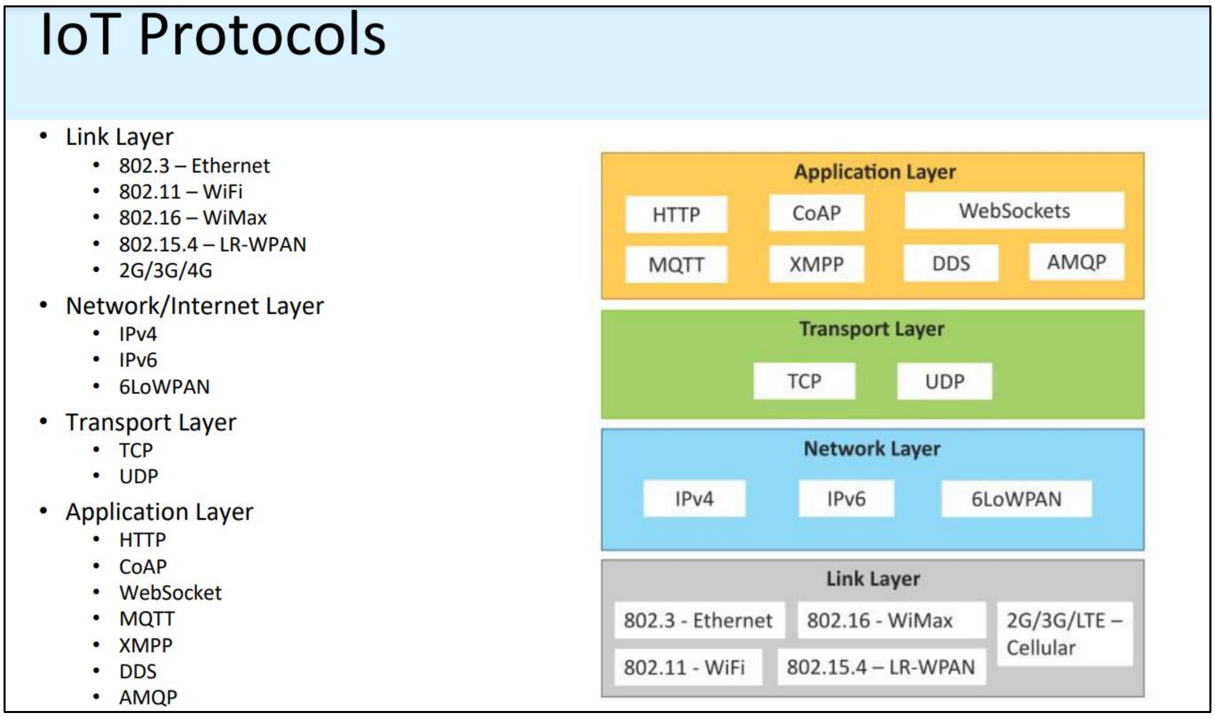
**IoT protocols:**
**Reference book: "Internet of Things: A Hands-on Approach", by Arshdeep Bahga and Vijay Madisetti**

## Link Layer

Link layer protocols determine how the data is physically sent over the network's physical layer or medium (e.g., copper wire, coaxial cable, or a radio wave). The scope of the link layer is the local network connection to which host is attached. Hosts on the same link exchange data packets over the link layer using link layer protocols. Link layer determines how the packets are coded and signaled by the hardware device over the medium to which the host is attached (such as a coaxial cable). Let us now look at some link layer protocols which are relevant in the context of IoT.



IoT Protocols

- Link Layer
  - 802.3 – Ethernet
  - 802.11 – WiFi
  - 802.16 – WiMax
  - 802.15.4 – LR-WPAN
  - 2G/3G/4G
- Network/Internet Layer
  - IPv4
  - IPv6
  - 6LoWPAN
- Transport Layer
  - TCP
  - UDP
- Application Layer
  - HTTP
  - CoAP
  - WebSocket
  - MQTT
  - XMPP
  - DDS
  - AMQP

## Network/Internet Layer

The network layers are responsible for sending of IP datagrams from the source network to the destination network. This layer performs the host addressing and packet routing. The datagrams contain the source and destination addresses which are used to route them from the source to destination across multiple networks. Host identification is done using hierarchical IP addressing schemes such as IPv4 or IPv6.

