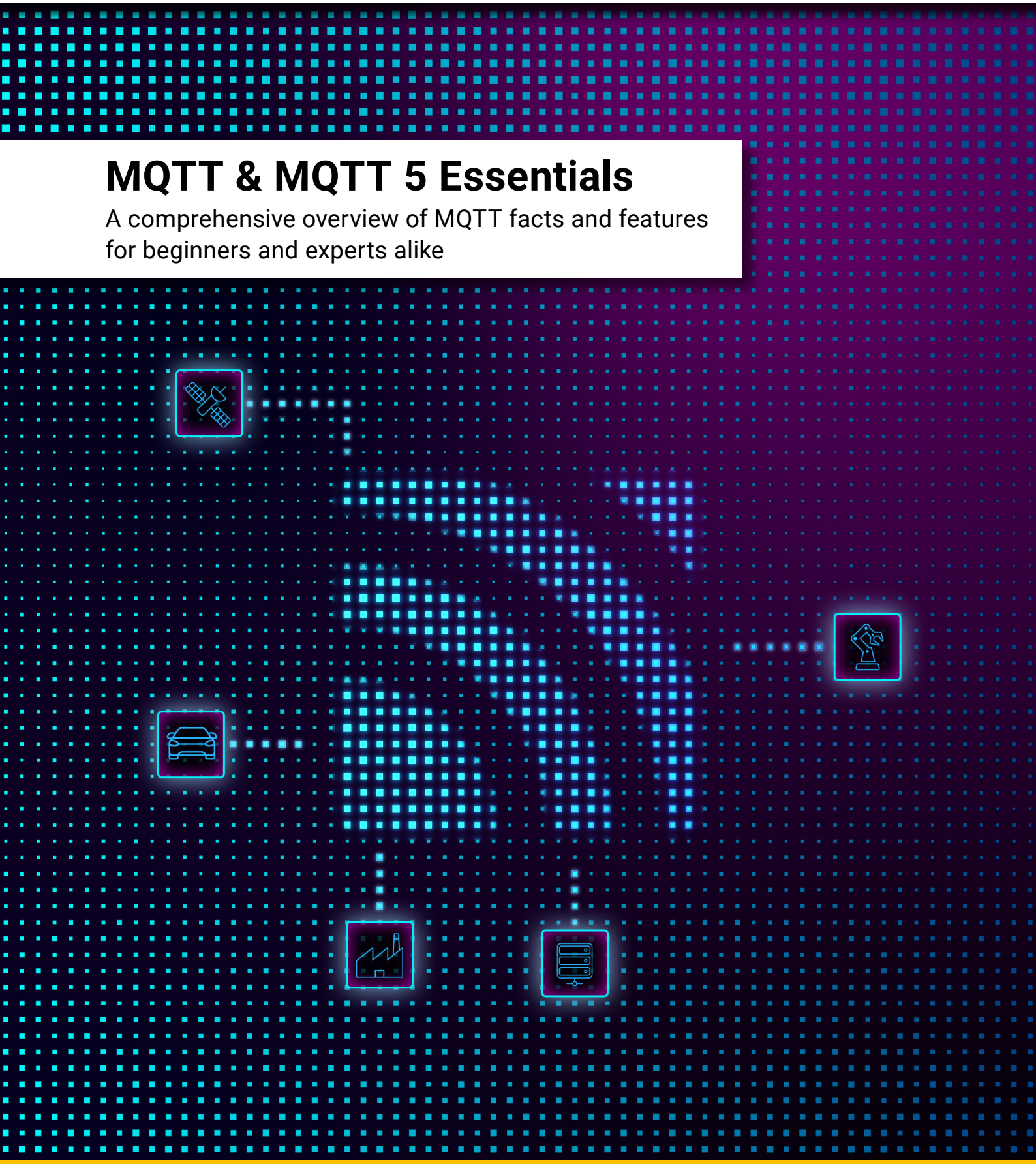
HIVEMQ

MQTT & MQTT 5 Essentials

A comprehensive overview of MQTT facts and features for beginners and experts alike



eBook

hivemq.com

Volume I

MQTT Essentials

The ultimate Kickstart for MQTT Beginners

Table of Contents

Volume II

MQTT 5 Essentials

A technical deep dive into new MQTT 5 features

Table of Contents

Volume I

MQTT Essentials

THE ULTIMATE KICKSTART FOR MQTT BEGINNERS

Chapter 1 - Introducing MQTT

The abstract of the MQTT specification does a good job describing what MQTT is all about. It is a very lightweight and binary protocol, and due to its minimal packet overhead, MQTT e**xcels when transferring data over the wire in comparison to protocols like HTTP**. The protocol is also extremely easy to implement on the client-side. Ease of use was a key concern in the development of MQTT and makes it a perfect fit for constrained devices with limited resources today.

History Lesson

The MQTT protocol was invented in 1999 by Andy Stanford-Clark (IBM) and Arlen Nipper (Arcom, now Cirrus Link). They needed a protocol for minimal battery loss and minimal bandwidth to connect with oil pipelines via satellite. The two inventors specified several requirements for the future protocol:

- Simple implementation
- Quality of Service data delivery
- Lightweight and bandwidth efficient
- Data agnostic
- Continuous session awareness

> *"MQTT is a Client Server publish/subscribe messaging transport protocol. It is lightweight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium."*
>
> **Citation from the official MQTT 3.1.1 specification**

These goals are still at the core of MQTT. However, the primary **focus of the protocol has changed from proprietary embedded systems to open Internet of Things (IoT) use cases**. This shift in focus has created a lot of confusion about what the acronym MQTT stands for. The short answer is that **MQTT is no longer considered an acronym**. MQTT is simply the name of the protocol.

The longer answer is that the former acronym stood for *MQ Telemetry Transport*:

"MQ" refers to the MQ Series, a product IBM developed to support MQ telemetry transport. When Andy Stanford-Clark and Arlen Nipper created their protocol in 1999, they named it after the IBM product. Many sources label MQTT incorrectly as a message queue protocol. That is simply not true.

MQTT is not a traditional message queuing solution (although it is possible to queue messages in certain cases, a fact that we discuss in detail in an upcoming post). Over the next ten years, IBM used the protocol internally until they released MQTT 3.1 as a royalty-free version in 2010. Since then, everyone is welcome to implement and use the protocol.

We became acquainted with MQTT in 2012 and built the first version of HiveMQ that very same year. In 2013, we released HiveMQ to the public. Along with the release of the protocol specification, IBM contributed MQTT client implementations to the newly founded Paho project of the Eclipse Foundation. These events were definitely a big thing for the protocol because there is little chance for wide adoption without a supportive ecosystem.

OASIS Standard & current version

Approximately 3 years after the initial publication, it was announced that MQTT would be standardized under thewings of OASIS, an open organization with the purpose of advancing standards. AMQP, SAML, and DocBook are just a few of the previously released OASIS standards. The standardization process took around 1 year. On October 29, 2014, **MQTT became an officially approved OASIS Standard**.

In March 2019, OASIS ratified the new MQTT 5 specification. This new MQTT version introduced new features to MQTT that are required for IoT applications deployed on cloud platforms, and those that require more reliability and error handling to implement mission-critical messaging.

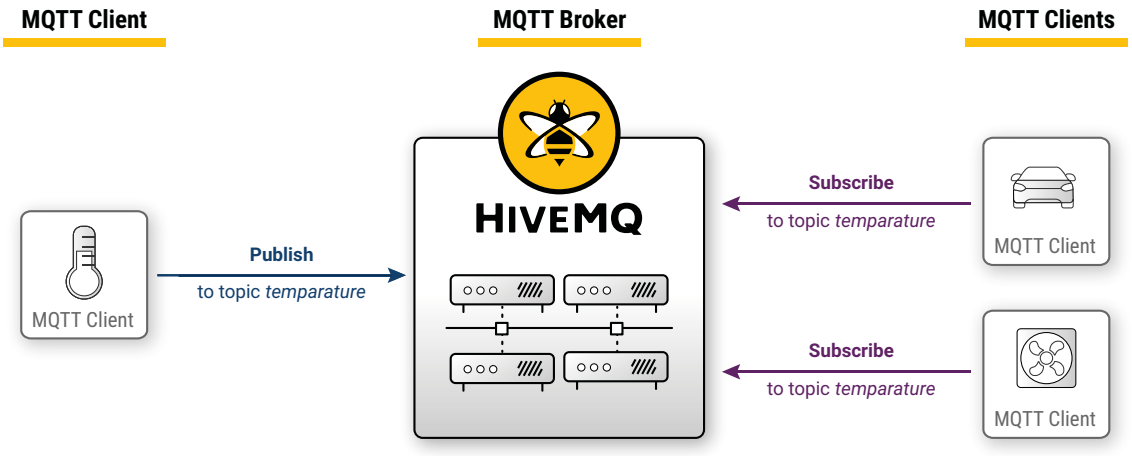
Chapter 2 - The Publish Subscribe Pattern

The publish/subscribe pattern (also known as pub/sub) provides an alternative to a traditional client-server architecture. In the client-server model, a client communicates directly with an endpoint. The pub/sub model **decouples the client that sends a message (the publisher) from the client or clients that receive the messages (the subscribers)**. The publishers and subscribers never contact each other directly. In fact, they are not even aware that the other exists. **The connection between them is handled by a third component (the broker)**. The job of the broker is to filter all incoming messages and distribute them correctly to subscribers.

What is Publish / Subscribe Architecture?

The pub/sub model removes direct communication between the publisher of the message and the recipient/subscriber. The filtering activity of the broker makes it possible to control which client/subscriber receives which message. The decoupling has three dimensions: space, time, and synchronization:

- **Space decoupling**: Publisher and subscriber do not need to know each other (for example, no exchange of IP address and port).
- **Time decoupling**: Publisher and subscriber do not need to run at the same time.
- **Synchronization decoupling**: Operations on both components do not need to be interrupted during publishing or receiving.



Publish/Subscribe Architecture

Scalability

Pub/Sub scales better than the traditional client-server approach. This is because operations on the broker can be highly parallelized and messages can be processed in an event-driven way. Message caching and intelligent routing of messages are often decisive factors for improving scalability. Nonetheless, scaling up to millions of connections is a challenge. Such a high level of connections can be achieved with clustered broker nodes to distribute the load over more individual servers using load balancers

Message Filtering

The broker plays a pivotal role in the pub/sub process. The broker has several filtering options that manage to filter all the messages so that each subscriber receives only messages of interest:

Option 1: Subject-based filtering
This filtering is based on the subject or topic that is part of each message. The receiving client subscribes to the broker for topics of interest. From that point on, the broker ensures that the receiving client gets all messages published to the subscribed topics. In general, topics are strings with a hierarchical structure that allow filtering based on a limited number of expressions.

Option 2: Content-based filtering
In content-based filtering, the broker filters the message based on a specific content filter-language. The receiving clients subscribe to filter queries of messages for which they are interested. A significant downside to this method is that the content of the message must be known beforehand and cannot be encrypted or easily changed.

Option 3: Type-based filtering
When object-oriented languages are used, filtering based on the type/class of a message (event) is a common practice. For example, a subscriber can listen to all messages which are of type exception or any sub-type.

There are a few things you need to consider before you use the publish/subscribe model. The decoupling of publisher and subscriber, which is the key in pub/sub, presents a few challenges of its own. Be aware of how the published data is structured beforehand: For subject-based filtering, both publisher and subscriber need to know which topics to use. Also, keep in mind the message delivery. The publisher can't assume that somebody is listening to the messages that are sent. In some instances, it is possible that no subscriber reads a particular message.

MQTT

Depending on what you want to achieve, **MQTT embodies all the aspects of pub/sub that we've mentioned**:

- MQTT decouples the publisher and subscriber spatially. To publish or receive messages, publishers and subscribers only need to know the hostname/IP and port of the broker
- MQTT decouples by time. Although most MQTT use cases deliver messages in near-real-time, if desired, the broker can store messages for clients that are not online. (Two conditions must be met to store messages: the client had connected with a persistent session and subscribed to a topic with a Quality of Service greater than 0).
- MQTT works asynchronously. Because most client libraries work asynchronously and are based on callbacks or a similar model, tasks are not blocked while waiting for a message or publishing a message. In certain use cases, synchronization is desirable and possible. To wait for a certain message, some libraries have synchronous APIs. But the flow is usually asynchronous.
- MQTT is especially easy to use on the client-side. Most pub/sub systems have the logic on the broker-side, but MQTT is really the essence of pub/sub when using a client library and that makes it a light-weight protocol for small and constrained devices.

MQTT uses subject-based filtering of messages. Every message contains a topic (subject) that the broker can use to determine whether a subscribing client gets the message or not. If desired, you can also set up content-based filtering with the HiveMQ MQTT broker and our custom plugin system.

To handle the challenges of a pub/sub system, **MQTT has quality of service (QoS) levels**. You can easily specify that a message gets successfully delivered from the client to the broker or from the broker to a client. But there is the chance that nobody subscribes to a particular topic. If this is a problem, it depends on the broker how to handles such cases.

For example, an MQTT broker might have a plugin system, which can identify such cases. You can either have the broker take action or simply log every message into a database for historical analyses. To keep the hierarchical topic tree flexible, it is important to design the topic tree very carefully and leave room for future use cases. If you follow these strategies, **MQTT is perfect for production setup**s.

Distinction: MQTT & Message Queues

There is a lot of confusion about the name MQTT and whether the protocol is implemented as a message queue or not. MQTT refers to the MQ Series product from IBM and has nothing to do with “message queue“. Regardless of where the name comes from, it’s useful to understand the differences between MQTT and a traditional message queue:

- **In a message queue, each incoming message is stored in the queue until it is picked up by a client** (often called a consumer). If no client picks up the message, the message remains stuck in the queue and waits to be consumed. In a message queue, it is not possible for a message not to be processed by any client, as it is in MQTT if nobody subscribes to a topic.
- **In a traditional message queue, a message can be processed by one consumer only**. The load is distributed between all consumers for a queue. In MQTT the behavior is quite the opposite: every subscriber that subscribes to the topic gets the message.
- **Queues are named and must be created explicitly**. A queue is far more rigid than a topic. Before a queue can be used, the queue must be created explicitly with a separate command. Only after the queue is named and created is it possible to publish or consume messages. In contrast, MQTT topics are extremely flexible and can be created on the fly.

Chapter 3 - Client, Broker and Connection Establishment

Let's now dive deeper into the roles of the MQTT client and broker, and the parameters and options that are available when you connect to a broker:

Client

MQTT clients are both publishers and subscribers. The publisher and subscriber labels refer to whether the client is currently publishing messages or subscribing to messages (publish and subscribe functionality can also be implemented in the same MQTT client). **An MQTT client is any device (from a micro controller up to a full-fledged server) that runs an MQTT library and connects to an MQTT broker over a network**.

