

MQTT Essentials: Part 1 – Introducing MQTT



Welcome to the first part of MQTT Essentials. A blog series about the core features and concepts in the MQTT protocol. This post introduces the MQTT Essentials series and what we'll cover on the blog in 2015. Also it will give an introduction to MQTT and some general information and background on the protocol.

Announcing the MQTT Monday!

The beginning of a year is always great to start new things, so in that motivational spirit, we want to change our blogging concept and feature more posts about MQTT in general. We have a lot of different topics in mind from the now starting essentials series to an in-depth look on security or client libraries. So if you are interested in MQTT you should definitely check our blog regularly or [sign up for our newsletter](#) to get new posts directly to your inbox, when they are published.

Together with this first post, we will also introduce the MQTT Monday. Every Monday we will publish a new blog post regarding MQTT. So you can expect a new part of the Essentials series each upcoming Monday. We hope that the content of these post will help MQTT users of any knowledge level to quickly understand and implement MQTT in their projects and use cases.

MQTT Essentials: Why and what we are going to cover?

Before covering today's topic, we want to explain why we have decided to write the series, who is it for and what topics will be covered. We are working with MQTT for a long time, over 3 years now, and we have answered basic questions about the core concepts like [publish/subscribe](#), [quality of services](#) and many others a lot on different platforms (customers, conferences, online). So with the MQTT Essentials we like to cover the main pillars as a reference guide for users of all kind. MQTT is an open protocol and therefore the information on how to use it should also be open and accessible to anybody, who's interested. We are very excited about this and hope you'll find this content useful.

Now what will be in the MQTT Essentials and what won't? At first we will cover **basic concepts of MQTT** ([publish/subscribe](#), [client/broker](#)) and basic functionality ([Connect](#), [Publish](#), [Subscribe](#)). After that we will look at features like [Quality of Service](#), **Retained Messages**, **Persistent Session**, **Last Will and Testament** and **SYS Topics** one by one. The whole series will be around 10 individual posts. What we explicitly excluded in the Essentials is security. We know that this is a very important topic and shouldn't be neglected. That's why **we plan a separate series just about MQTT and Security** sometime after the Essentials.

Introducing MQTT

MQTT is a Client Server publish/subscribe messaging transport protocol. It is light weight, 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](#)

The abstract of the **MQTT** specification does a good job in describing what MQTT is all about. It is very light weight and binary protocol, which **excels when transferring data over the wire in comparison to protocols like HTTP**, because it has only a minimal packet overhead. Another important aspect is that MQTT is extremely easy to implement on the client side. This fits perfectly for constrained devices with limited resources. Actually this was one of the goals when MQTT was invented in the first place.

A little bit of history

MQTT was invented by Andy Stanford-Clark (IBM) and Arlen Nipper (Arcom, now Cirrus Link) back in 1999, when their use case was to create a protocol for minimal battery loss and minimal bandwidth [connecting oil pipelines over satellite connection](#). They specified the following goals, which the future protocol should have:

- Simple to implement
- Provide a Quality of Service Data Delivery
- Lightweight and Bandwidth Efficient
- Data Agnostic
- Continuous Session Awareness

These goals are still the core of MQTT, while the **focus has changed from proprietary embedded systems to open Internet of Things use cases**. Another thing that is often confused about MQTT is the appropriate meaning of the abbreviation MQTT. It's a long story, the [short answer](#) is that **MQTT officially does not have an acronym anymore, it's just MQTT**.

The long story is that the former acronym stood for:

MQ Telemetry Transport

While MQ is referencing to MQ Series, a product developed by IBM which supports MQTT and the protocol was named after, when it was invented 1999. Often MQTT is incorrectly named as message queue protocol, but this is not true. There are no queues as in traditional message queuing solutions. However, it is possible to queue message in certain cases, but this will be discussed in detail in a later post. So after MQTT had been used by IBM internally for quite some times, version 3.1 was released royalty free in 2010. From there on everybody could implement and use it. We were introduced to it in 2012 and build the first version of HiveMQ in the same year, before releasing it to the public in 2013. But not only the protocol specification was released also various client implementation were contributed to the newly founded Paho project underneath the Eclipse Foundation. This was definitely a big thing for the protocol, because there is little chance for wide adoption when there is no ecosystem around it.

OASIS Standard and current version

Around 3 years after the initial publication, it was announced that MQTT should be standardized under the wings of OASIS, an open organization with the purpose of advancing standards. AMQP, SAML, DocBook are only a few of the already released standards. The standardization process took around 1 year and on October 29th 2014 **MQTT was [officially approved as OASIS Standard](#)**. MQTT 3.1.1 is now the newest version of the protocol. The minor version change from 3.1 to 3.1.1 symbolizes that there were only little changes made to the previous version. The primary goal was to deliver a standard as soon as possible and improve MQTT from there on. For detailed information about the changes, see our blog post about [why it's worth to upgrade to 3.1.1](#).

We would definitely recommend to use MQTT 3.1.1.

So that's already the end of part 1 in our multi-part series of MQTT Essentials. We hope you did enjoy it and learned something new about MQTT. If you have any feedback or questions we should answer, while covering future topics in the next posts, let us know in the comments. By the way the topic of next week will be a introduction into the [publish and subscribe pattern as well as explaining the difference between MQTT and a message queue](#).

MQTT Essentials Part 2: Publish & Subscribe

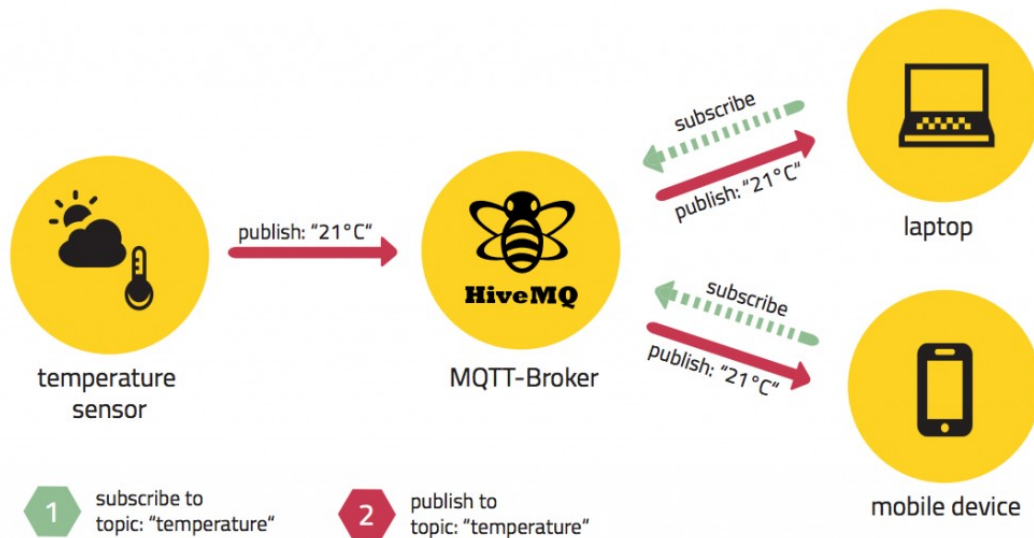


Welcome to the second part of the MQTT Essentials. A blog series about the core features and concepts in the MQTT protocol. In this second post, we'll discuss the Publish/Subscribe pattern. **First we look at the general characteristics of Publish/Subscribe itself and then we'll be focusing on MQTT.** We'll also explain how MQTT is different from traditional message queuing protocols.

In the first post, we [introduced MQTT, explained its origin and history](#). If you haven't already read it, you should definitely check it out.

The publish/subscribe pattern

The publish/subscribe pattern (pub/sub) is an alternative to the traditional client-server model, where a client communicates directly with an endpoint. However, **Pub/Sub decouples a client, who is sending a particular message (called publisher) from another client (or more clients), who is receiving the message (called subscriber).** This means that the publisher and subscriber don't know about the existence of one another. There is a third component, called broker, which is known by both the publisher and subscriber, which filters all incoming messages and distributes them accordingly. So let's dive into a little bit more details about the just mentioned aspects. Remember this is still the basic part about pub/sub in general, we'll talk about MQTT specifics in just a minute.



MQTT Publish / Subscribe

As already mentioned the main aspect in pub/sub is the decoupling of publisher and receiver, which can be differentiated in more dimensions:

- **Space decoupling:** Publisher and subscriber do not need to know each other (by ip address and port for example)
- **Time decoupling:** Publisher and subscriber do not need to run at the same time.
- **Synchronization decoupling:** Operations on both components are not halted during publish or receiving

In summary publish/subscribe decouples publisher and receiver of a message, through filtering of the messages it is possible that only certain clients receive certain messages. The decoupling has three dimensions: Space, Time, Synchronization.

Scalability

Pub/Sub also provides a greater scalability than the traditional client-server approach. This is because operations on the broker can be highly parallelized and processed event-driven. Also often message caching and intelligent routing of messages is decisive for improving the scalability. But it is definitely a challenge to scale publish/subscribe to millions of connections. This can be achieved using clustered broker nodes in order to distribute the load over more individual servers with load balancers. (We will discuss this in detail in a separate post, this would go beyond the scope).

