A Brief Overview of MQTT V1.1

Last Modified on: 31/03/2017

Compiled by
Abhay S Bharadwaj (Device Design Engineer)
For Frugal Labs Tech Solutions Pvt. Ltd.

Disclaimer

1. Introduction

MQTT is a Client-Server publish/subscribe messaging 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.

It is a 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.

2. History

MQTT was invented by Andy Stanford-Clark (IBM) and Arlen Nipper (Eurotech) back in 1998-99, when their use case was to create a protocol for minimal battery loss and minimal bandwidth connecting oil pipelines over satellite connection. It got published as "MQTT in August 2010 and was taken up by the eclipse foundation under the PAHO project in November 2011. Later, in September 2014, MQTT became a standard and is officially an ISO (ISO/IEC 20922:2016) and OASIS standard. The timeline of MQTT is as follows:

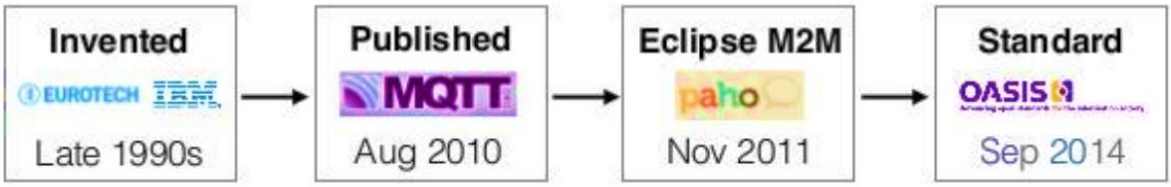

Figure 1: MQTT Timeline

3. The MQTT Paradigm

There are two main components in a MQTT communication system: a MQTT BROKER and a MQTT CLIENT. The data communication is always between the MQTT Broker and a MQTT Client and never between two clients. The Broker is a piece of software installed on the cloud or server or gateway that is accessible to all the clients in that network. Clients can be a piece of program, software, physical devices, etc.

The "data" or MQTT Message is always sent / received over MQTT "Topics". Topics are unique "addresses" for message delivery.

4. The publish/subscribe model

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.

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. (Works on Asynchronous communication model)

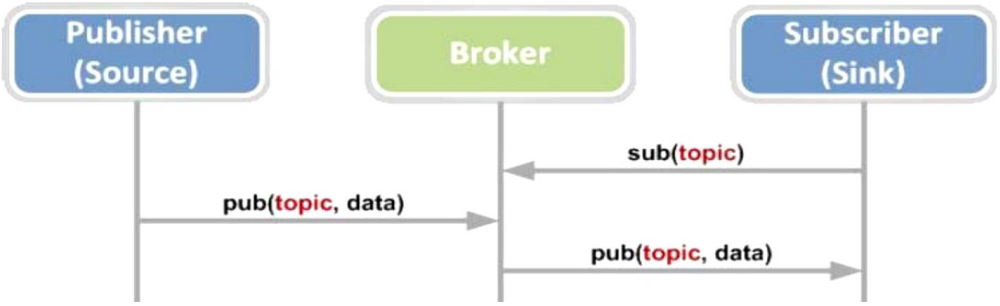

Figure 2: MQTT Pub-Sub model

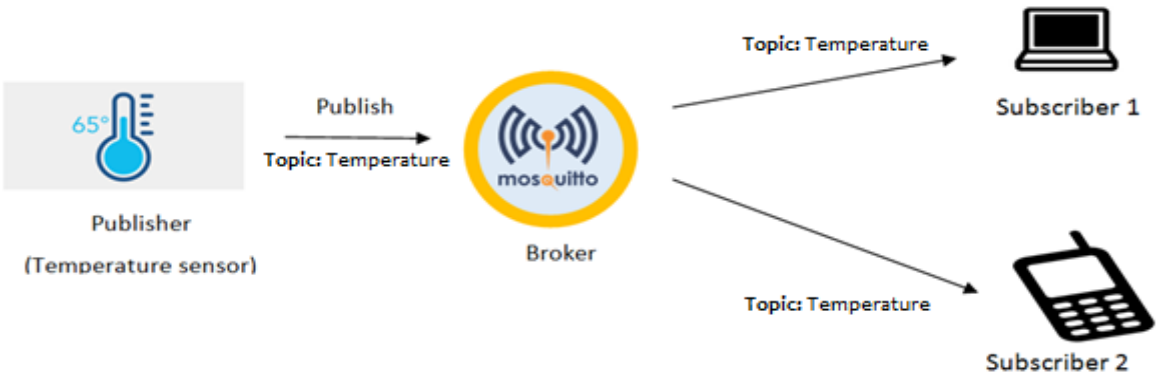

Figure 3: MQTT Illustration

1. Two "clients": a Web app and a mobile app connect to the MQTT Broker and subscribe to the MQTT Topic: "Temperature".
2. Another "client": a Temperature sensor connects to the MQTT Broker and Publishes "Temperature in *C" data to the topic: "Temperature".
3. The "Broker" forwards this data to all the "Clients" who have subscribed to the topic: "Temperature"

5. Message Filtering

The three ways of message filtering are:

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 a hierarchical structure that allow filtering based on a limited number of expressions.

