Release: R23-11

Mastering MQTT

Your Ultimate Tutorial for MQTT

Introduction

According to the research report “Status of the IoT Spring 2022” from IoT Analytics, the IoT market is expected to grow 18% and reach 14.4 billion active connections by 2022.

With such large-scale IoT demand, massive device access and device management pose huge challenges to network bandwidth, communication protocols, and platform service architecture. After many years of development, the MQTT protocol has become the preferred protocol for the IoT industry with its advantages of lightweight, efficiency, reliable messaging, massive connection support, and secure bidirectional communication.

Whether you're a seasoned developer seeking to sharpen your IoT skills or a novice looking to delve into the heart of IoT communication, this eBook is your gateway to mastering MQTT and harnessing its transformative power. We will embark on a journey to decode the intricacies of MQTT, from its core principles to advanced techniques, equipping you with the knowledge to harness its capabilities effectively.

Let's dive in.

Table of Contents

What is MQTT Protocol?

MQTT is a lightweight messaging protocol based on publish/subscribe model, specifically designed for IoT applications in low bandwidth and unstable network environments. It can provide real-time reliable messaging services for network-connected devices with minimal code. MQTT protocol is widely used in IoT, Mobile Internet, Smart Hardware, Internet of Vehicles, Smart Cities, Telemedicine, Power, Oil, Energy, and other fields.

MQTT was created by Andy Stanford-Clark of IBM, and Arlen Nipper (then of Arcom Systems, later CTO of Eurotech). According to Nipper, MQTT must have the following features:

- Simple and easy to implement
- QoS support (complex device network environment)
- Lightweight and bandwidth-saving (because bandwidth was expensive back then)
- Data irrelevant (Payload data format does not matter)
- Continuous session awareness (always know whether the device is online)

According to Arlen Nipper on IBM Podcast, MQTT was originally named MQ TT. Note the space between MQ and TT. The full name is MQ Telemetry Transport. It is a real-time data transmission protocol that he developed while working on a crude oil pipeline SCADA system for Conoco Phillips in the early 1990s. Its purpose was to allow sensors to communicate with IBM's MQ Integrator via VSAT, which has limited bandwidth. The name MQ TT was chosen in accordance with industry practice because Nipper is a remote sensing and data acquisition and monitoring professional.

Unique Capabilities of MQTT for IoT

Lightweight and Efficient

MQTT minimizes the extra consumption occupied by the protocol itself, and the minimum message header only needs to occupy 2 bytes. It can run stably in bandwidth-constrained network environments. At the same time, MQTT clients need very small hardware resources and can run on a variety of resource-constrained edge devices.

Reliable Message Delivery

The MQTT protocol provides 3 levels of Quality of Service for messaging, ensuring reliable messaging in different network environments.

- **QoS 0**: The message is transmitted at most once. If the client is not available at that time, the message is lost. After a publisher sends a message, it no longer cares whether it is sent to the other side or not, and no retransmission mechanism is set up.
- **QoS 1**: The message is transmitted at least once. It contains a simple retransmission mechanism: the publisher sends a message, then waits for an ACK from the receiver, and resends the message if the ACK is not received. This model guarantees that the message will arrive at least once, but it does not guarantee that the message will be repeated.
- **QoS 2**: The message is transmitted only once. A retransmission and duplicate message discovery mechanism is designed to ensure that messages reach the other side and arrive strictly only once.

More about MQTT QoS can be found in the blog: Introduction to MQTT QoS.

In addition to QoS, MQTT provides a mechanism of Clean Session. For clients that want to receive messages that were missed during the offline period after reconnecting, you can set the Clean Session to false at connection time. At this time, the server will store the subscription relationship and offline messages for the client and send them to the client when the client is online again.

Connect IoT Devices at Massive Scale

Since its birth, MQTT protocol has taken into account the growing mass of IoT devices. Thanks to its excellent design, MQTT-based IoT applications and services can easily have the capabilities of high concurrency, high throughput, and high scalability.

The support of MQTT broker is indispensable to the connection of massive IoT devices. Currently, the MQTT broker that supports the largest number of concurrent connections is EMQX. The recently released EMQX 5.0 achieved 100 million MQTT connections + 1 million per second messages through a 23-node cluster, making itself the most scalable MQTT broker in the world to date.

Secure Bi-Directional Communication

Relying on the publish-subscribe model, MQTT allows bidirectional messaging between devices and the cloud. The advantage of the publish-subscribe model is that publishers and subscribers do not need to establish a direct connection or be online at the same time. Instead, the message server is responsible for routing and distributing all messages.

Security is the cornerstone of all IoT applications. MQTT supports secure bidirectional communication via TLS/SSL, while the client ID, username and password provided in the MQTT protocol allow users to implement authentication and authorization at the application layer.

Keep Alive and Stateful Sessions

To cope with network instability, MQTT provides a Keep Alive mechanism. In the event of a long period of no message interaction between the client and the server, Keep Alive keeps the connection from being disconnected. If the connection is disconnected, the client can instantly sense it and reconnect immediately.

At the same time, MQTT is designed with Will Message which allows the server to help the client post a will message to a specified MQTT topic if the client is found to be offline abnormally.

In addition, some MQTT brokers, such as EMQX, also provide online and offline event notifications. When the backend service subscribes to a specific topic, it can receive all the clients' online and offline events, which helps the backend service unify the processing of the client's online and offline events.

Understanding Key MQTT Components

MQTT Broker

The MQTT broker is responsible for receiving client-initiated connections and forwarding messages sent by the client to some other eligible clients. A mature MQTT broker can support massive connections and millions of messages throughput, helping IoT business providers focus on business functionality and quickly create a reliable MQTT application.

In the upcoming tutorials of this eBook, we will use EMQX as the MQTT broker to showcase the functionalities of MQTT. EMQX is a widely-used large-scale distributed MQTT broker for IoT. Since its open-source release on GitHub in 2013, it has been downloaded more than 10 million times worldwide and the cumulative number of connected IoT key devices exceeds 100 million.

You can install EMQX 5.x open-source version with the following Docker command to experience it.

```
docker run -d --name emqx -p 1883:1883 -p 8083:8083 -p 8084:8084 -p
8883:8883 -p 18083:18083 emqx/emqx:latest
```

You can also create fully hosted MQTT services directly on EMQX Cloud. Free trial of EMQX Cloud is available, with no credit card required.

MQTT Client

MQTT applications usually need to implement MQTT communication based on MQTT client libraries. At present, basically all programming languages have matured open-source MQTT client libraries. So, you can refer to the Comprehensive list of MQTT client libraries collated by EMQ to choose a suitable client library to build an MQTT client that meets their business needs. You can also visit the MQTT Programming blog series provided by EMQ to learn how to use MQTT in Java, Python, PHP, Node.js and other programming languages.

MQTT application development is also inseparable from the support of the MQTT testing tool. An easy-to-use and powerful MQTT testing tool can help developers shorten the development cycle and create a stable IoT application.

MQTTX will be utilized in this eBook as the MQTT client. MQTTX is an open-source cross-platform desktop client. It is easy to use and provides comprehensive MQTT 5.0 functionality, feature testing, and runs on macOS, Linux and Windows. It also provides command line and browser versions to meet MQTT testing needs in different scenarios. You can visit the MQTTX website to download and try it out: https://mqttx.app/.

MQTT Publish-Subscribe Pattern

The Publish-subscribe pattern is a messaging pattern that decouples the clients that send messages (publishers) from the clients that receive messages (subscribers) by allowing them to communicate without having direct connections or knowledge of each other's existence.

The essence of MQTT's Publish-Subscribe pattern is that a middleman role called a Broker is responsible for routing and distributing all messages. Publishers send messages with topics to the Broker, and subscribers subscribe to topics from the Broker to receive messages of interest.

In MQTT, topics and subscriptions cannot be pre-registered or created. As a result, the broker cannot predict how many subscribers will be interested in a particular topic. When a publisher sends a message, the broker will only forward it to the subscribers that are currently subscribed to the topic. **If there are no current subscribers for the topic, the message will be discarded**.

The MQTT Publish-Subscribe pattern has four main components: Publisher, Subscriber, Broker, and Topic.

- **Publisher**

  The publisher is responsible for publishing messages to a topic. It can only send data to one topic at a time and does not need to be concerned about whether the subscribers are online when publishing a message.

- **Subscriber**

  The subscriber receives messages by subscribing to a topic and can subscribe to multiple topics at once. MQTT also supports load-balancing subscriptions among multiple subscribers through shared subscriptions.

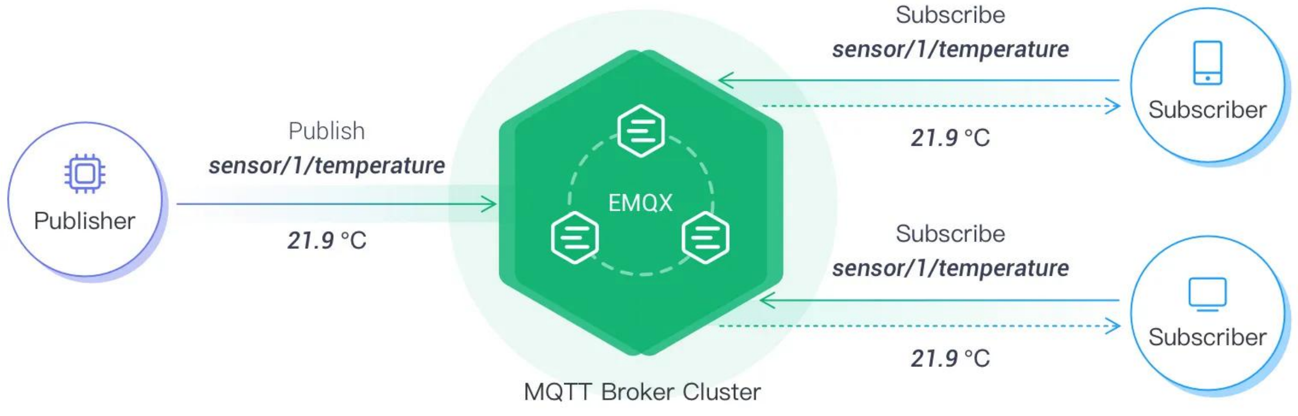
- **Broker**

  The broker is responsible for receiving messages from publishers and forwarding them to the appropriate subscribers. In addition, the broker also handles requests from clients for connecting, disconnecting, subscribing, and unsubscribing.

- **Topic**

  MQTT routes messages based on topics. A topic is typically leveled and separated with a slash / between the levels, this is similar to URL paths. For example, a topic could be sensor/1/temperature. Multiple subscribers can subscribe for the same topic, and the broker will forward all messages on that topic to these subscribers. Multiple publishers can also send messages to the same topic, and the broker will route these messages in the order they are received to the subscribed clients.

  In MQTT, subscribers can subscribe to multiple topics simultaneously using topic wildcards. This allows them to receive messages on multiple topics with a single subscription. Check out the blog Understanding MQTT Topics & Wildcards by Case for more details.



MQTT Publish-subscribe Architecture

In the MQTT publish-subscribe pattern, a client can function as a publisher, a subscriber, or both. When a client publishes a message, it sends it to the broker, which then routes the message to all subscribed clients on that topic. If a client subscribes to a topic, it will receive all the messages that the broker forwards for that topic.

There are generally two common approaches for filtering and routing messages in publish-subscribe systems.

- By topics

  Subscribers can subscribe with the broker for topics that are of interest to them. When a publisher sends a message, it includes the topic to which the message belongs. The broker uses this information to determine which subscribers should receive the message and routes it to the appropriate subscribers.

- By content-based filtering

  Subscribers can specify the conditions that a message must meet to be delivered to them. If the attributes or content of a message matches the conditions defined by the subscriber, the message will be delivered to that subscriber. If the message does not meet the subscriber's conditions, it will not be delivered.

In addition to routing messages based on topics, EMQX provides advanced message routing capabilities through its SQL-based Rule Engine starting with version 3.1. This feature allows for the routing of messages based on the content of the message. For more information about the Rule Engine and how it works, you can refer to the EMQX documentation.

Establishing an MQTT Connection

Introduction to MQTT Connection

MQTT connections are initiated from the client to the broker. Any application or device running the MQTT client library is an MQTT client. The MQTT Broker handles client connection, disconnection, subscribe (or unsubscribe) requests, and routes messages up on receiving publish requests.

