



Mastering MQTT

Your Ultimate Tutorial for MQTT

2024

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?	1
Understanding Key MQTT Components	5
MQTT Publish–Subscribe Pattern	7
Establishing an MQTT Connection	10
Topics and Wildcards	18
Persistent Session and Clean Session	27
QoS	36
Keep Alive	47
Will Message	52
Request / Response	55
User Properties	65
Topic Alias	72
Payload Format Indicator and Content Type	77
Shared Subscriptions	85
Subscription Options	93
Subscription Identifier	105
Message Expiry Interval	112
Maximum Packet Size	118
Reason Code	123
Enhanced Authentication	127
Control Packets	133
Conclusion	140

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](#).

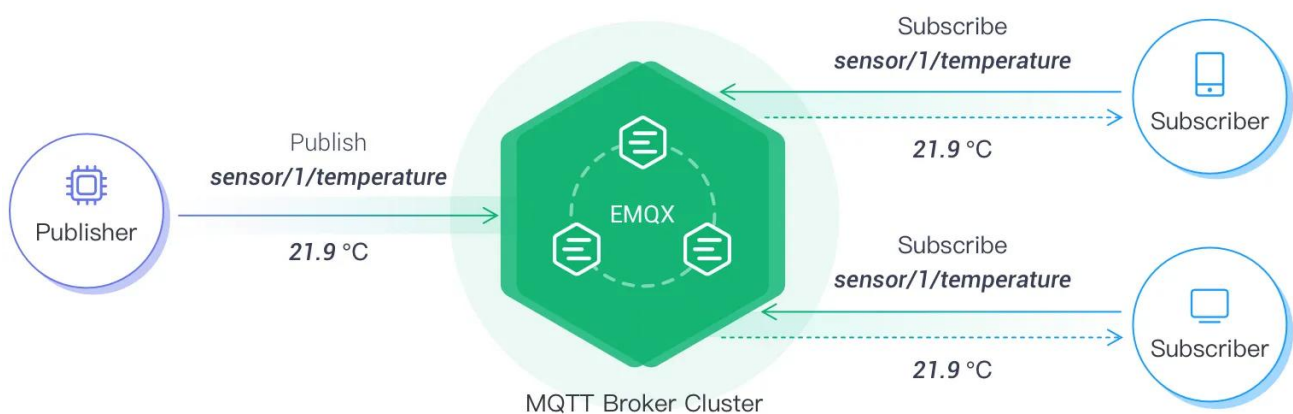
- **Broker**

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 `mqtt://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, `mqtt`s 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.

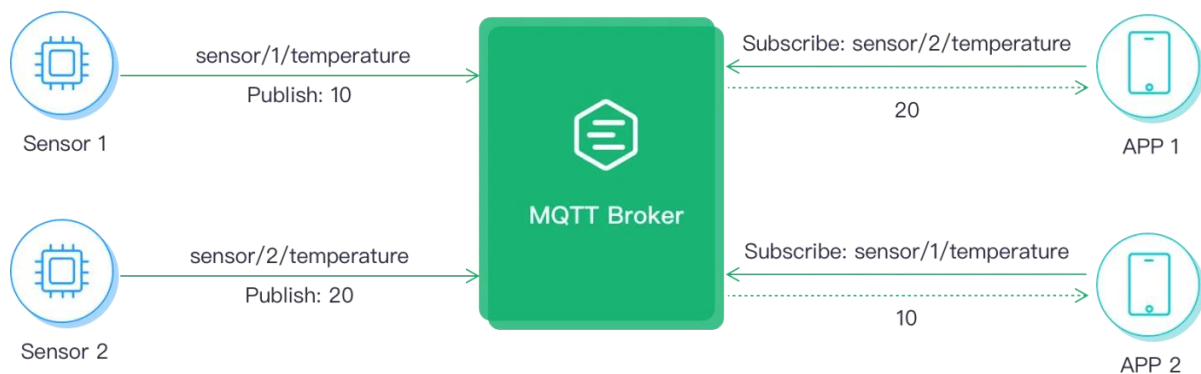
<https://github.com/mqttjs/MQTT.js/issues/1515>

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
<code>\$SYS/brokers</code>	EMQX cluster node list
<code>\$SYS/brokers/\${node}/version</code>	EMQX Broker version
<code>\$SYS/brokers/\${node}/uptime</code>	EMQX Broker startup time
<code>\$SYS/brokers/\${node}/datetime</code>	EMQX Broker time
<code>\$SYS/brokers/\${node}/sysdescr</code>	EMQX Broker description

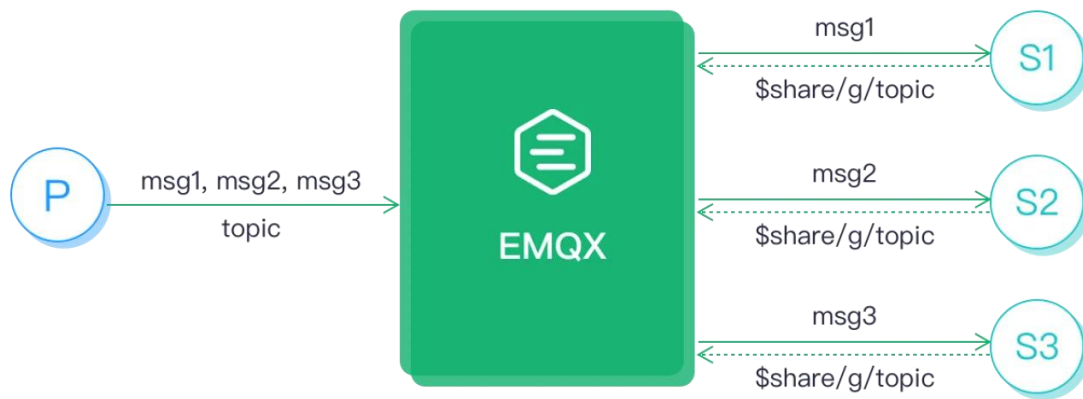
EMQX also supports rich system topics such as client online/offline events, statistics, system monitoring and alarms. For more details, please see the [EMQX System Topics](#) documentation.

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/#.

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:

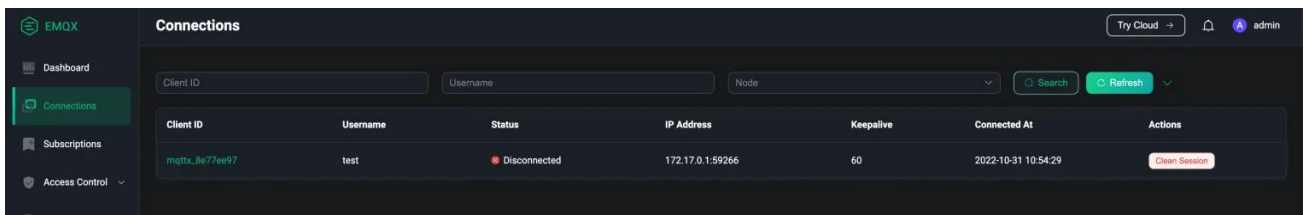
- Whether the session exists, even if the rest of the session status is empty.
- QoS 1 and QoS 2 messages that have been sent to the client but have not yet completed acknowledgment.
- QoS 0 messages (optional), QoS 1 and QoS 2 messages that are waiting to be transmitted to the client.
- QoS 2 messages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

Using MQTT Clean Session

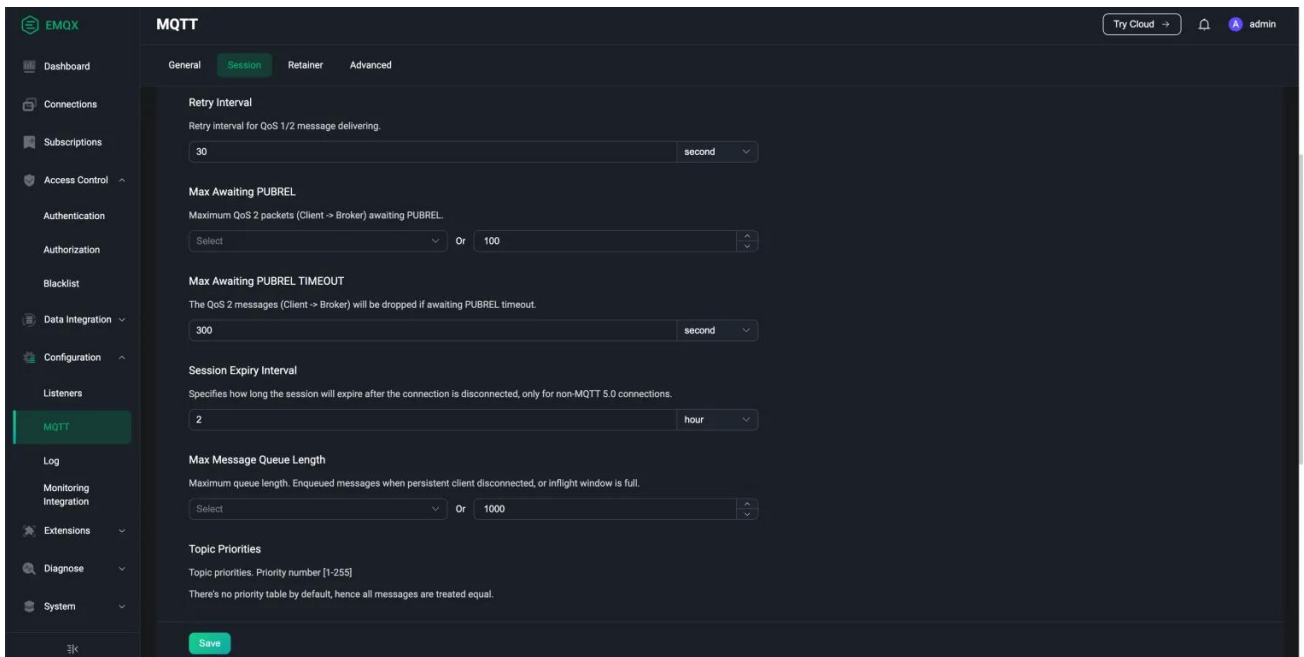
Clean Session is a flag bit used to control the life cycle of the session state. A value of 1 means that a brand new session will be created on connection, and the session will be automatically destroyed when the client disconnects. If it is 0, it means that it will try to reuse the previous session 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.



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

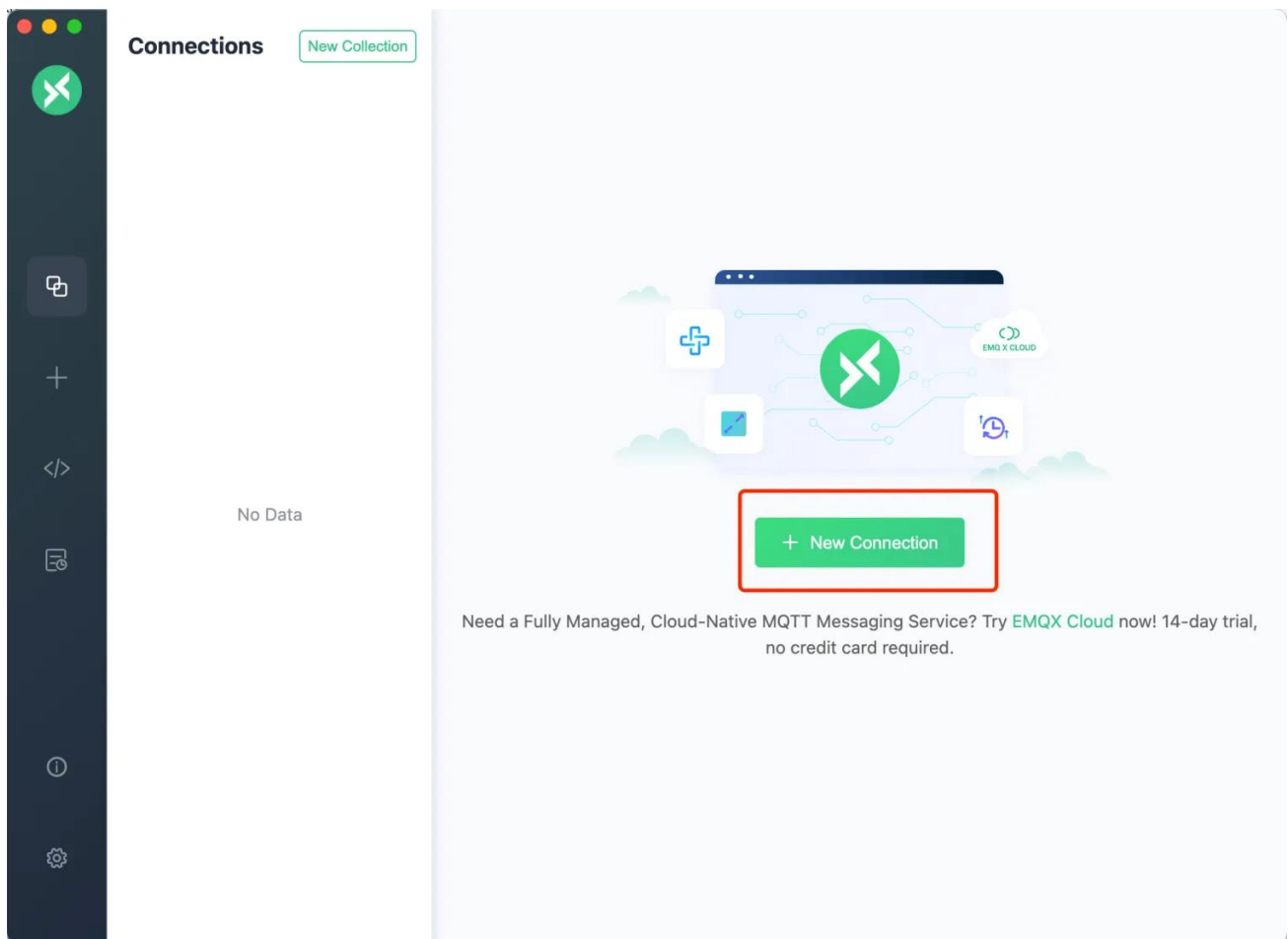


MQTT 3.1.1 does not specify when a Persistent Session will expire; if understood at the protocol level alone, this Persistent Session should be permanent. However, this is not practical in a real-world scenario because it takes up a lot of resources on the server side. So, the server usually does not 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.

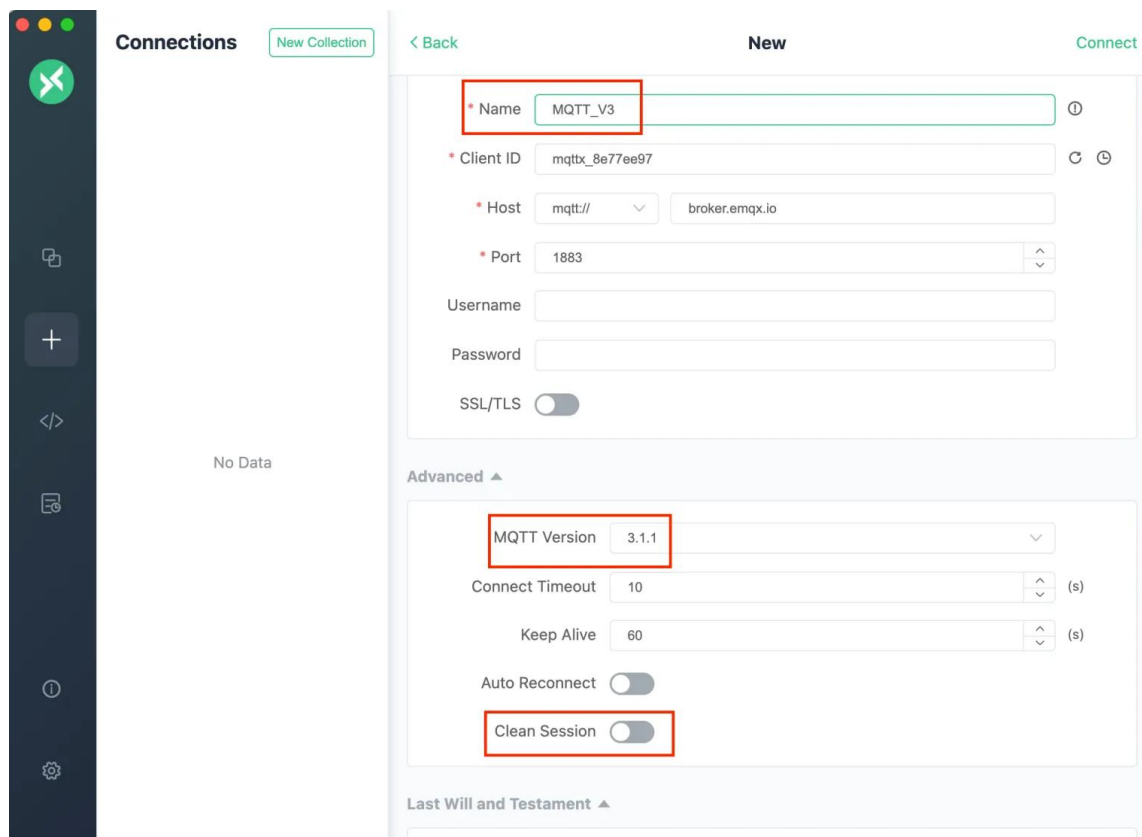
Next, we will demonstrate the use of Clean Session with the open-source cross-platform [MQTT 5.0 desktop client tool – MQTTX](#).

After opening MQTTX, click **New Connection** button to create an [MQTT connection](#) as shown below.

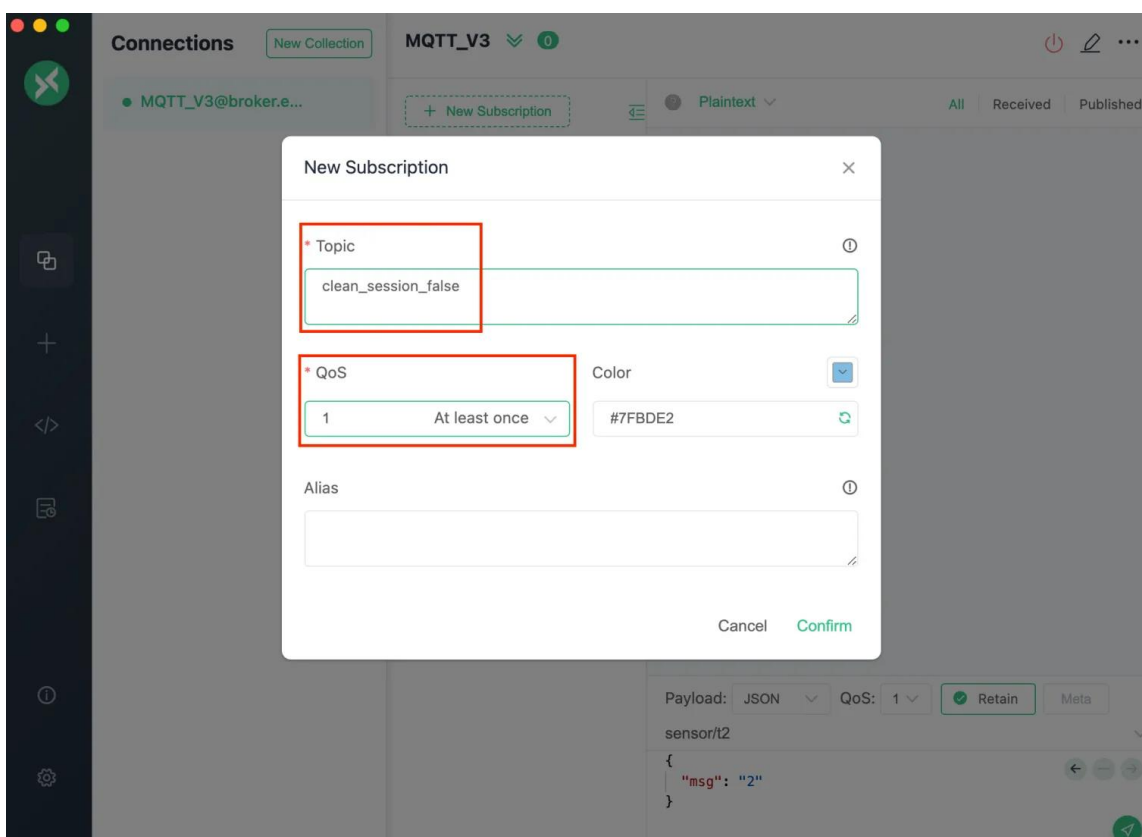


Create a connection named **MQTT_V3** with Clean Session off (i.e., false), and select MQTT version 3.1.1, then click **Connect** button in the upper right corner.

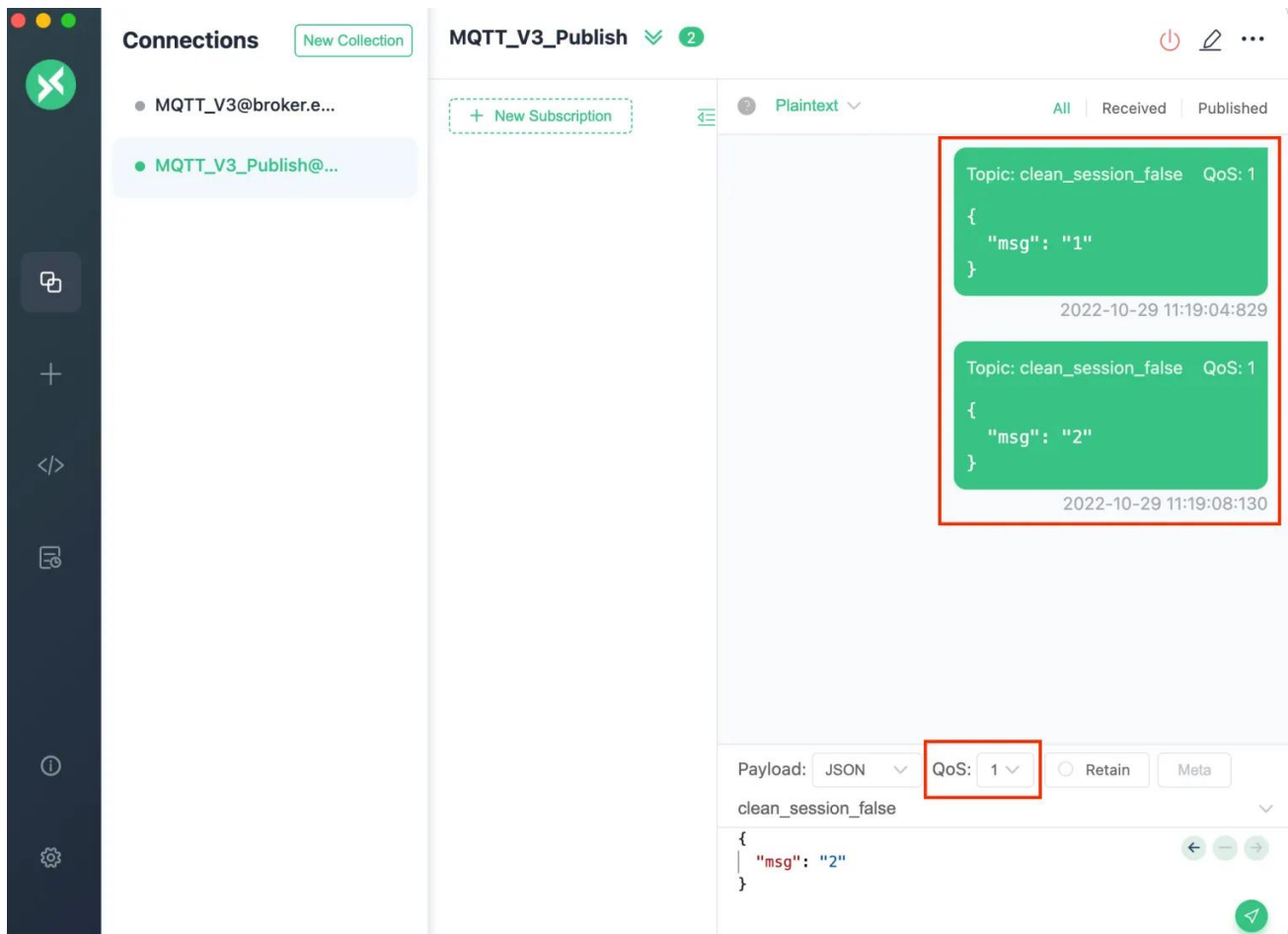
The default server connected to is the [Free Public MQTT Broker](#) provided by EMQ.



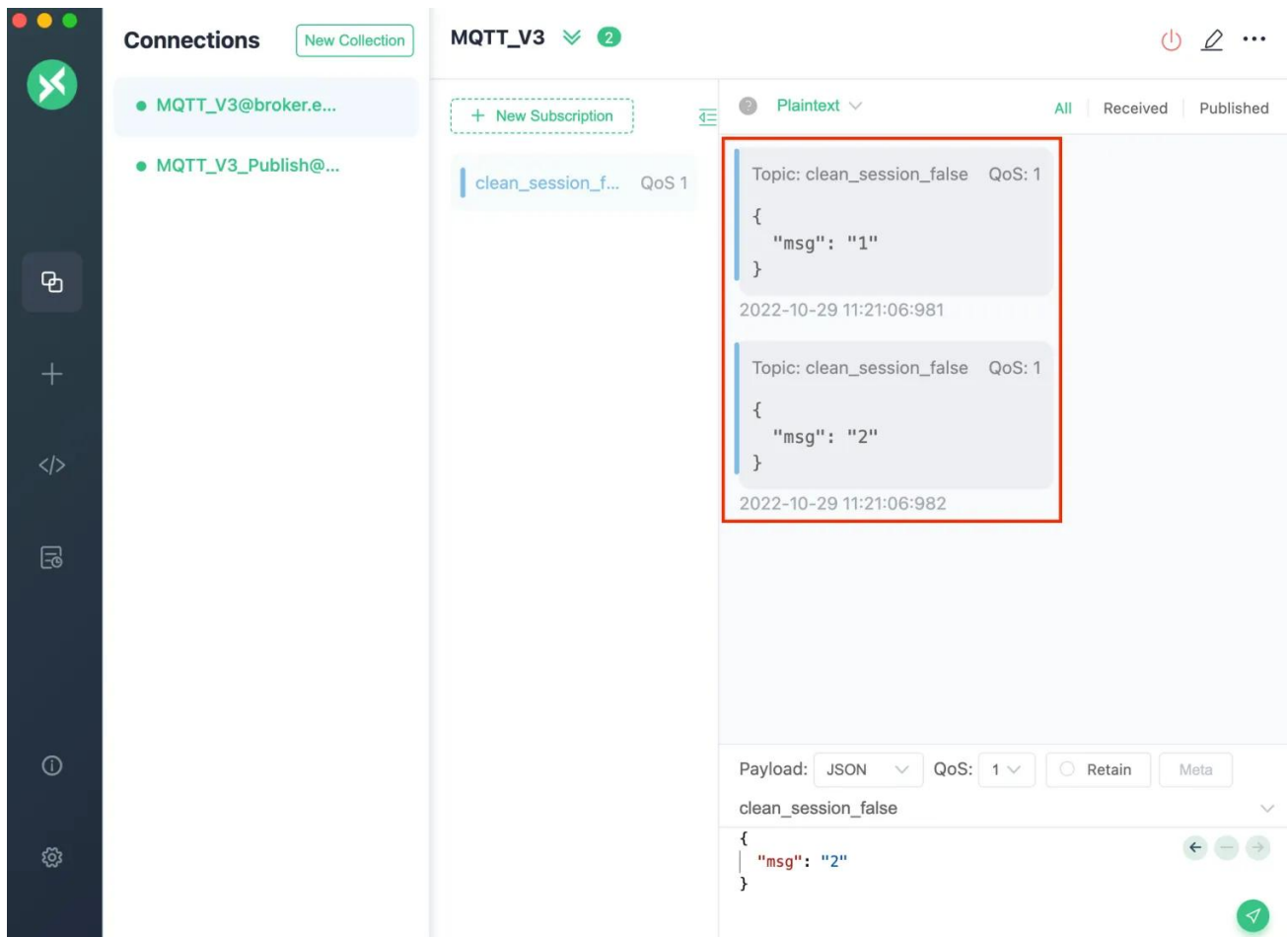
Subscribe to `clean_session_false` topic after a successful connection and QoS is set to 1.



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.



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.



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.

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:

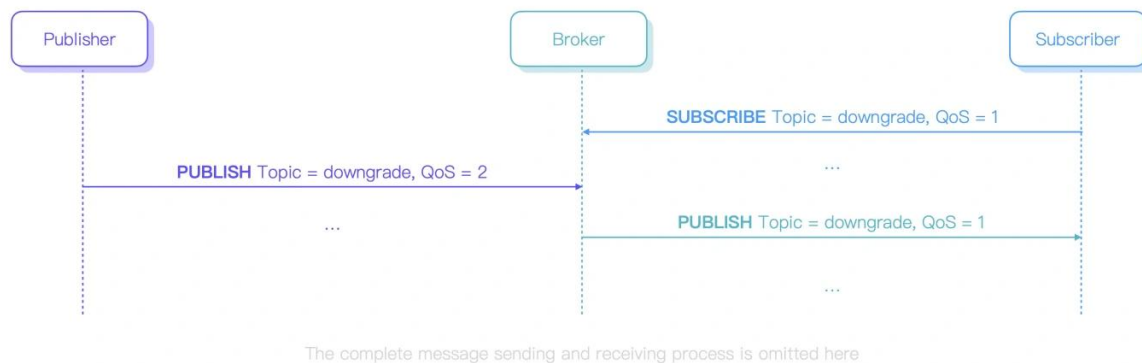
- QoS 0, at most once.
- QoS 1, at least once.
- QoS 2, exactly once.

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.



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.



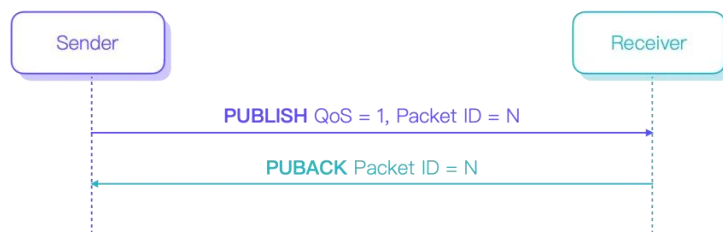
Why Are QoS 0 Messages Lost?

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.



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:



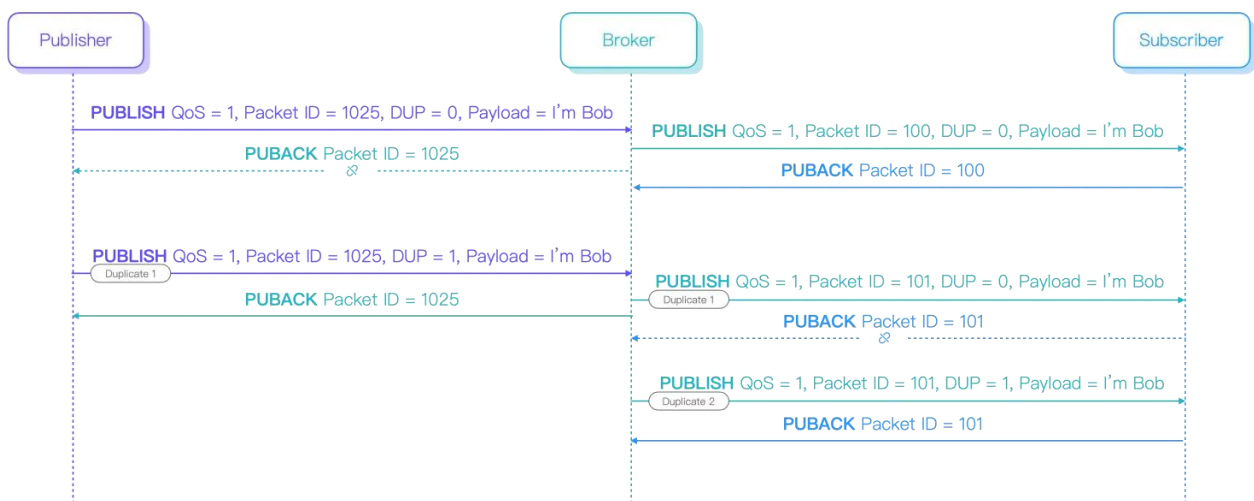
In the first case, the sender retransmits the PUBLISH packet because it did not receive a PUBACK packet. The receiver receives two PUBLISH packets with the same Packet ID and the second PUBLISH packet has a DUP flag of 1. The second packet is indeed a duplicate message.

