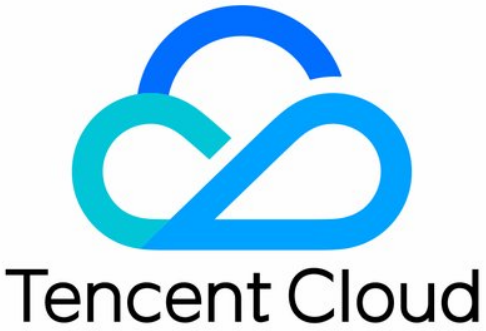
# TDMQ for RabbitMQ Practical Tutorial



## Copyright Notice



## Trademark Notice



## Service Notice

This document provides an overview of the as-is details of Tencent Cloud's products and services in their entirety or part. The descriptions of certain products and services may be subject to adjustments from time to time.

The commercial contract concluded by you and Tencent Cloud will provide the specific types of Tencent Cloud products and services you purchase and the service standards. Unless otherwise agreed upon by both parties, Tencent Cloud does not make any explicit or implied commitments or warranties regarding the content of this document.

## Contact Us

We are committed to providing personalized pre-sales consultation and technical after-sale support. Don't hesitate to contact us at 4009100100 or 95716 for any inquiries or concerns.

# Contents

# Practical Tutorial
# RabbitMQ Client Practice Tutorial

Last updated: 2024-08-02 15:48:01

## Please do not create a connection/channel every time you send a message

Creating a connection in RabbitMQ is a time-consuming/resource-consuming operation. Each connection uses at least 100KB of memory, and too many connections will increase the memory pressure on the Broker. It is recommended to create a connection when the program starts and reuse this long connection every time a message is sent to improve sending performance and reduce server memory usage.
Channels are a lighter form of communication and it is recommended to use as many channels as possible to reuse connections. However, it is best not to use the same channel across threads concurrently, as many RabbitMQ client implementations are not thread-safe.

## Set a reasonable send timeout time for producers

RabbitMQ clients of different languages and versions have set different default send times, with some clients having an excessively long default timeout, such as 580 seconds or 900 seconds. In the event of a network anomaly, an excessively long send timeout can block the sending thread and even cause an avalanche effect. It is recommended to set a reasonable timeout time according to the business scenario, with 3 seconds being a recommended value.

## Producers and consumers use separate connections

Due to RabbitMQ's unique flow control mechanism, if producers and consumers reuse the same physical connection, and the consumer traffic triggers flow control, it may cause the producer to be flow-controlled, leading to slow sending or timeouts. Therefore, it is recommended that producers and consumers use different physical connections during initialization to avoid mutual interference.

## It is not recommended for consumers to enable automatic message acknowledgment

RabbitMQ server provides at least once delivery semantics to ensure that messages are correctly delivered to downstream business systems. Once the consumer enables automatic message acknowledgment, the server will automatically confirm and delete the message after pushing it to the consumer side, even if there is an exception during message processing, which may lead to missed message processing in the business.

## Consumer idempotent processing of messages

RabbitMQ server provides at least once delivery semantics, and in extreme cases, messages may be delivered repeatedly. Therefore, it is recommended that key business processes must perform idempotent processing when handling messages, so that even if duplicate messages are received, there will be no negative business impact.
Business idempotent processing can be achieved by adding a unique business identifier to the message, and the consumer checks such identifiers and message status during consumption, processing duplicate messages according to business needs, ensuring that even if duplicate messages are received, there will be no negative business impact.

## Limit queue length to avoid a large accumulation of messages

An excessively long queue (a large accumulation of messages) will occupy a lot of memory and consume more server system resources. Not only does it take longer to perform state synchronization during operation, but it also leads to a significant increase in the startup recovery time of the server Broker.
A shorter queue will provide faster processing speed and system performance.
Therefore, it is necessary for the client to improve the consumption capacity as much as possible, and use queue dimension limitations such as max-length to ensure that the queue is as short as possible.

## Use Consume or Get to consume messages?

Get is a polling-based pull consumption mode where each message consumed requires a request to be sent to the Broker. If there are no messages in the queue, it might result in a large number of ineffective empty pulls leading to resource occupation. On the other hand, Consume can receive a batch of messages at once, with the server pushing messages based on actual conditions. In most cases, Consume should be used instead of Get for message consumption.
If the business process must use Get to consume messages, attention should be paid to the Get mechanism at the business level to avoid continuous Get Empty pulls (when the queue has no messages pending consumption, but the consumer continuously performs Get), causing high server CPU load.

## Set a reasonable Prefetch Count for consumers

Prefetch setting is a mechanism where the consumer side, in order to improve consumption throughput, pushes messages to the consumer’s cache in advance, reducing consumption waiting time and latency. However, if the Prefetch Count is too high or unlimited, it can lead to a large number of messages being cached on the consumer side, and the server Broker also maintaining the state of unacknowledged messages in memory, occupying a lot of resources;

if messages remain in unacknowledged state, these messages cannot be consumed by other idle consumers, manifesting as increased consumption delay or unbalanced consumer load. It is recommended to set the Prefetch Count within a reasonable range according to the business consumption rate.

## Set a reasonable exception handling strategy for consumers

When consuming messages, encountering unprocessable exceptions, messages not set for automatic acknowledgment will trigger message retries. If the exception continues without a normal exit, it will trigger infinite retries of the message, not only causing a high load on the Broker side but also preventing subsequent messages from being reasonably consumed.