## Transport Layer

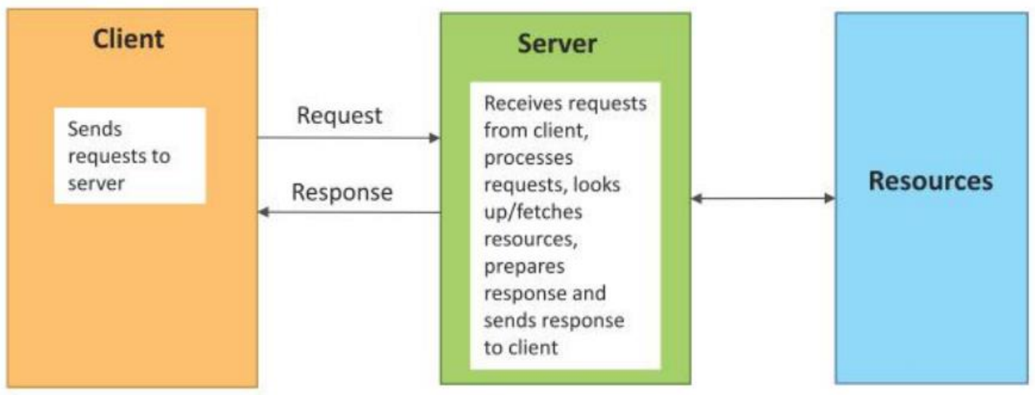
The transport layer protocols provide end-to-end message transfer capability independent of the underlying network. The message transfer capability can be set up on connections, either using handshakes (as in TCP) or without handshakes/acknowledgements (as in UDP). The transport layer provides functions such as error control, segmentation, flow control and congestion control.

## Application Layer

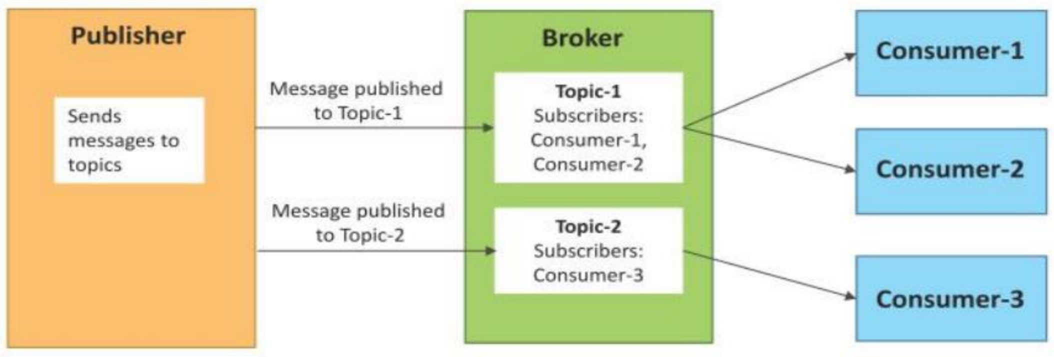
Application layer protocols define how the applications interface with the lower layer protocols to send the data over the network. The application data, typically in files, is encoded by the application layer protocol and encapsulated in the transport layer protocol which provides connection or transaction oriented communication over the network. Port numbers are used for application addressing (for example port 80 for HTTP, port 22 for SSH, etc.). Application layer protocols enable process-to-process connections using ports.

