Distributed Message Service for RabbitMQ

Best Practices

Issue 01

Date 2024-04-01

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 RabbitMQ Services

Scenario

There are two RabbitMQ service migration scenarios:

- Migrating an on-premises single-node or cluster RabbitMQ instance to a cloud RabbitMQ instance.
- Migrating an earlier RabbitMQ instance to a later one, for example, from 3.7.17 to 3.8.35.

Migration Principles

A RabbitMQ instance has multiple producers and consumers. Services are migrated by adding and removing consumers and producers one by one without changing data. This process does not affect services.

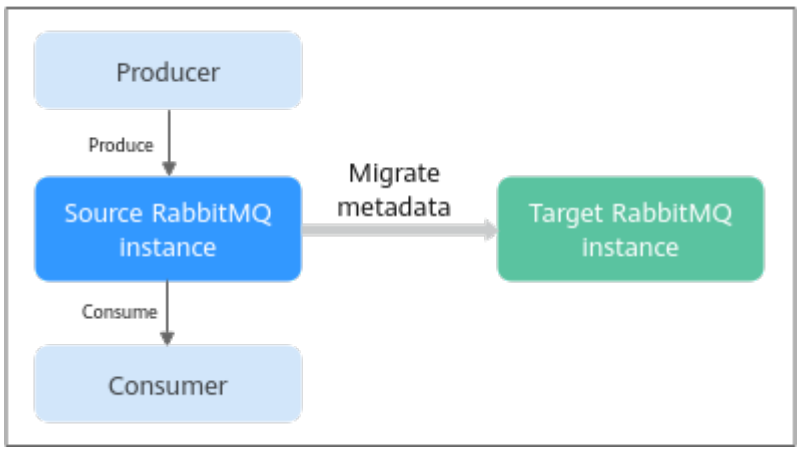
Preparation

A cloud RabbitMQ instance has been created. For details, see **Buying an Instance**.

Implementation (Dual-Read)

Step 1 Migrate source RabbitMQ instance metadata to a target RabbitMQ instance.

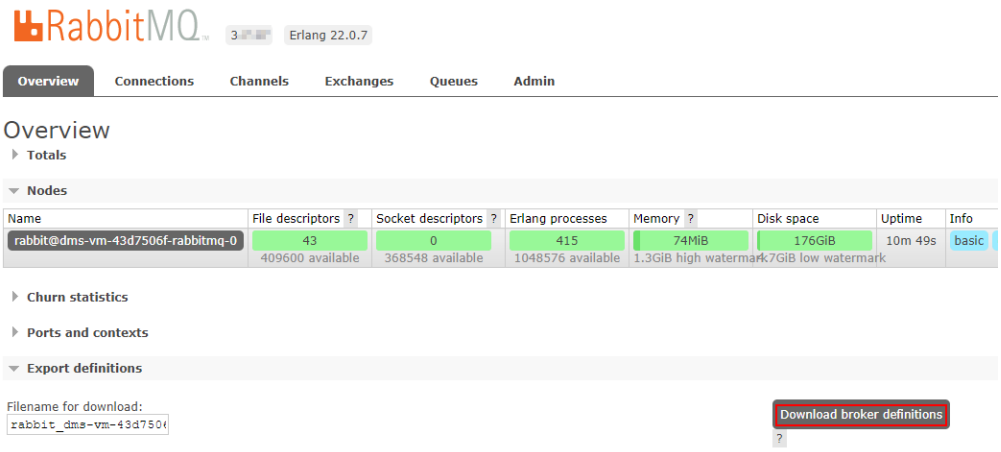
Figure 1-1 Migrating metadata



Do as follows:

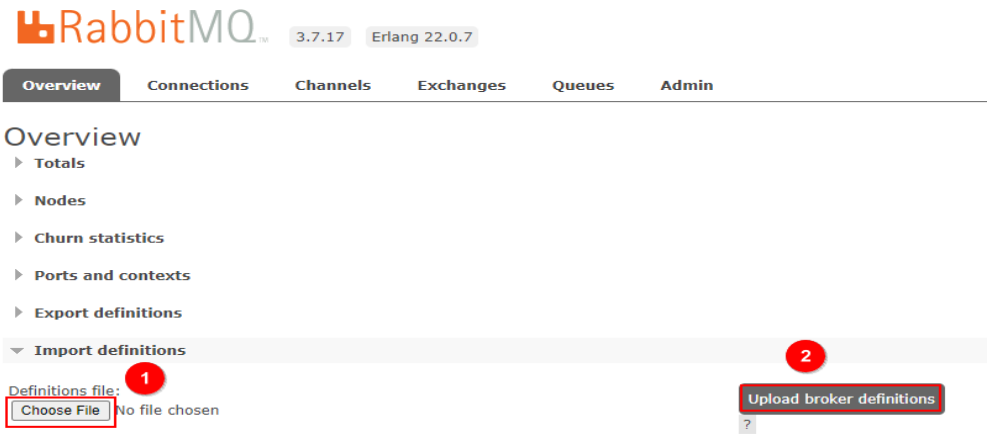
1. Log in to the management UI of the source RabbitMQ. On the **Overview** tab page, click **Download broker definitions** to export the metadata.

   Figure 1-2 Exporting metadata

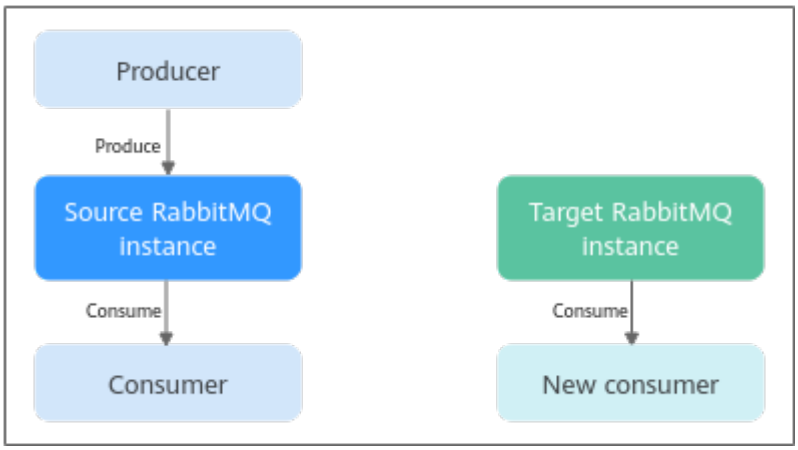
2. Log in to the management UI of the target RabbitMQ. On the **Overview** tab page, click **Choose File** and select the metadata exported in **Step 1.1**, and click **Upload broker definitions** to upload the metadata.

   Figure 1-3 Importing metadata

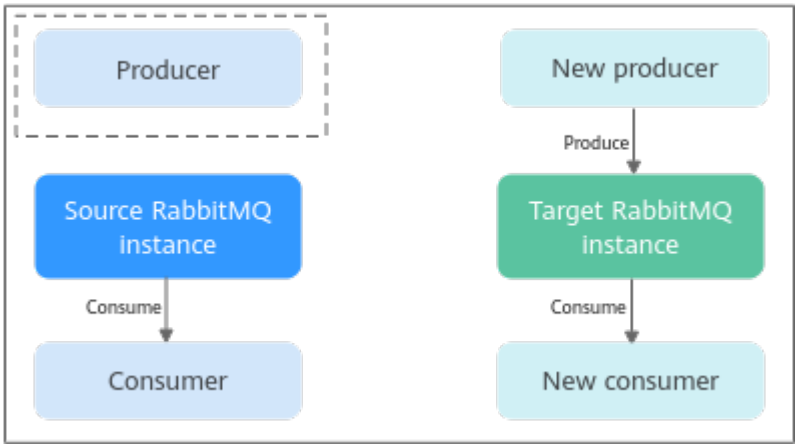
Step 2 Add new consumers for the target RabbitMQ instance.

