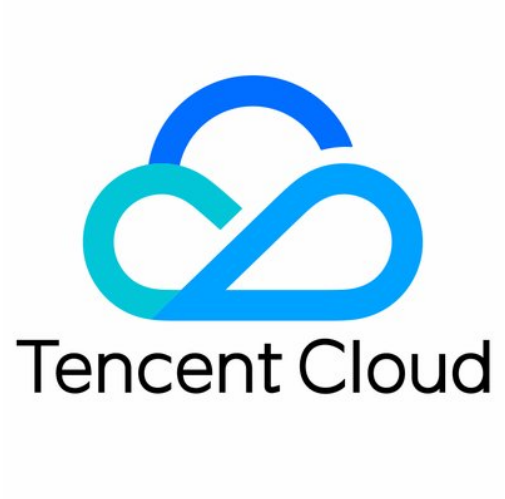
# TDMQ for RabbitMQ

# Practical Tutorial

# Product Documentation



# Contents

# Practical Tutorial
# RabbitMQ Client Practice Tutorial

Last updated：2024-09-10 14:36:41

## Avoid Creating a Connection/Channel for Every Message Sent

Creating a connection in RabbitMQ is a time and resource-intensive operation, with each connection consuming at least 100 KB of memory. Too many connections can put significant memory pressure on the Broker. It is recommended to create a connection when the program starts and reuse this persistent connection for each message sent, improving sending performance and reducing memory usage on the server side.
Channels are a more lightweight communication method. It is recommended to maximize the use of channels to reuse connections. However, using the same channel concurrently across threads is not recommended, as many RabbitMQ clients do not implement thread-safe channels.

## Setting a Reasonable Send Timeout Period for the Producer

Different RabbitMQ clients across various languages and versions have different default send timeout settings, with some clients having excessively long default timeout periods, such as 580 seconds or 900 seconds. During network exceptions, excessively long send timeout periods can block the sender thread and even cause an avalanche effect. It is recommended to set a reasonable timeout period according to the business scenario, with 3 seconds being a recommended value.

## Using Separate Connections for Producers and Consumers

Due to RabbitMQ's unique flow control mechanism, if the producer and consumer reuse the same physical connection and the consumption traffic is high enough to trigger flow control, the producer may be throttled, causing slow sending or timeout. Therefore, it is recommended that producers and consumers use separate physical connections during initialization to avoid mutual interference.

## Avoiding Automatic Message Acknowledgment for Consumers

The RabbitMQ server provides semantics delivered at least once to ensure that messages are correctly delivered to the downstream business system. Once the consumer enables automatic acknowledgment, the server will

automatically confirm and delete the message after it is pushed to the consumer. This occurs even when the consumer encounters an exception during processing, which prevents retries and may lead to message loss in the business logic.

## Implementing Idempotent Processing of Messages for Consumers

The RabbitMQ server provides semantics delivered at least once, which may result in duplicate message delivery in extreme scenarios. Therefore, it is recommended to implement idempotent processing for critical business messages to avoid negative business impacts from duplicated messages.
The business idempotent processing can be implemented by including a unique business identifier in the message, with the consumer checking this identifier and the message status during consumption. This ensures that duplicate messages do not negatively affect the business.

## Limiting the Queue Length to Avoid Message Backlog

Excessively long queues (with a large number of message heap) consume a significant amount of memory and system resources. This not only extends the time required for status synchronization during runtime but also greatly increases the Broker's startup recovery time.
Shorter queues provide faster processing speed and better system performance.
Therefore, clients need to maximize consumption ability and improve queue dimension limits, such as setting the `max-length` , to keep queues as short as possible.

## Using `Consume` or `Get` for Message Consumption

Get is a polling-based pull mode consumption method. Each time a message is consumed, a request is sent to the Broker. If there are no messages in the queue, it may lead to many unnecessary empty pulls, causing resource occupancy. In contrast, Consume allows receiving a batch of messages at once, with the server pushing messages based on current conditions. In most cases, Consume should be used instead of Get to consume messages.
If your business logic requires to use Get to consume messages, pay attention to the business-level Get mechanism to avoid continuous Get Empty (where the queue is empty but the consumer keeps making Get requests), which can lead to abnormally high CPU load on the server.

## Setting a Reasonable Prefetch Count for Consumers

Prefetch settings allow the consumer to preemptively fetch messages into the consumer cache to improve consumption throughput, reducing wait time and delay. However, if the Prefetch Count is too large or unlimited, it can lead to a large number of messages being cached in the consumer, and the server Broker also has to maintain the status of unacknowledged messages in memory, consuming substantial resources. If messages remain in the unacknowledged status for too long, they cannot be consumed by other idle consumers, increasing consumption delay or causing a load imbalance.
It is recommended to set the Prefetch Count within a reasonable range based on the business consumption rate.

## Setting a Reasonable Exception Processing Policy for Consumers

When consumers encounter exceptions while consuming messages that they cannot process, and automatic acknowledgment is not enabled, message retries will be triggered. If the exceptions cannot be processed properly, it will result in infinite message retries, causing high load on the Broker and preventing subsequent messages from being consumed appropriately.

## Client Reconnection Mechanism Confirmation

In extreme scenarios such as OOM or container host failures, the server Broker will heal itself and restart. Routine business operations, such as cluster configuration upgrades, can also trigger Broker restarts. To avoid continuous connection exceptions during Broker restarts, ensure that the client implements an automatic reconnection mechanism.

## Avoiding Disabling Heartbeat Settings for Client SDKs

The heartbeat have a configuration value on both the server side and client side (with the server set to 60 seconds). The effective heartbeat is negotiated between the server and client, and the negotiation mechanism varies by client language/version. Setting the client heartbeat to 0 disables heartbeat detection, causing the server to be unable to automatically remove long-term inactive connections. This can lead to unexpected connection leaks.