**IoT communication models:**

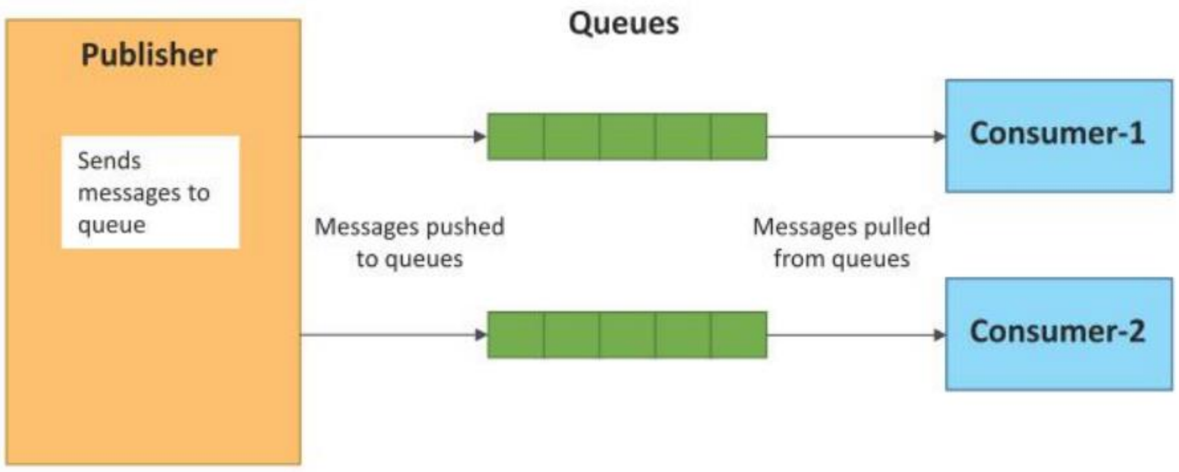
**Request Response** : Request-Response is a communication model in which the client sends requests to the server and the server responds to the requests. When the server receives a request, it decides how to respond, fetches the data, retrieves resource representations, prepares the response, and then sends the response to the client. Request-Response model is a stateless communication model and each o request-response pair is independent of others. Following figure shows the client-server interactions in the request-response model.


Client
Sends requests to server
Request
Response
Server
Receives requests from client, processes requests, looks up/fetches resources, prepares response and sends response to client
Resources

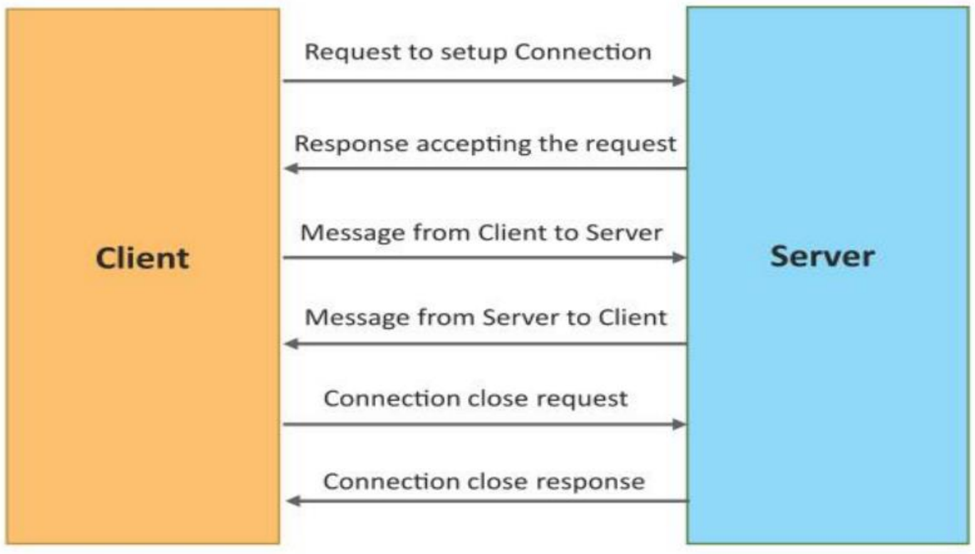
**Publish-Subscribe**: Publish-Subscribe is a communication model that involves publishers, brokers and consumers. Publishers are the source of data. Publishers send the data to the topics which are managed by the broker. Publishers are not aware of the consumers. Consumers subscribe to the topics which are managed by the broker. When the broker receives data for a topic from the publisher, it sends the data to all the subscribed consumers. Following figure  shows the publisher-broker-consumer interactions in the publish-subscribe model.


