MQTT

MQTT is a Client Server publish/subscribe messaging transport protocol. It is light weight, binary protocol, 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.“

# The publish/subscribe pattern

In the client-sever model, a client communicates directly with an endpoint.

The publishers and subscribers never contact each other directly.  **The connection between them is handled by a third component (the broker)**.

* **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.
* MQTT decouples the publisher and subscriber spatially. To publish or receive messages, publishers and subscribers only need to know the hostname/IP and port of the broker.
* MQTT decouples by time. Although most MQTT use cases deliver messages in near-real time, if desired, the broker can store messages for clients that are not online. (Two conditions must be met to store messages: the client had connected with a persistent session and subscribed to a topic with a [**Quality of Service**](https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/) greater than 0).
* MQTT works asynchronously. Because most client libraries work asynchronously and are based on callbacks or a similar model, tasks are not blocked while waiting for a message or publishing a message. In certain use cases, synchronization is desirable and possible. To wait for a certain message, some libraries have synchronous APIs. But the flow is usually asynchronous.
* **MQTT uses subject-based filtering of messages. Every message contains a topic (subject)** that the broker can use to determine whether a subscribing client gets the message or not.

When we talk about a client, we almost always mean an [**MQTT client**](https://www.hivemq.com/blog/seven-best-mqtt-client-tools/). Both publishers and subscribers are MQTT clients.

Basically, any device that speaks MQTT over a TCP/IP stack can be called an MQTT client

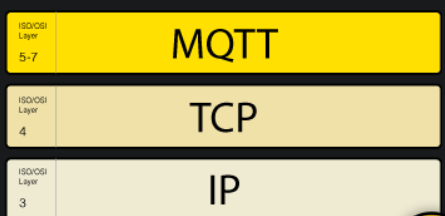
BROKER

**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.**

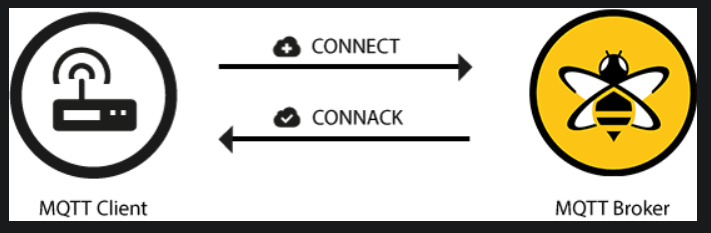
Another responsibility of the broker is the authentication and authorization of clients. Usually, the broker is extensible, which facilitates custom authentication, authorization, and integration into backend systems.

# MQTT Connection

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



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.



network address translation (NAT) ) to translate from a private network address (like 192.168.x.x, 10.0.x.x) to a public facing address.



#### ClientId

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

#### Clean Session

The clean session flag tells the broker whether the client wants to establish a persistent session or not. In a persistent session (CleanSession = false), the broker stores all subscriptions for the client and all missed messages for the client that subscribed with a [**Quality of Service (QoS)**](https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/) level 1 or 2. If the session is not persistent (CleanSession = true), the broker does not store anything for the client and purges all information from any previous persistent session.

The data will be stored when client disconnect to broker, and the number of data storing is dependent on the config of broker ( test in mosquito: 100values) .

#### Username/Password

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

#### Will Message

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

#### KeepAlive

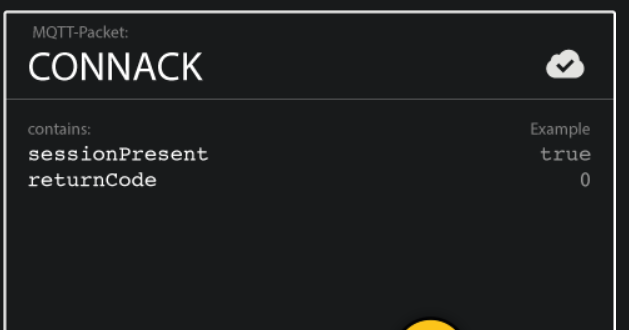
The keep alive is **a time interval in seconds** that the client specifies and communicates to the broker when the connection established. This interval defines the longest period of time that the broker and client can endure without sending a message. The client commits to sending regular PING Request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