# RabbitMQ Message Reliability Practical Tutorial

Last updated：2024-09-10 14:37:18

## Message Persistence

To ensure the queue metadata and messages in the queue are not lost after the Broker restarts, it is recommended to set the queue to durable and the messages to persistent. Then the queue will immediately persist the messages to disk upon receiving them.
Non-persistent messages will also occupy more server memory resources, which can, in extreme cases, lead to high memory load on the server.

## Sender Confirm

The Confirm mechanism ensures messages are successfully sent to the Broker. However, if the mandatory is not set when a message is sent, the Broker will respond with confirm to the sender regardless of whether the message is successfully routed to the target queue. If the mandatory is set (note that the delay exchange does not support setting mandatory), and the message cannot be routed, the Broker will return the message to the client. The client can detect these unroutable messages by implementing **basic.return**; only when the message can be successfully routed to the target queue will the Broker respond with confirm to the sender.

## Consumer Acknowledgement

The ACK mechanism on the consumer side ensures that messages are received by the client and provides an at least once consumption semantics guarantee, ensuring the message can only be deleted after being correctly processed. However, this also requires the client to implement idempotency to avoid errors caused by duplicate message consumption. Additionally, unacknowledged messages will accumulate in memory, increasing memory usage on both the client and server.

## Enabling Image Queues

Image queues ensure high availability by replicating queue data to other Brokers in the cluster. Configuring image queue policies may increase the Broker's startup duration and resource usage, but it can ensure that the queue

remains available in the event of a single Broker failure, minimizing message loss.
When image queue policies are configured, avoid setting ha-sync-mode=automatic, as this configuration will cause the server Broker to automatically perform a full synchronization of the queue data after a restart regardless of whether the data has already been synchronized. If the synchronized queue has too much data heap, it will result in prolonged Broker synchronization time, continuous memory usage, and other issues. Additionally, the queue will be unavailable until the synchronization is complete, severely affecting both business availability and server stability.

# Usage Instructions for MQTT Protocol Supported by RabbitMQ

Last updated：2024-09-10 14:38:07

## Overview

MQTT is a widely used protocol for Internet of Things (IoT) applications. RabbitMQ, a popular open-source message queue product based on the AMQP 0.9.1 protocol, supports MQTT through plugins. This allows RabbitMQ clusters to easily support the MQTT protocol, facilitating its use in IoT and other business scenarios.
Community reference documentation:
1. The MQTT protocol supported in RabbitMQ versions earlier than 3.11: MQTT Plugin — RabbitMQ
2. The Native MQTT protocol supported in RabbitMQ 3.12: Serving Millions of Clients with Native MQTT | RabbitMQ - Blog

## Directions

### Step 1: Purchasing or Self-Building a RabbitMQ Cluster

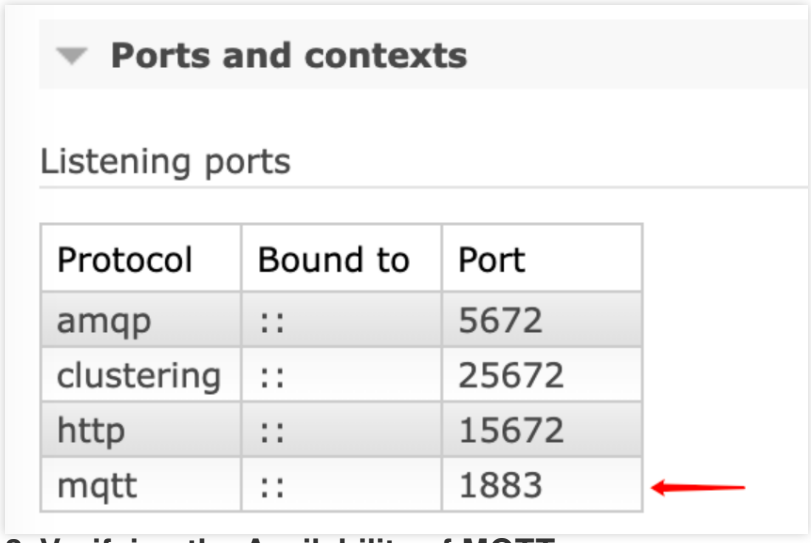
Directly purchase a RabbitMQ cluster in the cloud. Buy Now.
Or build your own RabbitMQ cluster. For detailed instructions, see Downloading and Installing RabbitMQ — RabbitMQ.