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 cannot be encrypted or changed easily.

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.

6. Client

A “Client” can be a publisher or subscriber; both of them are called an MQTT client. In general a MQTT client can be both a publisher & subscriber at the same time. A MQTT client can be anything; from a micro-controller to a piece of software 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.

7. 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. 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.

8. MQTT Connection

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

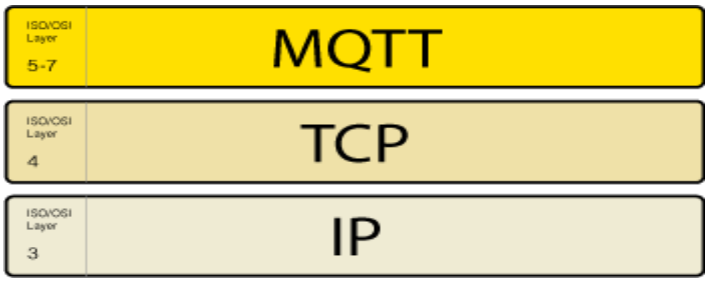

Figure 4: MQTT Protocol Stack

With reference to the ISO-OSI Model, layer-3 is taken up by IP, layer-4 by TCP and layers-5,6,7 by MQTT.

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 responds 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 looses the connection.

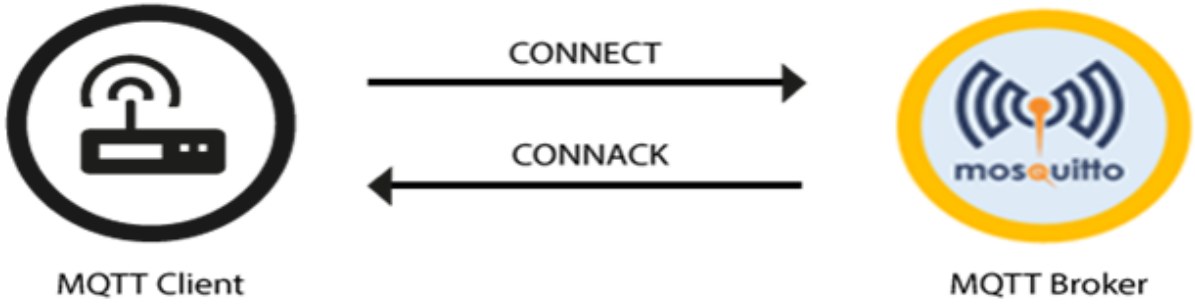

Figure 4a: MQTT Connection Initiation

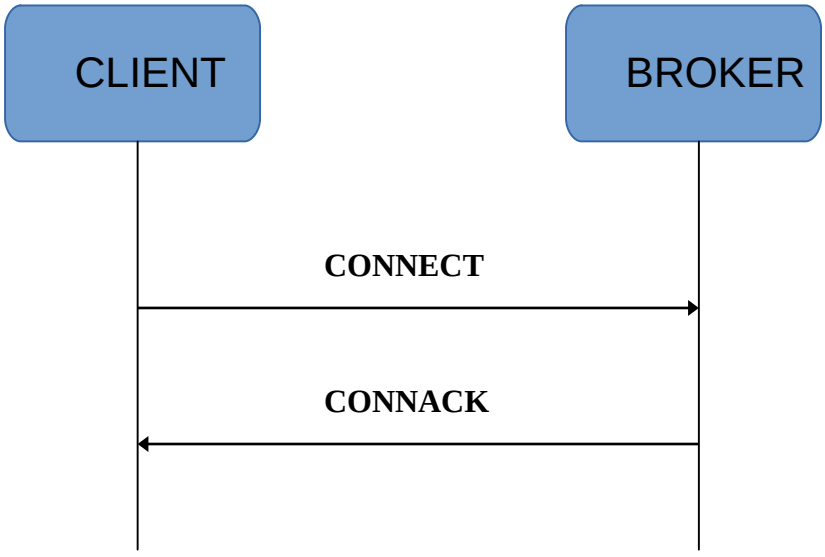

Figure 4b: MQTT connection flow diagram

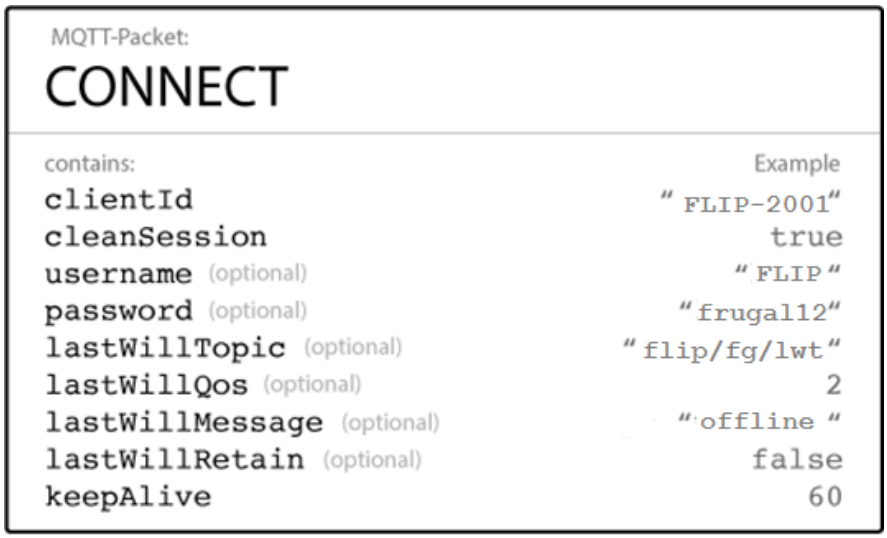

Figure 4c: MQTT CONNECT Message packet

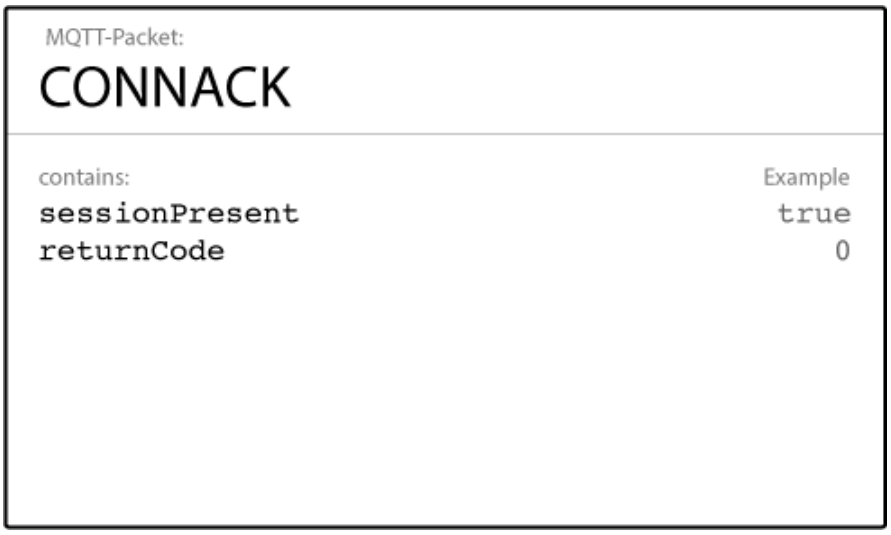

Figure 4d: MQTT CONNACK message Packet

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.

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

Username/Password