Publisher
Sends messages to topics
Message published to Topic-1
Message published to Topic-2
Broker
Topic-1 Subscribers: Consumer-1, Consumer-2
Topic-2 Subscribers: Consumer-3
Consumer-1
Consumer-2
Consumer-3

**Push-Pull**: Push-Pull is a communication model in which the data producers push the data to queues and the consumers pull the data from the queues. Producers do not need to be aware of the consumers. Queues help in decoupling the messaging between the producers and consumers. Queues also act as a buffer which helps in situations when there is a mismatch between the rate at which the producers push data and the rate rate at which the consumers pull data. Following figure shows the publisher-queue-consumer interactions in the push-pull model.



**Exclusive Pair** : Exclusive Pair is a bi-directional, fully duplex communication model that uses a persistent connection between the client and server. Once the connection is setup it remains open until the client sends a request to close the connection. Client and server can send messages to each other after connection setup. Exclusive pair is a stateful communication model and the server is aware of all the open connections. Following figure shows the client-server interactions in the exclusive pair model.


Client
Server
Request to setup Connection
Response accepting the request
Message from Client to Server
Message from Server to Client
Connection close request
Connection close response

MQTT:
Ref: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718028
Ref: https://en.wikipedia.org/wiki/MQTT
Ref: https://archive.nptel.ac.in/courses/106/105/106105166/
Ref: https://www.hivemq.com/mqtt-essentials/
Ref: http://www.steves-internet-guide.com/understanding-mqtt-qos-levels-part-1/
Ref: http://www.steves-internet-guide.com/understanding-mqtt-qos-2/

MQTT stands for Message Queue Telemetry Transport. MQTT is the most commonly used messaging protocol for the Internet of Things (IoT).
MQTT (Message Queue Telemetry Transport) is a lightweight protocol that is designed for connections with remote locations that have devices with resource constraints or limited network bandwidth.

MQTT is used for messaging and data exchange between IoT and industrial IoT (IIoT) devices, such as embedded devices, sensors, industrial PLCs, etc.
The protocol is event driven and connects devices using the publish /subscribe (Pub/Sub) pattern.
The sender (Publisher) and the receiver (Subscriber) communicate via Topics and are decoupled from each other. A topic to which a client is subscribed is updated in the form of messages and distributed by the message broker.
The connection between them is handled by the MQTT broker. The MQTT broker filters all incoming messages and distributes them correctly to the Subscribers. 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 fully-fledged server) that runs an MQTT library and connects to an MQTT broker over a network.
It must run over a transport protocol that provides ordered, lossless, bi-directional connections—typically, TCP/IP.
It is an open OASIS standard and an ISO recommendation (ISO/IEC 20922).
MQTT was introduced by IBM in 1999 and standardized by OASIS in 2013.
It is designed for:

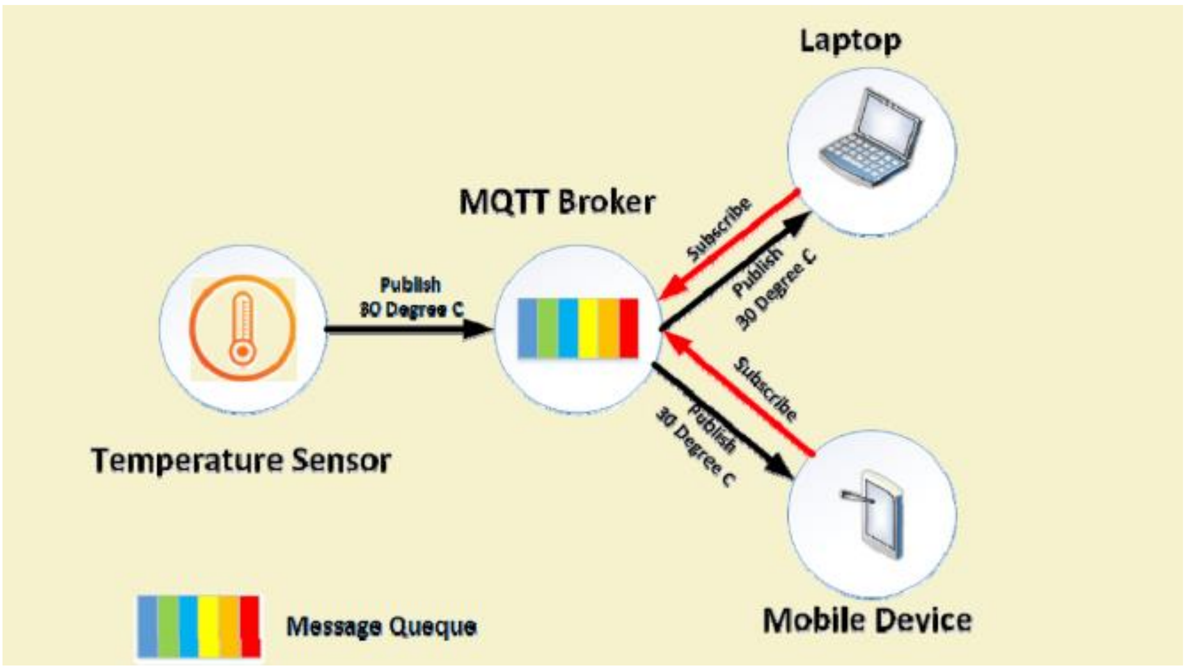
- Remote connections
- Limited bandwidth
- Small-code footprint