### Step 2: Enabling the MQTT Plugin

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

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

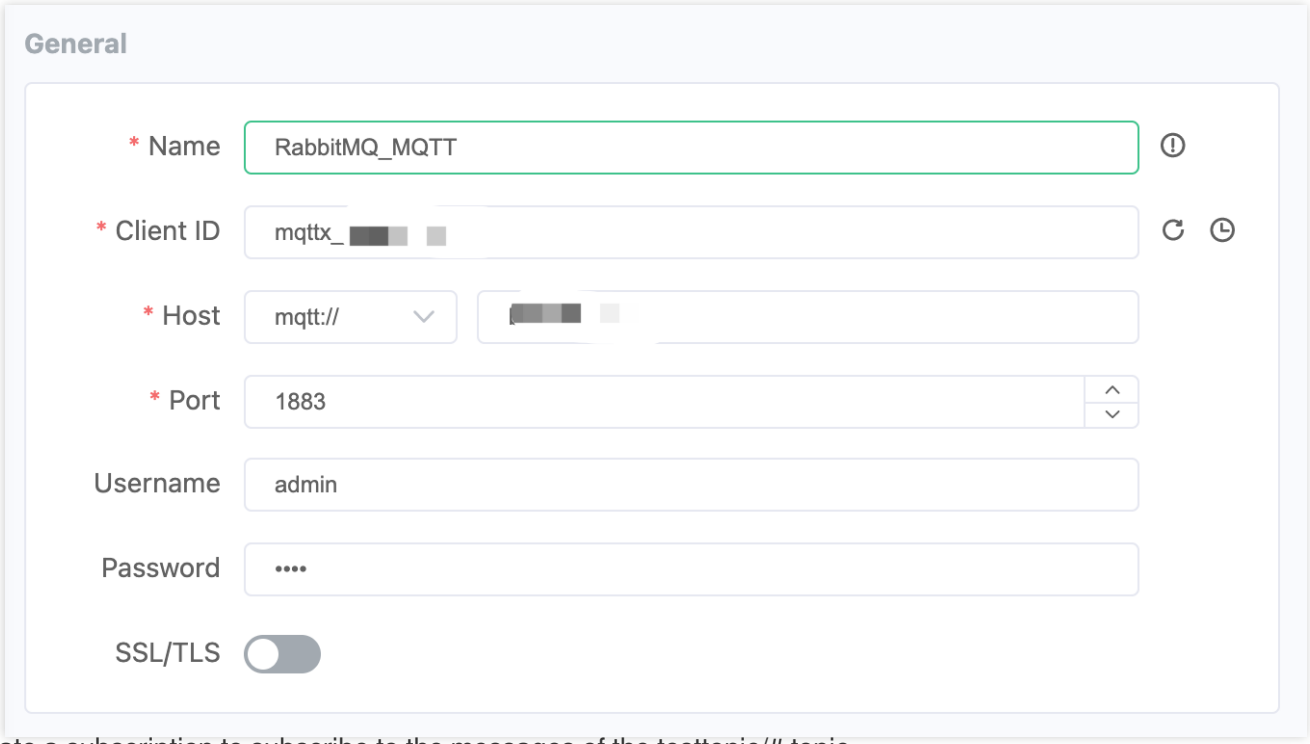
The RabbitMQ plugin management feature on Tencent Cloud is under development. Currently, you can enable the MQTT plugin and network connectivity by submitting a ticket.
After enabling the MQTT plugin, you can see the newly added port 1883 in the console:



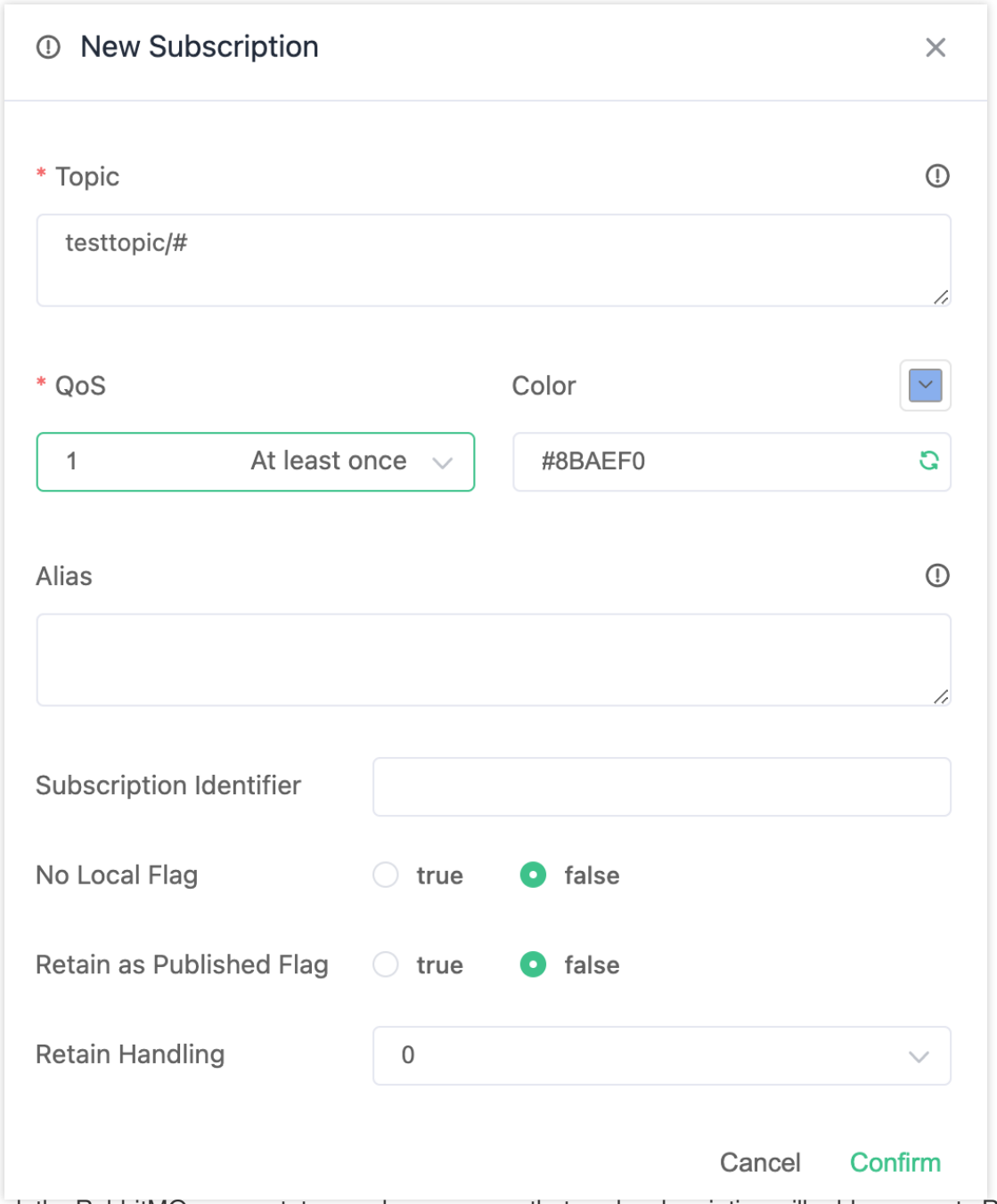

### Step 3: Verifying the Availability of MQTT

Download the commonly used mqttx (MQTTX: A Full-Featured MQTT Client Tool) for verification.

