Distributed Message Service for RabbitMQ

Best Practices

Issue 01

Date 2024-12-03

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Contents

1 RabbitMQ Best Practices

This section summarizes best practices of Distributed Message Service (DMS) for RabbitMQ in common scenarios. Each practice is given a description and procedure.

Table 1-1 RabbitMQ best practices

Best Practice	Description
Automatic Recovery of a RabbitMQ Client from Network Exceptions	This document provides the client reconnection sample code. Clients can be automatically reconnected when network exceptions occur, reducing the impact of network faults on services.
Automatic Consumer Reconnection After a RabbitMQ Node Restart	This document provides the consumer reconnection sample code after a RabbitMQ broker restart. A channel can be created for consumers to continue consumption after the previous channel is closed.
Improving RabbitMQ Performance	This document describes how to achieve high RabbitMQ performance by setting the queue length, cluster load balancing, priority queues, and other parameters.
Configuring Queue Load Balancing	This document describes how to improve cluster utilization by configuring queue load balancing among brokers.
Deduplicating Messages Through Message Idempotence	This document describes the causes of duplicate messages production and the handling measures.

2 Automatic Recovery of a RabbitMQ Client from Network Exceptions

Overview

Messages cannot be produced or consumed on the client due to server restart or network jitter.

With a connection retry mechanism on the client, the network connection on the client can be automatically restored. Automatic network recovery is triggered in the following scenarios:

- An exception is thrown in a connection's I/O loop.
- Socket read times out.
- Server heartbeat is lost.

NOTE

- Java clients of version 4.0.0 or later support automatic network recovery by default.
- If an application uses the **Connection.Close** method to close a connection, automatic network recovery will not be enabled or triggered.

Sample Code for Connection Retry on a RabbitMQ Client During Network Exceptions

If the initial connection between the client and server fails, automatic recovery is not triggered. Edit the corresponding application code on the client and retry the connection to solve the problem.

The following example shows how to use a Java client to resolve an initial connection failure by retrying a connection.

```
ConnectionFactory factory = new ConnectionFactory();
// For RabbitMQ Java clients earlier than 4.0.0, enable the automatic recovery function.
factory.setAutomaticRecoveryEnabled(true);

// Configure connection settings.
try {
  Connection conn = factory.newConnection();
} catch (java.net.ConnectException e) {
  Thread.sleep(5000);
  // apply retry logic
}
```

Client Suggestions

- **Producer**

  Set the **Mandatory** parameter to **true** on the producer client to avoid message losses caused by network exceptions. This setting calls the **Basic.Return** method to return messages to the producer when they cannot be routed to matching queues.

  The following sample enables **Mandatory** on a Java client:

```
ConnectionFactory factory = new ConnectionFactory();
factory.setAutomaticRecoveryEnabled(true);
Connection conn = factory.newConnection();
channel = connection.createChannel();
channel.confirmSelect();
channel.addReturnListener((replyCode, replyText, exchange, routingKey, properties, body) -> {
    // Process the return value.
});
// The third parameter is Mandatory.
channel.basicPublish("testExchange", "key", true, null, "test".getBytes());
```

- **Consumer**

  Use idempotent messages on consumer clients to avoid repeated messages due to network exceptions.

  The following sample sets idempotence on a Java client:

```
Set<Long> messageStore = new HashSet<Long>();
channel.basicConsume("queue", autoAck, "a-consumer-tag",
        new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag,
                                       Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body)
                    throws IOException
            {
                long deliveryTag = envelope.getDeliveryTag();
                if (messageStore.contains(deliveryTag)) {
        // Idempotent processing
                    channel.basicAck(deliveryTag, true);
                } else {
                    try {
                        // handle message logic
          // Process the message.
                        handleMessage(envelope);
            // Normally acknowledge the message.
                        channel.basicAck(deliveryTag, true);
                        messageStore.add(deliveryTag);
                    } catch (Exception e) {
                        channel.basicNack(deliveryTag, true, true);
                    }
                }
            }
        });
```

3 Automatic Consumer Reconnection After a RabbitMQ Node Restart

Overview

amqp-client of RabbitMQ has a built-in reconnection mechanism with only one retry. If the reconnection fails, there will be no further retries which means that the connection is lost, and the consumer will no longer be able to consume messages, unless the consumer has an additional retry mechanism.

After amqp-client is disconnected from a node, different errors are generated depending on the node that the channel is connected to.

