Internet of Things (IoT) - MQTT

By Abhinav Jain, Intern, Special Projects

What is MQTT?

- MQTT stands for MQ Telemetry Transport. It is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol.
- It was designed as an extremely lightweight publish/subscribe messaging transport.
- It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium.
- For example, it has been used in sensors communicating to a broker via satellite link, over occasional dial-up connections with healthcare providers, and in a range of home automation and small device scenarios.
- Was invented by Dr. Andy Stanford-Clark of IBM, and Arlen Nipper of Arcom (now Eurotech), in 1999.

Is MQTT a standard?

- MQTT v3.1.1 has now become an OASIS Standard.
- The protocol specification has been openly published with a royalty-free license for many years, and companies such as Eurotech (formerly known as Arcom) have implemented the protocol in their products.
- In November 2011 IBM and Eurotech announced their joint participation in the Eclipse M2M Industry Working Group and donation of MQTT code to the proposed Eclipse Paho project.

Core Elements of MQTT - 1

- **Clients:**
  - Clients subscribe to topics to publish and receive messages. Thus subscriber and publisher are special roles of a client.
  - 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.
  - MQTT client libraries are available for a huge variety of programming languages, for example Android, Arduino, C, C++, C#, Go, iOS, Java, JavaScript, .NET
- **Servers (Brokers)**
  - Servers run topics, i.e. receive subscriptions from clients on topics, receive messages from clients and forward these, based on client’s subscriptions, to interested clients.
  - Another responsibility of the broker is the authentication and authorization of clients. (While using encrypted connection)

Core Elements of MQTT - 2

- **Topics:**
  - Technically, topics are message queues. Topics support the publish/subscribe pattern for clients. Logically, topics allow clients to exchange information with defined semantics.
- **Sessions:**
  - A session identifies a (possibly temporary) attachment of a client to a server. All communication between client and server takes place as part of a session
- **Subscriptions:**
  - Unlike sessions, a subscription logically attaches a client to a topic. When subscribed to a topic, a client can exchange messages with a topic.

Publisher/Subscriber Model - 1

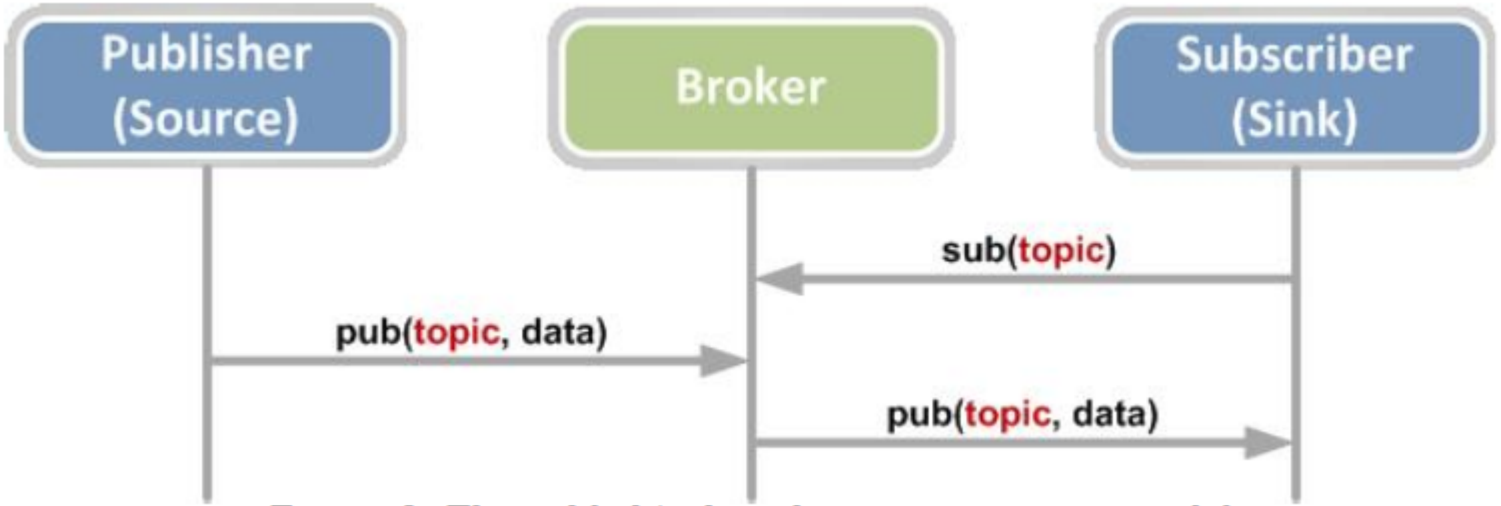


Figure 1: The publish/subscribe communication model

Publisher/Subscriber Model - 2

- 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

Publisher/Subscriber Model - 3

- 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.
- 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.
- 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.

Implementation in MQTT

- 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.

- 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).

