MQTT

28 June 2020 19:18

© RAJAT KUMAR
https://www.linkedin.com/in/imRajat/
https://github.com/im-Rajat

MQTT or Message Queuing Telemetry Transport is an open OASIS and ISO standard lightweight, publish-subscribe network protocol that transports messages between devices.

The protocol usually runs over TCP/IP; however, any network protocol that provides ordered, lossless, bi-directional connections can support MQTT.

It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited.

The MQTT protocol defines two types of network entities:
- A message broker and a number of clients.
- An MQTT broker is a server that receives all messages from the clients and then routes the messages to the appropriate destination clients.
- An MQTT client is any device (from a micro controller up to a full-fledged server) that runs an MQTT library and connects to an MQTT broker over a network.

MQTT relies on the TCP protocol for data transmission.

The broker acts as a post office, MQTT doesn't use the address of the intended recipient but uses the subject line called "Topic", and anyone who wants a copy of that message will subscribe to that topic. Multiple clients can receive the message from a single broker (one to many capability). Similarly, multiple publishers can publish topics to a single subscriber (many to one).

Message types :-
- Connect : Waits for a connection to be established with the server and creates a link between the nodes.
- Disconnect : Waits for the MQTT client to finish any work it must do, and for the TCP/IP session to disconnect.
- Publish : Returns immediately to the application thread after passing the request to the MQTT client.

```
mosquitto_sub -h 3.17.133.23 -p 1883 -t "test" -u "username" -P "password"

mosquitto_pub -h localhost -t "test" -m "Message" -u "username" -P
"password"

nano publish.c
gcc -o pub publish.c -lpaho-mqtt3c
./pub
```

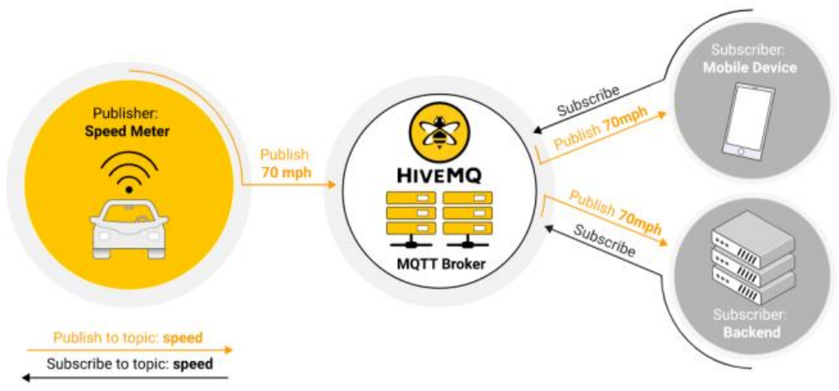
Following : https://www.hivemq.com/mqtt-essentials/
It includes core of MQTT concepts, its features and other essential information.

1) Introducing the MQTT Protocol - MQTT Essentials: Part 1

- "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."

2) Publish & Subscribe - MQTT Essentials: Part 2

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



The most important aspect of pub/sub is the decoupling of the publisher of the message from the recipient (subscriber). This decoupling has several dimensions:

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

In summary,

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

Scalability : Pub/Sub scales better than the traditional client-server approach. This is because operations on the broker can be highly parallelized and messages can be processed in an event-driven way.

Message filtering:-

1) SUBJECT-BASED FILTERING : This filtering is based on the subject or topic that is part of each message. The receiving client subscribes to the broker for topics of interest. From that point on, the broker ensures that the receiving client gets all message published to the subscribed topics.
Both publisher and subscriber need to know which topics to use. Another thing to keep in mind is message delivery.
2) CONTENT-BASED FILTERING : In content-based filtering, the broker filters the message based on a specific content filter-language. The receiving clients subscribe to filter queries of messages for which they are interested. A significant downside to this method is that the content of the message must be known beforehand and cannot be encrypted or easily changed.
3) TYPE-BASED FILTERING : When object-oriented languages are used, filtering based on the type/class of a message (event) is a common practice. For example,, a subscriber can listen to all messages, which are of type Exception or any sub-type.

MQTT:

- MQTT decouples the publisher and subscriber spatially.
  - To publish or receive messages, publishers and subscribers only need to know the hostname/IP and port of the broker.
- MQTT decouples by time.
  - Although most MQTT use cases deliver messages in near-real time, if desired, the broker can store messages for clients that are not online.
- MQTT works asynchronously.
  - Because most client libraries work asynchronously and are based on callbacks or a similar model, tasks are not blocked while waiting for a message or publishing a message.