## Ensure client reconnection mechanism

In extreme scenarios such as OOM, host machine failure, etc., the server Broker may self-heal and restart. Everyday business operations such as cluster upgrade can also trigger Broker restart. To avoid continuous connection exceptions during the Broker's restart period, please ensure that the client has implemented an automatic reconnection mechanism.

## Do not disable the heartbeat setting in the client SDK

heartbeat has a configuration value on both the server and client side (60 seconds for the server), the effective heartbeat is determined through negotiation between the server and client, and different languages/versions of clients have different negotiation mechanisms. Setting heartbeat=0 on the client side, which turns off heartbeat detection, will prevent the server from automatically removing long-term idle connections, potentially leading to unexpected connection leaks.

# RabbitMQ Message Reliability Practice Guide

Last updated: 2024-08-02 15:48:28

## Message Persistence

To ensure that the queue metadata and the messages within the queue are not lost after the Broker restarts, it is recommended to set the queue as durable and the messages as persistent. This way, the queue will immediately persist the messages to disk upon receiving them.
Non-persistent messages will also occupy more server-side memory resources, potentially causing high memory load on the server under extreme conditions.

## Sender Confirmation

The Confirmation Mechanism can ensure that the message is successfully sent to the Broker. However, if the mandatory setting is not configured when sending the message, the Broker will respond with a confirm to the sender regardless of whether the message is successfully routed to the target queue. If the mandatory setting is configured (delayed exchange does not support the mandatory setting), the Broker will return the message to the client if it cannot be routed. The client can sense these unroutable messages by implementing **basic.return** handling; the Broker will only respond with a confirm to the sender when the message is successfully routed to the target queue.

## Consumption End Acknowledgement

The ACK Mechanism at the consumption end ensures that the client receives the messages, providing at least once level consumption semantics guarantee, ensuring the message is correctly processed before it can be deleted. However, the client also needs to implement idempotence to avoid errors from consuming duplicate messages, and messages that are not ACKed will pile up in memory, increasing memory usage on both the client and server sides.

## Enabling Image Queue

Image Queue ensures the high availability of the queue by replicating the queue data to other Brokers within the cluster. Configuring the image queue policy may increase Broker startup duration and resource usage, but it ensures that the queue remains available in the event of a single Broker failure, making every effort to prevent message loss.
When configuring the image queue policy, it is advisable to avoid setting ha-sync-mode=automatic, as this configuration will trigger automatic full synchronization of the queue

data after the server-side Broker restarts (regardless of whether the queue data has been previously synchronized). If the queue accumulates too much data, it will eventually result in a prolonged synchronization time, continued memory resource usage, and the queue will remain unavailable until synchronization is complete. This has serious implications on business availability and server stability.

# Notes

Last updated: 2024-08-02 15:48:57

## The rabbitmq_delayed_message_exchange plugin implements delayed messages

1. The current plugin design is not suitable for scenarios with a large volume of delayed messages (hundreds of thousands or even millions of un-scheduled messages). In a production environment, please carefully assess the message volume to avoid unexpected long delays and message loss.
2. Delayed messages have only one persistent replica on each node. If the node cannot run properly (e.g., due to continuous OOM caused by message accumulation, leading to a restart and failure to recover), the delayed messages on that node cannot be consumed by the consumer.
3. The delayed exchange does not support setting **mandatory**. Producers cannot be aware of unrouted messages through the **basic.return** event. Therefore, before sending delayed messages, please ensure the corresponding exchange, queue, and routing relationships exist.

In summary, we strongly recommend against using this plugin and instead using dead letter queues to indirectly implement Delayed Messages. If you still choose to use this plugin after understanding its several flaws, we highly recommend keeping the number of delayed messages as low as possible to avoid triggering high memory load issues.

## Network Partition

1. Network partitioning is an issue that must be faced when using RabbitMQ. Network partitions can lead to inconsistencies in cluster states, and even after network recovery, RabbitMQ still needs to restart the Broker to resynchronize the state. Tencent Cloud RabbitMQ currently uses the autoheal mode, which automatically determines a winning partition and then restarts the Brokers within the non-trusted partition.
2. We recommend that clients take the following measures to minimize the negative impact of network partitions:
   - Message sender, consider using the **mandatory** mechanism when sending messages and have the ability to handle **basic.return** events to timely handle message routing failures during network partitions.
   - Message consumer, during the occurrence/processing of network partitions, there may be message duplication, and the consumer side needs to handle idempotent processing.

## Alarm Configuration

Tencent Cloud provides multi-dimensional monitoring metrics such as cluster/node, etc. For details, see Monitoring and Alarm.
We strongly recommend focusing on node CPU, memory, disk utilization rate, message backlog, and other indicators and configuring alarms to avoid continuous high server load affecting cluster stability.

## Message Tracking Usage Limitations

Overview of the implementation principles of message querying: After the Trace plugin is enabled in the Tencent Cloud console for a VHost, the service component will consume the corresponding RabbitMQ cluster's trajectory messages, and after a series of processing, it can realize the feature of querying message trajectories in the console.
Based on the above principle, message tracking depends on the service component consuming trajectory messages. Since the service component is a underlying public service, it cannot guarantee that the trajectory messages of RabbitMQ clusters with high traffic can be consumed in time; if trajectory messages accumulate, it will cause high memory load issues, affecting RabbitMQ cluster stability.
Therefore, it is not recommended to enable the Trace plugin in production environments, especially in scenarios where the overall cluster (including all VHosts) sends TPS exceeding 1000. The Trace plugin is recommended for use in low-traffic verification/troubleshooting scenarios and is not suitable for production environments.