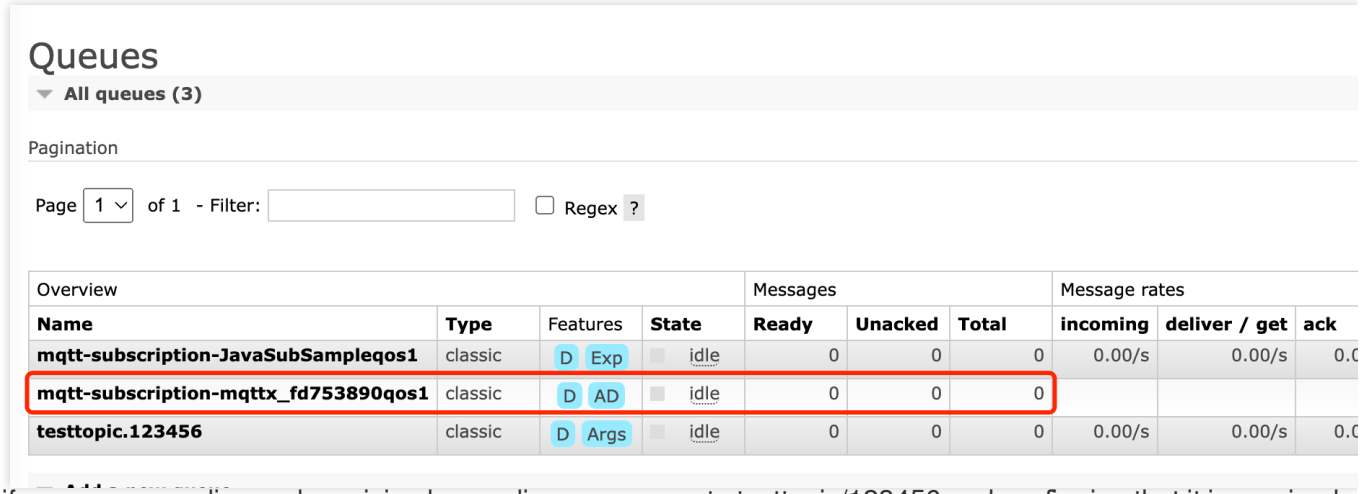
1. Create a connection and fill in the address and port. Use the RabbitMQ username and password for the username and password fields.