MQTT allows sending 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 using username and password together with a secure transport on it.

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.

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.

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.

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.

9. 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). All topics are case sensitive and must have at least one character to be valid.

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:

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 forward slash

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 $

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.

Best Practices:

Don't use leaving forward slash. Eg: /house/light
Don't use spaces in a topic.
Keep topic short and concise.
Embed unique identifier into topic.

10. 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

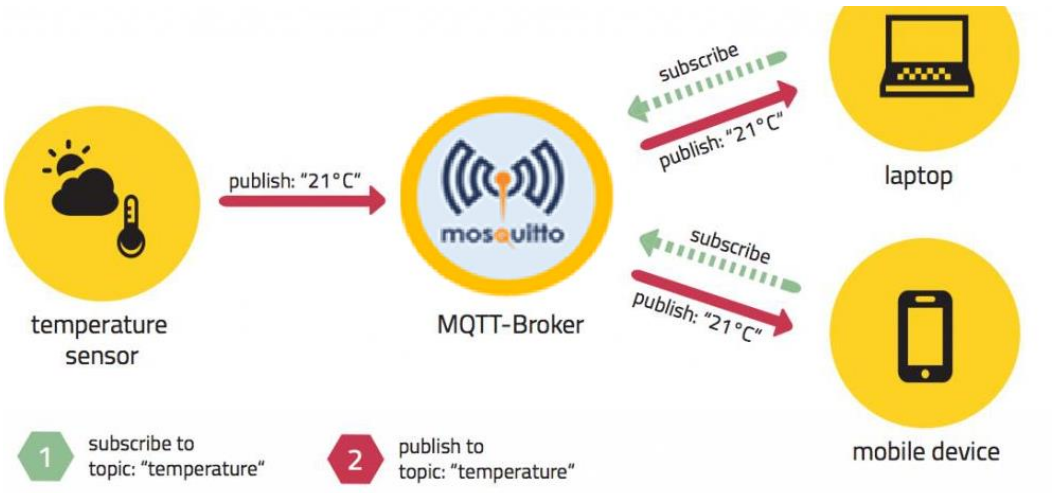

Figure 5a: MQTT Publish

Here, the "temperature Sensor" (MQTT Client) is publishing the temperature data to the MQTT Broker. The broker is then publishing the data to the Clients who have subscribed to it.

Figure 5b: MQTT Publish packet

Topic Name

A simple string, which is hierarchically structured with forward slashes as delimiters. Some example would be: “home/room1/tempC” or “abcd” or “farm1/field3/moisture” or “Germany/Munich/Octoberfest/people”.

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 info on QoS discussed later.