### Broker response with a CONNACK message

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

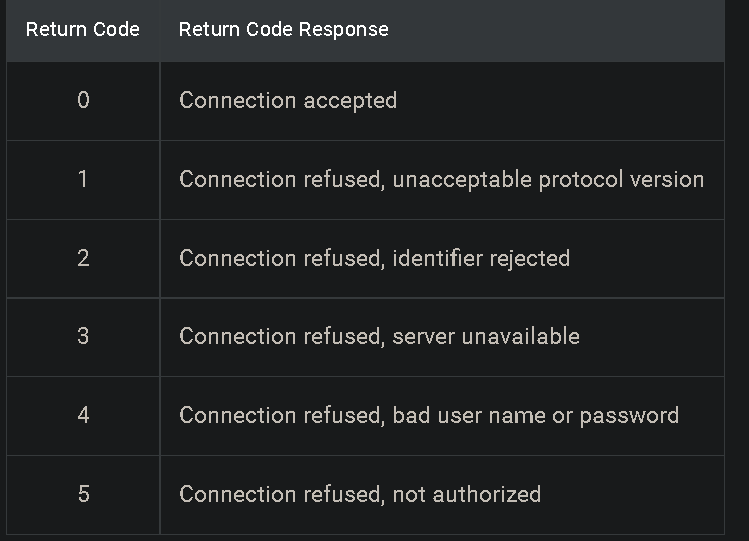
The CONNACK message contains two data entries:

* The session present flag
* A connect return code



#### Session Present flag

The **session present flag tells the client whether the broker already has a persistent session available from previous interactions with the client**. When a client connects with Clean Session set to true, the session present flag is always false because there is no session available. If a client connects with Clean Session set to false, there are two possibilities: If session information is available for the clientId. and the broker has stored session information, the session present flag is true. Otherwise, if the broker does not have any session information for the clientId, the session present flag is false.



# **Publish**

**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.**



**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”. For details on topics, see [**part 5 of MQTT Essentials**](https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/).

**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). For details on QoS, see [**part 6 of MQTT Essentials**](https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/).

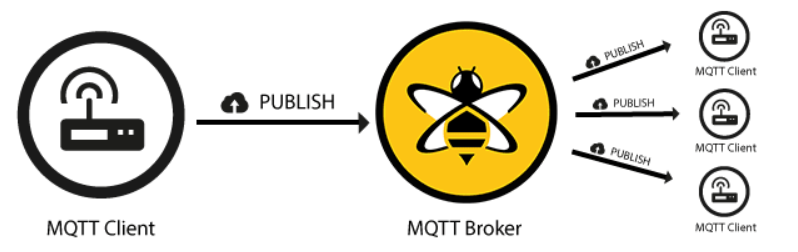
**Retain Flag** This flag defines whether the message is saved by the broker as the last known good value for a specified topic. When a new client subscribes to a topic, they receive the last message that is retained on that topic. For details on retained messages, see [**part 8 of MQTT Essentials**](https://www.hivemq.com/blog/mqtt-essentials-part-8-retained-messages/).

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

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

**DUP flag** The flag indicates that the message is a duplicate and was resent because the intended recipient (client or broker) did not acknowledge the original message. This is only relevant for QoS greater than 0. Usually, the resend/duplicate mechanism is handled by the MQTT client library or the broker as an implementation detail. For more information, [**part 6 of MQTT Essentials**](https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/).

When a client sends a message to an [**MQTT broker**](https://www.hivemq.com/hivemq/) 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.



# SUBCRIBE:



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

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

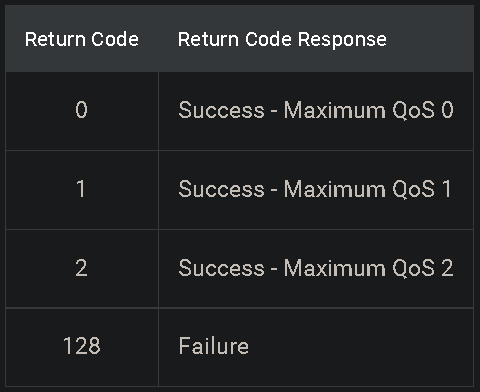
**Suback**

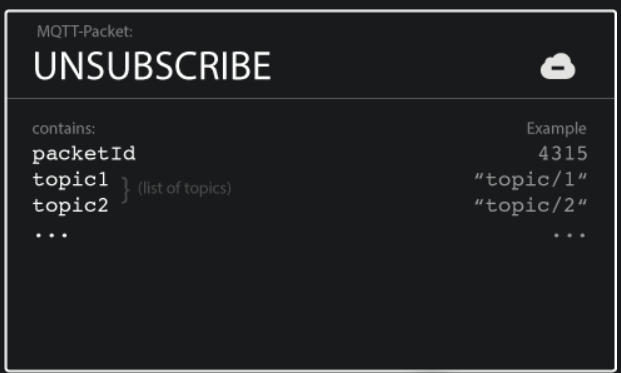
To confirm each subscription, the broker sends a [**SUBACK**](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718068) acknowledgement message to the client.