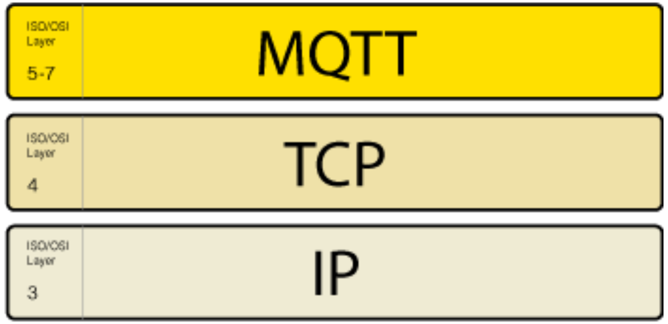
- MQTT is especially easy to use on the client-side. Most pub/sub systems have the most logic on the broker-side, but MQTT is really the essence of pub/sub when using a client library and that makes it a light-weight protocol for small and constrained devices.

MQTT vs Message Queues

- **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.

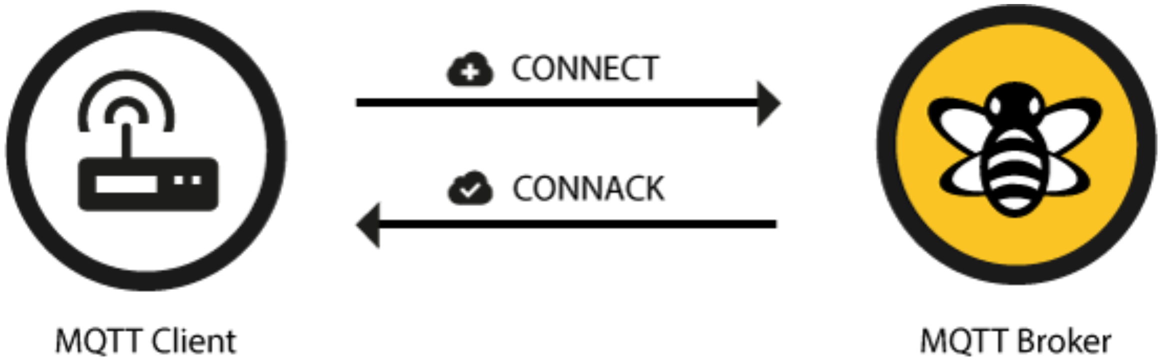
MQTT Connection - 1

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



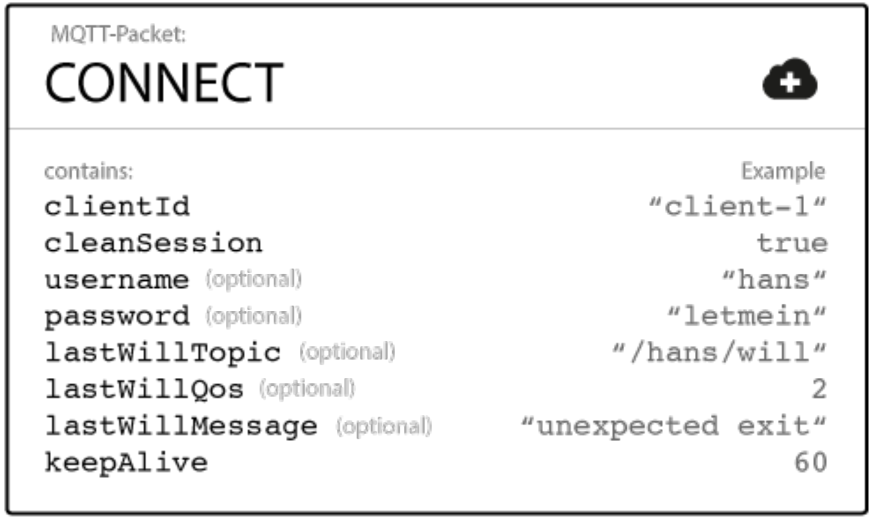
MQTT Connection - 2

- 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 initiated MQTT will keep it open as long as the client doesn’t send a disconnect command or it looses the connection.



MQTT Connection - 3

- 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 specifications) 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"
keepAlive	60

CONNECT message parameters - 1

- **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.

CONNECT message parameters - 2

- **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.

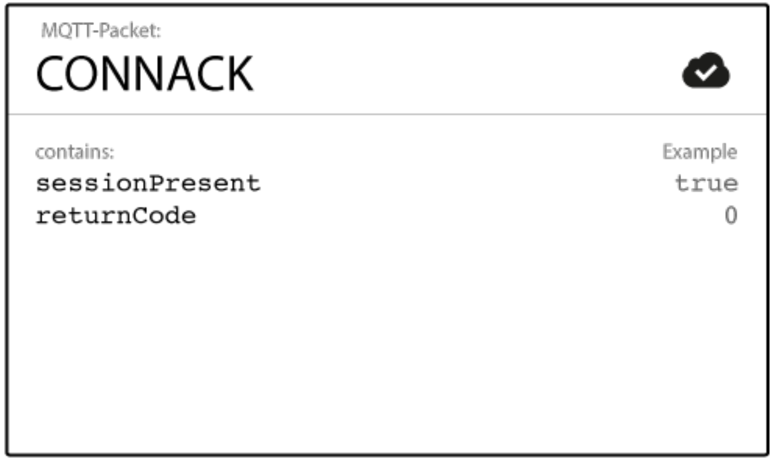
- **KeepAlive:** The keep alive is a time interval, the clients commits to by sending regular PING Request messages to the broker. The broker response with PING Response and this mechanism will allow both sides to determine if the other one is still alive and reachable. We'll talk about this in detail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

CONNACK message

- **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.