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.

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.

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.

11. Subscribe

A client needs to send a SUBSCRIBE message to the MQTT broker in order to receive relevant messages. “SUBSCRIPTION” is done at the time of MQTT Connection. If a client that is already connected to the MQTT Broker has to subscribe to more new topics, the client has to disconnect and re-connect with new subscriptions.

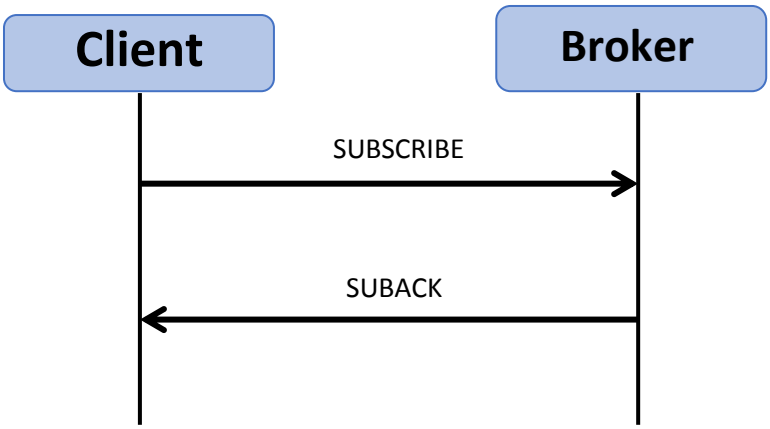

Figure 6a: MQTT “Subscribe” Flow Diagram

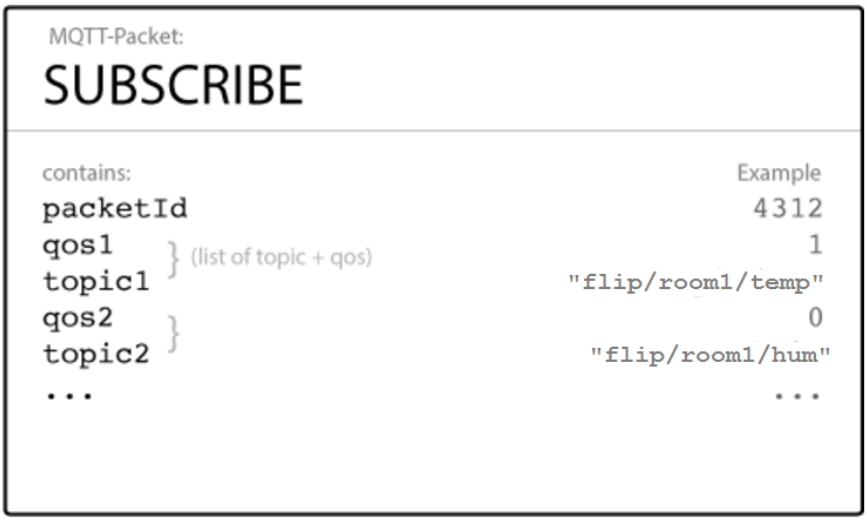

Figure 6b: MQTT Subscribe packet

Packet Identifier

The packet identifier is a unique identifier between client and broker to identify a message in a message flow.

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.

Suback

Each subscription will be confirmed by the broker through sending an acknowledgment to the client in form of the SUBACK message.

Figure 6c: MQTT SUBACK packet

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, 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.

12. Unsubscribe

The counterpart of the SUBSCRIBE message is the UNSUBSCRIBE message which deletes existing subscriptions of a client on the broker. Whenever a client wants to stop receiving messages from the BROKER, it unsubscribes itself from that topic.

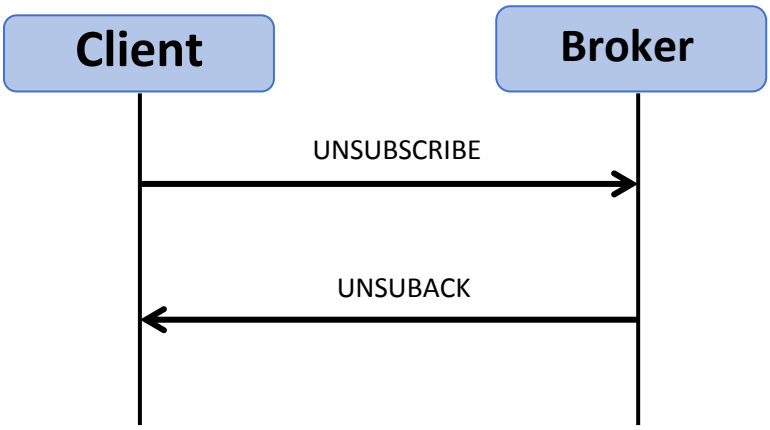

Figure 7a: MQTT “Unsubscribe” Flow Diagram

Figure 7b: MQTT "Unsubscribe" Packet

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.