After establishing a network connection with the broker, the very first message the client must send is a `CONNECT` packet. The broker must reply with a `CONNACK` to the client as a response, and the MQTT connection is established successfully after the client receives the `CONNACK` packet. If the client does not receive a `CONNACK` packet from the broker in time (usually a configurable timeout from the client side), it may actively close the network connection.

MQTT protocol specification does not limit which transport to use. The most commonly adopted transport protocol for MQTT is TCP/TLS and WebSocket. EMQ has also implemented MQTT over QUIC.

MQTT over TCP/TLS

TCP/TLS is widely used and is a connection-oriented, reliable, byte-stream-based transport layer communication protocol. It ensures that the bytes received are the same as those sent through the acknowledgment and retransmission mechanism.

MQTT is usually based on TCP/TLS, which inherits many of the advantages of TCP/TLS and can run stably in low bandwidth, high latency, and resource-constrained environments.

MQTT over WebSocket

With the rapid development of the Web technology, more and more applications can be implemented in the web browser taking advantage of the powerful rendering engine for UI. WebSocket, the native communication method for Web applications, is also widely used.

Many IoT web-based applications such as device monitoring systems need to display device data in real-time in a browser. However, browsers transmit data based on the HTTP protocol and cannot use MQTT over TCP.

The founder of the MQTT protocol foresees the importance of Web applications, so the MQTT protocol supports communication through MQTT over WebSocket since its creation. Check out the blog for more information on how to use MQTT over WebSocket.

MQTT over QUIC

QUIC (RFC 9000) is the underlying transport protocol of the next-generation Internet protocol HTTP/3, which provides connectivity for the modern mobile Internet with less connection overhead and message latency compared to TCP/TLS protocols.

Based on the advantages of QUIC, which make it highly suitable for IoT messaging scenarios, EMQX 5.0 introduces MQTT over QUIC. Check out the blog for more information.

Use of MQTT Connection Parameters

Connection Address

The connection address of MQTT usually includes the IP (or domain name), port, and protocol. In case of clustered MQTT brokers, there is typically a load-balancer put in front, so the IP or domain name might actually be the load-balancer.

TCP-based MQTT connections

For example, mqtt://broker.emqx.io:1883 is a TCP-based MQTT connection address, and `mqtts://broker.emqx.io:1883` is a TLS/SSL based MQTT secure connection address.

> In some client libraries, TCP-based connection is `tcp://ip:1883`

WebSocket-based connection

The connection address also needs to contain the Path when using a WebSocket connection. The default Path configured for EMQX is `/mqtt`.

For example, `ws://broker.emqx.io:8083/mqtt` is a WebSocket-based MQTT connection address, and `wss://broker.emqx.io:8083/mqtt` is a WebSocket-based MQTT secure connection address.

Client ID

The MQTT Broker uses Client ID to identify clients, and each client connecting to the broker must have a unique Client ID. Client ID is a UTF-8 encoded string. If the client connects with a zero-length string, the broker should assign a unique one for it.

Depending on the MQTT protocol version and implementation details of the broker, the valid set of characters accepted by the broker varies. The most conservative scheme is to use characters `[0-9a-zA-Z]` and limit the length to 23 bytes.

Due to the uniqueness nature of the Client ID, if two clients connect to the same broker with the same Client ID, the client connects later will force the one connected earlier to go offline.

Username & Password

The MQTT protocol supports username-password authentication, but if the underlying transport layer is not encrypted, the username and password will be transmitted in plaintext, hence for the best security, `mqtts` or `wss` protocol is recommended.

Most MQTT brokers default to allow anonymous login, meaning there is no need to provide username or password (or set empty strings). So considering the security, it is recommended to enable the appropriate authentication features.

Connect Timeout

The waiting time before receiving the broker `CONNACK` packet, if the `CONNACK` is not received within this time, the connection is closed.

Keep Alive

Keep Alive is an interval in seconds. When there is no message to send, the client will periodically send a heartbeat message to the broker according to the value of Keep Alive to ensure that the broker will not disconnect the connection.

After the connection is established successfully, if the broker does not receive any packets from the client within 1.5 times of Keep Alive, it will consider that there is a problem with the connection with the client, and the broker will disconnect from the client.

Clean Session

Set to false means to create a persistent session. When the client disconnects, the session remains and saves offline messages until the session expires. Set to true to create a new temporary session that is automatically destroyed when the client disconnects.

The persistent session makes it possible for the subscribe client to receive messages while it has gone offline. This feature is very useful in IoT scenarios where the network is unstable.

The number of messages the broker keeps for persistent sessions depends on the broker's settings. For example, the public MQTT broker provided by EMQ is set to keep offline messages for 5 minutes, and the maximum number of messages is 1000 (for QoS 1 and QoS 2 messages).

> **Note**: The premise of persistent session recovery is that the client reconnects with a fixed Client ID. If the Client ID is dynamic, then a new persistent session will be created.

Will Message

When an MQTT client that has set a Will Message goes offline abnormally, the MQTT broker publishes the Will Message set by that client.

> **Unexpected offline includes**: the connection is closed by the server due to network failure; the device is suddenly powered off; the device attempts to perform an unallowable operation and the connection is closed by the server, etc.

The Will Message can be seen as a simplified MQTT message that also contains Topic, Payload, QoS, Retain, etc.

- When the device goes offline unexpectedly, the will message will be sent to the Will Topic.
- The Will Payload is the content of the message to be sent.
- The Will QoS is the same as the QoS of standard MQTT messages. Check out the blog to learn more about MQTT QoS.
- The Will Retain set to true means the will message is a retained message. Upon receiving a message with the retain flag set, the MQTT broker must store the message for the topic to which the message was published, and it must store only the latest message. So the subscribers which are interested in this topic can go offline, and reconnect at any time to receive the latest message instead of having to wait for the next message from the publisher after the subscription.

New Connection Parameters in MQTT v5.0

Clean Start & Session Expiry Interval

Clean Session was removed in MQTT 5.0, but Clean Start and Session Expiry Interval were added.

When Clean Start is true it discards any existing session and creates a new session. A false value means that the server must use the session associated with the Client ID to resume communication with the client (unless the session does not exist).

If Session Expiry Interval is set to 0 or is absent, the session ends when the network connection is closed. If it is 0xFFFFFFFF (UINT_MAX), the session does not expire. If it is greater than 0, the number of seconds the session will remain after the network connection is closed.

Check out the blog to learn more about Clean Start & Session Expiry Interval.

Connect Properties

MQTT 5.0 also adds new connection properties to enhance the extensibility of the protocol. Check out the blog to learn more about Connect Properties.

How to Establish a Secure MQTT Connection?

Although the MQTT protocol provides authentication mechanisms such as username-password, Client ID, etc., this is not enough for IoT security. With TCP-based plaintext transmission communication, it is difficult to guarantee data security.

TLS (Transport Layer Security), or in some context, the new-deprecated name SSL, aims primarily to provide privacy and data integrity between two or more communicating computer applications. Running on top of TLS, MQTT can take full advantage of its security features to secure data integrity and client trustworthy.

The steps to enable SSL/TLS vary from MQTT broker to MQTT broker. EMQX has built-in support for TLS/SSL, including support for one-way/two-way authentication, X.509 certificates, OCSP Stapling, and many other security certifications.

One-way authentication is a way to establish secure communication only by verifying the server certificate. It ensures the communication is encrypted but cannot verify the client's authenticity. It usually needs to be combined with authentication mechanisms such as username-password and client ID. Check out the blog to learn how to establish a secure One-way authenticated MQTT connection.

Two-way authentication means that both the server and the client must provide certificates when authenticating communications, and both parties need to authenticate to ensure that the other is trusted. Some application scenarios with high–security requirements require Two–way authentication to be enabled. Check out the blog to learn how to establish a secure Two–way authenticated MQTT connection.

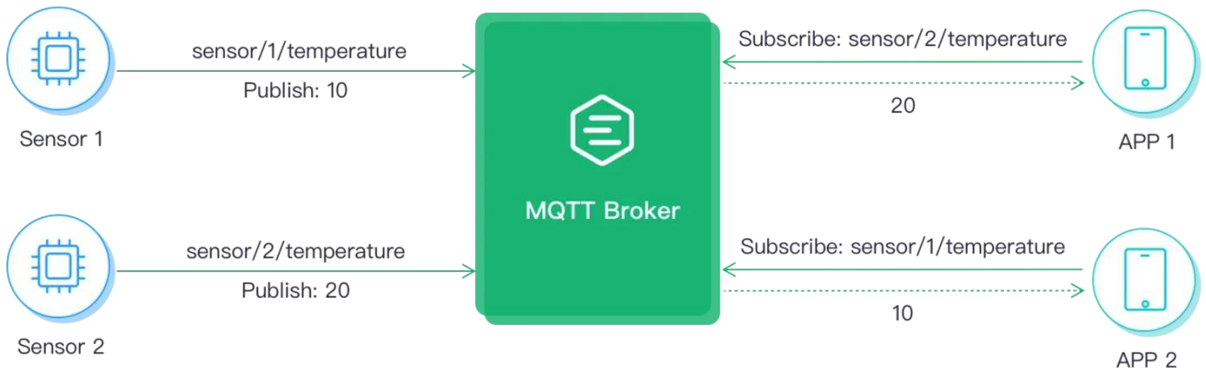
> **Note:** If you use MQTT over WebSocket on the browser, Two–way authentication communication is not yet supported.
>
> https://github.com/mqttjs/MQTT.js/issues/1515

MQTT Topics and Wildcards

MQTT topic is a string used in the MQTT protocol to identify and route messages. It is a key element in communication between MQTT publishers and subscribers. In the MQTT publish/subscribe model, publishers send messages to specific topics, while subscribers can subscribe to those topics to receive the messages.

In comparison to topics in other messaging systems, for example Kafka and Pulsar, MQTT topics are not to be created in advance. **The client will create the topic automatically when subscribing or publishing, and does not need to delete the topic.**

The following is a simple MQTT publish and subscribe flow. If APP 1 subscribes to the `sensor/2/temperature` topic, it will receive messages from Sensor 2 publishing to this topic.



Topics

A topic is a UTF-8 encoded string that is the basis for message routing in the MQTT protocol. A topic is typically leveled and separated with a slash `/` between the levels. This is similar to URL paths, for example:

```
chat/room/1
sensor/10/temperature
sensor/+/temperature
sensor/#
```

Although allowed, it is usually not recommended to use topics begin or end with `/`, such as `/chat` or `chat/`.

MQTT Wildcards

MQTT wildcards are a special type of topic that can only be used for subscription and not publishing. Clients can subscribe to a wildcard topic to receive messages from multiple matching topics, eliminating the need to subscribe to each topic individually and reducing overhead. MQTT supports two types of wildcards: `+` (single-level) and `#` (multi-level).

Single-level Wildcard

`+` (U+002B) is a wildcard character that matches only one topic level. When using a single-level wildcard, the single-level wildcard must occupy an entire level, for example:

```
"+" is valid
"sensor/+" is valid
"sensor/+/temperature" is valid
"sensor+" is invalid (does not occupy an entire level)
```

If the client subscribes to the topic `sensor/+/temperature`, it will receive messages from the following topics:

```
sensor/1/temperature
sensor/2/temperature
...
sensor/n/temperature
```

But it will not match the following topics:

```
sensor/temperature
sensor/bedroom/1/temperature
```

Multi-level Wildcard

`#` (U+0023) is a wildcard character that matches any number of levels within a topic. When using a multi-level wildcard, it must occupy an entire level and must be the last character of the topic, for example:

```
"#" is valid, matches all topics
"sensor/#" is valid
"sensor/bedroom#" is invalid (+ or # are only used as a wildcard level)
"sensor/#/temperature" is invalid (# must be the last level)
```

Topics Beginning with $

System Topics

The topics starting with `$SYS/` are system topics mainly used to get metadata about the MQTT broker's running status, statistics, client online/offline events, etc. `$SYS/` topic is not defined in the MQTT specification. However, most MQTT brokers follow this recommendation.

For example, the EMQX supports getting cluster status through the following topics.