Message Filtering

So what's interesting is, how does the broker filter all messages, so each subscriber only gets the messages it is interested in?

Option 1: Subject-based filtering

The filtering is based on a subject or topic, which is part of each message. The receiving client subscribes on the topics it is interested in with the broker and from there on it gets all message

based on the subscribed topics. Topics are in general strings with an hierarchical structure, that allow filtering based on a limited number of expression.

Option 2: Content-based filtering

Content-based filtering is as the name already implies, when the broker filters the message based on a specific content filter-language. Therefore clients subscribe to filter queries of messages they are interested in. A big downside to this is, that the content of the message must be known beforehand and can not be encrypted or changed easily.

Option 3: Type-based filtering

When using object-oriented languages it is a common practice to filter based on the type/class of the message (event). In this case a subscriber could listen to all messages, which are from type Exception or any subtype of it.

Of course publish/subscribe is not a silver bullet and there are some things to consider, before using it. The decoupling of publisher and subscriber, which is the key in pub/sub, brings a few challenges with it. You have to be aware of the structuring of the published data beforehand. In case of subject-based filtering, both publisher and subscriber need to know about the right topics to use. Another aspect is the delivery of message and that a publisher can't assume that somebody is listening to the messages he sends. Therefore it could be the case that a message is not read by any subscriber.

MQTT

So now we have learned a lot about publish/subscribe in general, but what about MQTT in specific. **MQTT embodies all of the mentioned aspects**, depending on what you want to achieve with it. MQTT decouples the space of publisher and subscriber. So they just have to know hostname/ip and port of the broker in order to publish/subscribe to messages. It also decouples from time, but often this is just a fall-back behavior, because the use case mostly is to delivery message in near-realtime. Of course the broker is able to store messages for clients that are not online. (This requires two conditions: client has connected once and its session is persistent and it has subscribed to a topic with [Quality of Service](#) greater than 0). MQTT is also able to decouple the synchronization, because most client libraries are working asynchronously and are based on callbacks or similar model. So it won't block other tasks while waiting for a message or publishing a message. But there are certain use cases where synchronization is desirable and also possible. Therefore some libraries have synchronous APIs in order to wait for a certain message. But usually the flow is asynchronous. Another thing that should be mentioned is that MQTT is especially easy to use on the client-side. Most pub/sub systems have the most 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. So each message contains a topic, which the broker uses to find out, if a subscribing client will receive the message or not. More details about [topics are explained in Part 5 of the MQTT Essentials](#). It would also be possible to do content-based filtering with the HiveMQ MQTT broker and its [custom plugin system](#).

In order to handle the challenges of a pub/sub system in general, **MQTT has the quality of service (QoS) levels**. It is easily possible to specify that a message gets successfully delivered from the client to the broker or from the broker to a client. But still there is the chance that nobody subscribes to the particular topic. If this is a problem, it depends on the broker how to handles such cases. For example, the [HiveMQ MQTT broker has a plugin system](#), which is capable of identifying such cases and take action or just log every message into a database for historical analytics. In order to mitigate the inflexibility of topics, it is important to design the topic tree very carefully and leave room for extension for use cases in the future. **If you follow these strategies, MQTT is perfect for production setups.**

Distinction from Message Queues

So there are many confusions about MQTT, its name and if it is implemented as a message queue or not. We will try to bring light into the dark and explain the differences. In our [last post](#) we already pointed out that the name MQTT comes from an IBM product called MQseries and has nothing to do with “message queue“. But regardless of the name, what are the differences between MQTT and a traditional message queue?

A message queue stores message until they are consumed

When using message queues, each incoming message will be stored on that queue until it is picked up by any client (often called consumer). Otherwise the message will just be stuck in the queue and waits for getting consumed. It is not possible that message are not processed by any client, like it is in MQTT if nobody subscribes to a topic.

A message will only be consumed by one client

Another big difference is the fact that in a traditional queue a message is processed by only one consumer. So that the load can be distributed between all consumers for a particular queue. In MQTT it is quite the opposite, every subscriber gets the message, if they subscribed to the topic.

Queues are named and must be created explicitly

A queue is far more inflexible than a topic. Before using a queue it has to be created explicitly with a separate command. Only after that it is possible to publish or consume messages. In MQTT topics are extremely flexible and can be created on the fly.

MQTT Essentials Part 3: Client, Broker and Connection Establishment



Welcome to the third part of the MQTT Essentials. A blog series about the core features and concepts in the MQTT protocol. **In this post, we'll discuss the role of MQTT client and broker and the parameters and options available, when connecting to a broker.**

In the last post, we [explained how the publish/subscribe pattern works and how it is applied in MQTT](#). The following is a quick recap of the essence: **Publish/Subscribe decouples a client, which is sending a particular message (called publisher) from another client (or more clients), which is receiving the message (called subscriber).** In order to determine, which message gets to which client, MQTT uses topics. A topic is a hierarchical structured string, which is used for message filtering and routing (More [details](#)).

The last post was really more academical nature as we examined what publish/subscribe is about and how it can be differentiated from a message queuing approach. **This post will be way more practical and stuffed with basic knowledge about MQTT.** Some topics we discuss are definition of MQTT client & broker, basics of an MQTT connection, the Connect Message with its parameters and the establishing of the connection by the acknowledgement of the broker.

Introduction

As we have seen MQTT decouples publisher and subscriber, so a connection of any client is always with the broker. Before we start diving into the connection details, let's make clear what we mean by client and broker.

Client

When talking about a client it almost always means an MQTT client. This includes publisher or subscribers, both of them label an MQTT client that is only doing publishing or subscribing. (In general a MQTT client can be both a publisher & subscriber at the same time). **A MQTT client is any device from a micro controller up to a full fledged server, that has a MQTT library**

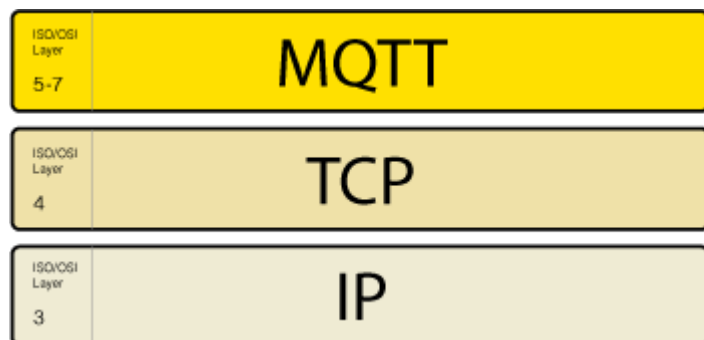
running and is connecting to an MQTT broker over any kind of network. This could be a really small and resource constrained device, that is connected over a wireless network and has a library strapped to the minimum or a typical computer running a graphical MQTT client for testing purposes, basically any device that has a TCP/IP stack and speaks MQTT over it. The client implementation of the MQTT protocol is very straight-forward and really reduced to the essence. That's one aspect, why MQTT is ideally suitable 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, .NET.** A complete list can be found on the [MQTT wiki](#).

Broker

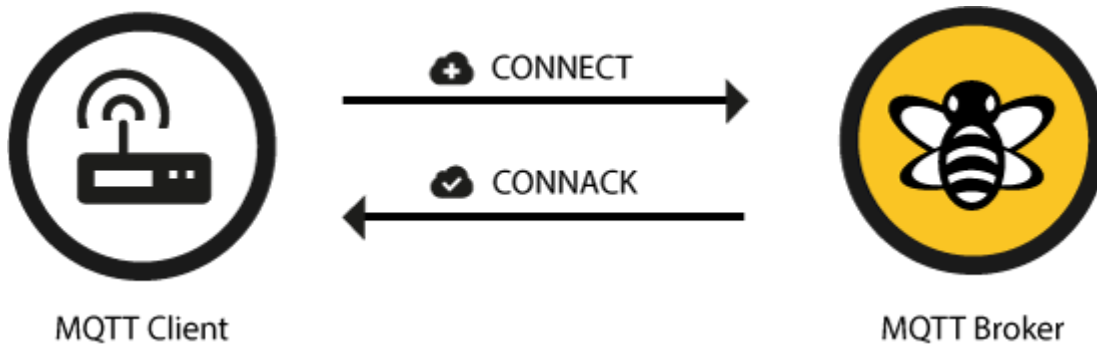
The counterpart to a MQTT client is the MQTT broker, which is the heart of any publish/subscribe protocol. Depending on the concrete implementation, a broker can handle up to thousands of concurrently connected MQTT clients. **The broker is primarily responsible for receiving all messages, filtering them, decide who is interested in it and then sending the message to all subscribed clients.** It also holds the session of all persisted clients including subscriptions and missed messages (More [details](#)). Another responsibility of the broker is the authentication and authorization of clients. And at most of the times a broker is also extensible, which allows to easily integrate custom authentication, authorization and integration into backend systems. Especially the integration is an important aspect, because often the broker is the component, which is directly exposed on the internet and handles a lot of clients and then passes messages along to downstream analyzing and processing systems. As we described in [one of our early blog post](#) subscribing to all message is not really an option. All in all the broker is the central hub, which every message needs to pass. Therefore **it is important, that it is highly scalable, integratable into backend systems, easy to monitor and of course failure-resistant.** For example HiveMQ solves this challenges by using state-of-the-art event driven network processing, an open plugin system and standard providers for monitoring.

