Distributed Message Service for RocketMQ

Best Practices

Issue 01

Date 2024-04-17

HUAWEI

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 Migrating RocketMQ Services

Scenario

RocketMQ service migration involves the following scenarios:

- Migrating RocketMQ from other vendors to DMS for RocketMQ.
- Migrating self-built RocketMQ to DMS for RocketMQ.
- Migrating one RocketMQ instance to another RocketMQ instance in the cloud.

To migrate metadata, you can use either of the following methods as required:

- Method 1: Run the **mqadmin** command to export the source instance metadata and then create a migration task in DMS for RocketMQ.
- Method 2: Export the source topics and consumer groups and then import them to DMS for RocketMQ using scripts. (Use this method when metadata cannot be exported using the **mqadmin** command.)

Prerequisites

1. Configure the network environment.

   A RocketMQ 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 1-1 Security group rules

Direction	Protocol	Port	Source	Description
Inbound	TCP	8200	0.0.0.0/0	The port is used for public access to metadata nodes.
Inbound	TCP	10100-10199	0.0.0.0/0	The port is used for accessing service nodes.

2. Buy a RocketMQ instance.

   For details, see **Buying a RocketMQ Instance**.

3. A Linux host is available, **JDK v1.8.111 or later** has been installed on the host, and related environment variables have been configured.

Procedure (Using the mqadmin Command to Export the Metadata of the Source Instance)

Step 1 Migrate metadata to the RocketMQ instance.

1. Obtain the RocketMQ metadata from another cloud or self-hosted RocketMQ.
   a. Log in to the host and download the RocketMQ software package.
   ```
   wget https://archive.apache.org/dist/rocketmq/4.9.8/rocketmq-all-4.9.8-bin-release.zip
   ```
   b. Decompress the software package.
   ```
   unzip rocketmq-all-4.9.8-bin-release.zip
   ```
   c. (Optional) If ACL is enabled for the RocketMQ instance, authentication is required when you run the **mqadmin** command.

   Switch to the directory where the decompressed software package is stored and add the following content to the **conf/tools.yml** file:
   ```
   accessKey:*******
   secretKey:*******
   ```
   accessKey and **secretKey** are the username and secret key set on the **Users** page of the console.
   d. Go to the directory where the decompressed software package is stored and run the following command to query the cluster name:
   ```
   sh ./bin/mqadmin clusterList -n {nameserver address and port number}
   ```
   For example, if the nameserver address and port number are **192.168.0.65:8100**, run the following command:
   ```
   sh ./bin/mqadmin clusterList -n 192.168.0.65:8100
   ```
   e. Run the following command to export metadata:
   - If SSL is disabled, run the following command:
   ```
   sh ./bin/mqadmin exportMetadata -n {nameserver address and port number} -c {RocketMQ cluster name} -f {Path for storing the exported metadata file}
   ```
   For example, if the nameserver address and port number are **192.168.0.65:8100**, the RocketMQ cluster name is **DmsCluster**, and the path for storing exported metadata files is **/tmp/rocketmq/export**, run the following command:
   ```
   sh ./bin/mqadmin exportMetadata -n 192.168.0.65:8100 -c DmsCluster -f /tmp/rocketmq/export
   ```
   - If SSL is enabled, run the following command:
   ```
   JAVA_OPT=-Dtls.enable=true sh ./bin/mqadmin exportMetadata -n {nameserver address and port number} -c {RocketMQ cluster name} -f {path for storing the exported metadata file}
   ```
   For example, if the nameserver address and port number are **192.168.0.65:8100**, the RocketMQ cluster name is **DmsCluster**, and the path for storing exported metadata files is **/tmp/rocketmq/export**, run the following command:
   ```
   JAVA_OPT=-Dtls.enable=true sh ./bin/mqadmin exportMetadata -n 192.168.0.65:8100 -c DmsCluster -f /tmp/rocketmq/export
   ```
2. Migrate metadata on the console.
   a. Log in to the **DMS for RocketMQ console**.
   b. Click a RocketMQ instance to go to the instance details page.

c. In the navigation pane, choose **Metadata Migration**.
d. Click **Create Migration Task**.
e. Configure the migration task by referring to **Table 1-2**.

