Distributed Message Service

Best Practices

Issue 01

Date 2019-01-04

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions

HUAWEI and other Huawei trademarks are trademarks 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 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

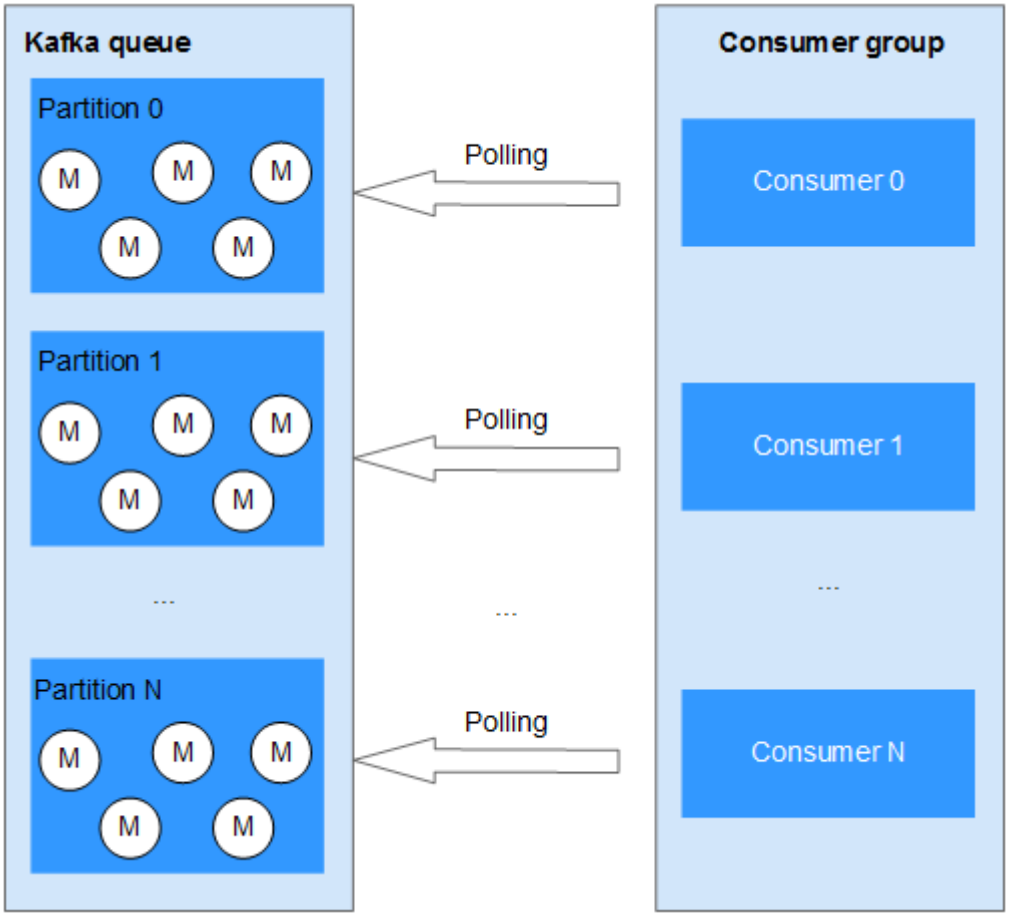
1 Optimizing Message Polling of DMS Kafka Consumers

Overview

In the native Kafka SDK provided by DMS, 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 Kafka queue 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 queue, the connection persists, and all consumers pull messages continuously, which causes a waste of resources.