Topic	Description
$SYS/brokers	EMQX cluster node list
$SYS/brokers/${node}/version	EMQX Broker version
$SYS/brokers/${node}/uptime	EMQX Broker startup time
$SYS/brokers/${node}/datetime	EMQX Broker time
$SYS/brokers/${node}/sysdescr	EMQX Broker description

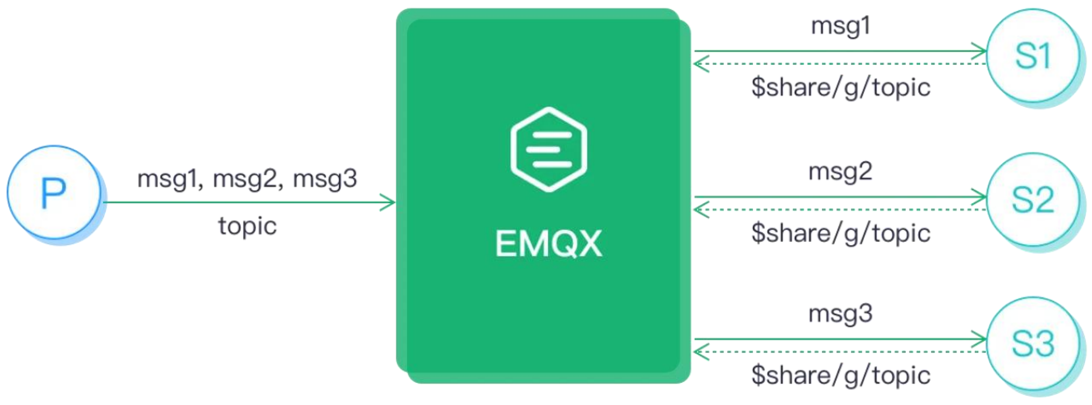
EMQX also supports rich system topics such as client online/offline events, statistics, system monitoring and alarms. For more details, please see the EMQX System Topics documentation.

Shared Subscriptions

Shared subscriptions are a feature of MQTT 5.0, a subscription method that achieves load balancing among multiple subscribers. The topic of a shared subscription starts with $share.

> Although the MQTT protocol added shared subscriptions in 5.0, EMQX has supported shared subscriptions since MQTT 3.1.1.

In the following diagram, three subscribers subscribe to the same topic `$share/g/topic` using a shared subscription method, where the `topic` is the real topic name they subscribe to, and the publishers publish messages to the `topic`, but NOT to `$share/g/topic`.



In addition, EMQX also supports the use of the shared subscription prefix $queue in MQTT 3.1.1. It is a special case of a shared subscription, which is equivalent to having all subscribers in one group.

For more details about shared subscriptions, please refer to EMQX Shared Subscriptions documentation.

Topics in Different Scenarios

Smart Home

For example, we use sensors to monitor the temperature, humidity and air quality of bedrooms, living rooms and kitchens. We can design the following topics:

- myhome/bedroom/temperature
- myhome/bedroom/humidity
- myhome/bedroom/airquality
- myhome/livingroom/temperature
- myhome/livingroom/humidity
- myhome/livingroom/airquality
- myhome/kitchen/temperature

- `myhome/kitchen/humidity`
- `myhome/kitchen/airquality`

Next, you can subscribe to the `myhome/bedroom/+` topic to get temperature, humidity and air quality data for the bedroom, the `myhome/+/temperature` topic to get temperature data for all three rooms, and the `myhome/#` topic to get all the data.

Charging Piles

- `ocpp/cp/cp001/notify/bootNotification`

  Publish an online request to this topic when the charging pile is online.
- `ocpp/cp/cp001/notify/startTransaction`

  Publish a charging request to this topic.
- `ocpp/cp/cp001/reply/bootNotification`

  Before the charging pile goes online, it needs to subscribe to this topic to receive the online response.
- `ocpp/cp/cp001/reply/startTransaction`

  Before the charging pile initiates the charging request, it needs to subscribe to this topic to receive the charging request response.

Instant Messaging

- `chat/user/${user_id}/inbox`

  One-to-one chat: Users subscribe to this topic after they are online and will receive messages from their friends. When replying to a friend, just replace the user_id of the topic with the friend's id.
- `chat/group/${group_id}/inbox`

  Group chat: After the user successfully joins a group, they can subscribe to the topic to get the group's messages.

- `req/user/${user_id}/add`

  Add a friend: Publish a friend request to this topic (user_id is the friend's id).

  Receive friend requests: Subscribe to this topic (user_id is the subscriber's id) to receive friend requests from other users.

- `resp/user/${user_id}/add`

  Receive replies to friend requests: Before adding friends, the user needs to subscribe to this topic (user_id is the subscriber's id) to receive the request results.

  Reply to friend request: Send a message to this topic (user_id is the friend's id) about whether or not to approve the friend request.

- `user/${user_id}/state`

  User Status: Subscribe to this topic to get your friends' online status.

MQTT Topics FAQ

What is the maximum level and length of an MQTT topic?

MQTT topic is UTF-8 encoded strings, and it MUST NOT be more than 65535 bytes. However in practice, using shorter length topic names and fewer levels means less resource consumption.

Try not to use more topic levels “just because I can”. For example, `my-home/room1/data` is a better choice than `my/home/room1/data`.

Is there a limit to the number of topics?

Different message servers have different limits on the number of topics. Currently, the default configuration of EMQX has no limit on the number of topics, but the more topics, the more server memory will be used.

Given the large number of devices connected to the MQTT Broker, we recommend that a client subscribes to no more than ten topics.

Do wildcard subscriptions degrade performance?

When routing messages to wildcard subscriptions, the broker may require more resources than non-wildcard topics. It is a wise choice if the wildcard subscription can be avoided.

This very much depends on how the data schema is modeled for the MQTT message payload.

For example, if a publisher publishes to `device-id/stream1/foo` and `device-id/stream1/bar` and the subscriber needs to subscribe to both, then it may subscribe `device-id/stream1/#`. A better alternative is perhaps to push the foo and bar part of the namespace down to the payload, so it publishes to only one topic `device-id/stream1`, and the subscriber just subscribes to this one topic.

How are messages received for overlapping subscriptions of normal and wildcard topics?

For example, if a client subscribes to both `#` and `test` topics, will it receive two duplicate messages when publishing to `test`? This depends on the MQTT broker implementation. EMQX will send messages for each matched subscription. Thus, duplicates may occur. However, users can leverage MQTT 5.0 subscription identifiers to differentiate message sources and handle such duplicate messages in the client based on the identifiers.

Can I subscribe to the same topic with a shared subscription and a normal subscription?

Yes, but it is not recommended.

Per MQTT specification, multiple subscriptions will result in multiple (duplicated) message deliveries.

What are the best practices for MQTT topics?

- Do not use `#` to subscribe to all topics.
- The topic should not start or end with `/`, such as `/chat` or `chat/`.
- Do not use spaces and non-ASCII characters in the topic.
- Use `_` or `-` to connect words (or camel case) within a topic level.
- Try to use less topic levels.
- Try to model the message data schema in favor to avoid using wildcard topics.
- When wildcard is in use, try to move the more unique topic level closer to root. e.g. `device/00000001/command/#` is a better choice than `device/command/00000001/#`.

MQTT Persistent Session and Clean Session

MQTT Persistent Session

Unstable networks and limited hardware resources are the two major problems that IoT applications need to face. The connection between MQTT clients and brokers can be abnormally disconnected at any time due to network fluctuations and resource constraints. To address the impact of network disconnection on communication, the MQTT protocol provides Persistent Session.

MQTT client can set whether to use a Persistent Session when initiating a connection to the server. A Persistent Session will hold some important data to allow the session to continue over multiple network connections. Persistent Session has three main functions as follows:

- Avoid the additional overhead of need to subscribe repeatedly due to network outages.
- Avoid missing messages during offline periods.
- Ensuring that QoS 1 and QoS 2 messages are not affected by network outages.

What Data Need to Store for a Persistent Session?

We know from the above that Persistent Session needs to store some important data in order for the session to be recovered. Some of this data is stored on the client side and some on the server side.

Session data stored in the client:

- QoS 1 and QoS 2 messages have been sent to the server but have not yet completed acknowledgment.

- QoS 2 messages that were received from the server but have not yet completed acknowledgment.

Session data stored in the server:

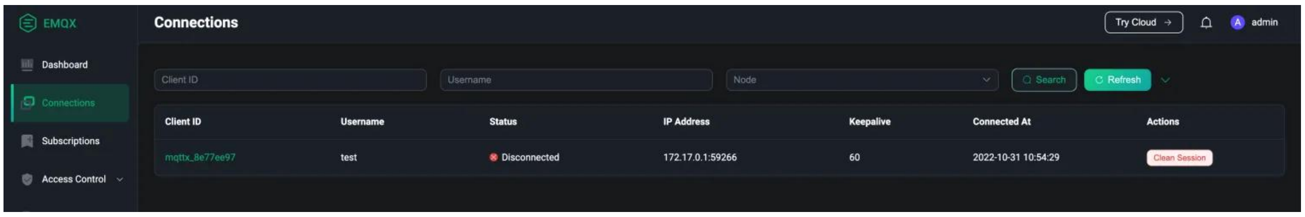
- Whether the session exists, even if the rest of the session status is empty.
- QoS 1 and QoS 2 messages that have been sent to the client but have not yet completed acknowledgment.
- QoS 0 messages (optional), QoS 1 and QoS 2 messages that are waiting to be transmitted to the client.
- QoS 2 messages that are received from the client but have not yet completed acknowledgment, Will Messages, and Will Delay Intervals.

Using MQTT Clean Session

Clean Session is a flag bit used to control the life cycle of the session state. A value of 1 means that a brand new session will be created on connection, and the session will be automatically destroyed when the client disconnects. If it is 0, it means that it will try to reuse the previous session when connecting. If there is no corresponding session, a new session will be created, which will always exist after the client disconnects.

> **Note**: A Persistent Session can be resumed only if the client connects again using a fixed Client ID. If the Client ID is dynamic, a new Persistent Session will be created after a successful connection.

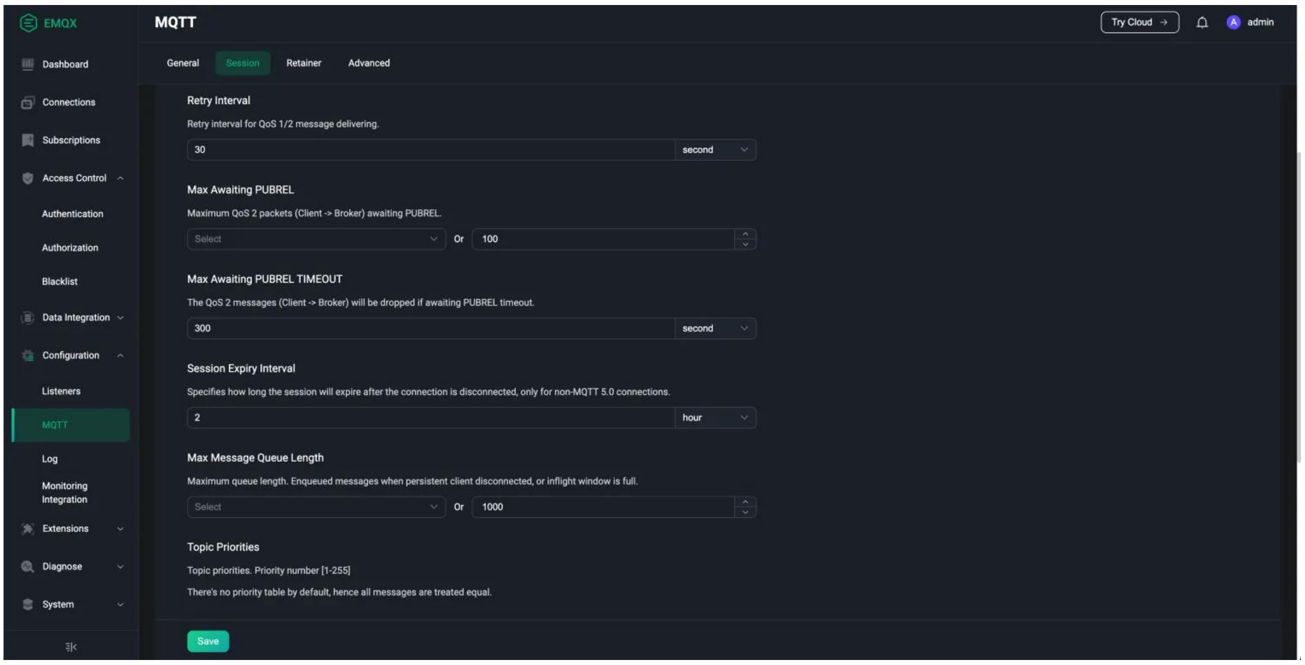
The following is the Dashboard of the open-source MQTT broker EMQX. You can see that the connection in the diagram is disconnected, but because it is a Persistent Session, it can still be viewed in the Dashboard.