For example, the MQTT client can be a very small, resource-constrained device that connects over a wireless network and has a bare-minimum library. The MQTT client can also be a typical computer running a graphical MQTT client for testing purposes. Basically, any device that speaks MQTT over a TCP/IP stack can be called an MQTT client. The client implementation of the MQTT protocol is very straightforward and streamlined. The ease of implementation is one of the reasons why MQTT is ideally suited for small devices. **MQTT client libraries are available for a huge variety of programming languages. For example, Android, Arduino, C, C++, C#, Go, iOS, Java, JavaScript, and .NET**.

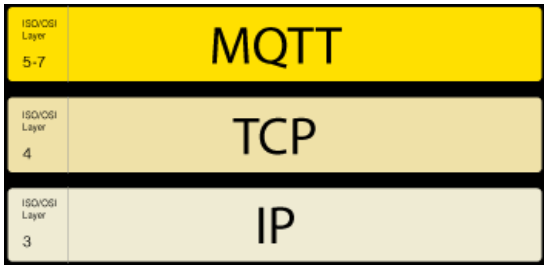
Broker

The counterpart of the MQTT client is the MQTT broker. The broker is at the heart of any publish/subscribe protocol. Depending on the implementation, a broker can handle up to thousands of concurrently connected MQTT clients.

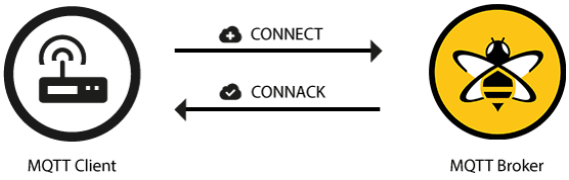
The broker is responsible for receiving all messages, filtering the messages, determining who is subscribed to each message, and sending the message to these subscribed clients. It also holds the sessions of all persisted clients, including subscriptions and missed messages. Another responsibility of the broker is to authenticate and authorize the clients. Usually, the broker is extensible, which facilitates custom authentication, authorization, and integration into backend systems. Integration is particularly important because the broker is frequently the component that is directly exposed on the internet, handles a lot of clients, and needs to pass messages to downstream analyzing and processing systems. In brief, the broker is the central hub through which every message must pass. Therefore, **it is important that your broker is highly scalable, integratable into backend systems, easy to monitor, and (of course) failure-resistant**. HiveMQ meets these requirements by using state-of-the-art event-driven network processing, an open plugin system, and standard monitoring providers.

MQTT Connection

The MQTT protocol is based on TCP/IP. Both the client and the broker need to have a TCP/IP stack.



The MQTT connection is always between one client and the broker. Clients never connect to each other directly.



To initiate a connection, **the client sends a CONNECT message to the broker. The broker responds with a CONNACK (Acknowledgement) message** and a status code. Once the connection is established, the broker keeps it open until the client sends a disconnect command or the connection breaks.

MQTT connection through a NAT

In many common use cases, the MQTT client is located behind a router that uses network address translation (NAT) to translate from a private network address (like 192.168.x.x, 10.0.x.x) to a public facing address. The MQTT client initiates the connection by sending a CONNECT message to the broker. As the broker has a public address and keeps the connection open to allow bidirectional sending and receiving of messages (after the initial CONNECT), there is no problem at all with clients that are located behind a NAT.

CONNECT Message - Client Initiates Connection

To initiate a connection, the client sends a command message to the broker. If this CONNECT message is malformed (according to the MQTT specification) or too much time passes between opening a network socket and sending the connect message, the broker closes the connection. This behavior deters malicious clients that can slow the broker down. **A good-natured client sends a connect message with the following content** → (among other things)

MQTT-Packet:

CONNECT

contains:	Example
clientId	"client-1"
cleanSession	true
username (optional)	"hans"
password (optional)	"letmein"
lastWillTopic (optional)	"/hans/will"
lastWillQos (optional)	2
lastWillMessage (optional)	"unexpected exit"
lastWillRetain (optional)	false
keepAlive	60

Some information included in a CONNECT message is probably more interesting to implementers of an MQTT library rather than to users of that library. For all the details, have a look at the MQTT 3.1.1 specification.

We will focus on the following options:

ClientId: The client identifier (ClientId) **identifies each MQTT client** that connects to an MQTT broker. The broker uses the ClientID to identify the client and the current state of the client.Therefore, this ID should be unique per client and broker. In MQTT 3.1.1 (the current standard), you can send an empty ClientId, if you don't need a state to be held by the broker. The empty ClientID results in a connection without any state. In this case, the clean session flag must be set to true or the broker will reject the connection.

Clean Session: The clean session flag tells the broker whether the client wants to establish a persistent session or not. In a persistent session (CleanSession = false), the broker stores all subscriptions for the client and all missed messages for the client that subscribed with a Quality of Service (QoS) level 1 or 2. If the session is not persistent (CleanSession = true), the broker does not store anything for the client and purges all information from any previous persistent session.

Username/Password: MQTT can send a **user name and password for client authentication and authorization**. However, if this information isn't encrypted or hashed (either by implementation or TLS), the password is sent in plain text. We highly recommend the use of user names and passwords together with a secure transport. Brokers like HiveMQ can authenticate clients with an SSL certificate, so no username and password is needed.

Will Message: The last will message is part of the Last Will and Testament (LWT) feature of MQTT. **This message notifies other clients when a client disconnects ungracefully.** When a client connects, it can provide the broker with a last will in the form of an MQTT message and topic within the CONNECT message. If the client disconnects ungracefully, the broker sends the LWT message on behalf of the client.

KeepAlive: The keep alive is a **time interval in seconds** that the client specifies and communicates to the broker when the connection established. This interval defines the longest period of time that the broker and client can endure without sending a message.The client commits to sending regular PING Request messages to the broker. The broker responds with a PING response. This method allows both sides to determine if the other one is still available.

Basically, that is all the information that is all you need to connect to an MQTT broker from an MQTT client. Individual libraries often have additional options that you can configure. For example, the way that queued messages are stored in a specific implementation.

CONNACK Message - Broker Response

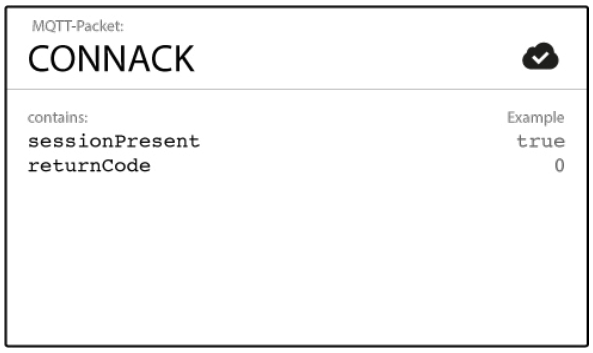
When a broker receives a CONNECT message, it is obligated to respond with a CONNACK message.

The CONNACK message contains two data entries:

- The session present flag
- A connect acknowledge flag

Session Present Flag
The session present flag tells the client whether the broker already has a persistent session available from previous interactions with the client. When a client connects with Clean Session set to true, the session present flag is always false because there is no session available. If a client connects with Clean Session set to false, there are two possibilities: If session information is available for the client Id and the broker has stored session information, the session present flag is true. Otherwise, if the broker does not have any session information for the client ID, the session present flag is false. This is to help clients determine whether they need to subscribe to topics or if the topics are still stored in a persistent session.

Connect Acknowledge Flag:
The second flag in the CONNACK message is the connect acknowledge flag. This flag contains a return code that tells the client whether the connection attempt was successful



Here are the return codes at a glance:

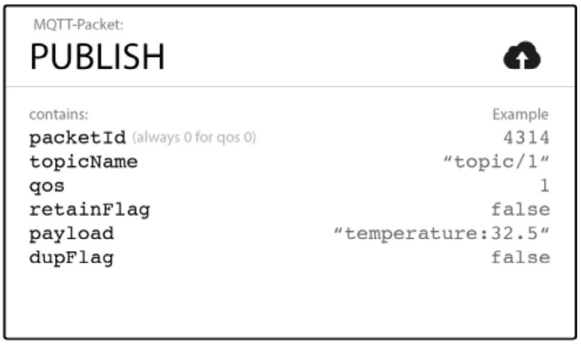
Return Code	Return Code Response
0	Connection accepted
1	Connection refused, unacceptable protocol version
2	Connection refused, identifier rejected
3	Connection refused, server unavailable
4	Connection refused, bad user name or password
5	Connection refused, not authorized

Chapter 4 - MQTT Publish, Subscribe & Unsubscribe

Publish

An MQTT client can publish messages as soon as it connects to a broker. MQTT utilizes topic-based filtering of the messages on the broker. **Each message must contain a topic that the broker can use to forward the message to interested clients. Typically, each message has a payload which contains the data to transmit in byte format**. MQTT is data-agnostic. The use case of the client determines how the payload is structured. The sending client (publisher) decides whether it wants to send binary data, text data, or even full-fledged XML or JSON.

A PUBLISH message in MQTT has several attributes that we want to discuss in detail:



Topic Name: The topic name is a simple string that is hierarchically structured with forward slashes as delimiters. For example:

myhome/livingroom/temperature
or
Germany/Munich/Octoberfest/people

QoS: This number indicates the Quality of Service Level (QoS) of the message. There are three levels: 0,1, and 2. The service level determines what kind of guarantee a message has for reaching the intended recipient (client or broker).

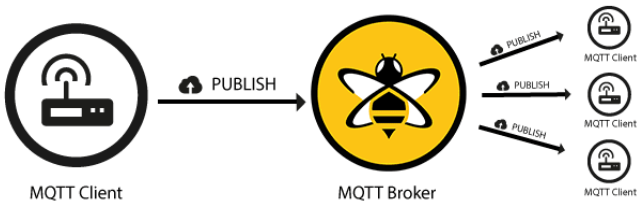
Retain Flag: This flag defines whether the message is saved by the broker as the last known good value for a specified topic. When a new client subscribes to a topic, they receive the last message that is retained on that topic.

Payload: This is the actual content of the message. MQTT is data-agnostic. It is possible to send images, text in any encoding, encrypted data, and virtually every data in binary.

Packet Identifier: The packet identifier uniquely identifies a message as it flows between the client and broker. The packet identifier is only relevant for QoS levels greater than zero. The client library and/or the broker is responsible for setting this internal MQTT identifier.

DUP Flag: The flag indicates that the message is a duplicate and was resent because the intended recipient (client or broker) did not acknowledge the original message. This is only relevant for QoS greater than 0. Usually, the resend/duplicate mechanism is handled by the MQTT client library or the broker as an implementation detail.

When a client sends a message to an MQTT broker for publication, **the broker reads the message, acknowledges the message (according to the QoS Level), and processes the message**. Processing by the broker includes determining which clients have subscribed to the topic and sending the message to them.



The client that initially publishes the message is only concerned about delivering the PUBLISH message to the broker. Once the broker receives the PUBLISH message, it is the responsibility of the broker to deliver the message to all subscribers. The publishing client does not get any feedback about whether anyone is interested in the published message or how many clients received the message from the broker.

Subscribe

Publishing a message doesn't make sense if no one ever receives it. In other words, if there are no clients to subscribe to the topics of the messages. To receive messages on topics of interest, the client sends a SUBSCRIBE message to the MQTT broker. This subscribe message is very simple, it contains a unique packet identifier and a list of subscriptions.

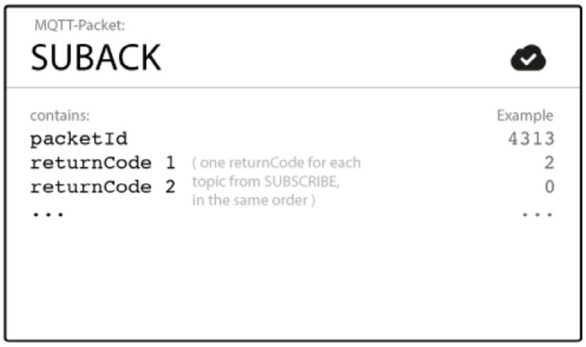
MQTT Subscribe Attributes

Packet Identifier: The packet identifier uniquely identifies a message as it flows between the client and broker. The client library and/or the broker is responsible for setting this internal MQTT identifier.

List of Subscriptions: A SUBSCRIBE message can contain multiple subscriptions for a client. Each subscription is made up of a topic and a QoS level. The topic in the subscribe message can contain wildcards that make it possible to subscribe to a topic pattern rather than a specific topic. If there are overlapping subscriptions for one client, the broker delivers the message that has the highest QoS level for that topic.

Suback

To confirm each subscription, the broker sends a SUBACK acknowledgement message to the client. This message contains the packet identifier of the original Subscribe message (to clearly identify the message) and a list of return codes.

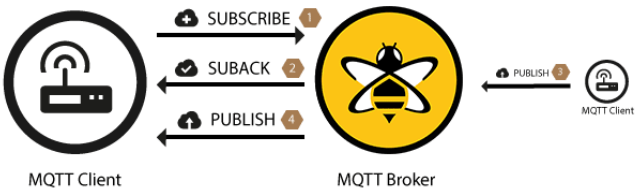


Packet Identifier: The packet identifier is a unique identifier used to identify a message. It is the same as in the SUBSCRIBE message.

Return Code: The broker sends one return code for each topic/QoS-pair that it receives in the SUBSCRIBE message. For example, if the SUBSCRIBE message has five subscriptions, the SUBACK message contains five return codes. The return code acknowledges each topic and shows the QoS level that is granted by the broker. If the broker refuses a subscription, the SUBACK message conains a failure return code for that specific topic. For example, if the client has insufficient permission to subscribe to the topic or the topic is malformed.

Return Code	Return Code Response
0	Success - Maximum QoS 0
1	Success - Maximum QoS 1
2	Success - Maximum QoS 2
128	Failure

After a client successfully sends the SUBSCRIBE message and receives the SUBACK message, it gets every published message that matches a topic in the subscriptions that the SUBSCRIBE message contained.



Unsubscribe

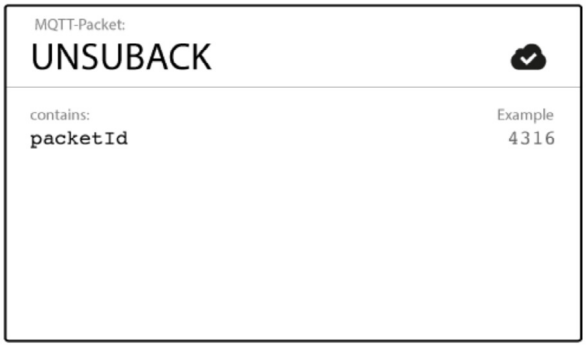
The counterpart of the SUBSCRIBE message is the UNSUBSCRIBE message. This message deletes existing subscriptions of a client on the broker. The UNSUBSCRIBE message is similar to the SUBSCRIBE message and has a packet identifier and a list of topics.

Packet Identifier: The packet identifier uniquely identifies a message as it flows between the client and broker. The client library and/or the broker is responsible for setting this internal MQTT identifier.

List of Topic: The list of topics can contain multiple topics from which the client wants to unsubscribe. It is only necessary to send the topic (without QoS). The broker unsubscribes the topic, regardless of the QoS level with which it was originally subscribed.