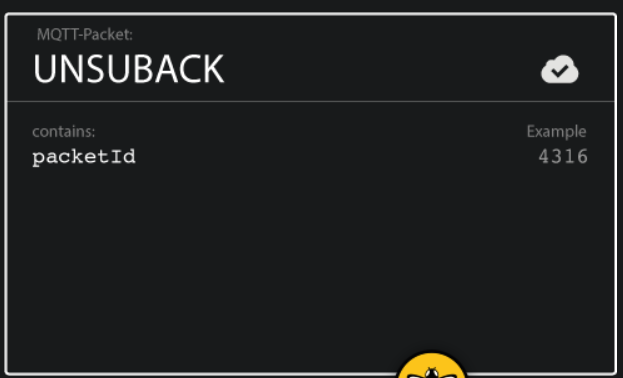


**Packet Identifier** The packet identifier is a unique identifier used to identify a message. It is the same as in the SUBSCRIBE message.

**Return Code** The broker sends one return code for each topic/QoS-pair that it receives in the SUBSCRIBE message. For example, if the SUBSCRIBE message has five subscriptions, the SUBACK message contains five return codes.

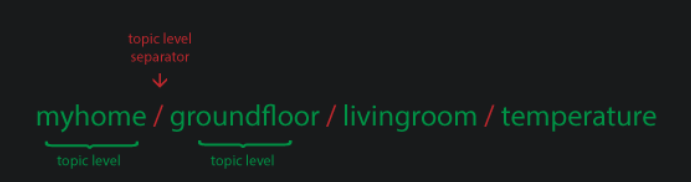






# **Topics**

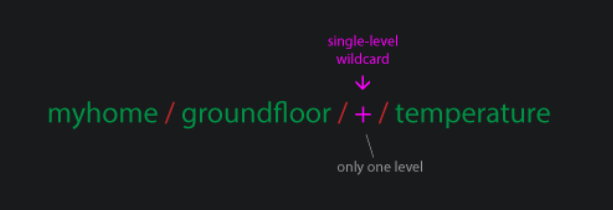
In MQTT, the word topic refers to an UTF-8 string that the broker uses to filter messages for each connected client.



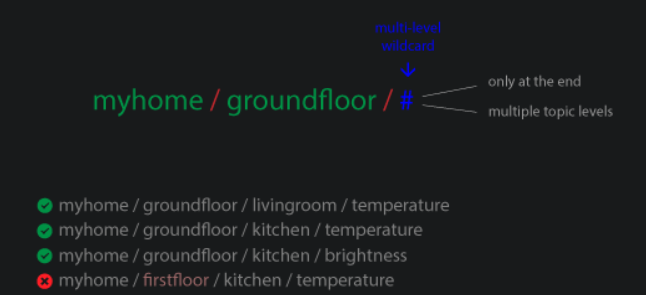
Note that each topic must contain **at least 1 character** and that the topic string permits empty spaces. **Topics are case-sensitive.** For example, myhome/temperature and MyHome/Temperature are two different topics. Additionally, the forward slash alone is a valid topic.

## Wildcards

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







## Topics beginning with $

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

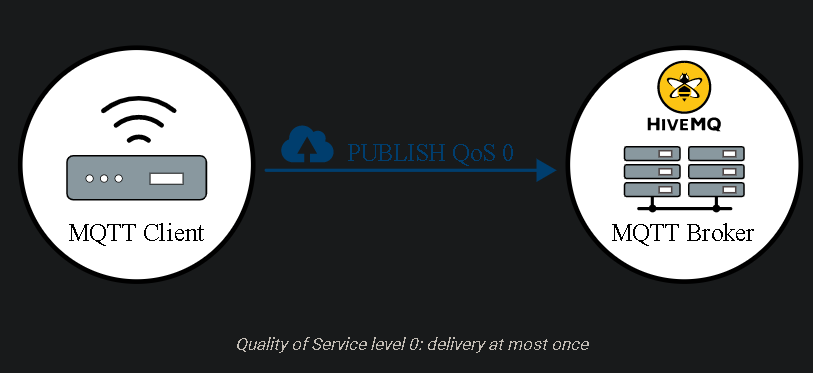
# **Quality of Service**

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

If the subscribing client defines a lower QoS than the publishing client, the broker transmits the message with the lower quality of service.