- If the channel is connected to the node where the queue is located, the consumer receives a shutdown signal. Then, the amqp-client reconnection mechanism takes effect and the consumer attempts to reconnect to the server. If the connection is successful, the channel continues to be connected for consumption. If the connection fails, the **channel.close** method is used to close the channel.
- If the channel is not connected to the node where the queue is located, consumer closure is not triggered. Instead, the server sends a cancel notification. This is not an exception for amqp-client, so no obvious error is reported in the log. However, the connection will be closed eventually.

When these two errors occur, amqp-client calls back the **handleShutdownSignal** and **handleCancel** methods. You can rewrite these methods to execute the rewritten reconnection logic during the callback. In this way, a new channel can be created for the consumer to continue consumption after a previous channel is closed.

Sample Code of Automatic Consumer Reconnection After a RabbitMQ Node Restart

The following is a Java code example which can solve the preceding two errors for continuous consumption.

```
package rabbitmq;

import com.rabbitmq.client.*;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeoutException;

public class RabbitConsumer {

    public static void main(String... args) throws IOException, TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        // Configure the connection address and port of the instance.
        factory.setHost("192.168.0.2");
        factory.setPort(5672);

        // Configure the username and password for instance connection.
        factory.setUsername("name");
        factory.setPassword("password");
        Connection connection = factory.newConnection();

        createNewConnection(connection);
    }

    // Reconnection
    public static void createNewConnection(Connection connection) {
        try {
            Thread.sleep(1000);
            Channel channel = connection.createChannel();
            channel.basicQos(64);
            channel.basicConsume("queue-01", false, new CustomConsumer(channel, connection));
        } catch (Exception e) {
//            e.printStackTrace();
            createNewConnection(connection);
        }
    }

    static class CustomConsumer implements Consumer {

        private final Channel _channel;
        private final Connection _connection;

        public CustomConsumer(Channel channel, Connection connection) {
            _channel = channel;
            _connection = connection;
        }

        @Override
        public void handleConsumeOk(String consumerTag) {
        }

        @Override
        public void handleCancelOk(String consumerTag) {

        }

        @Override
        public void handleCancel(String consumerTag) throws IOException {
            System.out.println("handleCancel");
            System.out.println(consumerTag);
            createNewConnection(_connection);
        }

        @Override
        public void handleShutdownSignal(String consumerTag, ShutdownSignalException sig) {
            System.out.println("handleShutdownSignal");
            System.out.println(consumerTag);
            System.out.println(sig.getReason());
            createNewConnection(_connection);
        }

        @Override
        public void handleRecoverOk(String consumerTag) {

        }
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, 
byte[] body) throws IOException {
            String message = new String(body, StandardCharsets.UTF_8);
            System.out.println(" [x] Received '" + message + "'");
            _channel.basicAck(envelope.getDeliveryTag(), false);
        }

    }
}
```

4 Improving RabbitMQ Performance

This topic introduces methods to achieve high RabbitMQ performance (considering throughput and reliability) by configuring the queue length, cluster load balancing, priority queues, and other parameters.

Using Short Queues

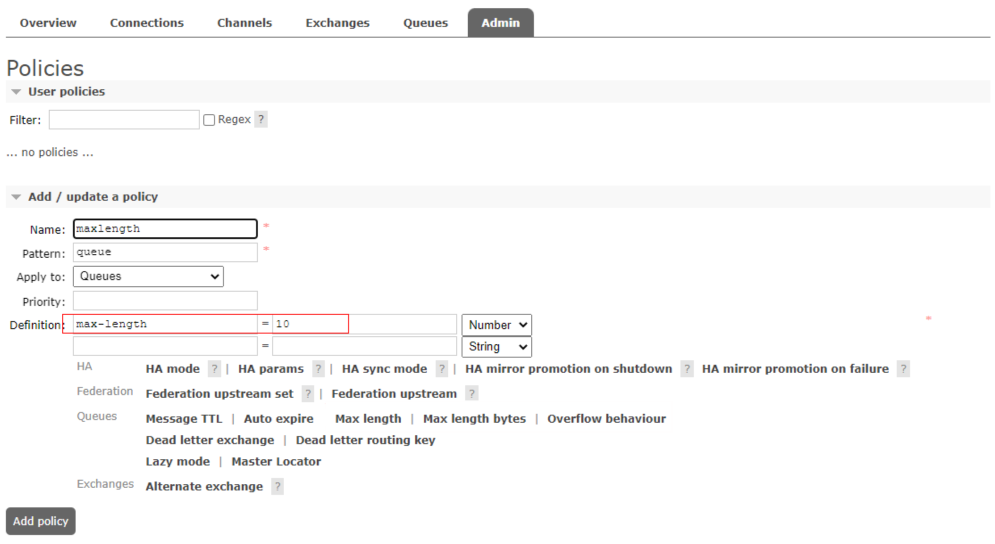
If a queue has a large number of messages, memory is under heavy pressure. To relieve pressure, RabbitMQ pages out messages to the disk. This process is time-consuming because it involves recreating the index on the disk and restarting a cluster that contains a large number of messages. If there are too many messages paged out to the disk, queues will be blocked, which slows down queue processing, and affects the performance of RabbitMQ nodes.