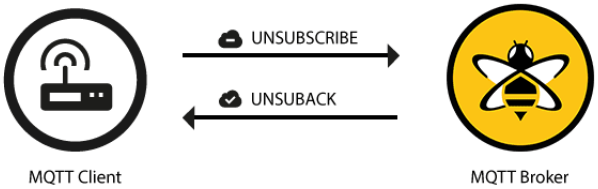
Unsuback

To confirm the unsubscribe, the broker sends an UNSUBACK acknowledgement message to the client. This message contains only the packet identifier of the original UNSUBSCRIBE message (to clearly identify the message).



Packet Identifier The packet identifier uniquely identifies the message. As already mentioned, this is the same packet identifier that is in the UNSUBSCRIBE message.

After receiving the UNSUBACK from the broker, the client can assume that the subscriptions in the UNSUBSCRIBE message are deleted.



Chapter 5 - Topics & Best Practices

Topics

In MQTT, the word **topic refers to an UTF-8 string that the broker uses to filter messages for each connected client.** The topic consists of one or more topic levels. Each topic level is separated by a forward slash (topic level separator).

In comparison to a message queue, MQTT topics are very lightweight. The client does not need to create the desired topic before they publish or subscribe to it. The broker accepts each valid topic without any prior initialization.

Topic Examples:

myhome/groundfloor/livingroom/temperature

USA/California/San Francisco/Silicon Valley

5ff4a2ce-e485-40f4-826c-b1a5d81be9b6/status

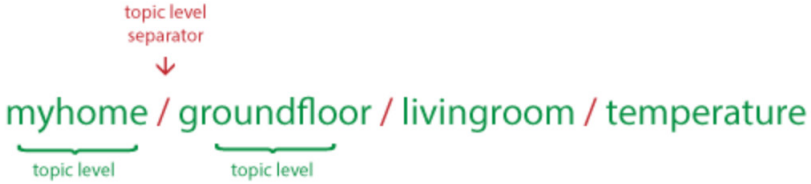
Germany/Bavaria/car/2382340923453/latitude

Note that each topic must contain at least 1 character and that the topic string permits empty spaces. Topics are case-sensitive. For example,
myhome/temperature
and
MyHome/Temperature *are two different topics. Additionally, the forward slash alone is a valid topic.*

Wildcards

When a client subscribes to a topic, it can subscribe to the exact topic of a published message or it can use wildcards to subscribe to multiple topics simultaneously. A wildcard can only be used to subscribe to topics, not to publish a message. There are two different kinds of wildcards: single-level and multi-level.

Single Level: +
As the name suggests, a single-level wildcard replaces one topic level. The plus symbol represents a single-level wildcard in a topic.



Any topic matches a topic with single-level wildcard if it contains an arbitrary string instead of the wildcard. For example a subscription to myhome/groundfloor/+/temperature can produce the following results:

MULTI LEVEL: #
The multi-level wildcard covers many topic levels. The hash symbol represents the multi-level wild card in the topic. For the broker to determine which topics match, the multi-level wildcard must be placed as the last character in the topic and preceded by a forward slash.

When a client subscribes to a topic with a multi-level wildcard, it receives all messages of a topic that begins with the pattern before the wildcard character, no matter how long or deep the topic is. If you specify only the multi-level wildcard as a topic (#), you receive all messages that are sent to the MQTT broker. If you expect high throughput, subscription with a multi-level wildcard alone is an anti-pattern.

Topics Beginning with $

Generally, you can name your MQTT topics as you wish. However, there is one exception: Topics that start with a $ symbol have a different purpose. These topics are not part of the subscription when you subscribe to the multi-level wildcard as a topic (#). The $-symbol topics are reserved for internal statistics of the MQTT broker. Clients cannot publish messages to these topics. At the moment, there is no official standardization for such topics. Commonly, $SYS/ is used for all the following information, but broker implementations varies.

Examples:

$SYS/broker/clients/connected
$SYS/broker/clients/disconnected
$SYS/broker/clients/total
$SYS/broker/messages/sent
$SYS/broker/uptime

Summary

These are the basics of MQTT message topics. As you can see, MQTT topics are dynamic and provide great flexibility. When you use wildcards in real-world applications, there are some challenges you should be aware of. We have collected the best practices that we have learned from working extensively with MQTT in various projects and are always open to suggestions or a discussion about these practices. Use the comments to start a conversation, Let us know your best practices or if you disagree with one of ours!

Best Practices

Never use a leading forward slash
A leading forward slash is permitted in MQTT. For example, /myhome/groundfloor/livingroom. However, the leading forward slash introduces an unnecessary topic level with a zero character at the front. The zero does not provide any benefit and often leads to confusion.

Never use spaces in a topic
A space is the natural enemy of every programmer. When things are not going the way they should, spaces make it much harder to read and debug topics. As with leading forward slashes, just because something is allowed, doesn't mean it should be used. UTF-8 has many different white space types, such uncommon characters should be avoided.

Keep the topic short and concise
Each topic is included in every message in which it is used. Make your topics as short and concise as possible. When it comes to small devices, every byte counts and topic length has a big impact.

Use only ASCII characters, avoid non printable characters
Because non-ASCII UTF-8 characters often display incorrectly, it is very difficult to find typos or issues related to the character set. Unless it is absolutely necessary, we recommend avoiding the use of non-ASCII characters in a topic.

Chapter 6 - Quality of Service Levels

The **Quality of Service** (QoS) level is an agreement between the sender of a message and the receiver of a message that defines the guarantee of delivery for a specific message. There are 3 QoS levels in MQTT:

- At most once (0)
- At least once (1)
- Exactly once (2).

When you talk about QoS in MQTT, you need to consider the two sides of message delivery:

1. Message delivery form the publishing client to the broker.
2. Message delivery from the broker to the subscribing client.

We will look at the two sides of the message delivery separately because there are subtle differences between the two. The client that publishes the message to the broker defines the QoS level of the message when it sends the message to the broker. The broker transmits this message to subscribing clients using the QoS level that each subscribing client defines during the subscription process. If the subscribing client defines a lower QoS than the publishing client, the broker transmits the message with the lower quality of service.

Why is Quality of Service Important?

QoS is a key feature of the MQTT protocol. QoS gives the client the power to choose a level of service that matches its network reliability and application logic. Because MQTT manages the re-transmission of messages and guarantees delivery (even when the underlying transport is not reliable), QoS makes communication in unreliable networks a lot easier.

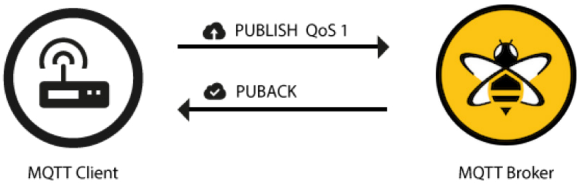
How Does It work?

Let's take a closer look at how each QoS level is implemented in the MQTT protocol and how it functions:

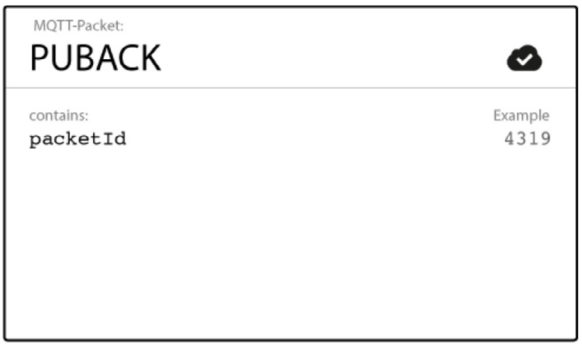
QoS 0 - at most once
The minimal QoS level is zero. This service level guarantees a best-effort delivery. There is no guarantee of delivery. The recipient does not acknowledge receipt of the message and the message is not stored and re-transmitted by the sender. QoS level 0 is often called "fire and forget" and provides the same guarantee as the underlying TCP protocol.

QoS 1 - at least once

QoS level 1 guarantees that a message is delivered at least one time to the receiver. The sender stores the message until it gets a PUBACK packet from the receiver that acknowledges receipt of the message. It is possible for a message to be sent or delivered multiple times.



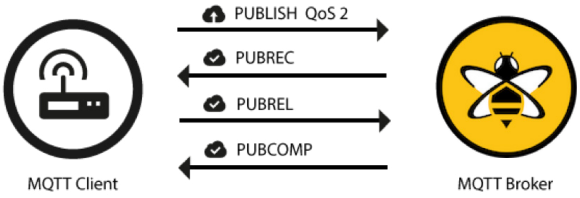
The sender uses the packet identifier in each packet to match the PUBLISH packet to the corresponding PUBACK packet. If the sender does not receive a PUBACK packet in a reasonable amount of time, the sender resends the PUBLISH packet. When a receiver gets a message with QoS 1, it can process it immediately. For example, if the receiver is a broker, the broker sends the message to all subscribing clients and then replies with a PUBACK packet.

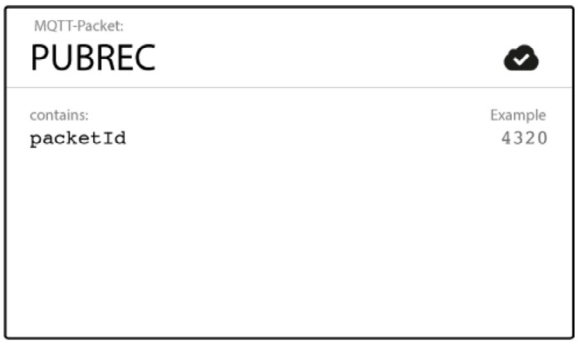


If the publishing client sends the message again it sets a duplicate (DUP) flag. In QoS 1, this DUP flag is only used for internal purposes and is not processed by broker or client. The receiver of the message sends a PUBACK, regardless of the DUP flag.

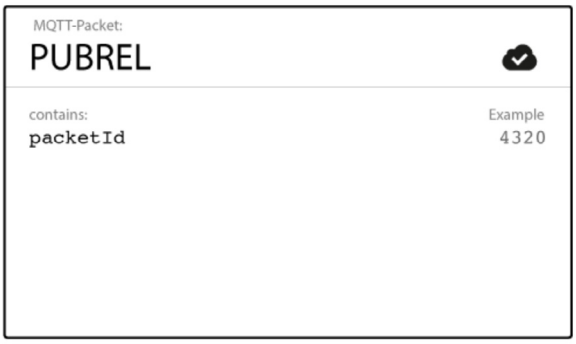
QoS 2 - exactly once

QoS 2 is the highest level of service in MQTT. This level guarantees that each message is received only once by the intended recipients. QoS 2 is the safest and slowest quality of service level. The guarantee is provided by at least two request/response flows (a four-part handshake) between the sender and the receiver. The sender and receiver use the packet identifier of the original PUBLISH message to coordinate delivery of the message.

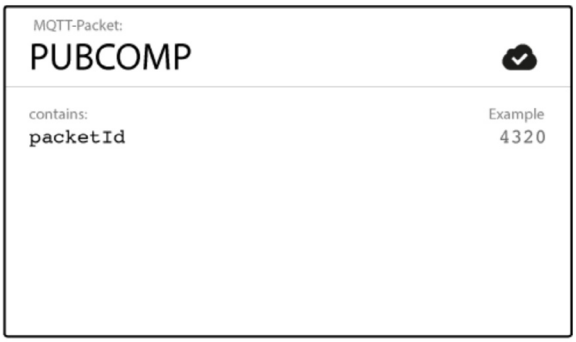




When a receiver gets a QoS 2 PUBLISH packet from a sender, it processes the publish message accordingly and replies to the sender with a PUBREC packet that acknowledges the PUBLISH packet. If the sender does not get a PUBREC packet from the receiver, it sends the PUBLISH packet again with a duplicate (DUP) flag until it receives an acknowledgement.



Once the sender receives a PUBREC packet from the receiver, the sender can safely discard the initial PUBLISH packet. The sender stores the PUBREC packet from the receiver and responds with a PUBREL packet.



After the receiver gets the PUBREL packet, it can discard all stored states and answer with a PUBCOMP packet (the same is true when the sender receives the PUBCOMP). Until the receiver completes processing and sends the PUBCOMP packet back to the sender, the receiver stores a reference to the packet identifier of the original PUBLISH packet. This step is important to avoid processing the message a second time. After the sender receives the PUBCOMP packet, the packet identifier of the published message becomes available for reuse.

When the QoS 2 flow is complete, both parties are sure that the message is delivered and the sender has confirmation of the delivery.

If a packet gets lost along the way, the sender is responsible to retransmit the message within a reasonable amount of time. This is equally true if the sender is an MQTT client or an MQTT broker. The recipient has the responsibility to respond to each command message accordingly.

QoS Good to Know

Some aspects of QoS are not very obvious at first glance. Here are a few things to keep in mind when you use QoS:

Downgrade of QoS
As we already mentioned, the QoS definition and levels between the client that sends (publishes) the message and the client that receives the message are two different things. The QoS levels of these two interactions can also be different. The client that sends the PUBLISH message to the broker defines the QoS of the message. However, when the broker delivers the message to recipients (subscribers), the broker uses the QoS that the receiver (subscriber) defined during subscription. For example, client A is the sender of the message. Client B is the receiver of the message. If client B subscribes to the broker with QoS 1 and client A sends the message to the broker with QoS 2, the broker delivers the message to client B (receiver/subscriber) with QoS 1. The message can be delivered more than once to client B, because QoS 1 guarantees delivery of the message at least one time and does not prevent multiple deliveries of the same message.

Packet identifiers are unique per client
The packet identifier that MQTT uses for QoS 1 and QoS 2 is unique between a specific client and a broker within an interaction. This identifier is not unique between all clients. Once the flow is complete, the packet identifier is available for reuse. This reuse is the reason why the packet identifier does not need to exceed 65535. It is unrealistic that a client can send more than this number of messages without completing an interaction.

Best Practice

We are often asked for advice about how to choose the correct QoS level. Here are some guidelines that can help you in your decision making process. The QoS that is right for you depends heavily on your use case.

USE QOS 0 WHEN …

- You have a completely or mostly stable connection between sender and receiver. A classic use case for QoS 0 is connecting a test client or a front end application to an MQTT broker over a wired connection.
- You don't mind if a few messages are lost occasionally. The loss of some messages can be acceptable if the data is not that important or when data is sent at short intervals.
- You don't need message queuing. Messages are only queued for disconnected clients if they have QoS 1 or 2 and a persistent session.

USE QOS 1 WHEN …

- You need to get every message and your use case can handle duplicates. QoS level 1 is the most frequently used service level because it guarantees the message arrives at least once but allows for multiple deliveries. Of course, your application must tolerate duplicates and be able to process them accordingly.
- You can't bear the overhead of QoS 2. QoS 1 delivers messages much faster than QoS 2.

USE QOS 2 WHEN …

- It is critical to your application to receive all messages exactly once. This is often the case if a duplicate delivery can harm application users or subscribing clients. Be aware of the overhead and that the QoS 2 interaction takes more time to complete.