MQTT Connection

The MQTT protocol is based on top of TCP/IP and both client and broker need to have a TCP/IP stack.



The MQTT connection itself is always between one client and the broker, no client is connected to another client directly. **The connection is initiated through a client sending a CONNECT message to the broker. The broker response with a CONNACK** and a status code. Once the connection is established, the broker will keep it open as long as the client doesn't send a disconnect command or it loses the connection.



MQTT connection through a NAT

It is a common use case that MQTT clients are behind routers, which are using network address translation (NAT) in order to translate from a private network address (like 192.168.x.x, 10.0.x.x) to a public facing one. As already mentioned the MQTT client is doing the first step by sending a CONNECT message. So there is no problem at all with clients behind a NAT, because the broker has a public address and the connection will be kept open to allow sending and receiving message bidirectional after the initial CONNECT.

Client initiates connection with the CONNECT message

So let's look at the [MQTT CONNECT](#) command message. As already mentioned this is sent from the client to the broker to initiate a connection. If the CONNECT message is malformed (according to the MQTT spec) or it takes too long from opening a network socket to sending it, the broker will close the connection. This is a reasonable behavior to avoid that malicious clients can slow down the broker.

A good-natured client will send 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

Additionally there are other informations included in a CONNECT message, which are more a concern to the implementer of a MQTT library than to the user of a library. If you are interested in the details have a look at the [MQTT 3.1.1 specification](#). So let's go through all these options one by one:

ClientId

The client identifier (short ClientId) is an **identifier of each MQTT client** connecting to a MQTT broker. As the word identifier already suggests, it should be unique per broker. The broker uses it for identifying the client and the current state of the client. If you don't need a state to be hold by the broker, in MQTT 3.1.1 (current standard) it is also possible to send an empty ClientId, which results in a connection without any state. A condition is that clean session is true, otherwise the connection will be rejected.

Clean Session

The clean session flag indicates the broker, whether the **client wants to establish a persistent session or not**. A persistent session (CleanSession is false) means, that the broker will store all subscriptions for the client and also all missed messages, when subscribing with [Quality of Service \(QoS\) 1 or 2](#). If clean session is set to true, the broker won't store anything for the client and will also purge all information from a previous persistent session.

Username/Password

MQTT allows to send a **username and password for authenticating the client and also authorization**. However, the password is sent in plaintext, if it isn't encrypted or hashed by implementation or TLS is used underneath. We highly recommend to use username and password together with a secure transport of it. In brokers like HiveMQ it is also possible to authenticate clients with an SSL certificate, so no username and password is needed.

Will Message

The will message is part of the last will and testament feature of MQTT. **It allows to notify other clients, when a client disconnects ungracefully.** A connecting client will provide his will in form of an MQTT message and topic in the CONNECT message. If this clients gets disconnected ungracefully, the broker sends this message on behalf of the client. We will talk about this in detail in an individual post.

KeepAlive

The keep alive is a time interval, the clients commits to by sending regular PING Request messages to the broker. The broker response with PING Response and this mechanism will allow both sides to determine if the other one is still alive and reachable. We'll talk about this in detail in a future post.

That are basically all information that are necessary to connect to a MQTT broker from a MQTT client. Often each individual library will have additional options, which can be configured. They are most likely regarding the specific implementation, for example how should queued message be stored.

Broker responds with the CONNACK message

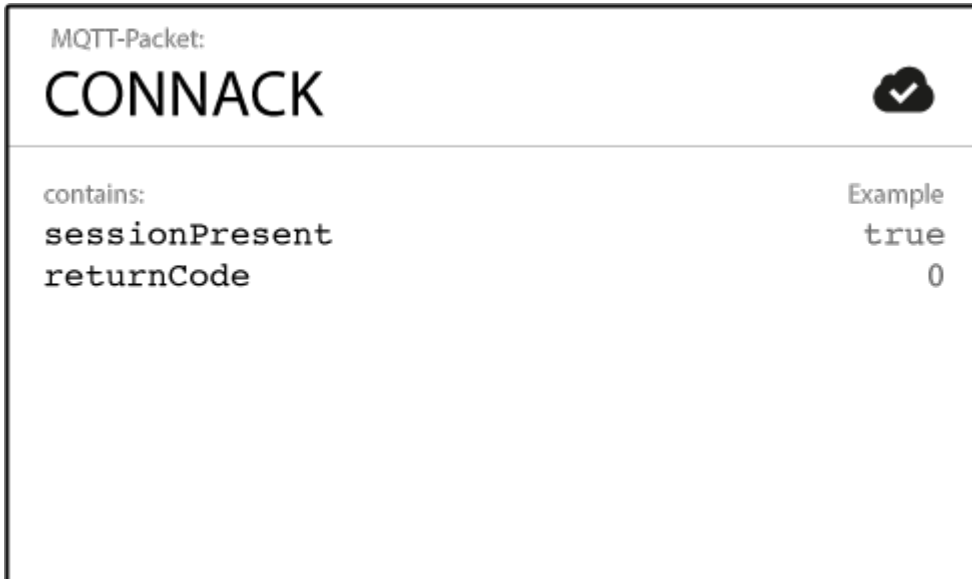
When a broker obtains a CONNECT message, it is obligated to respond with a CONNACK message. The CONNACK contains only two data entries: session present flag, connect return code.

Session Present flag

The **session present flag indicate, whether the broker already has a persistent session of the client from previous interactions.** If a client connects and has set CleanSession to true, this flag is always false, because there is no session available. If the client has set CleanSession to false, the flag is depending on, if there are session information available for the ClientId. If stored session information exist, then the flag is true and otherwise it is false. This flag was added newly in MQTT 3.1.1 and helps the client to determine, whether it has to subscribe to topics or if these are still stored in his session.

Connect acknowledge flag

The second flag in the [CONNACK](#) is the connect acknowledge flag. It signals the client, if the **connection attempt was successful and otherwise what the issue is.**



In the following table you see all 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

A more detailed explanation of each of these can be found in the [MQTT specification](#).

Loose ends

You maybe ask, how MQTT keeps the connection open, even when there are no messages send? Or how to know when a connection is lost? You have to be patient, but we will devote a whole blog inside of the essentials series to that topic later on.

So that's the end of part three in our MQTT Essentials series. We hope you learned at least one new thing about MQTT and looking forward to the next post about [how publishing, subscribing and unsubscribing works in MQTT](#).

MQTT Essentials Part 4: MQTT Publish, Subscribe & Unsubscribe



Welcome to the fourth part of the MQTT Essentials, a blog series about the core features and concepts in the MQTT protocol. **In this post, we'll focus on publish, subscribe and unsubscribe in MQTT.** In contrast to the second part, which was about publish/subscribe basics, this post will explain the specifics of publish and subscribe in the MQTT protocol. If you haven't read [part 2 about the basics of the publish/subscribe pattern](#), we strongly encourage you to read it first.

Last week we looked at [establishing a connection between MQTT client and broker](#). So this week's post ties on to this and we'll talk about sending and receiving messages.

Publish

After a MQTT client is connected to a broker, it can publish messages. MQTT has a topic-based filtering of the messages on the broker (check [part 2](#) for more details on that), so **each message must contain a topic, which will be used by the broker to forward the message to interested clients. Each message typically has a payload which contains the actual data to transmit in byte format.** MQTT is data-agnostic and it totally depends on the use case how the payload is structured. It's completely up to the sender if it wants to send binary data, textual data or even full-fledged XML or JSON. A MQTT publish message also has some more attributes, which we're going to discuss in detail:

MQTT-Packet:	
PUBLISH	
contains:	Example
<code>packetId</code> (always 0 for qos 0)	4314
<code>topicName</code>	"topic/1"
<code>qos</code>	1
<code>retainFlag</code>	false
<code>payload</code>	"temperature:32.5"
<code>dupFlag</code>	false

Topic

Name

A simple string, which is hierarchically structured with forward slashes as delimiters. An example would be "myhome/livingroom/temperature" or "Germany/Munich/Octoberfest/people". More details about topics can be found in [part 5 of the MQTT Essentials](#).

QoS

A Quality of Service Level (QoS) for this message. The level (0,1 or 2) determines the guarantee of a message reaching the other end (client or broker). More details about QoS can be found in [part 6 of the MQTT Essentials](#).

Retain-Flag

This flag determines if the message will be saved by the broker for the specified topic as last known good value. New clients that subscribe to that topic will receive the last retained message on that topic instantly after subscribing. More on retained messages and best practices in one of the next posts.

Payload

This is the actual content of the message. MQTT is totally data-agnostic, it's possible to send images, texts in any encoding, encrypted data and virtually every data in binary.

Packet

Identifier

The packet identifier is a unique identifier between client and broker to identify a message in a message flow. This is only relevant for QoS greater than zero. Setting this MQTT internal identifier is the responsibility of the client library and/or the broker.