3) MQTT Client and Broker and MQTT Server and Connection Establishment Explained - MQTT Essentials: Part 3

Client:

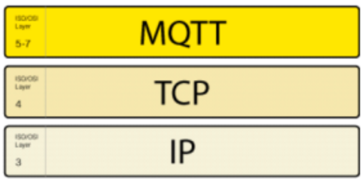
- When we talk about a client, we almost always mean an MQTT client. Both publishers and subscribers are MQTT clients.
- The publisher and subscriber labels refer to whether the client is currently publishing messages or subscribed to receive messages (publish and subscribe functionality can also be implemented in the same MQTT client).
- **An MQTT client is any device (from a micro controller up to a full-fledged server) that runs an MQTT library and connects to an MQTT broker over a network.**
- Basically, any device that speaks MQTT over a TCP/IP stack can be called an MQTT client.

Broker:

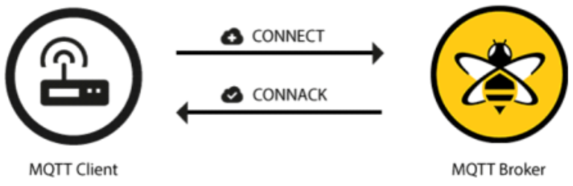
- The counterpart of the MQTT client is the MQTT broker.
- The broker is at the heart of any publish/subscribe protocol. Depending on the implementation, a broker can handle up to millions of concurrently connected MQTT clients.
- **The broker is responsible for receiving all messages, filtering the messages, determining who is subscribed to each message, and sending the message to these subscribed clients.**
- it is important that your broker is highly scalable, integratable into backend systems, easy to monitor, and (of course) failure-resistant.

MQTT Connection:

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



- The MQTT connection is always between one client and the broker.
- Clients never connect to each other directly.
- **To initiate a connection, the client sends a CONNECT message to the broker. The broker responds with a CONNACK message and a status code.**

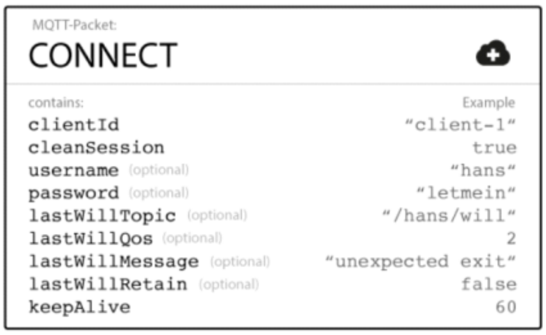


- Once the connection is established, the broker keeps it open until the client sends a disconnect command or the connection breaks.

Client initiates connection with the CONNECT message : A good-natured MQTT 3 client sends a connect message with the following content :

1. **ClientId :** The client identifier (ClientId) identifies each MQTT client that connects to an MQTT broker.
2. **Clean Session :** The clean session flag tells the broker whether the client wants to establish a persistent session or not.
3. **Username/Password :** MQTT can send a user name and password for client authentication and authorization.
4. **Will Message :** The last will message is part of the Last Will and Testament (LWT) feature of MQTT. This message notifies other clients when a client disconnects ungracefully.
5. **KeepAlive** : The keep alive is a time interval in seconds that the client specifies and communicates to the broker when the connection established.

6.



MQTT-Packet:

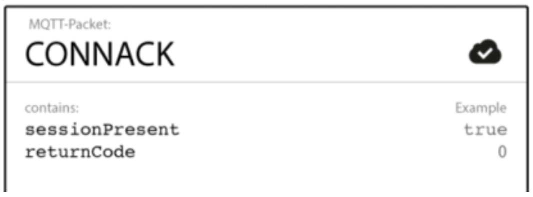
CONNECT

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

Broker response with a CONNACK message : -
When a broker receives a CONNECT message, it is obligated to respond with a CONNACK message.

The CONNACK message contains two data entries:

1. **The session present flag** : The session present flag tells the client whether the broker already has a persistent session available from previous interactions with the client.
2. **A connect return code** :The second flag in the CONNACK message is the connect acknowledge flag. This flag contains a return code that tells the client whether the connection attempt was successful or not.