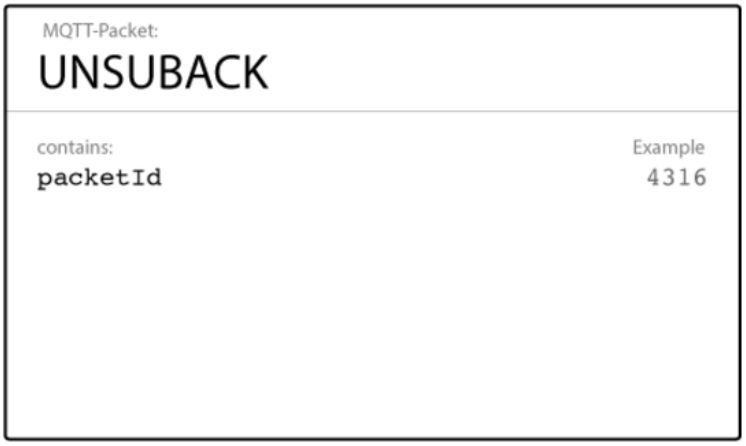

Figure 7c: MQTT "UNSUBACK" Packet

The complete working flow is illustrated below. The sequence of events are as follows:

1. CONNECT
2. CONNACK
3. SUBSCRIBE
4. SUBACK
5. UNSUBSCRIBE
6. UNSUBACK
7. DISCONNECT

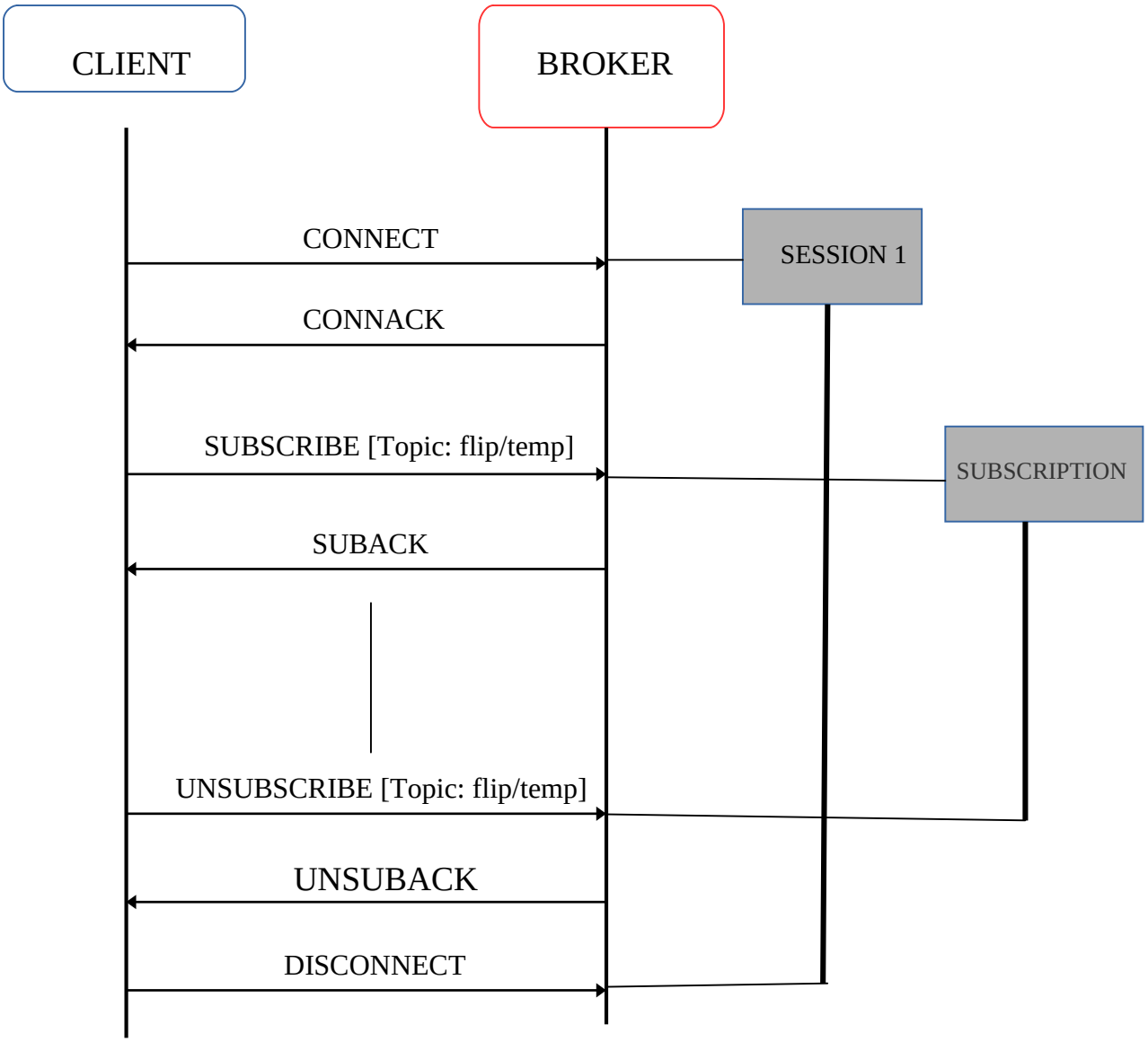

Figure 8: MQTT work flow

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

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.