QUEUING OF QOS 1 AND 2 MESSAGES

All messages sent with QoS 1 and 2 are queued for offline clients until the client is available again. However, this queuing is only possible if the client has a persistent session.

QUEUING OF QOS 1 AND 2 MESSAGES

All messages sent with QoS 1 and 2 are queued for offline clients until the client is available again. However, this queuing is only possible if the client has a persistent session.

Chapter 7 - Persistent Session and Queuing Messages

Although MQTT is not a message queue by definition, it can queue messages for clients.

Persistent Session

To receive messages from an MQTT broker, a client connects to the broker and creates subscriptions to the topics in which it is interested. If the connection between the client and broker is interrupted during a non-persistent session, these topics are lost and the client needs to subscribe again on reconnect. Re-subscribing every time the connection is interrupted is a burden for constrained clients with limited resources. To avoid this problem, the client can request a persistent session when it connects to the broker. Persistent sessions save all information that is relevant for the client on the broker. The clientId that the client provides when it establishes connection to the broker identifies the session.

What's stored in a persistent session?
In a persistent session, the broker stores the following information (even if the client is offline). When the client reconnects the information is available immediately.

- Existence of a session (even if there are no subscriptions).
- All the subscriptions of the client.
- All messages in a Quality of Service (QoS) 1 or 2 flow that the client has not yet confirmed.
- All new QoS 1 or 2 messages that the client missed while offline.
- All QoS 2 messages received from the client that are not yet completely acknowledged.

How do you start or end a persistent session?
When the client connects to the broker, it can request a persistent session. The client uses a cleanSession flag to tell the broker what kind of session it needs:

- When the clean session flag is set to true, the client does not want a persistent session. If the client disconnects for any reason, all information and messages that are queued from a previous persistent session are lost.
- When the clean session flag is set to false, the broker creates a persistent session for the client. All information and messages are preserved until the next time that the client requests a clean session. If the clean session flag is set to false and the broker already has a session available for the client, it uses the existing session and delivers previously queued messages to the client.

How does the client know if a session is already stored?
Since MQTT 3.1.1, the CONNACK message from the broker contains a session present flag. This flag tells the client if a previously established session is still available on the broker. For more information on connection establishment, see part 3 of MQTT Essentials.

Persistent session on the client side
Similar to the broker, each MQTT client must also store a persistent session. When a client requests the server to hold session data, the client is responsible for storing the following information:

- All messages in a QoS 1 or 2 flow, that are not yet confirmed by the broker.
- All QoS 2 messages received from the broker that are not yet completely acknowledged.

Best practices

Here are some guidelines that can help you decide when to use a persistent session or a clean session:

Persistent Session

- The client must get all messages from a certain topic, even if it is offline. You want the broker to queue the messages for the client and deliver them as soon as the client is back online.
- The client has limited resources. You want the broker to store the subscription information of the client and restore the interrupted communication quickly.
- The client needs to resume all QoS 1 and 2 publish messages after a reconnect.

Clean session

- The client needs only to publish messages to topics, the client does not need to subscribe to topics.You don't want the broker to store session information or retry transmission of QoS 1 and 2 messages.
- The client does not need to get messages that it misses offline.

How long does the broker store messages?
People often ask how long the broker stores the session. The easy answer is: The broker stores the session until the clients comes back online and receives the message. However, *what happens if a client does not come back online for a long time?* Usually, the memory limit of the operating system is the primary constraint on message storage. There is no standard answer for this scenario. The right solution depends on your use case. For example, in HiveMQ we provide a possibility to purge queued messages.

Chapter 8 - Retained Messages

In MQTT, the client that publishes a message has no guarantee that a subscribing client actually receives the message. The publishing client can only make sure that the message gets delivered safely to the broker. Basically, the same is true for a subscribing client. The client that connects and subscribes to topics has no guarantee on when the publishing client will publish a message in one of their topics of interest. It can take a few seconds, minutes, or hours for the publisher to send a new message in one of the subscribed topics. Until the next message is published, the subscribing client is totally in the dark about the current status of the topic. This situation is where retained messages come into play.

Retained Messages

A retained message is a normal MQTT message with the retained flag set to true. The broker stores the last retained message and the corresponding QoS for that topic. Each client that subscribes to a topic pattern that matches the topic of the retained message receives the retained message immediately after they subscribe. The broker stores only one retained message per topic.

If the subscribing client includes wildcards in the topic pattern they subscribe to, it receives a retained message even if the topic of the retained message is not an exact match.

> **Here's an example:**
>
> Client A publishes a retained message to `myhome/livingroom/temperature`. Sometime later, client B subscribes to `myhome/#`. Client B receives the `myhome/livingroom/temperature` retained message directly after subscribing to `myhome/#`. Client B (the subscribing client) can see that the message is a retained message because the broker sends retained messages with the retained flag set to true. The client can decide how it wants to process the retained messages.

Retained messages help newly-subscribed clients get a status update immediately after they subscribe to a topic. The retained message eliminates the wait for the publishing clients to send the next update.

In other words, a retained message on a topic is the last known good value. The retained message doesn't have to be the last value, but it must be the last message with the retained flag set to true.

It is important to understand that a retained message has nothing to do with persistent sessions (a subject that we covered last week). Once a retained message is stored by the broker, there's only one way to remove it. Keep reading to find out how.

Send a retained message
From the perspective of a developer, sending a retained message is quite simple and straight-forward. You just set the retained flag of an MQTT publish message to true. Typically, your client library provides an easy way to set this flag.

Delete a retained message
There is also a very simple way to delete the retained message of a topic: send a retained message with a zero-byte payload on the topic where you want to delete the previous retained message. The broker deletes the retained message and new subscribers no longer get a retained message for that topic. Often, it is not even necessary to delete, because each new retained message overwrites the previous one.

Why and when should you use Retained Messages?

A retained message makes sense when you want newly-connected subscribers to receive messages immediately (without waiting until a publishing client sends the next message). This is extremely helpful for status updates of components or devices on individual topics. For example, the status of device1 is on the topic myhome/devices/device1/status. When retained messages are used, new subscribers to the topic get the status (online/offline) of the device immediately after they subscribe. The same is true for clients that send data in intervals, temperature, GPS coordinates, and other data. **Without retained messages, new subscribers are kept in the dark between publish intervals**. Using retained messages helps provide the last good value to a connecting client immediately.

Chapter 9 - Last Will and Testament

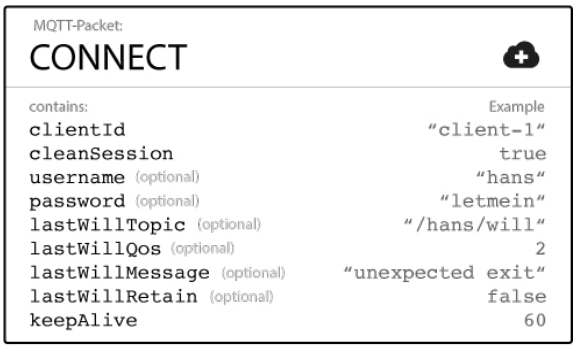
Because MQTT is often used in scenarios that include unreliable networks, it's reasonable to assume that some of the MQTT clients in these scenarios will occasionally disconnect ungracefully. An ungraceful disconnect can occur due to loss of connection, empty batteries, or many other reasons. Knowing whether a client disconnected gracefully (with an MQTT DISCONNECT message) or ungracefully (without a disconnect message), helps you respond correctly. The **Last Will and Testament** feature provides a way for clients to respond to ungraceful disconnects in an appropriate way.

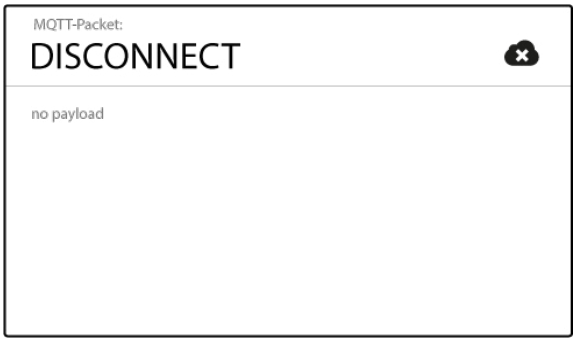
Last Will and Testament

In MQTT, you use the Last Will and Testament (LWT) feature to notify other clients about an ungracefully disconnected client. Each client can specify its last will message when it connects to a broker. The last will message is a normal MQTT message with a topic, retained message flag, QoS, and payload. The broker stores the message until it detects that the client has disconnected ungracefully. In response to the ungraceful disconnect, the broker sends the last-will message to all subscribed clients of the last-will message topic.

If the client disconnects gracefully with a correct DISCONNECT message, the broker discards the stored LWT message.

LWT helps you implement various strategies when the connection of a client drops (or at least inform other clients about the offline status).





How do you specify a LWT message for a client?
Clients can specify an LWT message in the CONNECT message that initiates the connection between the client and the broker.

When does a broker send the LWT message?
According to the MQTT 3.1.1 specification, the broker must distribute the LWT of a client in the following situations:

- The broker detects an I/O error or network failure.
- The client fails to communicate within the defined Keep Alive period.
- The client does not send a DISCONNECT packet before it closes the network connection.
- The broker closes the network connection because of a protocol error.
- We will hear more about the Keep Alive time in the next post.

Best Practices - When Should You Use LWT?

LWT is a great way to notify other subscribed clients about the unexpected loss of connection of another client.

In real-world scenarios, LWT is often combined with retained messages to store the state of a client on a specific topic.

> *For example, client1 first sends a CONNECT message to the broker with a lastWillMessage that has* Offline *as the payload, the lastWillRetain flag set to true, and the lastWillTopic set to* client1/status*. Next, the client sends a PUBLISH message with the payload* Online *and the retained flag set to true to the same topic (*client1/status*). As long as client1 stays connected, newly-subscribed clients to the* client1/status *topic receive the* Online *retained message. If client1 disconnects unexpectedly, the broker publishes the LWT message with the payload* Offline *as the new retained message. Clients that subscribe to the topic while client1 is offline, receive the LWT retained message (*Offline*) from the broker. This pattern of retained messages keeps other clients up to date on the current status of client1 on a specific topic.*

Chapter 10 - Keep Alive and Client Take-Over

The problem of half-open TCP connections

MQTT is based on the Transmission Control Protocol (TCP). This protocol ensures that packets are transferred over the internet in a "reliable, ordered, and error-checked" way. Nevertheless, from time to time, the transfer between communicating parties can get out of sync. For example, if one of the parties crashes or has transmission errors. In TCP, this state of incomplete connection is called a half-open connection. The important point to remember is that one side of the communication continues to function and is not notified about the failure of the other side. The side that is still connected keeps trying to send messages and waits for acknowledgements.

As Andy Stanford-Clark (the inventor of the MQTT protocol) points out, the problem with half-open connections increases in mobile networks:

> *„Although TCP/IP in theory notifies you when a socket breaks, in practice, particularly on things like mobile and satellite links, which often "fake" TCP over the air and put headers back on at each end, it's quite possible for a TCP session to "black hole", i.e. it appears to be open still, but in fact is just dumping anything you write to it onto the floor."*
>
> Andy Stanford-Clark on the topic „Why is the keep-alive needed?"

MQTT Keep Alive

MQTT includes a keep alive function that provides a workaround for the issue of half-open connections (or at least makes it possible to assess if the connection is still open).

Keep alive ensures that the connection between the broker and client is still open and that the broker and the client are aware of being connected. When the client establishes a connection to the broker, the client communicates a time interval in seconds to the broker. This interval defines the maximum length of time that the broker and client may not communicate with each other.

> *„The Keep Alive ... is the maximum time interval that is permitted to elapse between the point at which the Client finishes transmitting one Control Packet and the point it starts sending the next. It is the responsibility of the Client to ensure that the interval between Control Packets being sent does not exceed the Keep Alive value. In the absence of sending any other Control Packets, the Client MUST send a PINGREQ Packet."*
>
> MQTT Specification

As long as messages are exchanged frequently and the keep-alive interval is not exceeded, there is no need to send an extra message to establish whether the connection is still open.

If the client does not send a messages during the keep-alive period, it must send a PINGREQ packet to the broker to confirm that it is available and to make sure that the broker is also still available.

The broker must disconnect a client that does not send a a message or a PINGREQ packet in one and a half times the keep alive interval. Likewise, the client is expected to close the connection if it does not receive a response from the broker in a reasonable amount of time.

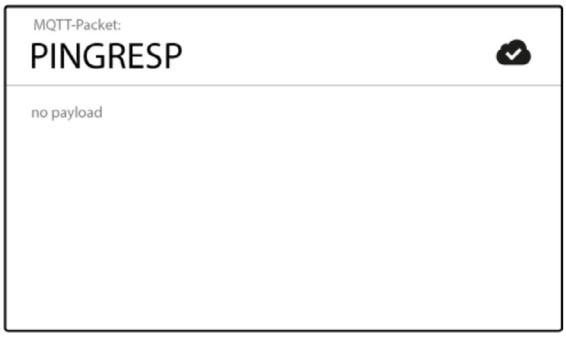
Keep Alive Flow

Let's take a closer look at the keep alive messages. The keep alive feature uses two packets:

PINGREQ

The PINGREQ is sent by the client and indicates to the broker that the client is still alive. If the client does not send any other type of packets (for example, a PUBLISH or SUBSCRIBE packet), the client must send a PINGREQ packet to the broker. The client can send a PINGREQ packet any time it wants to confirm that the network connection is still alive. The PINGREQ packet does not contain a payload.

PINGRESP



When the broker receives a PINGREQ packet, the broker must reply with a PINGRESP packet to show the client that it is still available. The PINGRESP packet also does not contain a payload.

Good to Know

- If the broker does not receive a PINGREQ or any other packet from a client, the broker closes the connection and sends the last will and testament message (if the client specified an LWT).
- It is the responsibility of the MQTT client to set an appropriate keep alive value. For example, the client can adjust the keep-alive interval to its current signal strength.
- The maximum keep alive is 18h 12min 15 sec.
- If the keep alive interval is 0, the keep alive mechanism is deactivated.

Client Take-Over

Usually, a disconnected client tries to reconnect. Sometimes, the broker still has a half-open connection for the client. In MQTT, if the broker detects a half-open connection, it performs a 'client take-over'. The broker closes the previous connection to the same client (determined by the client identifier), and establishes a new connection with the client. This behavior ensures that the half-open connection does not stop the disconnected client from re-establishing a connection.