MQTT-Packet:

CONNACK

contains:	Example
sessionPresent	true
returnCode	0

Here are the return codes at a glance:-

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

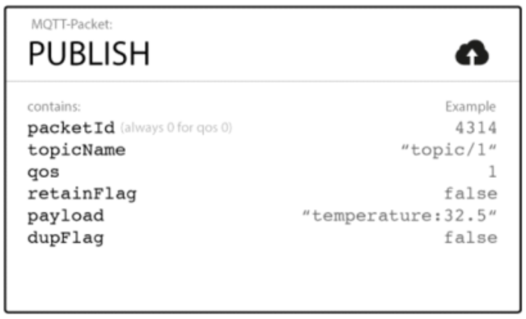
MQTT Publish, Subscribe & Unsubscribe - MQTT Essentials: Part 4

Publish:-

- An MQTT client can publish messages as soon as it connects to a broker.
- MQTT utilizes topic-based filtering of the messages on the broker.
- Each message must contain a topic that the broker can use to forward the message to interested clients. Typically, each message has a payload which contains the data to transmit in byte format. MQTT is data-agnostic.

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

1. **Topic Name** : The topic name is a simple string that is hierarchically structured with forward slashes as delimiters. For example, "myhome/livingroom/temperature" or "Germany/Munich/Octoberfest/people"
2. **QoS** : This number indicates the Quality of Service Level (QoS) of the message. There are three levels: 0, 1, and 2. The service level determines what kind of guarantee a message has for reaching the intended recipient (client or broker).
3. **Retain Flag** : This flag defines whether the message is saved by the broker as the last known good value for a specified topic. When a new client subscribes to a topic, they receive the last message that is retained on that topic.
4. **Payload** : This is the actual content of the message. MQTT is data-agnostic. It is possible to send images, text in any encoding, encrypted data, and virtually every data in binary.
5. **Packet Identifier** : The packet identifier uniquely identifies a message as it flows between the client and broker. The packet identifier is only relevant for QoS levels greater than zero. The client library and/or the broker is responsible for setting this internal MQTT identifier.
6. **DUP flag** : The flag indicates that the message is a duplicate and was resent because the intended recipient (client or broker) did not acknowledge the original message. This is only relevant for QoS greater than 0.

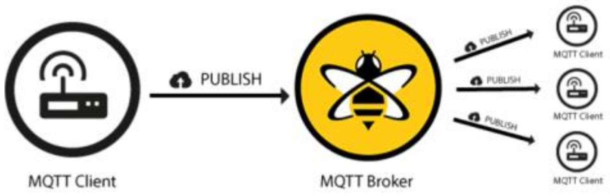


MQTT-Packet:

PUBLISH

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

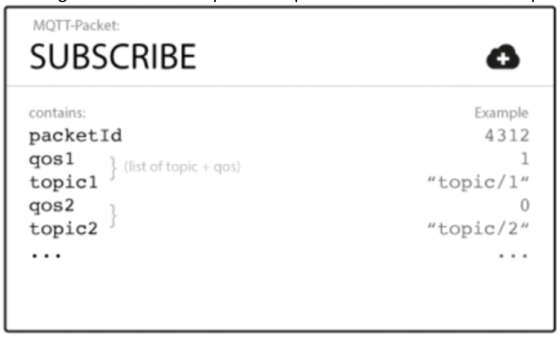
When a client sends a message to an **MQTT broker** for publication, **the broker reads the message, acknowledges the message (according to the QoS Level), and processes the message.** Processing by the broker includes determining which clients have subscribed to the topic and sending the message to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



Subscribe:-

- To receive messages on topics of interest, the client sends a SUBSCRIBE message to the MQTT broker.
- This subscribe message is very simple, it contains a unique packet identifier and a list of subscriptions.

1. **Packet Identifier** : The packet identifier uniquely identifies a message as it flows between the client and broker. The client library and/or the broker is responsible for setting this internal MQTT identifier.
2. **List of Subscriptions** : A SUBSCRIBE message can contain multiple subscriptions for a client. Each subscription is made up of a topic and a QoS level.

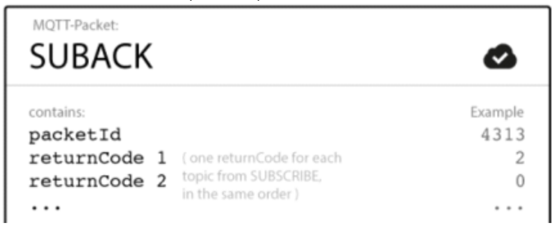

MQTT-Packet:
SUBSCRIBE

contains:		Example
packetId		4312
qos1	(list of topic + qos)	1
topic1		"topic/1"
qos2		0
topic2		"topic/2"
...		...