“MQTT is a Client Server publish/subscribe messaging transport protocol. It is light weight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium.“
*-Citation from the official MQTT 3.1.1 specification*
**The 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. So, let’s dive a little deeper into some of the general aspects of pub/sub (we’ll talk about MQTT specifics in a minute).

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.

Another thing that should be mentioned is that MQTT is especially easy to use on the client-side. Most pub/sub systems have the 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.

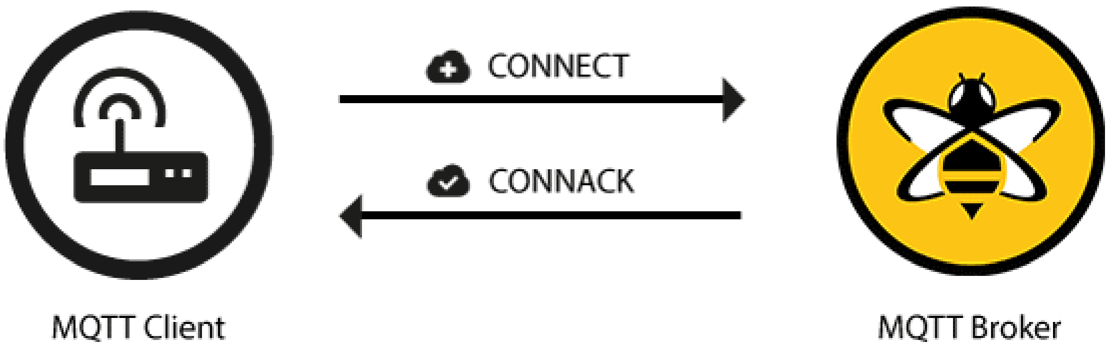
Both publishers and subscribers are MQTT clients. The publisher and subscriber labels refer to whether the client is currently publishing messages or subscribed to receive messages (publish and subscribe functionality can also be implemented in the same MQTT client) **The broker is responsible for receiving all messages, filtering the messages, determining who is subscribed to each message, and sending the message to these subscribed clients.** The broker also holds the session data of all clients that have persistent sessions, including subscriptions and missed messages (more **details**). Another responsibility of the broker is the authentication and authorization of clients.

**MQTT control packets:**

**Connect and Connack**

To initiate a connection, the client sends a command message to the broker. Connect packet includes the client identifier (ClientId) that **identifies each MQTT client** that connects to an MQTT broker. MQTT can send a **user name and password for client authentication and authorization** in a connect packet. Connect packet includes the keep alive. It 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