Figure 1-1 Multi-thread consumption mode of Kafka consumers

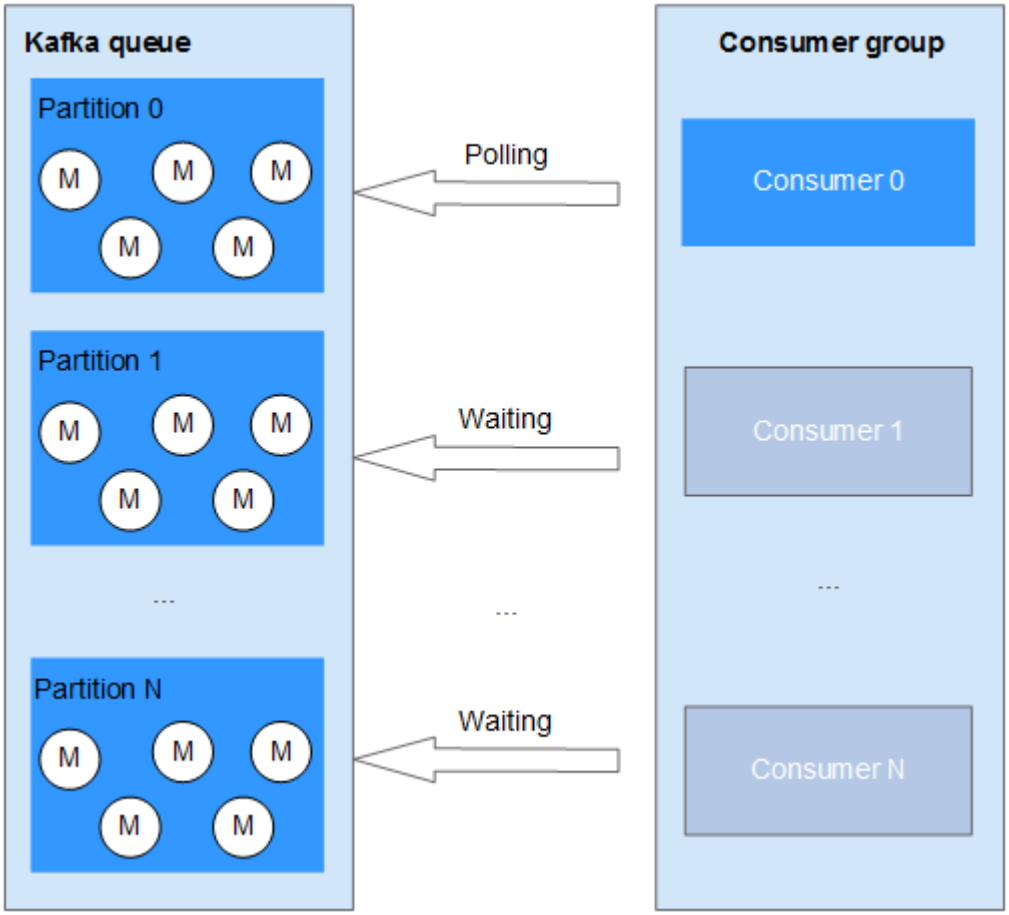


Optimization Solution

When multiple threads are accessed at the same time, if there is no message in the queue, 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 retrieval.
            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

Set **topicName** to a queue name or Kafka 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 Filtering Messages by Message Label

Overview

In an actual scenario, messages stored in a queue may contain different practical purposes. If the messages are not differentiated, consumers will pull messages in sequence until the consumption of all messages is complete.

If consumers are interested only in a certain type of messages, consumption of all messages will affect the processing efficiency.