Figure 1-4 Adding new consumers



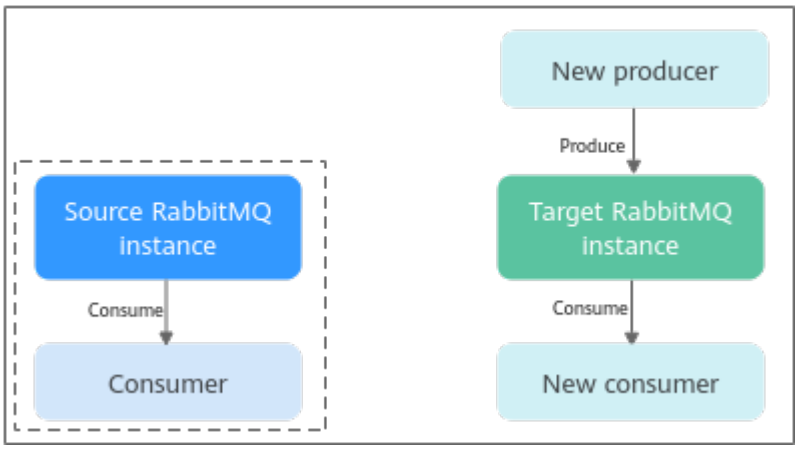
Step 3 Add new producers for the target RabbitMQ instance and remove the producers of the source RabbitMQ instance. The old consumers continue consuming messages from the source RabbitMQ instance.

Figure 1-5 Migrating producers



Step 4 Remove the old consumers and the source RabbitMQ instance after the old consumers have consumed all messages from the source RabbitMQ instance.

Figure 1-6 Removing an old consumer and a source RabbitMQ instance



----End

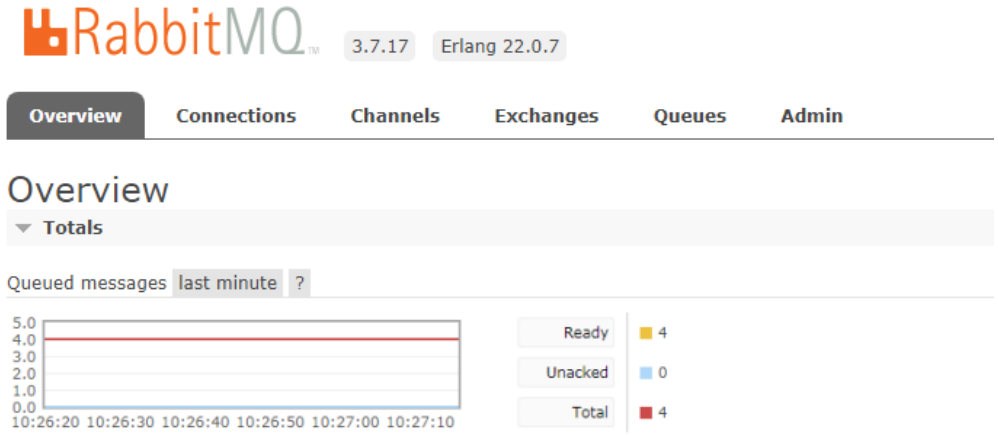
Check After Migration

Check whether all messages in the source instance have been consumed by using the following methods:

- Using the RabbitMQ management UI, as shown in **Figure 1-7**.

  On the **Overview** tab page, 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.

  Figure 1-7 RabbitMQ management UI

- Using an API

  curl -s -u *username*:*password* -XGET http://*ip*:*port*/api/overview

  Parameter description:

  – *username*: account used to log in to the RabbitMQ Management UI for the source instance

  – *password*: password used to log in to the RabbitMQ Management UI for the source instance

  – *ip*: IP address used to log in to the RabbitMQ Management UI for the source instance

– *port*: port used to log in to the RabbitMQ Management UI for the source instance

When the values of **messages_ready** and **messages_unacknowledged** in the command output are both **0**, the consumption is complete.

Figure 1-8 Command output


```
"queue_totals":{
        "messages":4,
        "messages_details":{
                "rate":0
        },
        "messages_ready":4,
        "messages_ready_details":{
                "rate":0
        },
        "messages_unacknowledged":0,
        "messages_unacknowledged_details":{
                "rate":0
        }
},
```


2 Queue Migration

If queues are not evenly distributed across the nodes in a RabbitMQ cluster due to node scale-out or queue deletion, some nodes will be overloaded and the cluster cannot be effectively used.