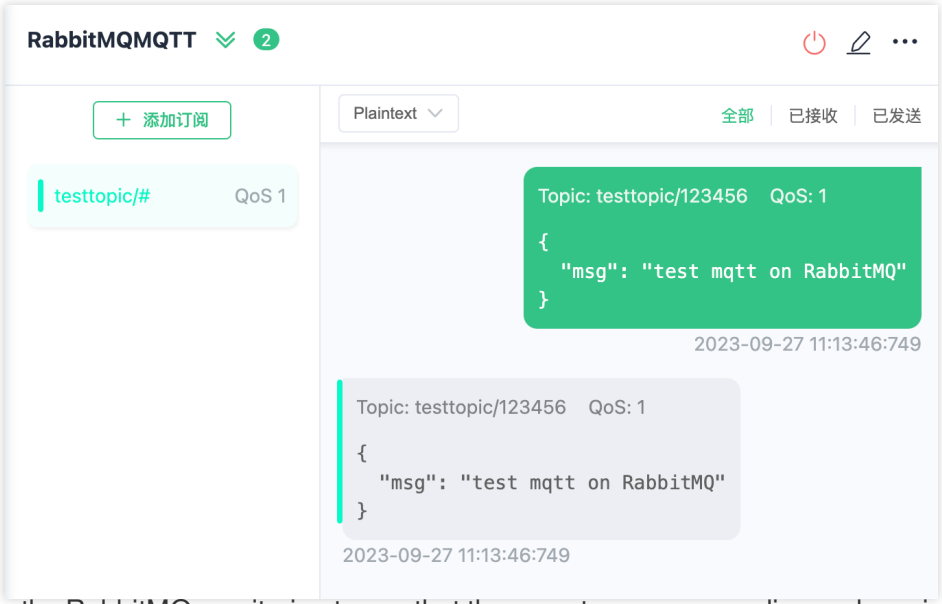

2. Create a subscription to subscribe to the messages of the testtopic/# topic.



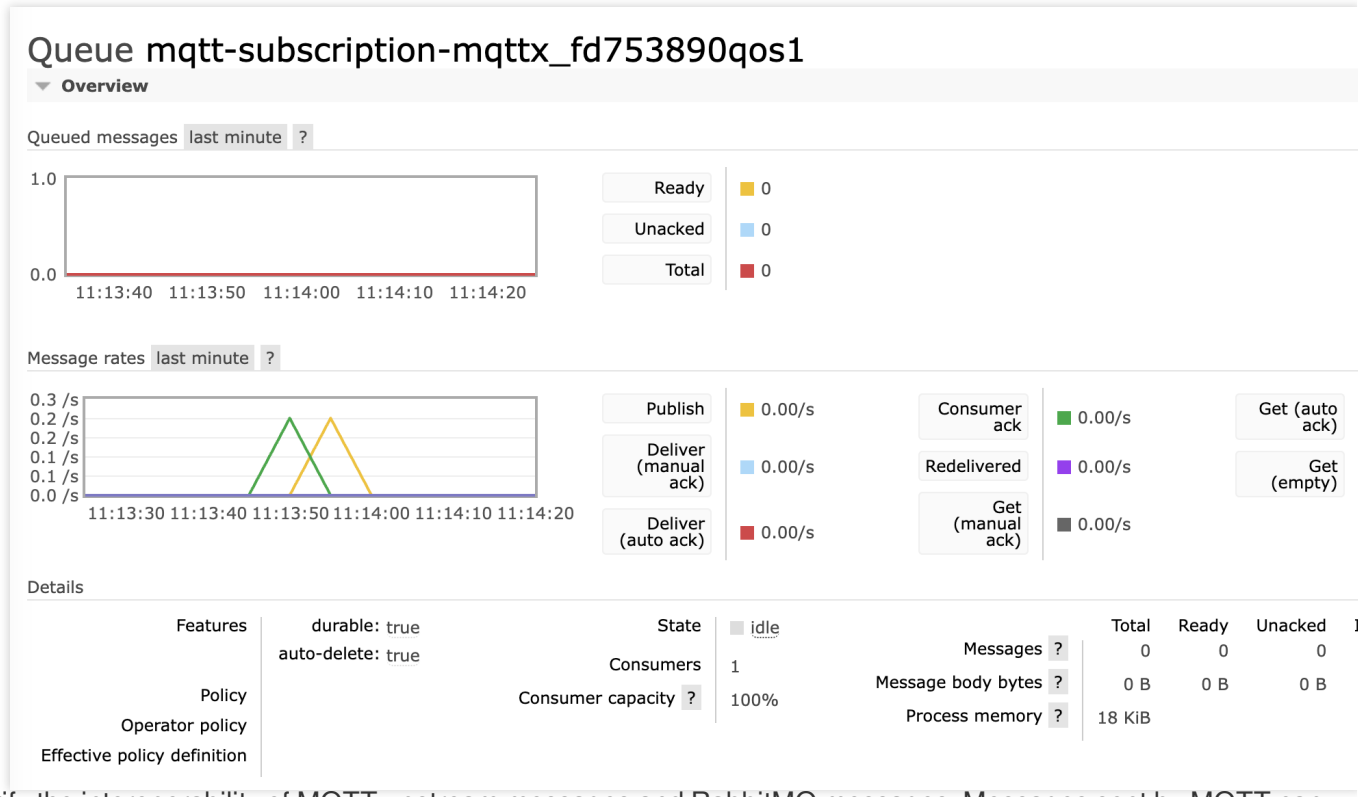

3. Check the RabbitMQ queue status, and you can see that each subscription will add a queue to RabbitMQ.



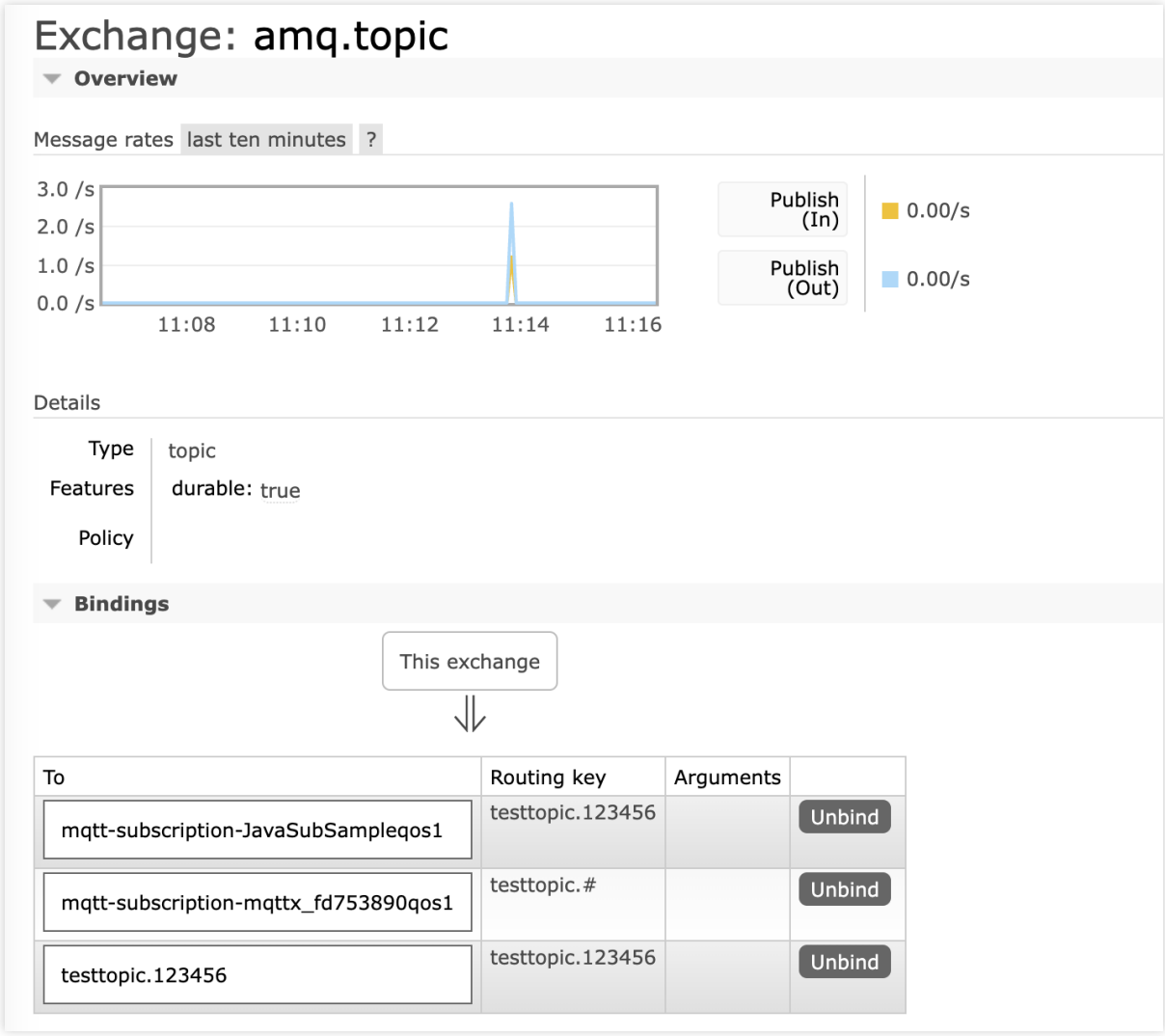

4. Verify message sending and receiving by sending a message to testtopic/123456 and confirming that it is received through the subscription.