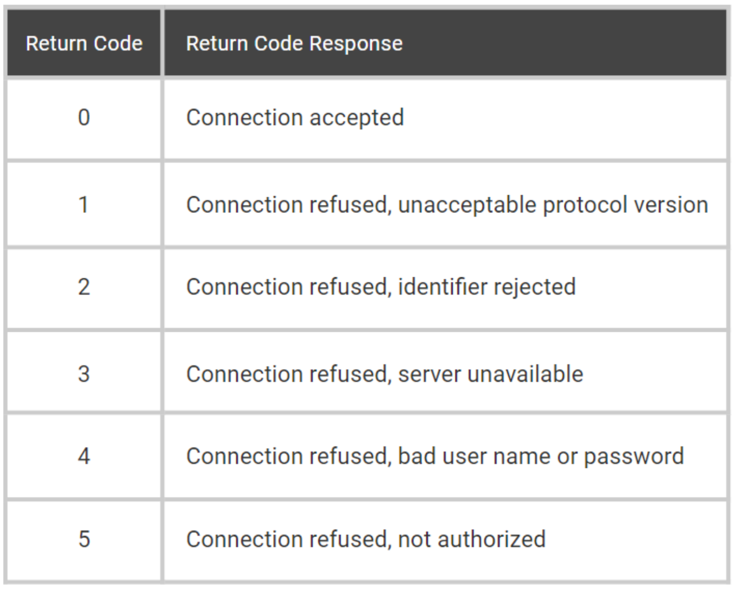
Request messages to the broker. The broker responds with a PING response. This method allows both sides to determine if the other one is still available



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 a connect return code that is **a return code that tells the client whether the connection attempt was successful or not**. If unsuccessful then why connection was refused.



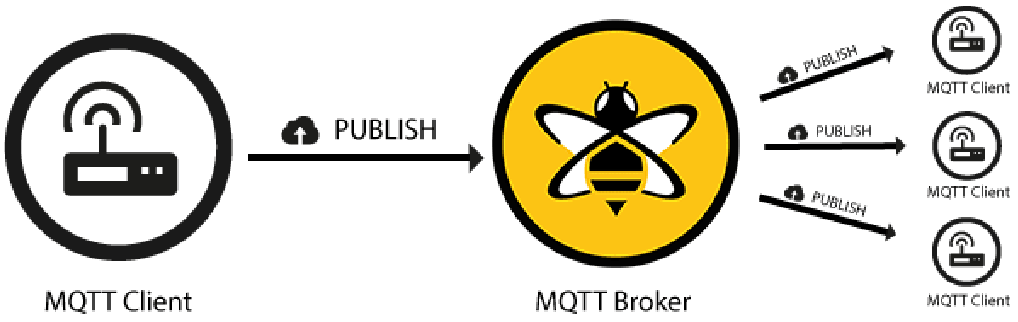
| Return Code | Return Code Response |
|---|---|
| 0 | Connection accepted |
| 1 | Connection refused, unacceptable protocol version |
| 2 | Connection refused, identifier rejected |
| 3 | Connection refused, server unavailable |
| 4 | Connection refused, bad user name or password |
| 5 | Connection refused, not authorized |

**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 (see **part 2** for details). **Each message must contain a topic that the broker can use to forward the message to interested clients It also contains the packet Identifier** that uniquely identifies a message as it flows between the client and broker. **QoS** This number indicates the Quality of Service Level (QoS) of the message. There are three levels: 0, 1, and 2.

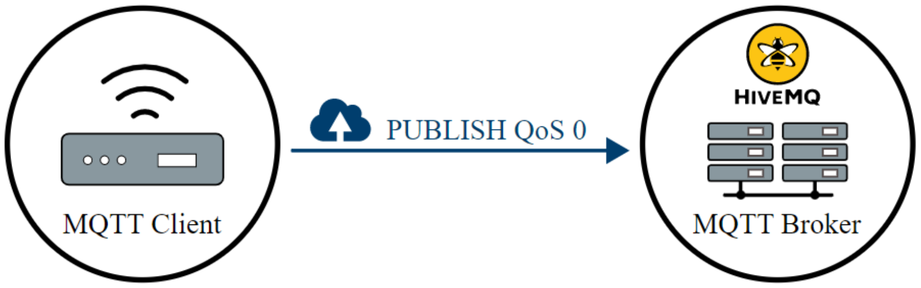
Table 3.2 - QoS definitions

| QoS value | Bit 2 | bit 1 | Description |
|---|---|---|---|
| 0 | 0 | 0 | At most once delivery |
| 1 | 0 | 1 | At least once delivery |
| 2 | 1 | 0 | Exactly once delivery |
| - | 1 | 1 | Reserved – must not be used |