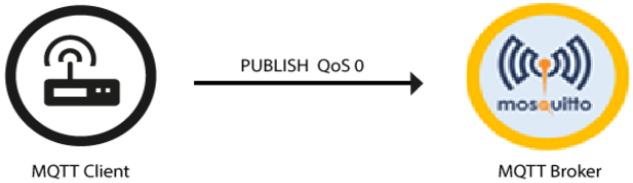

Figure 9a: Publish with QoS 0 Illustration

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.

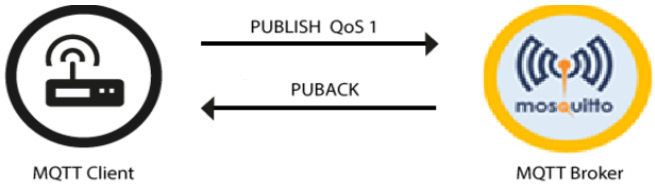

Figure 9b: Publish with QoS 1 Illustration

QoS 2 – exactly once

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 (Publish Received) message.

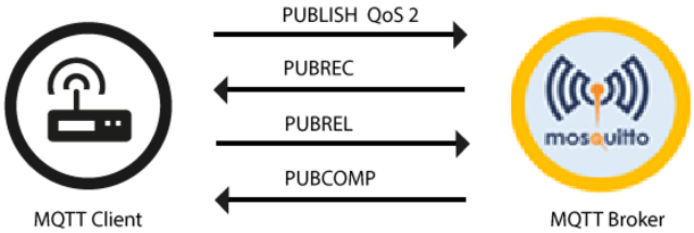

Figure 9c: Publish with QoS 2 Illustration

The receiver will store a reference to the packet identifier until it has sent 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 counterpart has successfully received the message. It will store the PUBREC and respond with a PUBREL (Publish Release). After the receiver gets the PUBREL it can discard every stored state and answer with a PUBCOMP (PublishComplete). 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.

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 fast 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.

14. 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 behaviour 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

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 offlne
- 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. 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.

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.

15. 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. 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.

Send a retained message

Sending a retained message from the perspective of a developer is quite simple and straight-forward. You just need to set the **retained flag** of a MQTT publish message to true.

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.

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.

16. 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 disconnect 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.

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.

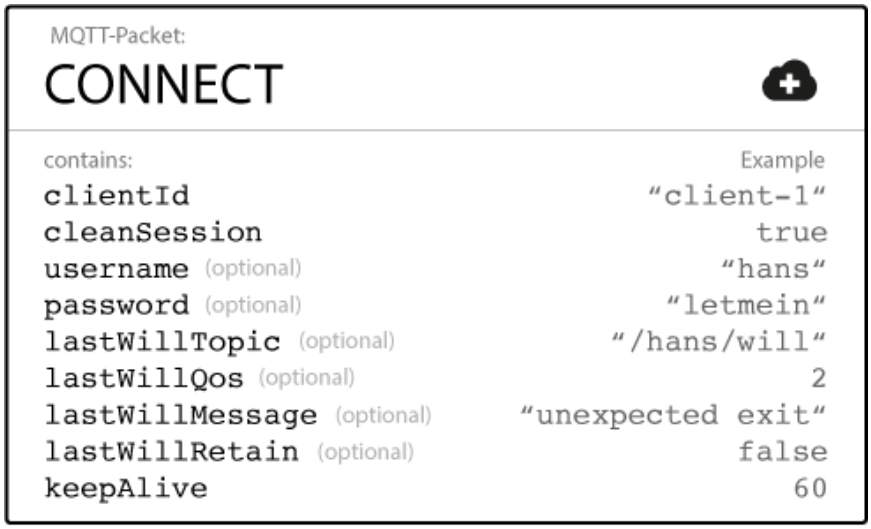

Figure 10: LWT Initilization

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.

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.

17. MQTT over WebSockets

We have learned that MQTT is ideal for constrained devices and unreliable networks. It's also perfect for sending messages with a very low overhead. It would be quite nice to send and receive MQTT messages directly in the browser of a mobile phone or in general. This is possible by MQTT over WebSockets. It enables the browser to leverage all MQTT features and this can be used for the following example use case:

- Display live information from a device or sensor
- Receive push notifications, for example if there is an alert or critical condition
- See the current status of devices with LWT and retained messages
- Communicate efficiently with a mobile web application

How does it work?

WebSocket is a network protocol that provides bi-directional communication between a browser and a web server. It was standardized in 2011 and all modern browsers have support for WebSockets built-in. Similar to MQTT, WebSockets are based on TCP.