- **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.

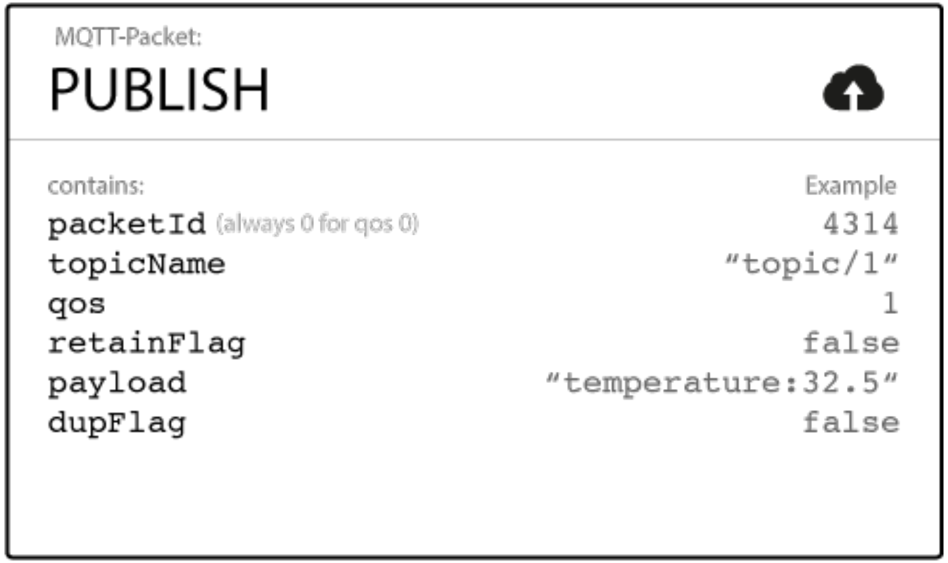


MQTT-Packet:
CONNACK

contains:	Example
sessionPresent	true
returnCode	0

PUBLISH message

- After a MQTT client is connected to a broker, it can publish messages.
- **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.



MQTT-Packet:

PUBLISH

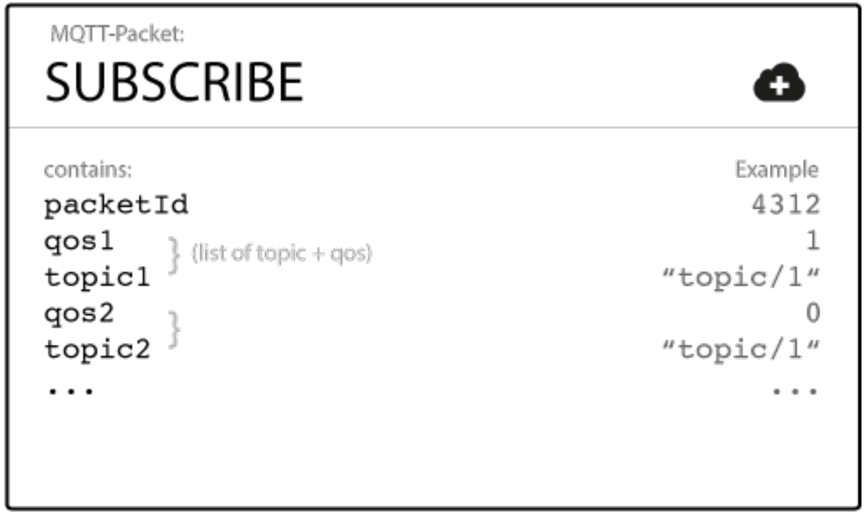
contains:	Example
packetId (always 0 for qos 0)	4314
topicName	“topic/1”
qos	1
retainFlag	false
payload	“temperature:32.5”
dupFlag	false

PUBLISH message parameters

- **Topic Name:** A simple string, which is hierarchically structured with forward slashes as delimiters. An example would be “myhome/livingroom/temperature”.
- **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).
- **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. 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. This resend/duplicate mechanism is typically handled by the MQTT client library or the broker as an implementation detail.

SUBSCRIBE message

- 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.


MQTT-Packet:
SUBSCRIBE
(list of topic + qos)

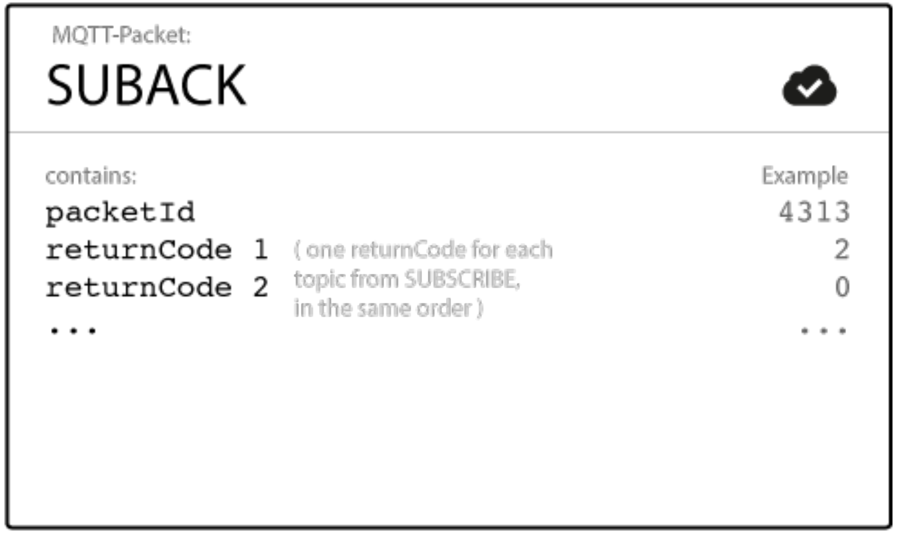
contains:	Example
packetId	4312
qos1	1
topic1	“topic/1“
qos2	0
topic2	“topic/1“
...	...