Table 1-2 Migration task parameters

Parameter	Description
Task Name	Unique name of the migration task.
Overwrite	▪ If this option is enabled, configurations in the metadata file with the same name as the uploaded file will be modified. Assume that Topic01 on the source instance has three read queues, and Topic01 on the DMS instance has two read queues. If **Overwrite** is enabled, Topic01 on the DMS instance will have three read queues after migration. ▪ If this option is disabled, migration of the metadata file with the same name as the uploaded file will fail. Assume that the source instance has Topic01 and Topic02, and the DMS instance has Topic01 and Topic03. If **Overwrite** is disabled, migration of the source Topic01 will fail.
Metadata	Upload the **RocketMQ metadata obtained from another cloud or self-hosted RocketMQ**.

f. Click **OK**.

After the migration is complete, view **Task Status** in the migration task list.

- If **Task Status** is **Complete**, all metadata has been successfully migrated.
- If **Task Status** is **Failed**, some or all metadata fails to be migrated. Click the migration task name to go to the migration task details page. In the **Migration Result** area, view the name of the topic or consumer group that fails to be migrated and the failure cause.

Step 2 Migrate the production service to the RocketMQ instance.

Change the metadata connection address on the production client to the metadata connection address of the RocketMQ instance and then restart the production service. New messages will be sent to the RocketMQ instance.

Step 3 Migrate the consumption service to the RocketMQ instance.

After all messages in the consumer group are consumed, change the metadata connection address of the consumer client to the metadata connection address of the RocketMQ instance. New messages will be consumed from the RocketMQ instance.

Step 4 If there are multiple source RocketMQ instances, migrate services from them one by one.

----End

Procedure (Metadata Cannot Be Exported Using the mqadmin Command)

Step 1 Log in to the console of another vendor and export the lists of source topics and consumer groups.

Step 2 Create the **topics.txt** and **groups.txt** files and add the source topic list and consumer group list to the files respectively. Each line contains a topic or consumer group name. For example:

```
topic-01
topic-02
...
topic-n
```

Note: The **groups.txt** file cannot contain blank lines (for example, a newline character at the end of a consumer group name). Otherwise, consumer groups with empty names will be created when the lists are imported to the RocketMQ instance.

Step 3 Log in to the host and download the RocketMQ software package.

```
wget https://archive.apache.org/dist/rocketmq/4.9.8/rocketmq-all-4.9.8-bin-release.zip
```

Step 4 Decompress the software package.

```
unzip rocketmq-all-4.9.8-bin-release.zip
```

Step 5 (Optional) If ACL is enabled for the RocketMQ instance, authentication is required when you run the **mqadmin** command.

Switch to the directory where the decompressed software package is stored and add the following content to the **conf/tools.yml** file:

```
accessKey:*******
secretKey:*******
```

accessKey and **secretKey** are the username and secret key set on the **Users** page of the console.

Step 6 Go to the **bin** directory of the decompressed software package and upload **topics.txt** and **groups.txt** to this directory.

Step 7 Run the following script to import the source topics and consumer groups to DMS for RocketMQ:

```
#!/bin/bash

# Read groups from groups.txt file
groups=()
while read -r group; do
  groups+=("$group")
done < "groups.txt"

# Read topics from topic.txt file
topics=()
while read -r topic; do
  topics+=("$topic")
done < "topics.txt"

# Add topics
for topic in "${topics[@]}"; do
  echo "Adding topic: $topic"
```



```
  sh mqadmin updateTopic -n <namesrvIp:8100> -c DmsCluster -t "$topic"
done

# Add consumer groups
for group in "${groups[@]}"; do
  echo "Adding consumer group: $group"
  sh mqadmin updateSubGroup -n <namesrvIp:8100> -c DmsCluster -g "$group"
done
```

namesrvIp:8100 indicates the address of the RocketMQ instance.

Step 8 Log in to DMS for RocketMQ console. Go to the **Topics** and **Consumer Groups** pages and check whether the topics and consumer groups are successfully imported.

Step 9 Migrate the production service to the RocketMQ instance.