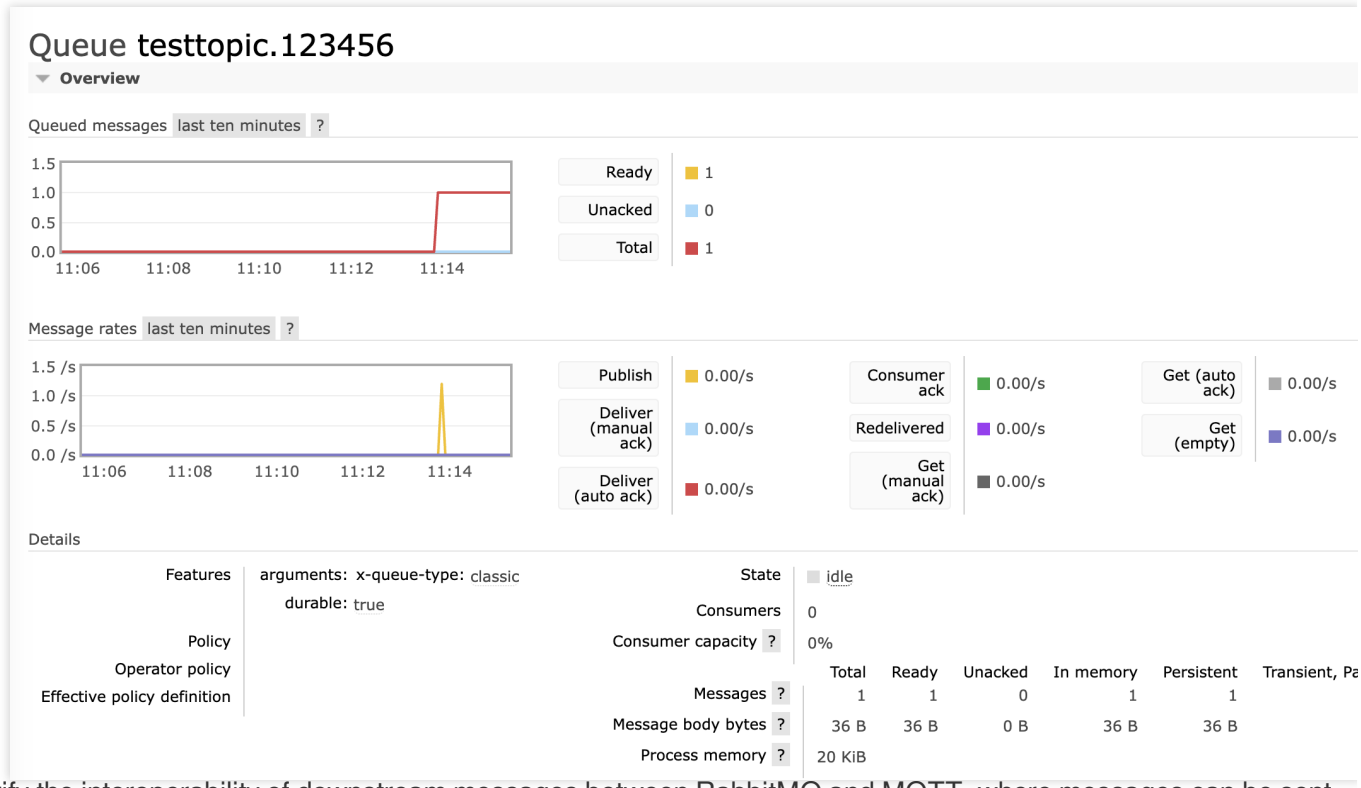



5. View the RabbitMQ monitoring to see that the recent message sending and receiving activity for the queue has been recorded.