Suback:-

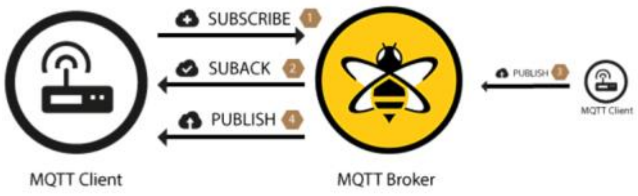
- To confirm each subscription, the broker sends a SUBACK acknowledgement message to the client.
- This message contains the packet identifier of the original Subscribe message (to clearly identify the message) and a list of return codes.

1. **Packet Identifier** : The packet identifier is a unique identifier used to identify a message. It is the same as in the SUBSCRIBE message.
2. **Return Code** : The broker sends one return code for each topic/QoS-pair that it receives in the SUBSCRIBE message.


MQTT-Packet:
SUBACK

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

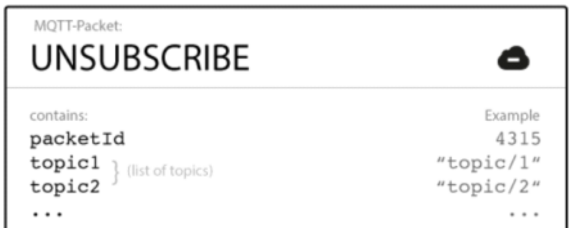
Return Code	Return Code Response
0	Success - Maximum QoS 0
1	Success - Maximum QoS 1
2	Success - Maximum QoS 2
128	Failure



Unsubscribe:-

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

1. **Packet Identifier** : The packet identifier uniquely identifies a message as it flows between the client and broker. The client library and/or the broker is responsible for setting this internal MQTT identifier.
2. **List of Topic** : The list of topics can contain multiple topics from which the client wants to unsubscribe. It is only necessary to send the topic (without QoS). The broker unsubscribes the topic, regardless of the QoS level with which it was originally subscribed.

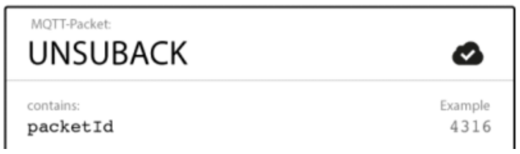

MQTT-Packet:
UNSUBSCRIBE

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

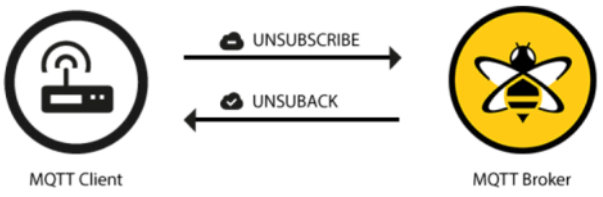
Unsuback:-

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

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

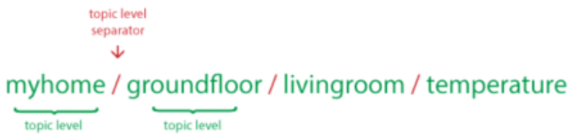


After receiving the UNSUBACK from the broker, the client can assume that the subscriptions in the UNSUBSCRIBE message are deleted.

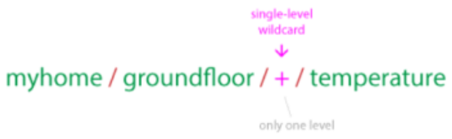


MQTT Topics & Best Practices - MQTT Essentials: Part 5

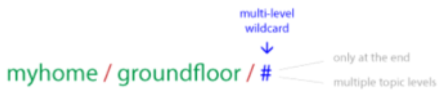
Topics:-
- In MQTT, the word topic refers to an UTF-8 string that the broker uses to filter messages for each connected client.
- The topic consists of one or more topic levels.
- Each topic level is separated by a forward slash (topic level separator).



Wildcards:-
- When a client subscribes to a topic, it can subscribe to the exact topic of a published message or it can use wildcards to subscribe to multiple topics simultaneously.
- A wildcard can only be used to subscribe to topics, not to publish a message.
- There are two different kinds of wildcards: single-level and multi-level.
  1. **Single Level: +**
     - As the name suggests, a single-level wildcard replaces one topic level. The plus symbol represents a single-level wildcard in a topic.