To achieve high performance, shorten queues as much as you can. You are advised to **keep no messages stacked in a queue**.

For applications that frequently encounter message count surges or require high throughput, you are advised to **limit the queue length**. The queue length can be kept within the limit by discarding messages at the head of a queue.

The limit can be configured in a policy or a queue declaration argument.

- Configuring a policy on the RabbitMQ management UI


Overview
Connections
Channels
Exchanges
Queues
Admin
Policies
User policies
Filter:
Regex
... no policies ...
Add / update a policy
HA: HA mode | HA params | HA sync mode | HA mirror promotion on shutdown | HA mirror promotion on failure
Federation: Federation upstream set | Federation upstream
Queues: Message TTL | Auto expire | Max length | Max length bytes | Overflow behaviour
Dead letter exchange | Dead letter routing key
Lazy mode | Master Locator
Exchanges: Alternate exchange
Add policy

Name: maxlength
Pattern: queue
Apply to: Queues
Priority:
Definition: max-length = 10 Number

- Configuring a queue declaration argument

```
// Create a queue.
HashMap<String, Object> map = new HashMap<>();
// Set the maximum queue length.
map.put("x-max-length",10 );
// Set the queue overflow mode, retaining the first 10 messages.
map.put("x-overflow","reject-publish" );
channel.queueDeclare(queueName,false,false,false,map);
```

By default, when the queue length exceeds the limit, messages at the head of the queue (the oldest messages) are discarded or become dead letter messages. The queue can also be processed in other ways by specifying the **overflow** parameter:

- If **overflow** is set to **drop-head**, the earliest messages at the head of the queue are discarded or made dead-letter, and the latest *n* messages are retained.
- If **overflow** is set to **reject-publish**, the latest messages are discarded, and the earliest *n* messages are retained.

NOTE

- If both these methods are used to set the maximum queue length, the smaller limit is used.
- Messages beyond the maximum queue length will be discarded.

Cluster Load Balancing

Queue performance depends a single CPU core. When the message processing capability of a RabbitMQ node reaches the bottleneck, you can expand the cluster to improve the throughput.

If multiple nodes are used, the cluster automatically distributes queues across the nodes. In addition to using a cluster, you can use the **Consistent hash exchange** plug-in to optimize load balancing: This plug-in uses an exchange to balance messages between queues. Messages sent to the exchange are consistently and evenly distributed across multiple queues based on the messages' routing keys. This plug-in creates a hash for the routing keys and distributes the messages to queues bound with the exchange. When using this plug-in, ensure that consumers consume messages from all queues. The following is an example:

- Route messages based on different routing keys.

```
public class ConsistentHashExchangeExample1 {
  private static String CONSISTENT_HASH_EXCHANGE_TYPE = "x-consistent-hash";

  public static void main(String[] argv) throws IOException, TimeoutException, InterruptedException {
    ConnectionFactory cf = new ConnectionFactory();
    Connection conn = cf.newConnection();
    Channel ch = conn.createChannel();

    for (String q : Arrays.asList("q1", "q2", "q3", "q4")) {
      ch.queueDeclare(q, true, false, false, null);
      ch.queuePurge(q);
    }

    ch.exchangeDeclare("e1", CONSISTENT_HASH_EXCHANGE_TYPE, true, false, null);

    for (String q : Arrays.asList("q1", "q2")) {
      ch.queueBind(q, "e1", "1");
    }

    for (String q : Arrays.asList("q3", "q4")) {
      ch.queueBind(q, "e1", "2");
    }

    ch.confirmSelect();

    AMQP.BasicProperties.Builder bldr = new AMQP.BasicProperties.Builder();
    for (int i = 0; i < 100000; i++) {
      ch.basicPublish("e1", String.valueOf(i), bldr.build(), "".getBytes("UTF-8"));
    }

    ch.waitForConfirmsOrDie(10000);

    System.out.println("Done publishing!");
    System.out.println("Evaluating results...");
    // wait for one stats emission interval so that queue counters
    // are up-to-date in the management UI
    Thread.sleep(5);

    System.out.println("Done.");
    conn.close();
  }
}
```

- Route messages based on headers.