Change the metadata connection address on the production client to the metadata connection address of the RocketMQ instance and then restart the production service. New messages will be sent to the RocketMQ instance.

Step 10 Migrate the consumption service to the RocketMQ instance.

After all messages in the consumer group are consumed, change the metadata connection address of the consumer client to the metadata connection address of the RocketMQ instance. New messages will be consumed from the RocketMQ instance.

Step 11 If there are multiple source RocketMQ instances, migrate services from them one by one.

----End

2 Deduplicating Messages Through Message Idempotence

Overview

In RocketMQ service processes, an idempotent message process refers to a situation where a message is re-sent and consumed for multiple times and each consumption result is the same, having no negative effects on services. Idempotent messages ensure consistency in the final processing results. Services are not affected no matter how many times a message is re-sent.

Message Repetition Scenarios

In actual applications, messages are re-sent because of intermittent network disconnections and client faults during message production or consumption. Message repetition can be classified into two scenarios.

- A producer repeatedly sends a message:

  If a producer successfully sends a message to the server but does not receive a successful response due to an intermittent network disconnection, the producer determines that the message failed to be sent and tries resending the message. In this case, the server receives two messages of the same content. Consumers consume two messages of different IDs but the same content.

- A consumer repeatedly consumes a message:

  A message is successfully delivered to a consumer and processed. If the consumer fails to commit an updated offset to the server due to an intermittent network disconnection, the server determines that the message failed to be delivered. To ensure that the message is consumed at least once, the server retries delivering the message. As a result, the consumer receives the same message (ID and content) as the previously processed one.

Take payment as an example. Assume that a customer makes payment and receives multiple bills due to unstable Internet connection. However, the billing should take place only once and the merchant should generate only one order placement.

Procedure

Messages with different IDs may have the same content, so the ID cannot be used as the unique identifier. RocketMQ supports idempotent messages by using the message key (unique service identifier) to identify messages. The sample code for configuring a message key is as follows:

```
Message message = new Message();
message.setKey("Order_id");    // Set the message key, which can be the unique service identifier such as the order placement ID.
SentResult sendResult = mqProducer.send(message);
```

When a producer sends a message, the message has a unique key. When consuming the message, a consumer reads the unique message identifier (such as the order placement ID) with **getKeys()**. The service logic can implement idempotence with the unique identifier.

3 Classifying Messages with Topic and Tag

Overview

Topics are the basic logical unit of messages in message production and consumption. Each topic contains several messages and each message belongs to only one topic.

Tags are used to identify message of different types. Messages for different purposes in the same business unit can have different tags in the same topic. Tags ensure the clarity and coherency of code and facilitate query in RocketMQ. Consumers can implement different consumption logic for different topics based on tags to achieve better scalability.

Messages are first classified into topics and then with tags as shown in the following figure.

Scenario

Use topics and tags properly to ensure clear and efficient service structure. You can decide how to use topics and tags based on your needs.

- **Message type:** RocketMQ messages include normal, ordered, scheduled/delayed, and transactional messages. Different types of messages should be classified with topics, not tags.
- **Message priority:** Messages of a high priority should be in topics different from those with a low priority.
- **Service relationship:** Messages from unrelated services should be classified in topics. Messages from closely related services should be sent to the same topic, and classified with tags based on subtypes or sequence.

Procedure

Take logistics transportation as an example. Order messages of fresh goods and other goods are of different types, so they can be classified by two topics:

Topic_Common and **Topic_Fresh**. For each message type, you can use different tags to identify order destination provinces.

- Topic: Topic_Common
  - Tag = Province_A
  - Tag = Province_B
- Topic: Topic_Fresh
  - Tag = Province_A
  - Tag = Province_B

The following is message production sample code for a common goods order sent to province A:

```
Message msg = new Message("Topic_Common", "Province_A" /* Tag */, ("Order_id " +
i).getBytes(RemotingHelper.DEFAULT_CHARSET));
```

The following is subscription sample code for a fresh goods order sent to province A and province B:

```
consumer.subscribe("Topic_Fresh", "Province_A || Province_B");
```