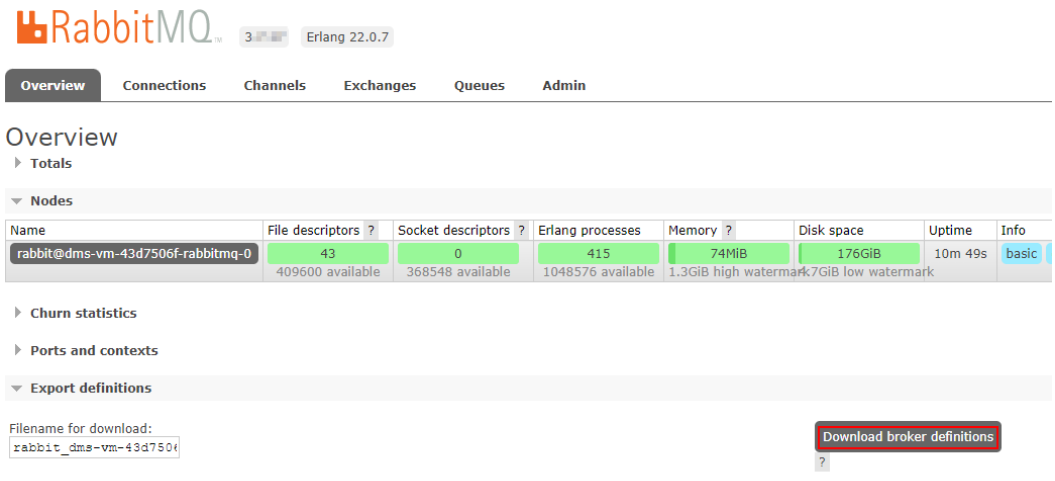
To configure queue load balancing, use the following methods:

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

Deleting and Recreating Queues

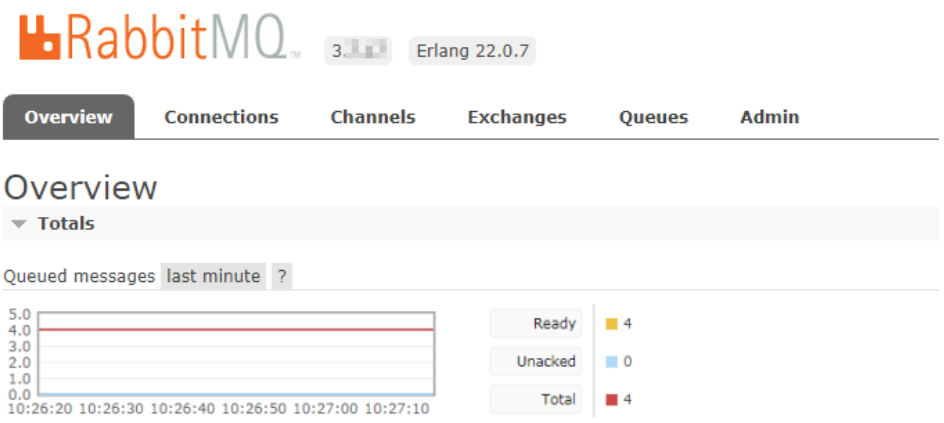
Step 1 **Log in to the RabbitMQ management UI**.

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



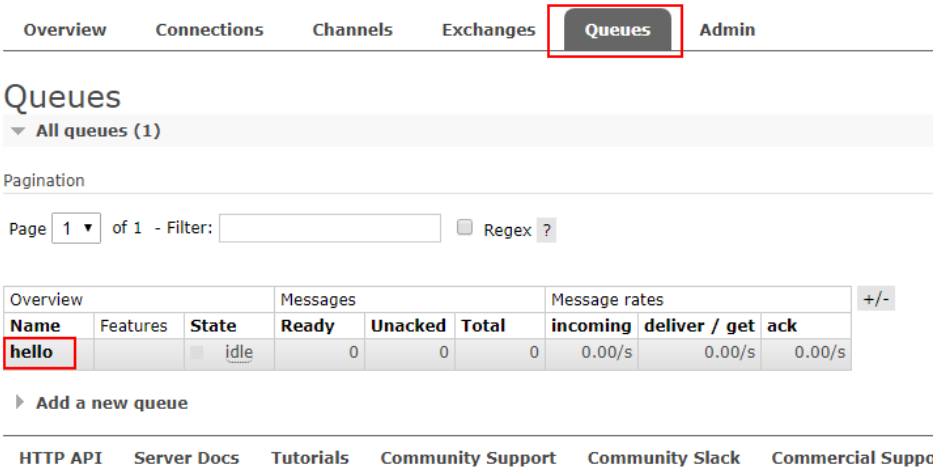
Step 3 Stop producing messages, wait until all messages are consumed, and then delete the original queues.