#### QoS 0 - at most once

The minimal QoS level is zero. 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.

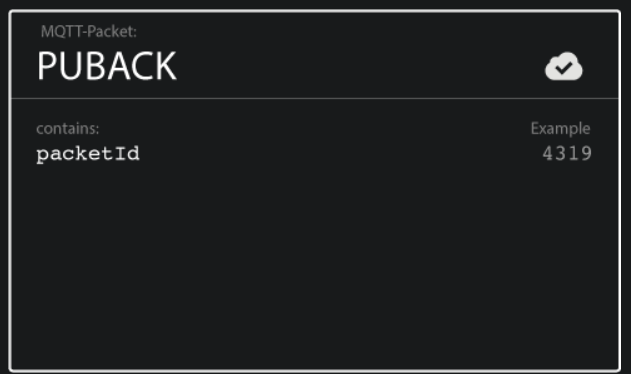


#### QoS 1 - at least once

The sender stores the message until it gets a [**PUBACK**](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718043) packet from the receiver that acknowledges receipt of the message. It is possible for a message to be sent or delivered multiple times.

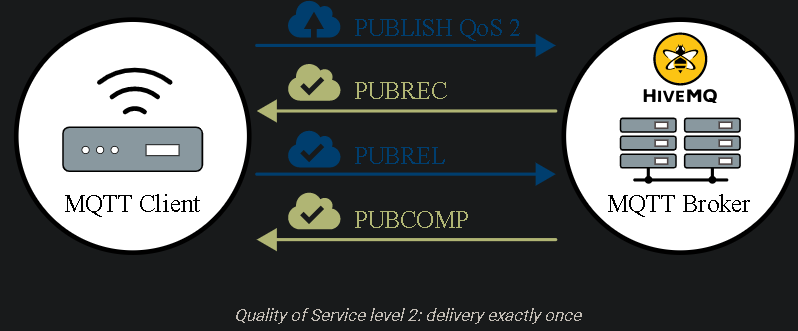


The sender uses the packet identifier in each packet to match the PUBLISH packet to the corresponding PUBACK packet. If the sender does not receive a PUBACK packet in a reasonable amount of time, the sender resends the PUBLISH packet. When a receiver gets a message with QoS 1, it can process it immediately.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

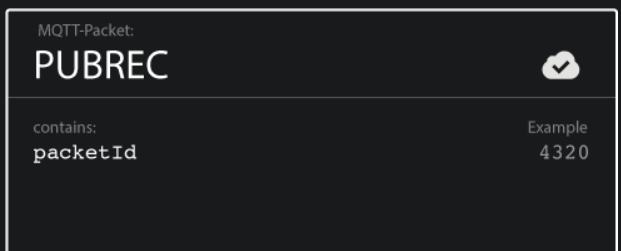


#### QoS 2 - exactly once

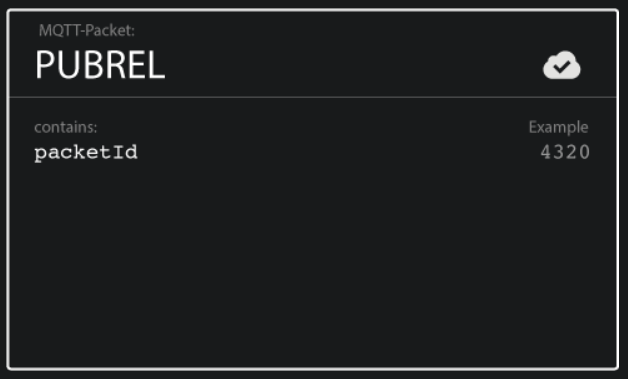
This level guarantees that each message is received only once by the intended recipients. QoS 2 is the safest and slowest quality of service level. The guarantee is provided by at least two request/response flows (a four-part handshake) between the sender and the receiver. The sender and receiver use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



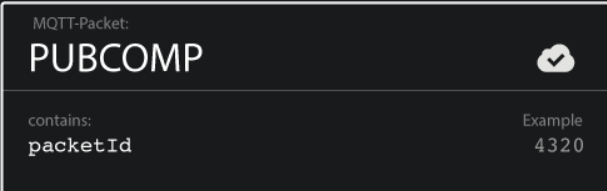
When a receiver gets a QoS 2 PUBLISH packet from a sender, it processes the publish message accordingly and replies to the sender with a [**PUBREC**](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718048) packet that acknowledges the PUBLISH packet. If the sender does not get a PUBREC packet from the receiver, it sends the PUBLISH packet again with a duplicate (DUP) flag until it receives an acknowledgement.