Chapter 11 - MQTT Over Websockets

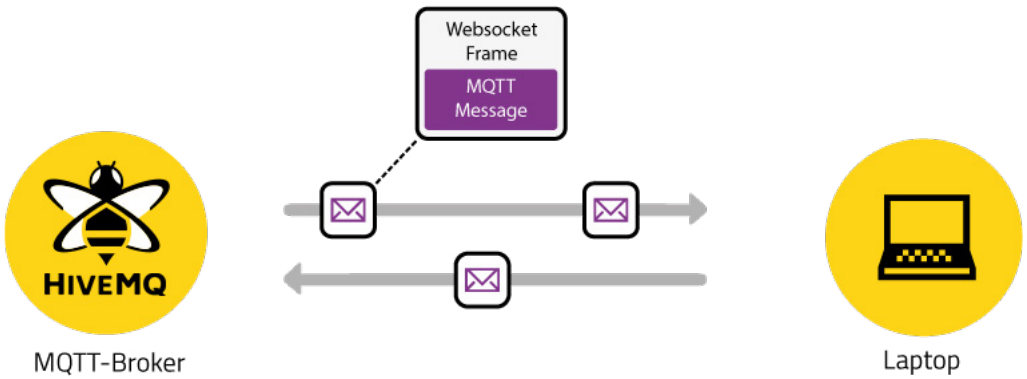
We've seen that MQTT is ideal for constrained devices and unreliable networks and that it is perfect for sending messages with a very low overhead. Naturally, it would be quite nice to send and receive MQTT messages directly in a browser. For example, on a mobile phone. MQTT over WebSockets is the answer. MQTT over WebSockets enables the browser to leverage all MQTT features. You can use these capabilities for many interesting use cases:

- Display live information from a device or sensor.
- Receive push notifications (for example, an alert or critical condition warning).
- See the current status of devices with LWT and retained messages.
- Communicate efficiently with mobile web applications.

What does all this mean from a technical point of view?

Every modern browser that supports WebSockets can be a full-fledged MQTT client and offer all the features described in the MQTT Essentials.The Keep Alive, Last Will and Testament, Quality of Service, and Retained Messages features work the same way in the browser as in a native MQTT client. All you need is a JavaScript library that enables MQTT over WebSockets and a broker that supports MQTT over webSockets. Of course, the HiveMQ broker offers this capability straight out-of-the-box.

How does it work?
WebSocket is a network protocol that provides bi-directional communication between a browser and a web server. The protocol was standardized in 2011 and all modern browsers provide built-in support for it. Similar to MQTT, the WebSocket protocol is based on TCP.



In MQTT over WebSockets, the MQTT message (for example, a CONNECT or PUBLISH packet) is transferred over the network and encapsulated by one or more WebSocket frames. WebSockets are a good transport method for MQTT because they provide bi-directional, ordered, and lossless communication (WebSockets also leverage TCP). To communicate with an MQTT broker over WebSockets, the broker must be able to handle native WebSockets. Occasionally, people use a webserver and bridge WebSockets to the MQTT broker, but we don't recommend this method. When using HiveMQ, it is very easy to get started with WebSockets. Simply enable the native support in the configuration. For more information, read MQTT over WebSockets with HiveMQ.

Why not use MQTT directly?
Currently, it is not possible to speak pure MQTT in a browser because it is not possible to open a raw TCP connection. Socket API will change that situation; however, few browsers implement this API yet.

Get started

If you want to get started with MQTT over WebSockets, here are some useful resources:

- For testing and debugging, the HiveMQ MQTT WebSocket client is ideal. The public broker of the MQTT Dashboard is the default broker of this client. All features of the client are documented in detail and the source code is available on GitHub.
- If you want to integrate MQTT into your existing web application, check out this step-by-step guide on how to build your own MQTT WebSockets client.
- To learn more about how to set up your own broker with WebSockets support, read MQTT over WebSockets.

Secure WebSockets

You can leverage Transport Layer Security (TLS) to use secure WebSockets with encryption of the whole connection. This method works seamlessly with HiveMQ. However, there are a few points that you need to keep in mind. For more information, see the Gotcha section of our user guide.

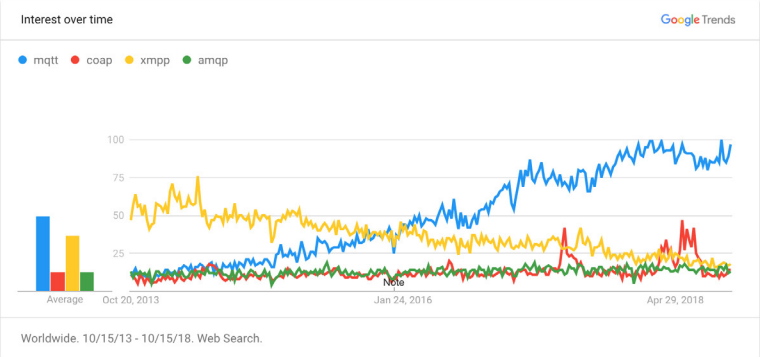
Volume II

MQTT5 Essentials

A technical deep dive into new MQTT 5 features

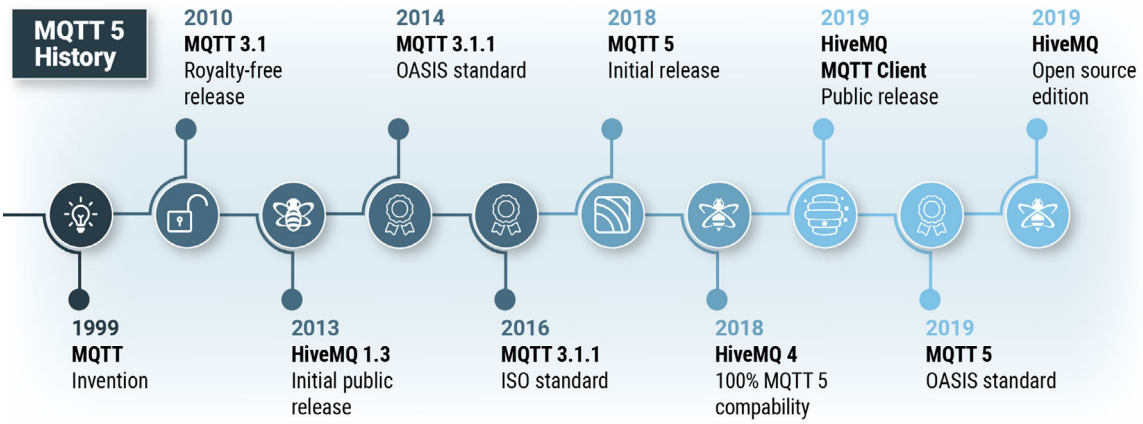
Chapter 1 - Introduction to MQTT 5

The MQTT protocol is the most popular and best received Internet of Things protocol in the world today (take a look at the Google trends chart). Since its introduction, MQTT has successfully connected countless numbers of constrained devices in deployments of all sizes. Popular use cases range from connected cars, manufacturing systems, logistics, and the military to enterprise chat applications, and mobile apps. It's no surprise that such widespread adoption of the protocol has fueled high demand for further advancement of the MQTT specification. MQTT v5 seeks to meet that demand.



MQTT Timeline

Although the MQTT protocol was invented in 1999, its rapid rise began years later. Open source combined with open standard made a winning combination and the MQTT community grew quickly. Five years after MQTT 3.1.1 was released as an OASIS and ISO standard, MQTT 5 followed. In March, 2019, MQTT 5 took its place as the newly approved OASIS and ISO standard.



MQTT 5 Design Goals

The OASIS technical committee (TC) that is responsible for specifying and standardizing MQTT faced a complex balancing act:

- Add features that long term users want without increasing overhead or decreasing ease of use.
- Improve performance and scalability without adding unnecessary complexity.
- The TC decided on the following functional objectives for the MQTT 5 specification:
- Enhancement for scalability and large scale systems
- Improved error reporting
- Formalize common patterns including capability discovery and request response
- Extensibility mechanisms including user properties
- Performance improvements and support for small clients

Based on these objectives and the needs of existing MQTT deployments, the TC managed to specify several extremely useful new features. Sophisticated MQTT brokers like the HiveMQ Enterprise MQTT Broker already implemented features such as Shared Subscriptions and Time to Live for messages and client sessions in MQTT 3.1.1. With the release of MQTT 5, these popular features became part of the official standard.

A key goal of the new specification is enhancement for scalability and large scale systems. MQTT 3.1.1 proved that MQTT is a uniquely scalable and stateful IoT protocol. (For example, the HiveMQ enterprise MQTT broker achieved benchmarking 10.000.000 MQTT simultaneous connections on cloud infrastructure for a single MQTT broker cluster. The design of MQTT 5 aims to make it even easier for an MQTT broker to scale to immense numbers of concurrently-connected clients. In this series, we'll examine how the new version handles a broad spectrum of IoT use cases and large-scale deployments of MQTT.

Trivia: What Happened to Four?

You might be curious why the successor to MQTT 3.1.1 is MQTT 5. The answer is surprisingly simple: The MQTT protocol defines a fixed header in the CONNECT packet. This header contains a single byte value for the protocol version.

If you inspect a few CONNECT packets on the wire, you'll notice something interesting: MQTT 3.1 has the value „3" as protocol version and MQTT 3.1.1 has the value „4". To synchronize the protocol version value on the wire with the official protocol version name, the new MQTT version gets to use „5" for both the protocol name and value.

Chapter 2 - Foundational Changes in the MQTT 5 Protocol

While MQTT 5 is a major update to the existing protocol specification, the new version of the lightweight IoT protocol is more of an evolution rather than a revolution and retained all characteristics that contributed to its success: Its lightweightness, push communication, unique features, ease of use, extreme scalability, suitability for mobile networks and decoupling of communication participants.

Although some foundational mechanics were added or changed slightly, the new version still feels like MQTT and it sticks to its principles that made it the most popular Internet of Things protocol to date. This blog post will analyze everything you need to know about the foundational changes in version 5 of the MQTT specification before digging deep into the details of the new features during the next weeks.

MQTT is still MQTT. Mostly.

The good news: If you're familiar with MQTT 3.1.1 (if you aren't, we recommend reading the MQTT Essentials series first!), then all principles and features you know about MQTT are still valid for MQTT v5. Some details of former features like Last Will and Testament changed a bit or some features were extended and additional popular features implemented by HiveMQ like TTL or Shared Subscriptions were added to the new specification.

The protocol also slightly changed on the wire and an additional control packet (AUTH) was added. But all in all, the MQTT version 5 is still clearly recognizable as MQTT.

Properties in the MQTT Header & Reason Codes

One of the most exciting and most flexible new MQTT 5 features is the possibility to add custom **key-value properties in the MQTT header**. This feature deserves its own blog post as it is a game changer for many deployments. Similar to protocols like HTTP, MQTT clients and brokers can add an arbitrary number of custom (or pre-defined) headers to carry metadata. This metadata can be used for application specific data. Pre-defined headers are used for the implementation of most of the new MQTT features.

Many MQTT packets now also include **Reason Codes**. A Reason Code indicates that a pre-defined protocol error occurred. These reason codes are typically carried on acknowledgement packets and allow client and broker to interpret error conditions (and potentially work around them). The Reason Codes are sometimes called Negative Acknowledgements. The following MQTT packets can carry Reason Codes:

- CONNACK
- PUBACK
- PUBREC
- PUBREL
- PUBCOMP
- SUBACK
- UNSUBACK
- AUTH
- DISCONNECT

Reason Codes for negative acknowledgements range from “Quota Exceeded” to “Protocol Error”. Clients and brokers are responsible for interpreting these new Reason Codes.

CONNACK Return Codes for Unsupported Features

With the popularity of MQTT, a lot of MQTT implementations were created and offered by companies. Not all of these implementations are completely MQTT specification compatible since sometimes features are not implemented like QoS 2, retained messages or persistent sessions. On a side note, HiveMQ is of course fully MQTT specification conformal and supports all features.

MQTT 5 provides a way for incomplete MQTT implementations (as often found in SaaS offerings) to indicate that the broker does not support specific features. It’s the client’s job to make sure that none of the unsupported features are used. The broker implementation uses pre-defined headers in the CONNACK packet (which is sent by the broker after the client sent a CONNECT packet) to indicate that specific features are not supported. These headers can of course also be used to send a notice to the client that it has no permission to use specific features.

The following pre-defined headers for indicating unimplemented features (or features not permitted for use by the client) are available in MQTT v5:

Pre-Defined Header	Data Type	Description
Retain Available	Boolean	Are retained messages available?
Maximum QoS	Number	The maximum QoS the client is allowed to use for publishing messages or subscribing to topics
Wildcard available	Boolean	If Wildcards can be used for topic subscriptions
Subscription identifiers available	Boolean	If Subscription Identifiers are available for the MQTT client
Shared Subscriptions available	Boolean	If Shared Subscriptions are available for the MQTT client
Maximum Message Size	Number	Defines the maximum message size a MQTT client can use
Server Keep Alive	Number	The Keep Alive Interval the server supports for the individual client

These return codes are a major step forward for communicating the permissions of individual MQTT clients in heterogeneous environments. The downside of this new functionality is that MQTT clients need to implement the interpretation of these codes themselves and need to make sure that application programmers don't use features that are not supported by the broker (or the client has no permission for). HiveMQ is going to support all MQTT 5 features 100%, so these custom headers are only expected to be used if the administrator choses to use them for permissions in deployments.

Clean Session is now Clean Start

A popular MQTT 3.1.1 functionality is the use of clean sessions by MQTT clients, which have temporary connections or don't subscribe to messages at all. When connecting to the broker, the client had to choose to send a CONNECT packet with the cleanSession flag enabled or disabled. With a clean session, a MQTT client indicates that the broker should discard any data for the client as soon as the underlying TCP connection breaks or if the client decides to disconnect from the broker. Also, if there was a previous session associated with the client identifier on the broker, a cleanSession CONNECT packet forced the broker to delete the previous data.

With MQTT v5, a client can choose to use a **Clean Start** (indicated by the Clean Start flag in the CONNECT message). When using this flag, the broker discards any previous session data and the client starts with a fresh session. The session won't be cleaned automatically after the TCP connection was closed between client and server. To trigger the deletion of the session after the client disconnected, a new header field called "Session Expiry Interval" must be set to the value 0.

The new Clean Start streamlines and simplifies the session handling of MQTT, as it allows more flexibility and is easier to implement than the cleanSession/persistent session concept. With MQTT 5 all session are persistent unless the "Session Expiry Interval" is 0. Deletion of a session occurs after the timeout or when the client reconnects with Clean Start.

Additional MQTT Packet

MQTT 5 introduces a new MQTT Packet: The AUTH packet. This new packet is extremely useful for implementing non-trivial authentication mechanisms and we expect that this packet will be used a lot in production environments. The exact semantics are covered in a future blog post.

For the moment, it's important to realize that this new packet can be sent by brokers and clients after connection establishment to use complex challenge/response authentication methods (like SCRAM or Kerberos as defined in the SASL framework), but can also be used for state-of-the-art authentication methods for IoT like OAuth. This packet also allows re-authentication of MQTT clients without closing the connection.

New Data Type: UTF-8 String Pairs

The advent of custom headers also required a new data type to be introduced. UTF-8 string pairs. This string pair is essentially a key-value structure with both, key and value, as String data type. This data type is currently only used for custom headers.