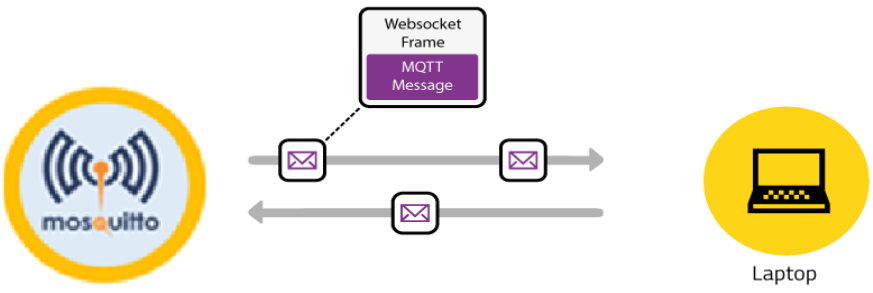

Figure 11: MQTT over web-sockets illustration

When using the term **MQTT *over* WebSockets**, it means that a MQTT message, for example CONNECT or PUBLISH is encapsulated by one or more WebSocket frames, when transferring over the network. WebSockets are suitable as transport for MQTT because the communication is bi-directional, ordered and lossless (which is essentially because WebSockets also leverage TCP). In order to communicate with an MQTT broker over WebSockets, the broker must be able to handle native WebSockets.

18. Security features of MQTT

Why is security essential for the Internet of Things?

In our digital and global world security is prominent every day, it doesn't matter if you are making a bank transfer, buying stuff online or access personal documents over the internet. **The idea of the Internet of Things is to connect every object in order to make process more efficient, provide more comfort or improve our work and personal life in any kind of way**. But connecting objects like cars, homes, machines also exposes lots of sensitive data. For example the location of all people in an household. Maybe it is good to know what your family members are up to, but it is not ideal to share these information with a burglar. There are different kinds of data, which are not meant for the public and should be protected by the pillars of information security: **confidentiality, integrity and availability**.

Approach to security in MQTT

Security in MQTT is divided in multiple layers. Each layer prevents different kind of attacks. The goal of the protocol is to provide a really lightweight and easy to use communication protocol for the internet of things. So that's why in the protocol itself are only a few security mechanisms clearly specified. But in all common implementations other state-of-the-art security standards are used, like SSL/TLS for transport security. The idea behind is that security is hard and there is no good in embedding non-standard security mechanisms and instead build upon generally accepted standards. So in the following the different levels will only be covered briefly, we will have designated posts for each of them in the series. This should only serve as high level overview and big picture.

Network Level

Using a physically secure network or VPN as foundation for any communication between clients and broker is one way to provide a secure and trustworthy connection. This would be suitable for gateway applications, where the gateway is connected to devices on the one hand and with the broker over VPN on the other side.

Transport Level

When the goal is to provide confidentiality in most cases TLS/SSL is being used for transport encryption. It provides a secure and proven way to make sure nobody can read along and even authenticate both sides, when using client certification authentication. We will also cover in detail the feasibility of TLS on constrained devices.

Application Level

On the transport level it can be ensured that the communication is encrypted and the identity is authenticated. The MQTT protocol provides a client identifier and username/password credentials, which can also be used to authenticate devices on the application level. These properties are provided by the protocol itself. When it comes to

authorization or what each device is allowed to do, it lays in the hand of the broker implementation, how to handle it. Another possibility is to use payload encryption on the application level in order to make the transmitted information secure even without having a full fledged transport encryption.

Authentication with Username and Password

Authentication is part of the transport and application level security in MQTT. On the transport level TLS can guarantee authentication of the client to the server using client certificates and of the server to the client validating the server certificate. On the application level the MQTT protocol provides username and password for authentication.

"Authentication is the act of confirming the truth of an attribute of a single piece of data or entity."

Authentication is a process we use everyday without even noticing. Every time you log into your computer you provide a username and a password. The username is your identity and the entry of the password authenticates you as the rightful owner.

MQTT authentication with username/password

When it comes to authentication in MQTT the protocol itself provides username and password fields in the CONNECT message. Therefore a client has the possibility to send a username and password when connecting to an MQTT broker.

Figure 12a: MQTT Connect Packet

The username is a UTF-8 encoded string and the password is binary data with each 65535 bytes max. The rather short non-normative recommendation of 12 characters for a password in the MQTT 3.1 specification was removed in the new 3.1.1 version released last year. The spec also states that a username without password is possible. It's not possible to just send a password without username.

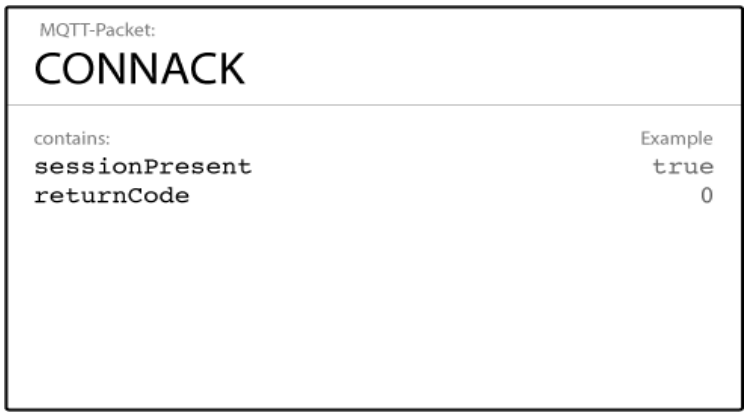

Figure 12b: MQTT CONNACK Packet

When using the built-in username/password authentication, the MQTT broker will evaluate the credential based on the implemented authentication mechanism (and return one of the following return codes.