SUBSCRIBE message parameters

- **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.
- **List of Subscriptions:** A SUBSCRIBE message can contain an arbitrary number of subscriptions for a client. Each subscription is a pair of a topic 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 message

- 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

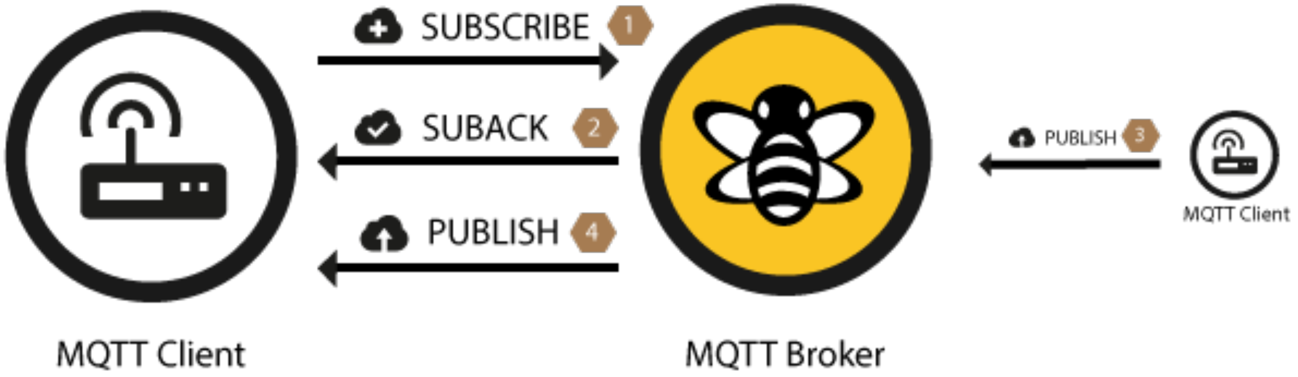
contains:		Example
packetId		4313
returnCode 1	(one returnCode for each topic from SUBSCRIBE, in the same order)	2
returnCode 2		0
...		...

SUBACK message parameters

- **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.

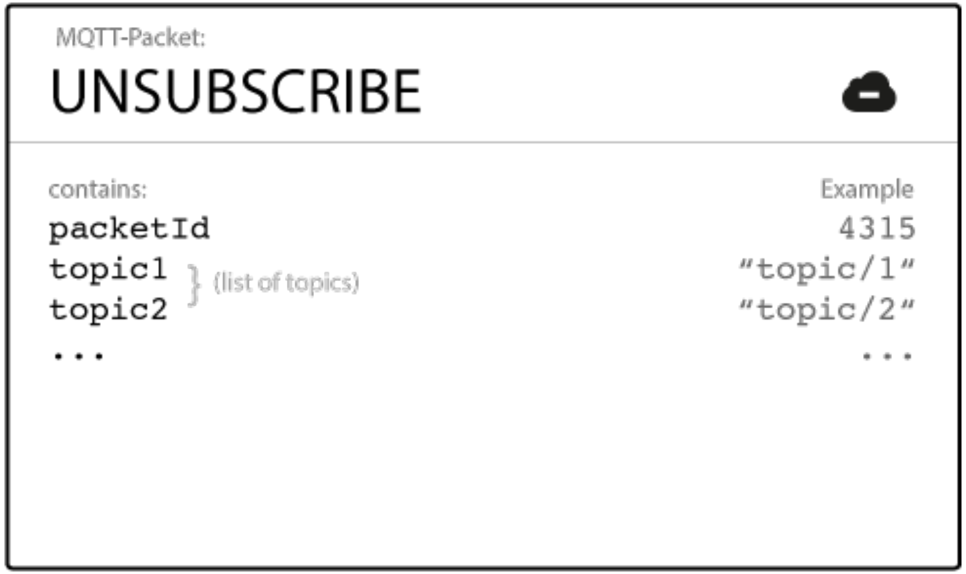
Return Code	Return Code Response
0	Success – Maximum QoS 0
1	Success – Maximum QoS 1
2	Success – Maximum QoS 2
128	Failure

Communication



UNSUBSCRIBE message

- 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.