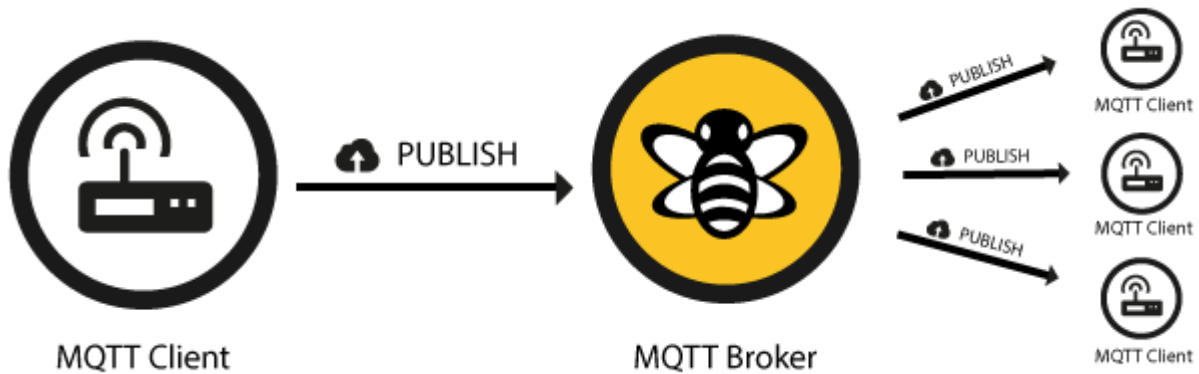
DUP

flag

The duplicate flag indicates, that this message is a duplicate and is resent because the other end didn't acknowledge the original message. This is only relevant for QoS greater than 0 and more details are in [part 6, which is about QoS levels](#). This resend/duplicate mechanism is typically handled by the MQTT client library or the broker as an implementation detail.

So when a client sends a publish to a MQTT broker, **the broker will read the publish, acknowledge the publish if needed (according to the QoS Level) and then process it.**

Processing includes determining which clients have subscribed on that topic and then sending out the message to the selected clients which subscribe to that topic.



The client, who initially published the message is only concerned about delivering the publish message to the broker. From there on it is the responsibility of the broker to deliver the message to all subscribers. The publishing client doesn't get any feedback, if someone was interested in this published message and how many clients received the message by the broker.

Subscribe

Publishing messages doesn't make sense if no one ever receives the message, or, in other words, if there are no clients subscribing to any topic. A client needs to send a [SUBSCRIBE](#) message to the MQTT broker in order to receive relevant messages. A subscribe message is pretty simple, it just contains a unique packet identifier and a list of subscriptions.



Packet

Identifier

The packet identifier is a unique identifier between client and broker to identify a message in a message flow. Setting this MQTT internal identifier is the responsibility of the client library and/or the broker.

List

of


Subscriptions

A SUBSCRIBE message can contain an arbitrary number of subscriptions for a client. Each

subscription is a pair of a topic and QoS level. The topic in the subscribe message can also contain wildcards, which makes it possible to subscribe to certain topic patterns. If there are overlapping subscriptions for one client, the highest QoS level for that topic wins and will be used by the broker for delivering the message.

Suback

Each subscription will be confirmed by the broker through sending an acknowledgment to the client in form of the [SUBACK](#) message. This message contains the same packet identifier as the original Subscribe message (in order to identify the message) and a list of return codes.

MQTT-Packet:		
SUBACK		
contains:		Example
packetId		4313
returnCode 1	(one returnCode for each	2
returnCode 2	topic from SUBSCRIBE,	0
...	in the same order)	...

Packet

Identifier

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

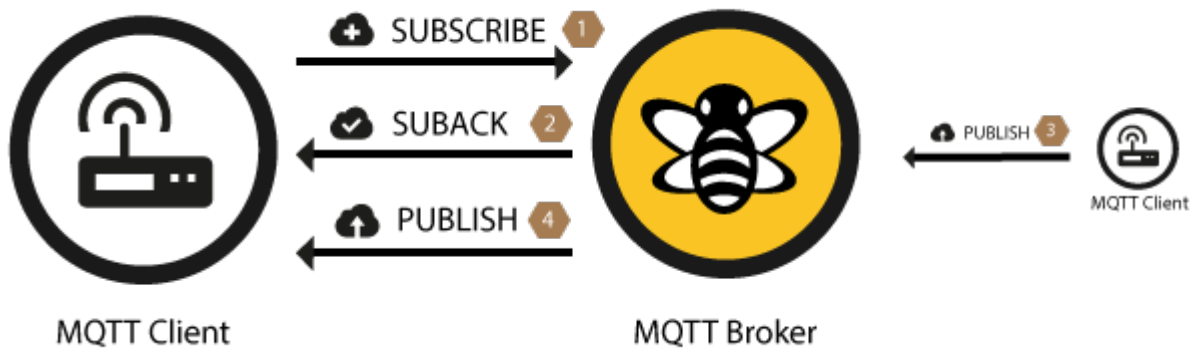
Return

Code

The broker sends one return code for each topic/QoS-pair it received in the SUBSCRIBE message. So if the SUBSCRIBE message had 5 subscriptions, there will be 5 return codes to acknowledge each topic with the QoS level granted by the broker. If the subscription was prohibited by the broker (e.g. if the client was not allowed to subscribe to this topic due to insufficient permission or if the topic was malformed), the broker will respond with a failure return code for that specific topic.

Return Code Return Code Response

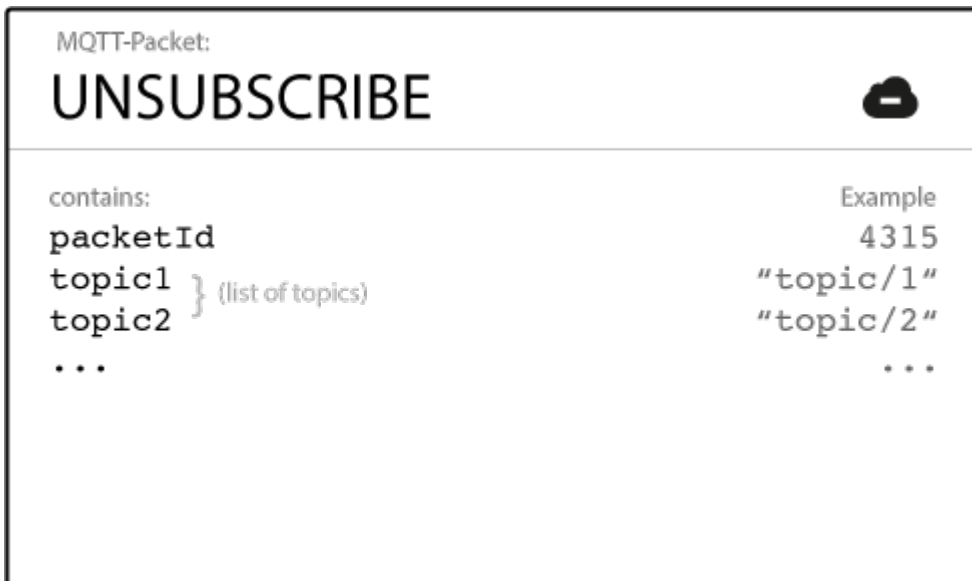
0	Success	– Maximum QoS 0
1	Success	– Maximum QoS 1
2	Success	– Maximum QoS 2
128	Failure	



After a client successfully sent the SUBSCRIBE message and received the SUBACK message, it will receive every published message matching the topic of the subscription.

Unsubscribe

The counterpart of the SUBSCRIBE message is the [UNSUBSCRIBE](#) message which deletes existing subscriptions of a client on the broker. The UNSUBSCRIBE message is similar to the SUBSCRIBE message and also has a packet identifier and a list of topics.



Packet

Identifier

The packet identifier is a unique identifier used to identify a message. The acknowledgement of an UNSUBSCRIBE message will contain the same identifier.

List

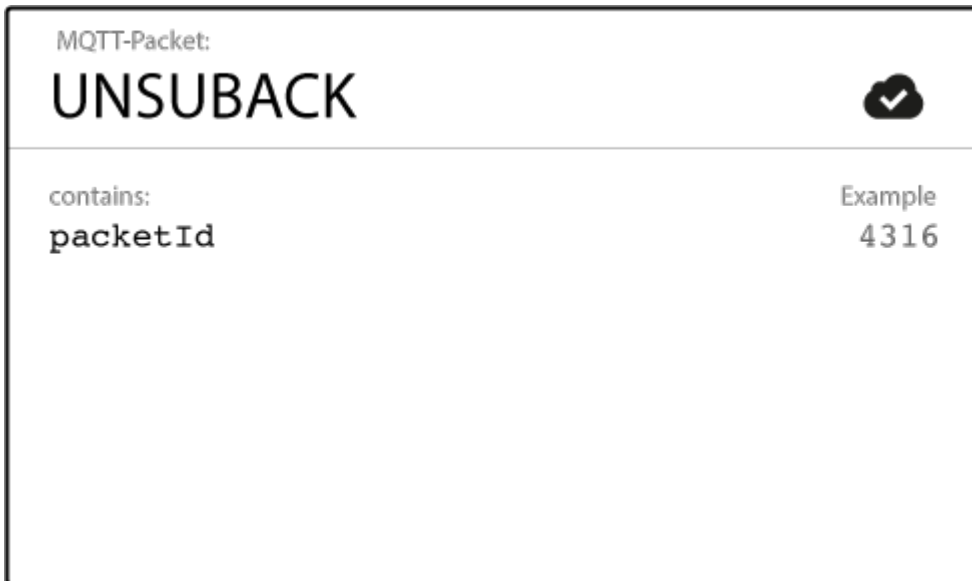
of

Topic

The list of topics contains an arbitrary number of topics, the client wishes to unsubscribe from. It is only necessary to send the topic as string (without QoS), the topic will be unsubscribed regardless of the QoS level it was initially subscribed with.