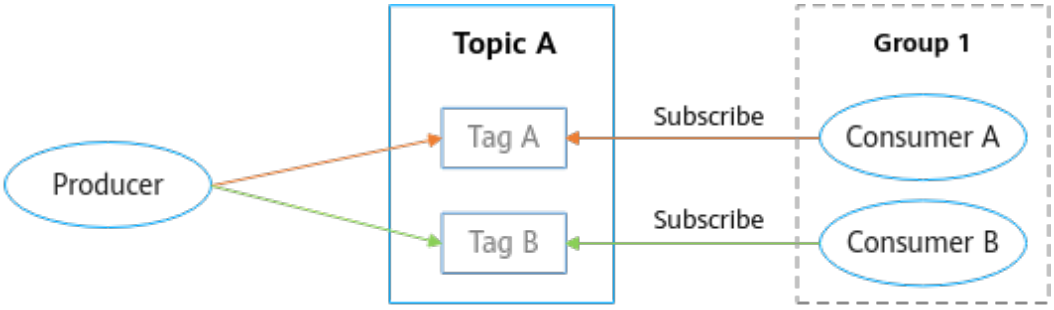
Different Consumers Consume Different Tags

Different consumers may consume messages with different tags in the same topic. For different tags in the same topic, improper consumer group settings lead to chaotic consumption.

For example, there are Tags A and B in Topic A. Consumer A subscribes to Tag A. Consumer B subscribes to Tag B.

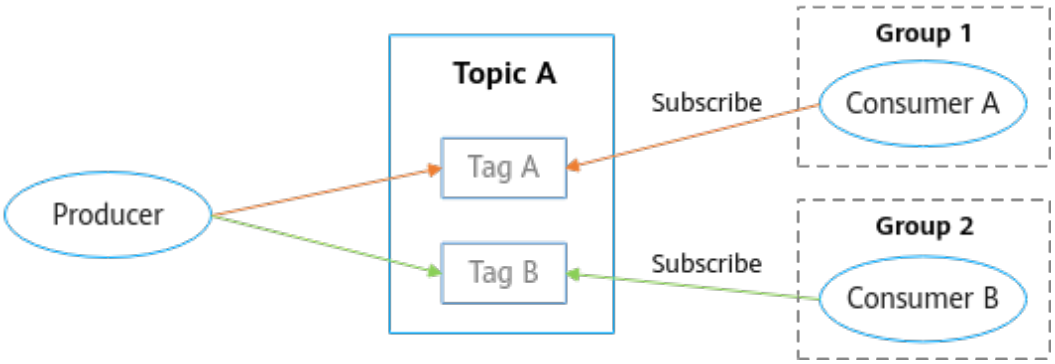
If Consumers A and B are in the same consumer group, messages with Tag A are evenly sent to Consumers A and B. Consumer B did not subscribe to Tag A, so it filters out messages with Tag A. As a result, some Tag A messages are not consumed.

Figure 3-1 Incorrect consumer group settings



To solve this problem, configure Consumers A and B with different consumer groups.

Figure 3-2 Correct consumer group settings



4 Ensuring Subscription Consistency

Overview

A consistent subscription indicates that all topics and tags subscribed by all consumers in the same consumer group are the same. An inconsistent subscription causes disordered consumption logic and even message losses.