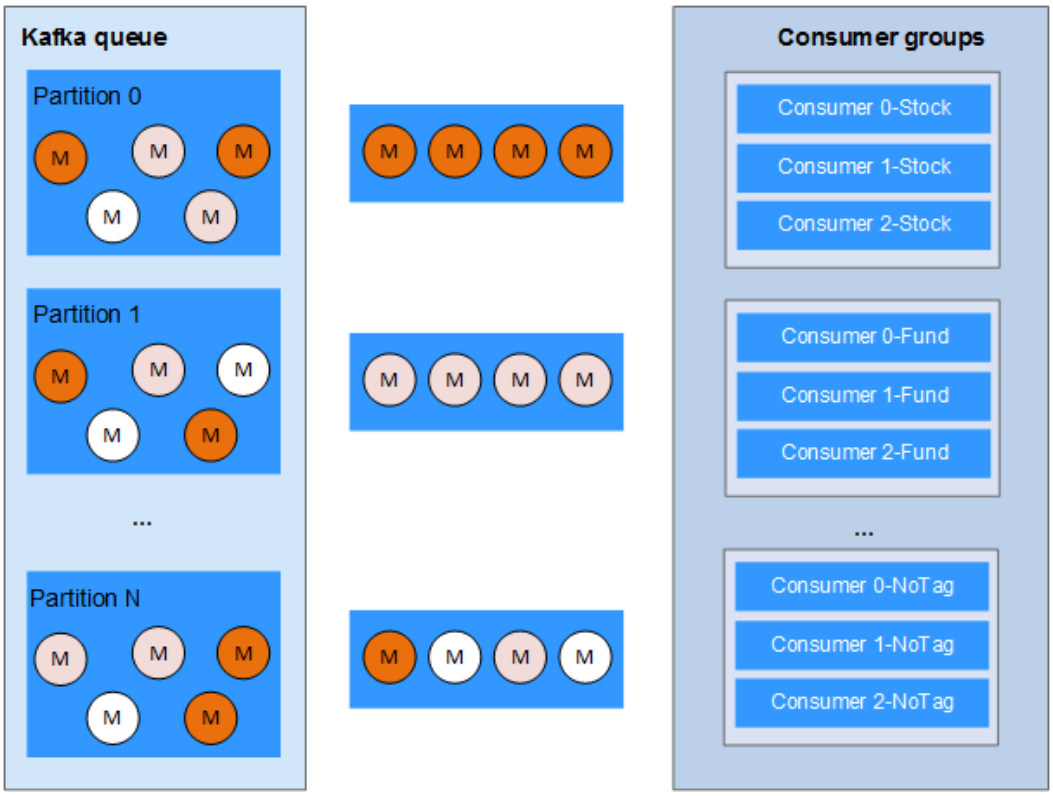
Optimization Solution

DMS provides the message label capability. A producer can provide one or more labels for each message. Messages are filtered based on the label content to ensure that consumers only consume the message type that they are interested in.

For example, in a financial scenario, multiple types of messages may be generated in a transaction, such as a stock, fund, and loan. These messages are transferred to different processing systems, such as the stock system, fund system, loan system, and real-time analysis system, through business topics. However, the fund system concerns only messages of the fund type, and the real-time analysis system may need to obtain all types of messages.

When producing a message, the producer adds a label to each message. When the consumer pulls messages, the consumer determines whether to obtain only messages with a specific label. This improves the message consumption efficiency, as shown in **Figure 2-1**.

Figure 2-1 Consumption of messages with labels



NOTICE

DMS standard queues and FIFO queues support the message label function. Kafka queues do not support this function.

Sample Code

NOTICE

The following describes only the codes related to the message label. To run the entire demo, download the complete **sample code package** and refer to the **DMS Development Guide** for deploying and running the code.

The example provides code based on the HTTP RESTful interface. For details about the APIs, see the **Distributed Message Service API Reference** in Help Center.