Unsuback

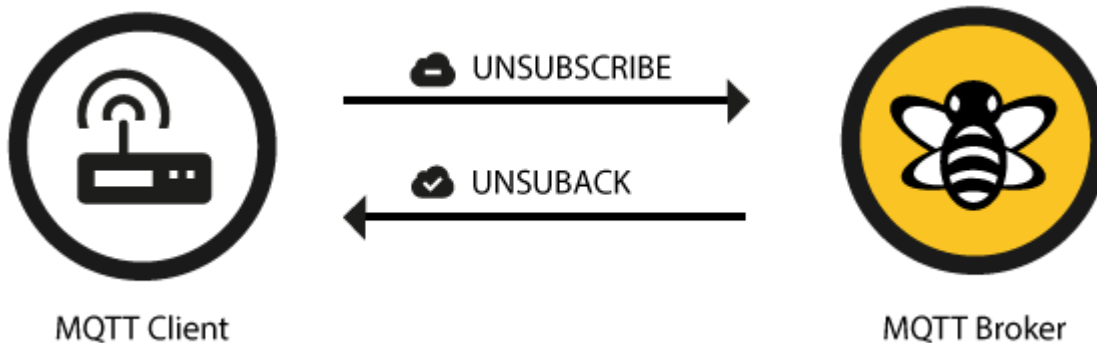
The broker will acknowledge the request to unsubscribe with the [UNSUBACK](#) message. This message only contains a packet identifier.



Packet

Identifier

The packet identifier is a unique identifier used to identify a message. It is the same as in the UNSUBSCRIBE message.



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

So that's the end of part four in our MQTT Essentials series. We hope you enjoyed it. In the next post we will dig deeper into the usage of MQTT topics. We'll explain the basics as well as the usage of wildcards and a lot of practical examples.

MQTT Essentials Part 5: MQTT Topics & Best Practices



Welcome to the fifth part of the MQTT Essentials, a blog series about the core features and concepts in the MQTT protocol. **In this post we'll focus on MQTT topics and best practices.** [As we have already mentioned](#), topics are used to decide on the MQTT broker which client receive which message. We will also discuss *SYS-topics*, which are special ones that reveal broker internal information. So let's get started.

Topics

A topic is a UTF-8 string, which is used by the broker to filter messages for each connected client. A 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 a topic is very lightweight. There is no need for a client to create the desired topic before publishing or subscribing to it, because a broker accepts each valid topic without any prior initialization.

Here are a few example topics:

```
myhome/groundfloor/livingroom/temperature  
USA/California/San Francisco/Silicon Valley  
5ff4a2ce-e485-40f4-826c-b1a5d81be9b6/status  
Germany/Bavaria/car/2382340923453/latitude
```


Noticeable is that each topic must have **at least 1 character** to be valid and it can also contain spaces. Also **a topic is case-sensitive**, which makes *myhome/temperature* and *MyHome/Temperature* two individual topics. Additionally the forward slash alone is a valid topic, too.

Wildcards

When a client subscribes to a topic it can use the exact topic the message was published to or it can subscribe to more topics at once by using wildcards. A wildcard can only be used when subscribing to topics and is not permitted when publishing a message. In the following we will look at the two different kinds one by one: *single level* and *multi level* wildcards.

Single Level: +

As the name already suggests, a single level wildcard is a substitute for one topic level. The plus symbol represents a single level wildcard in the topic.



Any topic matches to a topic including the single level wildcard if it contains an arbitrary string instead of the wildcard. For example a subscription to *myhome/groundfloor/+/temperature* would match or not match the following topics:

- ✓ myhome / groundfloor / livingroom / temperature
- ✓ myhome / groundfloor / kitchen / temperature
- ✗ myhome / groundfloor / kitchen / brightness
- ✗ myhome / firstfloor / kitchen / temperature
- ✗ myhome / groundfloor / kitchen / fridge / temperature

Multi Level:

While the single level wildcard only covers one topic level, the multi level wildcard covers an arbitrary number of topic levels. In order to determine the matching topics it is required that the multi level wildcard is always the last character in the topic and it is preceded by a forward slash.



- ✓ myhome / groundfloor / livingroom / temperature
- ✓ myhome / groundfloor / kitchen / temperature
- ✓ myhome / groundfloor / kitchen / brightness
- ✗ myhome / **firstfloor** / kitchen / temperature

A client subscribing to a topic with a multi level wildcard is receiving all messages, which start with the pattern before the wildcard character, no matter how long or deep the topics will get. If you only specify the multilevel wildcard as a topic (#), it means that you will get every message sent over the MQTT broker. If you expect high throughput this is an anti pattern, see the best practices below.

Topics beginning with \$

In general you are totally free in naming your topics, but there is one exception. **Each topic, which starts with a \$-symbol will be treated specially** and is for example not part of the subscription when subscribing to #. **These topics are reserved for internal statistics of the MQTT broker.** Therefore it is not possible for clients to publish messages to these topics. At the moment there is no clear official standardization of topics that must be published by the broker. It is common practice to use \$SYS/ for all these information and a lot of brokers implement these, but in different formats. One suggestion on \$SYS-topics is in the [MQTT GitHub wiki](#) and here are some examples:

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

Summary

So these were the basics about MQTT message topics. As you can see, MQTT topics are dynamically and give great flexibility to its creator. But when using these in real world applications there are some challenges you should be aware of. We collected our best practices, we learned the last year with excessively using MQTT in various projects. We are open to other suggestions or a discussion about these in the comments, so let us know your best practices or if you disagree with one of our best practices!

Best practices

Don't use a leading forward slash

It is allowed to use a leading forward slash in MQTT, for example */myhome/groundfloor/livingroom*. But that introduces a unnecessary topic level with a zero character at the front. That should be avoided, because it doesn't provide any benefit and often leads to confusion.

Don't use spaces in a topic

A space is the natural enemy of each programmer, they often make it much harder to read and debug topics, when things are not going the way, they should be. So similar to the first one, only because something is allowed doesn't mean it should be used. [UTF-8 knows many different white space types](#), it's pretty obvious that such uncommon characters should be avoided.

Keep the topic short and concise

Each topic will be included in every message it is used in, so you should think about making them short and concise. When it comes to small devices, each byte counts and makes really a difference.

Use only ASCII characters, avoid non printable characters

Using non-ASCII UTF-8 character makes it really hard to find typos or issues related to the character set, because often they can not be displayed correctly. Unless it is really necessary we recommend avoid using non ASCII character in a topic.

Embed a unique identifier or the ClientId into the topic

In some cases it is very helpful, when the topic contains a unique identifier of the client the publish is coming from. This helps identifying, who send the message. Another advantage is the enforcement of authorization, so that only a client with the same ClientId as contained in the topic is allowed to publish to that topic. So a client with the id *client1* is allowed to publish to *client1/status*, but not permitted to publish to *client2/status*.

Don't subscribe to #

Sometimes it is necessary to subscribe to all messages, which are transferred over the broker, for example when persisting all of them into a database. **This should not be done by using a MQTT client and subscribing to the multi level wildcard.** The reason is that often the subscribing client is not able to process the load of messages that is coming its way. Especially if you have a massive throughput. Our recommended solution is to implement an extension in the MQTT broker, for example the [plugin system of HiveMQ](#) allows you to hook into the behavior of HiveMQ and add a asynchronous routine to process each incoming message and persist it to a database.

Don't forget extensibility

Topics are a flexible concept and there is no need to preallocate them in any kind of way, regardless both the publisher and subscriber need to be aware of the topic. So it is important to think about how they can be extended in case you are adding new features to your product. For example when your smart home solution is extended by some new sensors, it should be possible to add these to your topic tree without changing the whole topic hierarchy.

Use specific topics, instead of general ones

When naming topics it is important not to use them like a queue, for example using only one topic for all messages is a anti pattern. You should use as specific topics as possible. So if you have three sensors in your living room, you should use topics *myhome/livingroom/temperature*, *myhome/livingroom/brightness* and *myhome/livingroom/humidity*, instead of sending all values over

myhome/livingroom. Also this enables you to use other MQTT features like retained messages, which we cover in one of the next posts.

So that's the end of part five in our MQTT Essentials series. We hope you enjoyed it. In the next post we cover the often mention Quality of Service (QoS) in MQTT. We'll explain why this is an essential feature and how you can leverage it.

MQTT Essentials Part 6: Quality of Service 0, 1 & 2



Welcome to the sixth part of the *MQTT Essentials*, a blog series about the core features and concepts in the MQTT protocol. **In this post we'll focus on the different Quality of Service levels within MQTT.** We already stumbled upon the term 'quality of service' a few times in some of the previous posts, so now is the time to explain what's behind it.

Quality of Service

What is Quality of Service?