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.

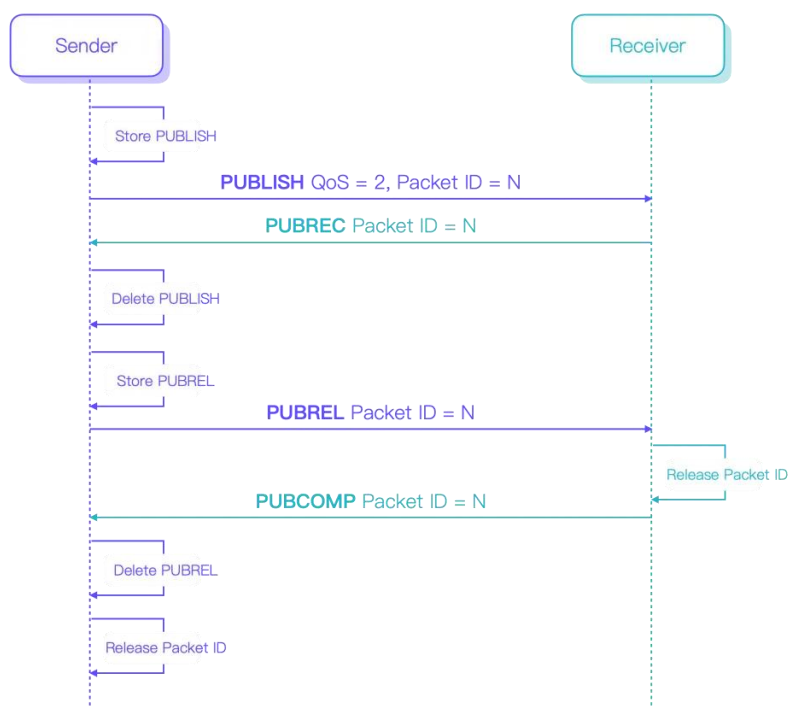
For example, although the publisher only sends one message, the receiver may eventually receive three identical messages.



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.

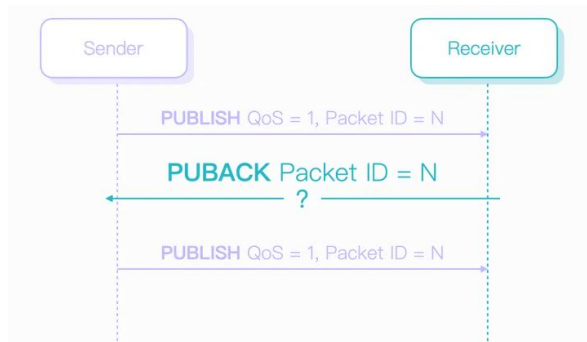
Why Are QoS 2 Messages Not Duplicated?

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.

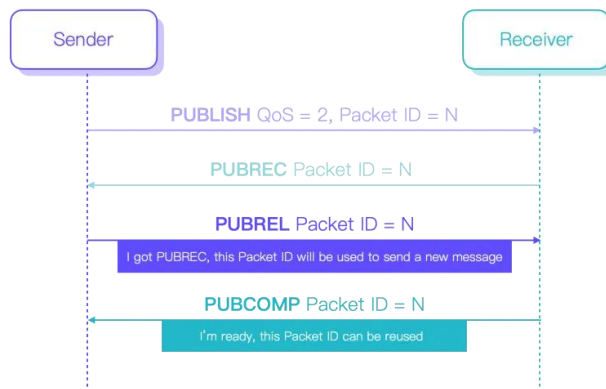
Compared with QoS 1, QoS 2 ensures that messages are not duplicated by adding a new process involving the PUBREL and PUBCOMP packets.

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

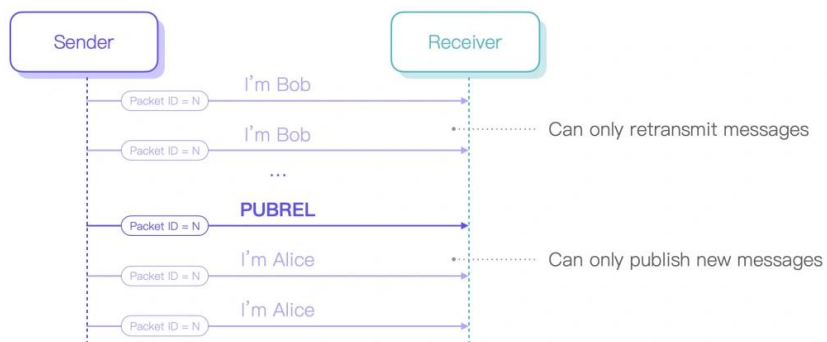
When we use QoS 1, for the receiver, the Packet ID becomes available again after the PUBACK packet is sent, regardless of whether the response has reached the sender. This means that the receiver cannot determine whether the PUBLISH packet it receives later, with the same Packet ID, is a retransmission 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.



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.



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 themselves 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.

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.

The screenshot displays the MQTT broker's configuration interface for a new client. On the left, a sidebar shows a list of connections, including 'Simple Demo@broker.emqx.io:8083' and several sensor topics. The main area is titled 'New' and contains various configuration fields. A red box highlights the 'Keep Alive (s)' field, which is set to 5. Below this, there are fields for 'Clean Session' (true), 'Auto Reconnect' (false), 'MQTT Version' (5.0), 'Session Expiry Interval' (s), 'Receive Maximum' (Byte), 'Maximum Packet Size' (Byte), 'Topic Alias Maximum', 'Request Response Info' (true), and 'Request Problem Info' (true). The 'User Properties' section is also visible. At the bottom, the 'Last Will and Testament' section is expanded, showing a red box around the 'Last-Will Topic' (last_will), 'Last-Will QoS' (1), 'Last-Will Retain' (true), and '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:

$$\text{Keepalive} * \text{backoff} * 2$$

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](#).

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 the `MqttConnectOptions` 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".

Request / Response

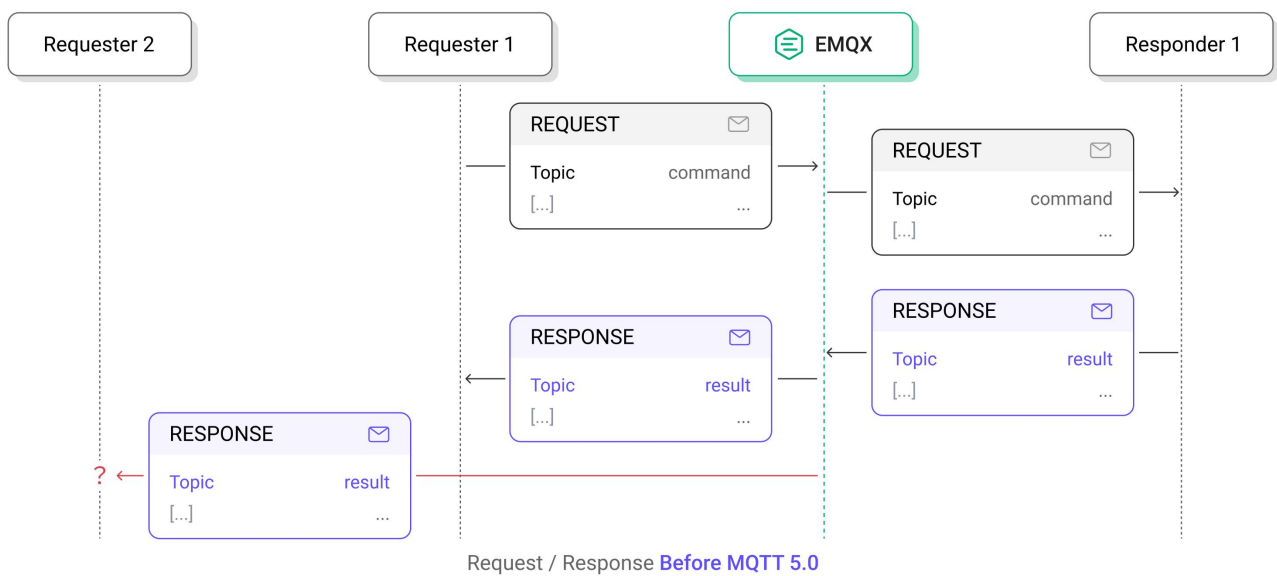
Request / Response Before MQTT 5.0

The publisher–subscriber mechanism of MQTT completely decouples the sender and receiver of messages, allowing messages to be delivered asynchronously. However, this also brings a problem: even with QoS 1 and 2 messages, the publisher can only ensure that the message reaches the server, but cannot know whether the subscriber has ultimately received the message. When executing some requests or commands, the publisher may want to know the execution result of the other end.

The most direct way is to have the subscriber return a response of the request.

In MQTT, this is not difficult to implement. It only requires the two communicating parties to negotiate the request topic and response topic in advance, and then the subscriber returns a response to the response topic after receiving the request. This is also the method generally adopted by clients before MQTT 5.0.

In this scheme, the response topic must be determined in advance and cannot be flexibly changed. When there are multiple different requestors, since they can only subscribe to the same response topic, all requestors will receive the response, and **they cannot tell whether the response belongs to themselves:**



Multiple requestors can easily cause response confusion

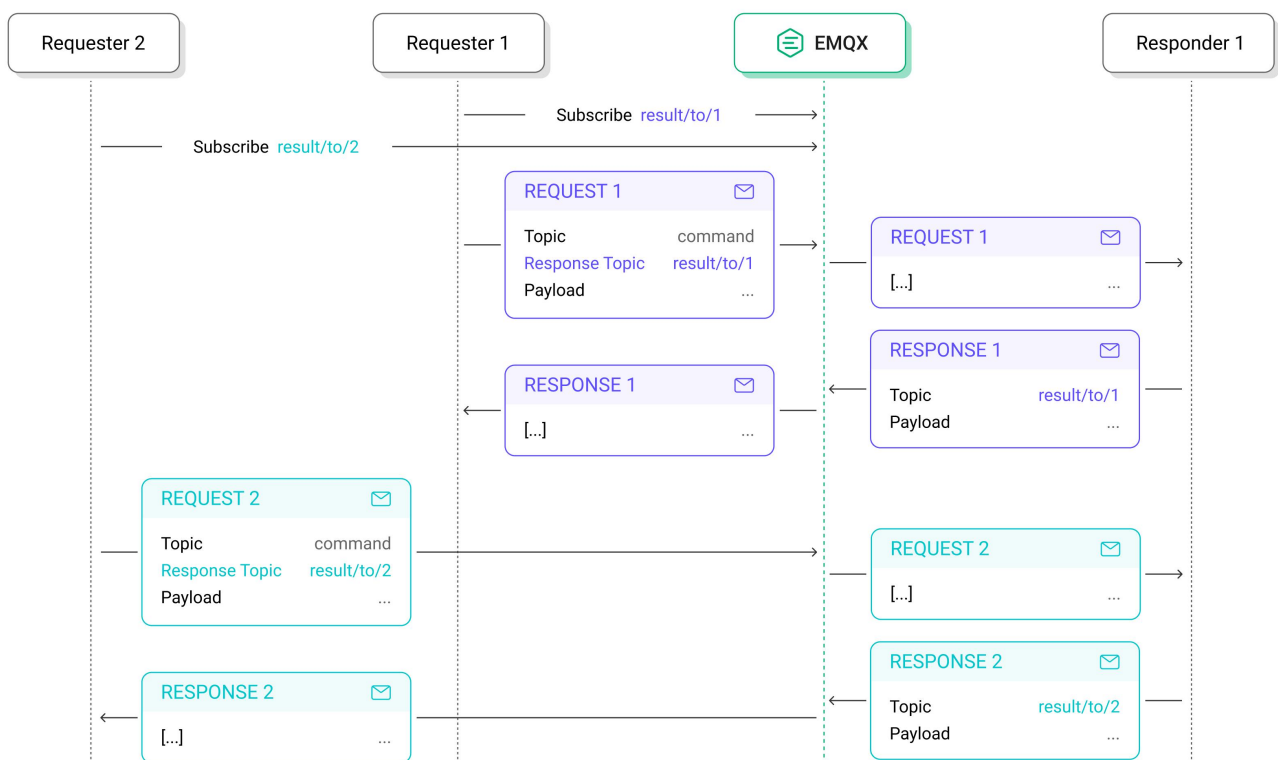
Although there are many ways to avoid this issue, it also leads to the possibility of completely different implementations among vendors, greatly increasing the difficulty and workload for users when integrating devices from different manufacturers.

To solve these problems, MQTT 5.0 introduced properties such as Response Topic, Correlation Data, and Response Information to standardize the **Request / Response** pattern in MQTT.

How Does MQTT 5.0 Request / Response Work?

Property 1 – Response Topic

In MQTT 5.0, the requester can specify an expected Response Topic in the request message. After taking appropriate action based on the request message, the responder publishes a response message to the Response Topic carried in the request. If the requester has subscribed to that Response Topic, it will receive the response.



The requester can use its Client ID as part of the Response Topic, effectively avoiding conflicts caused by different requesters inadvertently using the same Response Topic.

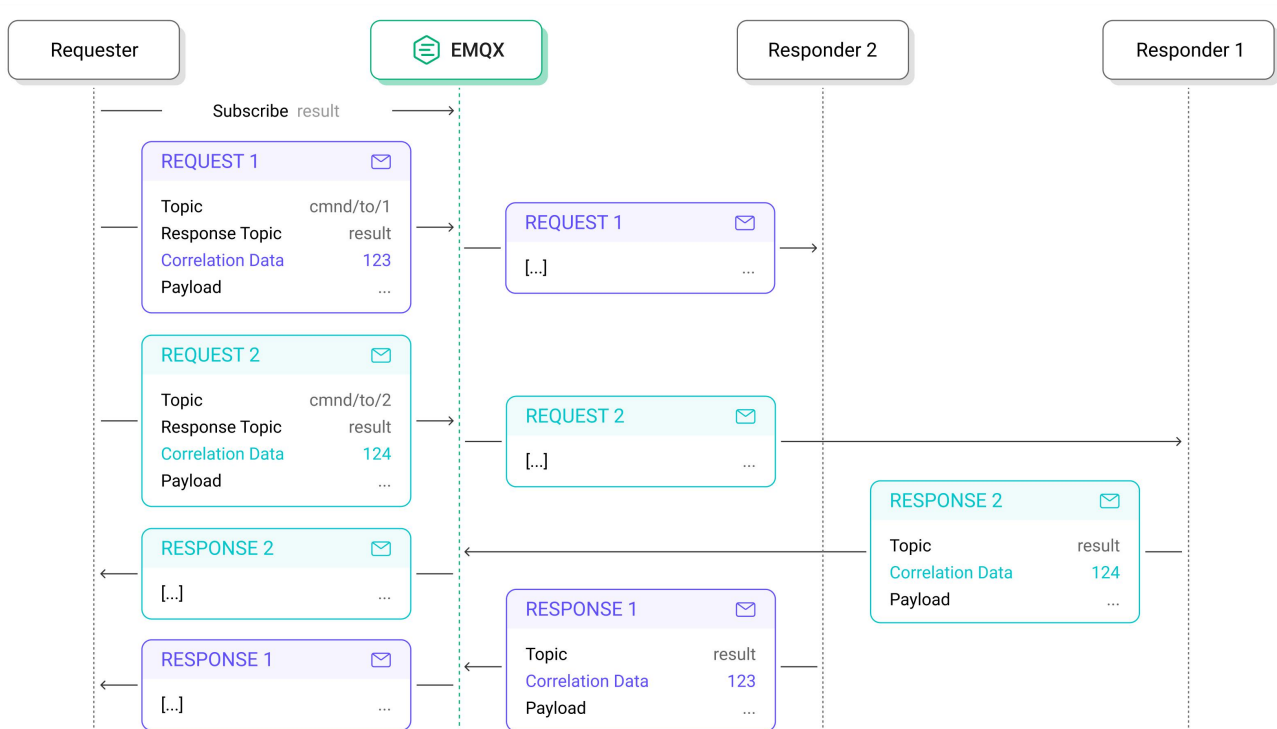
Property 2 – Correlation Data

The requester can also carry Correlation Data in the request, and the responder must return the Correlation Data intact in the response, allowing the requester to identify the original request to which the response belongs.

This can prevent the requester from incorrectly associating the response with the original request when the responder does not return responses in the order of requests, or when a response (QoS 0) is lost due to network disconnection.

On the other hand, the requester may need to interact with multiple responders, such as controlling various smart devices in the home via a mobile phone. The Correlation Data allows the requester to manage responses asynchronously returned from multiple

responders by subscribing to a single Response Topic.

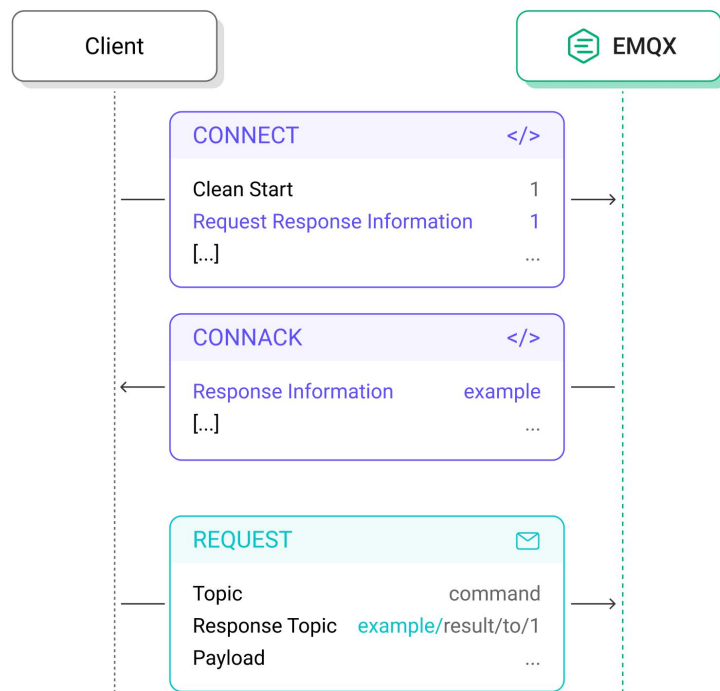


In the above **Request / Response** process, the [MQTT broker](#) does not change the Response Topic or Correlation Data, it only serves as a forwarding agent.

Property 3 – Response Information

For security reasons, the MQTT server usually restricts the topics that clients can publish and subscribe to. The requester can specify a random Response Topic, but cannot guarantee that it has permission to subscribe to that topic, nor can it guarantee that the responder has permission to publish messages to that Response Topic.

Therefore, MQTT 5.0 also introduced the Response Information property. By setting the Request Response Information identifier to 1 in the CONNECT packet, the client can request the server to return Response Information in the CONNACK packet. The client can use the content of the Response Information as a specific part of the Response Topic, to pass the server's permission check.



MQTT does not further specify the details of this part, such as the content format of the Response Information and how the client creates the Response Topic based on the Response Information, so different server and client implementations may vary.

For example, the server could use the Response Information “FRONT,mytopic” to indicate both the specific content of a certain part of the Response Topic and its position within the Response Topic. It could also agree with the client on how to use this specific part in advance, then use the Response Information “mytopic” to indicate only the specific content of this part.

Taking a smart home scenario as an example, smart devices will not be used across users. We can let the MQTT server return the ID of the user to whom the device belongs as Response Information, and the client uniformly uses this user ID as the prefix of the Response Topic. The MQTT server only needs to ensure that these clients have the publication and subscription permissions for topics starting with this user ID during the lifecycle of their sessions.

Suggestions for Using MQTT Request / Response

Here are some suggestions for using **Request / Response** in MQTT, following these will help you implement best practices:

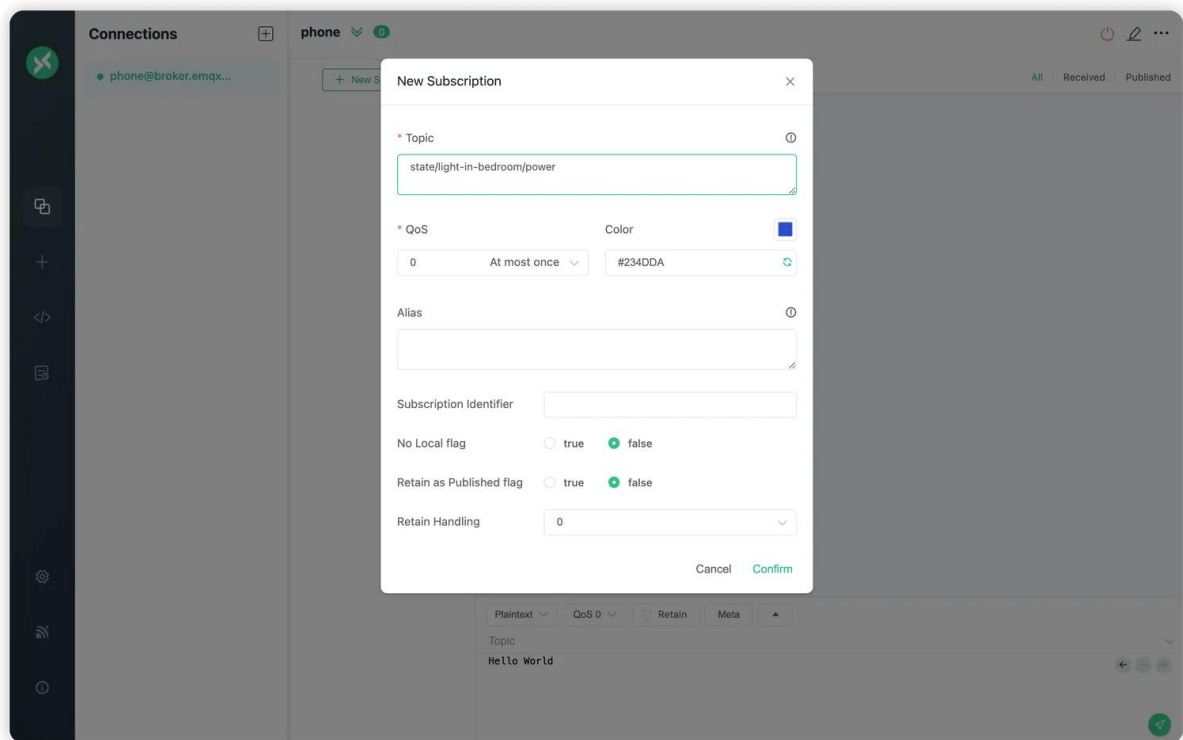
1. QoS 1 and 2 in MQTT can only ensure that messages reach the server. If you want to confirm whether the message has reached the subscriber, you can use the **Request / Response** pattern.
2. Subscribe to the Response Topic before sending the request to avoid missing the response.
3. Ensure that the responder and the requester have the necessary permissions to publish and subscribe to the Response Topic. Response Information can help us build a Response Topic that meets permission requirements.
4. When there are multiple requesters, they need to use different Response Topics to avoid response confusion. Using Client ID as part of the topic is a common practice.
5. When there are multiple responders, it is best for the requester to set Correlation Data in the request to avoid response confusion.
6. We can make the Will Message work with the **Request / Response**. We just need to set the Response Topic for the Will Message when connecting. This can help the client know whether the Will Message has been consumed during its offline period, so it can make appropriate adjustments.

Demo

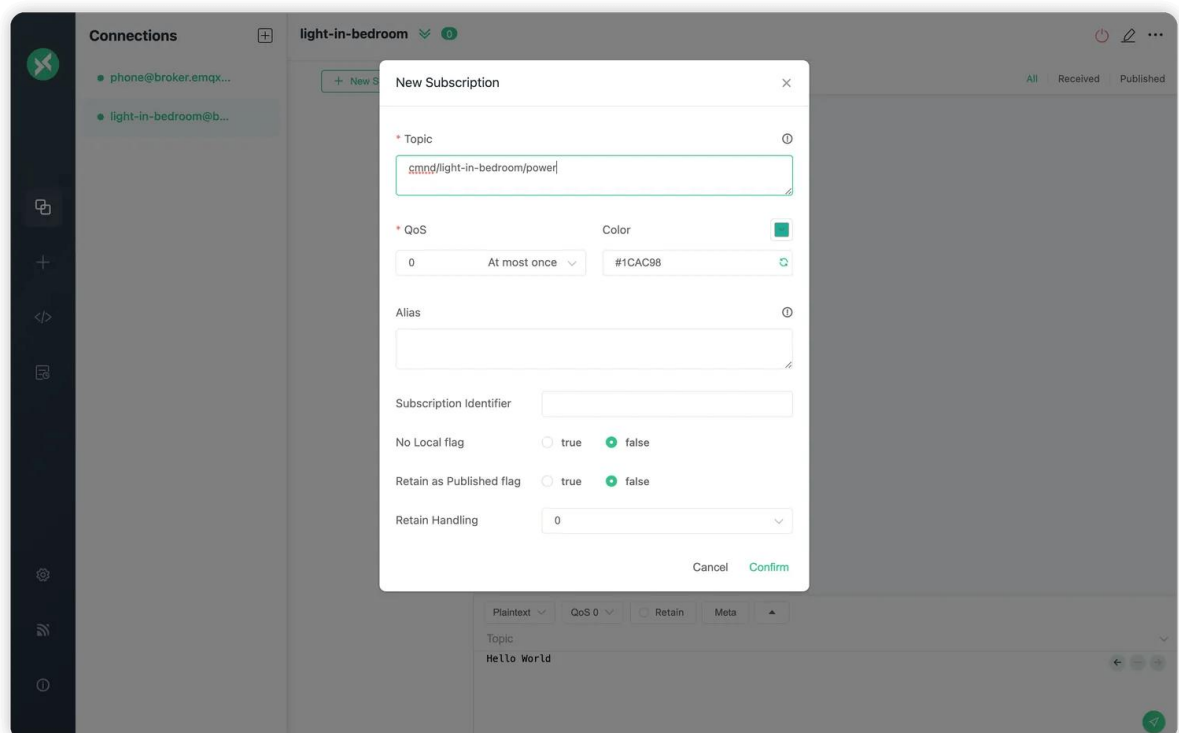
Next, we will use [MQTTX](#) to simulate the scenario of using a mobile phone to remotely control the bedroom light to turn on and receive the response.

Install and open MQTTX, first initiate a client connection to the [public MQTT broker](#) to simulate a mobile phone, and subscribe to the response topic

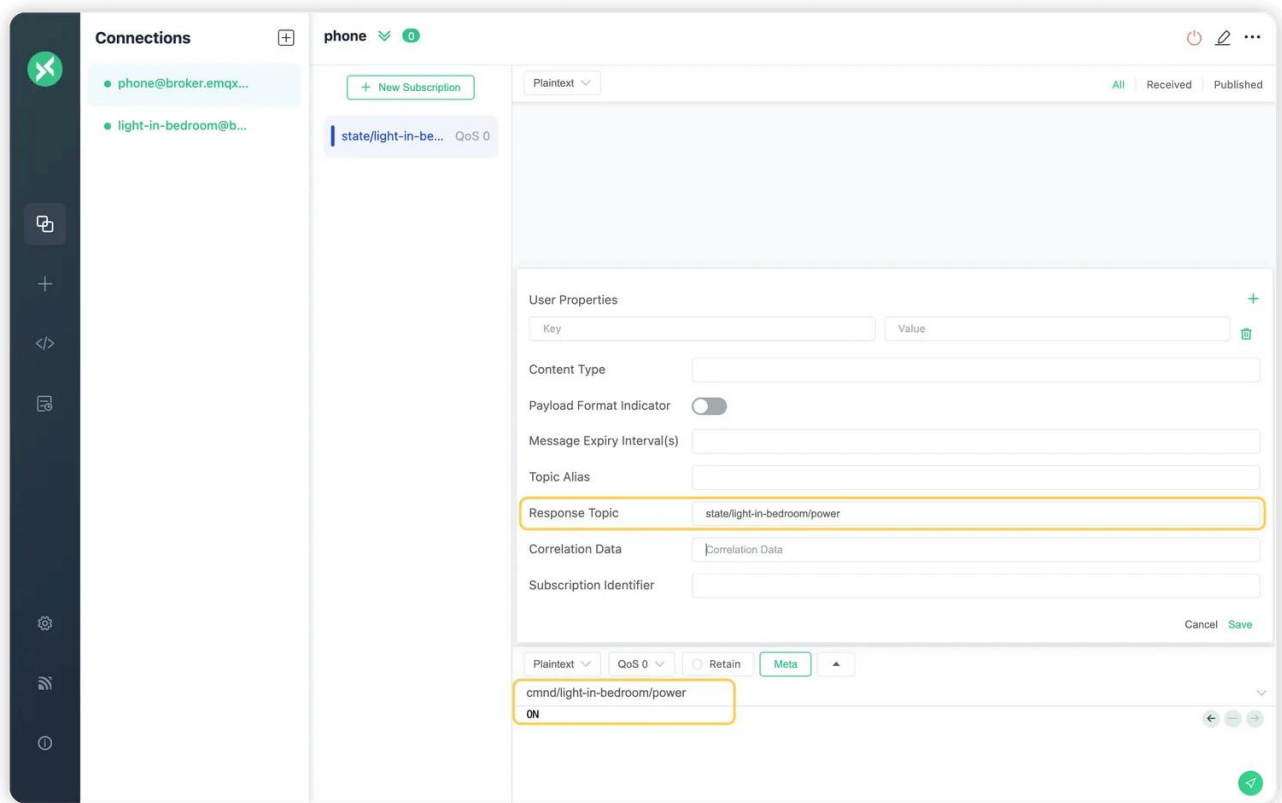
state/light-in-bedroom/power:



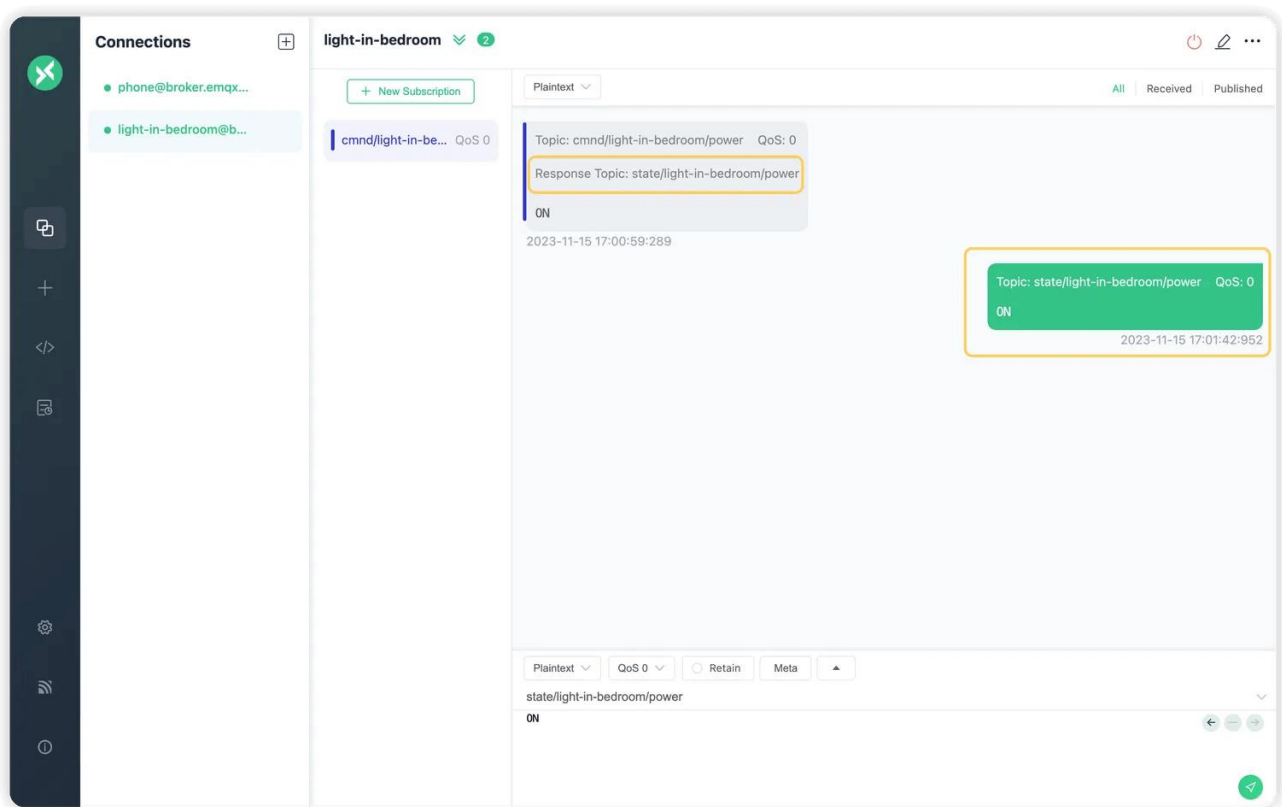
Create a new client connection to simulate the smart light, and subscribe to the request topic `cmnd/light-in-bedroom/power`:



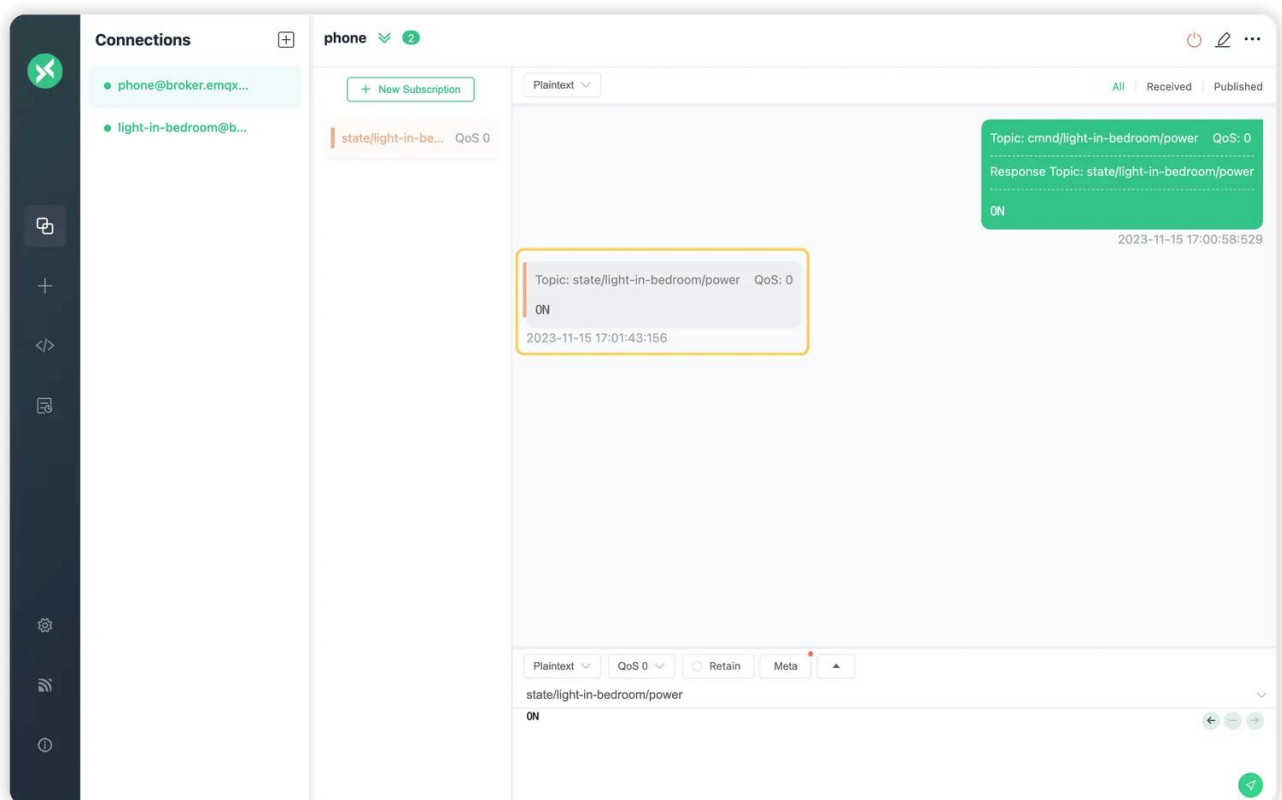
Return to the request client, and send a turn-on light command with the Response Topic to the request topic `cmdnd/light-in-bedroom/power`:



In the response client, we can see that the received message carries the Response Topic. So next, we can perform the turn-on light operation according to the command request, and then return the latest status of the light through this Response Topic:



Eventually, the request client will receive this response, and according to the content of the response message, we can know that the light has been successfully turned on:



This is a very simple example, you can also try to increase the number of publishers or responders, to experience how to design the request and response topics in these cases.

In addition, we provide Python sample code for **Request / Response** in [emqx/MQTT-Features-Example](#), you can use it as a reference.

User Properties

What are User Properties?

User Properties are user-defined properties that allow users to add their metadata to MQTT messages and transmit additional user-defined information to expand more application scenarios.

User Properties consist of a user-defined UTF-8 key/value pair array configured in the message property field. As long as the maximum message size is not exceeded, an unlimited number of user properties can be used to add metadata to the MQTT message and transfer information among publishers, MQTT Brokers, and subscribers.

If you are familiar with HTTP protocol, this function is very similar to the concept of HTTP Header. User Properties allow users to extend the [MQTT protocol](#) effectively and can appear in all messages and responses. Because User Properties are defined by the user, they are only meaningful to the user's action.

Why Do We Need User Properties?

User Properties are used to solve the poor scalability of MQTT 3. When there is the possibility that any information can be transmitted in the message, it ensures that the user can extend the functionality of the standard protocol to meet his own needs.

For message types with different options or configurations, User Properties can be sent between the client and the MQTT Broker or between the clients. For example, when User Properties are configured in the connected client, they can only be received on the MQTT Broker but not on the client. If User Properties is configured when sending a message, they can be received by other clients. The following two types of User Property

configuration are commonly used.

User Properties of the Connected Client

When the client initiates a connection with the MQTT Broker, the Broker can predefine some required metadata information that can be used, which is User Properties. When the connection is successful, the MQTT service can get the relevant information sent by the connection for use. Therefore, the User Properties of the connected client depend on the MQTT Broker.

User Properties During Message Publishing

User Properties during message publishing may be more commonly used because they can transfer metadata information between the clients. For example, you can add some common information when publishing messages, such as message number, timestamp, file, client information, and routing information.

In addition to the above-mentioned User Properties settings, you can also configure User Properties when subscribing to the Topic, unsubscribing, or disconnecting.

Use of User Properties

File Transfer

User Properties of [MQTT 5](#) can be extended for file transfer instead of putting data in the payload of the message body and using key-value pairs for User Properties in the previous MQTT 3. This also means that the file can be kept as binary because the metadata of the file is in the user properties. For example:

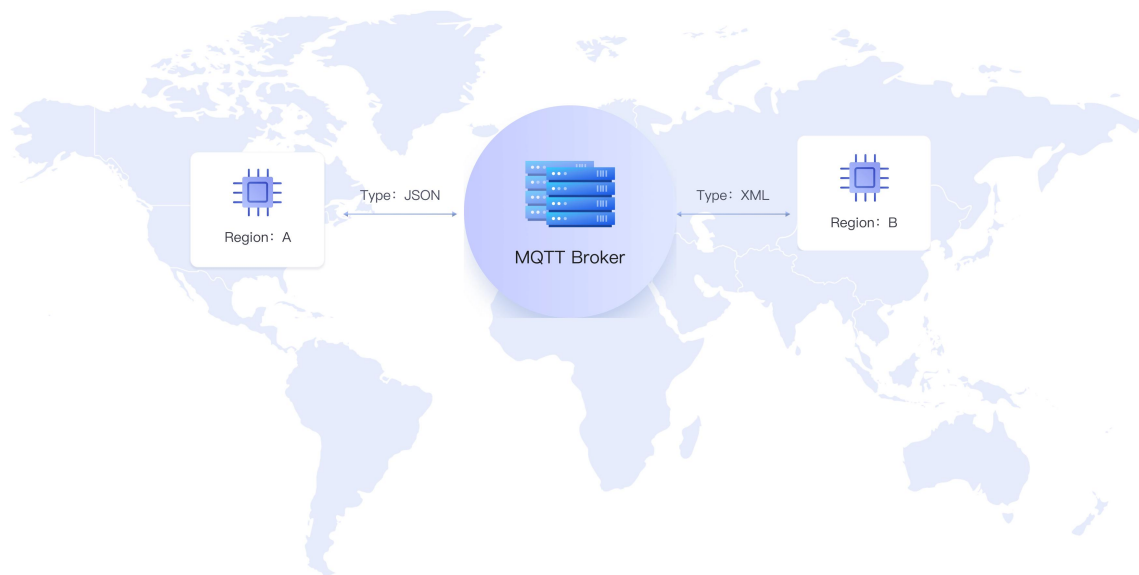
```
{  
  "filename": "test.txt",  
  "content": "xxxx"  
}
```

Resource Analysis

When the client connects to the MQTT Broker, different clients and platforms or systems from different vendors transmit message data in different ways. There may be structural differences in the message data format, and some clients are distributed in different regions. For example, the message format sent by the device in region A is JSON, and that sent by the device in region B is XML. At this time, the server may need to judge and compare one by one after receiving the message to find an appropriate parser for data analysis.

To improve efficiency and reduce computing load, we can use the User Properties function to add data format information and geographic information. When the server receives the message, it can use the metadata in the User Properties to analyze the data. Moreover, when the client subscription of area A receives the client message from area B, it can also quickly know which area the specific message comes from so that the message is traceable.

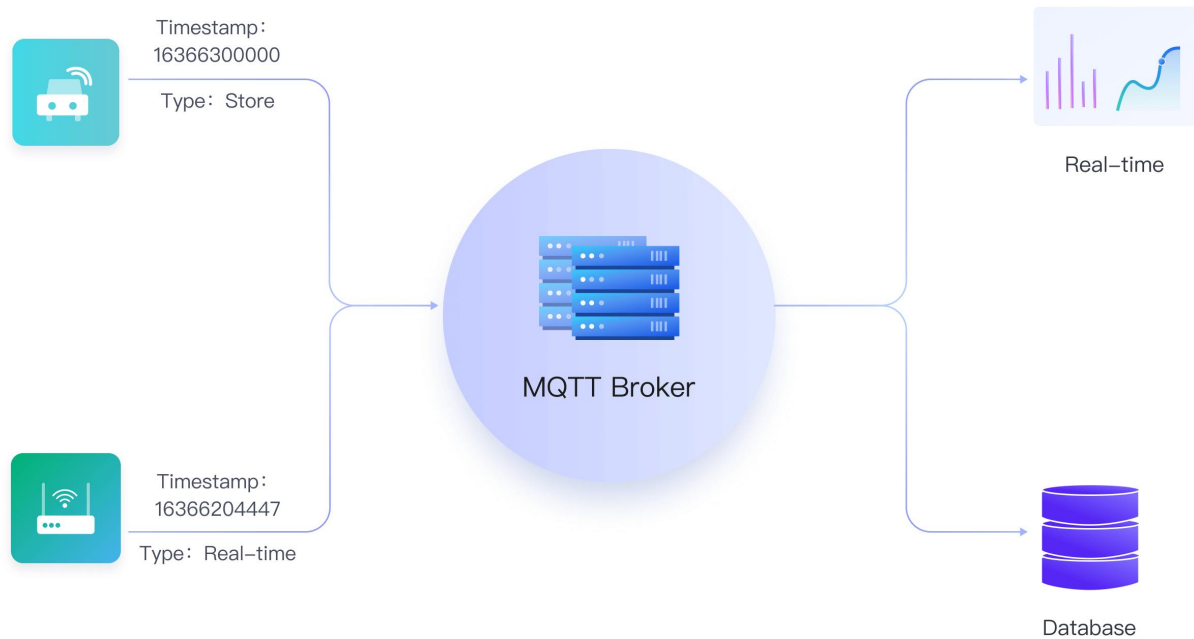
```
{  
  "region": "A",  
  "type": "JSON"  
}
```



Message Routing

We can also use User Properties to do application-level routing. As mentioned above, there are different systems and platforms, and there are different devices in each area. Multiple systems may receive messages from the same device. Some systems need to display data in real-time, and another system may store these data in time series. Therefore the MQTT server can determine whether to distribute the message to the system storing the message or to the system presenting the data by the User Properties configured in the reported message.

```
{
  "type": "real-time",
  "timestamp": 1636620444
}
```



How to Configure in MQTT Client

Let's take an example of programming with JavaScript, using the [MQTT.js](#) client. We can first specify the version of MQTT as MQTT 5.0 when connecting to the client.

Connect

When connecting, we set the User Properties in the properties options and add the type and region properties. After the connection is successful, the MQTT Broker will receive this user-defined message.

```
// connect options
const OPTIONS = {
  clientId: 'mqtt_test',
  clean: true,
  connectTimeout: 4000,
  username: 'emqx',
  password: 'public',
  reconnectPeriod: 1000,
```

```

    protocolVersion: 5,
    properties: {
      userProperties: {
        region: 'A',
        type: 'JSON',
      },
    },
  },
}

const client = mqtt.connect('mqtt://broker.emqx.io', OPTIONS)

```

Publish Messages

After the connection is successful, we subscribe and choose to publish the message and set the User Properties in the configuration of publishing messages. Then, we monitor the message reception. In the publish function, we configure the User Properties. We print the packet to see the User Properties configured just now in the function that listens to and receives the message.

```

client.publish(topic, 'nodejs mqtt test', {
  qos: 0,
  retain: false,
  properties: {
    userProperties: {
      region: 'A',
      type: 'JSON',
    },
  },
}, (error) => {
  if (error) {
    console.error(error)
  }
})

```

```
client.on('message', (topic, payload, packet) => {  
  console.log('packet:', packet)  
  console.log('Received Message:', topic, payload.toString())  
})
```

At this point, we can see that the User Properties configured just before publishing have been printed and output in the console.

Topic Alias

What is Topic Alias

Topic Alias allows users to reduce the possibly long and repeatedly used topic name to a 2-byte integer, so as to reduce the bandwidth consumption when publishing messages.

This 2-byte integer is encoded as an attribute field in the variable header of the PUBLISH packet. And it's limited by [Topic Alias Maximum](#) exchanged between client and broker in CONNECT and CONNACK packets. As long as this limit is not exceeded, any topic name can be reduced to one integer.

Why Use Topic Alias

In MQTT v3, if the client needs to publish a large number of messages to the same topic (over the same MQTT connection), the topic name will be repeated in all PUBLISH packets, which causes a waste of bandwidth resources. At the same time, parsing the UTF-8 string of the same topic name every time is a waste of computing resources for the server.

As an example, a sensor reports temperature and humidity at a fixed frequency from location A: It publishes to topic `/location/A/temperature` (23 bytes) for each temperature message. It also publishes to topic `/location/A/humidity` (20 bytes) for the humidity messages. Without topic aliases, not only for the first published message, each subsequent PUBLISH packet needs to carry the topic names (53 bytes in total) through the connection over and over again. And for the broker, it'll have to parse the location topics repeatedly.

As we can see, the purpose of introducing topic alias feature in MQTT 5.0 is mainly to reduce resource consumption, in both network resources and CPU resources.

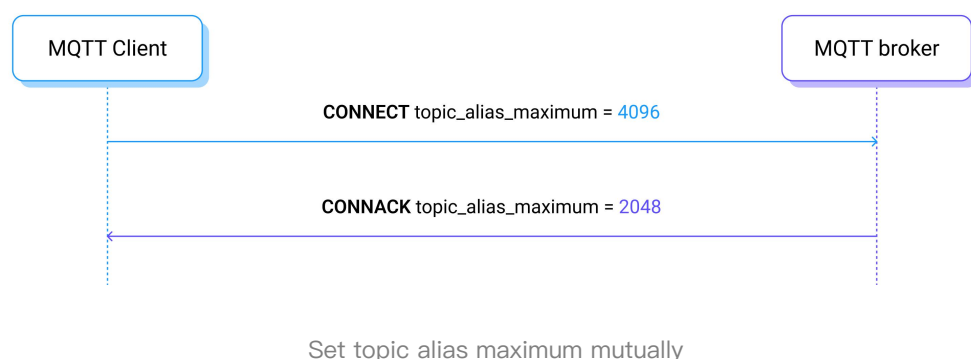
How to Use Topic Aliases

Topic Alias Life Cycle and Scope

Topic aliases are managed respectively by the client and server, and the life cycle and scope are limited to the current connection. When the topic aliases needs to be used again after disconnected, the mapping relationship of `topic alias <=> topic name` has to be rebuilt.

Topic Alias Maximum

Before the MQTT client or server can start using topic aliases, they need to agree on the maximum number of topic aliases allowed in the current connection. This part of the information exchange is done in the `CONNECT` packet and the `CONNACK` packet. The `Topic Alias Maximum` is encoded in the variable headers of the `CONNECT` and `CONNACK` packets as a message attribute.



`Topic Alias Maximum` in the client's `CONNECT` packet indicates the maximum number of topic alias that the server can use in this connection. Similarly, in the `CONNACK` packet

sent by the server, the value indicates the maximum number of topic alias that can be used by the opposite end (client) in the current connection.

A topic alias may range from 1 to Topic Alias Maximum. Setting Topic Alias Maximum to 0 for the peer to prohibit the usage of topic alias.

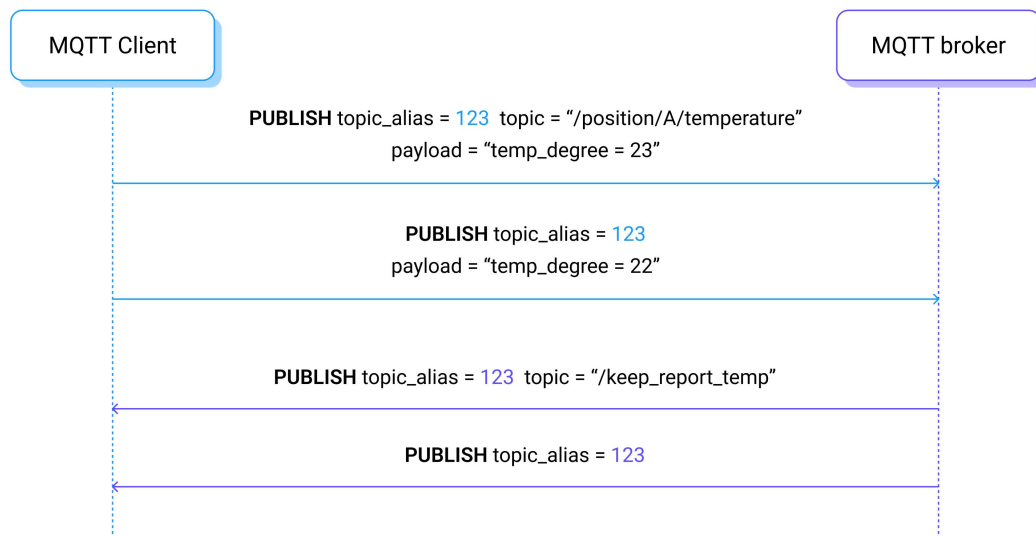
Creating and Using a Topic Alias

When a client (or server) sends a PUBLISH packet, it can use a one-byte identifier with a value of 0x23 in the attribute of the variable header to indicate that the next 2 bytes being the number of topic alias.

However, the topic alias number is not allowed to be 0, nor is it allowed to be greater than the Topic Alias Maximum set in the CONNACK (CONNECT) packet sent by the server (client).

After receiving a PUBLISH packet with a topic alias and a non-empty topic name, the receiving end will establish a mapping relationship between the topic alias and the topic name. In the PUBLISH packets sent after this, the topic alias with a length of 2 bytes can be used to publish a message, and the receiving end will use the previously built mapping relationship of topic alias \leftrightarrow topic name to find the topic for the messages.

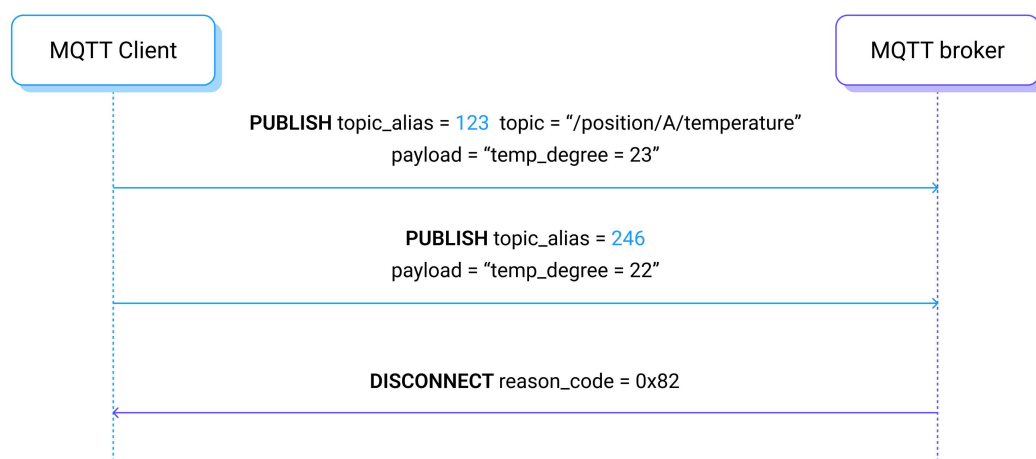
Since such mapping is maintained on both ends respectively (i.e. does not have to be identical), the client and the server can publish to different topics using the same alias number.



MQTT client and broker manage their aliases respectively

Using Unknown Topic Alias

If a topic alias used in the **PUBLISH** packet is not created previously, that is, the receiving end has not built the mapping relationship between the current topic alias and a topic name, and the topic name field in the variable header of this message is empty, the receiving end should send a **DISCONNECT** packet with a **REASON_CODE** of **0x82** to close the connection.

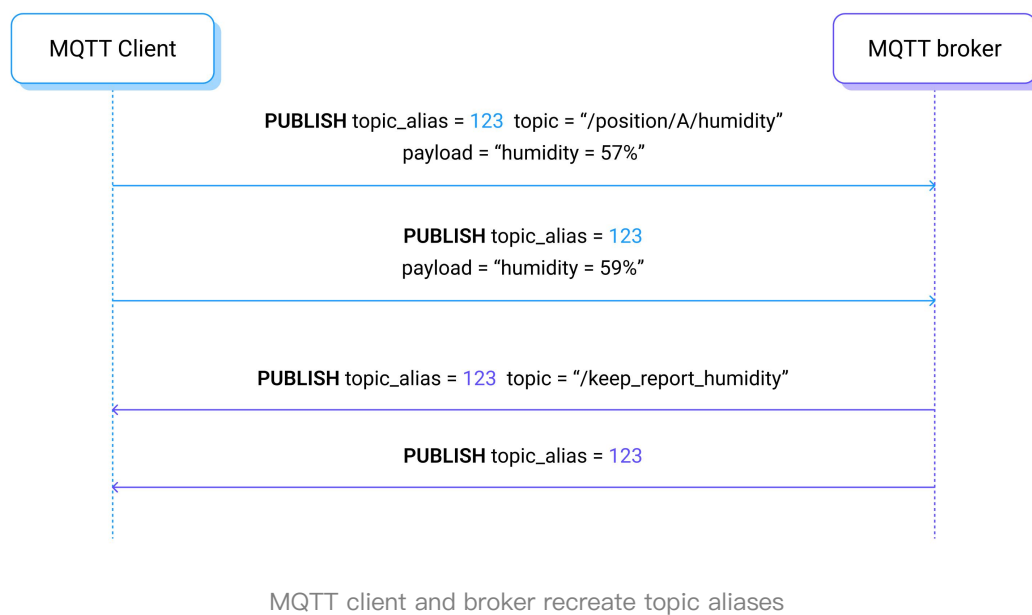


Unknown topic alias

Recreating Topic Alias

An already built alias to topic name mapping can be re-built with a new **PUBLISH** packet with a topic alias and a non-empty topic name.

As an example in the following diagram, the topic alias **123** which is previously used for the temperature topic now is updated to represent the humidity topic.



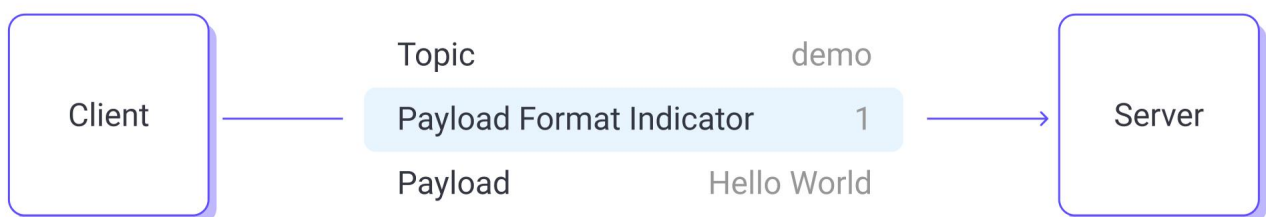
Payload Format Indicator and Content Type

What is Payload Format Indicator?

The Payload Format Indicator is a new property introduced in MQTT 5.0 to indicate the format of the payload in [MQTT packets](#). However, the format of the payload in CONNECT, SUBSCRIBE, and UNSUBSCRIBE packets is fixed, so in practice, only PUBLISH and CONNECT packets need to declare the payload format.

If the value of the Payload Format Indicator is 0 or if this property is not specified, the current payload is an unspecified byte stream; if the value of this property is 1, the current payload is UTF-8 encoded character data.

This allows the receiver to check the format of the payload without having to parse the specific content. For example, the server can check if the payload is a valid UTF-8 string to avoid distributing incorrectly formatted application messages to subscribers. However, given the burden this operation imposes on the server and the benefits that can actually be achieved, this is usually an optional setting.



What is Content Type?

Content Type is also a new property introduced in MQTT 5.0, and similar to the Payload Format Indicator, it is also only available in PUBLISH and CONNECT packets.

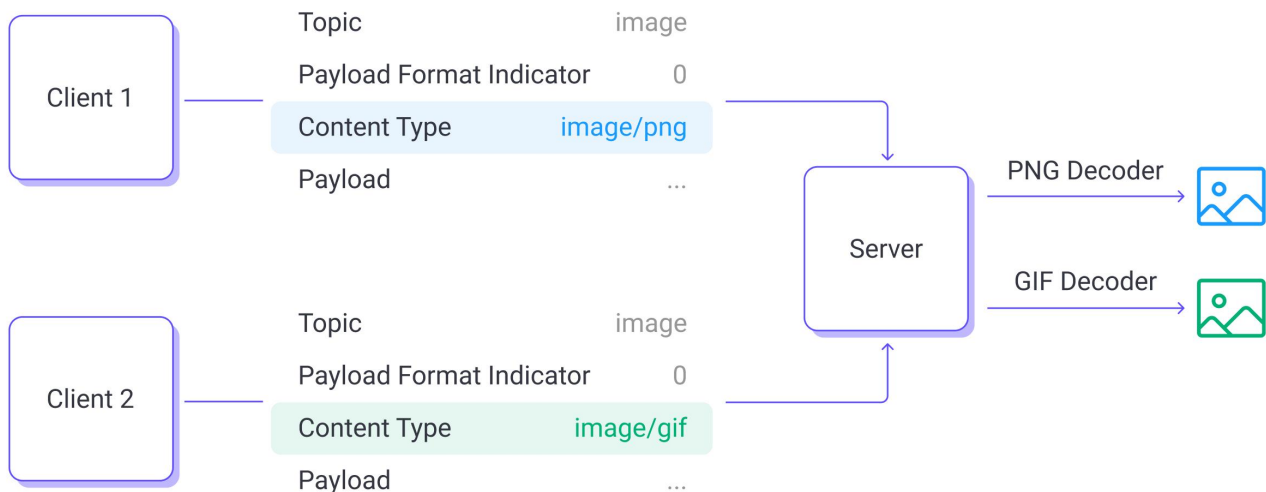
The value of Content Type is a UTF-8 encoded string that describes the content of the application message, which helps the receiver understand how to parse the application message payload. For example, if the content of the message is a JSON object, then the Content Type could be set to "json".

The exact content of this string is entirely up to the sender and receiver, and throughout the transmission of the message, the server does not use this property to verify that the message content is formatted correctly; it is only responsible for forwarding this property to the subscriber as is.

So you can even use "cocktail" to describe the JSON type as long as the receiver understands it. However, in order to avoid unnecessary troubles, we usually recommend using known MIME types to describe the message content, such as `application/json`, `application/xml`, etc.

Content Type is useful in scenarios where multiple data types need to be supported. For example, when we send an image to the other party in a chat program, and the image may be in PNG, GIF, JPEG, etc., how do we indicate to the other party the format of the image that corresponds to the binary data we are sending?

Prior to 5.0, we might choose to include the image format in a theme, such as `to/userA/image/png`, but obviously, as the number of supported image formats increases, clients need to subscribe to more and more topics for various data formats. In 5.0, we simply set the Content Type property to `image/png`.



Is It Necessary to Use Payload Format Indicator and Content Type together?

Whether Payload Format Indicator and Content Type need to be used together depends on our application scenario.

For the subscriber, it can determine whether the content of the message should be a UTF-8 string or binary data based on the value of the Content Type, so the Payload Format Indicator is not very meaningful.

For the server, however, doesn't know the meaning of the Content Type value, so if we want the server to check whether the message payload conforms to the UTF-8 encoding specification, we must use the Payload Format Indicator property.