  In this mode, the **hash-header** parameter must be specified for the exchange, and messages must contain headers. Otherwise, messages will be routed to the same queue.

```
public class ConsistentHashExchangeExample2 {
  public static final String EXCHANGE = "e2";
  private static String EXCHANGE_TYPE = "x-consistent-hash";

  public static void main(String[] argv) throws IOException, TimeoutException, InterruptedException {
    ConnectionFactory cf = new ConnectionFactory();
    Connection conn = cf.newConnection();
    Channel ch = conn.createChannel();

    for (String q : Arrays.asList("q1", "q2", "q3", "q4")) {
      ch.queueDeclare(q, true, false, false, null);
      ch.queuePurge(q);
    }

    Map<String, Object> args = new HashMap<>();
    args.put("hash-header", "hash-on");
    ch.exchangeDeclare(EXCHANGE, EXCHANGE_TYPE, true, false, args);

    for (String q : Arrays.asList("q1", "q2")) {
      ch.queueBind(q, EXCHANGE, "1");
    }

    for (String q : Arrays.asList("q3", "q4")) {
      ch.queueBind(q, EXCHANGE, "2");
    }

    ch.confirmSelect();


    for (int i = 0; i < 100000; i++) {
      AMQP.BasicProperties.Builder bldr = new AMQP.BasicProperties.Builder();
      Map<String, Object> hdrs = new HashMap<>();
      hdrs.put("hash-on", String.valueOf(i));
      ch.basicPublish(EXCHANGE, "", bldr.headers(hdrs).build(), "".getBytes("UTF-8"));
    }

    ch.waitForConfirmsOrDie(10000);

    System.out.println("Done publishing!");
    System.out.println("Evaluating results...");
    // wait for one stats emission interval so that queue counters
    // are up-to-date in the management UI
    Thread.sleep(5);

    System.out.println("Done.");
    conn.close();
  }
}
```

- Route messages based on their properties, such as **message_id**, **correlation_id**, or **timestamp**.

  In this mode, the **hash-property** parameter is required to declare the exchange, and messages must contain the specified property. Otherwise, messages will be routed to the same queue.

```
public class ConsistentHashExchangeExample3 {
  public static final String EXCHANGE = "e3";
  private static String EXCHANGE_TYPE = "x-consistent-hash";

  public static void main(String[] argv) throws IOException, TimeoutException, InterruptedException {
    ConnectionFactory cf = new ConnectionFactory();
    Connection conn = cf.newConnection();
    Channel ch = conn.createChannel();

    for (String q : Arrays.asList("q1", "q2", "q3", "q4")) {
      ch.queueDeclare(q, true, false, false, null);
      ch.queuePurge(q);
    }

    Map<String, Object> args = new HashMap<>();
    args.put("hash-property", "message_id");
    ch.exchangeDeclare(EXCHANGE, EXCHANGE_TYPE, true, false, args);

    for (String q : Arrays.asList("q1", "q2")) {
      ch.queueBind(q, EXCHANGE, "1");
    }

    for (String q : Arrays.asList("q3", "q4")) {
      ch.queueBind(q, EXCHANGE, "2");
    }

    ch.confirmSelect();


    for (int i = 0; i < 100000; i++) {
      AMQP.BasicProperties.Builder bldr = new AMQP.BasicProperties.Builder();
      ch.basicPublish(EXCHANGE, "", bldr.messageId(String.valueOf(i)).build(), "".getBytes("UTF-8"));
    }

    ch.waitForConfirmsOrDie(10000);

    System.out.println("Done publishing!");
    System.out.println("Evaluating results...");
    // wait for one stats emission interval so that queue counters
    // are up-to-date in the management UI
    Thread.sleep(5);

    System.out.println("Done.");
    conn.close();
  }
}
```

Automatically Deleting Unused Queues

The client may fail to be connected, resulting in residual queues that affect instance performance. RabbitMQ provides the following methods to automatically delete a queue:

- Set a TTL policy for the queue. For example, if TTL is set to 28 days, the queue will be deleted after staying idle for 28 days.

- Use an auto-delete queue. When the last consumer exits or the channel or connection is closed (or when its TCP connection with the server is lost), the auto-delete queue is deleted.
- Use an exclusive queue. This queue can be used only in the connection where it is created. When the connection is closed or disappears, the exclusive queue is deleted.

To enable the auto-delete and exclusive queues:

```
boolean exclusive = true;
boolean autoDelete = true;
channel.queueDeclare(QUEUENAME, durable, exclusive, autoDelete, arguments);
```

Limiting the Number of Priority Queues

Each priority queue starts an Erlang process. If there are too many priority queues, performance will be affected. In most cases, you are advised to have no more than five priority queues.