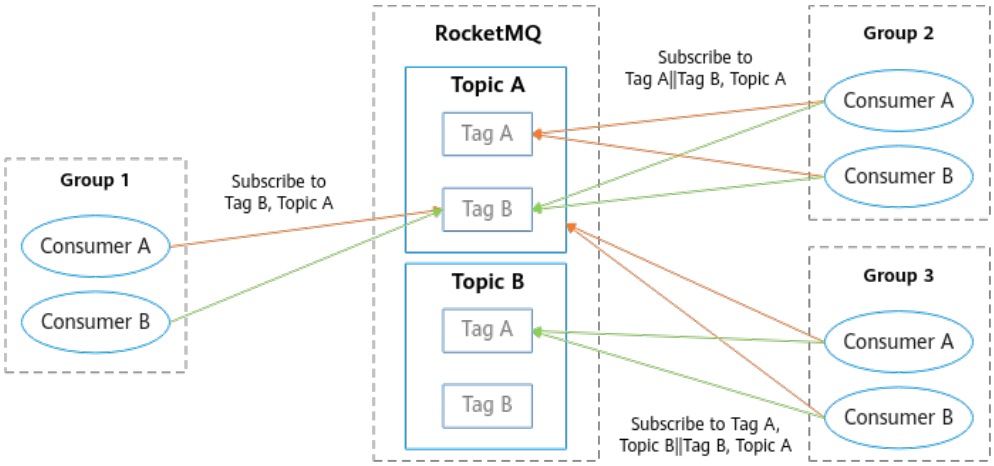
Principle

RocketMQ assigns message queues for each topic. The more queues, the higher the consumption concurrency. In distributed application scenarios, multiple consumers in the same consumer group jointly consume messages from all queues in a topic. Queues are assigned by consumer group and evenly assigned to the consumers in a consumer group, regardless of whether a consumer has subscribed to the topic. Each consumer is assigned some queues of a topic. Each queue is assigned to only one consumer.

Correct Subscription

In distributed application scenarios, all the consumers in a consumer group have the same consumer group ID. They must subscribe to the same topic and tag (consistent subscription) to ensure correct consumption logic and no message losses.

- Consumers in the same consumer group must subscribe to the same topic. For example, assume that Consumers A and B are in Consumer Group 1 and Consumer A subscribes to Topics A and B. Then, Consumer B must also subscribe to both Topics A and B, and cannot subscribe to only Topic A or B or even Topic C.
- The tags in the topic subscribed by consumers in the same consumer group must be the same, including the tag quantity and sequence. For example, assume that Consumers A and B are in Consumer Group 2. Consumer A subscribes to Tag1||Tag2 in Topic A. Then, when subscribing to Topic A, Consumer B must also subscribe to Tag1||Tag2, and cannot subscribe only to Tag 1 or 2 or Tag2||Tag1.



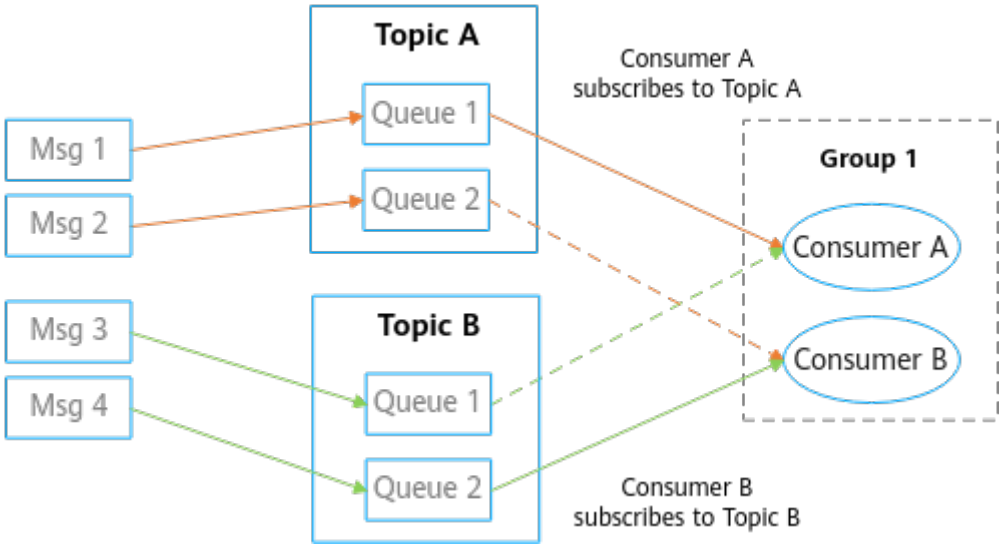
A consistent subscription ensures consumers in the same consumer group work properly, avoiding disordered message logic or message losses. Producers should classify messages properly for consumers to correct subscription to tags. Consumers should ensure consistent subscriptions.

Incorrect Subscription

- Consumers in the same consumer group subscribe to different topics.

  For example, assume that Consumers A and B are in Consumer Group 1. Consumer A subscribes to Topic A but Consumer B subscribes to Topic B. When producers send messages to Topic A, the messages are evenly sent to Consumers A and B by queue. Consumer B has not subscribed to Topic A, so it filters out messages from Topic A (Queue 2 in Topic A in the following figure), leaving them unconsumed.