With this new data type, MQTT uses 7 different data types that are used on the wire:

- Bit
- Two Byte Integer
- Four Byte Integer
- UTF-8 Encoded String
- Variable Byte Integer
- Binary Data
- UTF-8 String Pair

Most application users typically use Binary Data and UTF-8 encoded Strings in the APIs of their MQTT library. With MQTT 5, UTF-8 String Pairs may also be used frequently. All other data types are hidden from the user but are used on the wire to craft valid MQTT packets by the MQTT client libraries and brokers.

Bi-Directional DISCONNECT Packets

With MQTT 3.1.1, the client could indicate that it wanted to disconnect gracefully by sending a DISCONNECT packet prior to closing the underlying TCP connection. There was no way for the MQTT broker to notify a MQTT client that something bad happened and that the broker is going to close the TCP connection. This changed with the new protocol version. The broker is now allowed to send a MQTT DISCONNECT packet prior to closing the socket. The client is now able to interpret the reason why it was disconnected and take the respective action. A broker is not required to indicate the exact reason (e.g. for security reasons), but at least for developing applications this helps a lot to figure out why a connection was closed by the broker.

Of course DISCONNECT packets can carry Reason Codes, so it's easy to indicate what was the reason for the disconnection (e.g. in case of invalid permissions).

No Retry for QoS 1 and 2 Messages

MQTT clients use standing TCP (or similar protocols with the same guarantees) connections as underlying transport. A healthy TCP connection gives bi-directional connectivity with exactly-once and in-order

guarantees, so all MQTT packets sent by clients or brokers will arrive on the other end. In case the TCP connection breaks, while the message is in-flight, QoS 1 and 2 give message delivery guarantees over multiple TCP connections.

MQTT 3.1.1 allowed the re-delivery of MQTT messages while the TCP connection is healthy. In practice, this is a very bad idea, since overloaded MQTT clients may get overloaded even more. Just imagine a case where a MQTT client receives a message from a MQTT broker and needs 11 seconds to process the message (and would acknowledge the packet after the processing). Now imagine the broker would retransmit the message after a 10 second timeout. There is no advantage to this approach and it just uses precious bandwidth and overloads the MQTT client even more.

With MQTT 5, brokers and clients are not allowed to retransmit MQTT messages for healthy TCP connections. The brokers and clients must re-send unacknowledged packets when the TCP connection was closed, though. So the QoS 1 and 2 guarantees are as important as with MQTT 3.1.1.

In case you rely on retransmission packets in your use case (e.g. because your implementation does not acknowledge packets in certain cases), we suggest reconsidering this decision before upgrading to MQTT v5.

Using Passwords without Usernames

MQTT 3.1.1 required the MQTT client to send a username when using a password in the CONNECT packet. For certain use cases this was very inconvenient in case there was no username. A good example would be the use of OAuth, which uses a JSON web token as the only authentication and authorization information. When using such a token with MQTT 3.1.1, static usernames were used frequently, as the only relevant information was in the password field.

While there are more elegant ways in MQTT 5 to carry tokens (e.g. via the AUTH packet), it is still possible to utilize the password field of the CONNECT packet. Now users can just use the password field and don't need to fill out the username anymore.

Conclusion

Although the MQTT protocol stays basically the same, there are some small changes under the hood, which are enablers for many of the new features in version 5 of the popular IoT protocol. As a user of MQTT libraries, most of these changes are good to know but do not affect how MQTT is used. Developers for MQTT libraries and brokers need to take care of the changes, especially on the changes on the protocol nitty-grittys. There were also some small changes in the specified behavior (e.g. retransmission of messages), so it's good to revisit design decisions of deployments when upgrading to MQTT 5.

Chapter 3 - Why You Should Upgrade to MQTT 5

MQTT v5 is a significant update of the MQTT protocol. In response to the feedback from MQTT users, MQTT 5 adds the features that modern IoT applications need. These new features are specifically suited for applications that are deployed to the cloud, require robustness and reliable error handling to implement mission-critical messaging, and seek easier integration of MQTT messages into their existing computing infrastructure.

Better Error Handling for More Robust Systems

To create a more robust overall system, MQTT 5 adds a number of new features that improve error checking between the client and broker. A new **session and message expiry feature** allows you to set a time limit for each message and session. If a message is not delivered within a predefined time period, the message is deleted. For example, let's say that you send an MQTT message to start a safety-critical machine on your factory floor. If the message does not arrive within a certain time period, you can set the message to be automatically deleted. This ensures that the message is delivered only within the period of time that it is safe to start the machine and is never delivered with a delay due to network latency or outages.

MQTT 5 also introduces the concept of **negative acknowledgements**. Based on predefined restrictions, the broker can send an acknowledgement to reject particular messages. Restrictions can be based on maximum message size, maximum quality of service (QoS), unsupported features, etc. The ability to reject messages that exceed a preset maximum guards against MQTT clients that might start sending erroneous messages. If a client gets into an unstable state or a malicious client attempts a denial of service attack (DoS), the broker can automatically reject these oversized messages.

More Scalability for Cloud Native Computing

MQTT v5 standardizes the concept of **shared subscriptions**. Shared subscriptions allow multiple MQTT client instances to share the same subscription on the broker. This feature makes it possible to load balance MQTT clients that are deployed on a cloud cluster. This is useful when you use MQTT clients to store and forward MQTT messages into back-end enterprise systems such as a database or Enterprise Service Bus (ESB).

Topic aliases are another useful addition to the MQTT 5 specification. For large systems that have complex topic structures, topic strings can get very long. If you have thousands or millions of devices transmitting billions of messages, a very long topic string creates higher demand on the network. To provide more efficiency and better performance on very large systems, topic aliases let you substitute topic strings with an integer.

Greater Flexibility and Easier Integration

MQTT 5 introduces **User Properties** that add a key-value property to the message header of an MQTT message. These properties allow you to add application-specific information to each message that can be used in processing the message. For example, add a meta-tag to the header of an MQTT message that includes the unique identifier of the sending client or in the sending client, add the firmware version of the device platform that the receiver can use for analysis and processing.

To make message processing easier for the receiver, **Payload format indicators** (binary or text), including a MIME style content type, have been added to MQTT v5. These format descriptions are useful for a wide range of use cases. For example, the control system for a toll road might send pictures of license plates that need to be processed by image recognition software while other messages might include location coordinates that require a different style of processing.

The Rise of a Single IoT Standard

The MQTT 5 specification has become the obvious choice for most IoT use cases. The new MQTT 5 features successfully address the limitations of MQTT 3 and open the way for future innovation. Over the next few years, we expect to see a massive growth in MQTT adoption across all industries, including manufacturing, automotive, critical infrastructure, logistics, smart cities, etc. MQTT is on the verge of becoming the standard for all IoT.

HiveMQ and MQTT v5

HiveMQ has been intimately involved in the MQTT 5 standardization process and has already implemented one of the first 100% MQTT 5-compliant brokers and clients. HiveMQ 4 can simultaneously support all MQTT 5, MQTT 3.1, and MQTT 3.1.1 features. HiveMQ customers can deploy a mix of MQTT 3 and MQTT 5 clients to support heterogeneous deployments and migrations to MQTT 5. With HiveMQ 4, you can benefit from all new MQTT features and integrate them into your existing IoT deployments.

Chapter 4 - Session and Message Expiry Intervals

Session Expiry Interval

In the CONNECT packet, a connecting client can set a **session expiry interval** in seconds. This interval defines the period of time that the broker stores the session information of that particular MQTT client. When the session expiry interval is set to 0 or the CONNECT packet does not contain an expiry value, the session information is immediately removed from the broker as soon as the network connection of the client closes. The maximum session expiry interval is `UINT_MAX` (4,294,967,295), which makes the offline session last for slightly more than 136 years after the client disconnects.

Message Expiry Interval

A client can set the **message expiry interval** in seconds for each PUBLISH message individually. This interval defines the period of time that the broker stores the PUBLISH message for any matching subscribers that are not currently connected . When no message expiry interval is set, the broker must store the message for matching subscribers indefinitely. When the *retained=true* option is set on the PUBLISH message, this interval also defines how long a message is retained on a topic.

Why were the expiry intervals introduced?

Beyond understanding how these expiry intervals work, let's take a closer look at why these features were introduced into the MQTT 5 specification and explore some practical uses.

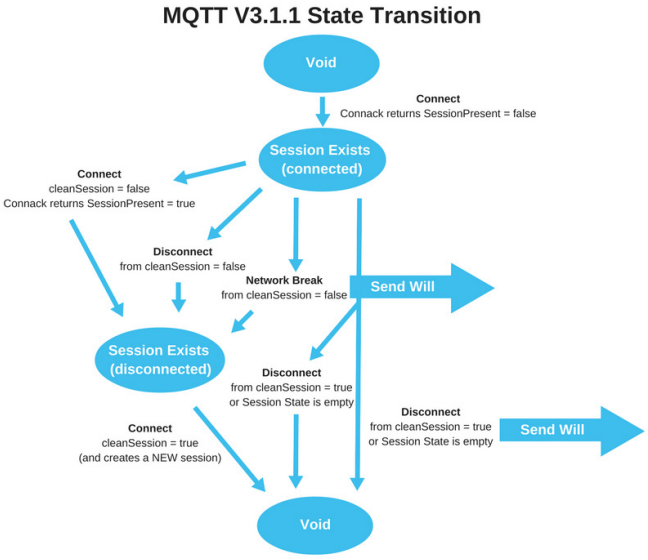
After much deliberation, MQTT v5 became a standard in March 2019. A big part of this deliberation within the OASIS committee (of which HiveMQ is a proud member) was spent gathering feedback, listening to long-term users of the previous MQTT 3.1 and 3.1.1 versions of the protocol, and figuring out how to advance the protocol with features that provide additional possibilities or increased simplicity for the user base.

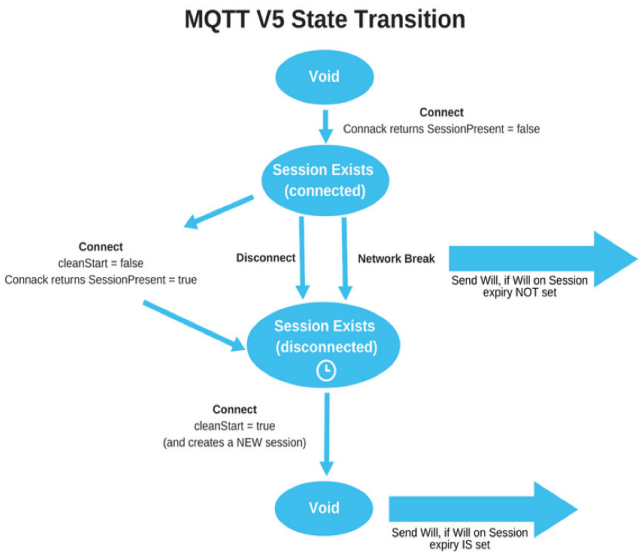
Session Expiry Interval

The session expiry interval feature achieves both of these goals at the same time. Using earlier MQTT versions, the only way to remove the so called persistent sessions provided by the specification was to connect an MQTT client that uses the same client ID as the session you want to remove with a `cleanSession=trueflag`. In scenarios where some of your IoT devices never reconnect, such as destruction or decommissioning of the devices or improperly cleaned up sessions that were left over from load testing environments, the session remnants can put unnecessary strain on a broker's persistence.

Enterprise ready MQTT brokers such as HiveMQ are equipped with additional administrative tools like the HiveMQ Control Center that make the management of unused sessions more convenient. However, you still need to know which session can actually be removed. The new session expiry interval feature lets you set a reasonable amount of time after which an unused session is automatically removed by the broker and frees up resources.

In addition to this automatic housekeeping functionality, the introduction of the session expiry interval has significantly simplified the handling of session states. Here are two diagrams, courtesy of Ian Craggs, that demonstrate the de-cluttering of the state transition.





Message Expiry Interval

Similar to the session expiry interval, the primary motivation to add the message expiry interval to the MQTT protocol standard was to get an automated housekeeping mechanism. Many kinds of IoT devices, such as connected cars, are designed with the distinct possibility of prolonged periods without an internet connection. For these scenarios, MQTT provides persistent sessions and message queueing. Messages that are intended for the offline device are stored on the broker and delivered when the device regains a connection. In large scale deployments with hundreds of thousands or even millions of connected devices, you need to limit the offline message queue of each individual client. Frequently, the period that messages sent to IoT devices remain relevant varies significantly. Staying with the connected car example, information about the current traffic status is only relevant for a limited time. However, over-the-air firmware updates must be executed even if the car is offline for several weeks. Setting an appropriate

message expiry interval for messages that lose relevance within a certain amount of time and leaving continuously-relevant messages without an expiry ensures that broker resources for clients that are offline long term are used properly.

This method also ensures that the clients don't have to deal with large numbers of irrelevant messages when they reconnect. With retained messages, the message expiry interval can be used in the same way to ensure that retained messages are only delivered to new subscribers for a predefined time period.

GOTCHA: When the session for a client expires, all of the messages that are queued for the client expire with the session, regardless of the individual message expiry status.

Summary and Additional Information

- Both the Session Expiry Interval and the Message Expiry Interval can be used to optimize resource management on the MQTT broker.
- As an aside: Expiry features were requested by many MQTT 3 users in the past. So much so that HiveMQ introduced session and message expiry as additional features in HiveMQ 3.3 (before MQTT 5 was released).
- The equivalent for *cleanSession=true* in the CONNECT packet of MQTTv3 in MQTTv5 is sessionExpiry=0 (or absent) and *cleanStart=true*
- The equivalent for cleanSession=false in the CONNECT packet of MQTTv3 in MQTTv5 is using a sessionExpire value that is greater than zero and *cleanStart=false*
- MQTT brokers like HiveMQ allow for a server side max value configuration for these expiry intervals. This is extremely useful in multi vendor projects, where the party operating the broker may not have control over the settings of the MQTT clients.

Chapter 5 - Improved Client Feedback & Negative ACKs

People have been using MQTT in their IoT projects for many years now. As we've mentioned previously, the OASIS committee made sure that feedback from actual protocol users was taken into consideration when they created the new MQTT 5 protocol standard. One of the top complaints from most users was a lack of transparency. The scant supply of return codes and a lack of possibilities to communicate limitations or specific circumstances from broker to client resulted in increased difficulty for debugging, especially in multi-vendor projects.

Whether investigating the reason for client disconnects, researching why messages fail to reach their intended target, or ensuring consistency in MQTT client deployments across multiple teams, MQTT v3 users often need to extend the pure protocol features with technology like the HiveMQ Extension SDK. To make overcoming these challenges easier and standardized, MQTT version 5 purposely introduced the following features.