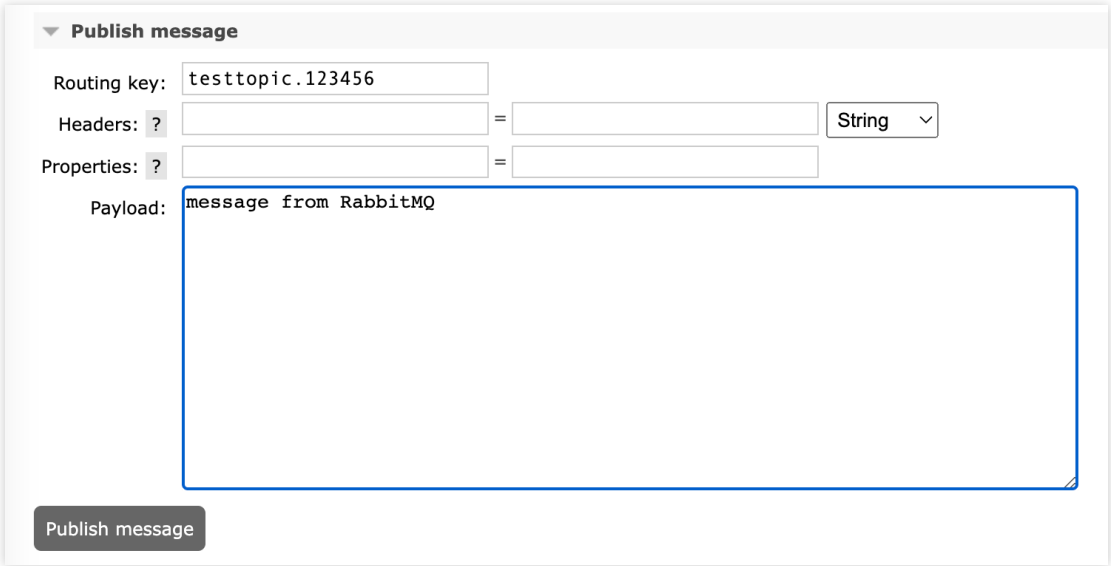


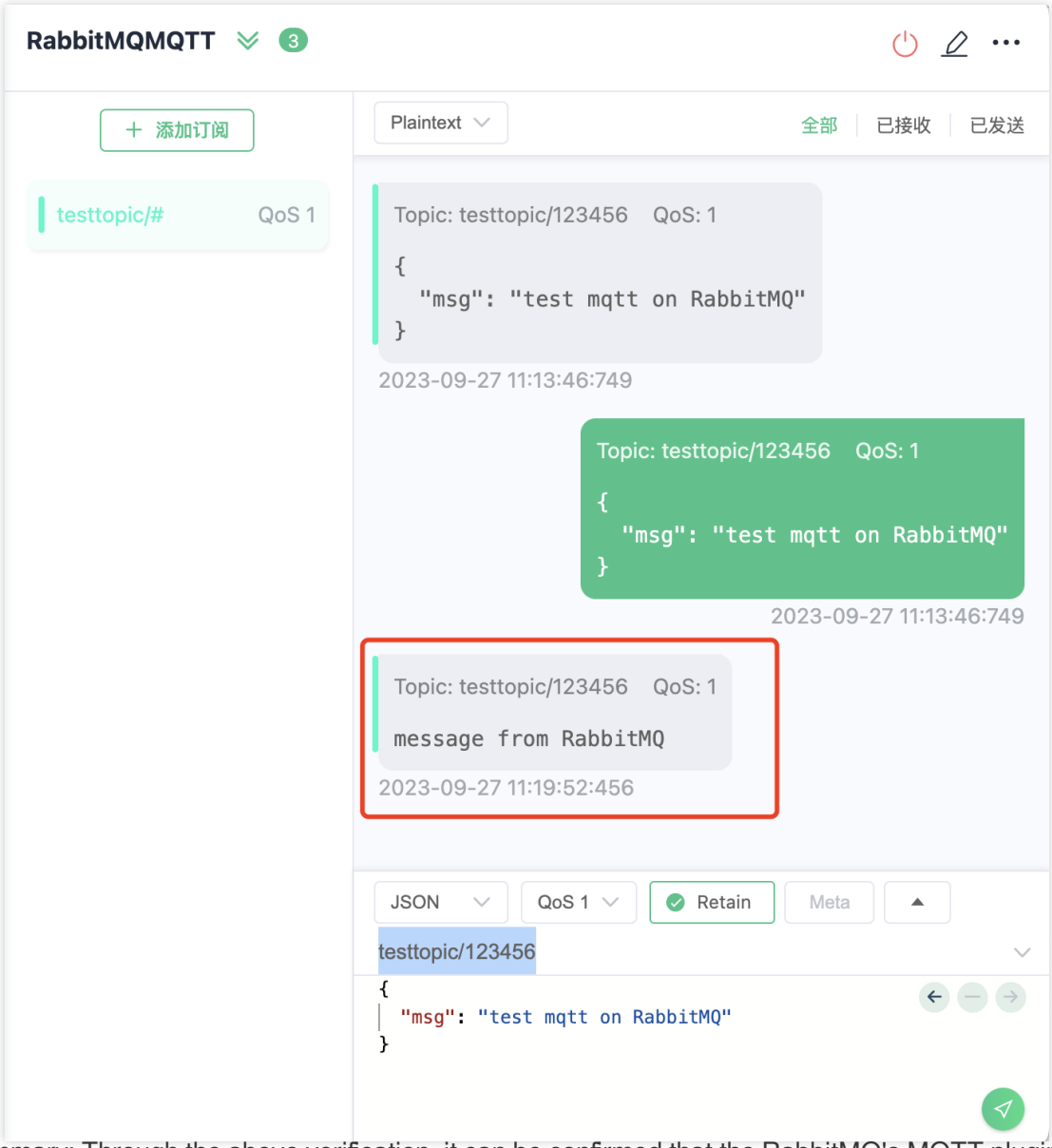

6. Verify the interoperability of MQTT upstream messages and RabbitMQ messages. Messages sent by MQTT can also be routed to a standard queue and consumed by RabbitMQ downstream applications.






7. Verify the interoperability of downstream messages between RabbitMQ and MQTT, where messages can be sent from RabbitMQ and subscribed to by MQTT.