# RabbitMQ MQTT Protocol Usage Instructions

Last updated: 2024-08-02 15:49:31

## Solution Introduction

MQTT is a widely used IoT protocol, and RabbitMQ is a widely used open-source message queue product based on the AMQP 0.9.1 protocol. RabbitMQ supports the MQTT protocol through plugins, making it easy to support MQTT on a RabbitMQ cluster to cater to IoT and other business scenarios.
Community Reference Documentation:

1. RabbitMQ versions before 3.11 support MQTT through a plugin: MQTT Plugin — RabbitMQ
2. RabbitMQ version 3.12 natively supports MQTT: Serving Millions of Clients with Native MQTT | RabbitMQ - Blog

## Operation step

### Step 1: Purchase a cloud or self-built RabbitMQ cluster

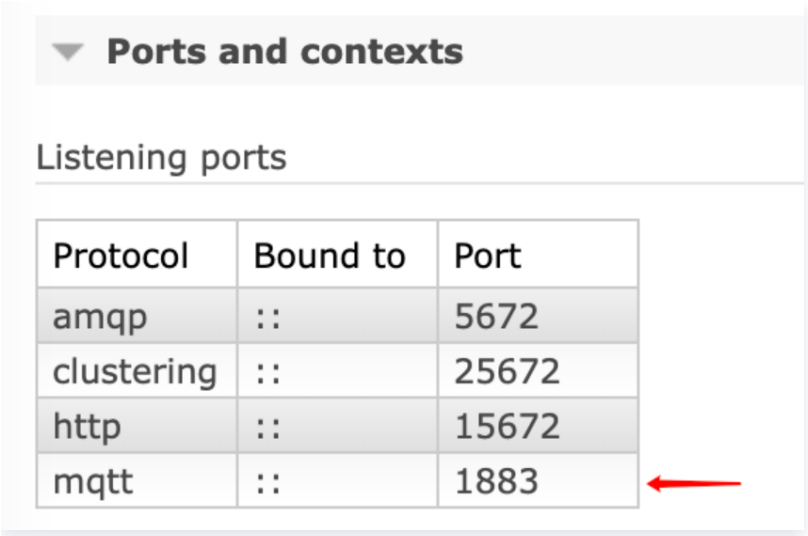
Purchase a RabbitMQ cluster on cloud, Buy Now .
Or build your own RabbitMQ cluster, see details: Downloading and Installing RabbitMQ — RabbitMQ .

### Step 2: Enable the MQTT plugin

Enable the MQTT plugin by executing the following command on cluster nodes:

```
sudo  rabbitmq-plugins enable rabbitmq_mqtt
```

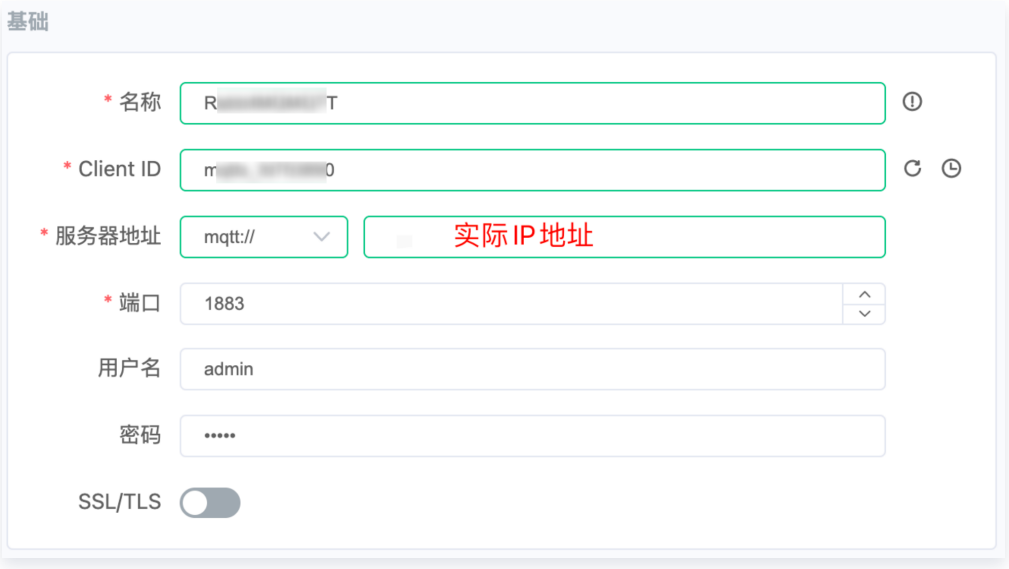
Tencent Cloud's RabbitMQ plugin management feature is under development. Currently, you can submit a ticket to enable the MQTT plugin and set up network operations.
After enabling the MQTT plugin, you can see the newly added port 1883 in the console:




## Step 3: Verify the availability of MQTT

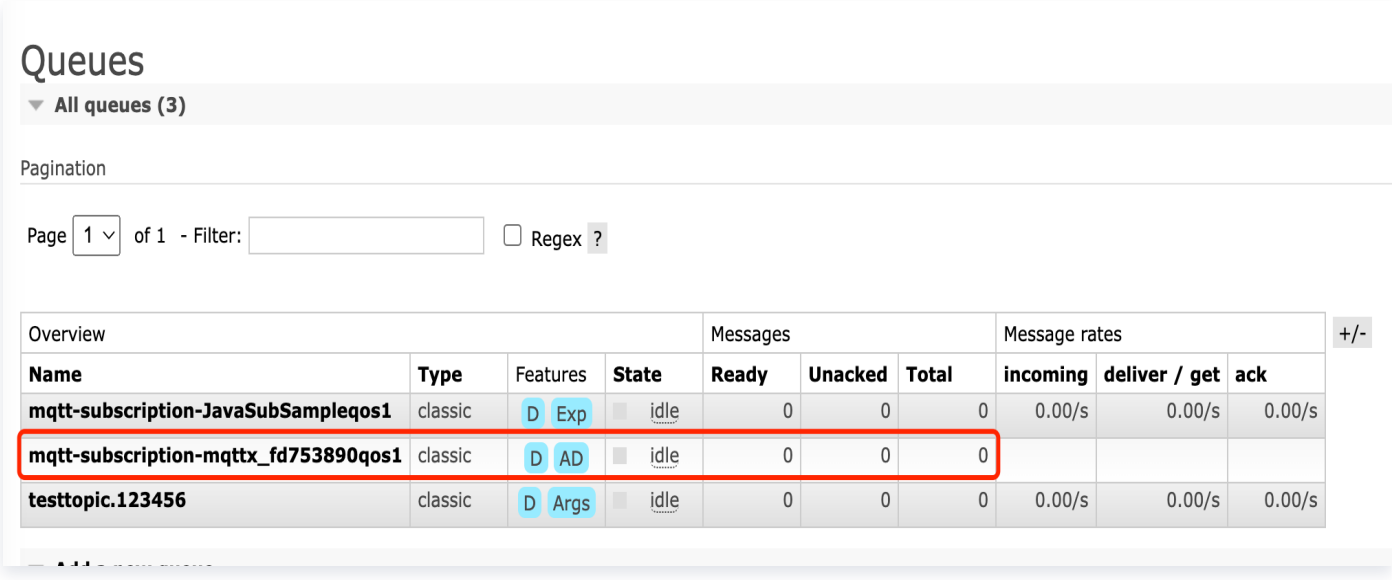
You can download the commonly used mqttx (MQTTX: Full-featured MQTT client tool) client tool to verify:

1. Create a new connection, and fill in the address and port. Use the RabbitMQ username and password for the username and password fields.