Once the sender receives a PUBREC packet from the receiver, the sender can safely discard the initial PUBLISH packet. The sender stores the PUBREC packet from the receiver and responds with a [**PUBREL**](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718053) packet.



After the receiver gets the PUBREL packet, it can discard all stored states and answer with a [**PUBCOMP**](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718058) packet (the same is true when the sender receives the PUBCOMP). Until the receiver completes processing and sends the PUBCOMP packet back to the sender, the receiver stores a reference to the packet identifier of the original PUBLISH packet. This step is important to avoid processing the message a second time. After the sender receives the PUBCOMP packet, the packet identifier of the published message becomes available for reuse.



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

If a packet gets lost along the way, the sender is responsible to retransmit the message within a reasonable amount of time. This is equally true if the sender is an MQTT client or an [**MQTT broker**](https://www.hivemq.com/hivemq/). The recipient has the responsibility to respond to each command message accordingly.

#### Use QoS 0 when …

* You have a completely or mostly stable connection between sender and receiver. A classic use case for QoS 0 is connecting a test client or a front end application to an MQTT broker over a wired connection.
* You don’t mind if a few messages are lost occasionally. The loss of some messages can be acceptable if the data is not that important or when data is sent at short intervals
* You don’t need message queuing. Messages are only queued for disconnected clients if they have QoS 1 or 2 and a [**persistent session**](https://www.hivemq.com/blog/mqtt-essentials-part-7-persistent-session-queuing-messages).

#### Use QoS 1 when …

* You need to get every message and your use case can handle duplicates. QoS level 1 is the most frequently used service level because it guarantees the message arrives at least once but allows for multiple ++able to process them accordingly.
* You can’t bear the overhead of QoS 2. QoS 1 delivers messages much faster than QoS 2.

#### Use QoS 2 when …

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

# **Persistent Session**

To receive messages from an [**MQTT broker**](https://www.hivemq.com/hivemq/), a client connects to the broker and creates [**subscriptions to the topics**](https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/) 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.

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)**](https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/) 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.

### Persistent session on the client side

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

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

### Clean session

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

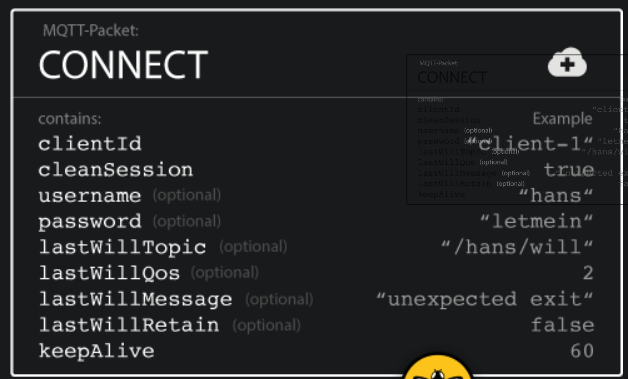
# **Retained Messages**

**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.** 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.**

# **Last Will and Testament**

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

****

According to the [**MQTT 3.1.1 specification**](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html), 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.

### Best Practices - When should you use LWT?

LWT is a great way to notify other subscribed clients about the unexpected loss of connection of another client. In real-world scenarios, LWT is often combined with [**retained messages**](https://www.hivemq.com/blog/mqtt-essentials-part-8-retained-messages/) to store the state of a client on a specific topic. For example, client1 first sends a CONNECT message to the broker with a lastWillMessage that has “Offline” as the payload, the lastWillRetain flag set to true, and the lastWillTopic set to client1/status. Next, the client sends a PUBLISH message with the payload “Online” and the retained flag set to true to the same topic (client1/status). As long as client1 stays connected, newly-subscribed clients to the client1/status topic receive the “Online” retained message. If client1 disconnects unexpectedly, the broker publishes the LWT message with the payload “Offline” as the new retained message. Clients that subscribe to the topic while client1 is offline, receive the LWT retained message ("Offline") from the broker. This pattern of retained messages keeps other clients up to date on the current status of client1 on a specific topic.