Demo

1. Access [MQTTX Web](#) on a Web browser.
2. Create an MQTT connection named `pub` for publishing messages, and connect it to the [Free Public MQTT Server](#):

Connections + < Back **New** Connect

General

* Name ⓘ

* Client ID ⓘ ⓘ

* Host

* Port ⬆ ⬆

Username

Password

SSL/TLS ☐

Advanced ▲

MQTT Version ⬇

Connect Timeout ⬆ ⬆ (s)

Keep Alive ⬆ ⬆ (s)

Auto Reconnect ☒

Reconnect Period ⬆ ⬆ (ms)

Clean Start ☒

Session Expiry Interval ⬆ ⬆ (s)

Receive Maximum

Maximum Packet Size

3. Create a client connection named **sub** in the same way and subscribe to the topic **mqttx_89e3d55e/test** using the Client ID as a prefix:

Connections + **sub** ✓ ⓘ

New Subscription ×

* Topic ⓘ

* QoS At most once Color ⓘ

Alias ⓘ

Subscription Identifier

No Local flag ☐ true ☒ false

Retain as Published flag ☐ true ☒ false

Retain Handling ⬆

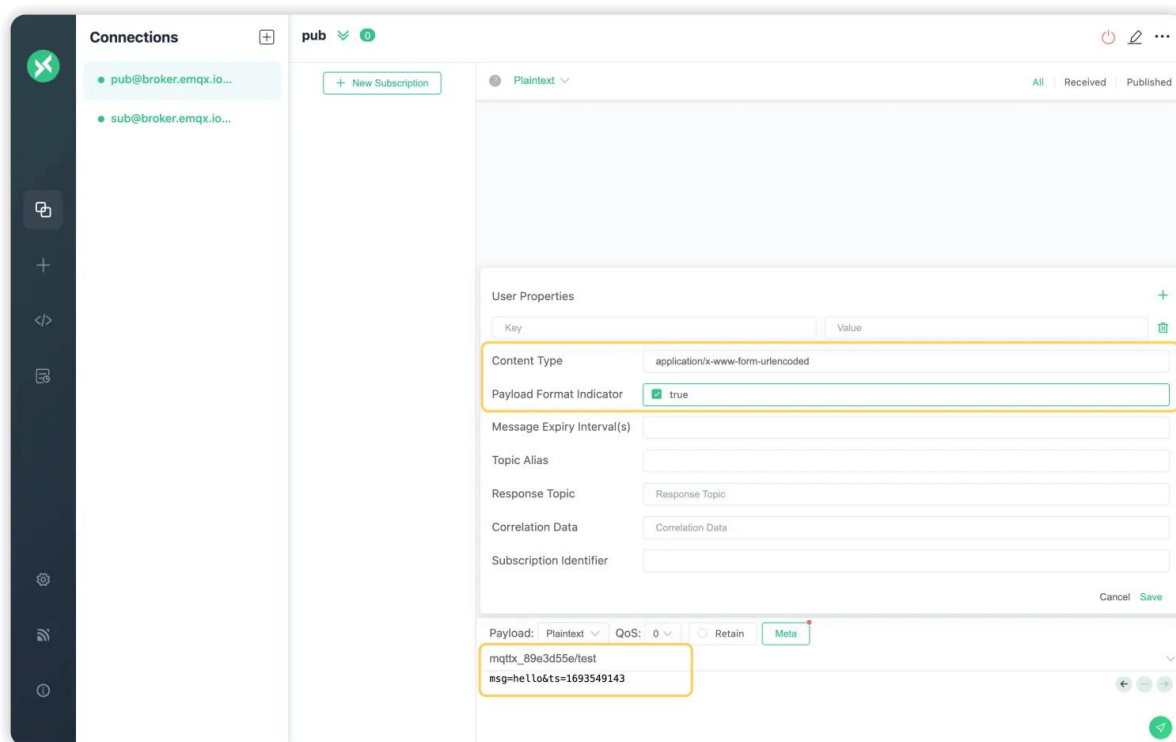
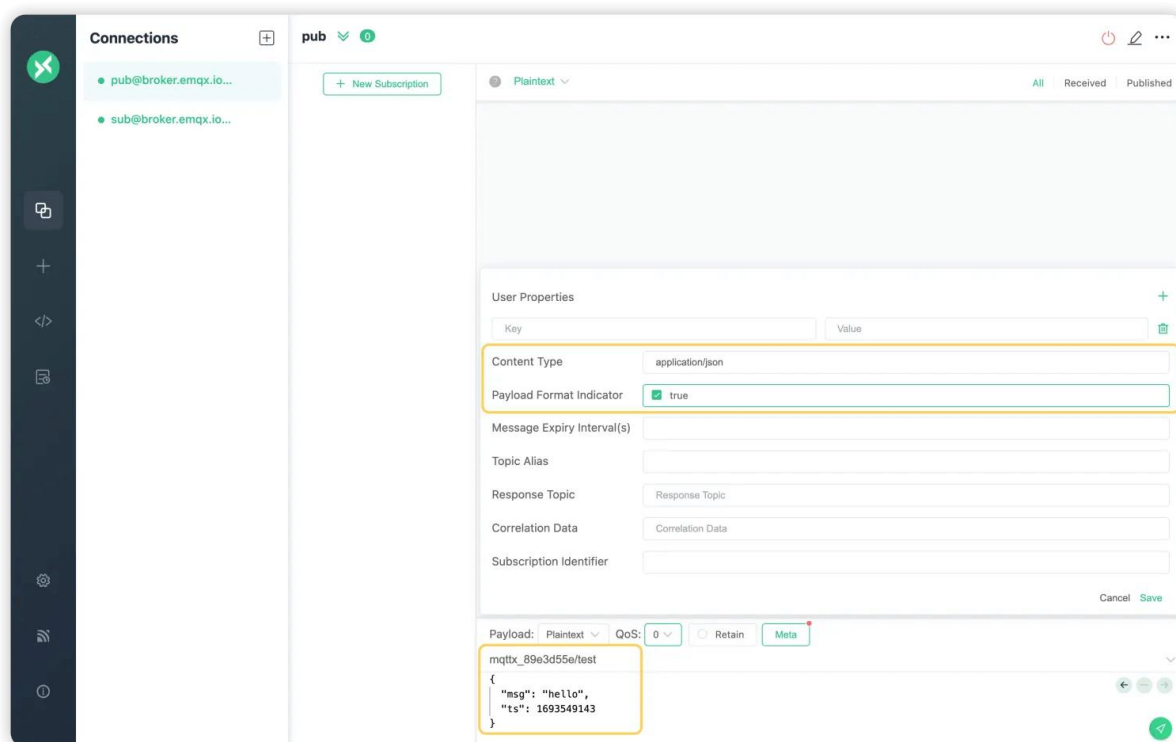
Cancel Confirm

Payload: JSON QoS: 1 Retain Meta

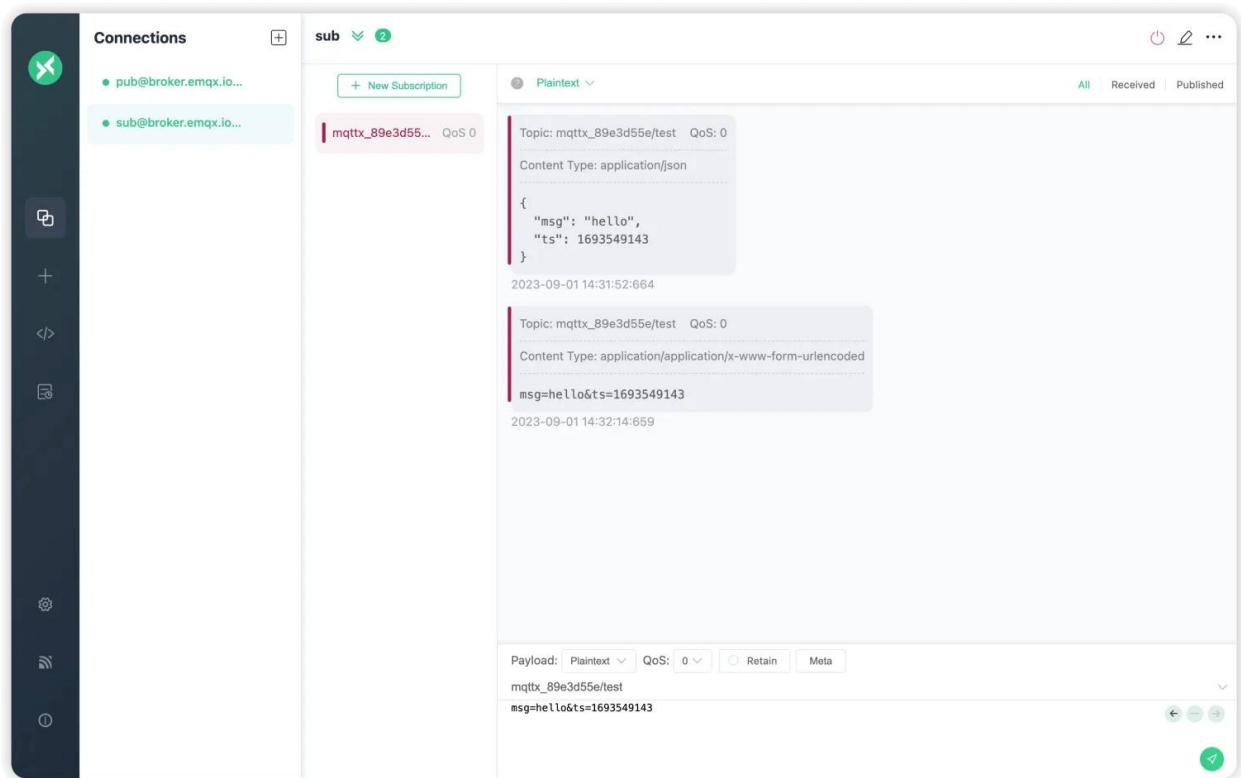
Topic

```
{
  "msg": "hello"
}
```

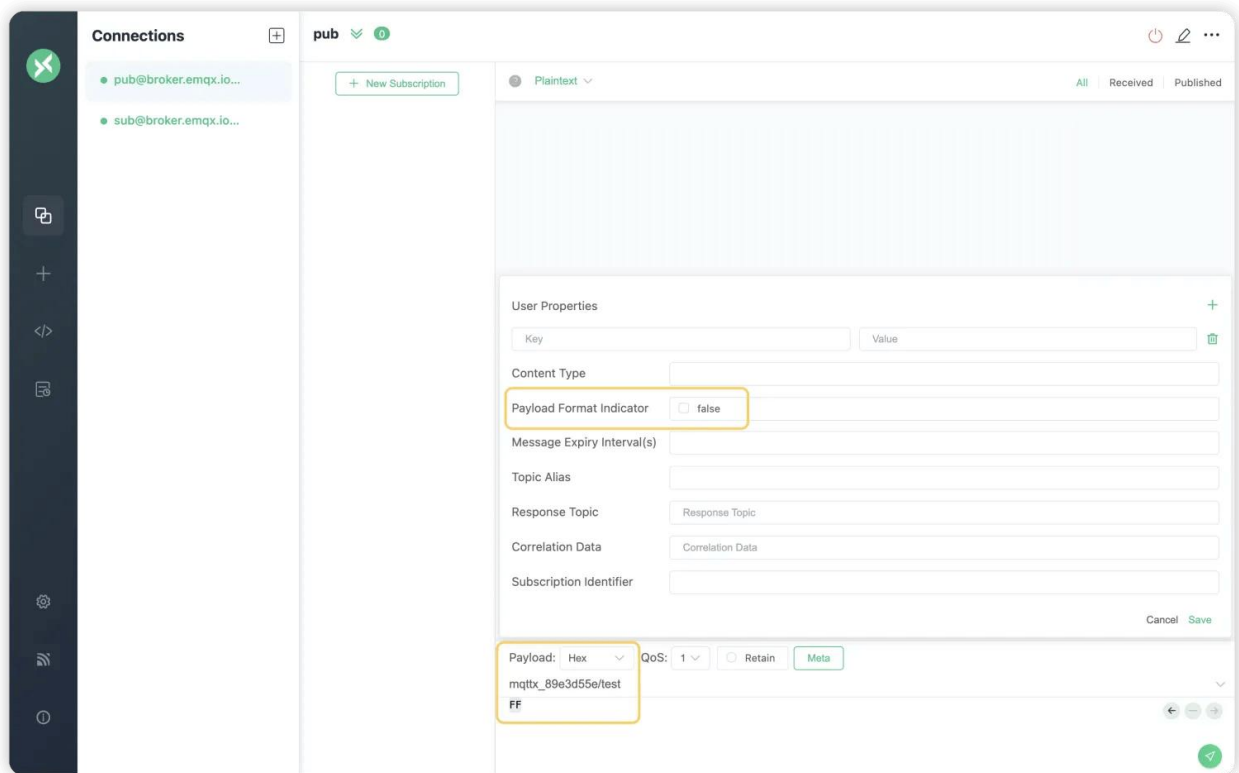
4. Then go back to the client `pub`, click the Meta button in the message bar, set the Payload Format Indicator to `true`, set the Content Type to `application/json`, publish a JSON-formatted message to the topic `mqttx_89e3d55e/test`, and then change it to `application/x-www-form-urlencoded` and then publish a form format message to the same topic:



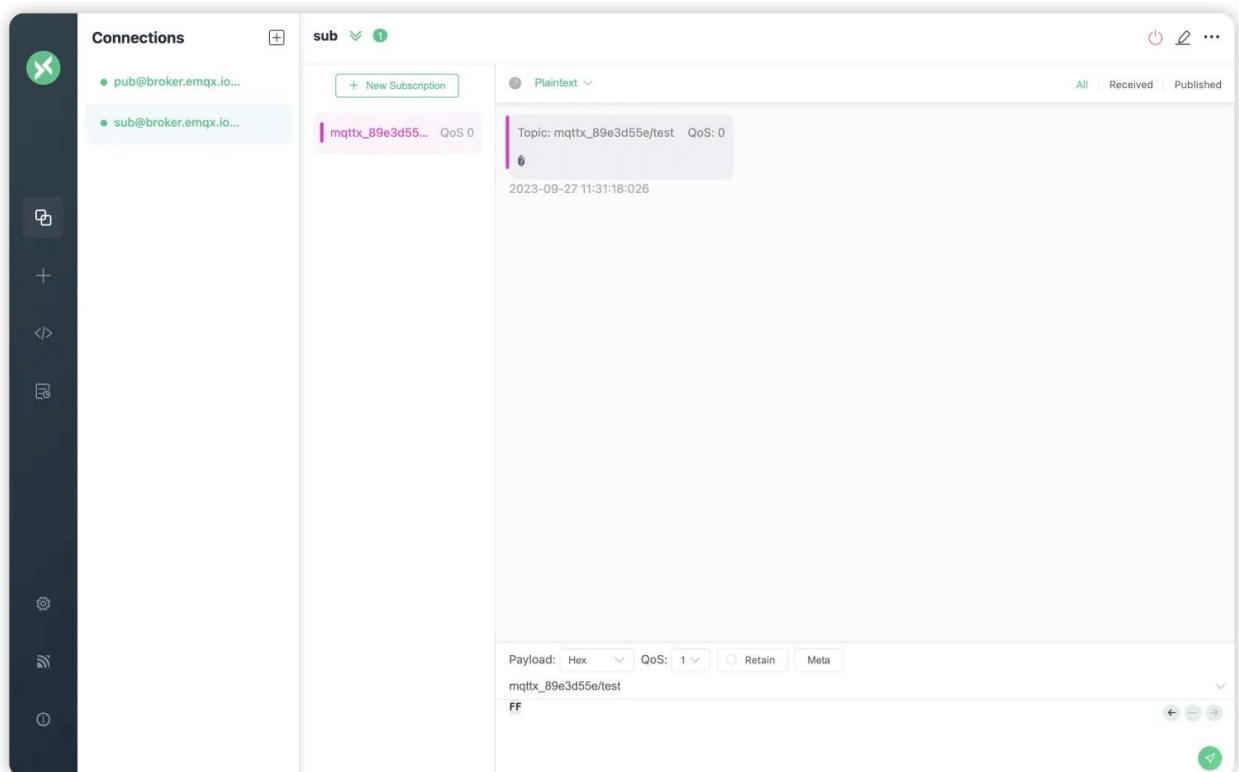
5. The Content Type of the message will be forwarded to the subscriber as is, so the subscriber can know how to parse the content in the Payload based on the value of the Content Type:



6. Back on the publishing side, set the Payload Format Indicator to `false` and change the encoding format of the Payload to `Hex`, then enter `FF` as the content of the Payload and send it. `0xFF` is a typical non-UTF-8 character:



7. Although it is displayed as garbled characters, the subscriber did receive the message with a Payload of 0xFF that we just sent. This is for performance reasons, EMQX currently does not check the Payload format:



In the terminal, we can also use the command line tool [MQTTX CLI](#) to accomplish the above, and we can subscribe to topics using the following command:

```
mqttx sub -h 'broker.emqx.io' -p 1883 -t 'random-string/demo' --output-mode clean
```

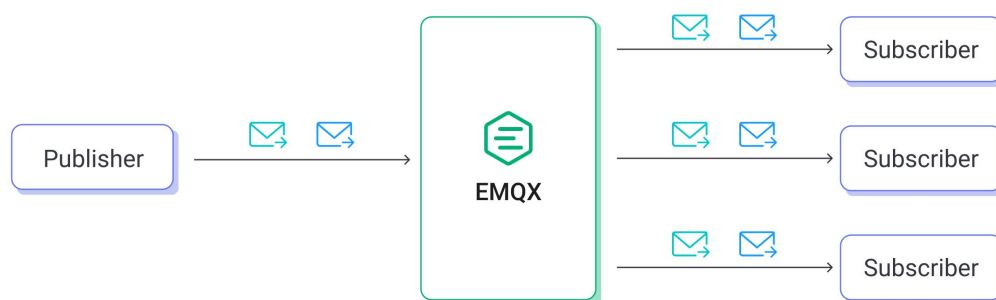
Then use the following commands to set the Payload Format Indicator and Content Type properties when you publish the message:

```
mqttx pub -h 'broker.emqx.io' -p 1883 -t 'random-string/demo' \  
--payload-format-indicator \  
--content-type 'application/json' \  
-m '{"msg": "hello"}'
```

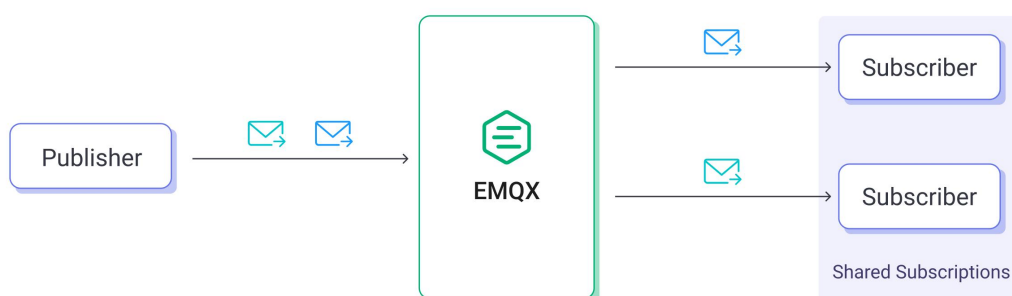
Shared Subscriptions

What are Shared Subscriptions

In normal subscriptions, every time we publish a message, all matching subscribers will receive a copy. When a subscriber's consumption speed can't keep up with the message production speed, we cannot divert some of the messages to other subscribers to share the pressure. The performance issue of a single subscriber client could easily impact the entire messaging system.



MQTT 5.0 introduced Shared Subscriptions, which allow the MQTT server to evenly distribute message load among clients using a specific subscription. This means that when we have two clients sharing a subscription, each message that matches the subscription will only have one copy delivered to one of the clients.



Shared subscriptions bring not only excellent horizontal scalability to consumers, enabling us to handle higher throughput, but also high availability. Even if one client

disconnects or fails, other clients sharing the same subscription can continue to process messages. When necessary, they can even take over the message flow that was originally intended for that client.

How Shared Subscriptions Works

With shared subscriptions, we don't need to make any changes to the underlying code of the clients. All we need to do is use topics that follow the naming convention when subscribing:

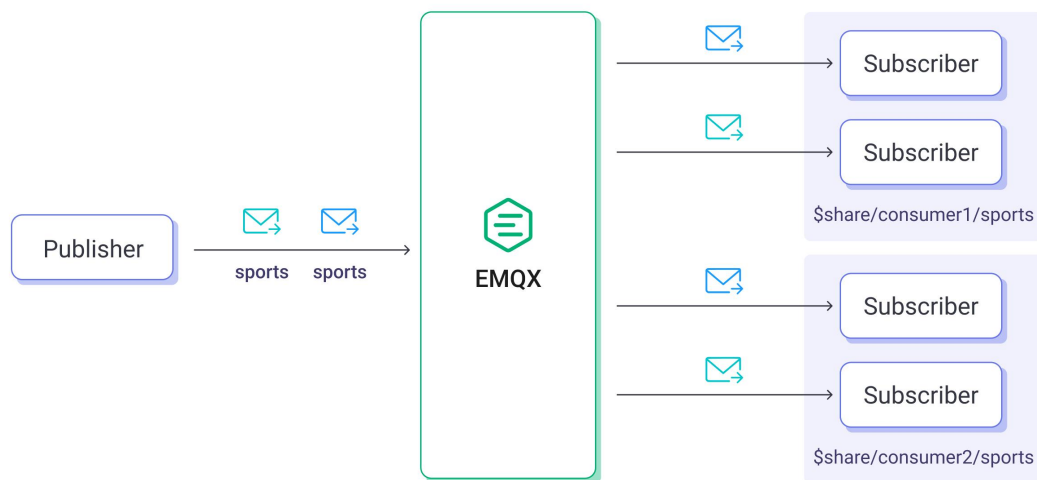
```
$share/{Share Name}/{Topic Filter}
```

`$share` is a reserved prefix, so the server knows this is a shared subscription topic. `{Topic Filter}` is the actual topic we want to subscribe to.

The middle `{Share Name}` is a string specified by the client, representing the share name used by the current shared subscription. Usually, the `{Share Name}` field is also referred to as Group Name or Group ID, making it easier to understand.

A group of subscription sessions that want to share the same subscription must use the same share name. So `$share/consumer1/sport/#` and `$share/consumer2/sport/#` belong to different shared subscription groups.

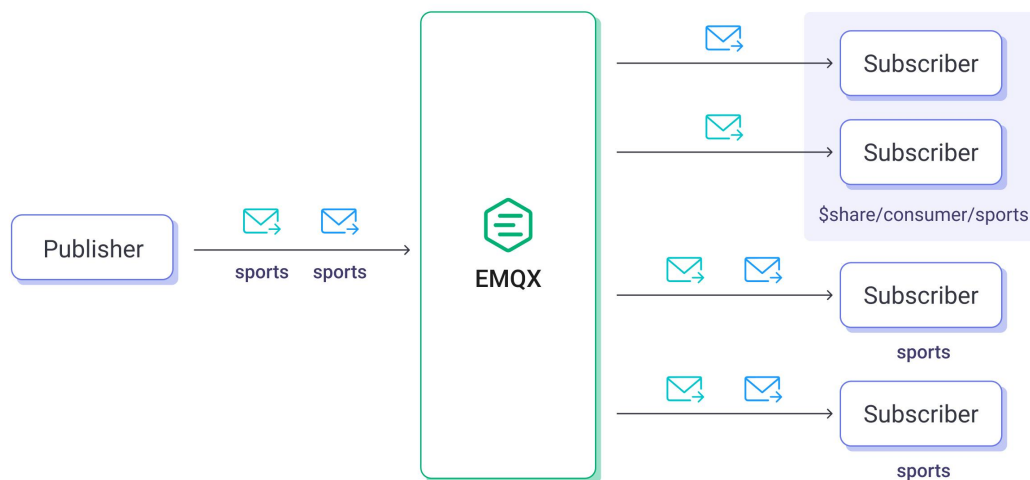
When a message matches the filters used by multiple shared subscription groups simultaneously, the server will choose a session from each matching shared subscription group to send a copy of the message. This is very useful when a topic's messages have multiple different types of consumers.



Even if two subscriptions have the same shared name {Share Name}, it does not mean they are the same shared subscription. Only {Share Name}/{Topic Filter} can uniquely identify a shared subscription group. The following subscription topics all belong to different shared subscription groups:

- `$share/consumer1/sport/tennis/+`
- `$share/consumer2/sport/tennis/+`
- `$share/consumer1/sport/#`
- `$share/comsumer1/finance/#`

Shared subscriptions and non-shared subscriptions do not affect each other. When a message matches both a shared subscription and a non-shared subscription simultaneously, the server will send a copy of the message to each client of the matched non-shared subscription, and also send a copy to one session in each of the matched shared subscription groups. If these subscriptions are from the same client, then this client may receive multiple copies of the message.



Load Balancing Strategy of Shared Subscriptions

The core of shared subscriptions lies in how the server allocates message load among clients. Common load-balancing strategies include the following:

- Random: Randomly select a session within the shared subscription group to send the message.
- Round Robin: Send messages to the sessions in the shared subscription group in turn.
- Hash: Select a session based on the hash result of a field
- Sticky: Randomly select a session within the shared subscription group to send the message. Maintain this selection until the session ends, and then repeat this process.
- Local First: Randomly select, but prioritize sessions on the same node as the message publisher. If no such session exists, it degrades to a normal random strategy on the remote nodes.

The balance effects achieved by the **Random** and **Round Robin** strategies are relatively similar, so there is not much difference in their application scenarios. However, the actual balance effect of the **Random** strategy is often affected by the random algorithm used by the server.

In practical applications, messages may correlate. For example, multiple fragments belonging to the same image are obviously unsuitable for distribution to multiple subscribers. In this case, we need to choose a session based on the **Hash** strategy of Client ID or Topic. This ensures that messages from the same publisher or topic are always processed by the same session in the shared subscription group. Of course, the **Sticky** strategy also has the same effect.

The **Local First** strategy is more suitable for cluster use than the **Random** strategy. Prioritizing the selection of local subscribers can effectively reduce message latency. However, the premise of using this strategy is to ensure that publishers and subscribers are distributed fairly evenly on each node to avoid excessive differences in message load on different subscribers.

How Do MQTT 3.1.1 Clients Use Shared Subscriptions?

Long before the release of MQTT 5.0, EMQX had designed a shared subscription scheme that many users have adopted. Similar to the official scheme of MQTT 5.0, EMQX recognizes the following format of topics as shared subscription topics in MQTT 3.1.1:

```
$queue/{Topic Filter}
```

The prefix `$queue` indicates that this is a shared subscription, and `{Topic Filter}` is the actual topic we want to subscribe to. It is equivalent to `$share/queue/{Topic Filter}` in MQTT 5.0, which means the share name is fixed as `queue`. So this scheme does not support multiple shared subscription groups with the same topic filter.

Since the naming convention of the shared subscriptions in MQTT 5.0, `$share/{Share Name}/{Topic Filter}`, is also a valid topic in MQTT 3.1.1, and the logic of shared

subscription is implemented entirely in the [MQTT broker](#), the client only needs to modify the topic content of the subscription. Therefore, even devices still using MQTT 3.1.1 can now directly use shared subscriptions provided by MQTT 5.0.

Shared Subscriptions Use Cases

Here are a few typical use cases for shared subscriptions:

- When the backend's consumption capacity does not match the message production capacity, we can use shared subscriptions to allow more clients to share the load.
- When the system needs to ensure high availability, especially for critical business with a large influx of messages, we can use shared subscriptions to avoid a single point of failure.
- When the influx of messages may grow rapidly in the future and the consumer side needs to be able to scale horizontally, we can use shared subscriptions to provide high scalability.

Suggestions for Using Shared Subscriptions

Use the Same QoS within a Shared Subscription Group

MQTT allows sessions within a shared subscription group to use different QoS levels, but this could result in different quality assurances when delivering messages to different sessions within the same group. Accordingly, debugging can become more difficult when problems arise. So it's best to use the same QoS within a shared subscription group.

Set Session Expiry Interval Reasonably

It is common to use shared subscriptions together with persistent sessions. However, it's important to note that even if a client in a shared subscription group goes offline, as long as its session and subscription still exist, the MQTT server will continue distributing messages to this session.

Considering that the client may be offline for a long time due to failures or other reasons, if the session expiry interval is too long, many messages will be unable to be processed because they are delivered to the offline client.

A better choice might be to stop considering the subscriber when allocating message load once the subscriber goes offline, even if the session has not expired. Although this behavior is different from normal subscriptions, it is allowed by the MQTT protocol.

Demo

To better demonstrate the effect of shared subscriptions, we will use the [MQTTX CLI](#), an MQTT command-line client tool, for the demonstration.

Start three terminal windows and use the following commands to create three clients connected to the [Free Public MQTT Broker](#) and subscribe to the topics

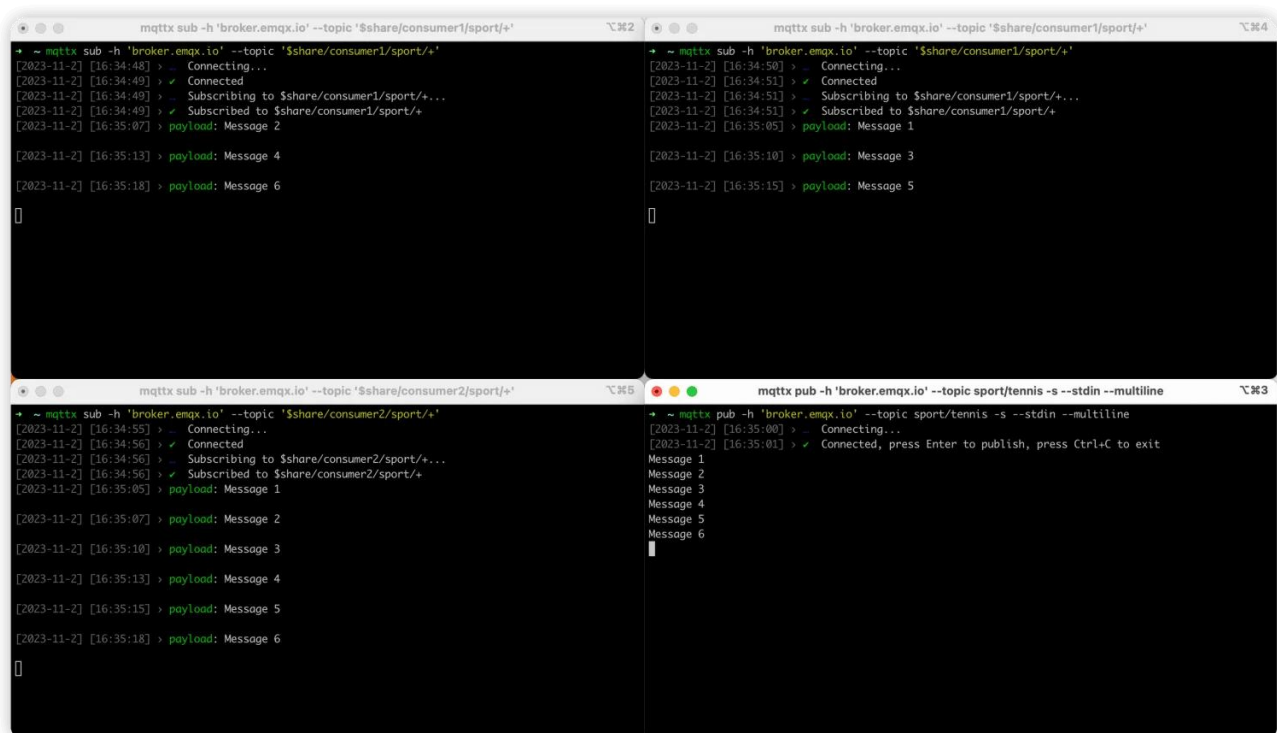
`$share/consumer1/sport/+, $share/consumer1/sport/+` and
`$share/consumer2/sport/+,` respectively:

```
mqttx sub -h 'broker.emqx.io' --topic '$share/consumer1/sport/+'
```

Then start a new terminal window and use the following command to publish 6 messages to the topic `sport/tennis`. Here we use the `--multiline` option to send multiple messages each time we hit enter:

```
mqttx pub -h 'broker.emqx.io' --topic sport/tennis -s --stdin --multiline
```

The default load balancing strategy for shared subscriptions in EMQX is **Round Robin**, so we will see the two subscribers within the `consumer1` group alternately receiving the messages we publish, while there is only one subscriber in the shared subscription group `consumer2`, so it will receive all the messages:



```
mqttx sub -h 'broker.emqx.io' --topic '$share/consumer1/sport/+'
[2023-11-2] [16:34:48] > Connecting...
[2023-11-2] [16:34:49] > ✓ Connected
[2023-11-2] [16:34:49] > Subscribing to $share/consumer1/sport/+...
[2023-11-2] [16:34:49] > ✓ Subscribed to $share/consumer1/sport/+
[2023-11-2] [16:35:07] > payload: Message 2

[2023-11-2] [16:35:13] > payload: Message 4

[2023-11-2] [16:35:18] > payload: Message 6

[]

mqttx sub -h 'broker.emqx.io' --topic '$share/consumer1/sport/+'
[2023-11-2] [16:34:50] > Connecting...
[2023-11-2] [16:34:51] > ✓ Connected
[2023-11-2] [16:34:51] > Subscribing to $share/consumer1/sport/+...
[2023-11-2] [16:34:51] > ✓ Subscribed to $share/consumer1/sport/+
[2023-11-2] [16:35:05] > payload: Message 1

[2023-11-2] [16:35:10] > payload: Message 3

[2023-11-2] [16:35:15] > payload: Message 5

[]

mqttx sub -h 'broker.emqx.io' --topic '$share/consumer2/sport/+'
[2023-11-2] [16:34:55] > Connecting...
[2023-11-2] [16:34:56] > ✓ Connected
[2023-11-2] [16:34:56] > Subscribing to $share/consumer2/sport/+...
[2023-11-2] [16:34:56] > ✓ Subscribed to $share/consumer2/sport/+
[2023-11-2] [16:35:05] > payload: Message 1

[2023-11-2] [16:35:07] > payload: Message 2

[2023-11-2] [16:35:10] > payload: Message 3

[2023-11-2] [16:35:13] > payload: Message 4

[2023-11-2] [16:35:15] > payload: Message 5

[2023-11-2] [16:35:18] > payload: Message 6

[]

mqttx pub -h 'broker.emqx.io' --topic sport/tennis -s --stdin --multiline
[2023-11-2] [16:35:00] > Connecting...
[2023-11-2] [16:35:01] > ✓ Connected, press Enter to publish, press Ctrl+C to exit
Message 1
Message 2
Message 3
Message 4
Message 5
Message 6

```

This is just a simple example. You can also try to join or leave the shared subscription group at any time, observe whether EMQX timely allocates the load according to the latest subscription, or install EMQX yourself and observe the behavior of different load balancing strategies.