Sample code for message label design:

```
package com.cloud.dms;

import java.net.URL;
import java.util.Properties;
import com.cloud.dms.access.AccessServiceUtils;

public class DMSHttpClient
{
    private static String endpointUrl = "";
```

```
    private static String region = "";

    private static String serviceName = "dms";

    private static String aKey = "";

    private static String sKey = "";

    private static String projectId = "546e52331ea74cd49722fda4fb23bf55";

    private static String queueId = "39cd8dcb-b901-43b4-9ea1-48730e9adc58";

    private static String queueGroupId = "g-ae8ed05f-464c-452c-9e37-d3bdd081000d";

    /*
     * Read Configure File And Initialize Variables
     */
    static
    {
        URL configPath = ClassLoader.getSystemResource("dms-service-config.properties");
        Properties prop = AccessServiceUtils.getPropsFromFile(configPath.getFile());
        region = prop.getProperty(Constants.DMS_SERVICE_REGION);
        aKey = prop.getProperty(Constants.DMS_SERVICE_AK);
        sKey = prop.getProperty(Constants.DMS_SERVICE_SK);
        endpointUrl = prop.getProperty(Constants.DMS_SERVICE_ENDPOINT_URL);
        if (endpointUrl.endsWith("/"))
        {
            endpointUrl = endpointUrl + "v1.0/";
        }
        else
        {
            endpointUrl = endpointUrl + "/v1.0/";
        }
        projectId = prop.getProperty(Constants.DMS_SERVICE_PROJECT_ID);
    }

    public static void main(String[] args)
    {
        runAllApiMethods();
    }

    public static void runAllApiMethods()
    {
        MsgAttri msg = new MsgAttri();
        msg.setaKey(aKey);
        msg.setEndpointUrl(endpointUrl);
        msg.setProjectId(projectId);
        msg.setQueueId(queueId);
        msg.setsKey(sKey);
        msg.setRegion(region);
        msg.setServiceName(serviceName);
        msg.setMsgLimit("10");
        msg.setGroupId(queueGroupId);
        /**
         *          * Construct producers and four types of consumers and set labels that they are interested in.
         */
        MsgProducer msgProducer = new MsgProducer(msg);
        MsgConsumer stock = new MsgConsumer(msg, "stock");
        MsgConsumer fund = new MsgConsumer(msg, "fund");
        MsgConsumer loan = new MsgConsumer(msg, "loan");
        MsgConsumer all = new MsgConsumer(msg, null);
        /**
         * Create threads, simulate production and consumption behaviors, and set thread names for
differentiation.
         */
        Thread producer = new Thread(msgProducer);
        Thread stockThread = new Thread(stock);
        Thread fundThread = new Thread(fund);
```

```
        Thread loanThread = new Thread(loan);
        Thread alls = new Thread(all);
        producer.setName("producer");
        stockThread.setName("stock");
        fundThread.setName("fund");
        loanThread.setName("loan");
        alls.setName("Analysis");
        /**
         * Start threads.
         */
        producer.start();
        stockThread.start();
        fundThread.start();
//        loanThread.start();
//        alls.start();
    }
}
```

Sample code for producing messages:

```
package com.cloud.dms;

import static com.cloud.dms.ApiUtils.constructTempMessages;
import static com.cloud.dms.ApiUtils.sendMessages;

import java.util.concurrent.TimeUnit;

public class MsgProducer implements Runnable{
    private MsgAttri msgAttri;

    public MsgProducer(MsgAttri msg) {
        this.msgAttri = msg;
    }

    public void run() {
        while (true)
        {
            /**
            * Simulate producers to construct messages. JSON contains the stock, fund, and loan messages.
            **/
            String messages = constructTempMessages(null);
            sendMessages(messages, this.msgAttri);
            try
            {
                TimeUnit.SECONDS.sleep(1);
            }
            catch (InterruptedException e) {
            }
        }
    }
}
```