Let's focus on several MQTT v5 features that fall under the category of providing more transparency and allowing centralized system control by virtue of the broker:

Feedback on Connection Establishment

With MQTT version 5, it is now possible for MQTT brokers to give additional feedback to MQTT clients on connection establishment. A number of different properties can be added to the connection acknowledgement packet that tell the client which features the broker supports or the client is allowed to use. This includes the following MQTT features:

- Retained messages
- Wildcard subscriptions
- Subscription identifiers
- Shared subscriptions
- Topic aliases
- Maximum quality of service level the client can use

In addition to notifying the client about enabled and disabled features, the new properties in the CONNACK packet also allow the server to give the client feedback on the limits that are granted to it by the broker. These limits include:

- Keep alive
- Session expiry interval
- Maximum packet size
- Maximum number of topic aliases the client can send

Beyond supporting all of these limits, HiveMQ allows you to configure a maximum for the limits and to disable MQTT features that are not needed in your use case.

Better Reason Codes

In MQTT version 3.1 and 3.1.1, the number of available reason codes is limited to 5 unsuccessful reason codes. MQTT v5 provides more than 20 unsuccessful reason codes. Additionally, packets that do not have reason codes in MQTT version 3.1 and 3.1.1 now have the possibility to include reason codes with MQTT 5. These packets are UNSUBACK, PUBACK, PUBREC, PUBREL and PUBCOMP.

New Reason Strings

More reason codes certainly improve the feedback to the clients in all cases, but reason codes lack specific context. Supplying context is where reason strings come into play. As the name implies, a reason string is a string that includes all the context the developer or operations needs for quickly identifying precisely why something happened. The specification describes the context as "… a human readable string designed for diagnostics …". While reason strings are very useful for development and diagnosis, they can be turned off in HiveMQ's configuration if the exposure of these details is not desired.

Server-sent Disconnect Packets

In MQTT 3.1 and 3.1.1, when a client violates any limit that the broker sets, the broker simply closes the client's connection without providing any direct information to the client about why the connection closed. MQTT version 5 adds server-sent disconnect packets that allow the server to send a DISCONNECT packet

to the client before it closes the connection. The server-sent disconnect packet contains a reason code and a reason string that give the client all the details about why the connection is closed. This information drastically simplifies the process of identifying why a broker closed the connection to a specific client.

Negative Acknowledgements

Multiple packets and message flows are now able to respond with a negative acknowledgement.

In MQTT version 3, the UNSUBACK packet that was sent to the client does not have a payload. Therefore, a client can never be informed if, or why, an UNSUBSCRIBE is unsuccessful. MQTT v5 changes all this. In MQTT 5, the UNSUBACK contains a reason code that informs the client about the success status of its UNSUBSCRIBE attempt with a number of possible reasons for failure. For example, no subscription existed in the first place or the client is not authorized to unsubscribe.

The acknowledgement packets from the publish flow (PUBACK, PUBREL, PUBREC, PUBCOMP) can now send a negative acknowledgement to the client if the server cannot currently process the message that the client sent. For example, because the client is not authorized to publish to this topic. The packet gives the client all the information needed to make the necessary changes without having to contact operations or support to know what went wrong.

Summary and Additional Information

- MQTT brokers such as HiveMQ let you set a server side max value configuration for the mentioned limits. This is extremely useful in multi-vendor projects in which the operator of the broker may not have control over the settings of the MQTT clients.
- The improved feedback for client significantly simplifies diagnosis for development and operations.
- The added transparency also improves the interoperability between different MQTT clients and brokers.

Chapter 6 - User Properties

In MQTT 5, user properties are basic UTF-8 string key-value pairs that you can append to almost every type of MQTT packet, excluding PINGREQ and PINGRESP. This includes control packets such as PUBREL, or PUBCOMP. As long as you don't exceed the maximum message size, you can use an unlimited number of user properties to add metadata to MQTT messages and pass information between publisher, broker, and subscriber. The feature is very similar to the HTTP header concept.

Why Were User Properties Introduced?

Two major drawbacks MQTT 3 users voiced were the lack of extensibility of the protocol and the difficulty of creating multi-vendor deployments. The User Properties feature in MQTT v5 addresses both of these concerns. With the added possibility to pass virtually any information across an entire MQTT system, the new specification makes sure that users can extend the standard protocol features to meet their specific needs.

Practical Use Case Examples

While the technical details of the User Property feature may seem trivial, the practical impact of having a way to pass metadata across an entire MQTT ecosystem is, in fact, enormous. Let's take a look at three of the more common use cases that I have personally heard many times from users who were yearning to have a feature like User Properties introduced in the MQTT specification.

Saving resources with payload metadata
In scenarios that use MQTT to connect multiple systems that have been implemented by separate teams and vendors, it is not uncommon that the payload structure varies greatly. Some clients may send JSON, XML, or compressed formats such as Protobuf. With the possibility of adding metadata to the messages that can include the specific markup language and version used to encode the payload, the receiving client or in some cases the broker no longer have the need to unpack the payload and try out a number of possible parsers until the right one is found. Instead, each message carries its own parsing information and reduces the computing load on the entire system.

Increased efficiency through application level routing
MQTT is perfectly suited for, and often utilized as, the transporting and routing component in large scale data processing and streaming deployments. Such deployments frequently contain multitudes of different devices, systems, and applications. It is not unusual for multiple systems to receive the same message for different purposes. For example, one system to display live data and another for long term storage of the same data. In this case, user properties can function as an additional application-level timestamp for the message. The broker can then very quickly decide to not pass certain messages to a specific set

of subscribers, based on a predefined validity period. This adds an additional application-level layer for providing message relevance to the Message Expiry Interval.

Transparent traceability in complex systems
IoT deployments are often complex and the existence of separate systems can make it hard to pinpoint where a specific message comes from or why a multi-layer message flow is unsuccessful. In MQTTv3.1.1 there is no way for the subscriber to know who the publisher of a received message was. While it is possible in 1-to-1 scenarios to embed a unique identifier in the topic, this method negates some of the biggest advantages of the publish-subscribe paradigm. The new User Property feature in MQTT 5 allows the publisher to easily add information about itself. For example, a client ID or publishing region. This information is forwarded to any receiver of the message without adding any extra business logic.

Similarly, you can add a specific system identifier to the MQTT messages to log and trace the entire message flow from sender to all subscribers. When properly implemented, these identifiers can even span multiple MQTT message flows. This ability opens up a completely new level of transparency and traceability for business-critical use cases such as premium paid services for end customers.

Summary and Additional Information

- User properties are UTF-8 string key-value pairs that can be appended to any MQTT message
- The possibilities to apply additional logic to an MQTT use case with this feature is almost limitless
- Deployments and projects that span across multiple systems and vendors can use this feature to ensure consistency

Chapter 7- Shared Subscriptions

Shared subscriptions are an MQTT v5 feature that allows MQTT clients to share the same subscription on the broker. In standard MQTT subscriptions, each subscribing client receives a copy of each message that is sent to that topic. In a shared subscription, all clients that share the same subscription in the same subscription group receive messages in an alternating fashion. This mechanism is sometimes called **client load balancing**, since the message load of a single topic is distributed across all subscribers.

MQTT clients can subscribe to a shared subscription with standard MQTT mechanisms. All common MQTT clients such as Eclipse Paho can be used without modifying anything on the client side. However, shared subscriptions use a special topic syntax for subscribing.

Shared subscriptions use the following topic structure:

$share/GROUPID/TOPIC

The shared subscription consists of 3 parts:

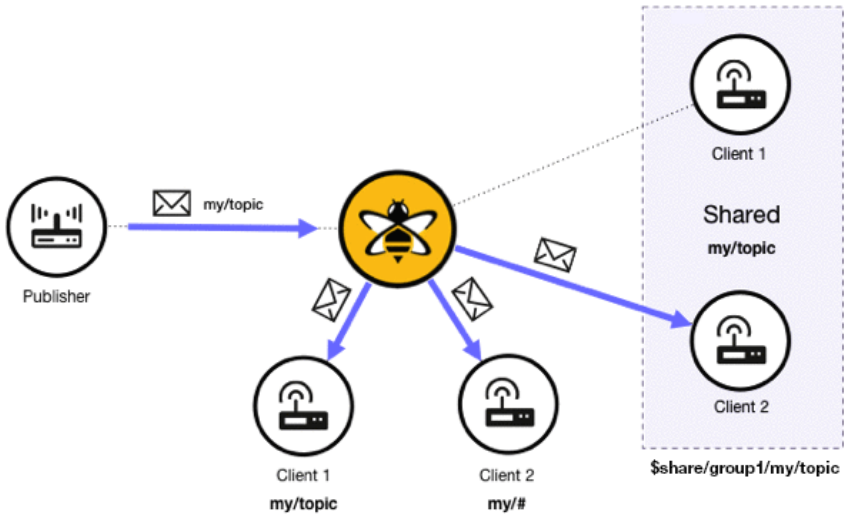
- A static shared subscription identifier (**$share**)
- A group identifier
- The actual topic subscriptions (may include wildcards)

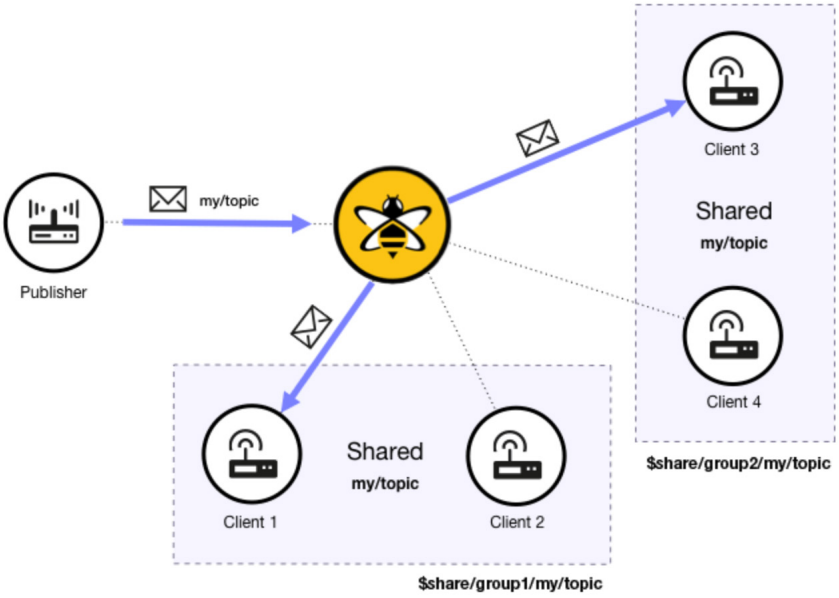
A concrete example for such a subscriber would be:

$share/my-shared-subscriber-group/myhome/groundfloor/+/temperature.

Shared Subscriptions In Depth

In shared subscriptions, each subscription group can be conceptually imagined as a virtual client that acts as a proxy for multiple individual subscribers simultaneously. HiveMQ selects one subscriber of the group and delivers the message to this client. By default, a round-robin approach is used. The following picture demonstrates the principle:





In a HiveMQ deployment, there is no limit on the number of shared subscription groups. For example, the following scenario is possible:

In this example there are two different groups with 2 subscribing clients in each shared subscription group. Both groups have the same subscription but have different group identifiers. When a publisher sends a message with a matching topic, one (and only one) client of each group receives the message.

Shared-subscription Use Cases

There are many use cases for shared subscriptions, especially in high-scalability scenarios. The following use case are some of the most popular:

- Client load balancing for MQTT clients that are unable to handle the load on subscribed topics.
- Worker (backend) applications that ingest MQTT streams and need to scale horizontally.
- Reduction of HiveMQ intra-cluster node traffic through optimization of subscriber node-locality for incoming publishes.
- QoS 1 and 2 are used for the delivery semantics but ordered-topic guarantees are not required.
- Hot topics with higher message rates than other topics in the system are causing a scalability bottleneck.

How to subscribe with Shared Subscriptions?

Subscribing clients with shared subscriptions is straight forward. The following command line execution code (that uses the MQTT CLI) shows two MQTT clients that subscribe to the same subscription group and topic:

```
mqtt sub -h broker.hivemq.com -t '$share/group1/my-share-topic' -i client1 -q 1

mqtt sub -h broker.hivemq.com -t '$share/group1/my-share-topic' -i client2 -q 1
```

Now, both MQTT clients share a subscription on the topic my-share-topic (they are in the virtual group group1). Each client receives 1/2 of the MQTT messages that are sent over the MQTT broker on the topic my-share-topic.

MQTT clients can join and leave the subscription group at any time. If a third client joins the group, each client receives 1/3 of the relevant MQTT messages.

Scaling MQTT Subscribers with Shared Subscriptions

The shared subscriptions are a great way to integrate backend systems with plain MQTT (for example, when it's not feasible to use HiveMQs extension system and dynamic scaling is needed). Shared subscribers can be quickly added whenever they're needed. An arbitrary number of "worker" applications can be used to consume the messages of the shared-subscription group.

In essence, you get work-distribution in a pushing fashion: the broker pushes the messages to the clients of the shared-subscription group.

Shared Subscriptions are perfect for scaling and load balancing MQTT clients. HiveMQ clusters give you additional advantages in terms of latency and scalability because message routing is optimized internally. Get more information on shared subscriptions and HiveMQ clusters in the official documentation.

Conclusion

We have seen that shared subscriptions are a great way to distribute messages across different MQTT subscribers with standard MQTT mechanisms. This feature lets you add MQTT-client load balancing without any proprietary additions to your MQTT clients. This is especially useful for backend systems or "hot-topics" that can quickly overwhelm a single MQTT client.

Chapter 8 - Payload Format Description

Modern IoT projects are often big and complex. These large-scale projects usually require the collaboration of multiple vendors and teams. Since MQTT is the de facto standard protocol for IoT, improved interoperability and transparency across systems topped the list of user requirements for MQTT version 5.

The Payload Format Description feature aims to meet those user requirements.

Payload Format Indicator

The Payload Format Indicator is part of any MQTT packet that can contain a payload. Specifically, a CONNECT packet that contains a WILL message or a PUBLISH packet. The indicator is an optional byte value. A value of 0 indicates an “unspecified byte stream”. A value of 1 indicates a “UTF-8 encoded payload”. If no Payload Format Indicator is provided, the default value is 0.

Content Type

Similar to the Payload Format Indicator, the Content Type is optional and can be part of a CONNECT that contains a WILL message or of any PUBLISH. The Content Type must be a UTF-8 encoded string. The Content Type value identifies the kind of UTF-8 encoded payload. When the Payload Format Indicator is set to 1, a MIME content type descriptor is expected (but not mandatory). Any valid UTF-8 String can be used.

Why Describe the Payload Format?

You can use a combination of the Payload Format Indicator and Content Type to transparently describe the payload content of any application message. This feature opens up the possibility to create and define industry-wide standards for MQTT with variable payload formats. Experts such as Ian Skerrett see this sort of standardization as a natural next step in the rise of the MQTT protocol.

When looking at individual deployments, payload content description in the headers can be really helpful. This information insures correct processing of each message without the need to look inside the actual payload. Based on the exact type of content, individual messages in a system can require different kinds of parsing. In some cases, the persistence of messages is dependant upon the exact type of payload the message contains. Since the specific content-type definitions are subject to user design, the potential applications of this feature are seemingly limitless.