If you want to know how to use the shared subscription feature in code, we provide Python example code in the [emqx/MQTT-Features-Example](#) project.

Subscription Options

Subscription Options

A subscription in MQTT consists of a topic filter and corresponding subscription options. So, we can set different subscription options for each subscription.

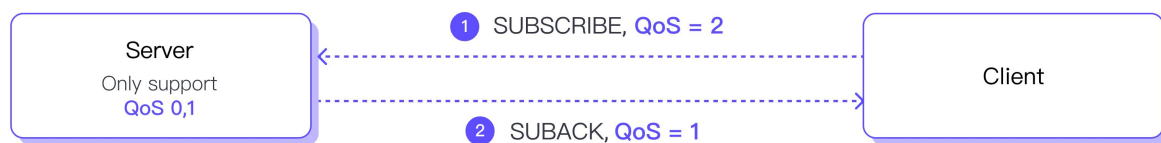
MQTT 5.0 introduces four subscription options: QoS, No Local, Retain As Published, and Retain Handling. On the other hand, MQTT 3.1.1 only provides the QoS subscription option. However, the default behavior of these new subscription options in MQTT 5.0 remains consistent with MQTT 3.1.1. This makes it user-friendly if you plan to upgrade from MQTT 3.1.1 to MQTT 5.0.

QoS

QoS is the most commonly used subscription option, which represents the maximum QoS level that the server can use when sending messages to the subscriber.

A client may specify a QoS level below 2 during subscription if its implementation does not support QoS 1 or 2.

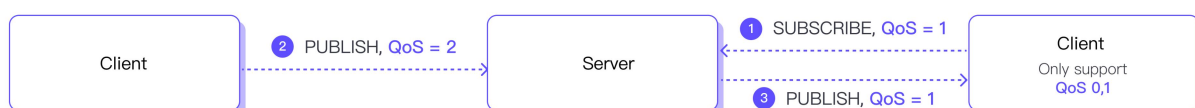
Additionally, if the server's maximum supported QoS level is lower than the QoS level requested by the client during the subscription, it becomes apparent that the server cannot meet the client's requirements. In such cases, the server informs the subscriber of the granted maximum QoS level through the subscription response packet (SUBACK). The subscriber can then assess whether to accept the granted QoS level and continue communication.



A simple calculation formula:

The maximum QoS granted by the server = min (The maximum QoS supported by the server, The maximum QoS requested by the client)

However, the maximum QoS level requested during subscription does not restrict the QoS level used by the publishing end when sending messages. When the requested maximum QoS level during subscription is lower than the QoS level used for message publishing, the server will not ignore these messages. To maximize message delivery, it will downgrade the QoS level of these messages before forwarding.



Similarly, we have a simple calculation formula:

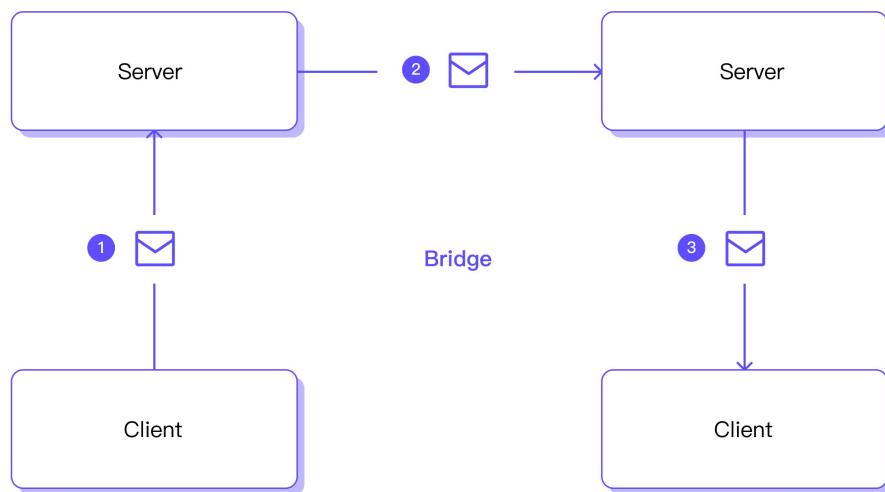
QoS in the forwarded message = min (The original QoS of the message, The maximum QoS granted by the server)

No Local

The No Local option has only two possible values, 0 and 1. A value of 1 indicates that the server must not forward the message to the client that published it, while 0 means the

opposite.

This option is commonly used in bridging scenarios. Bridging is essentially two MQTT Servers establishing an [MQTT connection](#), then they subscribe to some topics from each other. The server forwards client messages to another server, which can continue forwarding them to its clients.



Let's consider the most straightforward example where we assume two MQTT servers, Server A and Server B.

They have subscribed to the topic # from each other. Now, Server A forwards some messages from the client to Server B, and when Server B looks for a matching subscription, Server A is there too. If Server B forwards the messages to Server A, then Server A will forward them to Server B again after receiving them, thus falling into an endless forwarding storm.

However, if both Server A and Server B set the No Local option to 1 while subscribing to the topic #, this problem can be ideally avoided.

Retain As Published

The Retain As Published option also has two possible values, 0 and 1. Setting it to 1 means the server should preserve the Retain flag unchanged when forwarding application messages to subscribers, and setting it to 0 means that the Retain flag must be cleared.

Like the No Local option, Retain As Published primarily applies in bridge scenarios.

We know that when the server receives a retained message, in addition to storing it, it will also forward it to existing subscribers like a normal message, and the Retain flag of the message will be cleared when forwarding.

This presents a challenge in bridge scenarios. Continuing with the previous setup, when Server A forwards a retained message to Server B, the Retain flag is cleared, causing Server B not to recognize it as a retained message and not store it. This makes retained messages unusable across bridges.

In MQTT 5.0, we can let the bridged server set the “Retain” publish option to 1 when subscribing to solve this problem.



Retain Handling

The Retain Handling subscription option indicates to the server whether to send retained messages when a subscription is established.

When a subscription is established, the retained messages matching the subscription in

the server will be delivered by default.

However, there are cases where a client may not want to receive retained messages. For example, if a client reuses a session during connection but cannot confirm whether the previous connection successfully created the subscription, it may attempt to subscribe again. If the subscription already exists, the retained messages may have been consumed, or the server may have cached some messages that arrived during the client's offline period. In such cases, the client may not want to receive the retained messages the server publishes.

Additionally, the client may not want to receive the retained message at any time, even during the initial subscription. For example, we send the state of the switch as a retained message, but for a particular subscriber, the switch event will trigger some operations, so it is helpful not to send the retained message in this case.

We can choose among these three different behaviors using Retain Handling:

- Setting Retain Handling to 0 means that retained messages are sent whenever a subscription is established.
- Setting Retain Handling to 1 means retained messages are sent only when establishing a new subscription, not a repeated one.
- Setting Retain Handling to 2 means no retained messages are sent when a subscription is established.

Demo of the QoS Subscription Option

1. Access [MQTTX Web](#) on a Web browser.
2. Create an MQTT connection using WebSocket and connect to the [Free Public MQTT Server](#):

Connections New Connect

General

* Name ⓘ

* Client ID ⓘ ⓘ

* Host

* Port

* Path

Username

Password

SSL/TLS ☐

Advanced ▲

Connect Timeout (s)

Keep Alive (s)

Clean Session ☒ true ☐ false

Auto Reconnect ☐ true ☒ false

MQTT Version

Session Expiry Interval (s)

Receive Maximum

Maximum Packet Size

- After a successful connection, we subscribe to the topic `mqttx_4299c767/demo` with a QoS of 0. Since the public server may be used by many people simultaneously, to avoid topic conflicts with others, we can use the Client ID as a prefix for the topic:

Connections demo ✓

demo@broker.emqx.io:8083

Edit Subscription ✕

* Topic

* QoS At most once

Color

Alias

Subscription Identifier

No Local flag ☐ true ☒ false

Retain as Published flag ☐ true ☒ false

Retain Handling

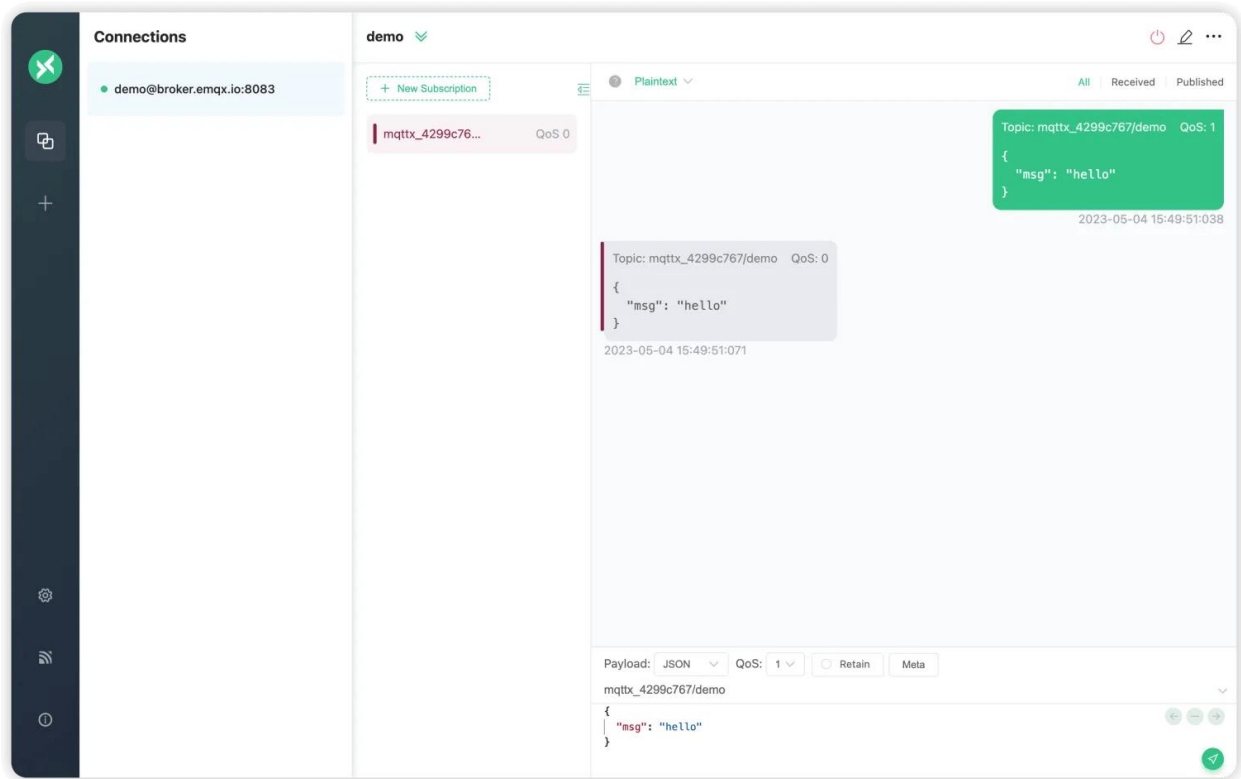
Cancel Confirm

Payload: JSON QoS: 1 Retain Meta

mqttx_4299c767/demo

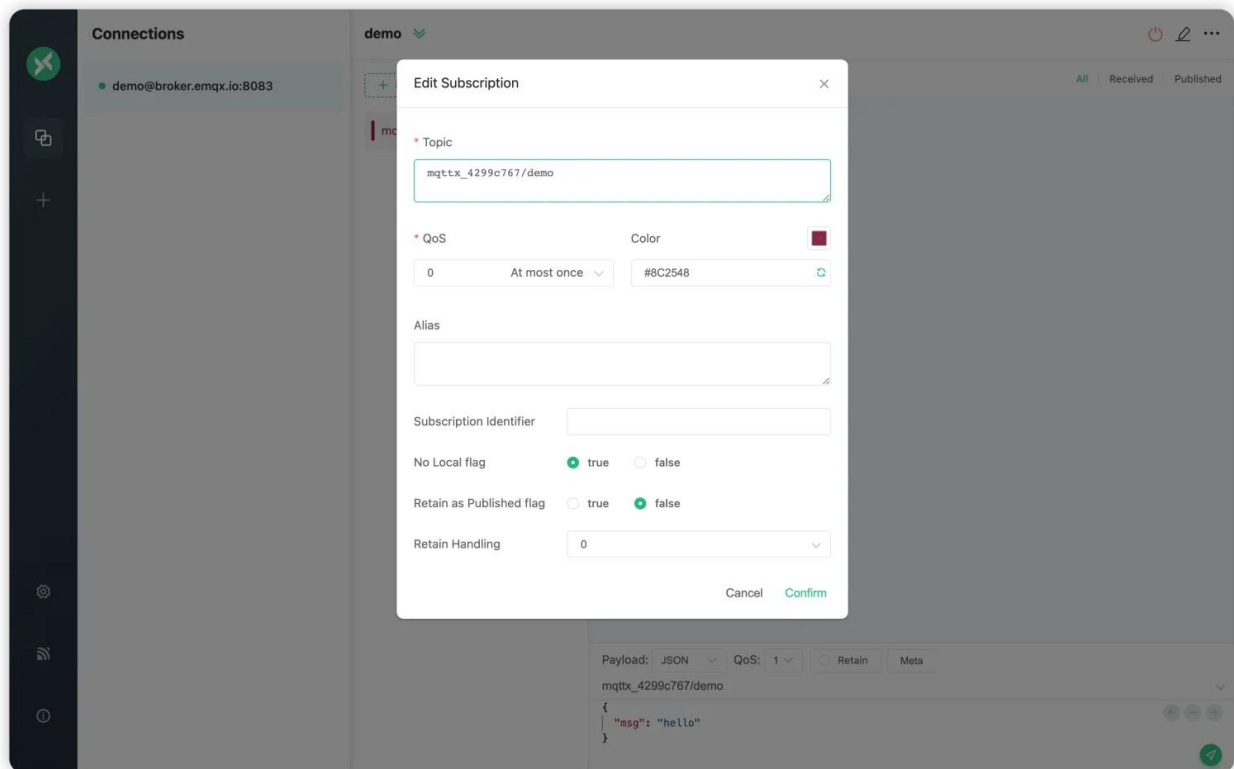
```
{
  "msg": "hello"
}
```

4. After a successful subscription, we publish a QoS 1 message to the topic `mqttx_4299c767/demo`. At this point, we will observe that while we have sent a QoS 1 message, we receive a QoS 0 message. This indicates that QoS degradation has occurred:

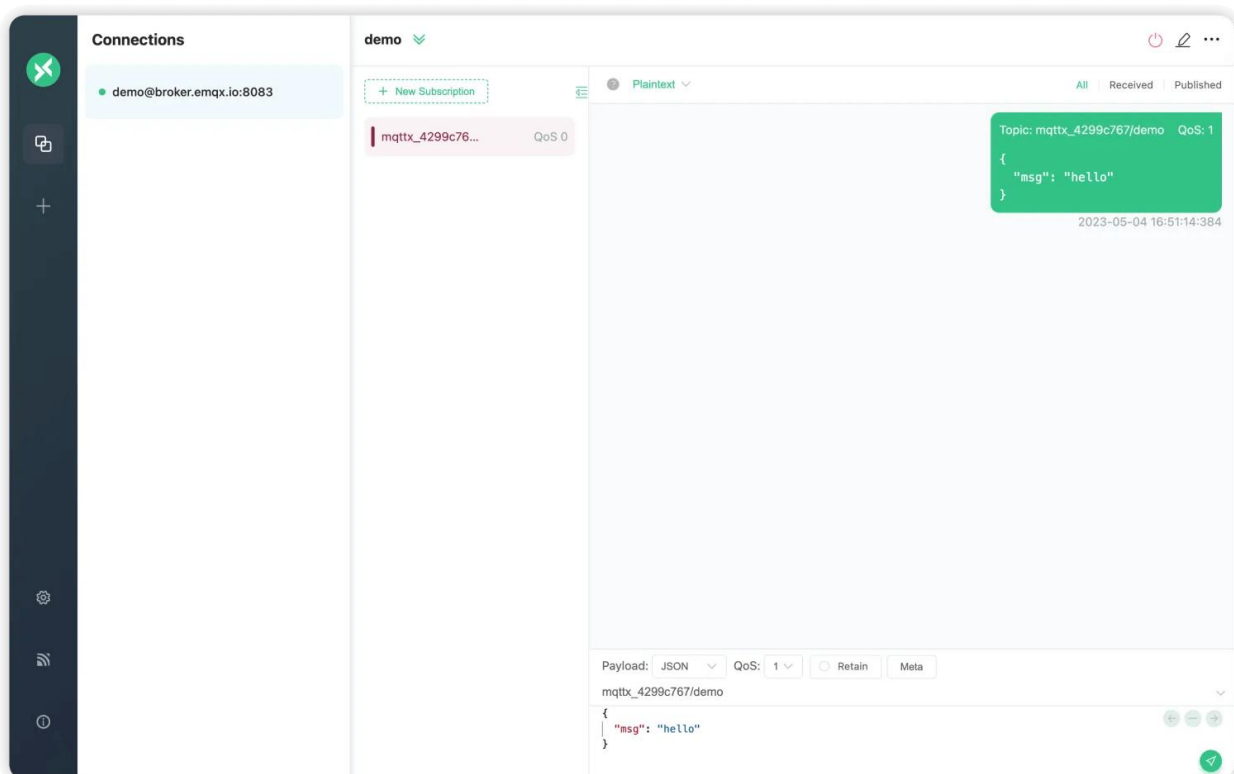


Demo of the No Local Subscription Option

1. Access [MQTTX Web](#) on a Web browser.
2. Create an MQTT connection using WebSocket and connect to the [Free Public MQTT Server](#).
3. After a successful connection, we subscribe to the topic `mqttx_4299c767/demo` and set the No Local option to true:

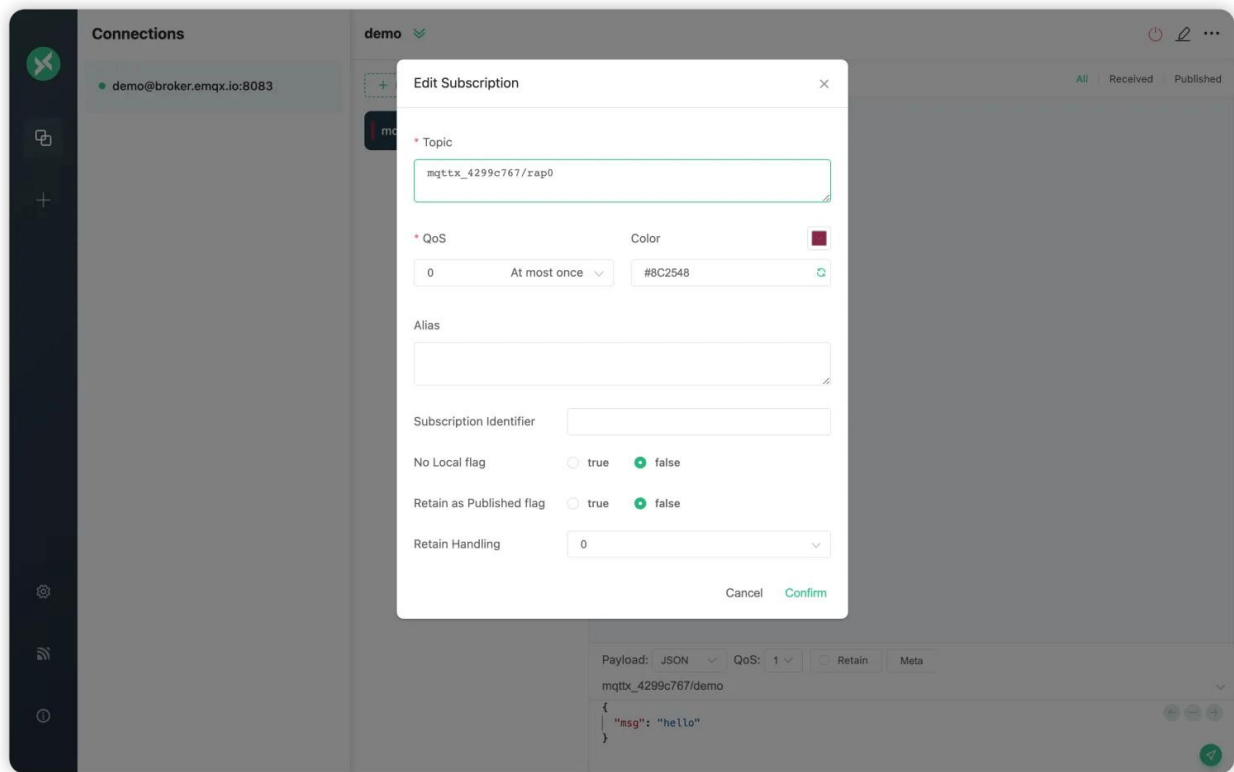


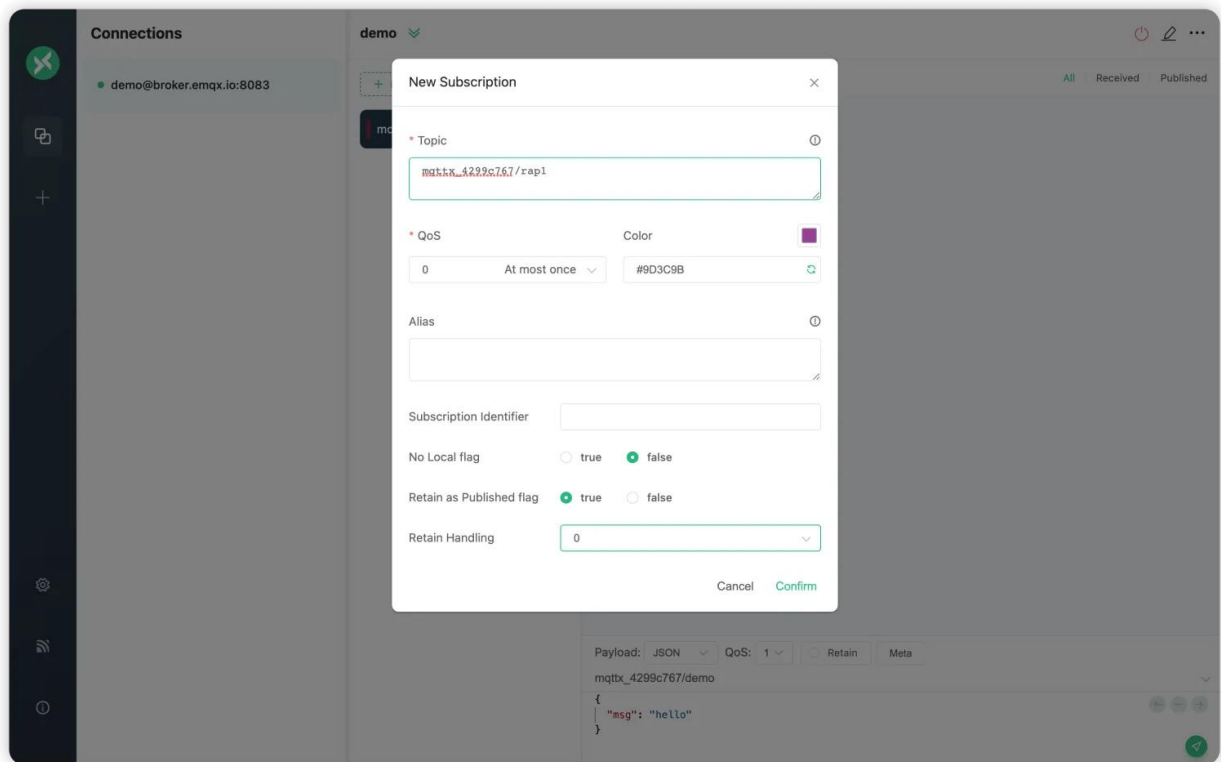
4. After a successful subscription, similar to the QoS demonstration mentioned earlier, we still let the subscriber publish the message. However, this time we will notice that the subscriber is unable to receive the message:



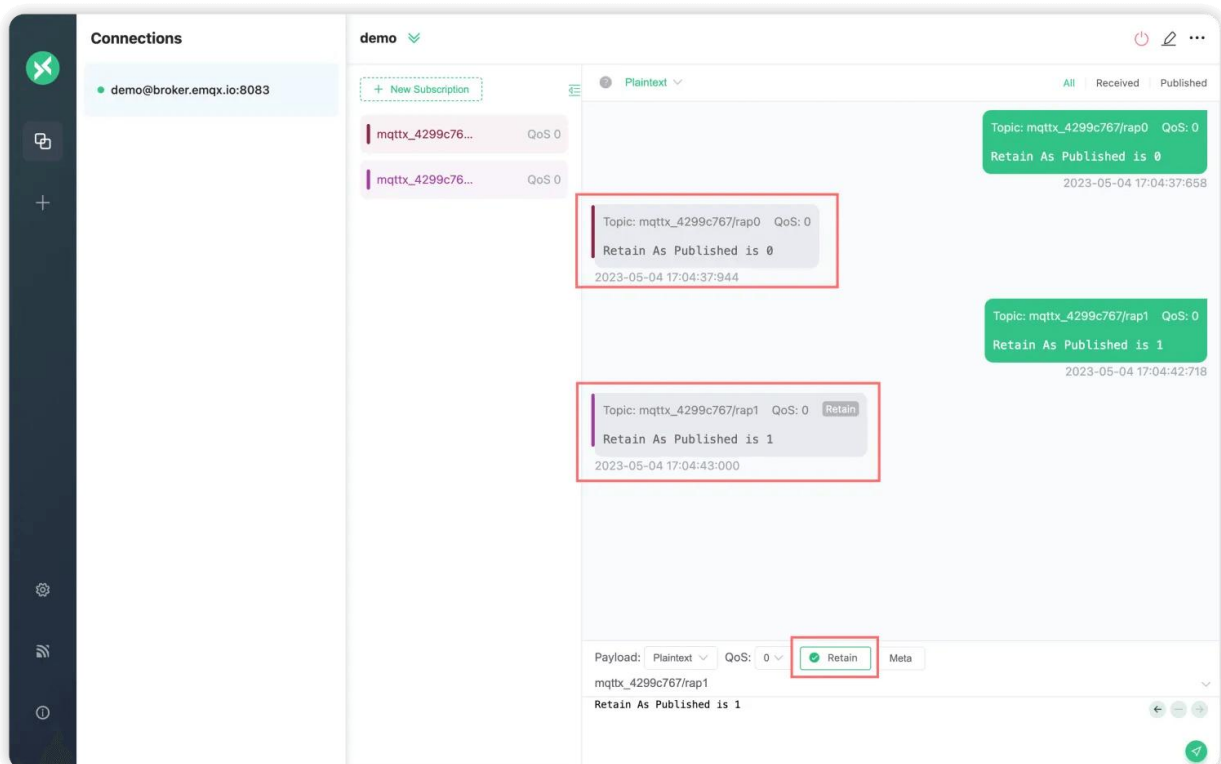
Demo of the Retain As Published Subscription Option

1. Access [MQTTX Web](#) on a Web browser.
2. Create an MQTT connection using WebSocket and connect to the [Free Public MQTT Server](#).
3. After a successful connection, we subscribe to the topic `mqttx_4299c767/rap0` with Retain As Published set to false. Then, we subscribe to the topic `mqttx_4299c767/rap1` with Retain As Published set to true:



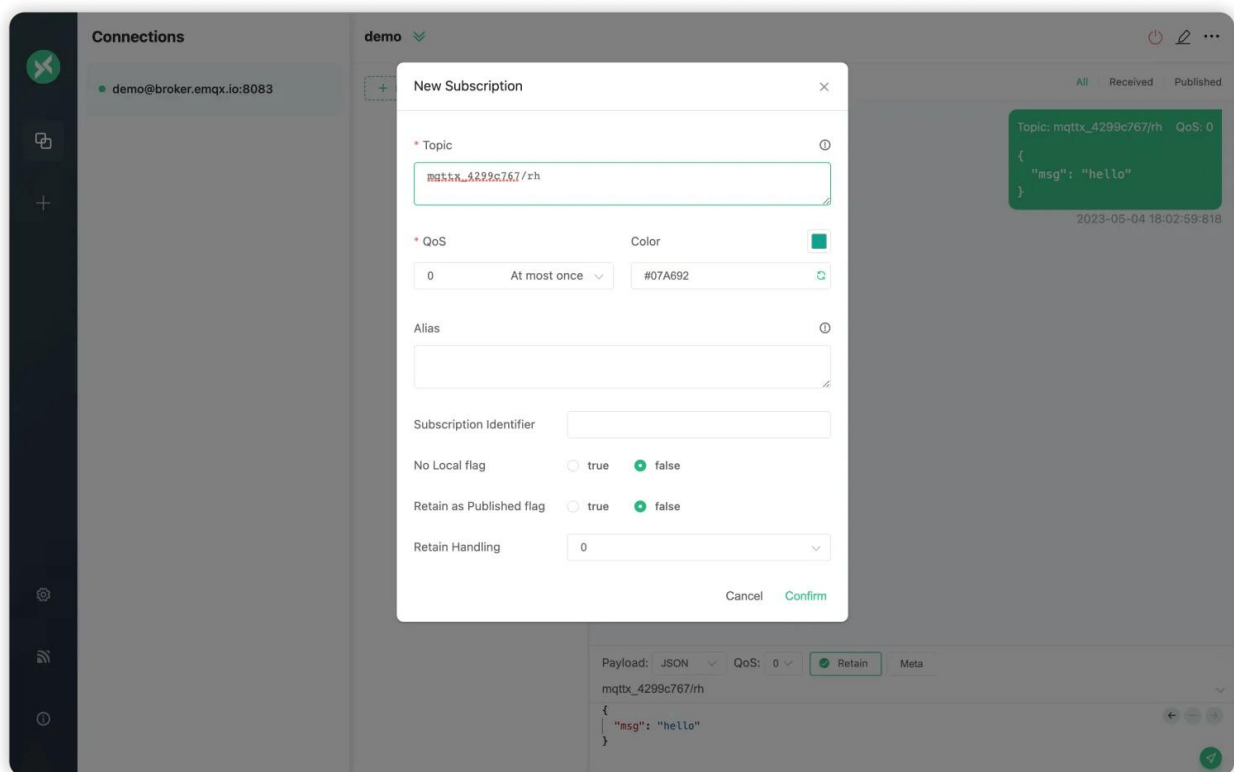


4. After the successful subscription, we publish a retained message to the topics `mqttx_4299c767/rap0` and `mqttx_4299c767/rap1`, respectively. We will see that the Retain flag in the message received by the former is cleared, and the Retain flag in the message received by the latter is retained:

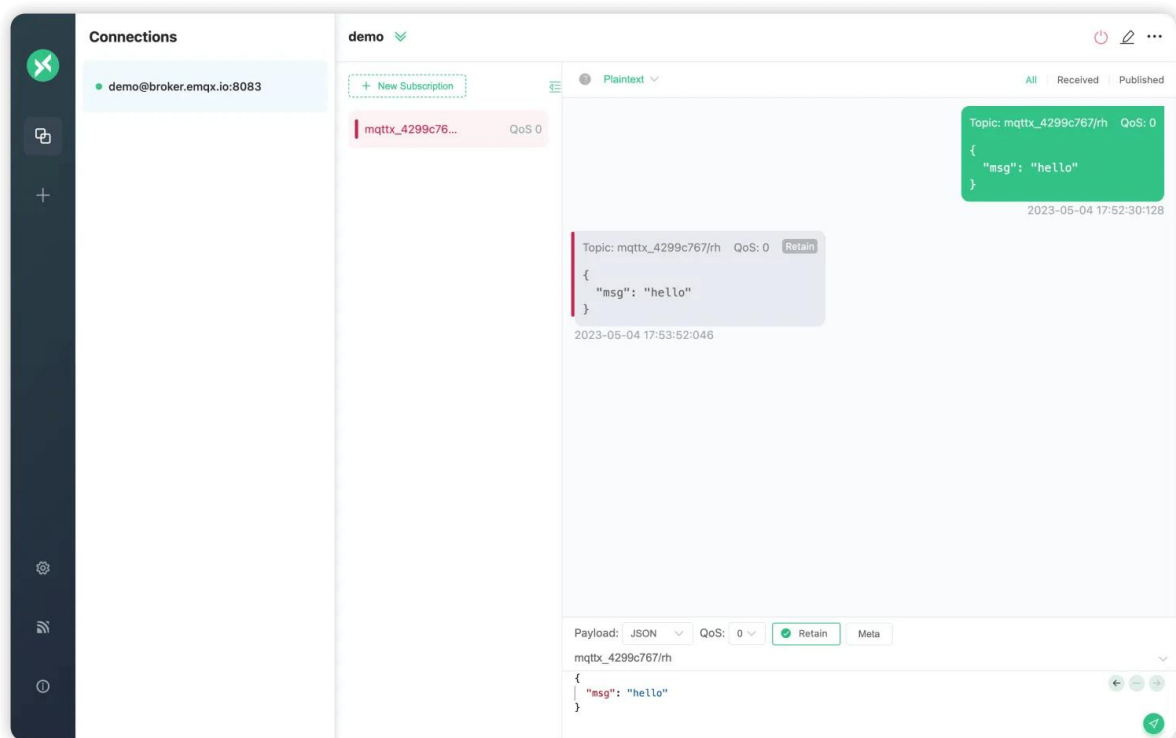


Demo of the Retain Handling Subscription Option

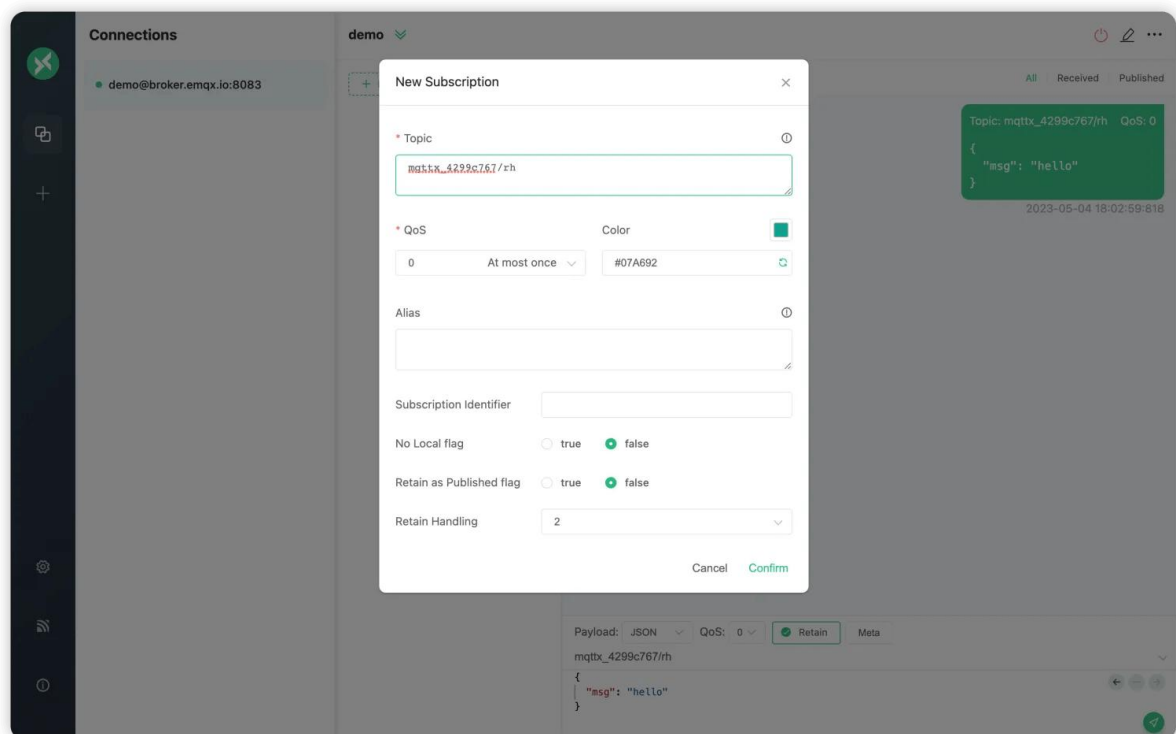
1. Access [MQTTX Web](#) on a Web browser.
2. Create an MQTT connection using WebSocket and connect to the [Free Public MQTT Server](#).
3. After the successful connection, we publish a retained message to the topic `mqttx_4299c767/rh`. Then, we subscribe to the topic `mqttx_4299c767/rh` and set the Retain Handling option to 0:



4. After the successful subscription, we will receive the retained message sent by the server:



5. Unsubscribe and resubscribe with Retain Handling set to 2. After the subscription is successful this time, we will not receive the retained message sent by the server:



In MQTTX, we cannot demonstrate the effect of Retain Handling set to 1. You can get a Python sample code for subscription options [here](#).

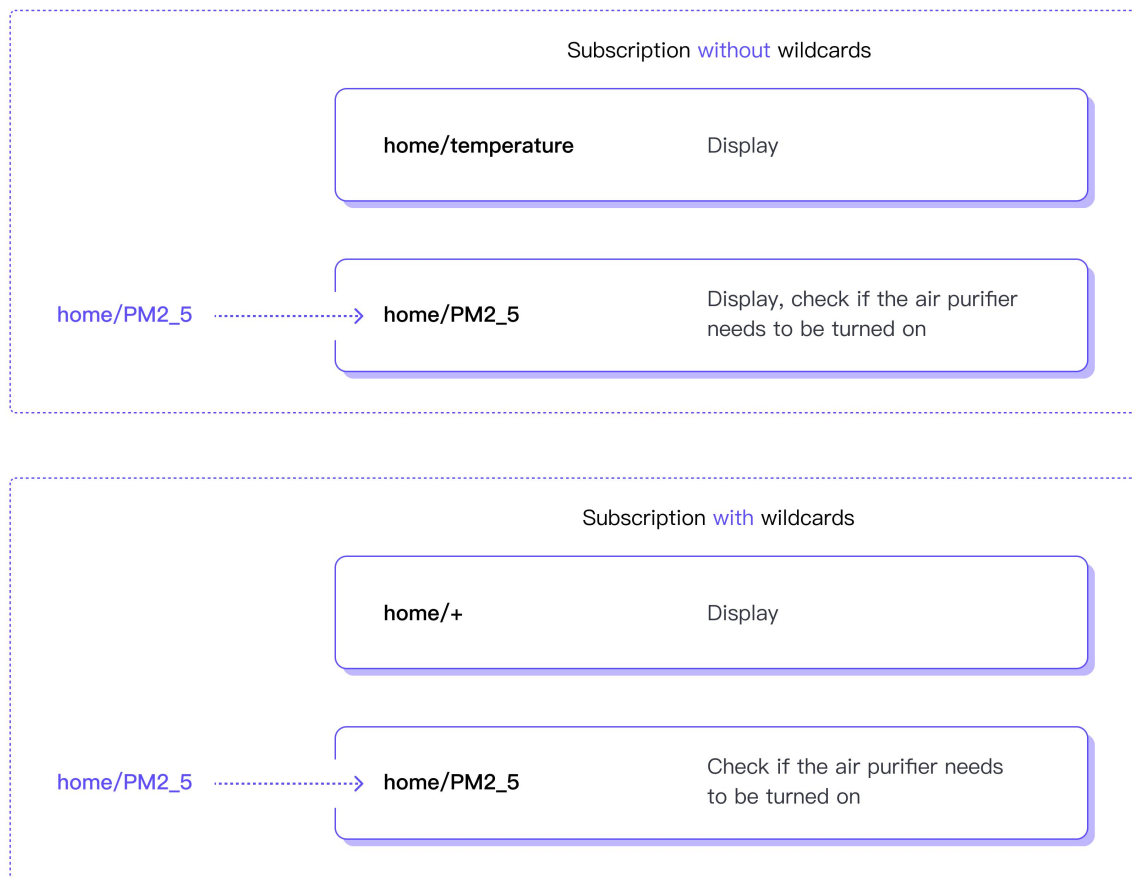
Subscription Identifier

Why Do We Need Subscription Identifiers?

Most implementations of [MQTT clients](#) use a callback mechanism to handle incoming messages.

Within the callback function, we only have access to the topic name of the message. If it is a non-wildcard subscription, the topic filter used during the subscription will be identical to the topic name in the message. Therefore, we can directly establish a mapping between the subscribed topics and callback functions. Then, upon message arrival, we can look up the corresponding callback based on the topic name in the message and execute it.

However, if it's a wildcard subscription, the topic name in the message will be different from the original topic filter used during the subscription. In this case, we need to match the topic name in the message with the original subscription one by one to determine which callback function should be executed. This obviously affects the processing efficiency of the client.

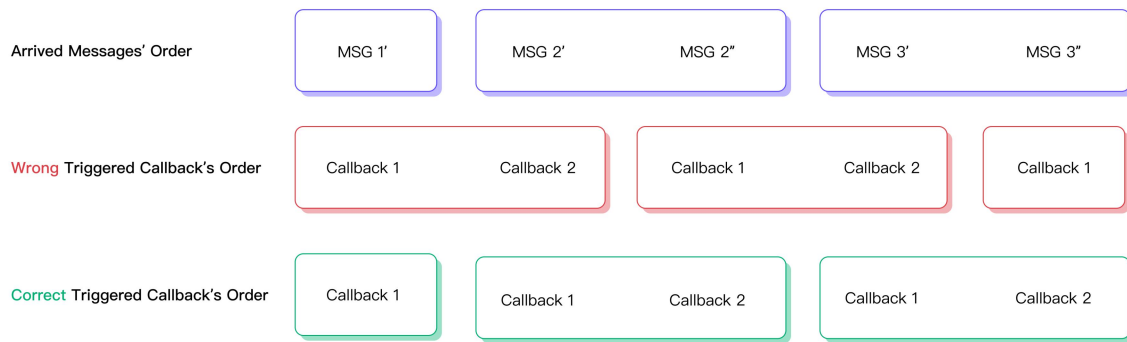


In addition, [MQTT](#) allows a client to establish multiple subscriptions, so a single message can match multiple client subscriptions when using the wildcard subscription.

In such cases, MQTT allows the server to send a separate message for each overlapping subscription or only one message for all the overlapping subscriptions. The former option means that the client will receive multiple duplicate messages.

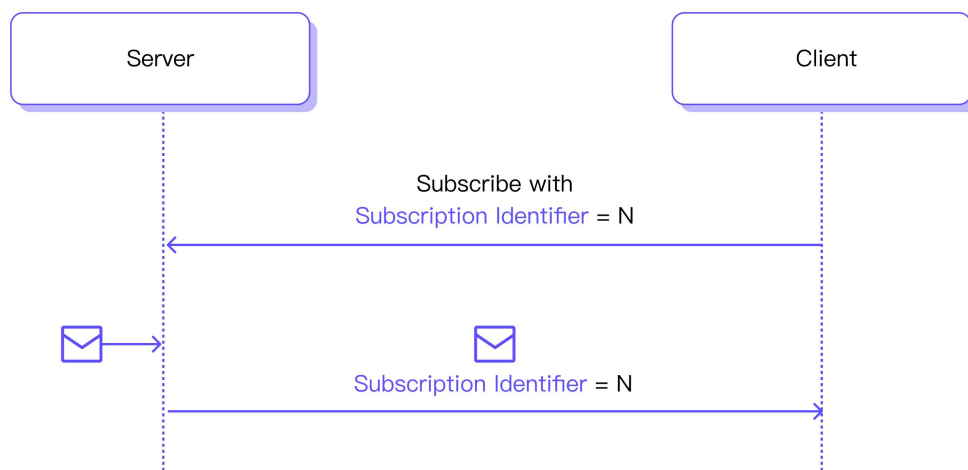
Regardless of whether it's the former or latter option, the client cannot determine which subscription(s) the message originated from. For example, even if the client finds that a message matches two of its subscriptions, it cannot guarantee that both subscriptions have been successfully created when the server forwards the message to itself.

Therefore, the client cannot trigger the correct callback for the message.



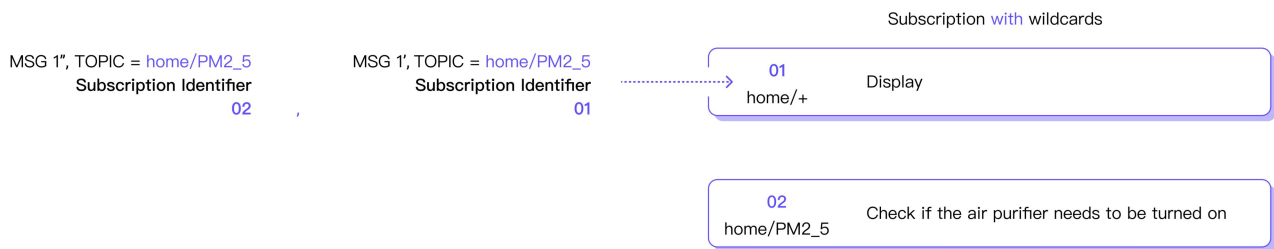
How the Subscription Identifier Works

To address this issue, MQTT 5.0 introduced Subscription Identifiers. Its usage is very simple: clients can specify a Subscription Identifier when subscribing, and the server needs to store the mapping relationship between the subscription and the Subscription Identifier. When a PUBLISH packet matches a subscription and needs to be forwarded to the client, the server will return the subscription identifier associated with the subscription to the client together with the PUBLISH packet.



If the server chooses to send separate messages for overlapping subscriptions, each PUBLISH packet should include the Subscription Identifier that matches the subscription. If the server chooses to send only one message for overlapping subscriptions, the PUBLISH packet will contain multiple Subscription Identifiers.

The client only needs to establish a mapping between Subscription Identifiers and callback functions. By using the Subscription Identifier in the message, the client can determine which subscription the message originated from and which callback function should be executed.

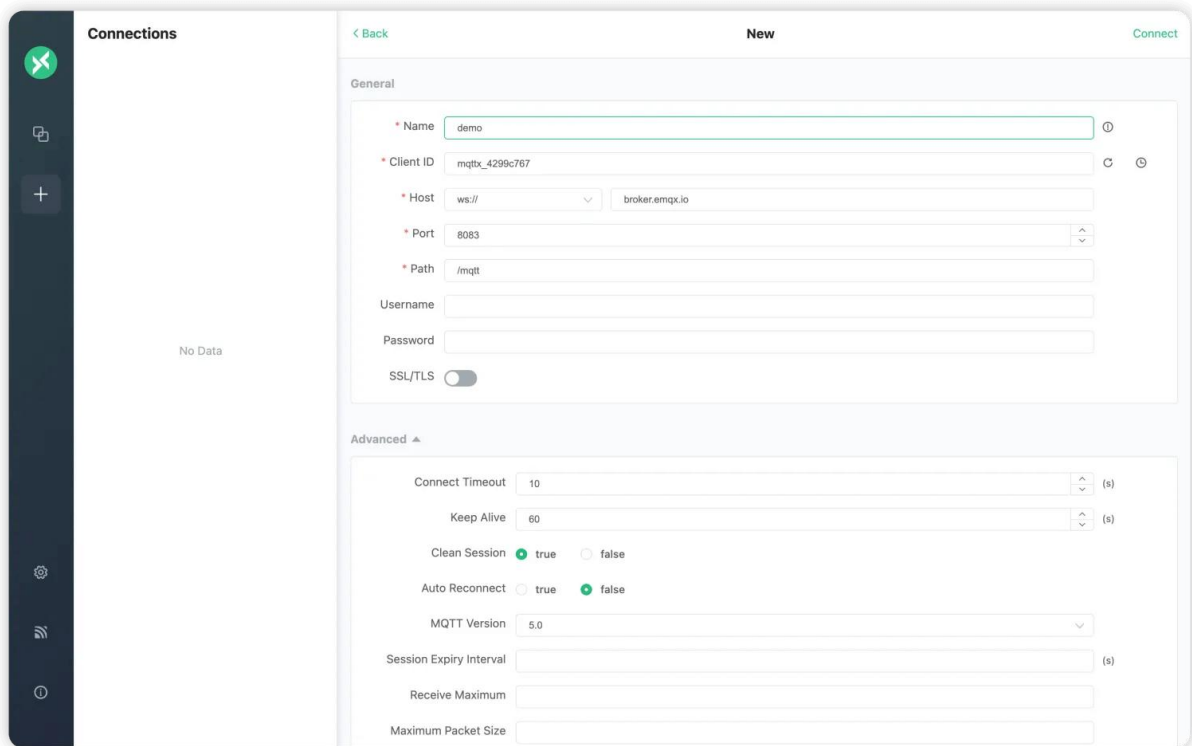


In the client, the Subscription Identifier is not part of the session state, and its association with any content is entirely determined by the client. Therefore, besides callback functions, we can also establish mappings between Subscription Identifiers and subscribed topics, or between Subscription Identifiers and the Client ID. The latter is particularly useful in gateway scenarios where the gateway receives messages from the server and needs to forward them to the appropriate client. With the Subscription Identifier, the gateway can quickly determine which client should receive the message without re-matching and routing the topics.

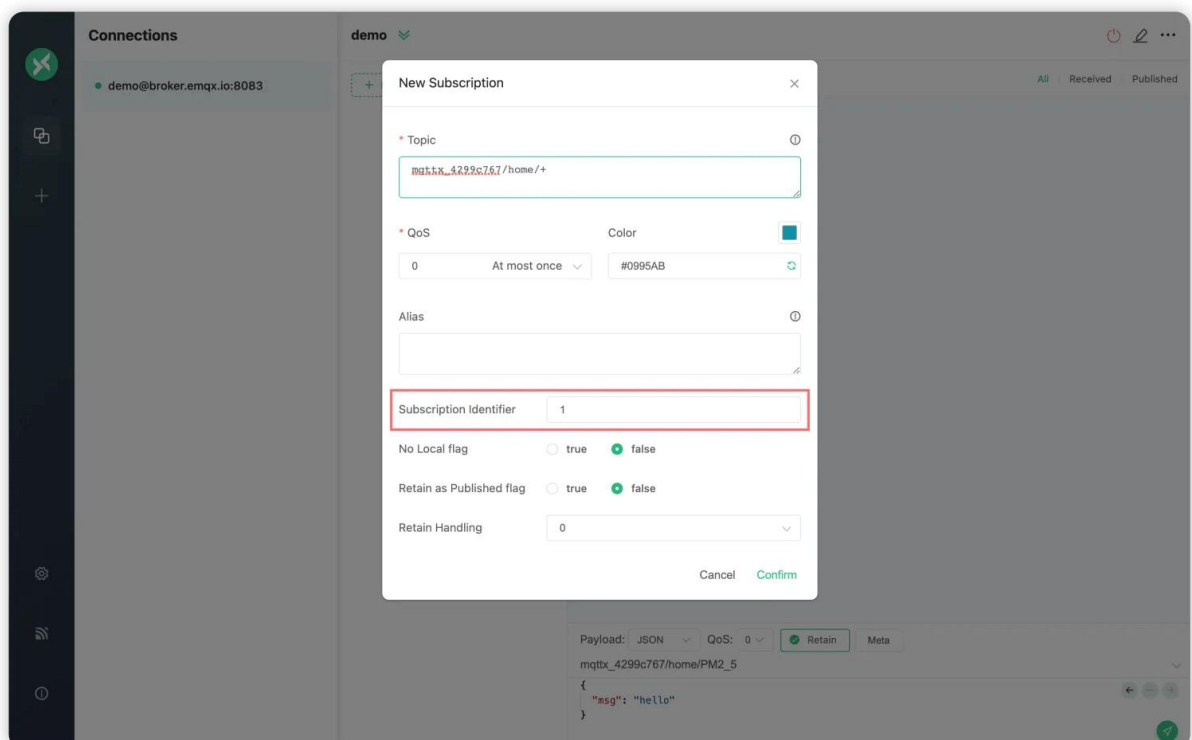
A SUBSCRIBE packet can only contain one Subscription Identifier. If a SUBSCRIBE packet includes multiple subscriptions, the same Subscription Identifier will be associated with all those subscriptions. So, please ensure that associating multiple subscriptions with the same callback function is intentional.

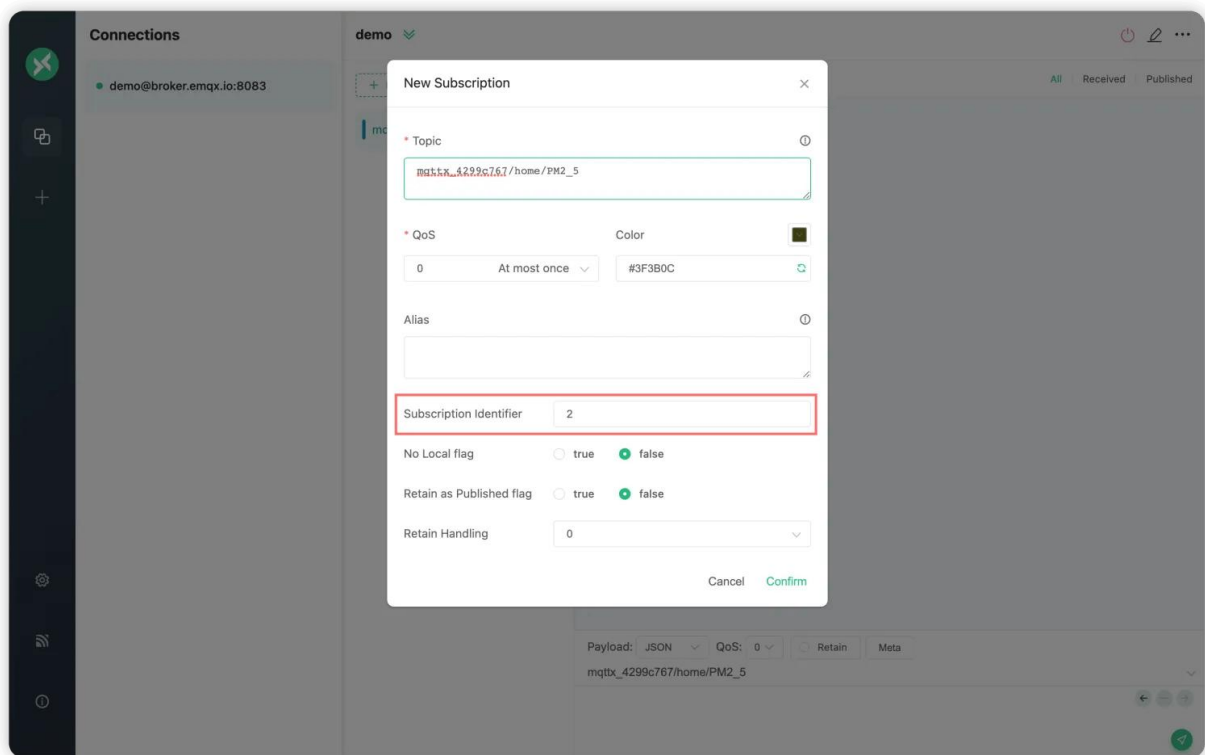
Demo

1. Access [MQTTX Web](#) on a Web browser.
2. Create an [MQTT over WebSocket](#) connection and connect to the [Free Public MQTT Server](#):

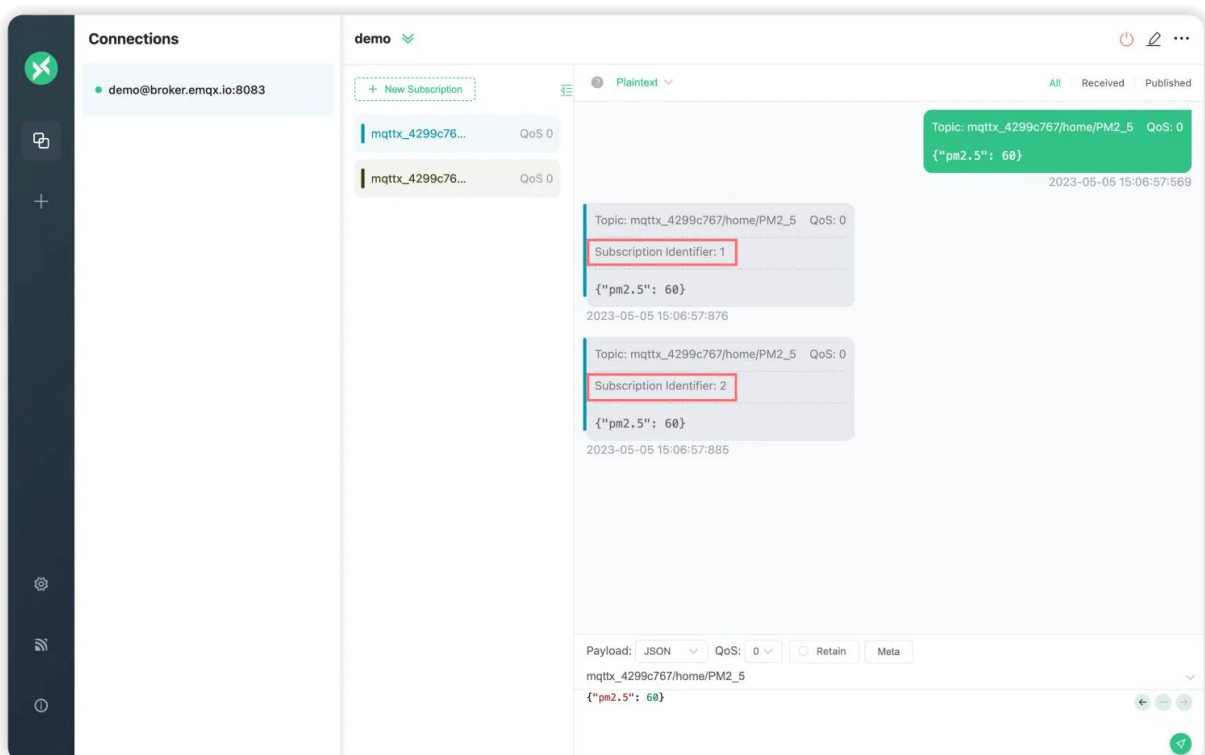


- After a successful connection, we subscribe to the topic `mqttx_4299c767/home/+` and specify the Subscription Identifier as 1. Then, we subscribe to the topic `mqttx_4299c767/home/PM2_5` and specify the Subscription Identifier as 2. Since the public server can be used by many people simultaneously, to avoid topic conflicts, we use the Client ID as the topic prefix:

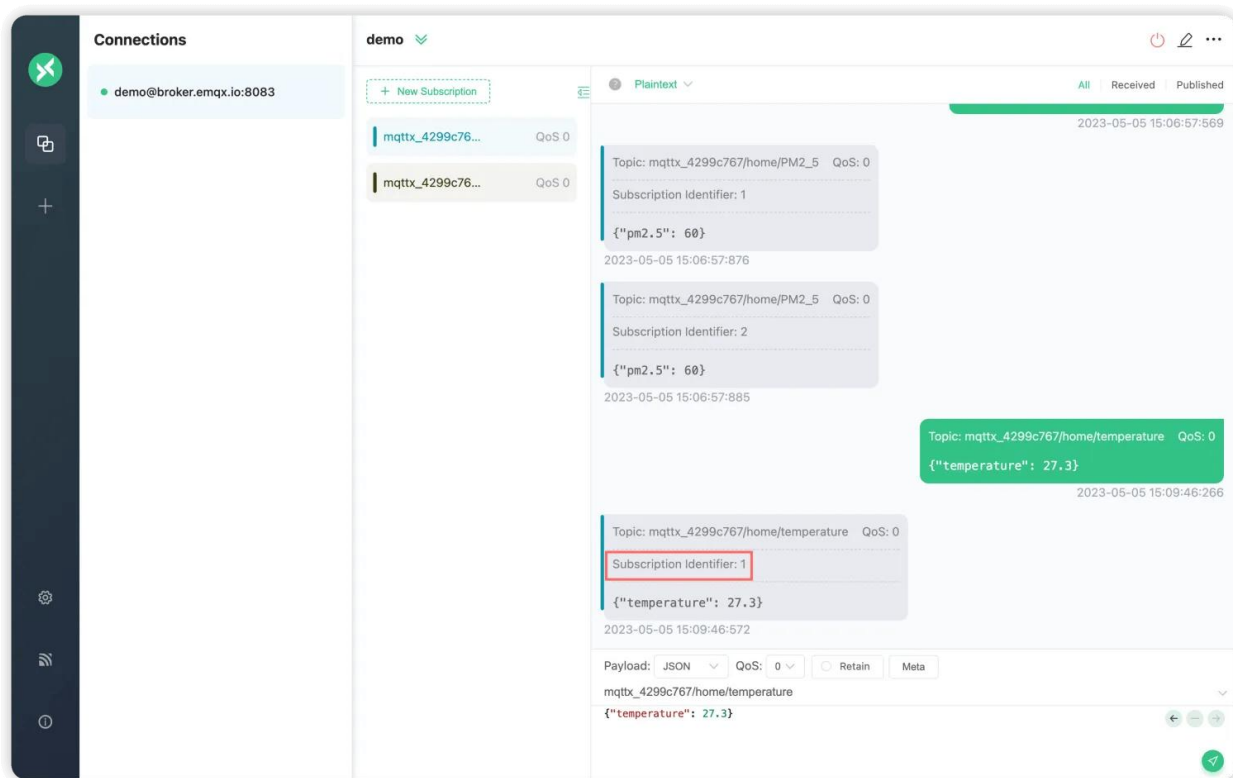




4. After a successful subscription, we publish a message to the topic `mqttx_4299c767/home/PM2_5`. We will observe that the current client receives two messages, and the Subscription Identifier in the messages is 1 and 2, respectively. This is because EMQX's implementation sends separate messages for overlapping subscriptions:



5. And if we publish a message to the topic `mqttx_4299c767/home/temperature`, we will see that the Subscription Identifier in the received message is 1:



So far, we have demonstrated how to set a Subscription Identifier for subscription through [MQTTX](#). If you are still curious about how to trigger different callbacks based on Subscription Identifier, you can get the Python sample code for Subscription Identifier [here](#).

Message Expiry Interval

What is the Message Expiry Interval?