Sample code for consuming messages:

```
package com.cloud.dms;

import static com.cloud.dms.ApiUtils.acknowledgeMessages;
import static com.cloud.dms.ApiUtils.consumeMessages;
import static com.cloud.dms.ApiUtils.parseHandlerIds;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class MsgConsumer implements Runnable{

    private MsgAttri msgAttri;
```

```
    private String tag;

    /**
    *Construct a function to obtain the label that consumers are interested in.
    **/
    public MsgConsumer(MsgAttri msg, String tag) {
        this.tag = tag;
        this.msgAttri = msg;
    }

    public void run() {
        while (true)
        {
            /**
            *Consume messages and obtain messages with the label.
            **/
            ResponseMessage consumeMessagesResMsg = consumeMessages(msgAttri, tag);
            /**
            *Parse messages.
            **/
            if (consumeMessagesResMsg.getStatusCode() == 200)
            {
                List<String> msgStrings = ApiUtils.decodeMsg(consumeMessagesResMsg);
                /**
                *Simulate message processing and print messages with the label.
                **/
                for (String s : msgStrings)
                {
                    System.out.println("Thread--"+ Thread.currentThread().getName() + "--Message Body is: "+ s);
                }
                /**
                * Confirm message consumption.
                **/
                ArrayList<String> handlerIds = parseHandlerIds(consumeMessagesResMsg);
                if (handlerIds.size() > 0)
                {
                    acknowledgeMessages(handlerIds, msgAttri);
                }
            }
            else
            {
                System.out.println("Http Response Code is: "
                        + consumeMessagesResMsg.getStatusCode() + "\n Http Body is: "
                        + consumeMessagesResMsg.getBody());

            }
            try
            {
                TimeUnit.SECONDS.sleep(2);
            }
            catch (InterruptedException e)
            {
            }
        }
    }
}
```

Running Results of Sample Code

The loan thread has specified the tag (loan) and therefore can consume only messages with the loan label.

```
Thread--loan--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607269605","tags":["loan"]}
Thread--loan--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607270813","tags":["loan"]}
Thread--loan--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607272069","tags":["loan"]}
Thread--loan--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607277819","tags":["loan"]}
Thread--loan--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607280210","tags":["loan"]}
Thread--loan--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607281437","tags":["loan"]}
Thread--loan--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607282671","tags":["loan"]}
Thread--loan--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607311902","tags":["loan"]}
Thread--loan--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607314401","tags":["loan"]}
Thread--loan--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607315635","tags":["loan"]}
[]

{"success":10,"fail":0}
```

The fund and stock threads can consume only the specified messages.

```
Thread--stock--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607341145","tags":["stock"]}
Thread--stock--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607342397","tags":["stock"]}
Thread--stock--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607343916","tags":["stock"]}
Thread--stock--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607345143","tags":["stock"]}
Thread--stock--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607346370","tags":["stock"]}
Thread--stock--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607435714","tags":["stock"]}
Thread--stock--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607437984","tags":["stock"]}
Thread--stock--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607439235","tags":["stock"]}
Thread--stock--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607440469","tags":["stock"]}
Thread--stock--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607451488","tags":["stock"]}
{"success":10,"fail":0}
```

The analysis thread does not specify a label and can consume all messages in the topic.

```
Thread--Analysis--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607021008","tags":["loan"]}
Thread--Analysis--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607153683","tags":["stock"]}
Thread--Analysis--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607153683","tags":["fund"]}
Thread--Analysis--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607153683","tags":["loan"]}
Thread--Analysis--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607156342","tags":["stock"]}
Thread--Analysis--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607156342","tags":["fund"]}
Thread--Analysis--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607156342","tags":["loan"]}
Thread--Analysis--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607157801","tags":["stock"]}
Thread--Analysis--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607157801","tags":["fund"]}
Thread--Analysis--Message Body is: {"attributes":{"name":"ignore","type":"string.start","value":"test"},"body":"msg_body_1513607157801","tags":["loan"]}
{"success":10,"fail":0}
```

3 Methods for Improving the Message Processing Efficiency

During message sending and consuming, Huawei DMS, producers, and consumers collaboratively ensure the service reliability. In addition, developers must use DMS message queues properly to improve the efficiency and accuracy of message sending and message consumption.

The following lists the usage suggestions for DMS producers and consumers.

Paying Attention to the Confirmation Process of Message Production and Consumption

Message production (sending)

After messages are sent, the producer checks the message sending confirmation returned from DMS. If messages fail to be sent, the producer re-sends the messages.