Connections

Client ID	Username	Status	IP Address	Keepalive	Connected At	Actions
mqttx_8e77ee97	test	Disconnected	172.17.0.1:59266	60	2022-10-31 10:54:29	Clean Session

EMQX also supports setting session–related parameters in the Dashboard.



MQTT

Retry Interval
Retry interval for QoS 1/2 message delivering.
30 second

Max Awaiting PUBREL
Maximum QoS 2 packets (Client -> Broker) awaiting PUBREL.
Select Or 100

Max Awaiting PUBREL TIMEOUT
The QoS 2 messages (Client -> Broker) will be dropped if awaiting PUBREL timeout.
300 second

Session Expiry Interval
Specifies how long the session will expire after the connection is disconnected, only for non-MQTT 5.0 connections.
2 hour

Max Message Queue Length
Maximum queue length. Enqueued messages when persistent client disconnected, or inflight window is full.
Select Or 1000

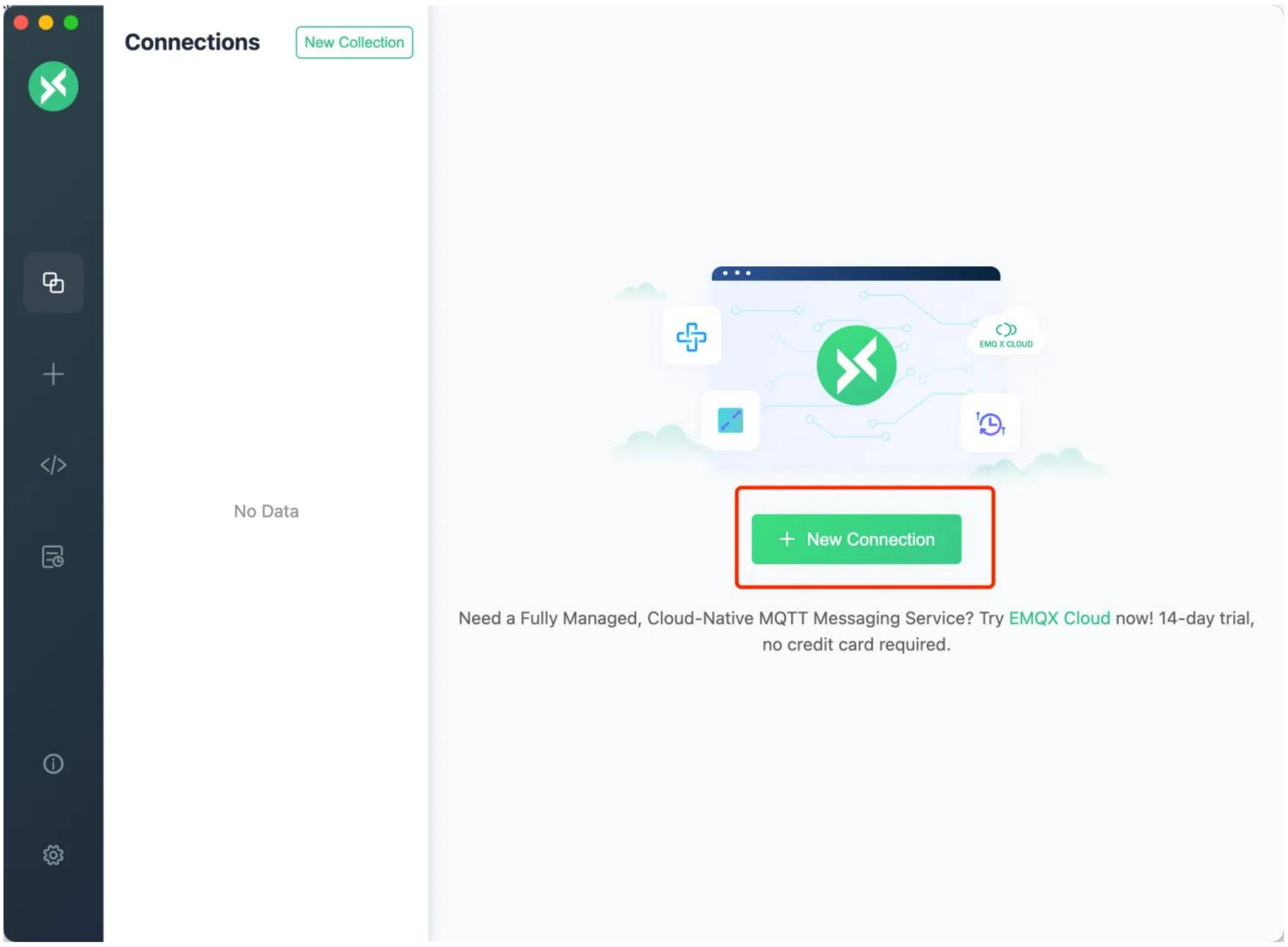
Topic Priorities
Topic priorities. Priority number [1-255]
There's no priority table by default, hence all messages are treated equal.

MQTT 3.1.1 does not specify when a Persistent Session will expire; if understood at the protocol level alone, this Persistent Session should be permanent. However, this is not practical in a real–world scenario because it takes up a lot of resources on the server side. So, the server usually does not follow the protocol exactly, but provides a global configuration to the user to limit the session expiration time.

For example, the Free Public MQTT Broker provided by EMQ sets a session expiration time of 5 minutes, a maximum number of 1000 messages, and does not save QoS 0 messages.

Next, we will demonstrate the use of Clean Session with the open–source cross–platform MQTT 5.0 desktop client tool – MQTTX.

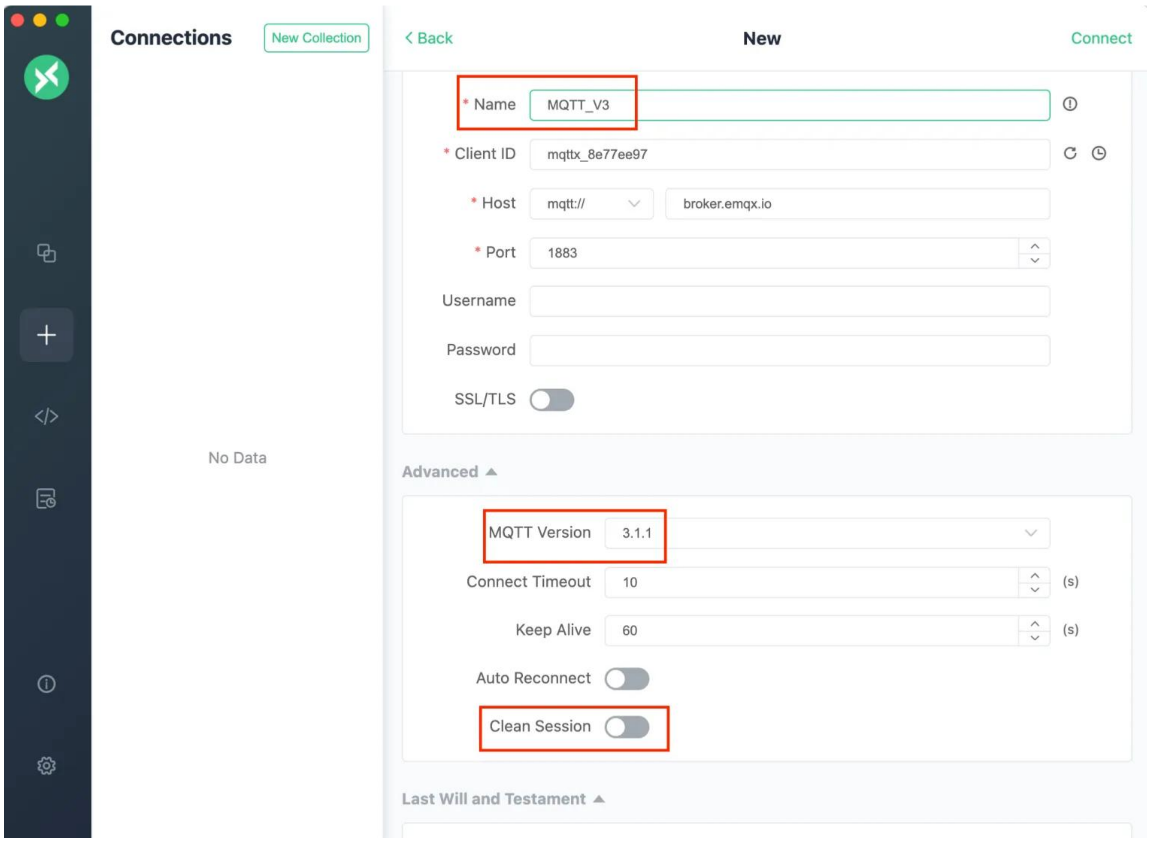
After opening MQTTX, click New Connection button to create an MQTT connection as shown below.



Need a Fully Managed, Cloud-Native MQTT Messaging Service? Try EMQX Cloud now! 14-day trial, no credit card required.

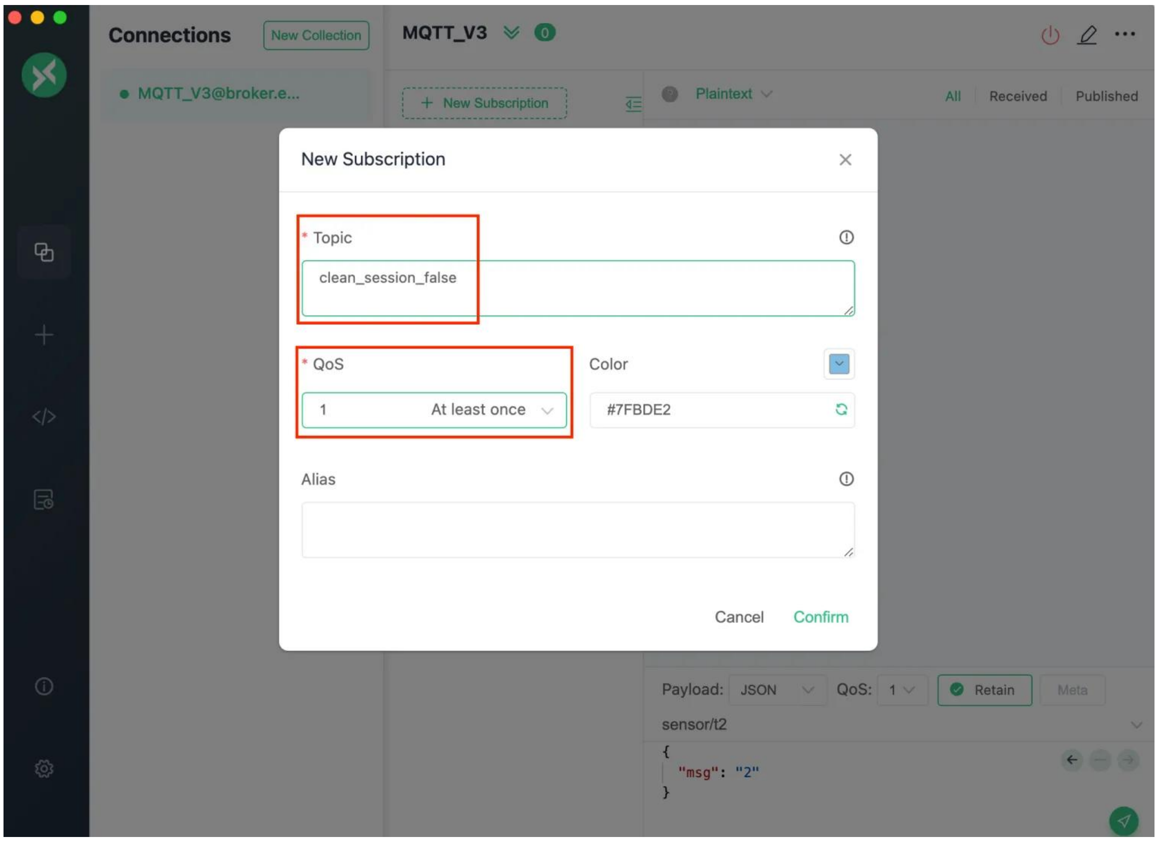
Create a connection named MQTT_V3 with Clean Session off (i.e., false), and select MQTT version 3.1.1, then click Connect button in the upper right corner.