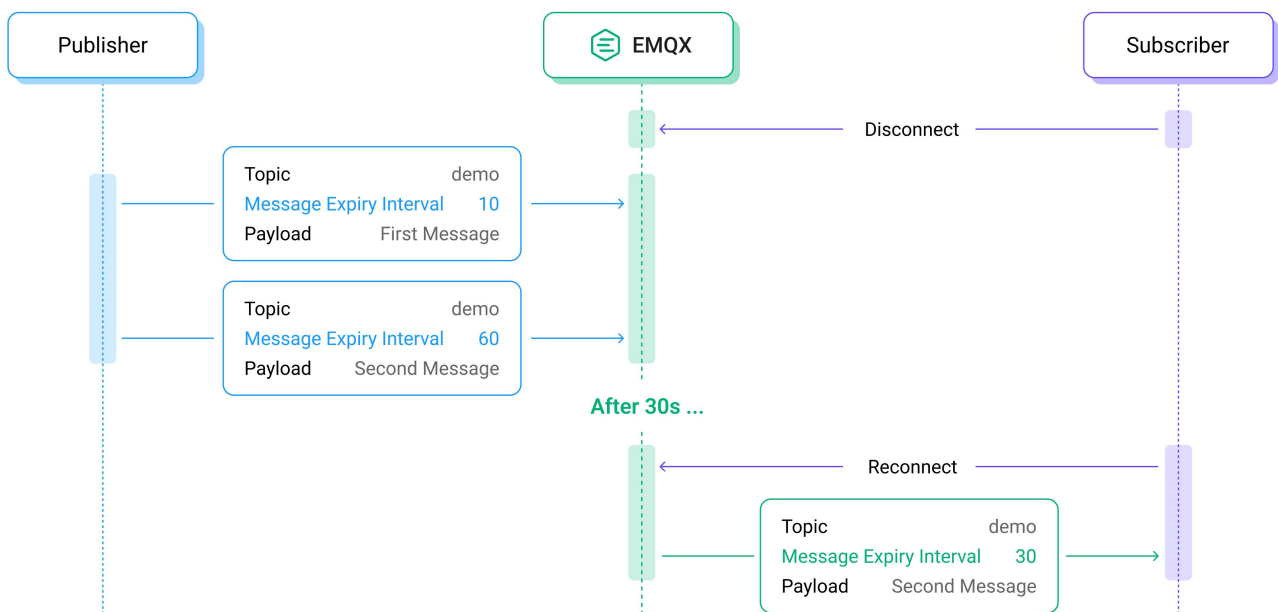
The Message Expiry Interval is a new feature introduced in MQTT 5.0, which allows the publisher to set an expiry interval for time-sensitive messages. If the message remains on the server beyond this specified interval, the server will no longer distribute it to the subscribers. By default, the message does not include the message expiry interval, which means the message will never expire.

MQTT's persistent sessions can cache unsent messages for offline clients and send them when the client reconnects. However, if the client is offline for a long time, there may be some short-lived messages that are no longer necessary to be sent to the client. Continuing to send these expired messages will only waste network bandwidth and client resources.

Take connected cars as an example, we can send suggested driving speeds to the vehicle so it can pass the intersection during the green light. These messages are usually only valid before the vehicle reaches the next intersection, with a very short life cycle. Messages like front congestion alerts have a longer life cycle, generally valid within half an hour to 1 hour.

If the client sets an expiry interval when publishing a message, the server will also include the expiry interval when forwarding this message, but the value of the expiry interval will be updated to the value received by the server minus the time the message stays on the server.

This can prevent the timeliness of the message from being lost during transmission, especially when bridging to another MQTT server.



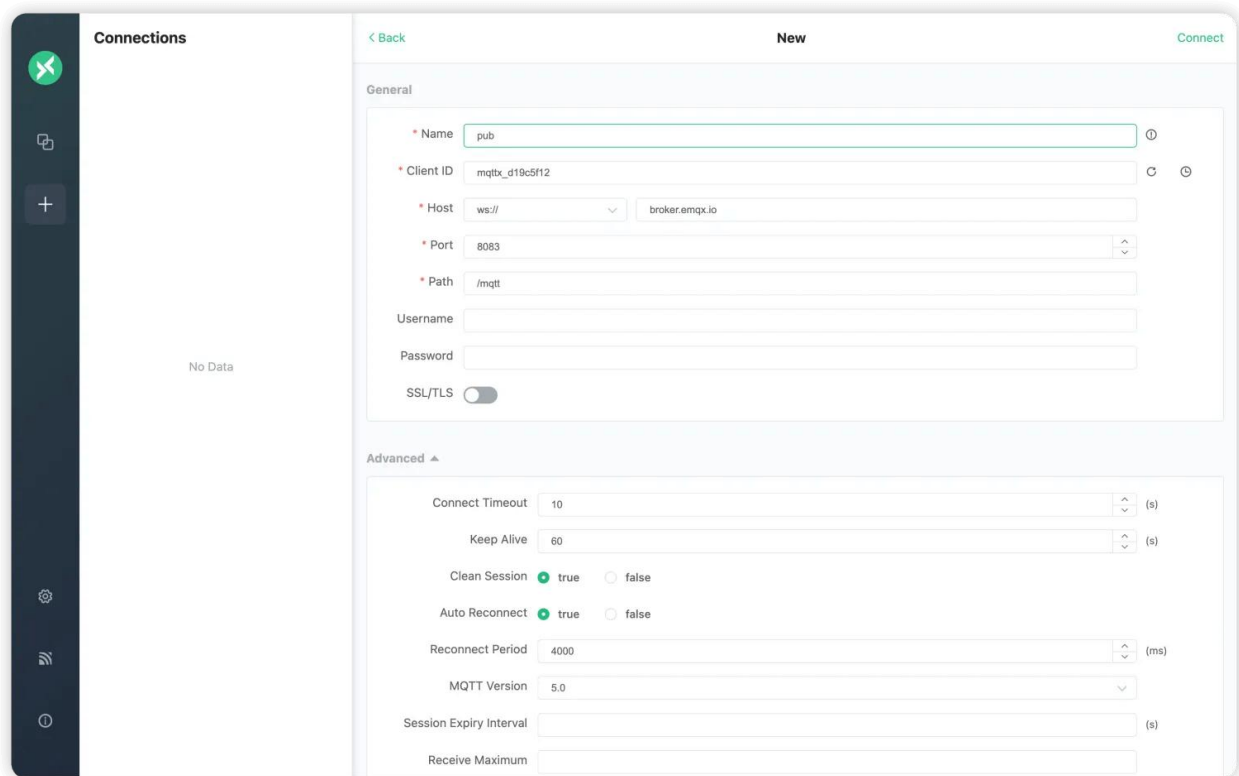
When to Use the Message Expiry Interval?

The Message Expiry Interval is very suitable for use in the following scenarios:

1. Messages that are strongly bound with time. For instance, a message like 'the discount ends in the next two hours'. If the user receives it after two hours, it would be meaningless.
2. Messages periodically inform the latest status. Continuing with the example of road congestion alerts, we need to periodically send vehicles the expected end time of congestion, which changes with the latest road conditions. So when the latest message arrives, there is no need to continue sending the previous unsent messages. In this case, the message expiry interval is determined by our actual sending cycle.
3. Retained messages. Compared to needing to resend a retained message with an empty Payload to clear the retained message under the corresponding topic, it is obviously more convenient to set an expiration time for it and then have the server automatically delete it, which can also effectively avoid retained messages occupying too much storage resources.

Demo

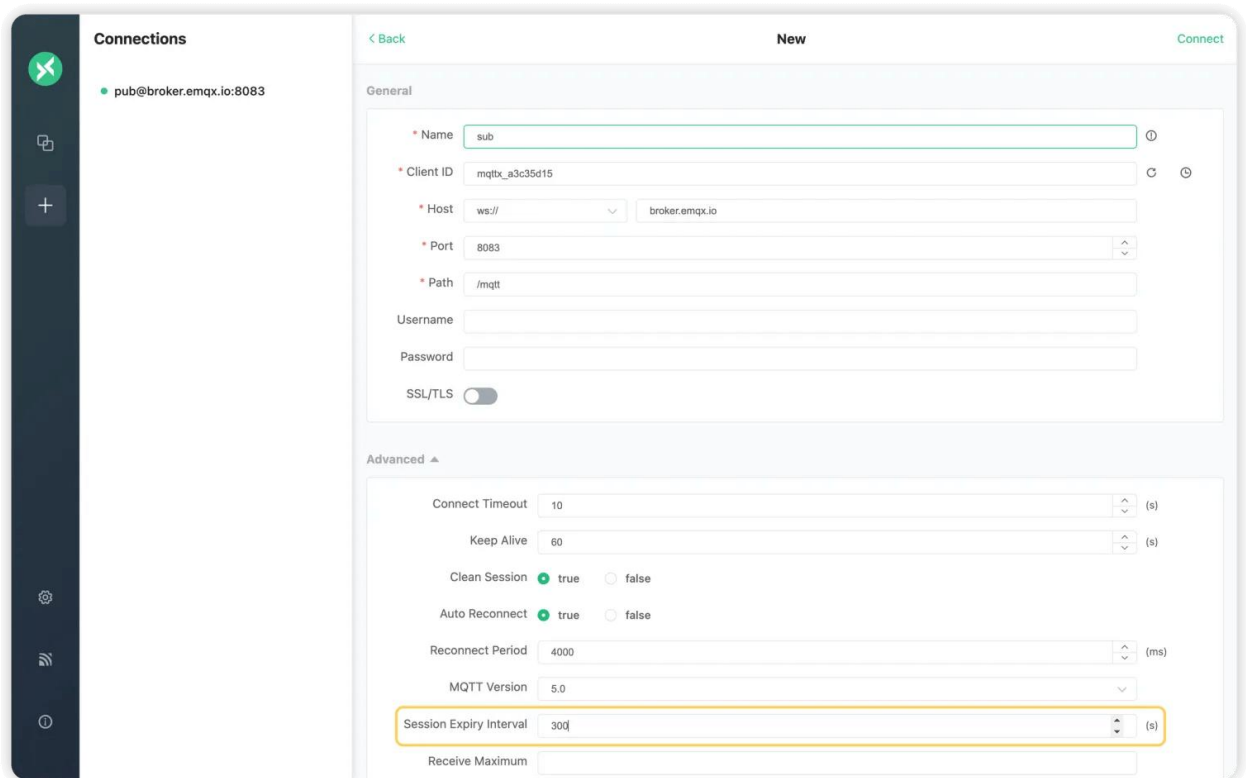
1. Access [MQTTX Web](#) on a Web browser.
2. Create an MQTT connection named `pub` for publishing messages, and connect it to the [Free Public MQTT Server](#):



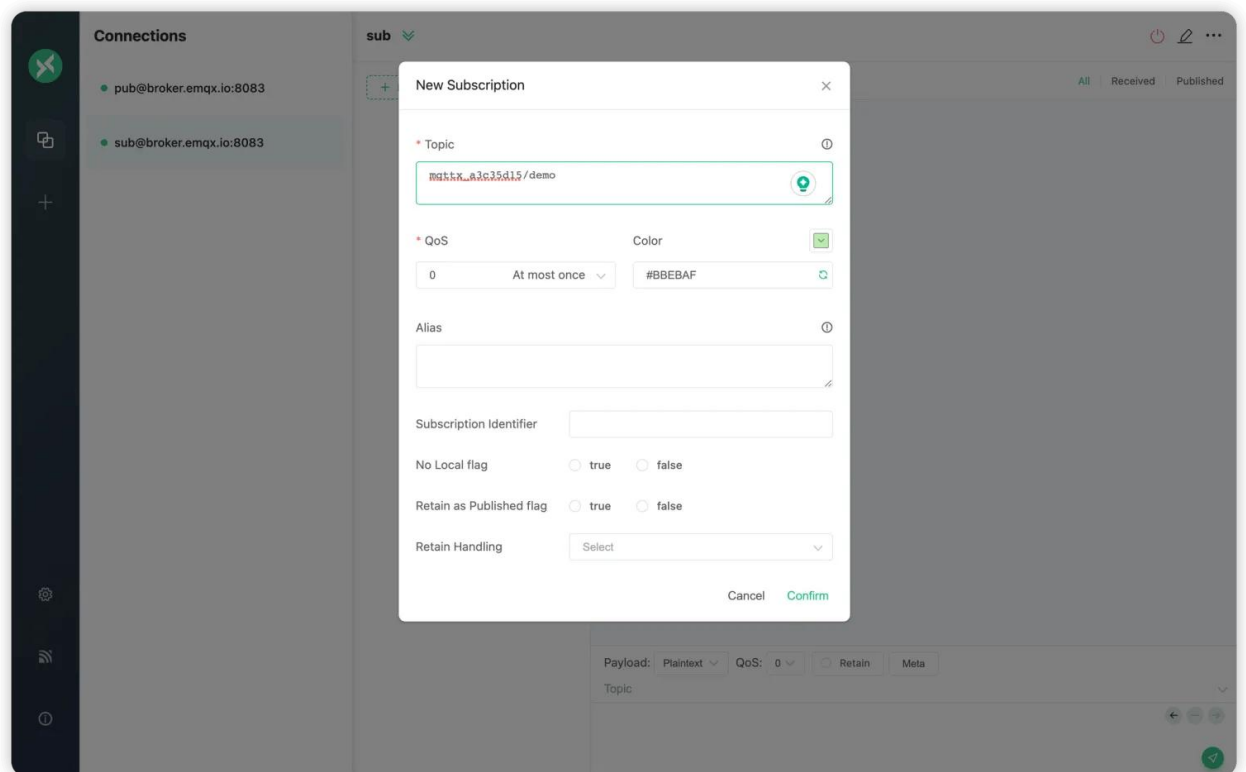
The screenshot displays the MQTTX Web interface. On the left is a dark sidebar with icons for connections, a plus sign, settings, and a refresh button. The main area is titled 'Connections' and shows 'No Data'. To the right, a 'New' connection form is open. The 'General' section includes fields for Name (pub), Client ID (mqttc_d19c5f12), Host (ws://broker.emqx.io), Port (8083), Path (/mqtt), Username, Password, and an SSL/TLS toggle. The 'Advanced' section includes Connect Timeout (10s), Keep Alive (60s), Clean Session (true), Auto Reconnect (true), Reconnect Period (4000ms), MQTT Version (5.0), Session Expiry Interval, and Receive Maximum.

Field	Value
Name	pub
Client ID	mqttc_d19c5f12
Host	ws://broker.emqx.io
Port	8083
Path	/mqtt
Username	
Password	
SSL/TLS	Off
Connect Timeout	10 (s)
Keep Alive	60 (s)
Clean Session	True
Auto Reconnect	True
Reconnect Period	4000 (ms)
MQTT Version	5.0
Session Expiry Interval	
Receive Maximum	

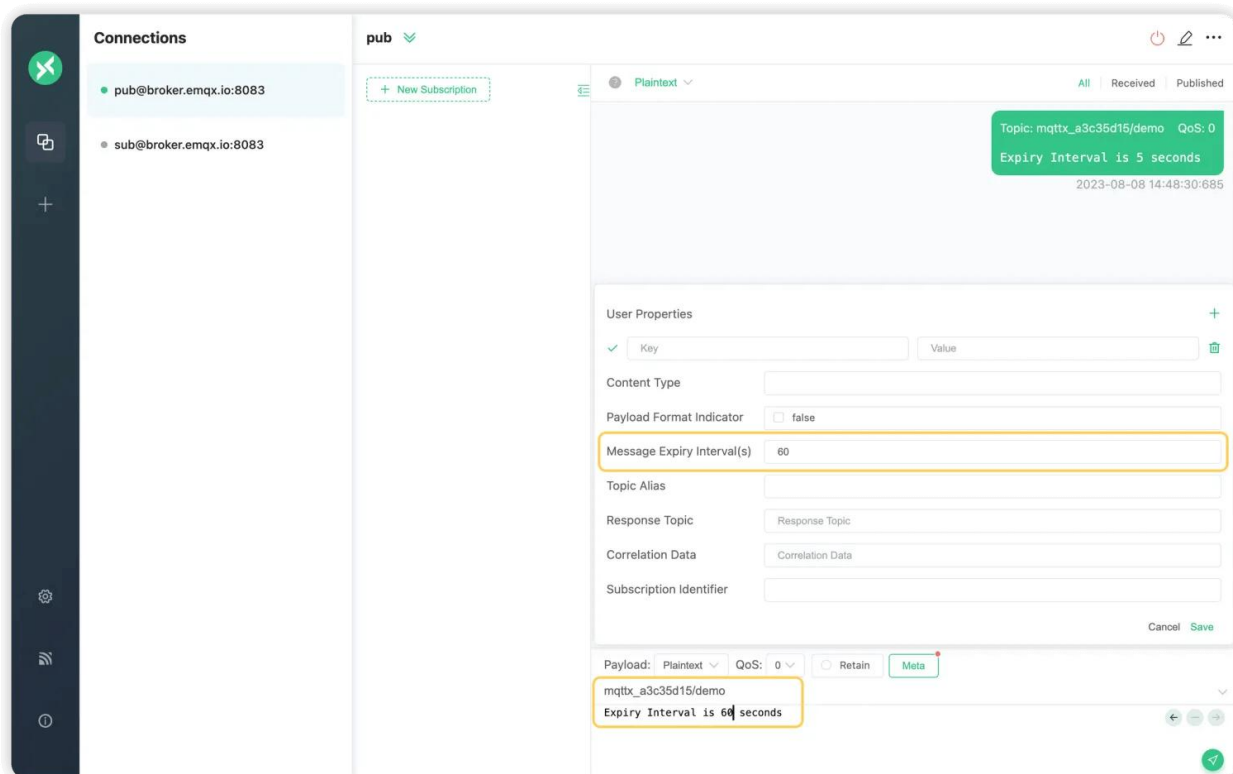
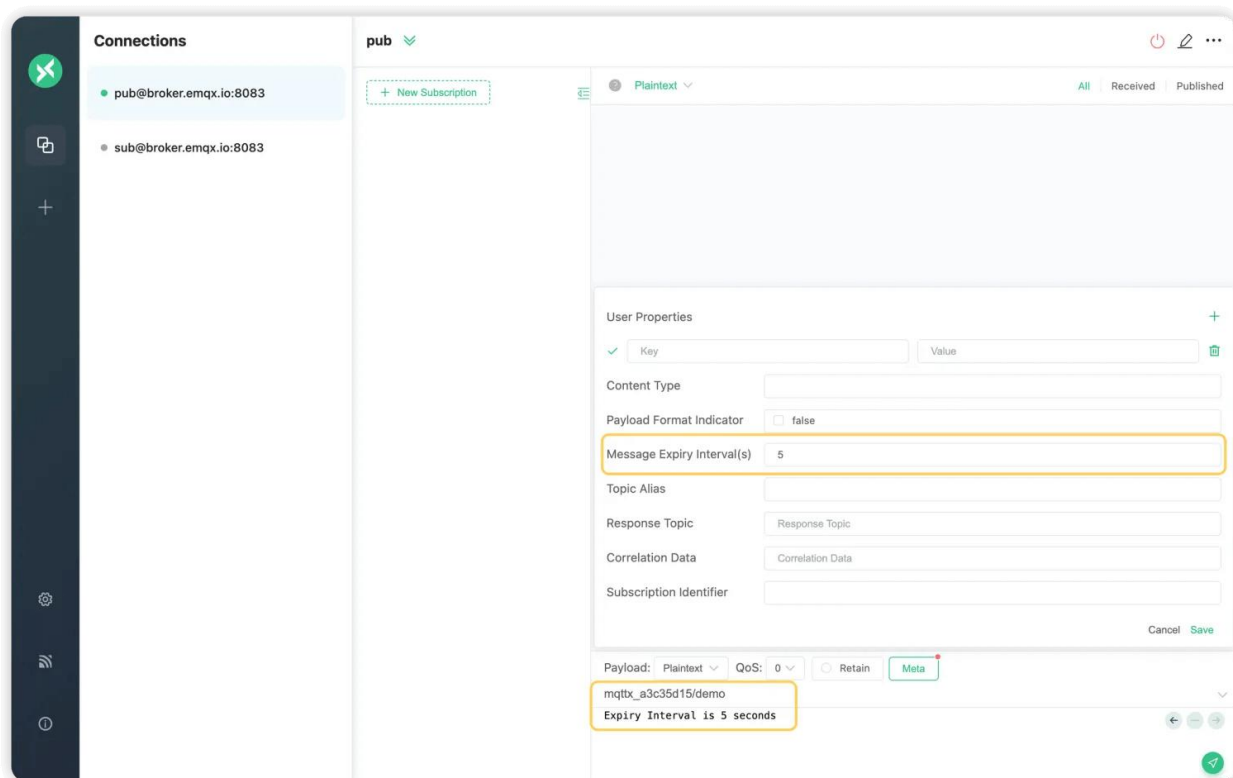
3. Create a new MQTT connection named `sub` for subscribing, and set the Session Expiry Interval to 300 seconds to indicate that it requires a persistent session:



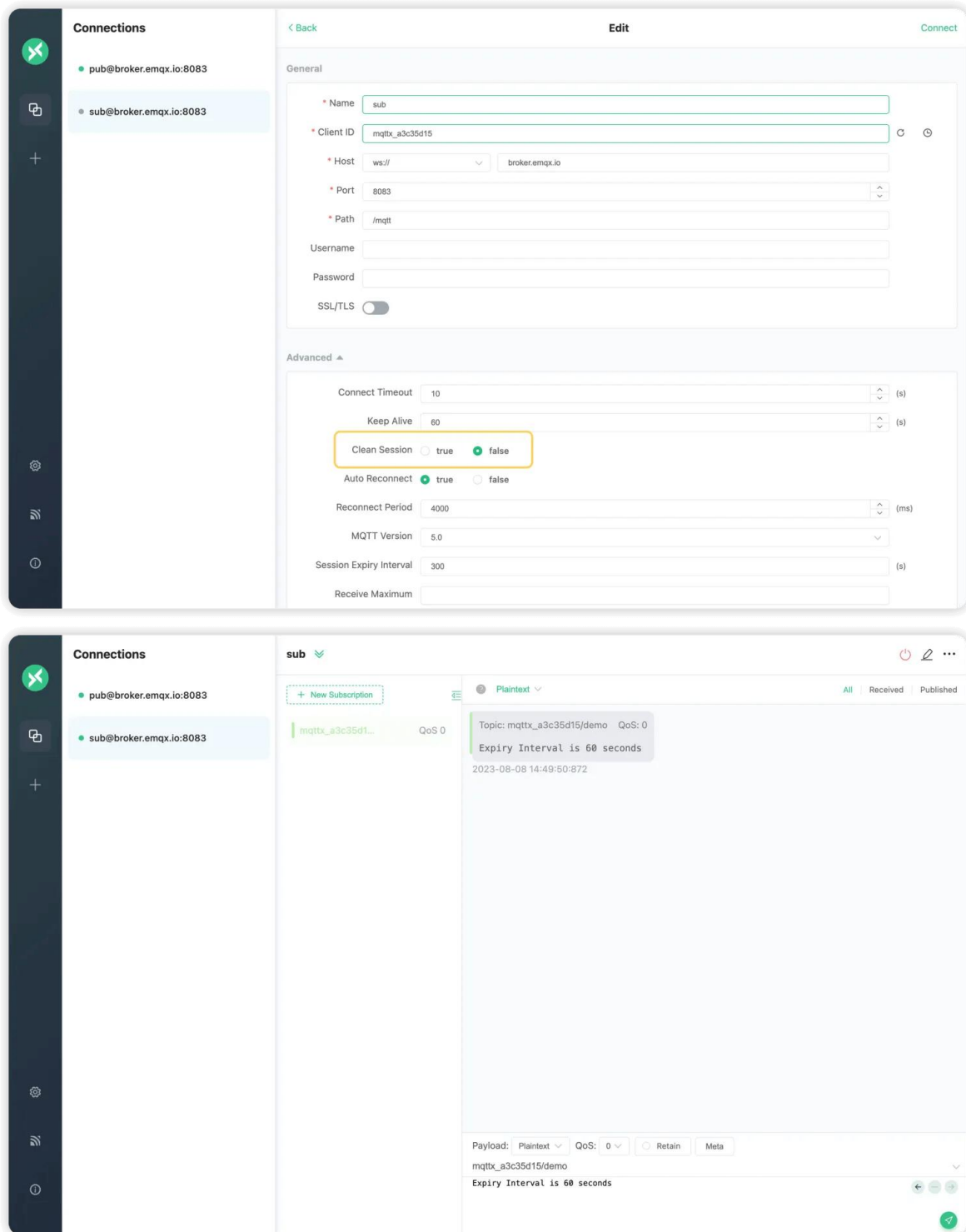
4. After successfully connecting, we subscribe to the topic `mqttx_a3c35d15/demo`, using the Client ID as the topic prefix can effectively avoid duplication with the topics used by other users in the public server:



5. After successfully subscribing, we disconnect the sub client from the server, then switch to the pub client, and publish the following two messages with Message Expiry Intervals of 5 seconds and 60 seconds respectively to the topic `mqttx_a3c35d15/demo`:



6. After publishing, switch to the **sub** client, set Clean Session to false to indicate the desire to restore the previous session, then wait at least 5 seconds before reconnecting. We will see that **sub** only received the message with an expiry time of 60 seconds because another message has already expired by this time:



You can get the Python sample code for Message Expiry Interval [here](#).

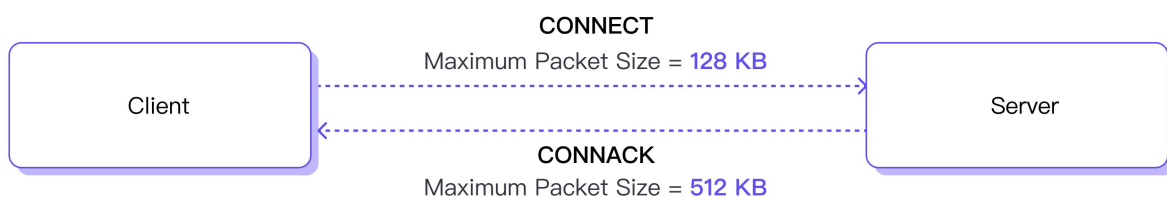
Maximum Packet Size

What is the Maximum Packet Size?

The theoretical maximum length of an [MQTT packet](#) is 268,435,456 bytes, equivalent to 256 MB. However, it is evident that resource-constrained clients and some MQTT servers operating as edge gateways may not be able to handle packets of this size.

Considering that different clients may have significantly different capabilities in handling packets, sending excessively large packets can not only affect the normal business processing of the peer but may also directly overwhelm the peer. Therefore, we need to use the Maximum Packet Size property to negotiate the maximum packet length that the client and server can handle.

The client first specifies the maximum allowed packet length that the server can send to it by using the Maximum Packet Size in the CONNECT packet. On the other hand, the server specifies the maximum allowed packet length that the client can send to it using the Maximum Packet Size in the CONNACK packet.



Once the connection is established, both parties must adhere to this agreement when sending packets. Neither party is allowed to send packets that exceed the agreed length limit. Otherwise, the receiver will return a DISCONNECT packet with a Reason Code of 0x95 and close the network connection.

It is important to note that if the client sets a [Will message](#) in the CONNECT packet, it may unknowingly cause the CONNECT packet to exceed the maximum packet length allowed by the server. In such cases, the server will respond with a CONNACK packet with a Reason Code of 0x95 and close the network connection.

How Does the Sender Work Within the Limit?

For the client, whether it is publishing or subscribing, as the active sender, it can split a packet into multiple parts to be sent in order to avoid exceeding the length limit.

But for the server, it is only responsible for forwarding the message, and cannot determine the size of the message. So if it finds that the size of the message to be forwarded exceeds the maximum value that the client can receive, then it will drop this message. If it is a [shared subscription](#), besides dropping, the server can also choose to send the message to other clients in the group that can receive the message.

In addition to the two strategies mentioned above, whether it is the client or the server, they can trim the packet's content to reduce the length. We know that the responder can include two properties, User Property and Reason String, in response packets such as CONNACK and PUBACK, to convey additional information to the peer.

However, the possibility of exceeding the maximum length limit in response packets is precisely caused by these two properties. Obviously, the priority of transmitting these properties is lower than ensuring the proper flow of the protocol. Therefore, the responder can remove these two properties from the packet when the length of the packet exceeds the limit, to maximize the chances of successfully transmitting the response packet.

I want to let you know that this only applies to response packets, not to PUBLISH packets. For PUBLISH packets, User Property is considered part of the packet, and the

server should not attempt to remove it from the packet to ensure packet delivery.