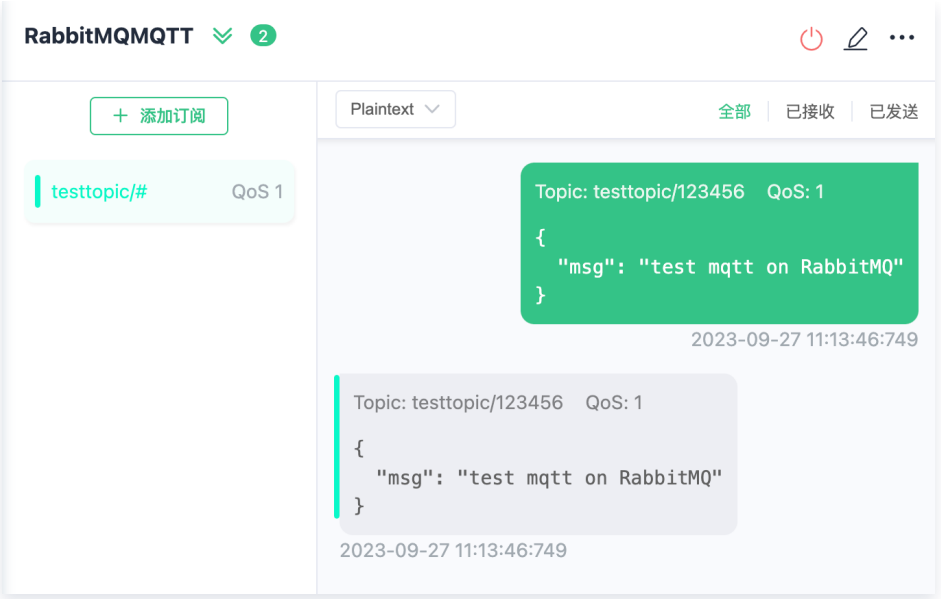



2. Create a new subscription to subscribe to the testtopic/# topic messages.

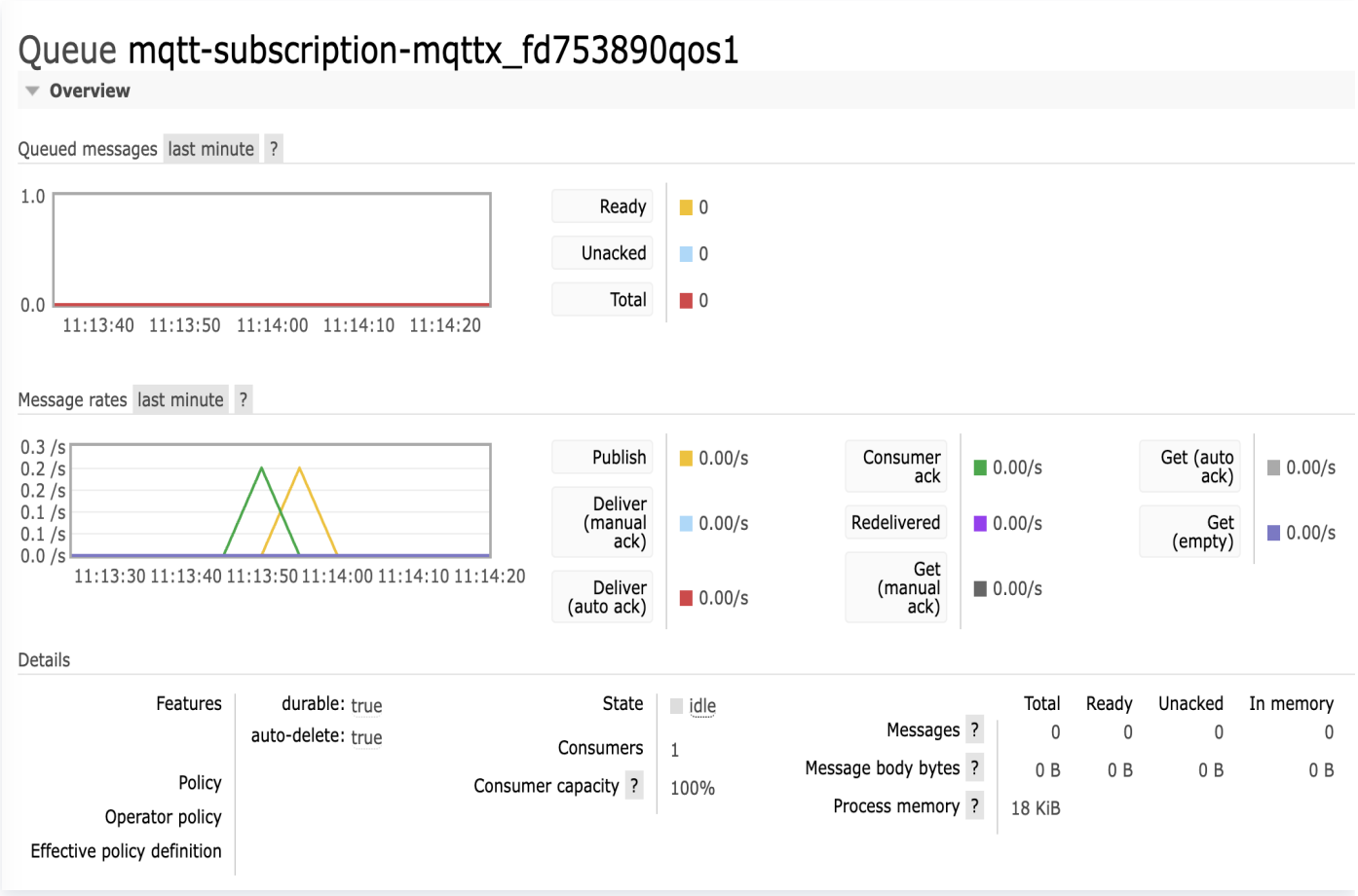

3. Check the RabbitMQ queue status, you can see that a new queue