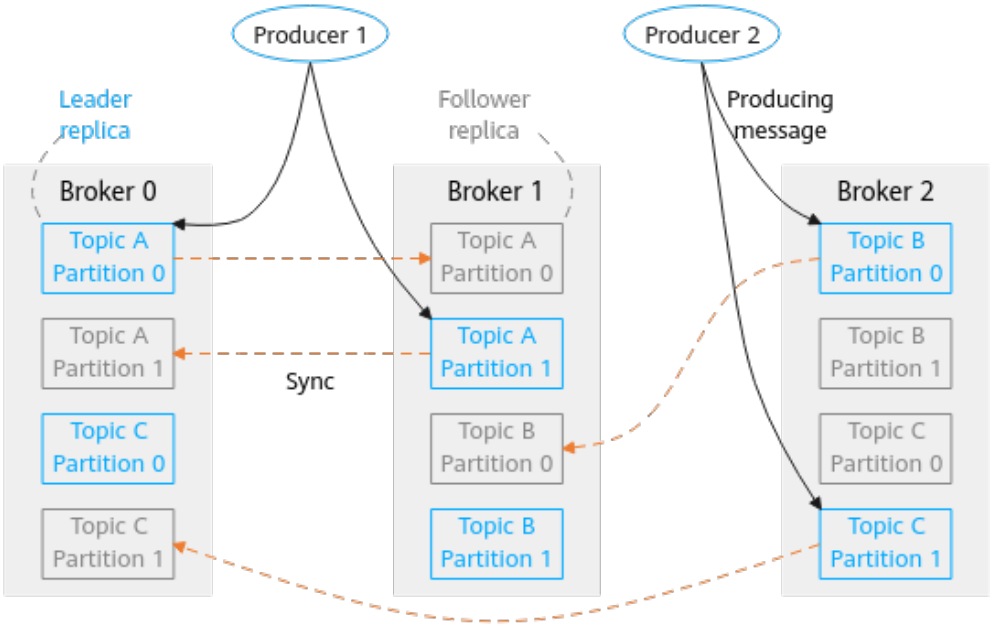
```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "all");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
// Enable GZIP.
props.put("compression.type", "gzip");

Producer<String, String> producer = new KafkaProducer<>(props);
```


11 Handling Uneven Service Data

Introduction

Kafka divides each topic into multiple partitions for distributed message storage. Each partition has one or more replicas distributed on different brokers. Each replica stores a copy of full data. Messages are synchronized among replicas. The following figure shows the relationships between topics, partitions, replicas, and brokers.



Uneven service data among brokers and partitions may happen, leading to low performance of Kafka clusters and low resource utilization.

Causes of uneven service data

- The traffic of some topics is much heavier than that of others.
- Producers specified partitions when sending messages, leaving unspecified partitions empty.
- Producers specified message keys to send messages to specific partitions.
- The system re-implements flawed partition allocation policies.

- There are new Kafka brokers with no partitions allocated.
- Cluster changes lead to switches and migration of leader replicas, causing data on some brokers to increase.

Solution

Handling uneven service data:

- Optimize the topic design. For a topic with considerable data, the data can be split across topics.
- Producers evenly send messages across partitions.
- When creating topics, distribute leader replicas across brokers.
- Kafka features partition reassignment. You can reassign replicas to different brokers to balance load among brokers. For details, see **Reassigning Partitions**.