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

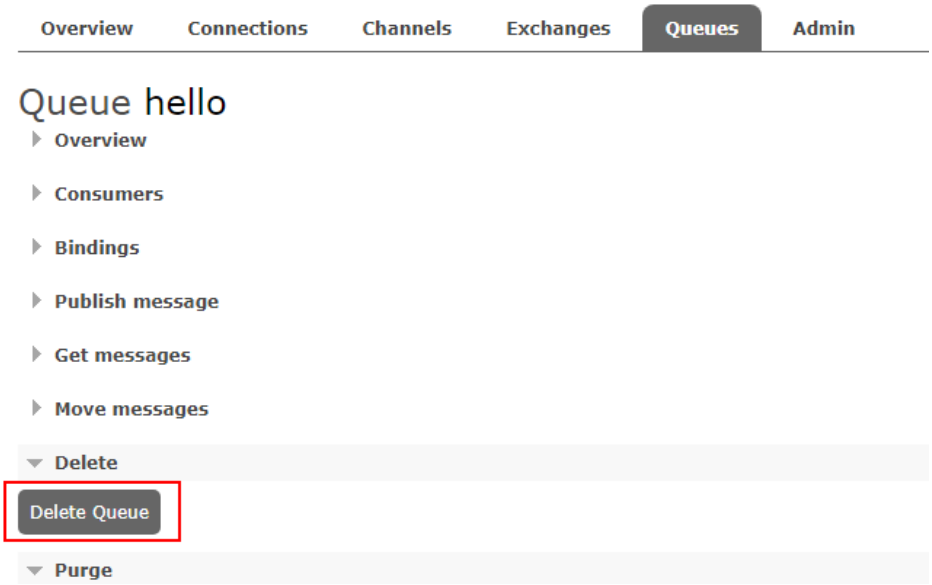


If the number of messages that can be consumed (**Ready**) and the number of messages that are not acknowledged (**Unacked**) are both 0, the consumption is complete.

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

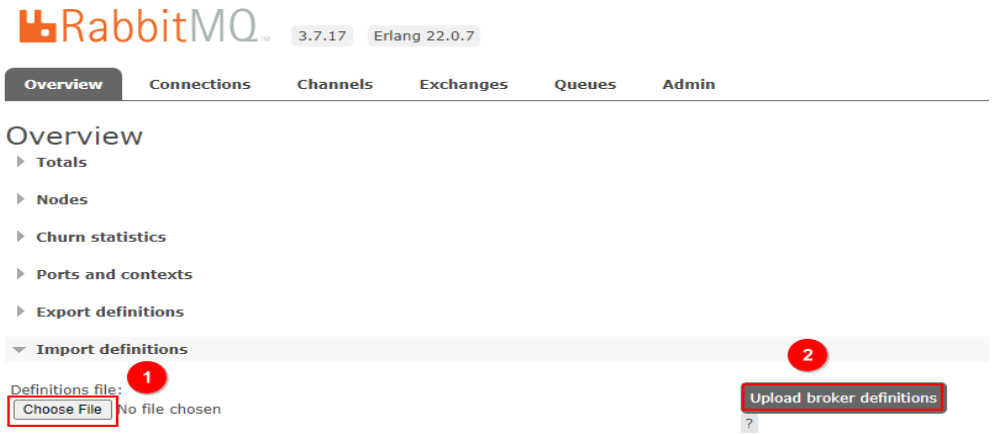


   b. Click **Delete Queue** to delete the queue.