> The default server connected to is the Free Public MQTT Broker provided by EMQ.



* Name: MQTT_V3
* Client ID: mqttx_8e77ee97
* Host: mqtt:// broker.emqx.io
* Port: 1883
Username:
Password:
SSL/TLS: off
MQTT Version: 3.1.1
Connect Timeout: 10 (s)
Keep Alive: 60 (s)
Auto Reconnect: off
Clean Session: off

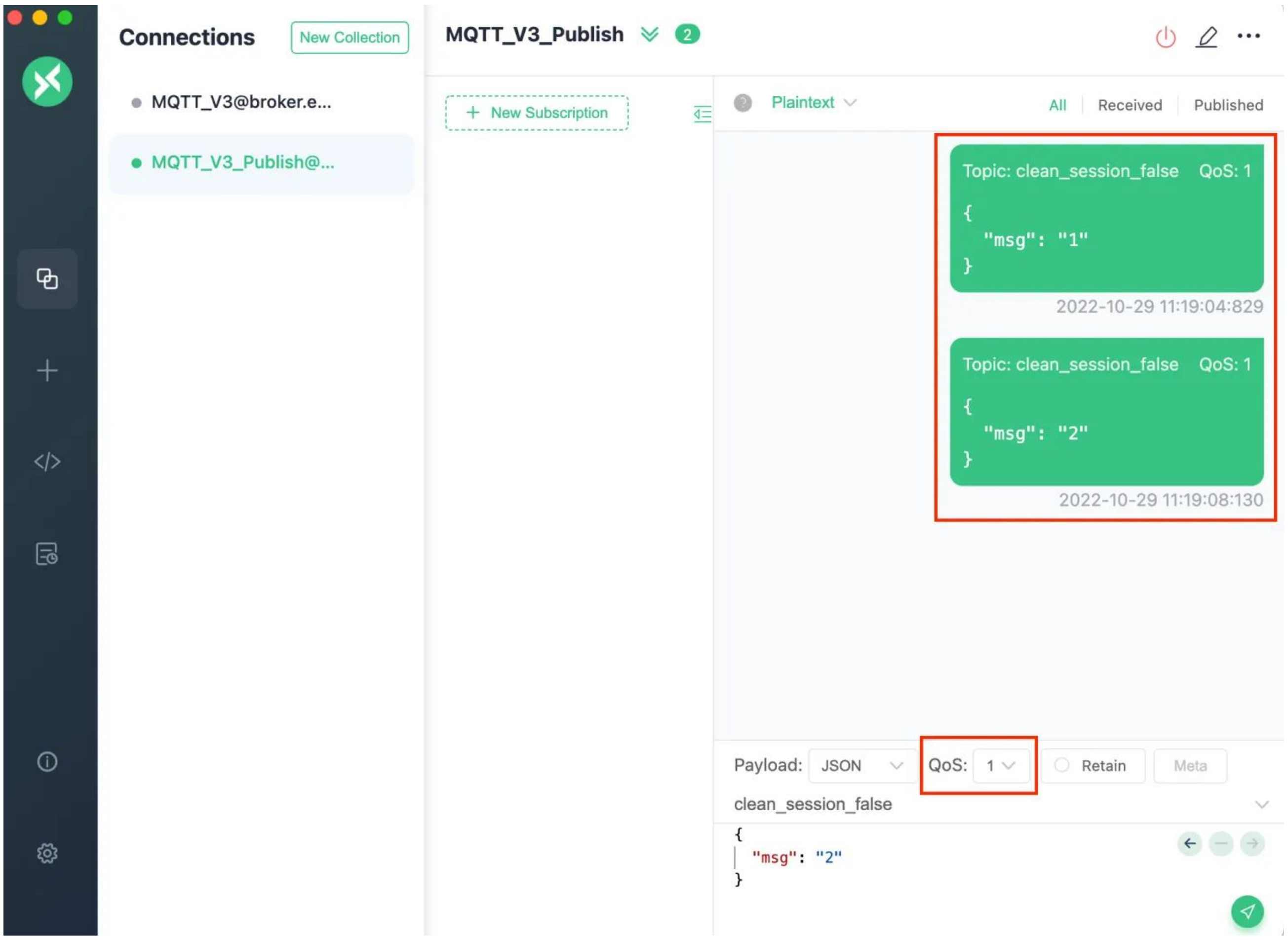
Subscribe to clean_session_false topic after a successful connection and QoS is set to 1.



New Subscription
* Topic: clean_session_false
* QoS: 1 At least once
Color: #7FBDE2
Alias:

```
{
  "msg": "2"
}
```

After a successful subscription, click Disconnect button in the upper right corner. Then, create a connection named MQTT_V3_Publish, again with the MQTT version set to 3.1.1, and publish two QoS 1 messages to clean_session_false topic after a successful connection.



Topic: clean_session_false QoS: 1

```
{
  "msg": "1"
}
```

2022-10-29 11:19:04:829

Topic: clean_session_false QoS: 1

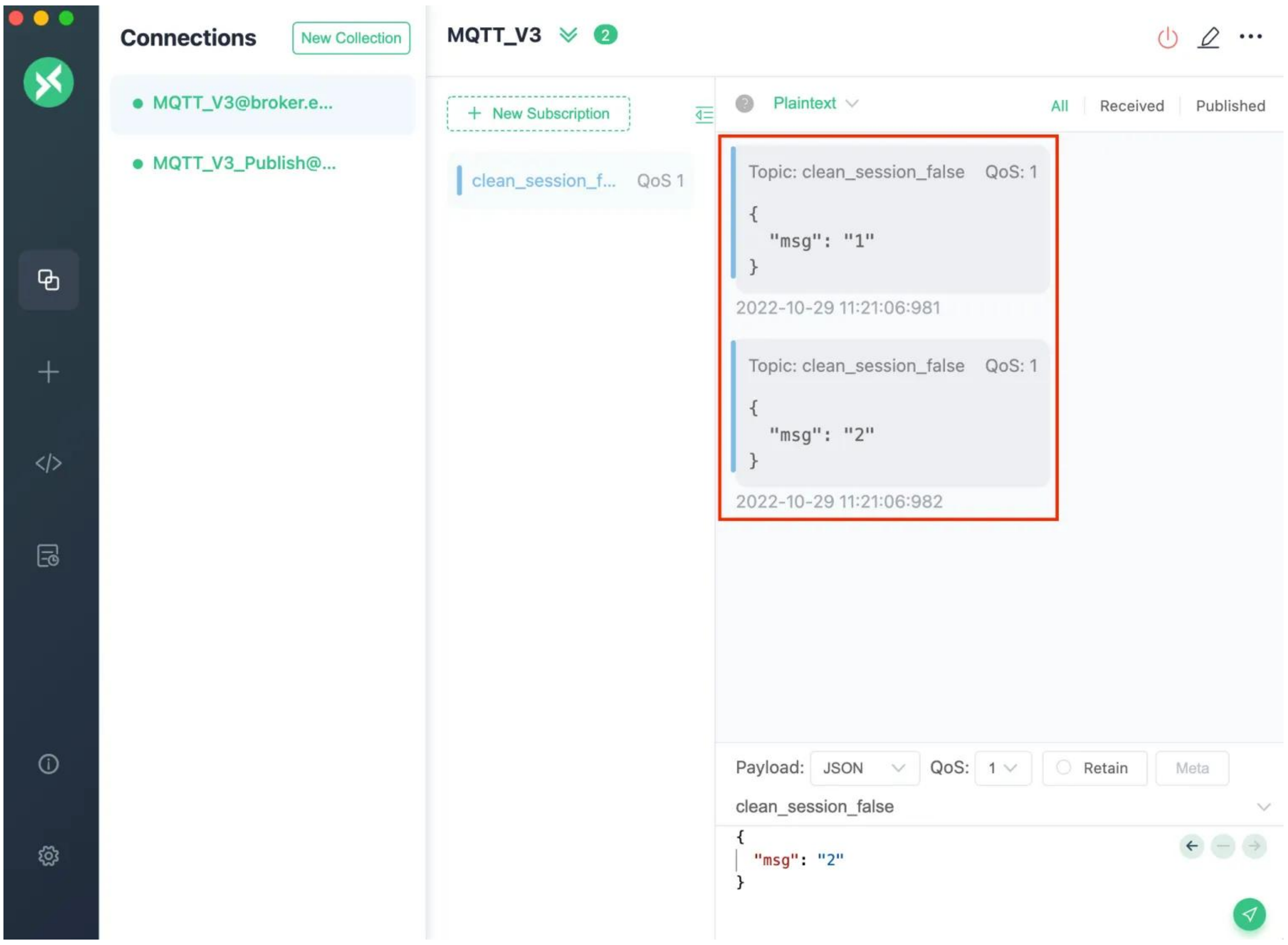
```
{
  "msg": "2"
}
```

2022-10-29 11:19:08:130

clean_session_false

```
{
  "msg": "2"
}
```

Then, select the MQTT_V3 connection and click Connect button to connect to the server. You will successfully receive two messages that were published during the offline period.



```
Topic: clean_session_false    QoS: 1
{
  "msg": "1"
}
2022-10-29 11:21:06:981

Topic: clean_session_false    QoS: 1
{
  "msg": "2"
}
2022-10-29 11:21:06:982
```

Payload: JSON QoS: 1 Retain Meta

clean_session_false

```
{
  "msg": "2"
}
```

Session Improvements in MQTT 5.0

In MQTT 5.0, Clean Session is split into Clean Start and Session Expiry Interval. Clean Start specifies whether to create a new session or try to reuse an existing session when connecting. Session Expiry Interval is used to specify how long the session will expire after the network connection is disconnected.

Clean Start of true means that any existing session must be discarded, and a completely new session is created; false indicates that the session associated with the Client ID must be used to resume communication with the client (unless the session does not exist).

Session Expiry Interval solves the server resource waste problem caused by the permanent existing of Persistent Sessions in MQTT 3.1.1. A setting of 0 or none indicates that the session expires when disconnected. A value greater than 0 indicates how many seconds the session will remain after the network connection is closed. A setting of `0xFFFFFFFF` means that the session will never expire.

More details are available in the blog: Clean Start and Session Expiry Interval.

Q&A About MQTT Session

When the session ends, do the Retained Messages still exist?

MQTT Retained Messages are not part of the session state and will not be deleted at the end of the session.

How does the client know that the current session is the resumed session?

The MQTT protocol has designed a Session Present field for CONNACK message since v3.1.1. When the server returns a value of 1 for this field, it means that the current connection will reuse the session saved by the server. The client can use this field value to decide whether to re-subscribe after a successful connection.

Best practices for using Persistent Session

- You cannot use dynamic Client ID. You need to ensure that the Client ID is fixed for each client connection.
- Properly evaluate the session expiration time based on server performance, network conditions, and client type. Setting it too long will take up more server-side resources. And setting it too short will cause the session to expire before reconnecting

successfully.

- When the client determines that the session is no longer needed, you can reconnect using Clean Session as true, and then disconnect after a successful reconnection. In the case of MQTT 5.0, you can set the Session Expiry Interval as 0 directly when disconnecting, indicating that the session will expire when the connection is disconnected.

MQTT QoS

What is QoS

In unstable network environments, MQTT devices may struggle to ensure reliable communication using only the TCP transport protocol. To address this issue, MQTT includes a Quality of Service (QoS) mechanism that offers various message interaction options to provide different levels of service, catering to the user's specific requirements for reliable message delivery in different scenarios.