MQTT-Packet:

UNSUBSCRIBE

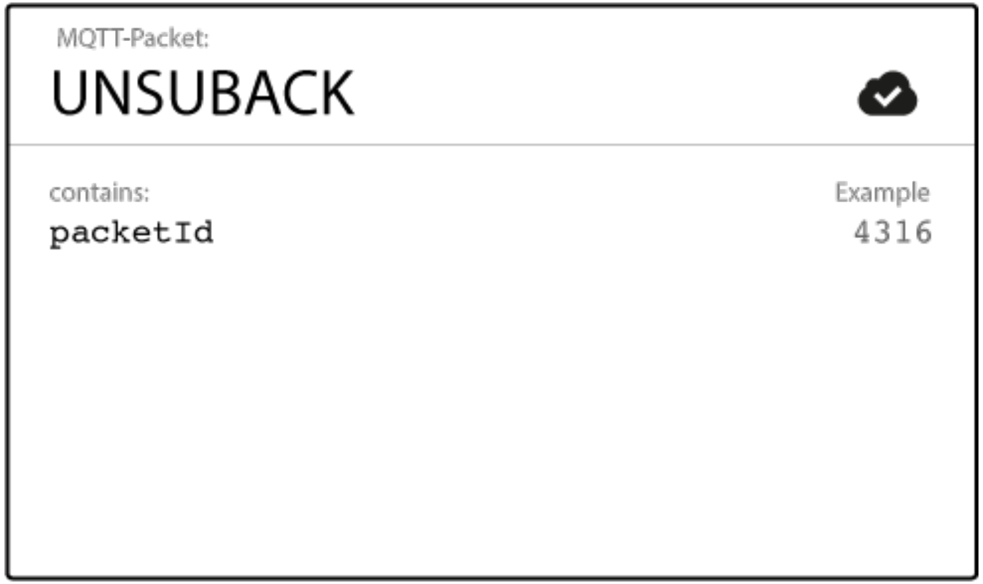
contains:		Example
packetId		4315
topic1	(list of topics)	“topic/1“
topic2		“topic/2“
...		...

UNSUBSCRIBE message parameters

- **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 message

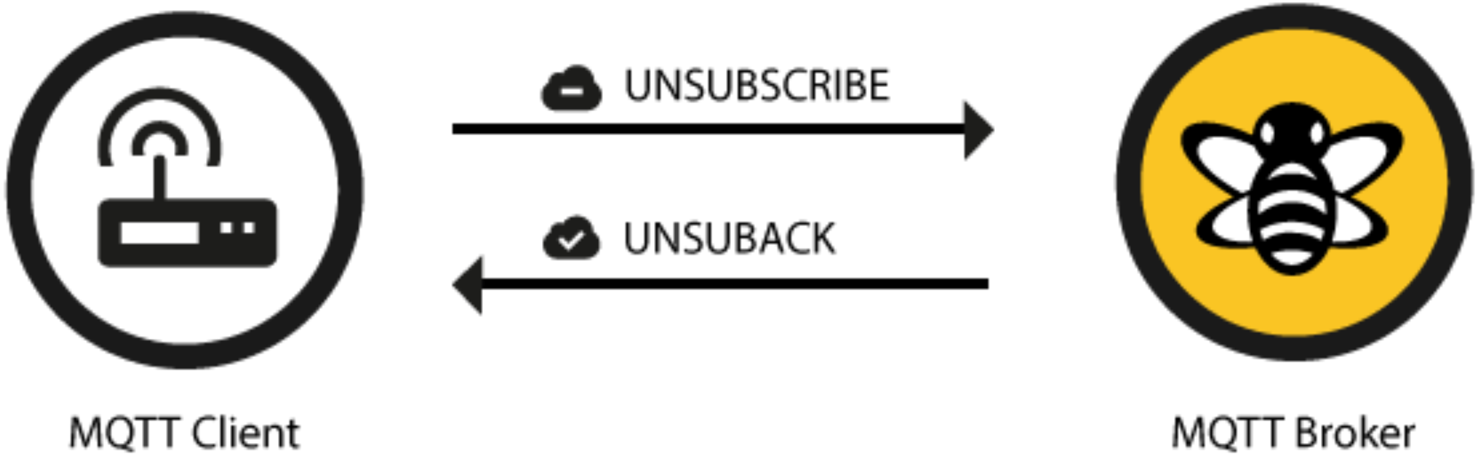
- The broker will acknowledge the request to unsubscribe with the UNSUBACK message. This message only contains a packet identifier.


MQTT-Packet:
UNSUBACK

contains:	Example
packetId	4316

UNSUBACK message parameters

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



Quality of Service (QoS)

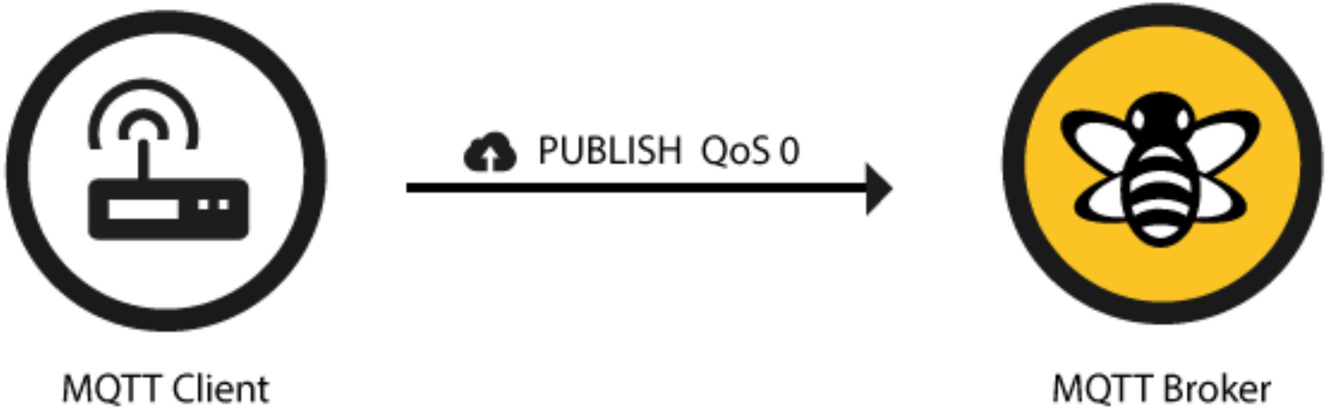
- The Quality of Service (*QoS*) level is an agreement between sender and receiver of a message regarding the guarantees of delivering a message. There are 3 QoS levels in MQTT:
  - ***At most once* (0)**
  - ***At least once* (1)**
  - ***Exactly once* (2).**
- When talking about QoS there are always 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.