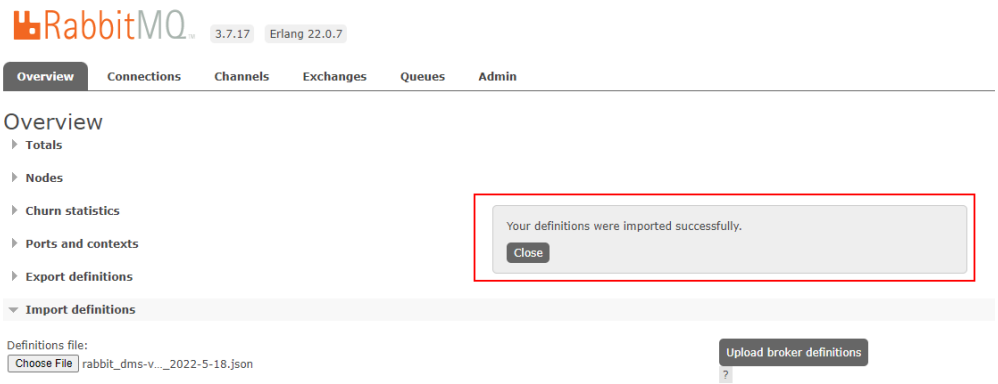


Step 4 On the **Overview** tab page, upload the exported metadata.

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:



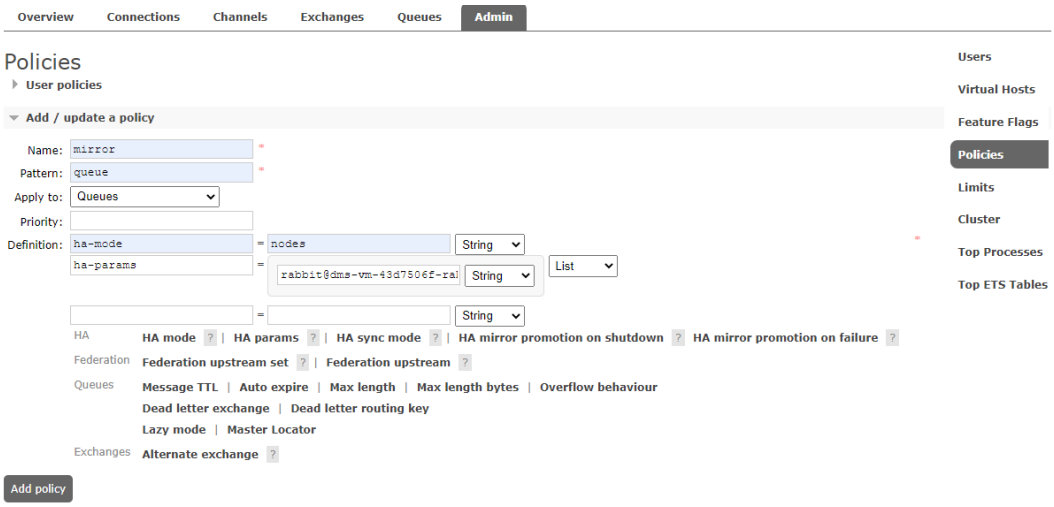
The instance automatically creates queues across nodes for load balancing. You can view the queue distribution details on the **Queues** tab page.

----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.



- **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. This parameter is optional.
- **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

3 Automatic Recovery from Network Exceptions

This topic describes how to configure automatic network recovery on a client when it is disconnected from the server due to server restart or network jitter. **Java clients of version 4.0.0 or later support automatic network recovery by default.**

NOTICE

If an application uses the **Connection.Close** method to close a connection, automatic network recovery will not be enabled or triggered.

Scenarios

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.