Conclusion

- The Payload Format Indicator defines whether a payload is an undefined array of bytes or a UTF-8 encoded message.
- In the case of UTF-8 encoded messages, the sender can use the content type to define the specifics of the payload.
- These two features open the door to transparent definition of payload content across large systems and, potentially, entire industries.
- As pre-parsing actual payloads becomes obsolete, telegraphing the proper message processing can have a significant positive impact on scalability.
- While it is expected that most users rely on known MIME types to describe content, arbitrary UTF-8 Strings can be used.

Chapter 9 - Request-Response Pattern

MQTT is based on asynchronous messaging that follows the publish-subscribe paradigm: Senders and receivers are decoupled from one another in synchronicity, time, and space and one-to-many relationships are possible. It's important to understand that the request-response pattern of MQTT functions and solves problems in a different way than synchronous, one-to-one based protocols like HTTP.

An MQTT response usually doesn't "answer" a "question" that the request presents. It is possible to implement a use case for MQTT in a way that is blocking and provides one-to-one messaging that responds with specific information based on parameters of the request. However, in most use cases, the request causes a specific action for the receiver and the response contains the result for this action.

Response Topic

A response topic is an optional UTF-8 string that is available in any PUBLISH or CONNECT packet. In a CONNECT packet, the response topic refers to the WILL publish. If the response topic contains a value, the sender automatically identifies the corresponding PUBLISH as a request.

The response topic field represents the topics on which the responses from the receivers of the message are expected. Both the actual topic of the initial PUBLISH (request) and the response topic can have one or more subscribers. It's good practice for the sender of the original PUBLISH (request) to subscribe on the contained response topic before sending out the request.

Correlation Data

Correlation data is optional binary data that follows the response topic. The sender of the request uses the data for identifying to which specific request a response that is received later relates. Response topics can be used without correlation data.

Using the correlation makes it possible for the original sender of the request to handle asynchronous responses that can possibly be sent from multiple receivers. This data is irrelevant to the MQTT broker and only functions as a means to identify the relationship between sender and receiver.

Response Information

In the spirit of enabling transparent implementation and better standardization, the MQTT 5 specification introduced the **Response Information** property. A client can request response information from the broker by setting a boolean field in the CONNECT.

When this **request response information** is set to *true*, the broker can send an optional UTF-8 String field (***response information***) in the CONNACK packet to pass information about the response topics that are expected to be used.

With this feature users can globally define a specific part of your topic tree on the broker, which can then be used by all clients that indicate they want to be using the request-response pattern at connection establishment.

End-to-End Acknowledgement

MQTT ensures that the sender and receiver of messages are completely decoupled. The subscriber (receiver) that gets a message from the broker is in a separate message flow from the publisher (sender) that sends the message to the broker.

There are many use cases that require an acknowledgement of message receipt from the intended recipient. A classic example is opening the door of your smart home. Not only does the sender of the "open door" command (usually a mobile app) want to know when and if the message was received, the sender would also like to know the result of the command.

These so-called "business ACKs" are the primary reason MQTT users were keen on having the request-response pattern introduced to the MQTT 5 specification. MQTT users needed the ability to provide end-to-end acknowledgements between the sender and the receiver of an application message.

Similar to other features in the new protocol specification, this pattern was already used by MQTTv3 users. The introduction of response topics, correlation data, and response information as protocol fields allows for significantly more extensible, dynamic, and transparent application development with the request-response pattern.

Source Code Example

On the next page is a quick source code example that leverages the HiveMQ MQTT Client. Please note, this is not a complete and functioning source code excerpt. The purpose of the example is to demonstrate the intended workflow of the request-response pattern in MQTT. You can find a complete example on GitHub.

```
//Requester subscribes to response topic
Mqtt5SubAck subAck = requester.subscribeWith()
        .topicFilter("job/client1234/result")
        .send();

//Requester publishes request
Mqtt5PublishResult result = requester.publishWith()
       .topic("job")
       .correlationData("1234".getBytes())
       .responseTopic("job/client1234/result")
       .payload(message.getBytes())
       .send();


//Responder subscribes to request topic
Mqtt5SubAck subAck = responder.subscribeWith()
        .topicFilter("job")
        .send();


//Responder sends response after receiving the request
 Mqtt5PublishResult result = responder.publishWith()
         .topic(publish.getResponseTopic().get())
         .payload(msg.getBytes())
         .correlationData(publish.getCorrelationData().get())
         .send();
```

Summary

- The MQTT request-response pattern is not the same as the request-response of synchronous, client-server based protocols like HTTP.
- Requests as well as responses can have more than one or no subscriber in MQTT.
- Correlation data makes sure that the relation between request and response can be upheld properly.
- This mechanism enables extensible, dynamic, and transparent implementation of “business acknowledgement” functionality.

Best Practises and Gotchas

- The requester should always subscribe to the response topic before sending the request.
- Use unique identifiers in the response topic.
- Make sure that intended responders and requesters have the necessary permissions to publish and subscribe to the response topics.
- Reserve a specific part of the topic tree for this purpose and use the response information field to pass it along to the clients.

Chapter 10 - Topic Alias

MQTT enables the use of standing connections between your devices and the brokers. The Keep Alive Mechanism ensures that connections between clients and broker are upheld as long as possible and that any connection loss, which may occur in unstable networks, is detected quickly. With PING packets in the size of two Bytes only having to be sent every few minutes, connections can be upheld at a low cost of power and bandwidth.

The Topic Alias feature is specifically useful in deployments where large numbers of devices are connected and smaller messages are sent at a high frequency.

Topic Aliases

Topic Aliases are an integer value that can be used as a substitute for topic names. A sender can set the Topic Alias value in the PUBLISH message, following the topic name. The message receiver then processes the message like any other PUBLISH and persists a mapping between the Integer (Topic Alias) and String (Topic Name). Any following PUBLISH message for the same topic name can then be sent with an empty topic name, only using the defined Topic Alias.

Restrictions

Both the client and broker can define a Topic Alias for a PUBLISH message, as long as they are the respective sender. Similarly the client and the broker can restrict the number of Topic Aliases they allow per connection. The Topic Alias Maximum is communicated during connection establishment. The client can set it in the CONNECT and the broker in the CONNACK packet. Consequently the client must only use Topic Alias values between 1 and the Topic Alias Maximum the broker sent in the CONNACK. Likewise the broker is only permitted to use values between 1 and the maximum sent in the CONNECT by the client. If no Topic Alias Maximum value is set, a value of 0 is assumed and the use of Topic Aliases is not permitted.

Use Case

MQTT is a lightweight communication protocol. Standing TCP connections between client and broker can be upheld with little energy and bandwidth consumption via the Keep Alive mechanism. This enables MQTT users to build deployments with permanently connected devices at low cost. Minimal data points like measurements can be delivered in real time. There is no need to collect data and send data in bulk packages periodically anymore, as it is necessary with other less lightweight technologies for data transfer.

In a number of use cases, such as predictive maintenance, the quality and responsiveness of the service can be improved by sending small data points in real time. Implementing a use case like this, the deep and individualized topic name may exceed the size of the actual payload, that may be represented by a single integer value. For example a topic name like

data/europe/germany/south/bavaria/munich/schwabing/box-32543y/junction/consumption/current

describes the current power consumption of a specific junction box. The value to be sent is a single integer.

So when you want to send many small messages in real time with long topic names, the Topic Alias feature really shines and helps to increase performance. More importantly it reduces network traffic significantly.

Summary and Gotchas

- Topic Aliases substitute UTF-8 String topic names with an integer
- Mapping of Topic Alias to Topic is always only relevant for a single connection
- Supporting this feature is optional for brokers and clients
- Broker and client negotiate to what degree this feature is supported during the connection establishment
- Be sure to use a broker and client implementation that supports Topic Aliases, if you want to introduce this feature
- Used correctly, Topic Aliases can have significant impact on the profit margins of your business case

Chapter 11 - Enhanced Authentication

Modern IoT projects are often big and complex, especially once security considerations come into play. These large-scale projects usually require the collaboration of multiple vendors and teams. Adherence to internationally accepted standards is one way to limit the challenges such projects entail.

Implementing Challenge-Response Authentication

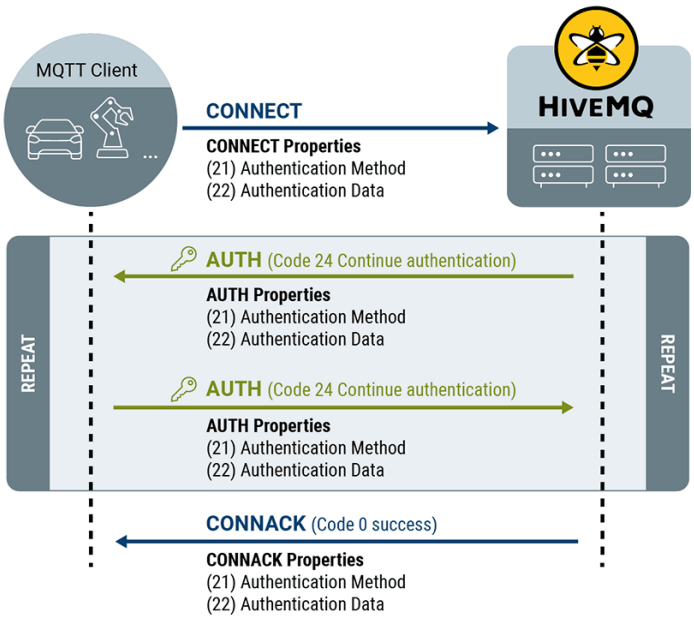
MQTT 5 Enhanced Authentication provides the tools you need to implement authentication in a challenge-response manner. In contrast to the traditional credential-based approach, the server authenticates a client by presenting a challenge that the client must respond to with a valid response.

This allows you to the implement authentication standards such as the Salted Challenge Response Authentication Mechanism (SCRAM) or the Kerberos protocol.

Authentication Flow

Enhanced authentication is based on three MQTT message types: the CONNECT and CONNACK messages that were already present in MQTT v3 and the new MQTT v5 AUTH message. CONNECT messages are only sent by clients and CONNACK messages are only sent by the server. Both types are used one time during each authentication process. AUTH messages can be used multiple times by the server and the client.

Two message properties are at the heart of the authentication flow: the Authentication Method that is identified by byte 21 and the Authentication Data that is identified by byte 22. These properties are set on every message that takes part in the enhanced authentication flow.



Authentication Method

The Authentication Method is used to choose and describe a way of authentication that the client and server have agreed upon. This is done with method strings that are commonly used to identify SASL mechanisms. For example, SCRAM-SHA-1 for SCRAM with SHA-1 or GS2-KRB5 for Kerberos.

The Authentication Method gives meaning to the data that is exchanged during the enhanced authentication and must not change.

Authentication Data

Authentication data is binary information. This data is usually used to transfer multiple iterations of encrypted secrets or protocol steps. The content is highly dependent on the specific mechanism that is used in the enhanced authentication and is application-specific.

Source Code Example

In this brief code snipped we use the HiveMQ extension SDK to implement enhanced authentication that checks for support of the Authentication Method and decides the state of a connecting MQTT client after the exchange of two AUTH messages.

```
public class MyEnhancedAuthenticator implements EnhancedAuthenticator {

    public void onConnect(EnhancedAuthConnectInput input, EnhancedAuthOutput output) {

        final ConnectPacket connectPacket = input.getConnectPacket();

        // Is the given authentication method supported?
        if (authenticationMethodIsSupported(connectPacket.getAuthenticationMethod())) {

            // Did the client provide valid authentication data?
            if (validateClientAuthenticationData(connectPacket.getAuthenticationData())) {

                // Send an AUTH message that contains a challenge!
                output.continueAuthentication(prepareServerAuthenticationData());
                return;
            }
        }

        // Fail the authentication and disconnect the client.
        output.failAuthentication();
    }

```

```
23
24      public void onAuth(EnhancedAuthInput input, EnhancedAuthOutput output) {
25
26          final AuthPacket authPacket = input.getAuthPacket();
27
28          // Try to validate the response.
29          if (validateClientAuthenticationData(authPacket.getAuthenticationData())) {
30
31              // Allow the client to connect to the server.
32              output.authenticateSuccessfully();
33              return;
34          }
35
36          // Fail the authentication and disconnect the client.
37          output.failAuthentication();
38      }
39  }
```

Conclusion

Enhanced authentication is the perfect way to integrate HiveMQ into your existing enterprise security network. This new feature allows you to secure IoT deployments to a degree that was previously not possible.

Chapter 12 - Flow Control

Flow Control

IoT deployments and use cases usually consist of multiple device types. MQTT clients that are embedded in a small sensor can have significantly different characteristics than those that are part of a high-performance back-end server. For example, processing speed or storage capabilities This means that the different MQTT clients have different tolerance levels for managing in-flight messages. In this context, an in-flight message is a PUBLSIH with a Quality of Service of one or two that has not yet been acknowledged. Similarly, an IoT device can connect to multiple MQTT brokers that have different restrictions for the number of in-flight messages from an MQTT client that they can manage.

To address the varying conditions among clients MQTT and brokers in a transparent manner, MQTT 5 introduces the Flow Control feature.

How it works

Client and broker negotiate each other‘s in-flight windows during the connection establishment. The client can set an optional property called *Receive Maximum* in the CONNECT packet (Client Receive Maximum). This value tells the broker the maximum number of unacknowledged PUBLISH messages the client is able to receive. The broker responds with an optional value for *Receive Maximum* in the CONNACK packet (Server Receive Maximum). This value tells the client the maximum number of unacknowledged PUBLISH messages the broker is willing to receive. If this the Receive Maximum value is absent, the default value of 65,535 is used.

Advantages and details

The Flow Control feature enables dynamic message flow adjustment for use cases that involve multiple of non-identical systems and devices (for example, an IoT platform that services multiple tenants). It also creates transparency and flexibility in circumstances where multiple teams or vendors are involved in the same project. With this feature, it is no longer necessary for all involved parties to negotiate in-flight windows beforehand.

If an MQTT 5 client sends more unacknowledged messages than the Server Receive Maximum allows, the broker sends DISCONNECT with Reason Code 0x93 (Receive Maximum exceeded). Both client and broker can choose to send less in-flight messages than the corresponding Receive Maximum allows.

Conclusion

- The use of Receive Maximum is optional
- Client and broker can define their own in-flight windows during connection establishment.
- Flow Control ensures that message processing does not overwhelm any of the involved parties.
- This feature fits right into one the main goals of MQTT 5 - providing increased flexibility and transparency.

Imprint

HiveMQ GmbH
Ergoldinger Str. 2A
84030 Landshut
Germany

Contact information:

Contact us info@hivemq.com
Find out more hivemq.com
Subscribe to our Newsletter

ISBN e-Book: 978-3-00-067913-1

HiveMQ
Ergoldingerstr. 2a
84030 Landshut / Germany

hivemq.com

HIVEMQ