# **The problem of half-open TCP connections**

[**MQTT is based on the Transmission Control Protocol (TCP)**](https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/). This protocol ensures that packets are transferred over the internet in a [**“reliable, ordered, and error-checked”**](http://en.wikipedia.org/wiki/Transmission_Control_Protocol) way. Nevertheless, from time to time, the transfer between communicating parties can get out of sync. For example, if one of the parties crashes or has transmission errors. In TCP, this state of incomplete connection is called a [**half-open connection**](http://en.wikipedia.org/wiki/TCP_half-open). 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**

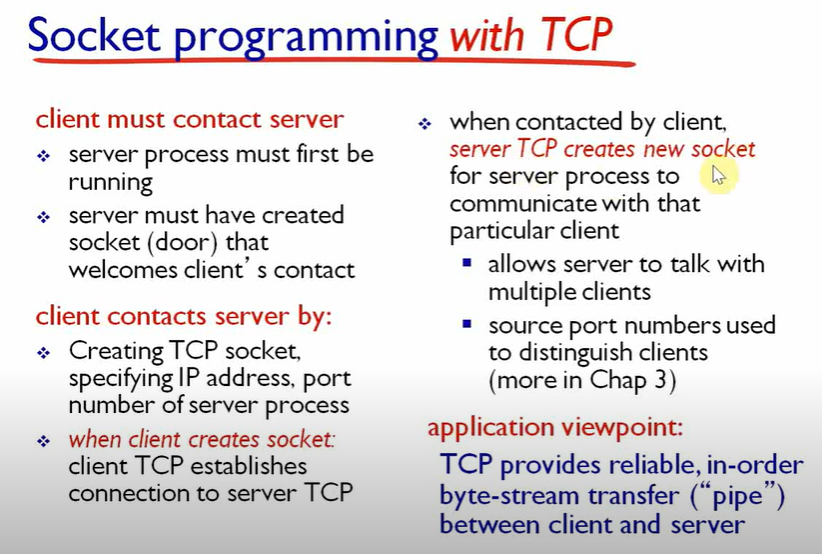
**Keep alive ensures that the connection between the broker and client is still open and that the broker and the client are aware of being connected.**

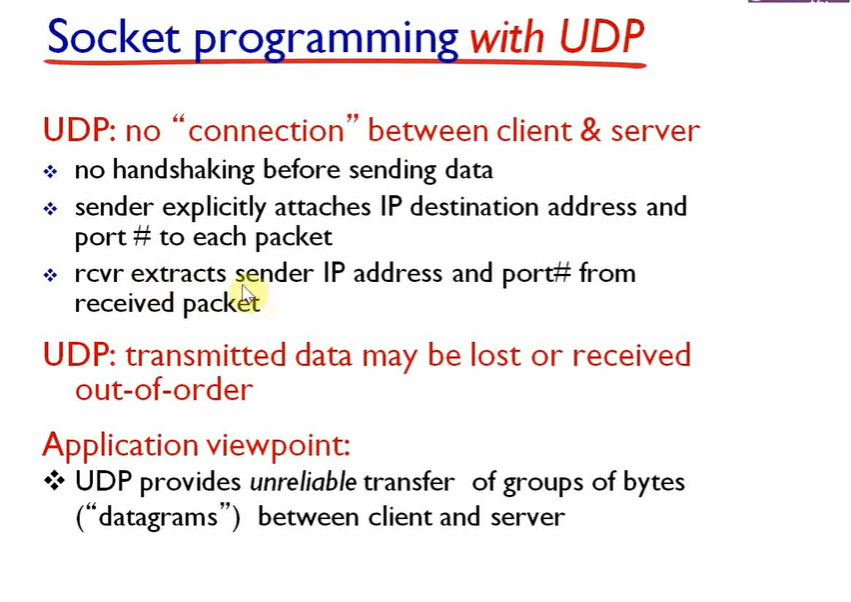
This interval defines the maximum length of time that the broker and client may not communicate with each other.

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

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

* The maximum keep alive is 18h 12min 15 sec.
* If the keep alive interval is 0, the keep alive mechanism is deactivated.

****

****