Connections and Channels

Each connection uses about 100 KB memory (or more if TLS is used). Thousands of connections cause high RabbitMQ load and even out-of-memory in extreme cases. The AMQP protocol introduces the concept of channels. Each connection can have multiple channels. Connections exist for a long time. The handshake process for an AMQP connection is complex and requires at least seven TCP data packets (or more if TLS is used). By contrast, it is easier to open and close a channel, and it is recommended that channels exist for a long time. For example, the same channel should be reused for a producer thread, and should not be opened for each production. The best practice is to reuse connections and multiplex a connection between threads with channels.

The Spring AMQP thread pool is recommended. ConnectionFactory is defined by Spring AMQP and is responsible for creating connections.

Do Not Share Channels Between Threads

Most clients do not implement thread safety security on channels, so do not share channels between threads.

Do Not Open and Close Connections or Channels Frequently

Frequently opening and closing connections or channels will lead to a large number of TCP packets being sent and received, resulting in higher latency.

Producers and Consumers Use Different Connections

This improves throughput. If a producer sends too many messages to the server for processing, RabbitMQ transfers the pressure to the TCP connection. If messages are consumed on the same TCP connection, the server may not receive acknowledgments from the client, affecting the consumption performance. If consumption is too slow, the server will be overloaded.

RabbitMQ Management Interface Performance Affected by Too Many Connections and Channels

RabbitMQ collects data of each connection and channel for analysis and display. If there are too many connections and channels, the performance of the RabbitMQ management interface will be affected.

Disabling Unused Plug-ins

Plug-ins may consume a large number of CPU or memory resources. You are advised to disable unused plug-ins.

5 Configuring Queue Load Balancing

Overview

On a RabbitMQ cluster, scaling up nodes or deleting queues cause uneven queues on each node. Some nodes are overloaded.

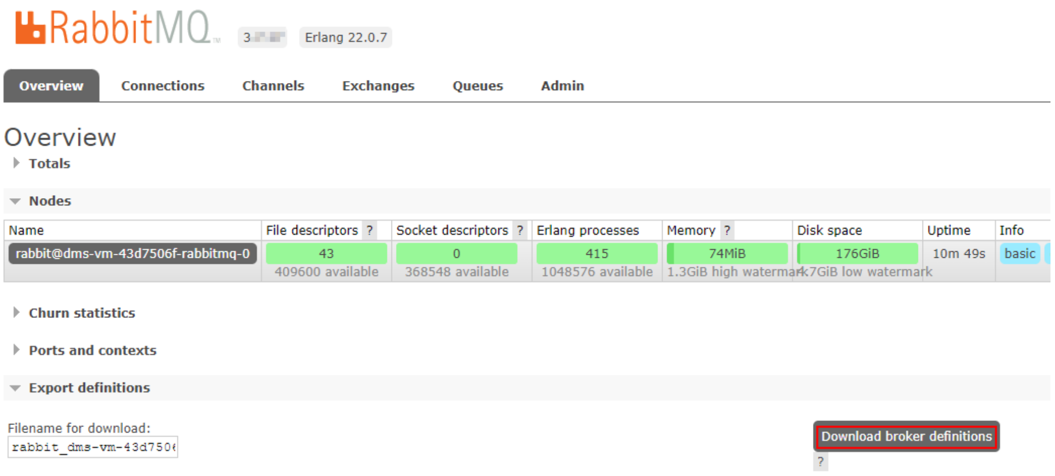
However, queue load can be manually balanced among nodes for better cluster utilization. To configure queue load balancing, use the following methods:

- Deleting and Recreating Queues
- Modifying the Master Node Using a Policy

Deleting and Recreating Queues

Step 1 Log in to the RabbitMQ management UI.

Step 2 On the **Overview** tab page, click **Download broker definitions** to export the metadata.

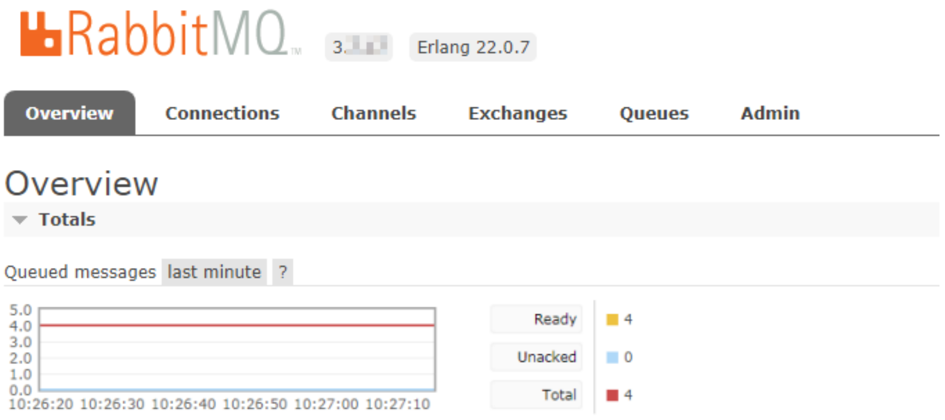


Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info
rabbit@dms-vm-43d7506f-rabbitmq-0	43 409600 available	0 368548 available	415 1048576 available	74MiB 1.3GiB high watermark	176GiB 47GiB low watermark	10m 49s	basic