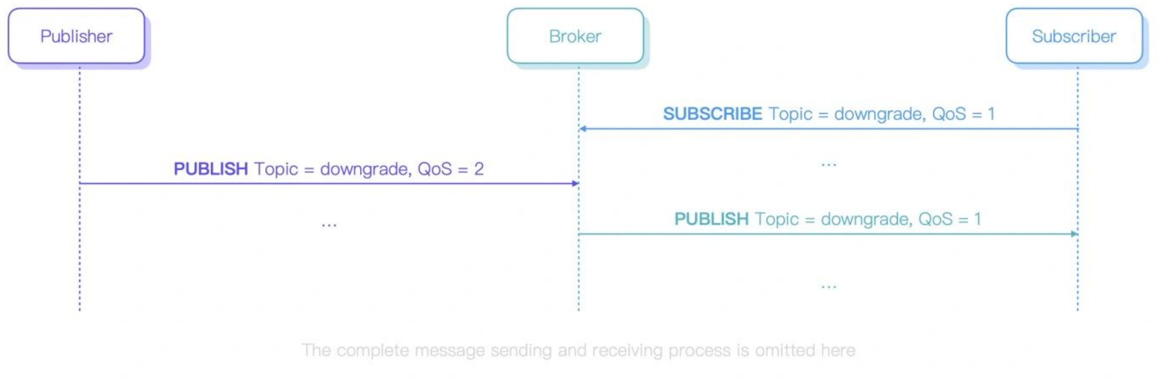
There are 3 QoS levels in MQTT:

- QoS 0, at most once.
- QoS 1, at least once.
- QoS 2, exactly once.

These levels correspond to increasing levels of reliability for message delivery. QoS 0 may lose messages, QoS 1 guarantees the message delivery but potentially exists duplicate messages, and QoS 2 ensures that messages are delivered exactly once without duplication. As the QoS level increases, the reliability of message delivery also increases, but so does the complexity of the transmission process.

In the publisher-to-subscriber delivery process, the publisher specifies the QoS level of a message in the PUBLISH packet. The broker typically forwards the message to the subscriber with the same QoS level. However, in some cases, the subscriber's requirements may necessitate a reduction in the QoS level of the forwarded message.

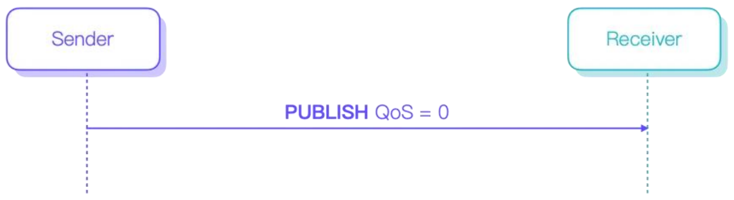
For example, if a subscriber specifies that they only want to receive messages with a QoS level of 1 or lower, the broker will downgrade any QoS 2 messages to QoS 1 before forwarding them to this subscriber. Messages with QoS 0 and QoS 1 will be transmitted to the subscriber with their original QoS levels unchanged.


Publisher
Broker
Subscriber
SUBSCRIBE Topic = downgrade, QoS = 1
PUBLISH Topic = downgrade, QoS = 2
PUBLISH Topic = downgrade, QoS = 1
The complete message sending and receiving process is omitted here

Let's see how QoS works.

QoS 0 – At Most Once

QoS 0 is the lowest level of service and is also known as "fire and forget". In this mode, the sender does not wait for acknowledgement or store and retransmit the message, so the receiver does not need to worry about receiving duplicate messages.



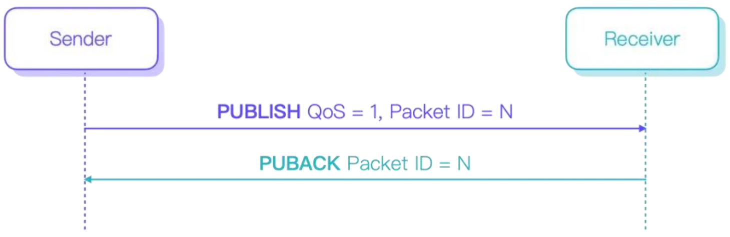
Why Are QoS 0 Messages Lost?

The reliability of QoS 0 message transmission depends on the stability of the TCP connection. If the connection is stable, TCP can ensure the successful delivery of messages. However, if the connection is closed or reset, there is a risk that messages in transit or messages in the operating system buffer may be lost, resulting in the unsuccessful delivery of QoS 0 messages.

QoS 1 – At Least Once

To ensure message delivery, QoS 1 introduces an acknowledgement and retransmission mechanism. When the sender receives a PUBACK packet from the receiver, it considers the message delivered successfully. Until then, the sender must store the PUBLISH packet for potential retransmission.

The sender uses the Packet ID in each packet to match the PUBLISH packet with the corresponding PUBACK packet. This allows the sender to identify and delete the correct PUBLISH packet from its cache.



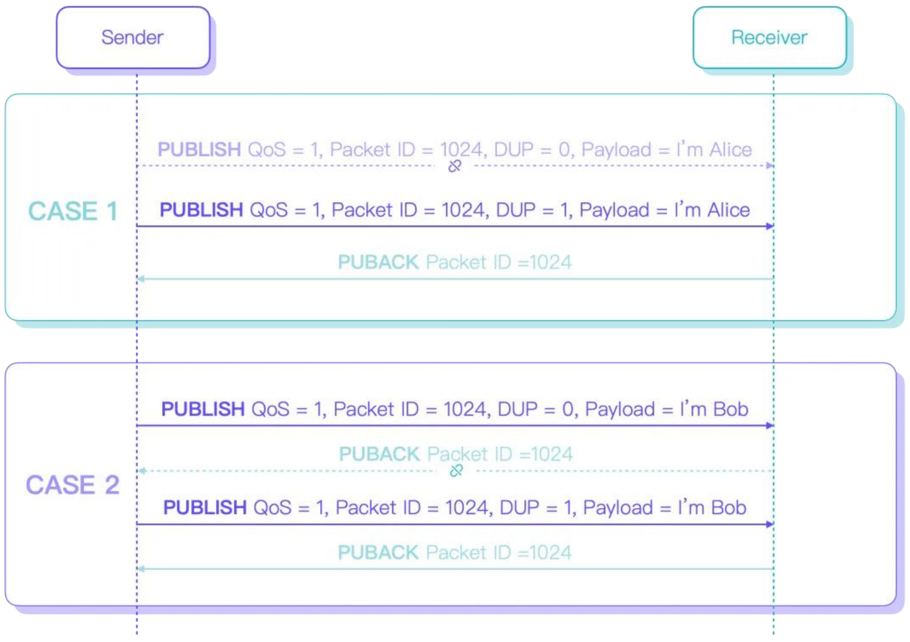
Why Are QoS 1 Messages Duplicated?

There are two cases in which the sender will not receive a PUBACK packet.

- The PUBLISH packet did not reach the receiver.
- The PUBLISH packet reached the receiver but the receiver's PUBACK packet has not yet been received by the sender.

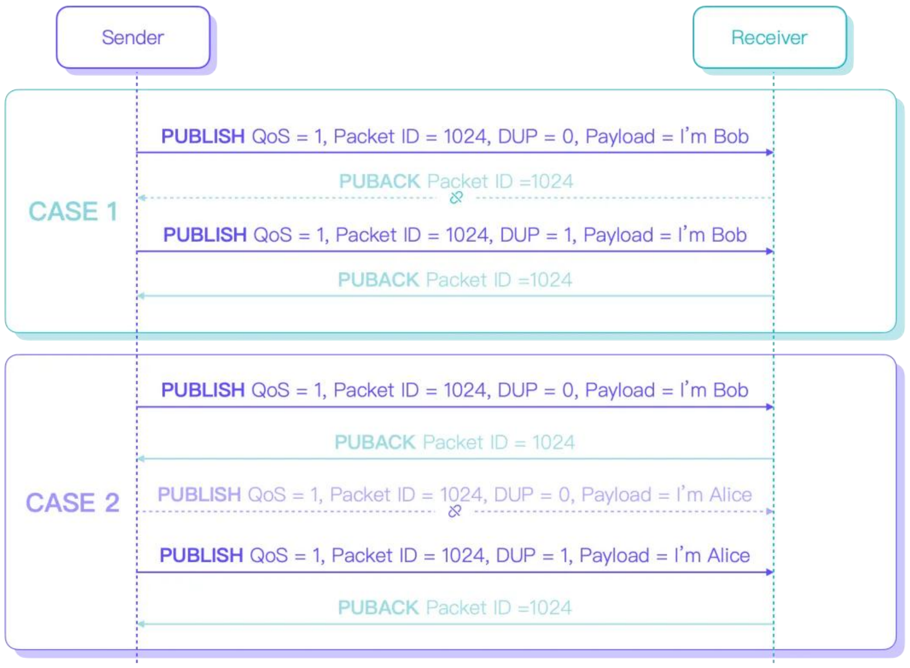
In the first case, the sender will retransmit the PUBLISH packet, but the receiver will only receive the message once.

In the second case, the sender will retransmit the PUBLISH packet and the receiver will receive it again, resulting in a duplicate message.


Sender
Receiver
CASE 1
PUBLISH QoS = 1, Packet ID = 1024, DUP = 0, Payload = I'm Alice
PUBLISH QoS = 1, Packet ID = 1024, DUP = 1, Payload = I'm Alice
PUBACK Packet ID =1024
CASE 2
PUBLISH QoS = 1, Packet ID = 1024, DUP = 0, Payload = I'm Bob
PUBACK Packet ID =1024
PUBLISH QoS = 1, Packet ID = 1024, DUP = 1, Payload = I'm Bob
PUBACK Packet ID =1024

Even though the DUP flag in the retransmitted PUBLISH packet is set to 1 to indicate that it is a duplicate message, the receiver cannot assume that it has already received the message and must still treat it as a new message.

It is because that there are two possible scenarios when the receiver receives a PUBLISH packet with a DUP flag of 1:


Sender
Receiver
CASE 1
PUBLISH QoS = 1, Packet ID = 1024, DUP = 0, Payload = I'm Bob
PUBACK Packet ID =1024
PUBLISH QoS = 1, Packet ID = 1024, DUP = 1, Payload = I'm Bob
PUBACK Packet ID =1024
CASE 2
PUBLISH QoS = 1, Packet ID = 1024, DUP = 0, Payload = I'm Bob
PUBACK Packet ID = 1024
PUBLISH QoS = 1, Packet ID = 1024, DUP = 0, Payload = I'm Alice
PUBLISH QoS = 1, Packet ID = 1024, DUP = 1, Payload = I'm Alice
PUBACK Packet ID =1024

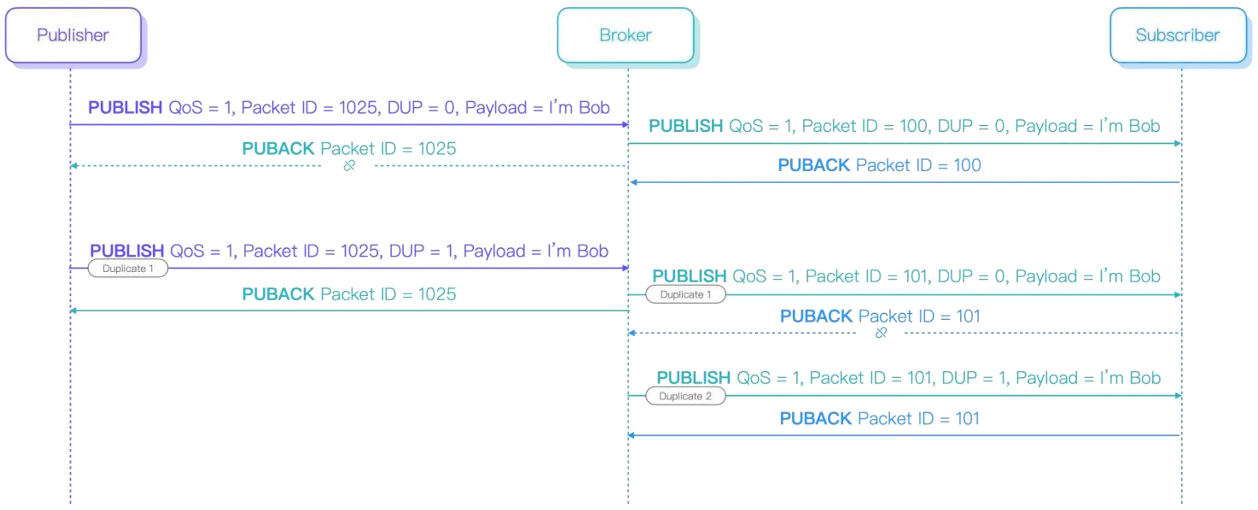
In the first case, the sender retransmits the PUBLISH packet because it did not receive a PUBACK packet. The receiver receives two PUBLISH packets with the same Packet ID and the second PUBLISH packet has a DUP flag of 1. The second packet is indeed a duplicate message.