is created in RabbitMQ for each subscription.



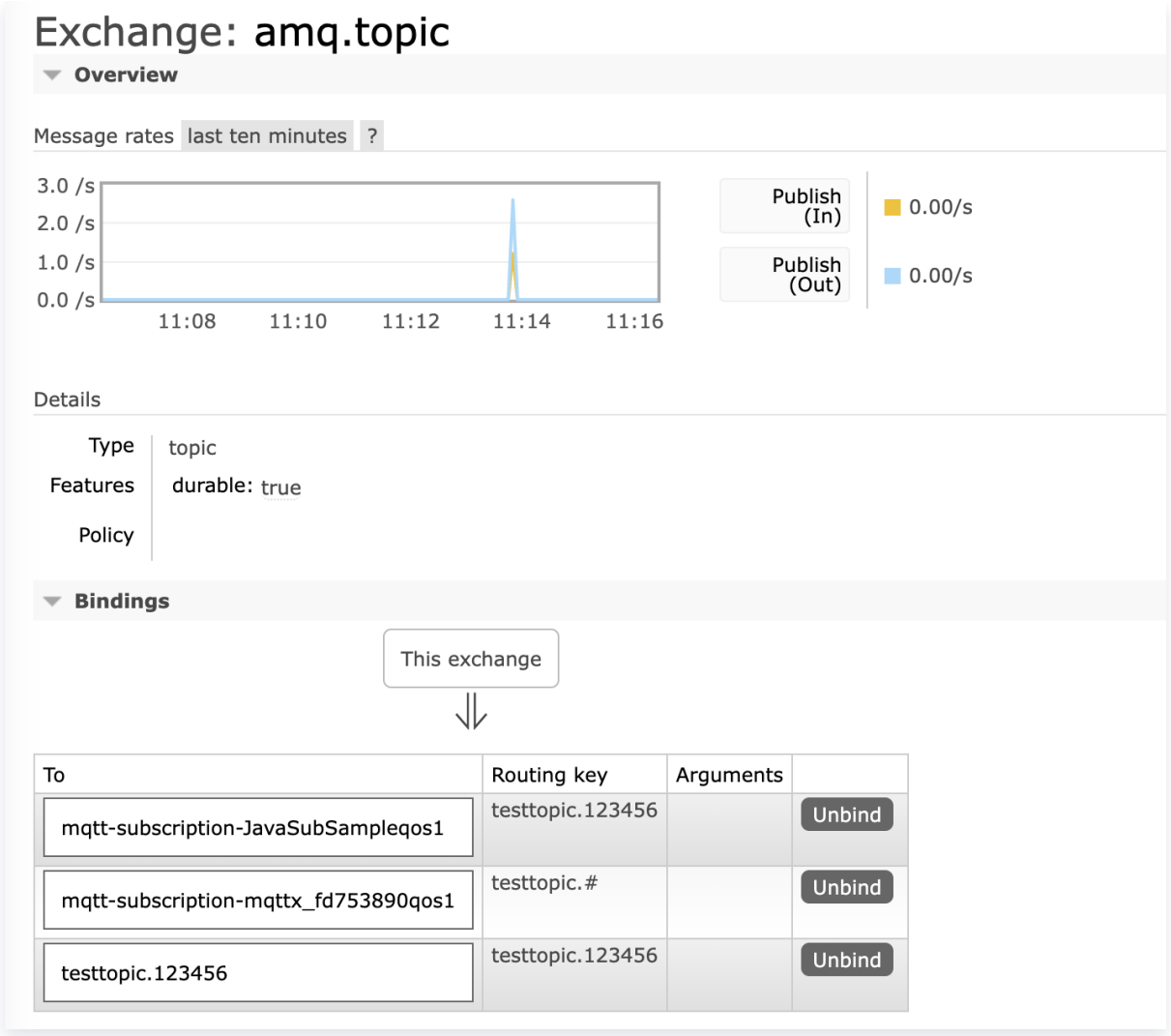

4. Verify message sending and receiving. Send a testtopic/123456 message, and it will be received immediately through the subscription.