After messages are produced, the producer waits for the acknowledgement (ACK) returned by the message sending API to determine whether messages are sent successfully. During message sending, if any abnormality occurs and the producer fails to receive the ACK, the producer determines whether to re-send messages. If the producer receives the ACK indicating that the message has been successfully sent, the message is reliably stored by DMS.

Message consumption

When a message is consumed, the consumer needs to check whether the message consumption is successful.

Produced messages are stored in the storage medium of DMS. During message consuming, the consumer obtains messages stored in DMS. Then the consumer consumes the messages, records message consumption status (successful or failed), and submits the consumption status to DMS. Based on the consumption status, DMS determines whether to consume next batch of messages or re-consume the messages that failed to be consumed.

Throughout the entire message consuming process, if the message consumption status fails to be submitted due to some abnormalities, this batch of messages will

be re-obtained by the consumer in the subsequent message consumption requests.

Idempotent Transferring of Message Production and Consumption

DMS provides a series of reliability measures to ensure that messages are not lost. 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 solve the exceptions that occur during message transmission.

Considering the extreme conditions such as network exceptions, you need to cooperate with DMS to design message sending and consumption in addition to confirming message production and consumption.

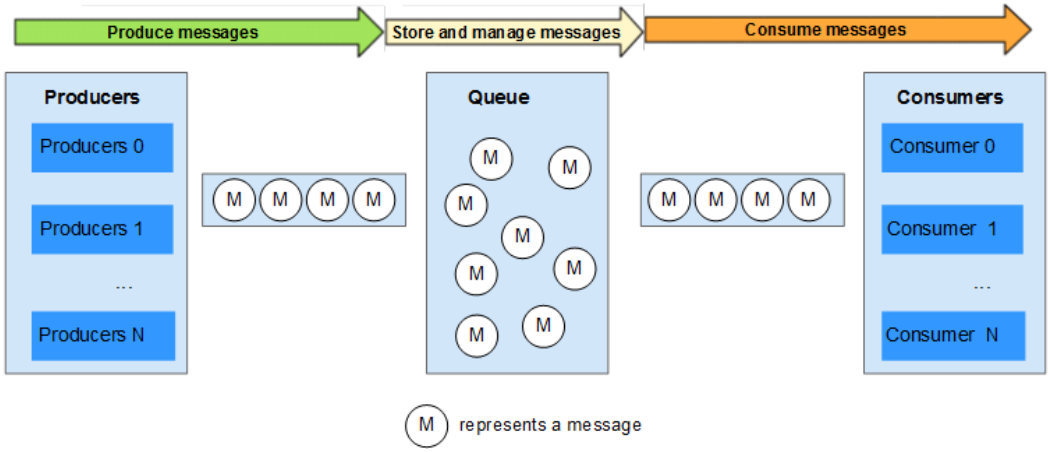
- If the message fails to be confirmed, the producer needs to send the message to DMS repeatedly.
- After receiving a message that has been processed, the consumer needs to notify that DMS consumption is successful and ensure that the message is not processed repeatedly.

Producing and Consuming Messages in Batches

To improve the message sending and consumption efficiency, consumers are advised to use the batch message sending and consumption mode, which can effectively lower the number of API calls and minimize service fees,

as shown in the following figures.