Filename for download:
rabbit_dms-vm-43d750(

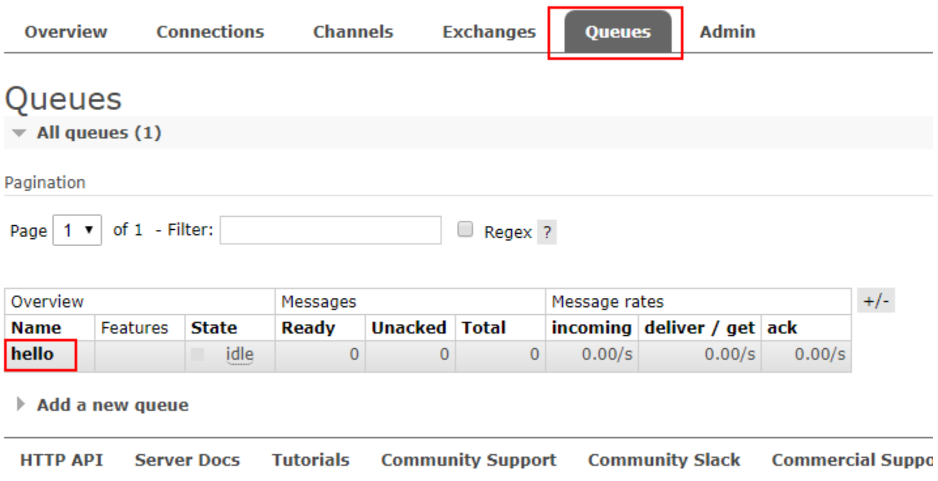
Step 3 Stop producing messages, wait until all messages are consumed, and then delete the original queues.

1. On the **Overview** tab page, check data consumption.



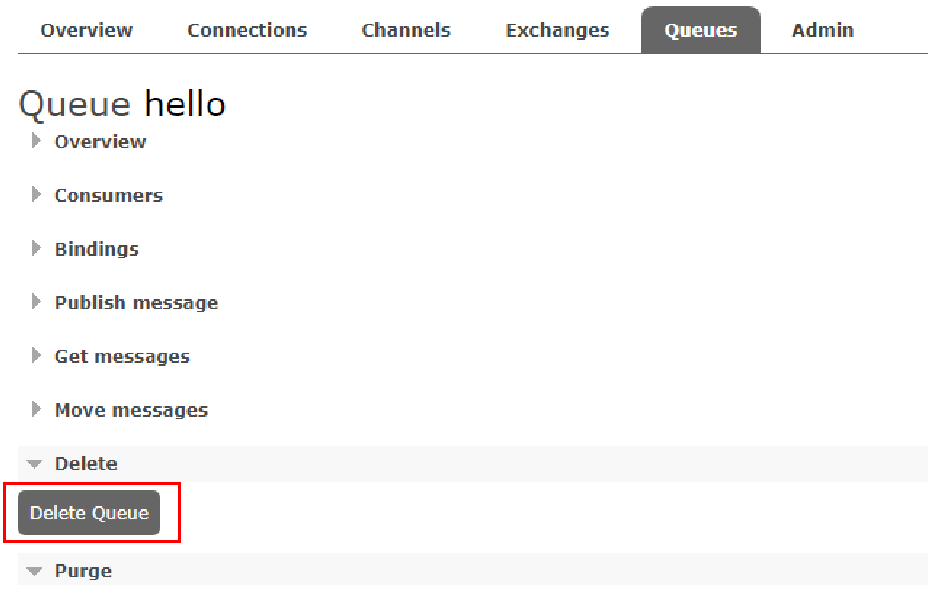
If the number of messages that can be consumed (**Ready**) and the number of messages that are not acknowledged (**Unacked**) are both 0, the consumption is complete.

2. When all data is consumed, delete the original queues.
   a. On the **Queues** tab page, click the name of the desired queue.