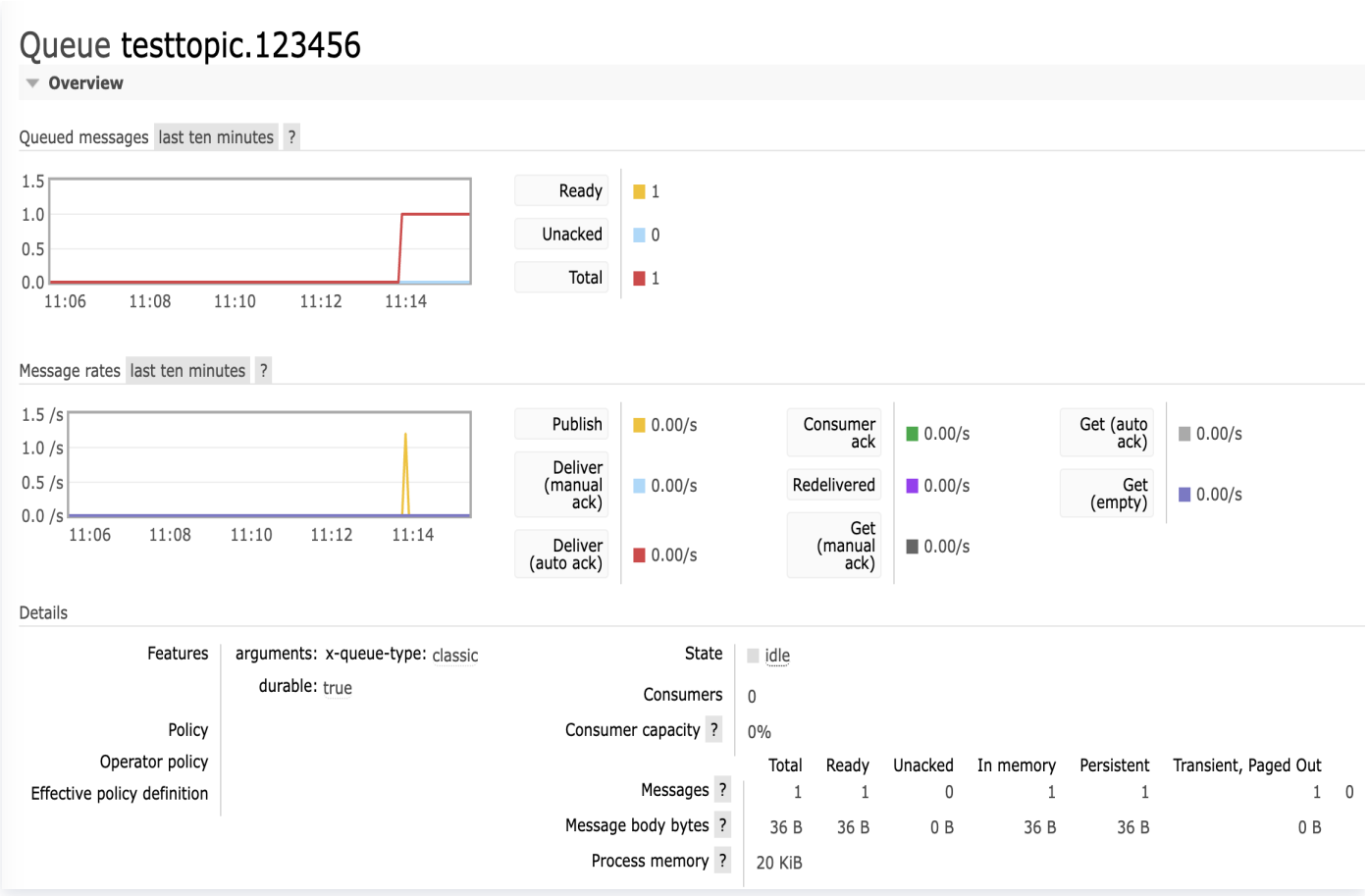



5. View RabbitMQ monitoring, you can see that the just-created queue has one send-receive message monitoring record.

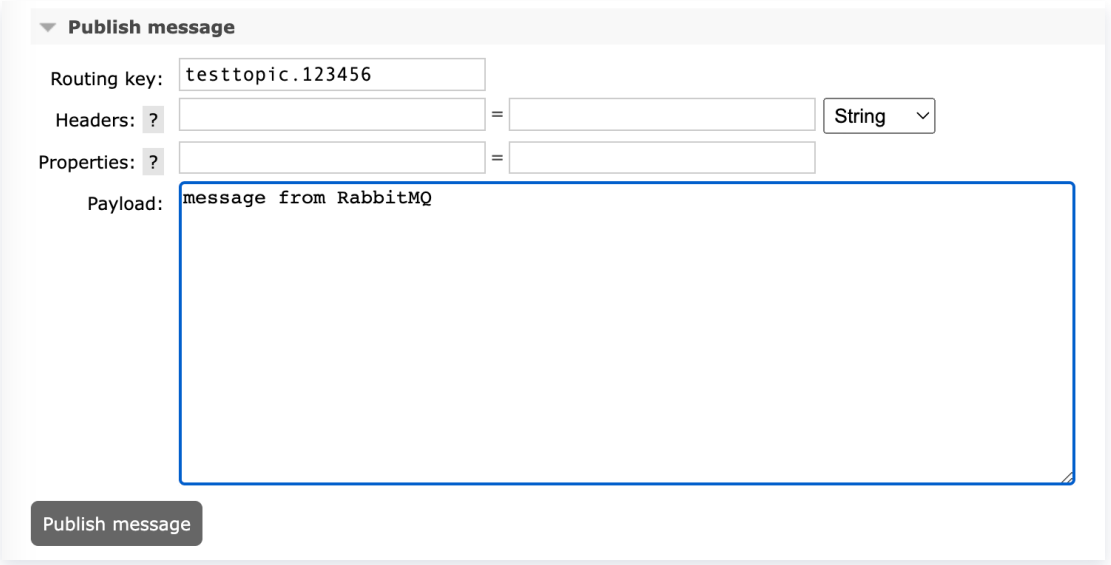


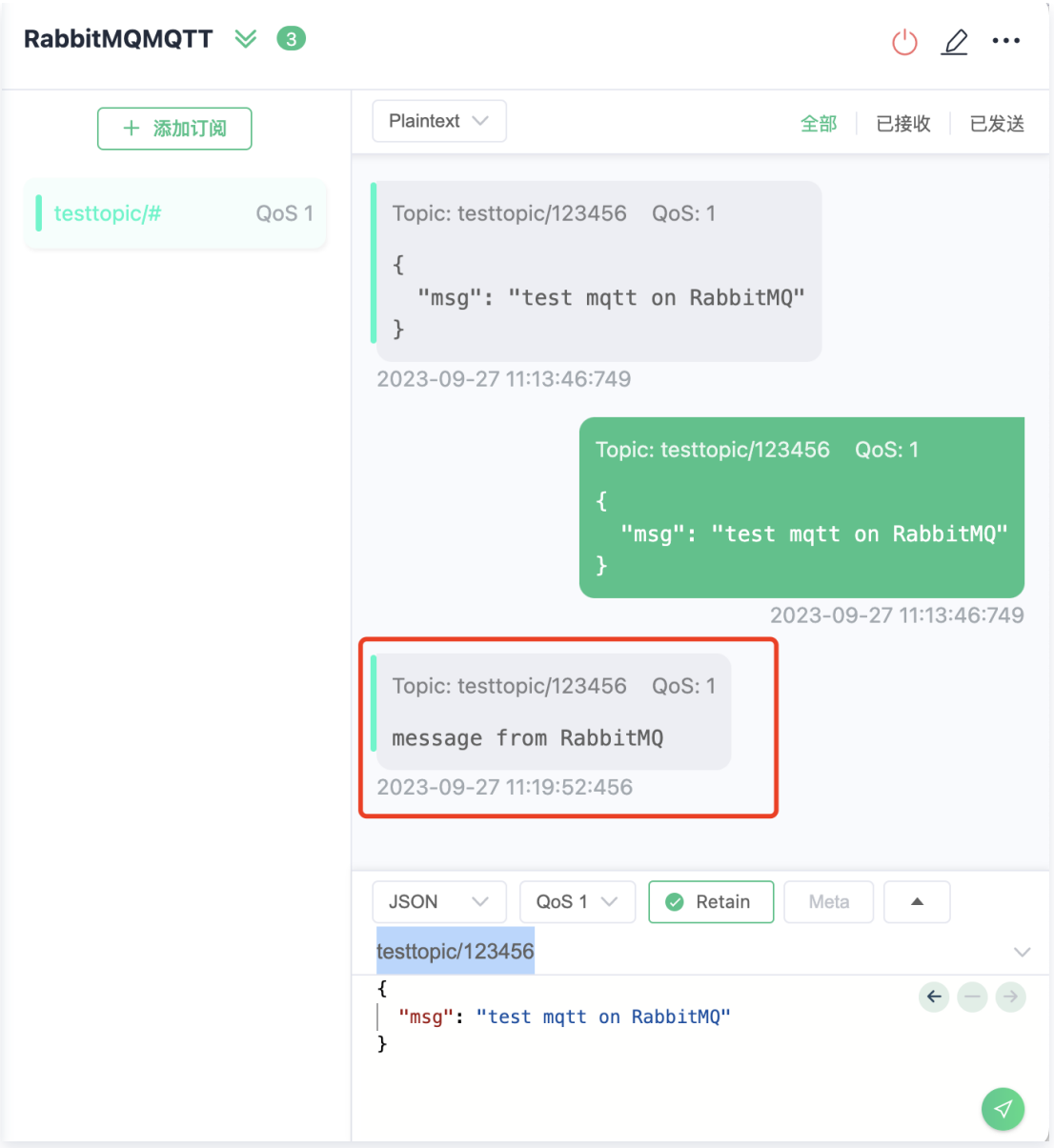

6. Verify the interoperability of MQTT upstream messages and RabbitMQ messages. MQTT messages can be routed to a normal queue for consumption by downstream RabbitMQ applications.






7. Verify the interoperability of RabbitMQ and MQTT downstream messages. RabbitMQ can send messages, and MQTT can subscribe to them.

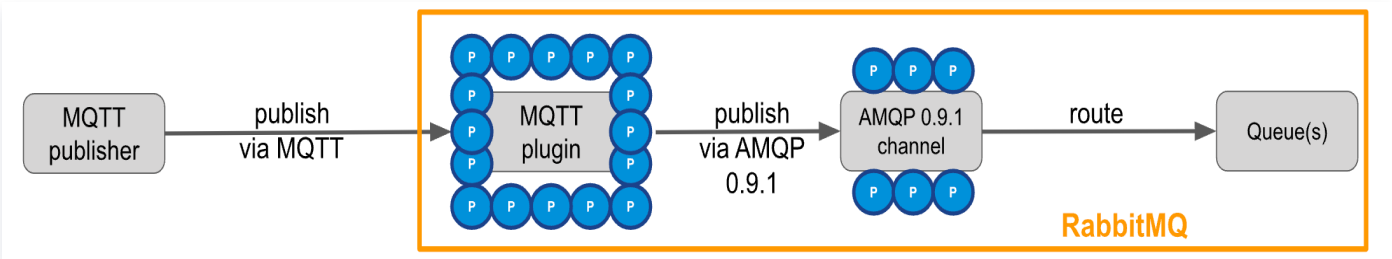




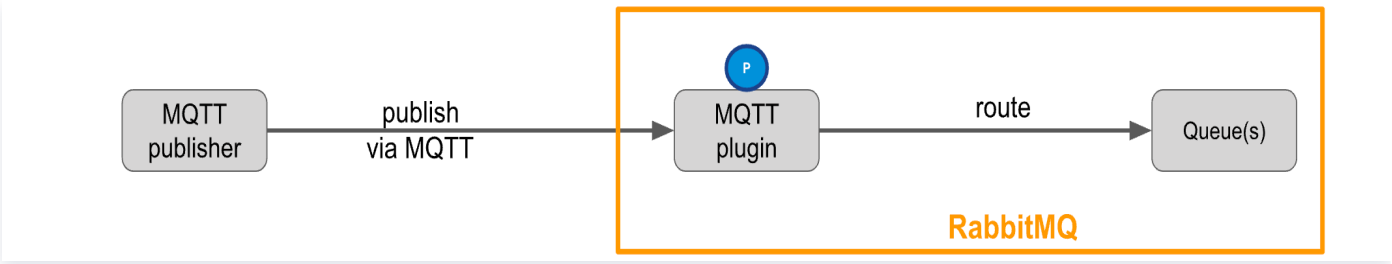

8. Verification summary: The above verification confirms that the RabbitMQ MQTT plugin supports normal MQTT message sending and receiving, upstream messaging to applications, downstream messaging from applications to subscriptions, and comprehensive monitoring.

## How It Works

- The implementation principle before version 3.11 was to convert MQTT messages to AMQP protocol for message sending and receiving.



- The implementation principle after version 3.12 no longer involves AMQP protocol conversion, theoretically offering better performance.




## Notes

- We recommend using stable versions like 3.8.x or 3.11.x, which have no known bugs.
  - Since version 3.12.x is newly released with a reimplemented initial version of MQTT, our validation found issues such as inaccurate monitoring and occasional RabbitMQ interoperability problems. Therefore, **we do not recommend using version 3.12 in production environments**.
- Currently, mainstream MQTT v3 and v3.1 versions are supported, but not v5. RabbitMQ is expected to support v5 in version 3.13.
- The MQTT protocol uses "/" to separate topics, while the AMQP protocol uses "." to separate topics (Routingkey). Automatic conversion happens during protocol translation, so applications should be aware of this difference.
- It is not recommended to use anonymous connections or "no login credentials" for MQTT because the AMQP protocol will automatically convert to the default user guest or mqtt.default_user, which complicates permission management.
- Regarding subscription persistence, note the mapping of MQTT and AMQP queue persistence.
  - Transient clients that use transient (non-persistent) messages
  - Stateful clients that use durable subscriptions (non-clean sessions, QoS1) should prioritize using image queues. Do not use Quorum Queues features because Quorums require at least three nodes, and the stability of new features is yet to be verified, so it is not recommended for now.
- Prioritize using image queues. Do not use Quorum Queues features because Quorums require at least three nodes, and the stability of new features is yet to be verified, so it is not recommended for now.