The **Quality of Service (QoS)** level is an agreement between sender and receiver of a message regarding the guarantees of delivering a message. There are 3 QoS levels in MQTT:

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

When talking about QoS there are always two different parts of delivering a message: publishing client to broker and broker to subscribing client. We need to look at them separately since there are subtle differences. The QoS level for publishing client to broker is depending on the QoS level the client sets for the particular message. When the broker transfers a message to a subscribing client it uses the QoS of the subscription made by the client earlier. That means, QoS guarantees can get downgraded for a particular receiving client if subscribed with a lower QoS.

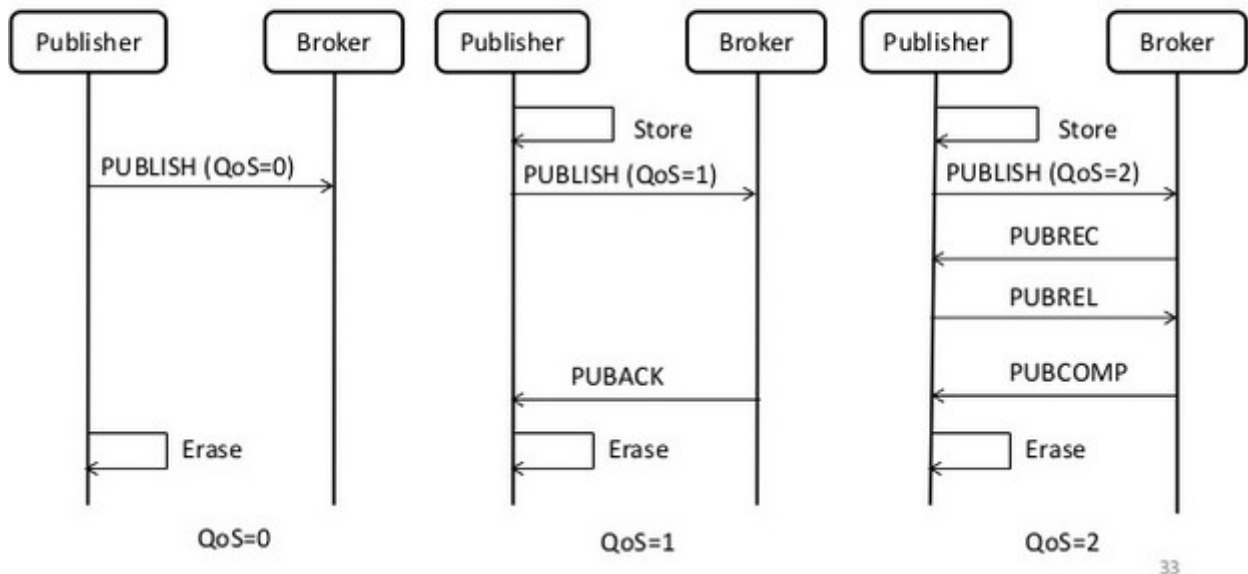
Why is Quality of Service important?

QoS is a major feature of MQTT, it makes communication in unreliable networks a lot easier because the protocol handles retransmission and guarantees the delivery of the message, regardless how unreliable the underlying transport is. Also it empowers a client to choose the QoS level depending on its network reliability and application logic.

How does it work?

So how is the quality of service implemented in the MQTT protocol ? We will look at each level one by one and explain the functionality.

MQTT V3.1 supports 3 levels of Quality of Service (QoS) that represents the message delivery confidence. As the QoS level increases, there are more acknowledgement messages and better reliability on message delivery.



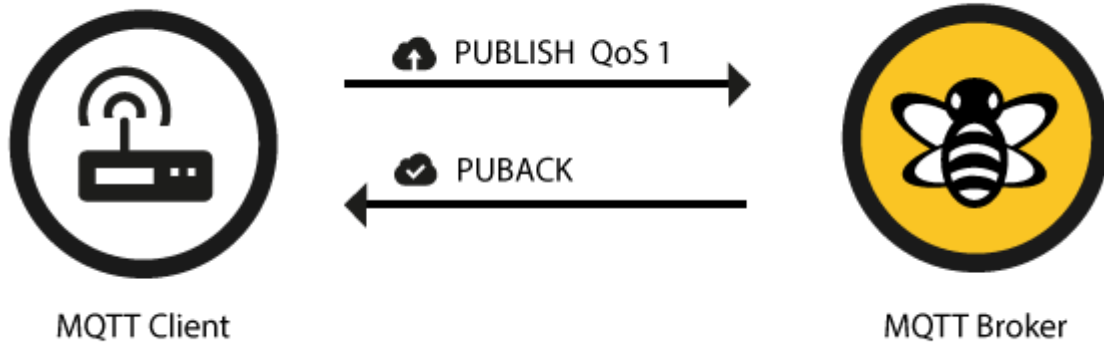
QoS 0 – at most once

The minimal level is zero and it guarantees a best effort delivery. A message won't be acknowledged by the receiver or stored and redelivered by the sender. This is often called “fire and forget” and provides the same guarantee as the underlying TCP protocol.

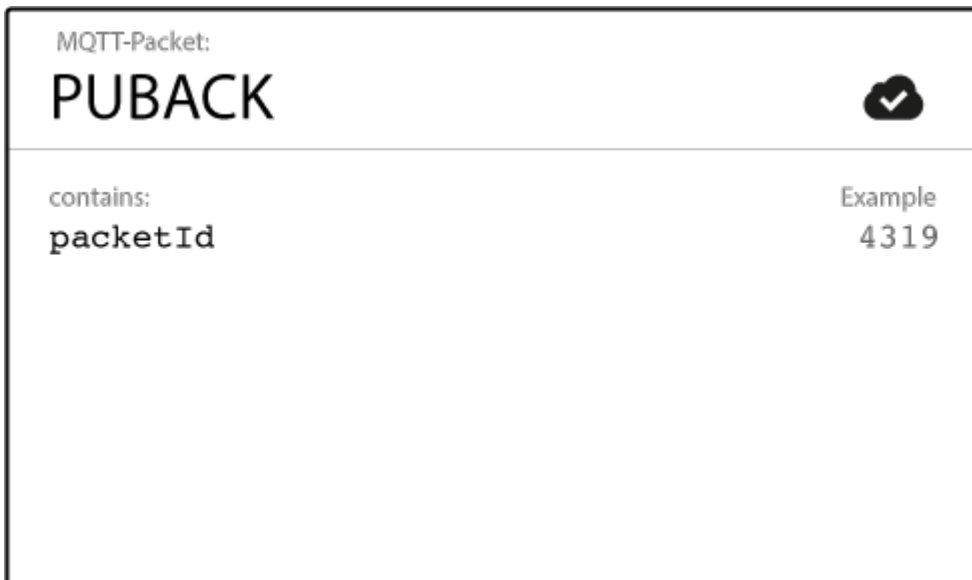


QoS 1 – at least once

When using QoS level 1, it is guaranteed that a message will be delivered at least once to the receiver. But the message can also be delivered more than once.



The sender will store the message until it gets an acknowledgement in form of a [PUBACK](#) command message from the receiver.

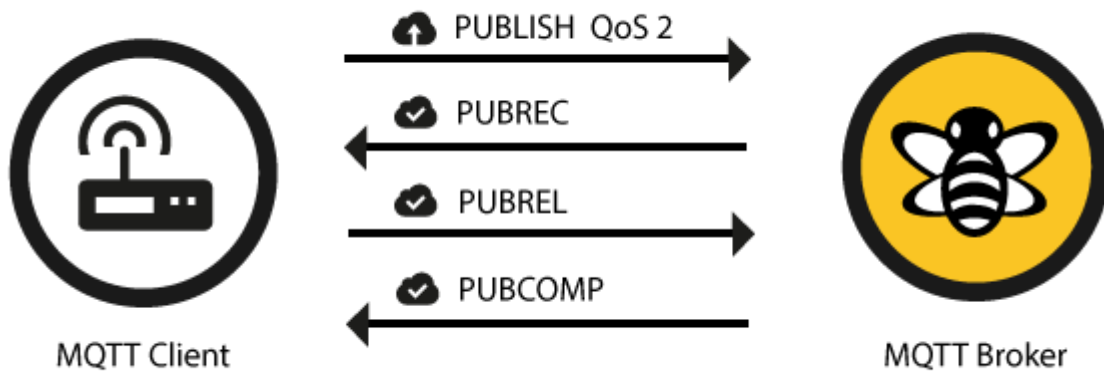


The association of PUBLISH and PUBACK is done by comparing the packet identifier in each packet. If the PUBACK isn't received in a reasonable amount of time the sender will resend the PUBLISH message. If a receiver gets a message with QoS 1, it can process it immediately, for example sending it to all subscribing clients in case of a broker and then replying with the PUBACK.

The duplicate (DUP) flag, which is set in the case a PUBLISH is redelivered, is only for internal purposes and won't be processed by broker or client in the case of QoS 1. The receiver will send a PUBACK regardless of the DUP flag.

QoS 2

The highest QoS is 2, it guarantees that each message is received only once by the counterpart. It is the safest and also the slowest quality of service level. The guarantee is provided by two flows there and back between sender and receiver.