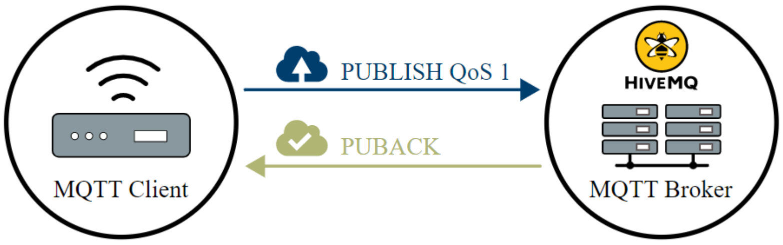
*QoS 0 - at most once*

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.

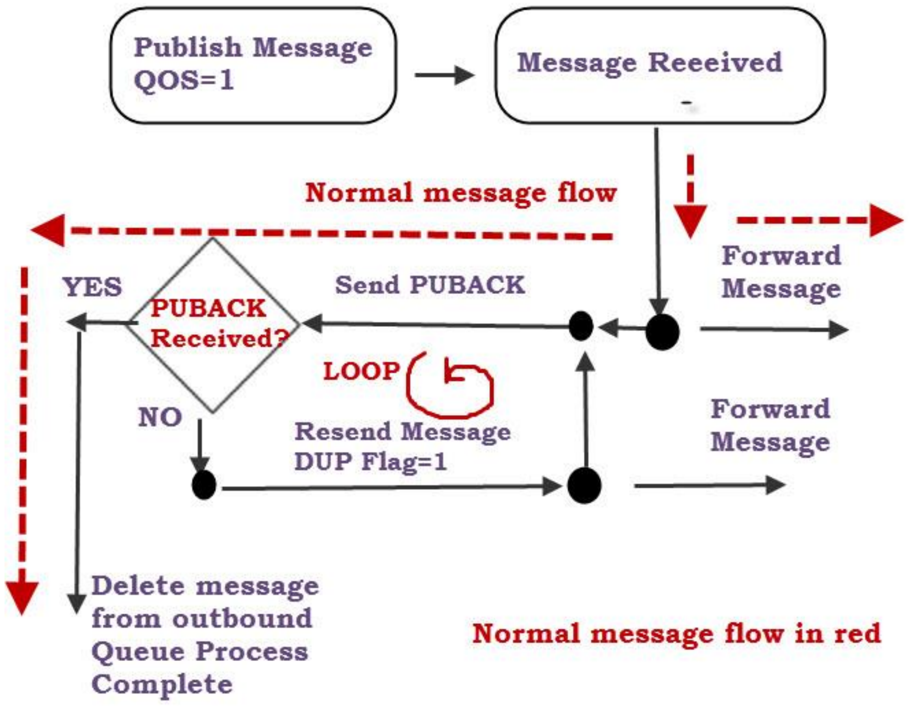


*QoS 1 - at least once*

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.