Return Code	Return Code Response
0	Connection Accepted
4	Connection Refused, bad user name or password
5	Connection Refused, not authorized

When setting username and password on the client, it will be sent to the broker in clear text. This would allow eavesdropping by an attacker and is an easy way of obtaining the credentials. The only way to guarantee a completely secure transmission of username and password is to use transport encryption.

Authorization in MQTT

In MQTT, Authorization is done by the BROKER. A MQTT client can basically do two things after it has connected to a broker, it can publish messages and it can subscribe to topics. So translating this to the previously stated definitions:

- A MQTT client is the subject, it wants authorization to do something
- The main resources or objects available to a client are the topics
- Other objects would be: Store Last Will and Testament or have a persistent session
- The main resources which need protection are the ability to publish or subscribe

Without proper authorization each authenticated client can publish and subscribe to all kinds of topics. This could be desirable in a closed system. For most use cases, fine-grained restrictions make a lot of sense and should be used. The official MQTT 3.1.1 specification states the following on this matter:

"MQTT solutions are often deployed in hostile communication environments. In such cases, implementations will often need to provide mechanisms for: […] Authorization of access to Server resources"

In order to restrict a client to publish or subscribe only to topics it is authorized to, it is necessary to implement topic permissions on the broker side. These permissions need to be configurable and adjustable during the runtime of the broker. A topic permission could for example look like the following:

- Allowed topic (exact topic or wild card topic)
- Allowed operation (publish, subscribe, both)
- Allowed quality of service level (0, 1, 2, all)

This kind of topic permission would allow the broker to specify authorization policies for clients and to limit their ability to subscribe and publish messages. An example would be giving a client the permission to subscribe only to a single topic and use only a certain quality of service level.

Deny

After having defined authorization policies, a very common question is how to notify a client that it doesn't have the permission to publish or subscribe on a certain topic.

Publish

When publishing to a topic the client has no permission for, the broker has two options:

- It can disconnect the client, because publishing to a restricted topic is disallowed.
- It can acknowledge the publish to the client in a normal fashion – in case of QoS 1 or 2 with PUBACK or PUBREL – and decide not to send the published message to the subscribers.

Subscribe

In the case of subscribing to a topic, the broker needs to acknowledge each subscription with a return code. There are 4 different codes for acknowledging each topic with a granted QoS or sending an error code. So if the client has no right to subscribe a specific topic, the broker can notify the client that the subscription was denied.

Best practices

A commonly used best practice is to include the client id of the publishing client in the permission. So the client is restricted to only publishing to topics, where it's client id is upfront. An example would be client123/temperature or client123/#. The same can be used for subscribing. This is a good pattern for topics that are only concern one client. Of course this are often not the only permission. Additionally a client commonly has permissions to subscribe to more general topics like: clients/status or clients/command. This depends highly on the use case and should only be a suggestion.

SSL/TLS and MQTT

What is SSL and TLS?

TLS (Transport Layer Security) and SSL (Secure Sockets Layer) provide a secure communication channel between a client and a server. At its core, TLS and SSL are cryptographic protocols which use a handshake mechanism to negotiate various parameters to create a secure connection between the client and the server. After the handshake is completed, an encrypted communication between client and server is established and no attacker can eavesdrop any part of the communication. Servers provide a X509 certificate, typically issued by a trusted authority, which clients use to verify the identity of the server.

Why is TLS important?

Imagine you are sending a postcard. It's clear who the recipient of the card is and the postman will make sure that the card arrives. However, nothing prevents the postman to read the contents of the card, in fact everyone, who is involved delivering the postcard, can read the contents. If you have a malicious postman, he can even alter some contents of the card!

The idea of this illustration is also true for computer networks in general and the Internet in particular. If you are using plain TCP/IP, it's like sending a postcard. The TCP packet is going to pass a lot of infrastructure components (routers, firewalls, Internet Exchange Points, etc.) before reaching the target. Every participant in between could now read the contents of your packet in clear text or even modify it. This is not a fictional scenario; recent history showed that Internet traffic is wiretapped all the time by intelligence agencies. While most attackers don't have that many resources to eavesdrop your connection, it's not too hard for sophisticated attackers to perform Man-In-The-Middle-Attacks. So TLS is all about providing a secure communication channel. With TLS it's safe to say that the communication contents cannot be read or altered by third parties*.

**Assuming secure cipher suites were used and there are no yet unknown attacks on the TLS version used.*

MQTT and TLS

MQTT relies on TCP as transport protocol, which means by default the connection does not use an encrypted communication. To encrypt the whole MQTT communication, most many MQTT brokers – like Mosquitto – allow to use TLS instead of plain TCP. If you are using the username and password fields of the MQTT CONNECT packet for authentication and authorization mechanisms, you should strongly consider using TLS.

Port 8883 is standardized for a secured MQTT connection. The standardized name at IANA is "secure-mqtt" and port 8883 is exclusively reserved for MQTT over TLS.