Sample Code for Reconnection

If the initial connection between the client and server fails, automatic recovery is not triggered. You are advised to edit the corresponding application code 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();
// enable automatic recovery if using RabbitMQ Java client library prior to version 4.0.0.
factory.setAutomaticRecoveryEnabled(true);
// configure various connection settings

try {
  Connection conn = factory.newConnection();
} catch (java.net.ConnectException e) {
  Thread.sleep(5000);
```



```
  // apply retry logic
}
```


4 Consumer Reconnection After a Node Restart

This section uses amqp-client, a RabbitMQ client in Java, as an example to describe how to reconnect to a node after the node is restarted.

amqp-client has a built-in reconnection mechanism with only one retry. If the reconnection fails, there will be no further retries 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.

The following is a simple 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();
        factory.setHost("100.00.000.000");
        factory.setPort(5672);

        factory.setUsername("name");
        factory.setPassword("password");
        Connection connection = factory.newConnection();

        createNewConnection(connection);
    }

    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);
        }

    }
}
```


5 RabbitMQ High 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

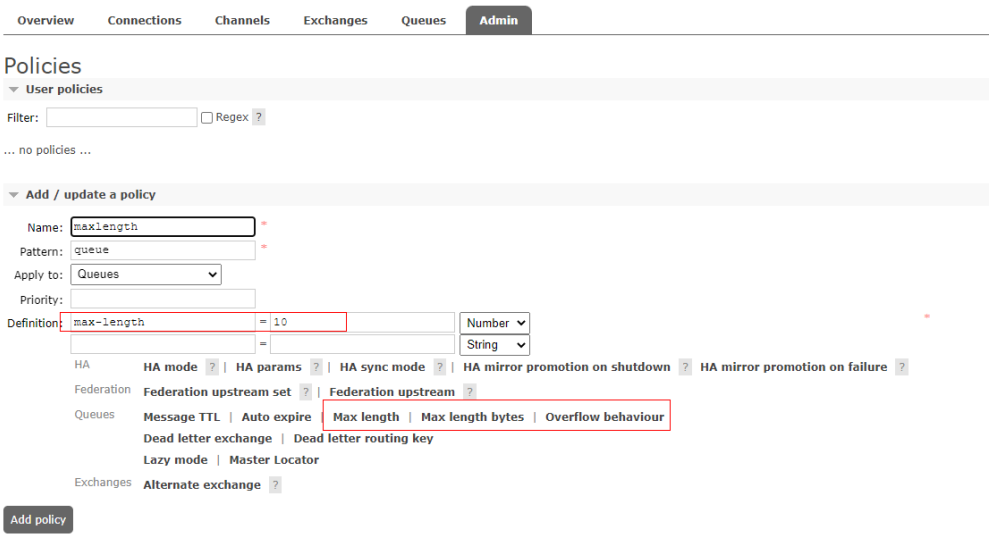
If a queue has a large number of messages, memory is under heavy pressure. To relieve pressure, RabbitMQ pages out messages to the disk. This process usually takes a long time because it involves recreating the index and restarting a cluster that contains a large number of messages. If there are too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



- 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. You can change this mode by setting **overflow** to different values. If **overflow** is set to **drop-head**, the oldest 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 oldest *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 following two plug-ins to optimize load balancing:

Consistent hash exchange

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 different 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();
  }
}
```

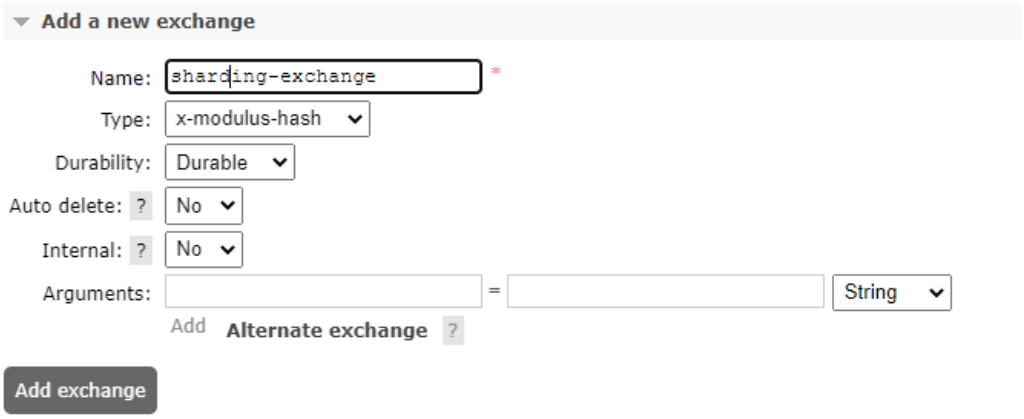
RabbitMQ sharding

This plug-in automatically partitions queues. Once you define an exchange as sharded, supporting queues are automatically created on each cluster node to share messages. This plug-in provides a centralized location for sending messages and implements load balancing by adding queues to other nodes in the cluster.

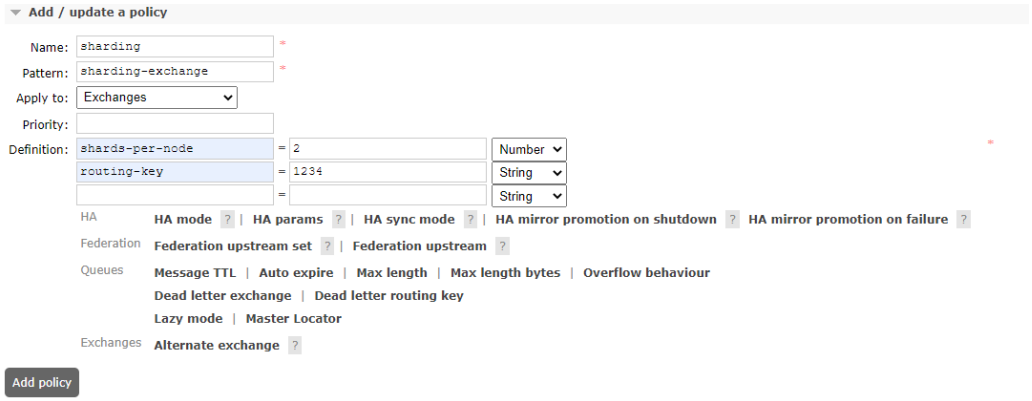
When using this plug-in, ensure that consumers consume messages from all queues.

Do as follows to configure the RabbitMQ sharding plug-in:

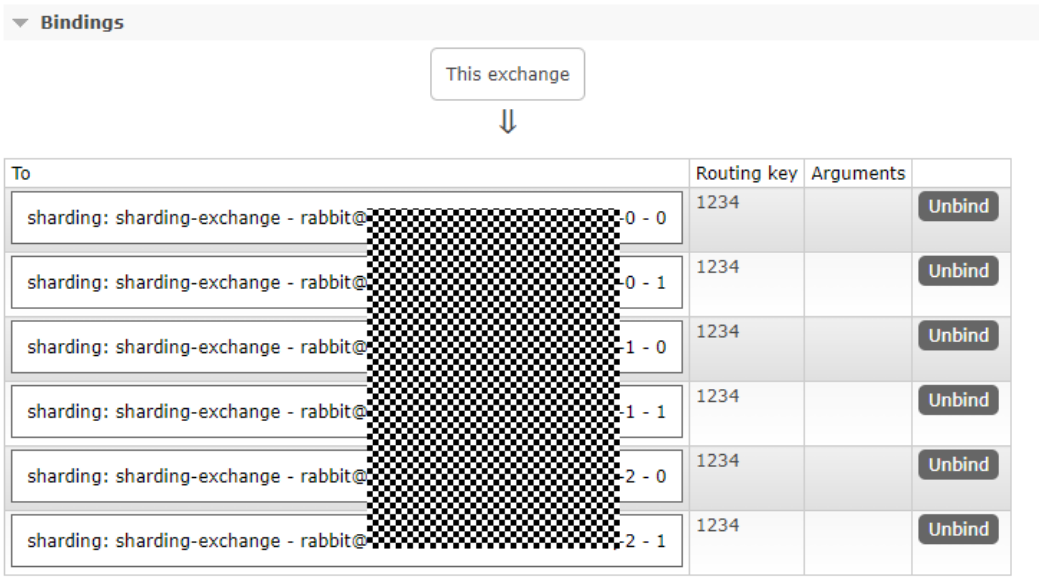
Step 1 Create an x-modulus-hash exchange.



Step 2 Add a policy to the exchange.



Step 3 View the exchange details to check whether the configuration is successful.



----End

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.

Configuration:

```
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.