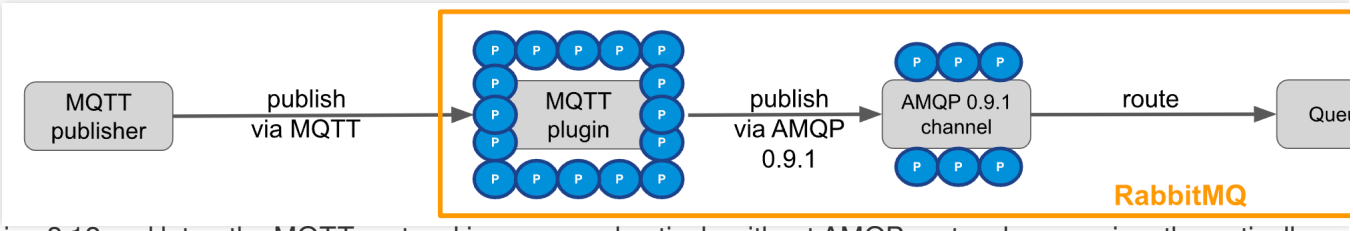




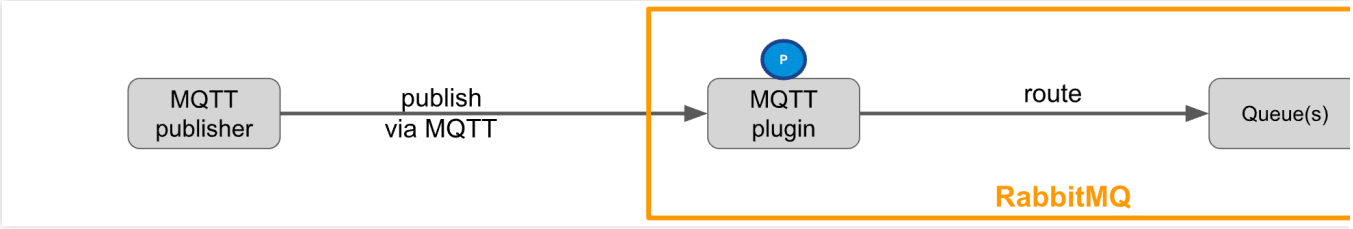

8. Summary: Through the above verification, it can be confirmed that the RabbitMQ's MQTT plugin supports normal MQTT message sending and receiving, processes MQTT upstream messages to applications, allows applications to send MQTT downstream messages to subscriptions, and provides comprehensive monitoring.

## How It Works

In versions earlier than 3.11, MQTT messages are converted to AMQP protocol to achieve MQTT message sending and receiving.




In version 3.12 and later, the MQTT protocol is processed natively without AMQP protocol conversion, theoretically achieving better performance.




# Notes

It is recommended to use stable versions such as 3.8.x or 3.11.x, which are free of known bugs.

Version 3.12.x is a newly released version. MQTT found that monitoring was not stable during the verification process for 3.12.x. Therefore, **it is not recommended to use version 3.12 in production environments**.

Currently, the mainstream MQTT v3 and v3.1 versions are supported, but v5 is not yet supported. RabbitMQ Expected to Support MQTT in Version 3.13 and Later.

The MQTT protocol uses / to separate topics, while the AMQP protocol uses . to separate topics (Routingkey). This difference is automatically processed during protocol conversion, but be aware of this when using the application.

Anonymous connections or no login credentials are not recommended for MQTT, as the AMQP protocol automatically converts them to the default user, either guest or mqtt.default_user, making permission control difficult.

Regarding subscription persistence, pay attention to 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 prefer using mirrored queues instead of Quorum Queues, as Quorum requires at least three nodes, and the stability of the new feature is yet to be verified. It is currently not recommended for use.

It is recommend to use image queues instead of Quorum Queues, as Quorum requires at least three nodes, and the stability of the new feature is yet to be verified. It is currently not recommended for use.

# Other Notes

Last updated：2024-09-10 14:38:52

## Delayed Messages Implemented with rabbitmq_delayed_message_exchange Plugin

1. The current design of the plugin is not suitable for scenarios involving a large number of delayed messages (hundreds of thousands or millions of unscheduled messages). Carefully assess the message throughput in the production environment to avoid unexpected long delays, message loss, and other issues.
2. The delayed messages have only one persistent replica on each node. If a node fails to operate normally (for example, the message heap causes continuous OOM errors, leading to restarts and recovery failure), the delayed messages on that node cannot be consumed by the consumer side.
3. The delay exchange does not support the **mandatory** option, so producers cannot be notified through the **basic.return** event about messages that could not be routed. Therefore, ensure the corresponding exchange, queue, and routing relationship exist before sending delayed messages.
In summary, we strongly recommend against using this plugin and suggest using dead-letter queues to indirectly implement Delayed Message instead. If you still choose to use this plugin despite understanding its various drawbacks, it is strongly recommended to keep the number of delayed messages as low as possible to avoid triggering high memory load issues.

## Network Partition

1. Network partition is an inevitable issue when using RabbitMQ. The network partition can lead to inconsistencies in the cluster status, and even after the network is recovered, RabbitMQ still requires a Broker restart to resynchronize the status. TDMQ for RabbitMQ currently uses the autoheal mode, which will automatically select a winning partition and then restart the Broker in the untrusted partition.
2. Clients are recommended to take the following measures to minimize the negative impact of network partition:
Message senders need to consider using the **mandatory** mechanism when sending messages and having the ability to process **basic.return** to promptly address message routing failures that may occur during processing networks partitions.
Message consumers need to implement idempotency at the consumer end since there may be duplicate messages during partitioning networks and processing network partitions.

## Alarm Configuration

Tencent Cloud provides monitoring metrics for various dimensions such as clusters and nodes. For details, see Monitoring and Alarms.
It is strongly recommended to focus on key metrics such as CPU, memory, disk utilization, and message heap on each node, and to configure alarms to avoid continuous high server load that can affect the cluster stability.

## Message Track Usage Limit

An overview of the implementation principle of message query: After the Trace plugin for a VHost is enabled in the Tencent Cloud console, the server-side components will consume the track messages of the corresponding RabbitMQ cluster. After a series of processes, the message track query feature can be enabled in the console.
Based on the principle described above, the message track relies on the service components to consume track messages. Since the service components are underlying public services, they cannot ensure timely consumption of track messages for high-traffic RabbitMQ clusters. If track messages heap occurs, it may cause high memory load within the cluster, affecting the stability of the RabbitMQ cluster.
Therefore, it is not recommended to enable the Trace plugin in the production environment, especially in scenarios where the overall cluster (including all VHosts) sending TPS exceeds 1,000. The Trace plugin is recommended to be used in low-traffic verification or troubleshooting scenarios instead of production environments.