In the second case, the original PUBLISH packet was delivered successfully. Then, this Packet ID is used for a new, unrelated message. But this new message was not successfully delivered to the peer the first time it was sent, so it was retransmitted. Finally, the retransmitted PUBLISH packet will have the same Packet ID and a DUP flag of 1, but it is a new message.

Since it is not possible to distinguish between these two cases, the receiver must treat all PUBLISH packets with a DUP flag of 1 as new messages. This means that it is inevitable for there to be duplicate messages at the protocol level when using QoS 1.

In rare cases, the broker may receive duplicate PUBLISH packets from the publisher and, during the process of forwarding them to the subscriber, retransmit them again. This can result in the subscriber receiving additional duplicate messages.

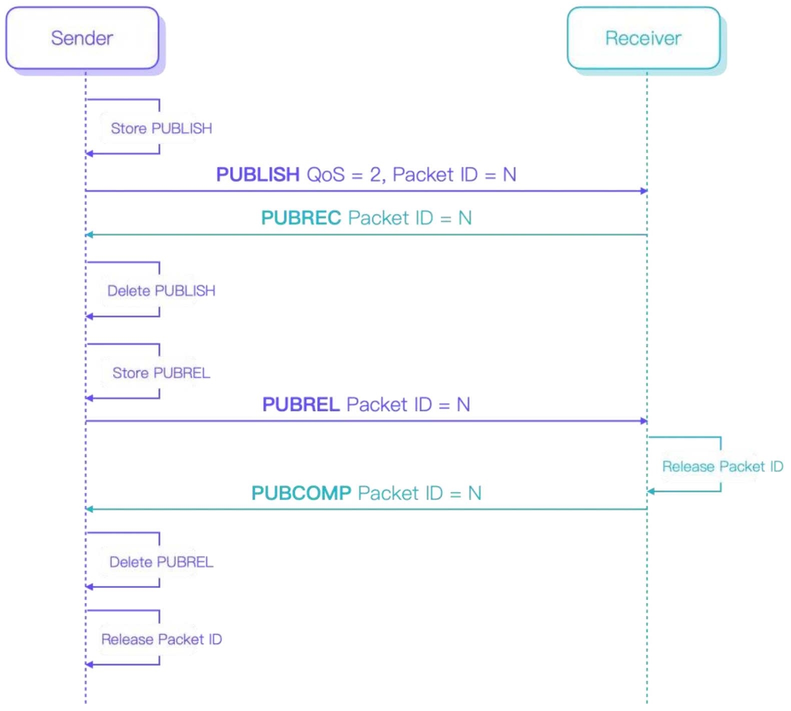
For example, although the publisher only sends one message, the receiver may eventually receive three identical messages.


Publisher
Broker
Subscriber
PUBLISH QoS = 1, Packet ID = 1025, DUP = 0, Payload = I'm Bob
PUBLISH QoS = 1, Packet ID = 100, DUP = 0, Payload = I'm Bob
PUBACK Packet ID = 1025
PUBACK Packet ID = 100
PUBLISH QoS = 1, Packet ID = 1025, DUP = 1, Payload = I'm Bob
Duplicate 1
PUBLISH QoS = 1, Packet ID = 101, DUP = 0, Payload = I'm Bob
Duplicate 1
PUBACK Packet ID = 1025
PUBACK Packet ID = 101
PUBLISH QoS = 1, Packet ID = 101, DUP = 1, Payload = I'm Bob
Duplicate 2
PUBACK Packet ID = 101

These are the drawbacks of using QoS 1.

QoS 2 – Exactly Once

QoS 2 ensures that messages are not lost or duplicated, unlike in QoS 0 and 1. However, it also has the most complex interactions and the highest overhead, as it requires at least two request/response flows between the sender and receiver for each message delivery.



1. To initiate a QoS 2 message transmission, the sender first stores and sends a PUBLISH packet with QoS 2 and then waits for a PUBREC response packet from the receiver. This process is similar to QoS 1, with the exception that the response packet is PUBREC instead of PUBACK.

2. Upon receiving a PUBREC packet, the sender can confirm that the PUBLISH packet was received by the receiver and can delete its locally stored copy. It **no longer needs and cannot retransmit** this packet. The sender then sends a PUBREL packet to inform the receiver that it is ready to release the Packet ID. Like the PUBLISH packet, the PUBREL packet needs to be reliably delivered to the receiver, so it is stored for

potential retransmission and a response packet is required.

3. When the receiver receives the PUBREL packet, it can confirm that no additional retransmitted PUBLISH packets will be received in this transmission flow. As a result, the receiver responds with a PUBCOMP packet to signal that it is prepared to reuse the current Packet ID for a new message.
4. When the sender receives the PUBCOMP packet, the QoS 2 flow is complete. The sender can then send a new message with the current Packet ID, which the receiver will treat as a new message.

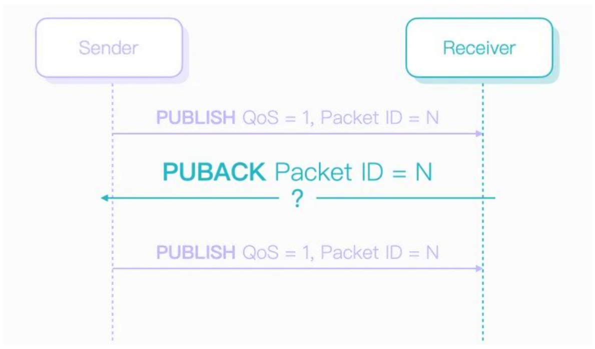
Why Are QoS 2 Messages Not Duplicated?

The mechanisms used to ensure that QoS 2 messages are not lost are the same as those used for QoS 1, so they will not be discussed again here.

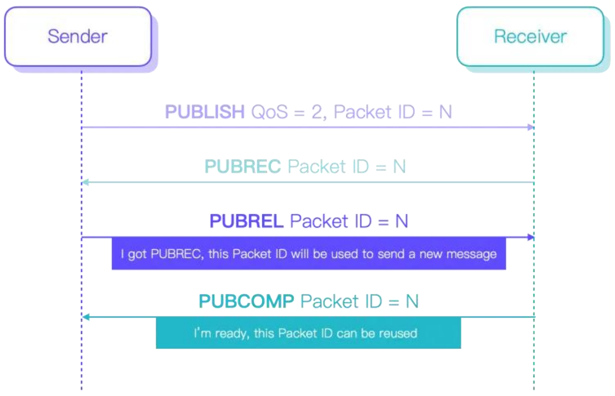
Compared with QoS 1, QoS 2 ensures that messages are not duplicated by adding a new process involving the PUBREL and PUBCOMP packets.

Before we go any further, let's quickly review the reasons why QoS 1 cannot avoid message duplication.

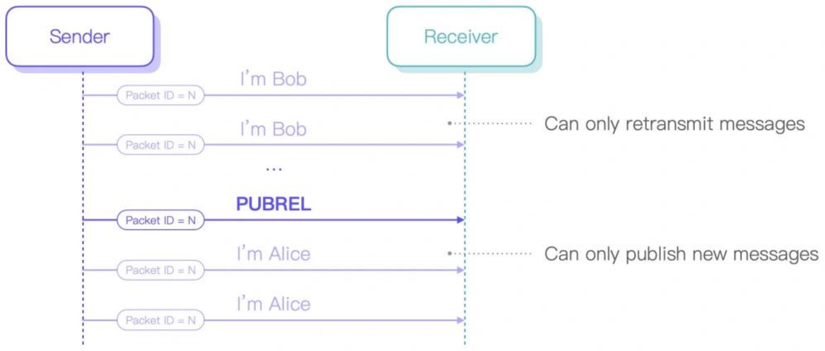
When we use QoS 1, for the receiver, the Packet ID becomes available again after the PUBACK packet is sent, regardless of whether the response has reached the sender. This means that the receiver cannot determine whether the PUBLISH packet it receives later, with the same Packet ID, is a retransmission from the sender due to not receiving the PUBACK response, or if the sender has reused the Packet ID to send a new message after receiving the PUBACK response. This is why QoS 1 cannot avoid message duplication.


Sender
Receiver
PUBLISH QoS = 1, Packet ID = N
PUBACK Packet ID = N
?
PUBLISH QoS = 1, Packet ID = N

In QoS 2, the sender and receiver use the PUBREL and PUBCOMP packets to synchronize the release of Packet IDs, ensuring that there is a consensus on whether the sender is retransmitting a message or sending a new one. This is the key to avoiding the issue of duplicate messages that can occur in QoS 1.


Sender
Receiver
PUBLISH QoS = 2, Packet ID = N
PUBREC Packet ID = N
PUBREL Packet ID = N
I got PUBREC, this Packet ID will be used to send a new message
PUBCOMP Packet ID = N
I'm ready, this Packet ID can be reused

In QoS 2, the sender is permitted to retransmit the PUBLISH packet before receiving the PUBREC packet from the receiver. Once the sender receives the PUBREC and sends a PUBREL packet, it enters the Packet ID release process. The sender cannot retransmit the PUBLISH packet or send a new message with the current Packet ID until it receives a PUBCOMP packet from the receiver.



As a result, the receiver can use the PUBREL packet as a boundary and consider any PUBLISH packet that arrives before it as a duplicate and any PUBLISH packet that arrives after it as new. This allows us to avoid message duplication at the protocol level when using QoS 2.

Scenarios and Considerations

QoS 0

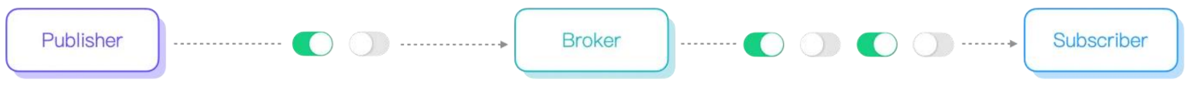
The main disadvantage of QoS 0 is that messages may be lost, depending on the network conditions. This means that you may miss messages if you are disconnected. However, the advantage of QoS 0 is that it is more efficient for message delivery.

Therefore, it is often used to send high-frequency, less important data, such as periodic sensor updates, where it is acceptable to miss a few updates.

QoS 1

QoS 1 ensures that messages are delivered at least once, but it can result in duplicate messages. This makes it suitable for transmitting important data such as critical instructions or real-time updates of important status. However, it is important to consider how to handle or allow for such duplication before deciding to use QoS 1 without de-duplication.

For instance, if the publisher sends messages in the order 1, 2, but the subscriber receives them in the order 1, 2, 1, 2, with 1 representing a command to turn a light on and 2 representing a command to turn it off, it may not be desirable for the light to repeatedly turn on and off due to duplicate messages.



QoS 2

QoS 2 ensures that messages are not lost or duplicated. However, it also has the highest overhead. If users are not willing to handle message duplication by themself and can accept the additional overhead of QoS 2, then it is a suitable choice. QoS 2 is often used in industries such as finance and aviation where it is critical to ensure reliable message delivery and avoid duplication.

Q&A

How to de-duplicate QoS 1 messages?

As duplication of QoS 1 messages is inherent at the protocol level, so we can only solve this problem at the business level .

One way to de-duplicate QoS 1 messages is to include a timestamp or a monotonically increasing count in the payload of each PUBLISH packet. This allows you to determine whether the current message is new by comparing its timestamp or count with that of the last received message.

When should QoS 2 messages be forwarded to subscribers?

As we have learned, QoS 2 has a high overhead. To avoid impacting the real-time nature of QoS 2 messages, it is best to initiate the process of forwarding them to subscribers when the QoS 2 PUBLISH packet is received for the first time. However, once this process has been initiated, subsequent PUBLISH packets that arrive before the PUBREL packet should not be forwarded again to prevent message duplication.