Importance of QoS

- 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.

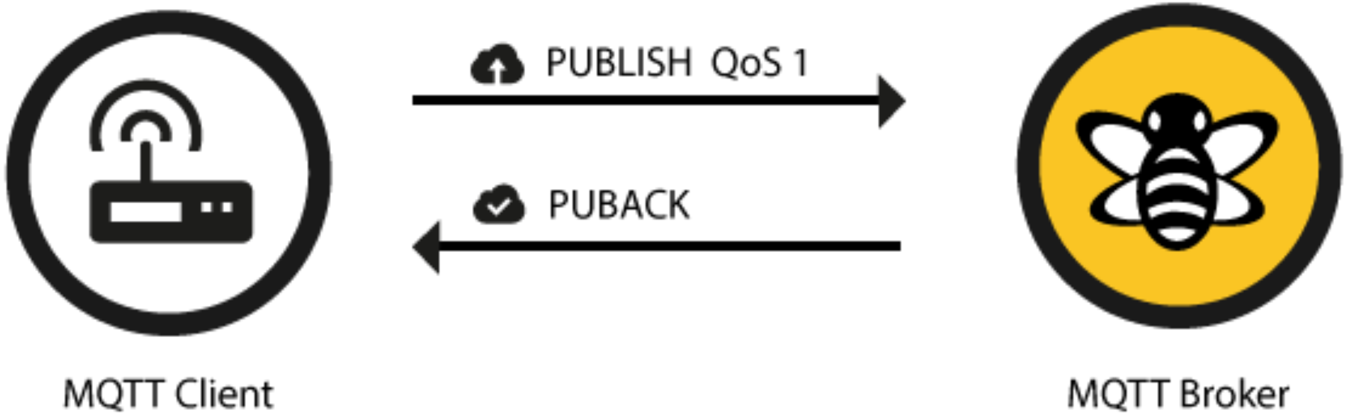
QoS 0 - at most once

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



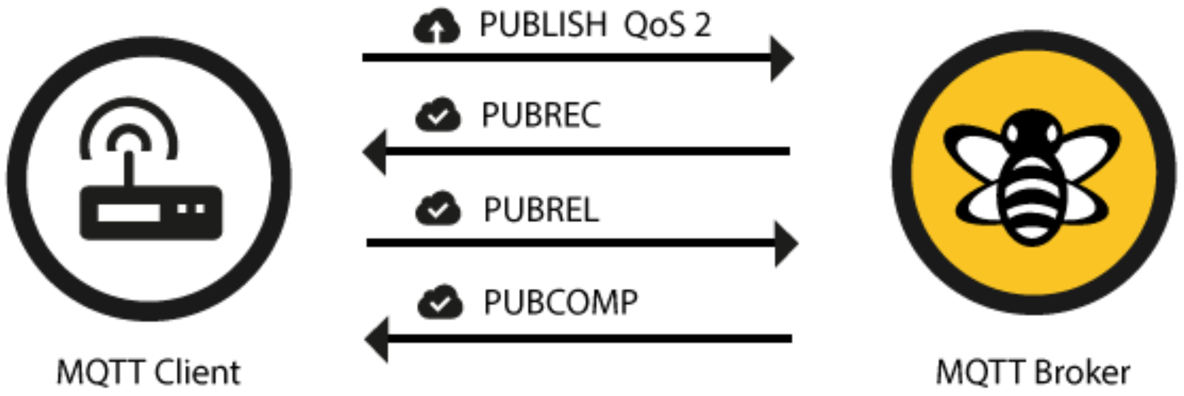
QoS 1 - at least once

- 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.