Demo

1. Open the installed [MQTTX](#).
2. Create an MQTT connection, set the Maximum Packet Size to 100, and connect to the [Free Public MQTT Server](#):

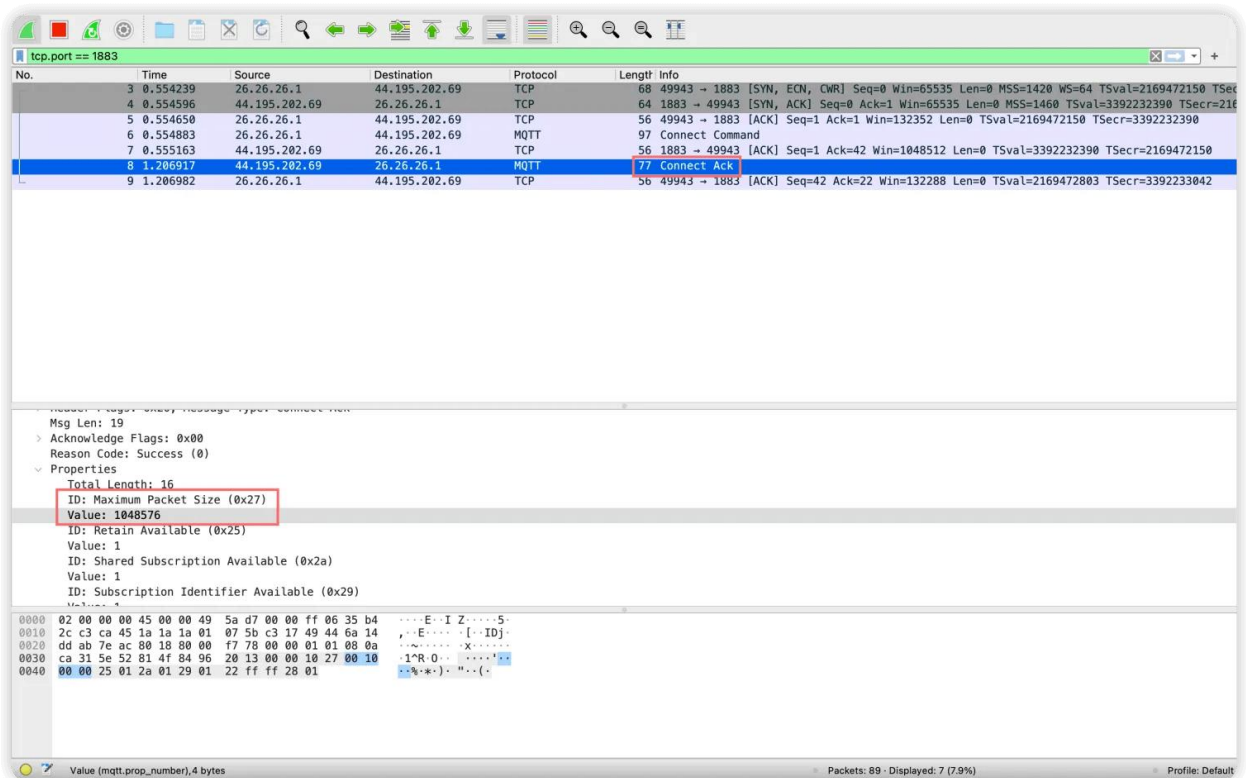
The screenshot shows the MQTTX application interface. On the left is a dark sidebar with icons for Connections, Topics, Messages, Settings, and Help. The main area is titled 'Connections' and shows a 'New' connection configuration screen. The 'General' tab is active, displaying the following fields:

- Name: demo
- Client ID: mqttx_0c668d0d
- Host: mqtt:// broker.emqx.io
- Port: 1883
- Username: (empty)
- Password: (empty)
- SSL/TLS: (disabled)

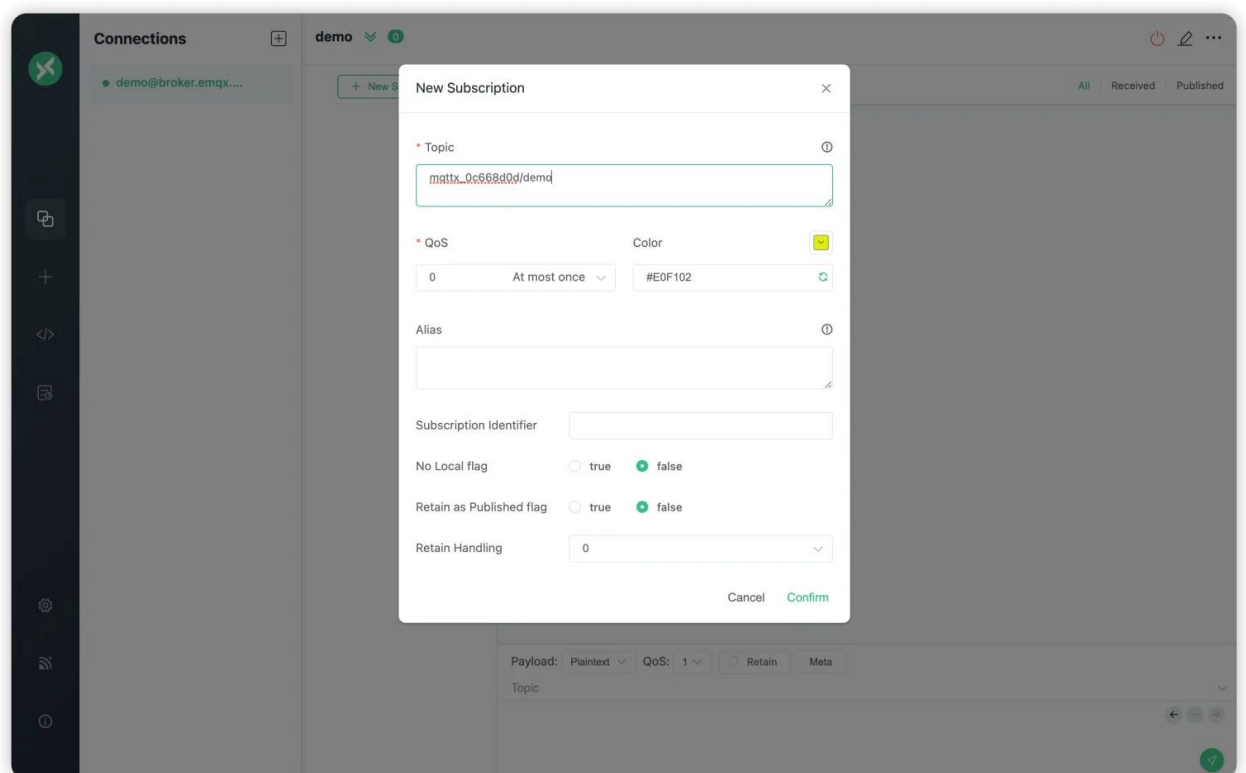
The 'Advanced' tab is also visible, showing the following settings:

- MQTT Version: 5.0
- Connect Timeout: 10 (s)
- Keep Alive: 60 (s)
- Auto Reconnect: (disabled)
- Clean Start: (enabled)
- Session Expiry Interval: 0 (s)
- Receive Maximum: (empty)
- Maximum Packet Size: 100 (highlighted with a red box)
- Topic Alias Maximum: (empty)

3. After a successful connection, we can observe through the Wireshark packet capture tool that the Maximum Packet Size property in the CONNACK packet returned by the server is 1048576. This means that the client can only send a packet of up to 1 KB to the public MQTT server each time:

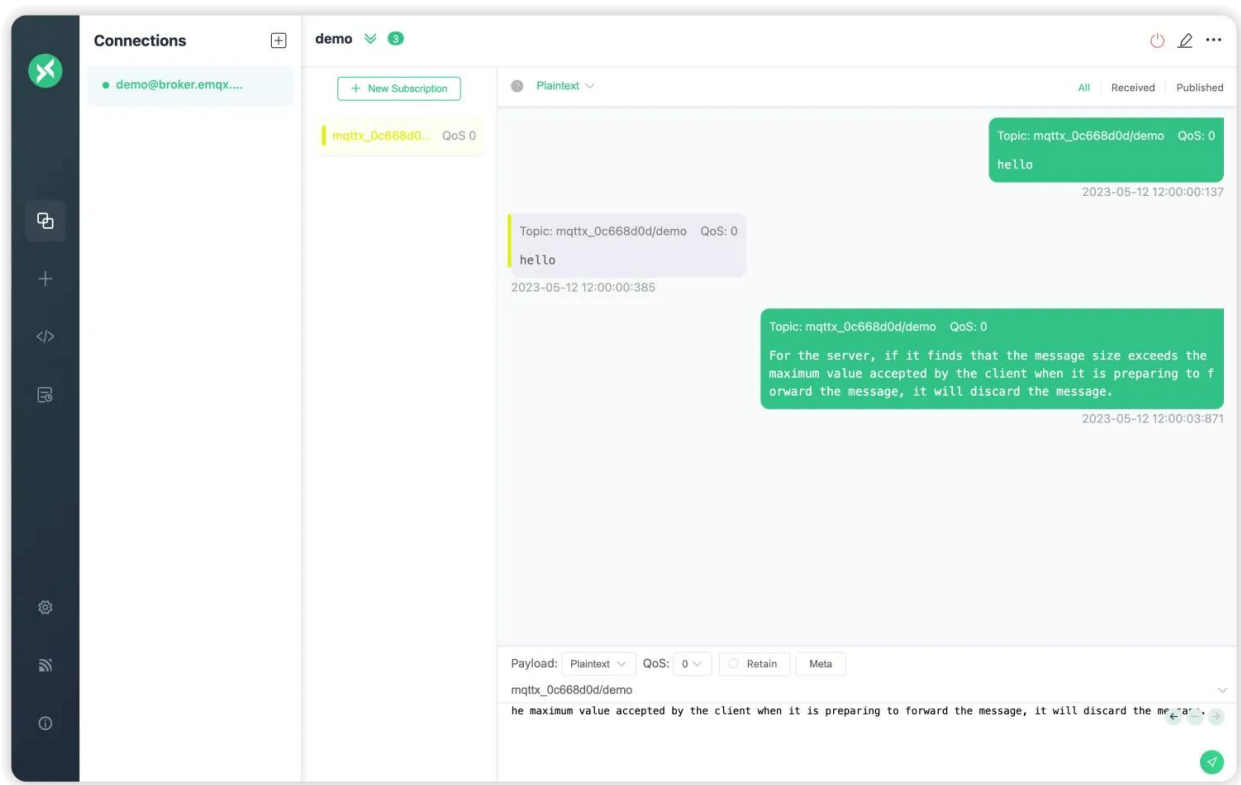


4. Go back to MQTTX and subscribe to the topic `mqttx_0c668d0d/demo`:



5. Then we publish two messages to the topic `mqttx_0c668d0d/demo`, one with a length of 5 bytes and another with a size of 172 bytes. We will observe that only the

message with a length of 5 bytes will be received, while another message exceeding the 100-byte length limit will not be forwarded by the server:



Reason Code

The primary purpose of Reason Code in [MQTT](#) is to provide more detailed feedback to the client and server. For example, we can feed back the Reason Code corresponding to the wrong username or password to the client in the CONNACK packet, so that the client can know why it cannot connect.

Reason Code in MQTT 3.1.1

Although MQTT 3.1.1 already supports Reason Code, it doesn't define many available Reason Codes. In fact, they are very limited.

Among the two [MQTT packets](#) that support Reason Code, the CONNACK packet has only 5 Reason Codes to indicate failures. And the SUBACK packet has only one Reason Code for indicating failure, it cannot even provide further details about the reason for the subscription failure. Moreover, for operations such as publishing and unsubscribing that don't support Reason Code, we can't even determine whether the operation was successful or not. This makes development debugging unfriendly and hinders the implementation of robust code.

Reason Code in MQTT 5.0

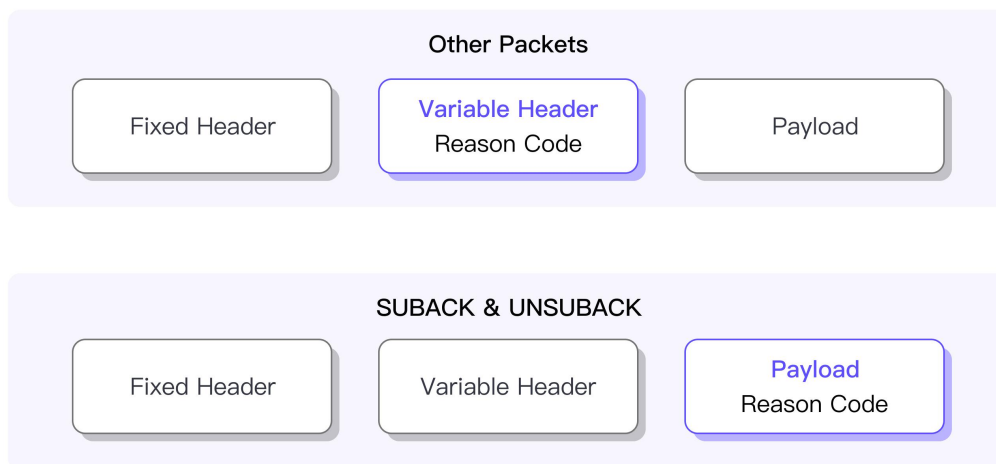
So In MQTT 5.0, the number of available Reason Codes has been expanded to 43, and it is specified that Reason Codes below 0x80 are used to indicate success, while Reason Codes greater than or equal to 0x80 are used to indicate failure. Unlike MQTT 3.1.1, where Reason Codes below 0x80 could also indicate failure. This allows clients to determine whether an operation was successful or not quickly.

Furthermore, MQTT 5.0 extends the support for Reason Codes to the following packets:

CONNACK, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBACK, UNSUBACK, DISCONNECT, and AUTH. Now, we can not only determine if a message publication was successful but also understand the reasons for failure, such as the absence of matching subscribers or insufficient privileges to publish to a specific topic.

	Reason Codes in MQTT 3.1.1	Reason Codes in MQTT 5.0
CONNACK	✓	✓
DISCONNECT	✗	✓
PUBACK	✗	✓
PUBREC	✗	✓
PUBREL	✗	✓
PUBCOMP	✗	✓
SUBACK	✓	✓
UNSUBACK	✗	✓
AUTH	—	✓

Most MQTT packets include only a single Reason Code, except for SUBACK and UNSUBACK. This is because the SUBSCRIBE and UNSUBSCRIBE packets can contain multiple topic filters, and each topic filter must have a corresponding Reason Code to indicate the result of the operation. Therefore, SUBACK and UNSUBACK packets need to accommodate multiple Reason Codes. That's why Reason Codes in other packets are located in the Variable Header, while the Reason Codes in SUBACK and UNSUBACK are located in the Payload.



Indicate to the client why the connection was disconnected

In MQTT 3.1 and 3.1.1, the DISCONNECT packet could only be sent by the client. Therefore, when a client violated certain restrictions, the server could only directly close the network connection without providing additional information to the client. This made it challenging to investigate the reason for the disconnection.

Now in MQTT 5.0, the server can send a DISCONNECT packet to the client before closing the network connection. The client can use the Reason Code in the DISCONNECT packet to get why the connection was disconnected, such as the packet being too large, the server being busy, and so on.

Reason String

Reason String is a complement to Reason Code in MQTT 5.0, providing a human-readable string designed for diagnostic purposes. While Reason Code can indicate most error reasons, developers or operators may still need more intuitive contextual information.

For example, when the server indicates an invalid topic filter to the client using the Reason Code (0x8F), developers still have no specific information about the reason. Is it

due to exceeding the maximum number of topic levels or including characters not accepted by the server? However, if the server can return a Reason String similar to "The number of topic levels exceeds the limit, the maximum is 10," developers can quickly understand the reason and make adjustments.

In practical use, the content of the Reason String depends on the specific implementation of the client and server. Therefore, a correctly implemented receiving end should not attempt to parse the content of the Reason String. Recommended usage includes but is not limited to using the Reason String when throwing exceptions or writing it to logs.

Finally, Reason String is an optional feature, and whether a Reason String is received depends on whether the peer supports it.

You can find a Reason Code quick reference table [here](#) to learn about detailed explanations and use cases for each Reason Code in MQTT 5.0.

Enhanced Authentication

What is Enhanced Authentication?

Enhanced authentication is a novel authentication framework introduced in MQTT 5.0. It offers a range of alternative methods that are more secure than traditional password authentication.

However, increased security comes with added complexity. Certain authentication methods, like SCRAM, require multiple exchanges of authentication data. This renders the single-exchange authentication framework of the CONNECT and CONNACK packets outdated. To address this limitation, MQTT 5.0 introduces the AUTH packet, which supports multiple exchanges of authentication data. It enables the use of [SASL](#) (Simple Authentication and Security Layer) mechanisms with a challenge-response style in [MQTT](#).

What Problems Does Enhanced Authentication Solve?

Before delving into enhanced authentication, it is essential to understand the shortcomings of password authentication in terms of security.

In fact, despite employing techniques like Salt and Hash to store passwords securely, the client must transmit the password in plain text over the network, making it vulnerable to theft. Even when employing TLS encryption for communication, there remains a risk of attackers obtaining sensitive data like passwords due to outdated SSL versions, weak cipher suites, or the presence of fake CA certificates.

Moreover, simple password authentication only lets the server check the identity of the client, but not the other way around, which allows the attacker to pretend to be the

server and get sensitive data from the client. This is what we often call a man-in-the-middle attack.

Enhanced authentication allows users to employ highly secure authentication methods within the SASL framework. These methods offer several advantages, such as eliminating the transmission of passwords over the network and facilitating mutual identity verification between the client and server. By presenting these options, users can select the authentication method that aligns with their specific needs and security preferences.

Common SASL Mechanisms Used for Enhanced Authentication

DIGEST-MD5

DIGEST-MD5 is an authentication method within the SASL framework. It utilizes the Message Digest 5 (MD5) hash algorithm and a challenge-response mechanism to verify the identity between the client and the server. One notable advantage is that the client does not need to transmit the password in plain text over the network.

In simple terms, when a client wants to access a protected resource, the server will send a challenge with a one-time random number and some required parameters. The client utilizes these parameters, along with its username and password, to generate a response, which is then transmitted back to the server. The server independently creates an expected response using the same method and compares it with the received response. If they match, authentication is successful. This approach effectively mitigates the risk of password exposure through network snooping. Additionally, by utilizing a one-time random number for each connection, it enhances protection against replay attacks.

However, it's important to note that DIGEST-MD5, while enabling server-side verification of the client's identity, lacks the ability for the client to verify the identity of the server. This limitation leaves room for potential man-in-the-middle attacks. Furthermore, since MD5 is no longer secure, it is strongly recommended to replace it with a hash function that offers stronger resistance to collisions, such as SHA-256.

SCRAM

SCRAM (Salted Challenge Response Authentication Mechanism) is another authentication method within the SASL framework. It shares similarities with DIGEST-MD5 in terms of approach. SCRAM prompts the client to generate a response using a one-time random number, thereby avoiding sending the password in plain text over the network. However, SCRAM further enhances security by incorporating Salt, Iterations, and more robust hash algorithms like SHA-256 and SHA-512. These additions significantly enhance the security of password storage, effectively mitigating the risks associated with offline attacks, replay attacks, and other potential vulnerabilities.

Furthermore, SCRAM incorporates a more intricate challenge-response process that includes server-side proof sent to the client. The client can then utilize this proof to verify the server's possession of the correct password, enabling mutual authentication. This additional step reduces the vulnerability to man-in-the-middle attacks.

However, using hash algorithms like SHA256 in SCRAM introduces additional computational overhead, which can potentially impact the performance of devices with limited resources.

Kerberos

Kerberos utilizes a trusted third-party Kerberos server to facilitate authentication

services. The server issues tokens to verified users, enabling them to access resource servers. A notable advantage is the ability for users to access multiple systems and services with a single authentication, thereby achieving the convenience of single sign-on (SSO).

The token issued by the Kerberos server has a limited lifespan, and clients can only use this token to access the service for a certain period, which can prevent security issues caused by token leakage. Of course, although a shorter lifespan can enhance security, it sacrifices some convenience. Users need to make their own trade-offs.

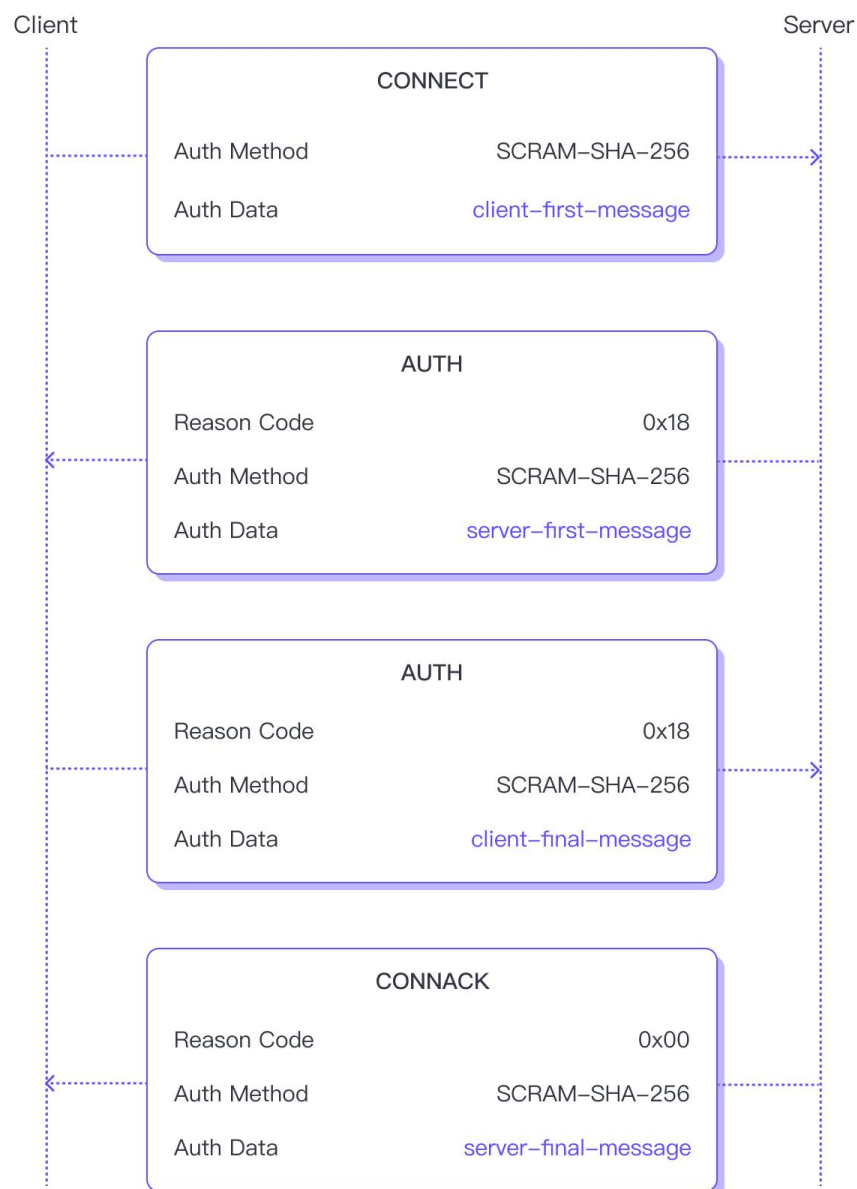
At the core of Kerberos lies the utilization of a symmetric encryption algorithm. The server employs locally stored password hashes to encrypt the authentication data, which is then transmitted to the client. The client, in turn, hashes its own password and utilizes it to decrypt the received authentication data. This process offers several advantages, including the elimination of the need to transmit passwords in plain text over the network and enabling mutual verification of the correct password between the server and client. Additionally, through symmetric encryption, the server and client can securely share session keys, which can be utilized for subsequent encrypted communication. Therefore, Kerberos also provides security measures for protecting subsequent communications beyond authentication.

While providing strong security, Kerberos also brings significant complexity. Implementing and configuring Kerberos comes with its own challenges, and its reliance on up to six handshakes can introduce requirements for high network latency and reliability. As a result, Kerberos is typically employed within the internal network environments of enterprises.

How Does Enhanced Authentication Work in MQTT?

Let's examine how enhanced authentication works in MQTT using the SCRAM as an example. While this section will not delve into the specific principles of SCRAM, it's important to note that SCRAM requires the following four messages to complete authentication:

- client-first-message
- server-first-message
- client-final-message
- server-final-message



To initiate SCRAM authentication, the client sends a CONNECT packet with the Authentication Method attribute set to SCRAM-SHA-256, indicating the intention to use SCRAM authentication. SHA-256 indicates the hash function to be used. The Authentication Data attribute is used to store the content of the client-first message. The Authentication Method attribute determines how the server should parse and process the data contained in the Authentication Data field.

If the server does not support SCRAM authentication, or if the content of the client-first message is found to be invalid, it will return a CONNACK packet containing a Reason Code indicating the reason for authentication failure, and then close the network connection.

Otherwise, the server will proceed with the next step: return an AUTH packet and set Reason Code to 0x18, indicating continued authentication. The Authentication Method in the packet will be the same as the CONNECT packet, and the Authentication Data attribute will contain the content of server-first message.

After verifying that the content of the server-first message is correct, the client also returns an AUTH packet with Reason Code 0x18, and the Authentication Data attribute will contain the content of client-final message.

After verifying that the content of the client-final message is correct, the server has completed the verification of the client's identity. So this time, the server will not return an AUTH packet, but a CONNACK packet with Reason Code 0 to indicate successful authentication, and pass the server-final-message through the Authentication Data attribute in the packet.

If the server's identity is successfully verified, the client can proceed to subscribe to topics or publish messages. However, if the verification fails, the client will send a DISCONNECT packet to terminate the connection.

Control Packets

What are MQTT Control Packets?

MQTT control packets are the smallest unit of data transfer in [MQTT](#). [MQTT clients](#) and servers exchange control packets for performing their work, such as subscribing to topics and publishing messages.

Currently, MQTT defines 15 types of control packets. If we classify them based on their functionality, we can categorize these packets into three categories: connection, publishing, and subscribing.

Connect	Publish	Subscribe
CONNECT	PUBLISH	SUBSCRIBE
CONNACK	PUBACK	SUBACK
DISCONNECT	PUBREC	UNSUBSCRIBE
AUTH Only MQTT 5.0	PUBREL	UNSUBACK
PINGREQ	PUBCOMP	
PINGRESP		

Among them, the **CONNECT** packet is used by the client to initiate a connection to the server, and the **CONNACK** packet is sent as a response to indicate the result of the connection. If one wants to terminate the communication or encounters an error that requires terminating the connection, the client and server can send a **DISCONNECT** packet and then close the network connection.

The **AUTH** packet is a new type of packet introduced in MQTT 5.0, and it is used solely for enhanced authentication, providing more secure authentication for clients and servers.

The **PINGREQ** and **PINGRESP** packets are used for connection keep-alive and probing. The client periodically sends a **PINGREQ** packet to the server to indicate that it is still active, then judges whether the server is active according to whether the **PINGRESP** packet is returned in time.

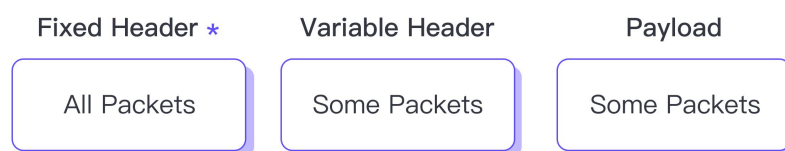
The **PUBLISH** packet is used to publish messages, and the remaining four packets are used to acknowledge QoS 1 and 2 messages.

The **SUBSCRIBE** packet is used by the client to subscribe to topics, while the **UNSUBSCRIBE** packet serves the opposite purpose. The **SUBACK** and **UNSUBACK** packets are used to return the results of subscription and unsubscription, respectively.

MQTT Packet Format

In MQTT, regardless of the type of control packet, they all consist of three parts: Fixed Header, Variable Header, and Payload.

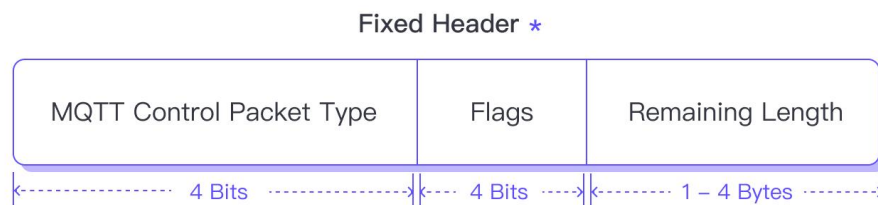
The Fixed Header always exists in all control packets. The existence and content of the Variable Header and Payload depend on the specific packet type. For example, the **PINGREQ** packet used for keeping alive only includes the Fixed Header, while the **PUBLISH** packet used for transmitting application messages includes all three parts.



Fixed Header

The Fixed Header consists of three fields: MQTT Control Packet Type, Flags, and

Remaining Length.



The MQTT Control Packet Type is located in the high 4 bits of the first byte of the Fixed Header. It is an unsigned integer that represents the type of the current packet. For example, 1 indicates a **CONNECT** packet, 2 indicates a **CONNACK** packet, and so on. The detailed mapping can be found in the [MQTT 5.0 specification – MQTT Control Packet Types](#). In fact, except for the MQTT Control Packet Type and Remaining Length fields, the content of the remaining part of the MQTT packet depends on the specific packet type. So, this field determines how the receiver should parse the following content of the packet.

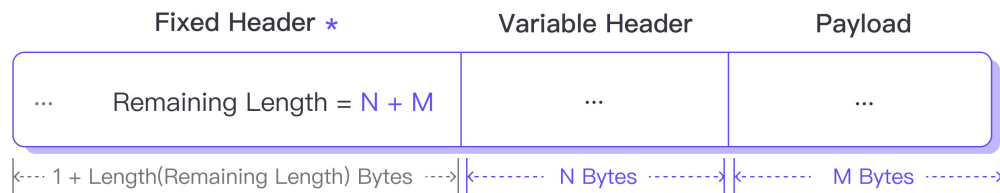
The remaining low 4 bits in the first byte of the Fixed Header contain flags determined by the control packet type. However, as of MQTT 5.0, only the four bits in the **PUBLISH** packet have been assigned specific meanings:

- Bit 3: DUP, indicates whether the current **PUBLISH** packet is a retransmitted packet.
- Bit 2,1: QoS, indicates the quality of service level used by the current **PUBLISH** packet.
- Bit 0: Retain, indicates whether the current **PUBLISH** packet is retained.

In all other packet types, these 4 bits remain reserved, meaning they have a fixed value that cannot be arbitrarily changed.

The final Remaining Length field indicates the number of bytes in the remaining part of the control packet, which includes the Variable Header and the Payload. Therefore, an

MQTT control packet's total length is equal to the Fixed Header's length plus the Remaining Length.



Variable Byte Integer

However, the length of the Fixed Header is not fixed. In order to minimize the packet size as much as possible, MQTT uses the Remaining Length field as a variable byte integer.

In MQTT, there are many fields of variable length. For example, the Payload part in the **PUBLISH** packet is used to carry the actual application message, and the length of the application message is clearly not fixed. So, we need an additional field to indicate the length of these variable-length contents so that the receiving end can parse them correctly.

For a 2 MB application message, which is a total of 2,097,152 bytes, we would need a 4-byte integer to indicate its length. However, not all application messages are that large; in many cases, they are only a few KB or even just a few bytes. Using a 4-byte integer to indicate a message length of only 2 bytes would be excessive.

Therefore, MQTT introduces variable byte integers, which utilize the lower 7 bits of each byte to encode data, while the highest bit indicates whether there are more bytes to follow. This way, when the packet length is less than 128 bytes, the variable byte integer only needs one byte to indicate. The maximum length of a variable byte integer is 4 bytes, allowing it to indicate a length of up to $(2^{28} - 1)$ bytes, which is 256 MB of data.

Digits	From	To
1	0 (0x00)	127 (0x7F)
2	128 (0x80, 0x01)	16,383 (0xFF, 0x7F)
3	16,384 (0x80, 0x80, 0x01)	2,097,151 (0xFF, 0xFF, 0x7F)
4	2,097,152 (0x80, 0x80, 0x80, 0x01)	268,435,455 (0xFF, 0xFF, 0xFF, 0x7F)

Variable Header

The contents of the Variable Header in MQTT depend on the specific packet type. For example, the Variable Header of the **CONNECT** packet includes the Protocol Name, Protocol Level, Connect Flags, Keep Alive, and Properties in that order. The Variable Header of a **PUBLISH** packet includes the Topic name, Packet Identifier (if QoS is not 0), and Properties in that order.

Variable Header in **CONNECT**

Protocol Name	Protocol Level	Connect Flags	Keep Alive	Properties
---------------	----------------	---------------	------------	------------

Variable Header in **PUBLISH**

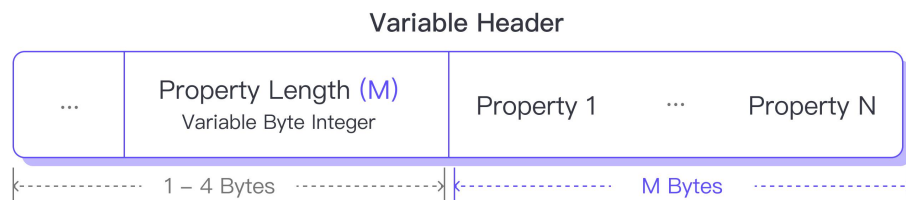
Topic Name	Packet Identifier	Properties
------------	-------------------	------------

The fields in the Variable Header must strictly follow the protocol specification because the receiver will only parse them in the specified order. We cannot omit any field unless the protocol explicitly requires or allows it. For example, in the Variable Header of the **CONNECT** packet, if the Connect Flags are placed directly after the Protocol Name, it

would result in a parsing failure. Similarly, in the Variable Header of the **PUBLISH** packet, the packet identifier is only present when QoS is not 0.

Properties

Properties are a concept introduced in MQTT 5.0. They are basically the last part of the Variable Header. The properties consist of the Property Length field followed by a set of properties. The Property Length indicates the total length of all the properties that follow.



All properties are optional, as they usually have a default value. If there is no property, then the value of the Property Length is 0.

Each property consists of an identifier that defines the purpose and data type of the property and a specific value. Different properties may have different data types. For example, one is a two-byte integer, and another is a UTF-8 encoded string, so we need to parse the properties according to the data type declared by their identifiers.



The order of properties can be arbitrary because we can know which property it is and its length based on the Identifier.

Properties are typically designed for specific purposes. For example, the **CONNECT** packet has a Session Expiry Interval property to set the session's expiration time. However, this property is not needed in a **PUBLISH** packet. Therefore, MQTT strictly defines the usage scope of properties, and a valid MQTT control packet should not contain properties that do not belong to it.

For a comprehensive list of MQTT properties, including their identifiers, property names, data types, and usage scopes, please refer to [MQTT 5.0 Specification – Properties](#).

Payload

Lastly, we have the Payload. The Variable Header of the packet can be seen as its supplementary information, while the Payload is used to achieve the core purpose of the packet.

For example, in the **PUBLISH** packet, the Payload is used to carry the actual application message, which is the primary function of the **PUBLISH** packet. The QoS, Retain, and other fields in the Variable Header of the **PUBLISH** packet provide additional capabilities related to the application message.

The **SUBSCRIBE** packet follows a similar pattern. The Payload contains the topics to subscribe to and their corresponding [subscription options](#), which is the primary task of the **SUBSCRIBE** packet.

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.

Try the world's most scalable MQTT broker to supercharge your MQTT journey

[Try for Free →](#)



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>