  Figure 4-1 Incorrect topic subscriptions

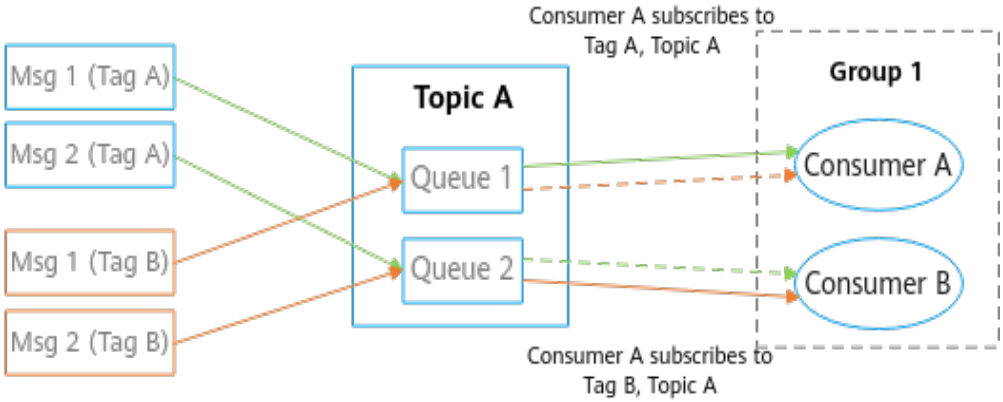


- Consumers in the same consumer group subscribe to different tags of the same topic.

  For example, assume that Consumers A and B are in Consumer Group 1. Consumer A subscribes to Tag A and Topic A. Consumer B subscribes to Tag B and Topic A. When producers send messages to Tag A in Topic A, messages with Tag A are evenly sent to Consumers A and B by queue. Consumer B has

not subscribed to Tag A, so it filters out messages with Tag A (Tag A in Queue 2 in the following figure), leaving them unconsumed.

Figure 4-2 Incorrect tag subscriptions



Procedure

- **Subscriptions to One Tag of One Topic**

  Consumers 1, 2, and 3 in Consumer Group 1 all subscribe to Tag_A and Topic_A. They have consistent subscriptions, meaning that their subscription code is the same. The sample code is as follows:

  ```
  DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("Group1");
  consumer.subscribe("Topic_A", "Tag_A");
  ```

- **Subscriptions to Multiple Tags of One Topic**

  Consumers 1, 2, and 3 in Consumer Group 1 all subscribe to **Tag_A** and **Tag_B** of **Topic_A**. The sequence is **Tag_A||Tag_B**. The consumers have consistent subscriptions, meaning that their subscription code is the same. The sample code is as follows:

  ```
  DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("Group1");
  consumer.subscribe("Topic_A", "Tag_A||Tag_B");
  ```

- **Subscriptions to Multiple Tags of Multiple Topics**

  Consumers 1, 2, and 3 in Consumer **Group 1** all subscribe to **Topic_A** (no specified tag) and **Topic_B** (**Tag_A** and **Tag_B**). The sequence is **Tag_A||Tag_B**. The consumers have consistent subscriptions, meaning that their subscription code is the same. The sample code is as follows:

  ```
  DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("Group1");
  consumer.subscribe("Topic_A", "*");
  consumer.subscribe("Topic_B", "Tag_A||Tag_B");
  ```

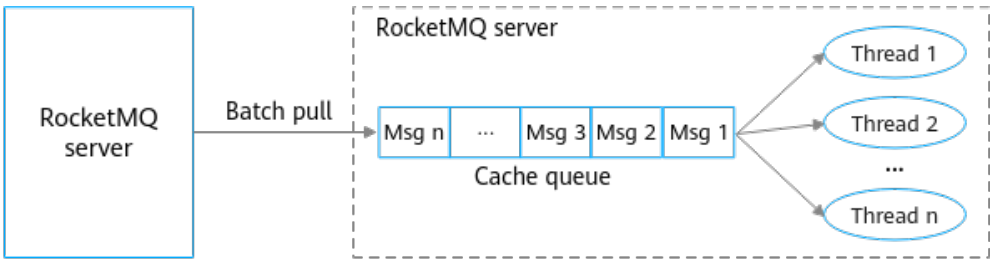

5 Avoiding Message Accumulation

Overview

Message accumulation is common in RocketMQ services. Unprocessed messages accumulate if the client's consumption is slower than the server's sending. Accumulated messages cannot be consumed in time. Service systems with high requirements on real-time consumption cannot afford even a short message delay caused by message accumulation. Message accumulation causes are as follows:

- Messages are not consumed in time because message production is faster than consumption. Messages accumulate and consumption cannot be restored automatically.
- The service system logic is time-consuming, causing low consumption efficiency.

Message Consumption Process



The message consumption process consists of two phases:

- Message pull

  Clients pull messages from servers in batches and store the messages to local cache queues. In this phase, no messages accumulate because throughput is high on the intranet.

- Message consumption

  Clients submit the cached messages to consumption threads, wait for the service consumption logic to process the messages, and receive the processing result. The consumption capability in this phase depends on the consumption duration and concurrency. The overall message throughput is affected if the service logic is complicated and spends a long time on a single message. Low message throughput causes local cache queues on the client to reach the

upper limit. Messages are no longer pulled from the server, resulting in accumulation.

Therefore, whether messages accumulate depends on the consumption capability of the client, and the consumption capability depends on the consumption duration and concurrency. Consumption time is prior to its concurrency. Users should ensure timely consumption before considering its concurrency.

Consumption Duration

Consumption duration is mainly affected by the service code, specially, the internal CPU computational code and the external I/O operational code. If there is no complex recursion or loop code, internal CPU computing duration can be ignored. Instead, you should focus on external I/O operations.

External I/O operations are as follows:

- Read/Write operations on external databases such as remote MySQL databases.
- Read/Write operations on external caches such as remote Redis.
- Invocations of downstream systems. For example, Dubbo invokes remote RPC and Spring Cloud invokes downstream HTTP APIs.

Learning about the downstream invoking logic helps you understand the duration of each invocation to determine whether the I/O operation duration in the service logic is proper. In general, faulty services or limited capacity in downstream systems causes longer consumption duration. Service faults can arise from network bandwidth issues as well as system errors.

Consumption Concurrency

The consumption concurrency on the client depends on number of clients (or consumers in a consumer group) and number of threads per client. The consumption concurrency of normal, scheduled/delayed, transactional, and ordered messages is calculated as follows.

Message Type	Concurrency Formula
Normal	Number of threads per client × Number of clients
Scheduled/Delayed	
Transactional	
Ordered	Min (Number of threads per client × Number of clients, Number of queues)

Note: The number of threads per client should be adjusted carefully. A large number of threads increases thread switch overhead.

An ideal calculation model for optimal number of threads per client: C × (T1+T2)/T1.

C indicates the number of vCPUs per broker. T1 indicates the internal CPU computation duration. T2 indicates the external I/O operation duration. Thread

switch overhead is ignored. I/O operations consume no CPU resources. A thread should have sufficient messages and memory for processing.

The model of calculating the maximum number of threads is only an ideal scenario. In actual scenarios, gradually increase threads based on the actual effect.

Procedure

To avoid unexpected message accumulation, the consumption duration should be accounted for and concurrency should be set properly in the design of service logic.

- **Accounting for consumption duration**

  Perform pressure test to obtain the consumption duration. Analyze and optimize time-consuming service logic code. Pay attention to:

  - Whether the computation of the consumption logic is too complex, and whether any complex recursions or loops exist in the code.
  - Whether I/O operations are necessary in the consumption logic and whether local caches can be used instead.
  - Whether the complicated, time-consuming operations in the consumption logic can be asynchronously processed.

- **Setting consumption concurrency**

  Consumption concurrency calculation can be adjusted with the following methods:

  a. Increase threads per client gradually to find an optimal number of consumption threads and message throughput per client.
  b. Calculate the number of clients needed based on the upstream and downstream traffic peaks: Number of clients = Traffic peak/Message throughput per client.