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



Topics beginning with $:-
- Generally, you can name your MQTT topics as you wish.
- However, there is one exception: **Topics that start with a $ symbol have a different purpose.**
- These topics are not part of the subscription when you subscribe to the multi-level wildcard as a topic (#).
- The **$-symbol topics are reserved for internal statistics of the MQTT broker.**

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

Quality of Service:-
- The Quality of Service (QoS) level is an agreement between the sender of a message and the receiver of a message that defines the guarantee of delivery for a specific message.
- There are 3 QoS levels in MQTT:
  - At most once (0)
  - At least once (1)
  - Exactly once (2).
- When you talk about QoS in MQTT, you need to consider the two sides of message delivery:
  - Message delivery form the publishing client to the broker.
  - Message delivery from the broker to the subscribing client.

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

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

QoS 1 - at least once:

- QoS level 1 guarantees that a message is delivered at least one time to the receiver.
- The sender stores the message until it gets a PUBACK packet from the receiver that acknowledges receipt of the message.
- It is possible for a message to be sent or delivered multiple times.
- The sender uses the packet identifier in each packet to match the PUBLISH packet to the corresponding PUBACK packet.
- If the sender does not receive a PUBACK packet in a reasonable amount of time, the sender resends the PUBLISH packet

QoS 2 - exactly once:

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

Queuing of QoS 1 and 2 messages:

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

Persistent Session and Queuing Messages - MQTT Essentials: Part 7

Persistent Session:

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

What's stored in a persistent session?

- In a persistent session, the broker stores the following information (even if the client is offline). When the client reconnects the information is available immediately.
  - Existence of a session (even if there are no subscriptions).
  - All the subscriptions of the client.
  - All messages in a Quality of Service (QoS) 1 or 2 flow that the client has not yet confirmed.
  - All new QoS 1 or 2 messages that the client missed while offline.
  - All QoS 2 messages received from the client that are not yet completely acknowledged.

Retained Messages - MQTT Essentials: Part 8

Retained Messages:

- A retained message is a normal MQTT message with the retained flag set to true.
- The broker stores the last retained message and the corresponding QoS for that topic.
- Each client that subscribes to a topic pattern that matches the topic of the retained message receives the retained message immediately after they subscribe.
- The broker stores only one retained message per topic.
- Retained messages help newly-subscribed clients get a status update immediately after they subscribe to a topic.
- The retained message eliminates the wait for the publishing clients to send the next update.

Send a retained message:

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

Delete a retained message:

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

Why and when should you use Retained Messages?

- A retained message makes sense when you want newly-connected subscribers to receive messages immediately (without waiting until a publishing client sends the next message).
- Without retained messages, new subscribers are kept in the dark between publish intervals.
- Using retained messages helps provide the last good value to a connecting client immediately.

Last Will and Testament - MQTT Essentials: Part 9

Last Will and Testament:

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

How do you specify a LWT message for a client?

- Clients can specify an LWT message in the CONNECT message that initiates the connection between the client and the broker.

When does a broker send the LWT message?

- According to the MQTT 3.1.1 specification, the broker must distribute the LWT of a client in the following situations:
  - The broker detects an I/O error or network failure.
  - The client fails to communicate within the defined Keep Alive period.
  - The client does not send a DISCONNECT packet before it closes the network connection.

        - The broker closes the network connection because of a protocol error.

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

The problem of half-open TCP connections:

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

MQTT Keep Alive:

- MQTT includes a keep alive function that provides a workaround for the issue of half-open connections (or at least makes it possible to assess if the connection is still open).
- Keep alive ensures that the connection between the broker and client is still open and that the broker and the client are aware of being connected.
- When the client establishes a connection to the broker, the client communicates a time interval in seconds to the broker.
- This interval defines the maximum length of time that the broker and client may not communicate with each other.
- "**The Keep Alive** ... is the maximum time interval that is permitted to elapse between the point at which the Client finishes transmitting one Control Packet and the point it starts sending the next. It is the responsibility of the Client to ensure that the interval between Control Packets being sent does not exceed the Keep Alive value. In the absence of sending any other Control Packets, the Client MUST send a PINGREQ Packet."
- As long as messages are exchanged frequently and the keep-alive interval is not exceeded, there is no need to send an extra message to establish whether the connection is still open.
- The broker must disconnect a client that does not send a message or a PINGREQ packet in one and a half times the keep alive interval.

PINGREQ:

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

PINGRESP:

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

Client Take-Over:

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