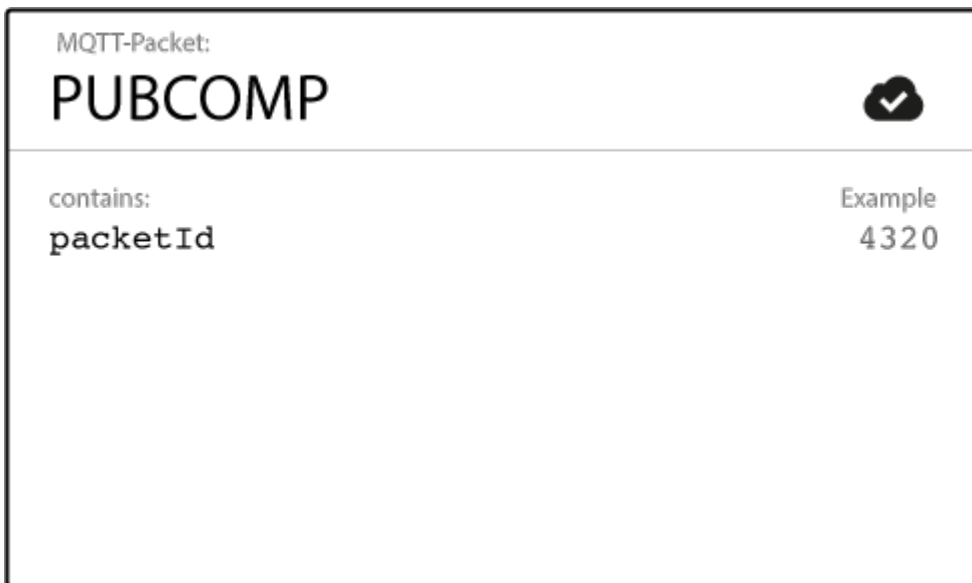
If a receiver gets a QoS 2 PUBLISH it will process the publish message accordingly and acknowledge it to the sender with a [PUBREC](#) message.



The receiver will store a reference to the packet identifier until it has send the PUBCOMP. This is important for avoid processing the message a second time. When the sender receives the PUBREC it can safely discard the initial publish, because it knows that the counter part has successfully received the message. It will store the PUBREC and respond with a [PUBREL](#).



After the receiver gets the PUBREL it can discard every stored state and answer with a [PUBCOMP](#). The same is true when the sender receives the PUBCOMP.



When the flow is completed both parties can be sure that the message has been delivered and the sender also knows about it.

Whenever a packet gets lost on the way, the sender is responsible for resending the last message after a reasonable amount of time. This is true when the sender is a MQTT client and also when a MQTT broker sends a message. The receiver has the responsibility to respond to each command message accordingly.

Good to know

There are a few things you should have in mind when using QoS. These are not obvious or clear on first sight.

Downgrade of QoS

As already said, the QoS flows between a publishing and subscribing client are two different things as well as the QoS can be different. That means the QoS level can be different from client A, who publishes a message, and client B, who receives the published message. Between the sender and the broker the QoS is defined by the sender. When the broker sends out the message to all subscribers, the QoS of the subscription from client B is used. If client B has subscribed to the broker with QoS 1 and client A sends a QoS 2 message, it will be received by client B with QoS 1. And of course it could be delivered more than once to client B, because QoS 1 only guarantees to deliver the message at least once.

Packet identifiers are unique per client

Also important to know is that each packet identifier (used for QoS 1 and QoS 2) is unique between one client and a broker and not between all clients. If a flow is completed the same packet identifier can be reused anytime. That's also the reason why the packet identifier doesn't need to be bigger than 65535, because it is unrealistic that a client sends a such large number of message, without completing the flow.

Best Practice

We are often asked, when to choose which QoS level. The following should provide you some guidance if you are also confronted with this decision. Often this is heavily depending on your use case.

Use QoS 0 when ...

- You have a complete or almost stable connection between sender and receiver. A classic use case is when connecting a test client or a front end application to a MQTT broker over a wired connection.
- You don't care if one or more messages are lost once a while. That is sometimes the case if the data is not that important or will be send at short intervals, where it is okay that messages might get lost.
- You don't need any 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. The most often used QoS is level 1, because it guarantees the message arrives at least once. Of course your application must be tolerating duplicates and process them accordingly.
- You can't bear the overhead of QoS 2. Of course QoS 1 is a lot faster in delivering messages without the guarantee of level 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 would do harm to application users or subscribing clients. You should be aware of the overhead and that it takes a bit longer to complete the QoS 2 flow.

Queuing of QoS 1 and 2 messages

All messages sent with QoS 1 and 2 will also be queued for offline clients, until they are available again. But queuing is only happening, if the client has a [persistent session](#).

So that's the end of part six in our MQTT Essentials series. We hope you enjoyed it. In the next post we'll cover persistent sessions in MQTT, which are tied up closely with Quality of Service levels.

MQTT Essentials Part 7: Persistent Session and Queuing Messages



Welcome to the seventh part of the MQTT Essentials, a blog series about the core features and concepts in the MQTT protocol. In this post we talk about persistent sessions and message queuing in MQTT. Although [MQTT is not a message queue per se](#), it is possible to queue messages for clients.

Persistent Session

When a client connects to a MQTT broker, it needs to create [subscriptions for all topics](#) that it is interested in in order to receive messages from the broker. On a reconnect these topics are lost and the client needs to subscribe again. This is the normal behavior with no persistent session. But for constrained clients with limited resources it would be a burden to subscribe again each time they lose the connection. So a persistent session saves all information relevant for the client on the broker. The session is identified by the *clientId* provided by the client on connection establishment ([more details](#)).

So what will be stored in the session?

- Existence of a session, even if there are no subscriptions
- All subscriptions
- All messages in a [Quality of Service \(QoS\) 1 or 2](#) flow, which are not confirmed by the client
- All new QoS 1 or 2 messages, which the client missed while it was offline
- All received QoS 2 messages, which are not yet confirmed to the client

That means even if the client is offline all the above will be stored by the broker and are available right after the client reconnects.

How to start/end a persistent session?

A persistent session can be requested by the client on connection establishment with the broker. The client can control, if the broker stores the session using the *cleanSession* flag ([see MQTT Essentials part 3 for more information on the connection establishment between client and broker](#)). If the clean session is set to true then the client does not have a persistent session and all information are lost when the client disconnects for any reason. When clean session is set to false, a persistent session is created and it will be preserved until the client requests a clean session again. If there is already a session available then it is used and queued messages will be delivered to the client if available.

How does the client know if there is already a session stored?

Since MQTT 3.1.1, the *CONNACK* message from the broker contains the *session present flag*, which indicates to the client if there is a session available on the broker. For detailed information on the [connection establishment see part 3 of the MQTT Essentials](#).

Persistent session on the client side

Similar to the broker, each MQTT client must store a persistent session too. So when a client requests the server to hold session data, it also has the responsibility to hold some information by itself:

- All messages in a QoS 1 or 2 flow, which are not confirmed by the broker
- All received QoS 2 messages, which are not yet confirmed to the broker

Best practices

When you should use a persistent session and when a clean session?

Persistent Session

- A client must get all messages from a certain topic, even if it is offline. The broker should queue the messages for the client and deliver them as soon as the client is online again.
- A client has limited resources and the broker should hold its subscription, so the communication can be restored quickly after it got interrupted.
- The client should resume all QoS 1 and 2 publish messages after a reconnect.

Clean session

- A client is not subscribing, but only publishing messages to topics. It doesn't need any session information to be stored on the broker and publishing messages with QoS 1 and 2 should not be retried.
- A client should explicitly not get messages for the time it is offline.

How long are messages stored on the broker ?

A often asked question is how long is a session stored on the broker. The easy answer is until the clients comes back online and receives the message. *But what happens if a client does not come*

online for a long time? The constraint for storing messages is often the memory limit of the operating system. There is no standard way on what to do in this scenario. It totally depends on the use case. In HiveMQ we will provide a possibility to manipulate queued message and purge them.

So that's the end of part seven in our MQTT Essentials series. We hope you enjoyed it. In the next post we'll cover Retained Messages. If you already tried out MQTT, you surely noticed the retained flag, when sending a message. Next week we'll cover what Retained Messages are and how they work.

MQTT Essentials Part 8: Retained Messages



Welcome to the eighth part of the MQTT Essentials, a blog series about the core features and concepts in the MQTT protocol. In this post we will introduce **retained messages**.

When publishing MQTT messages, a publishing client has no guarantee that a message is actually received by a subscribing client. It can only make sure its message gets delivered safely to the broker. The same is true for a subscribing client. If a client is connecting and subscribing to topics it is interested in, there is no guarantee when the subscriber will get the first message, because this totally depends on a publisher on that topic. It can take a few seconds, minutes or hours until the publisher sends a new message on that topic. Until then the subscribing client is totally in the dark about the current status. This 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 will store the last retained message and the corresponding QoS for that topic Each client that subscribes to a topic pattern, which matches the topic of the retained message, will receive the message immediately after subscribing. For each topic only one retained message will be stored by the broker.

The subscribing client doesn't have to match the exact topic, it will also receive a retained message if it subscribes to a topic pattern including wildcards. For example client A publishes a retained message to *myhome/livingroom/temperature* and client B subscribes to *myhome/#* later on, client B will receive this retained message directly after subscribing. Also the subscribing client can identify if a received message was a retained message or not, because the broker sends out retained messages with the retained flag still set to true. A client can then decide on how to process the message.