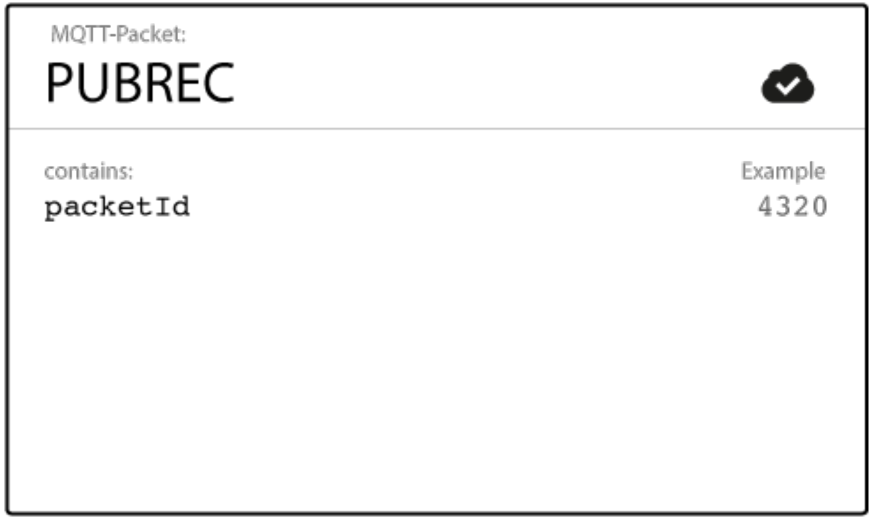
QoS 2 - exactly once

- 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.



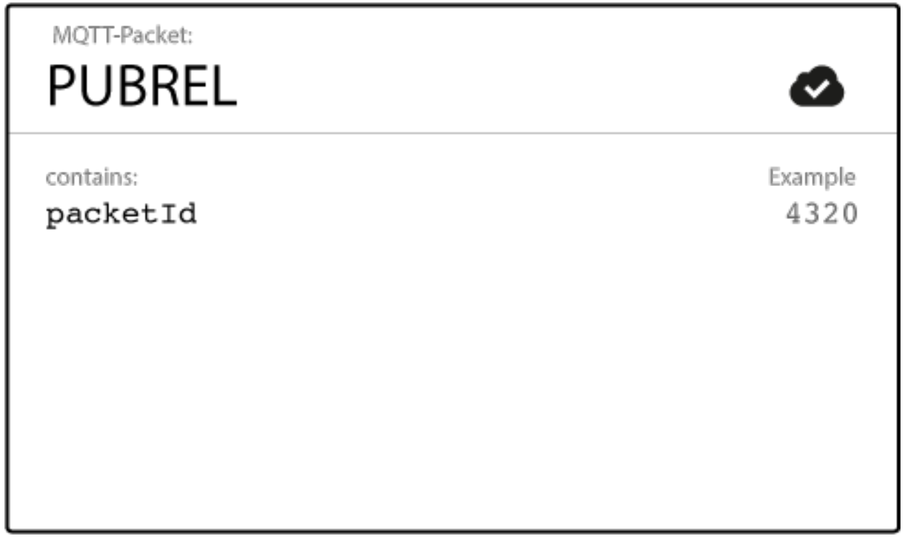
QoS 2 (Contd..)

- 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.



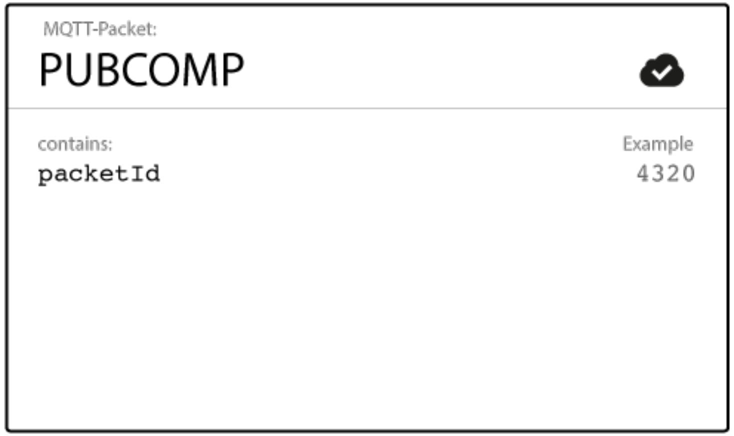
QoS 2 (Contd..)

- 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**.



QoS 2 (Contd..)

- 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.


MQTT-Packet:

PUBCOMP

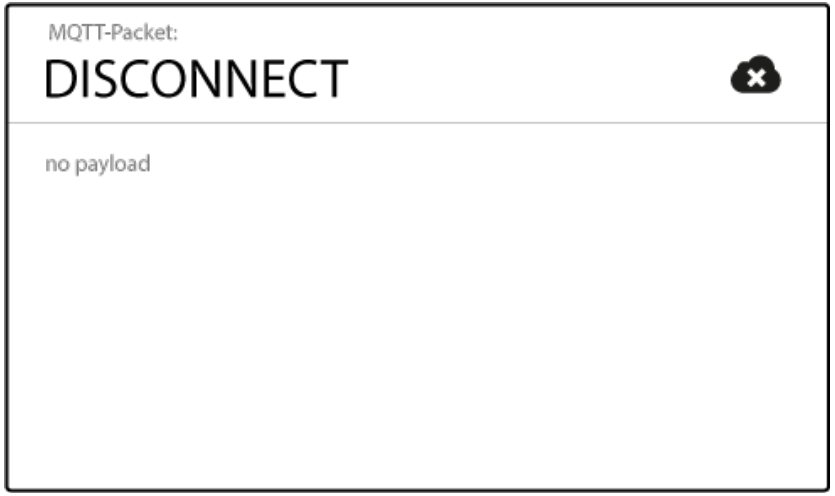
contains:	Example
packetId	4320

Last Will and Testament

- MQTT is often used in scenarios were 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.