Is there a difference in performance between QoS?

QoS 0 and QoS 1 typically have similar throughput when using EMQX with the same hardware configuration for peer-to-peer communication, but QoS 1 may have higher CPU usage. Additionally, under high load, QoS 1 has longer message latency compared to QoS 0. On the other hand, QoS 2 usually only has about half the throughput of QoS 0 and 1.

MQTT Keep Alive

Why Do We Need Keep Alive?

The MQTT protocol is hosted on top of the TCP protocol, which is connection-oriented, and provides a stable and orderly flow of bytes between two connected parties. However, in some cases, TCP can have half-connection problems. A half-connection is a connection that has been disconnected or not established on one side, while the connection on the other side is still maintained. In this case, the half-connected party may continuously send data, which obviously never reaches the other side. To avoid black holes in communication caused by half-connections, the MQTT protocol provides a Keep Alive mechanism that allows the client and MQTT server to determine whether there is a half-connection problem, and close the corresponding connection.

Mechanism and Use of MQTT Keep Alive

At Connection

When an MQTT client creates a connection to the MQTT broker, the Keep Alive mechanism can be enabled between the communicating parties by setting the Keep Alive variable header field in the connection request protocol packet to a non-zero value. Keep Alive is an integer from 0 to 65535, representing the maximum time in seconds allowed to elapse between MQTT protocol packets sent by the client.

When the broker receives a connection request from a client, it checks the value of the Keep Alive field in the variable header. When there is a value, the broker will enable the Keep Alive mechanism.

MQTT 5.0 Server Keep Alive

In the MQTT 5.0 standard, the concept of Server Keep Alive was also introduced, allowing the broker to choose to accept the Keep Alive value carried in the client request, or to override it, depending on its implementation and other factors. If the broker chooses to override this value, it needs to set the new value in the Server Keep Alive field of the Connection Acknowledgement Packet (CONNACK), and the client needs to use this value to override its own previous Keep Alive value when it reads it in the CONNACK.

The Keep Alive Process

Client process

After the connection is established, the client needs to ensure that the interval between any two MQTT protocol packets it sends does not exceed the Keep Alive value. If the client is idle and has no packets to send, it can send PINGREQ protocol packets, instead.

When the client sends a PINGREQ packet, the broker must return a PINGRESP packet. If the client does not receive a PINGRESP packet from the server within a reliable time, it means that there is a half-connection, the broker is offline, or there is a network failure, and the client should close the connection.

Broker process

After the connection is established, if the broker does not receive any packets from the client within 1.5 times the Keep Alive time, it will assume that there is a problem with the connection to the client, and the broker will disconnect from the client.

If the broker receives a PINGREQ protocol packet from the client, it needs to reply with a PINGRESP protocol packet for confirmation.

Client takeover mechanism

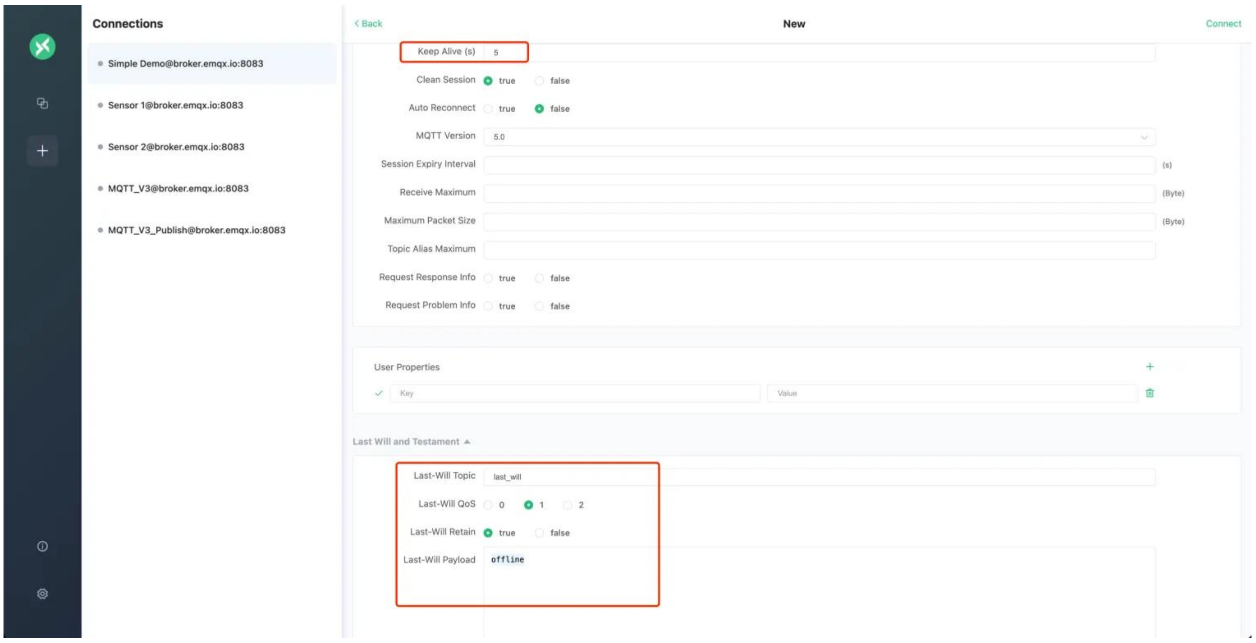
When there is a half-connection within the broker, and when the corresponding client initiates a reconnection or a new connection, the broker will start the client takeover mechanism: it closes the old half-connection and establishes a new connection with the client.

This mechanism ensures that the client will not be prevented from reconnecting due to a half-connection problem.

Keep Alive & Will Message

Keep Alive is typically used in conjunction with Will Message, which allow the device to promptly notify other clients in the event of an unexpected offline event.

As shown in the figure, when this client connects, Keep Alive is set to 5 seconds and a will message is set. If the server does not receive any packets from the client within 7.5 seconds (1.5 times the Keep Alive), it will send a will message with a payload of 'offline' to the 'last_will' topic.


Connections
Simple Demo@broker.emqx.io:8083
Sensor 1@broker.emqx.io:8083
Sensor 2@broker.emqx.io:8083
MQTT_V3@broker.emqx.io:8083
MQTT_V3_Publish@broker.emqx.io:8083
< Back
New
Connect

Keep Alive (s)	5
Clean Session	[✓ true] false
Auto Reconnect	true [✓ false]
MQTT Version	5.0
Session Expiry Interval	(s)
Receive Maximum	(Byte)
Maximum Packet Size	(Byte)
Topic Alias Maximum	
Request Response Info	true false
Request Problem Info	true false

User Properties

Key	Value

Last Will and Testament

Last-Will Topic	last_will
Last-Will QoS	0 [✓ 1] 2
Last-Will Retain	[✓ true] false
Last-Will Payload	`offline`

How to Use Keep Alive in EMQX

In EMQX, you can customize the behavior of the Server Keep Alive mechanism through the configuration file. The relevant field is as follows:

zone.external.server_keepalive

Type	Default
integer	–

If this value is not set, the Keep Alive time will be determined by the client at the time it creates a connection.

If this value is set, the broker forces the Server Keep Alive mechanism to be enabled for all connections in that zone and will use that value to override the value in the client connection request.

zone.external.keepalive_backoff

Type	Optional Value	Default
float	> 0.5	0.75

The MQTT protocol requires the broker to assume that the client is disconnected when it does not receive any protocol packets from the client within 1.5 times the Keep Alive time.

In EMQX, we introduced the keepalive backoff factor and exposed this factor through the configuration file in order to allow users to more flexibly control the Keep Alive behavior on the broker side.

After introducing the backoff factor, EMQX calculates the maximum timeout using the following formula:

```
Keepalive * backoff * 2
```

The default value of backoff is 0.75. Therefore, the behavior of EMQX will be fully compliant with the MQTT standard when the user does not modify this configuration.

Refer to the EMQX configuration documentation for more information.

> **Note: Setting Keep Alive for WebSocket connections**
>
> EMQX supports client access via WebSockets. When a client initiates a connection using WebSockets, it only needs to set the Keep Alive value in the client connection parameters. Refer to A Quickstart Guide to Using MQTT over WebSocket.

MQTT Will Message

When the client disconnects, a will message is sent to the relevant subscriber. Will Messages will be sent when:

- I / O error or network failure occurred on the server;
- The client loses contact during a defined heartbeat period;
- The client closes the network connection before sending offline packets;
- The server closes the network connection before receiving the offline packet.

Will messages are usually specified when the client is connected. As shown below, it is set during the connection by calling the `setWill` method of `theMqttConnectOptions` instance. Any client who subscribes to the topic below will receive the will message.

```
//method1
MqttConnectOptions.setWill(MqttTopic topic, byte[] payload, int qos, boolean
retained)
//method2
MqttConnectOptions.setWill(java.lang.String topic, byte[] payload, int qos,
boolean retained)
```

Usage Scenarios

When client A connects, the will message is set to "offline" and client B subscribes to this will topic. When A disconnects abnormally, client B will receive this will message of "offline" to know that client A is offline.

Connect Flag packet field

Bit	7	6	5	4	2	1	0
	User Name Flag	Password Flag	Will Retain	Will QoS	Will Flag	Clean Start	Reserved
byte 8	X	X	X	X	X	X	X

The will message is not sent after the client calls the disconnect method normally.

Will Flag Function

In short, it is the last will (also known as the Testament) that the client has defined in advance and left when it is disconnected abnormally. This will is a topic and a corresponding message pre-defined by the client, which is attached to the variable packet header of CONNECT. In case of abnormal connection of the client, the server actively publishes this message.

When the bit of Will Flag is 1, Will QoS and Will Retain will be read. At this time, the specific contents of Will Topic and Will Message will appear in the message body, otherwise the Will QoS and Will Retain will be ignored.

When the Will Flag bit is 0, Will Qos and Will Retain are invalid.

Command Line Example

Here is an example of Will Message:

1. Sub side ClientID = sub predefined will message:

```
mosquitto_sub --will-topic test --will-payload die --will-qos 2 -t topic -i sub
-h 192.168.1.1
```

2. clientid = alive subscribes to the will topic at 192.168.1.1 (EMQ server)

```
mosquitto_sub -t test -i alive -q 2 -h 192.168.1.1
```

3. Abnormally disconnect the sub end from the server end (EMQ server), and the pub end receives the will message.

Advanced Usage Scenarios

Here's how to use Retained messages with Will messages.

1. The will message of client A is set to "offline", and the topic of the will is set to `A/status` that is the same as the topic of a normal sending status;
2. When client A is connected, send the "Online" Retained message to the topic `A/status`. When other clients subscribe to the topic `A/status`, they obtain the Retained message as "online";
3. When client A disconnects abnormally, the system automatically sends an "offline" message to the topic `A/status`. Other clients that subscribe to this topic will immediately receive an" offline "message; if the will message is set Retained, and when a new client subscribing to the `A/status` topic comes online, the message obtained is“ offline ”.

Conclusion

As we conclude our voyage, we hope you've found this eBook to be an invaluable resource in your journey to master the MQTT protocol. Armed with a deep understanding of MQTT's core principles, its key components, and its advanced features, you're now well-equipped to navigate the intricate world of IoT communication.

Remember, MQTT is not just a protocol; it's a conduit to innovation. Whether you're shaping the future of smart cities, revolutionizing healthcare through connected devices, or crafting the next generation of industrial automation solutions, MQTT will be your steadfast ally.

EMQ is the world's leading software provider of open-source IoT data infrastructure. We are dedicated to empowering future-proof IoT applications through one-stop, cloud-native products that connect, move, process, and integrate real-time IoT data—from edge to cloud to multi-cloud.

Our core product EMQX, the world's most scalable and reliable open-source MQTT messaging platform, supports 100M concurrent IoT device connections per cluster while maintaining 1M message per second throughput and sub-millisecond latency. It boasts more than 20K+ enterprise users, connecting 100M+ IoT devices, and is trusted by over 400 customers in mission-critical IoT scenarios, including well-known brands like HPE, VMware, Verifone, SAIC Volkswagen and Ericsson.

To learn more, please visit: https://www.emqx.com/en