So retained messages can help newly subscribed clients to get a status update immediately after subscribing to a topic and don't have to wait until a publishing clients send the next update.

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

It is important to understand that a retained message has nothing to do with a [persistent session of any client, which we covered in the last episode](#). Once a retained message is stored by the broker, the only way to remove it is explained below.

Send a retained message

Sending a retained message from the perspective of a developer is quite simple and straightforward. You just need to set the *retained flag* of a [MQTT publish message](#) to true. Each client library typically provides an easy way to do that.

Delete a retained message

There is also a very simple way for deleting a retained message on a topic: Just send a retained message with a zero byte payload on that topic where the previous retained message should be deleted. The broker deletes the retained message and all new subscribers won't get a retained message for that topic anymore. Often deleting is not necessary, because each new retained message will overwrite the last one.

Why and when you should use Retained Messages ?

A retained message makes sense, when newly connected subscribers should receive messages immediately and shouldn't have to wait until a publishing client sends the next message. This is extremely helpful when for status updates of components or devices on individual topics. For example the status of device1 is on the topic *myhome/devices/device1/status*, a new subscriber to the topic will get the status (online/offline) of the device immediately after subscribing when retained messages are used. The same is true for clients, which send data in intervals, temperature, GPS coordinates and other data. **Without retained messages new subscribers are kept in the dark between publish intervals.** So using retained messages helps to provide the last good value to a connecting client immediately.

So that's the end of part eight in our MQTT Essentials series. We hope you enjoyed it. In the next post we'll cover a feature called Last Will and Testament. It makes it possible to send a last message, when a client is disconnected abruptly.

MQTT Essentials Part 9: Last Will and Testament



Welcome to the ninth part of the MQTT Essentials, a blog series about the core features and concepts in the MQTT protocol. In this post we will cover the **Last Will and Testament** feature of MQTT.

MQTT is often used in scenarios where unreliable networks are very common. Therefore it is assumed that some clients will disconnect ungracefully from time to time, because they lost the connection, the battery is empty or any other imaginable case. Therefore it would be good to know, if a connected client has disconnected gracefully (which means with a MQTT *DISCONNECT* message) or not, in order to take appropriate action. The Last Will and Testament feature provides a way for clients to do exactly that.

Last Will and Testament

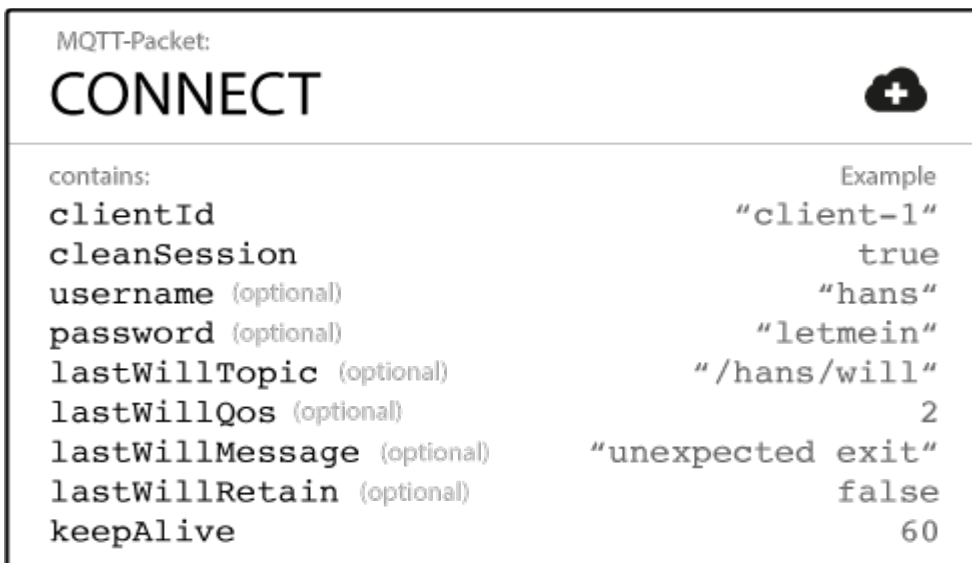
The Last Will and Testament (LWT) feature is used in MQTT to notify other clients about an ungracefully disconnected client. Each client can specify its last will message (a normal MQTT message with topic, retained flag, QoS and payload) when connecting to a broker. The broker will store the message until it detects that the client has disconnected ungracefully. If the client disconnects abruptly, the broker sends the message to all subscribed clients on the topic, which was specified in the last will message. The stored LWT message will be discarded if a client disconnects gracefully by sending a *DISCONNECT* message.



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

How to specify a LWT message for a client?

The LWT message can be specified by each client as part of the CONNECT message, which serves as connection initiation between client and broker.



More details about establishing a connection between client and broker can be found in [part 3](#) of the MQTT Essentials series.

When will a broker send the LWT message?

According to the [MQTT 3.1.1 specification](#) the broker will distribute the LWT of a client in the following cases:

- An I/O error or network failure is detected by the server.
- The client fails to communicate within the Keep Alive time.
- The client closes the network connection without sending a DISCONNECT packet first.

- The server 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 ideal for notifying other interested clients about the connection loss. In real world scenarios LWT is often used together with [retained messages](#), in order to store the state of a client on a specific topic. For example after a client has connected to a broker, it will send a retained message to the topic *client1/status* with the payload “*online*“. When connecting to the broker, the client sets the LWT message on the same topic to the payload “*offline*” and marks this LWT message as a retained message. If the client now disconnects ungracefully, the broker will publish the retained message with the content “*offline*“. This pattern allows for other clients to observe the status of the client on a single topic and due to the retained message even newly connected client now immediately the current status.

So that’s the end of part nine in our MQTT Essentials series. We hope you enjoyed it. In the next and last post we’ll cover the [MQTT heartbeat mechanism and how the broker knows a client is online or offline](#).

MQTT Essentials Part 10: Keep Alive and Client Take-Over



Welcome to the tenth part of the MQTT Essentials, a blog series about the core features and concepts in the MQTT protocol. In this post we will cover the **Keep Alive** feature of MQTT and why it is especially important for mobile networks.

Problem of half-open TCP connections

As we already know [MQTT is based on TCP](#) and that includes a certain guarantee that packets over the internet are transferred “[reliable, ordered and error-checked](#)”. Nevertheless it can happen that one of the communicating parties gets out of sync with the other, often due to a crash of one side or because of transmission errors. This state is called a [half-open connection](#). The important point is that the still functioning end is not notified about the failure of the other side and is still trying to send messages and wait for acknowledgements.

The problems with half-open connection increase in mobile networks as the following citation from Andy Stanford-Clark, inventor of the MQTT protocol, explains:

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?*“ ([Source](#))

MQTT Keep Alive

In order to work around this issue of half-open connection or at least give a possibility to access if the connection is still open, MQTT provides the *keep alive* functionality.

The keep alive functionality assures that the connection is still open and both broker and client are connected to one another. Therefore the client specifies a time interval in seconds and

communicates it to the broker during the establishment of the connection. The interval is the longest possible period of time, which broker and client can endure without sending a message.

The MQTT specification says the following:

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.

That means 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 ensure that the connection is still open.

But if the client doesn't send any messages during the period of the keep alive it must send a PINGREQ packet to the broker to confirm its availability and also make sure the broker is still available.

The broker must disconnect a client, which doesn't send PINGREQ or any other message in one and a half times of the keep alive interval. Likewise should the client close the connection if the response from the broker isn't received in a reasonable amount of time.

Keep Alive Flow

Let's have a look at the keep alive messages in detail. There are two messages involved in the keep alive functionality.

PINGREQ



The PINGREQ is sent by the client and indicates to the broker that the client is still alive, even if it hasn't send any other packets (PUBLISH, SUBSCRIBE, etc..). The client can send a PINGREQ at any time to make sure the network connection is still alive. The PINGREQ packet doesn't have any payload.

PINGRESP



When receiving a PINGREQ the broker must reply with a PINGRESP packet to indicate its availability to the client. Similar to the PINGREQ the packet doesn't contain any payload.

Good to Know

- If the broker doesn't receive a PINGREQ or any other packet from a particular client, it will close the connection and send out the [last will and testament message](#) (if the client had specified one).
- The MQTT client is responsible of setting the right keep alive value. For example, it can adapt the interval to its current signal strength.
- The maximum keep alive is 18h 12min 15 sec.
- If the keep alive interval is set to 0, the keep alive mechanism is deactivated.

Client Take-Over

A disconnected client will most likely try to connect again. It could be the case that the broker still has a half-open connection for the same client. In this scenario the MQTT will perform a so-called client take-over. **The broker will close the previous connection to the same client (determined by the same client identifier) and establishes the connection with the newly connected client.** This behavior makes sure that half-open connection won't stand in the way of a new connection establishment of the same client.

So that's the end of part ten in our MQTT Essentials series. We hope you enjoyed the whole series. This was the last official post, but we have planned a MQTT Essential Special for next week, which will be about MQTT over Websockets. And we have already a lot of great ideas for topics we will cover in the future, so stay tuned for more helpful content about MQTT and HiveMQ