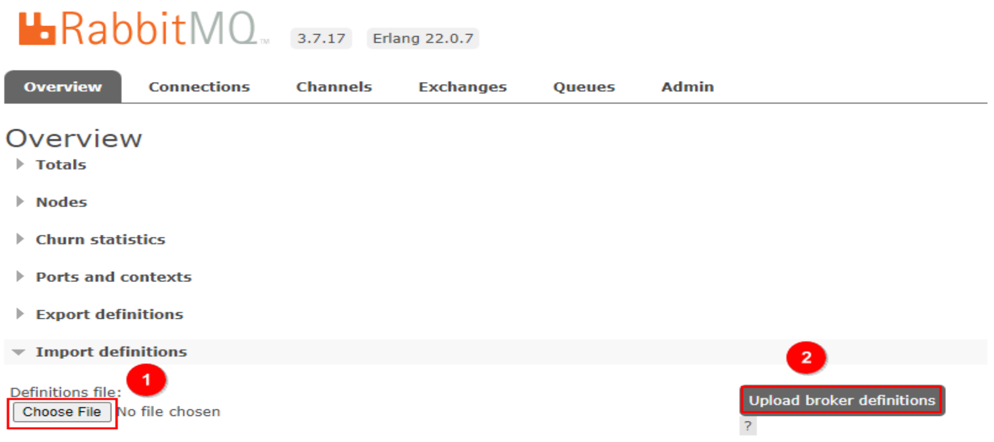
Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
hello		idle	0	0	0	0.00/s	0.00/s	0.00/s

   b. Click **Delete Queue** to delete the queue.

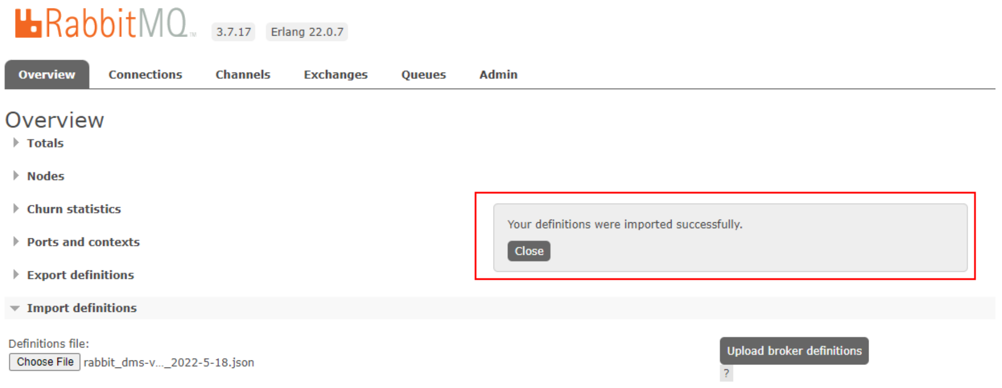


Step 4 On the **Overview** tab page, upload the exported metadata.

1. On the **Overview** tab page, click **Choose File** and select the exported metadata.
2. Click **Upload broker definitions** to upload the metadata.



If the upload is successful, the following information is displayed:



Your definitions were imported successfully.

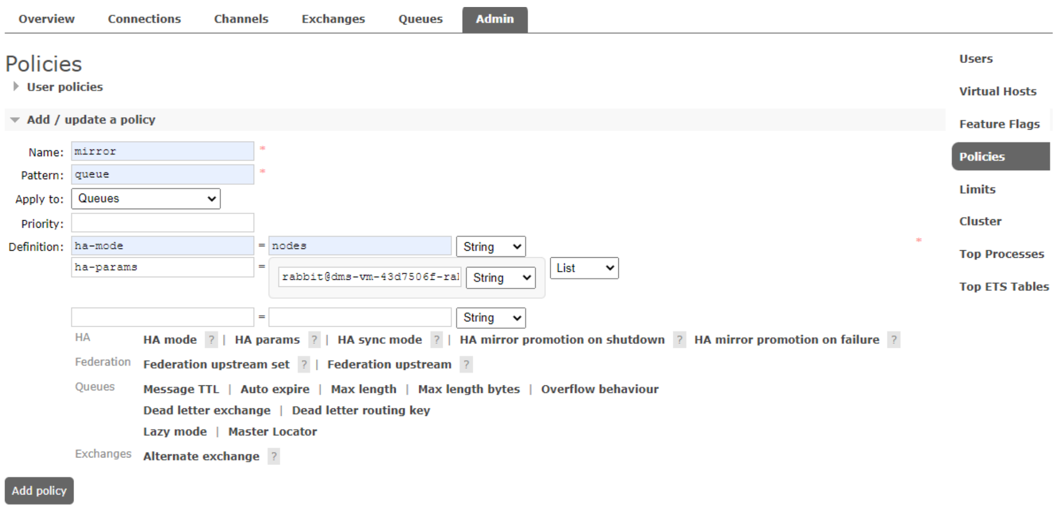
The instance automatically creates queues across nodes for load balancing. You can view the queue distribution details on the **Queues** tab page.