LWT (Contd..)

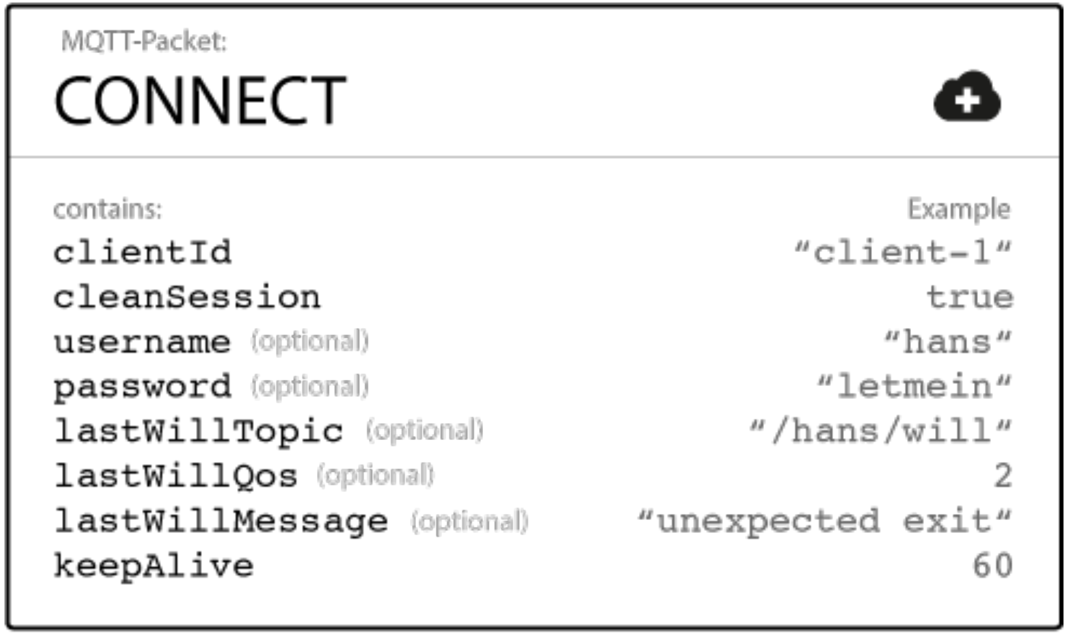
- Each client can specify its last will message (a normal MQTT message with topic, retained flag, QoS and payload) when connecting to a broker.
- The broker will store the message until it detects that the client has disconnected ungracefully.
- If the client disconnect abruptly, the broker sends the 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.


MQTT-Packet:
DISCONNECT

no payload

Specifying a LWT message

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



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“
keepAlive	60

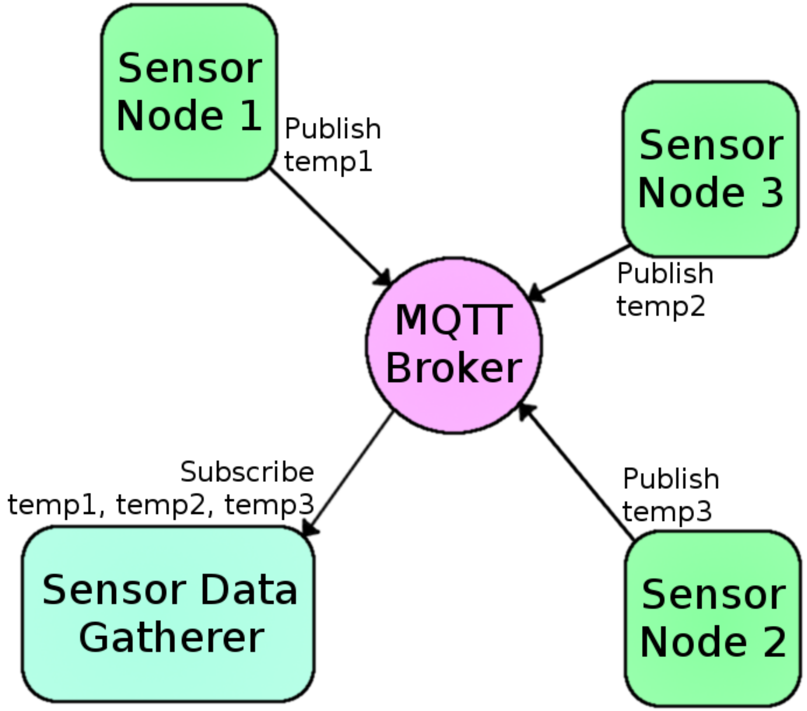
When is LWT message sent?

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.

Use cases

- Useful for Home Automation/Wireless Sensor Networks as depicted in this image.
- You can get readings from various sensors and devices around and gather them centrally at one location.



Use cases (Contd..)

- Facebook uses parts of MQTT protocol in its ‘Messenger App’ for fast delivery of messages.

Use cases (Contd..)

- **FloodNet:**

  (monitoring river levels and environmental information to provide early warning of flooding)

  “The FloodNet project centres upon the development of providing a pervasive, continuous, embedded monitoring presence. By processing and synthesizing collected information over a river and functional floodplain, FloodNet obtains an environmental self-awareness and resilience to ensure robust transmission of data in adverse conditions and environments.”

THE END

Thank You!