Figure 3-1 Message production (sending) and consumption in batches

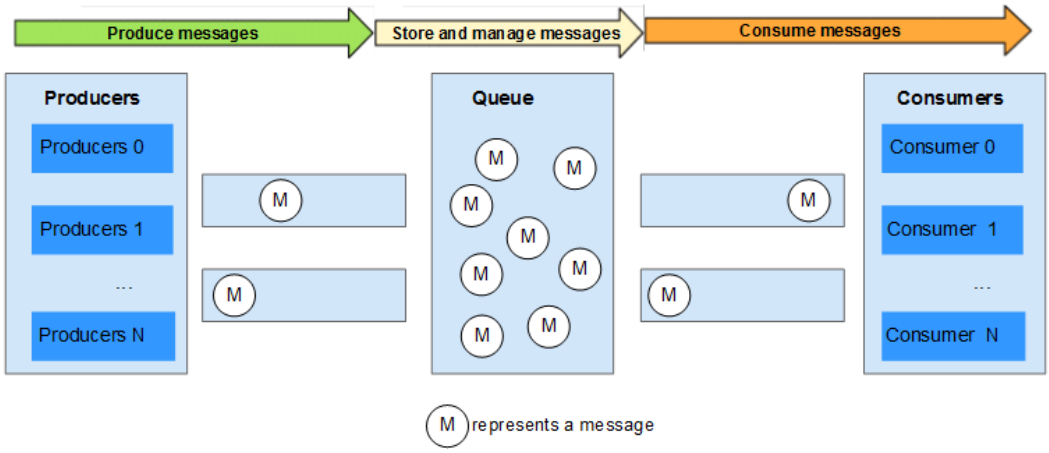


NOTICE

A maximum of 10 messages can be sent in batches. The total size cannot exceed 512 KB.

Batch production (sending) messages can be flexibly used. When there are a large number of concurrent messages, the messages are sent in batches. When the number of concurrent messages is small, the messages are sent one by one. In this way, the number of API calls is reduced and the real-time message sending is ensured.

Figure 3-2 Message production (sending) and consumption one by one



In addition, consumers need to process and confirm messages in the sequence of receiving messages in batches. Therefore, when a consumer fails to consume a message, you are advised to stop the consumer from consuming the rest messages, and directly submit consumption confirmations of the successfully consumed messages to DMS.

Using Consumer Groups to Assist O&M

You can use DMS as the message management system. Viewing the message content of a queue is crucial for locating problems and debugging services.

When problems occur during message production and consumption, you can create different consumer groups to locate and analyze problems or debug service interconnection. You can create a consumer group to consume messages in queues and analyze the consumption process. This does not affect the message processing of other services.

4 Setting Parameters for Kafka Clients

This section provides recommendations on configuring common parameters for Kafka producers and consumers.

Table 4-1 Producer parameters

Parameter	Default Value	Recommended Value	Description
acks	1	**all** (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. The value of this parameter can be any of the following: **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**: 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.

Parameter	Default Value	Recommended Value	Description
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.
request.timeout.ms	30000	Set as required.	Maximum amount of time 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, **120000** (120s), 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.

Parameter	Default Value	Recommended Value	Description
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 helps improve 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.
buffer.memory	33554432	67108864	Total bytes of memory the producer can use to buffer records waiting to be sent to the server. If records are sent faster than they can be delivered to the broker, the producer will stop sending records 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 4-2 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.
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.
connections.max.idle.ms	600000	30000	Timeout interval 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.

5 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. A consumer cannot frequently join or leave a group. Otherwise, the consumer will frequently perform rebalancing, which blocks consumption.
4. The number of consumers cannot be greater than the number of partitions in the topic. Otherwise, some consumers may fail to poll for messages.
5. 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.
6. Ensure that there is a limitation on the size of messages buffered locally to avoid an out-of-memory (OOM) situation.
7. Set the timeout for the consumer session to 30 seconds: session.timeout.ms=30000.
8. Kafka supports exactly-once delivery. Therefore, ensure the idempotency of processing messages for services.
9. Always close the consumer before exiting. Otherwise, consumers in the same group may be blocked within the timeout set by **session.timeout.ms**.

Producers

1. Synchronous replication: Set **acks** to **all**.
2. Retry message sending: Set **retries** to **3**.
3. Message sending optimization: Set **linger.ms** to **0**.
4. Ensure that the producer has sufficient JVM memory to avoid blockages.

Configuring 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 created in or retrieved from a topic that does not exist.

The recommended maximum number of partitions for a topic is 20.

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.