----End

Modifying the Master Node Using a Policy

Step 1 Log in to the RabbitMQ management UI.

Step 2 On the **Admin** > **Policies** page, add a policy.


Overview
Connections
Channels
Exchanges
Queues
Admin
Users
Virtual Hosts
Feature Flags
Policies
Limits
Cluster
Top Processes
Top ETS Tables
HA
HA mode | HA params | HA sync mode | HA mirror promotion on shutdown | HA mirror promotion on failure
Federation
Federation upstream set | Federation upstream
Queues
Message TTL | Auto expire | Max length | Max length bytes | Overflow behaviour
Dead letter exchange | Dead letter routing key
Lazy mode | Master Locator
Exchanges
Alternate exchange
Add policy

Policies

User policies

Add / update a policy

Name: mirror
Pattern: queue
Apply to: Queues
Priority:
Definition: ha-mode = nodes String
ha-params = rabbit@dms-vm-43d7506f-ral String List

- **Name**: Enter a policy name.
- **Pattern**: queue matching mode. Enter a queue name. Queues with the same prefix will be matched.
- **Apply to**: Select **Queues**.
- **Priority**: policy priority. A larger value indicates a higher priority.
- **Definition**: mapping definitions. Set **ha-mode** to **nodes** and **ha-params** to the name of node to which the queues are to be migrated.

Step 3 Click **Add policy**.

NOTE

- Queue data synchronization takes a long time. To prevent message loss, the original master node is still available before queue data synchronization is complete.
- After the queue switchover is complete, you can delete the policy added in **Step 2**.

----End

6 Deduplicating Messages Through Message Idempotence

Overview

In RabbitMQ 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.

Take paying as an example. Assume that a user selects a product, makes payment, and receives multiple bills due to unstable Internet connection. The bills are all paid. However, the billing should take place only once and the merchant should generate only one order placement. In this case, idempotent messages can be used to avoid the repetition.

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 the same content.

- A consumer repeatedly consumes a message:

  A message is successfully delivered to a consumer and processed. If the server does not receive a response from the consumer 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 consumes two messages of the same content.

Implementation

A globally unique ID can be used to determine whether a message is consumed repeatedly. If yes, the consumption result is returned. If no, consume the message and record the global ID.

- Producers set a unique ID for each message. Sample code is as follows:

```
// The message is persisted with a globally unique ID which is randomly generated.
AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties().builder();
builder.deliveryMode(2);
builder.messageId(UUID.randomUUID().toString());

// The custom message.
String message = "message content";

// Produce messages. Set exchangeName and routingKey to the actual values.
channel.basicPublish("exchangeName", "routingKey", false, builder.build(),
message.getBytes(StandardCharsets.UTF_8));
String messageId = builder.build().getMessageId();
System.out.println("messageID: " + messageId);
System.out.println("Send message success!");
// Close the channel.
channel.close();
// Close the connection.
connection.close();
```

- Consumers deduplicate messages based on their IDs. Sample code is as follows:

```
// Create a table with message ID as the primary key. Unique primary keys of databases can be used
to process RabbitMQ idempotence.
// Before consumption, query the message from the database. If the message exists, it is consumed.
Otherwise, consume it.
//queueName: Use the actual queue name.
channel.basicConsume("queueName", false, new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties
properties, byte[] body) throws IOException {
            //Obtain the message ID and check whether it is null.
            String messageId = properties.getMessageId();
            if (StringUtils.isBlank(messageId)){
                logger.info("messageId is null");
                return;
            }
            // Query the database by primary key "message ID". If records exist, the message is
consumed. Otherwise, consume the message and write it into the database.
            // Database querying logic ...
            //todo

            // If no record exists, consume the message. Or notify that the message is consumed.
            if (null == "{Found in the database}"){
                // Obtain the message.
                String message = new String(body,StandardCharsets.UTF_8);
                // Manual response.
                channel.basicAck(envelope.getDeliveryTag(),false);
                logger.info("[x] received message: "  + message + "," + "messageId:" + messageId);

                // Save the message to the database table, indicating that the message has been
consumed.
                // The database input operation ...
                //todo
            } else {
                // If the message is consumed, skip it.
                logger.error("The message is already consumed.");
            }
    }
});
```