Quality of Service level 1: delivery at least once

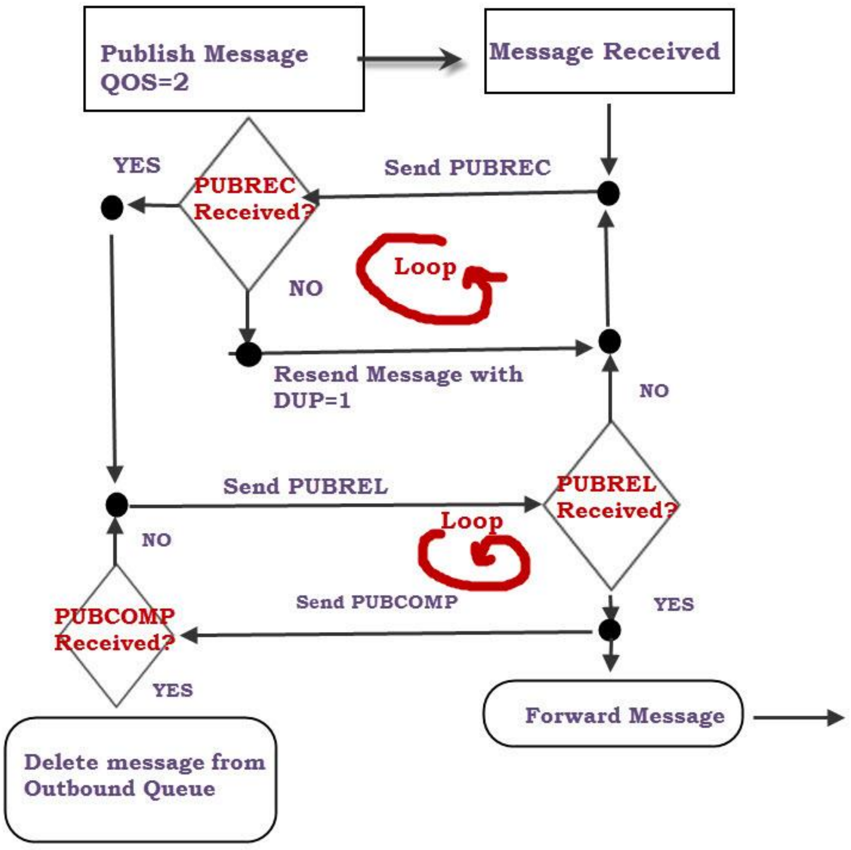


**MQTT QOS 1 Message Flow Diagram**

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. If the publishing client sends the message again it sets a duplicate (DUP) flag.

QOS 2 – Only Once
This level guarantees that the message will be delivered **only once**.
This is the slowest method as it requires 4 messages.
1- The sender sends a message and waits for an acknowledgement (**PUBREC**)
2 -The receiver sends a **PUBREC** message
3. If the sender doesn't receive an acknowledgement ( **PUBREC**) it will resend the message with the **DUP flag** set.
4. When the sender receives an acknowledgement message **PUBREC** it then sends a message release message (**PUBREL**). The message can be deleted from the queue.
5. If the receiver doesn't receive the **PUBREL** it will resend the **PUBREC** message

5. When the receiver receives the **PUBREL** message it can now forward the message onto any subscribers.
6. The receiver then send a publish complete (**PUBCOMP**) .
7. If the sender doesn't receive the PUBCOMP message it will resend the **PUBREL** message.
8. When the sender receives the **PUBCOMP** the process is complete and it can **delete the message** from the outbound queue, and also the message state.
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.



**MQTT QOS 2 Message Flow Diagram**

Client to Client or End to End QOS

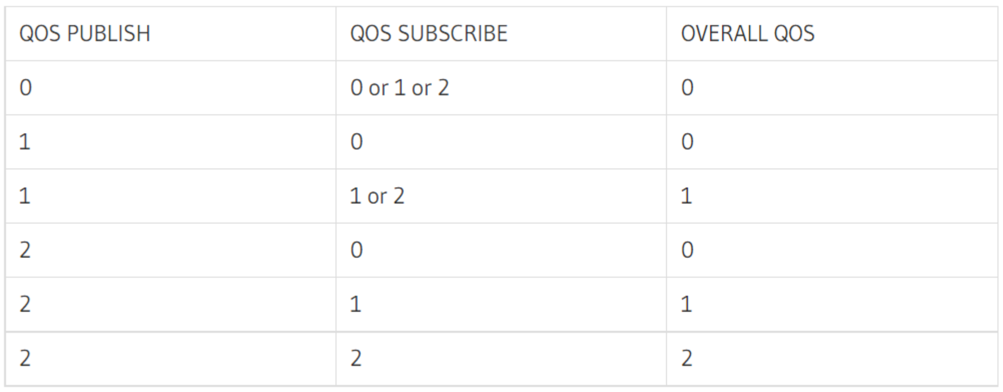
The Quality of service between **two clients** connected to a broker is determined by the QOS of the published message,and the QOS of the subscribing client.
If a subscribing Client has been granted maximum QoS 1 for a particular Topic Filter, then a QoS 0 Application Message matching the filter is delivered to the Client at QoS 0. This means that at most one copy of the message is received by the Client. On the other hand a QoS 2 Message published to the same topic is downgraded by the Server to QoS 1 for delivery to the Client, so that Client might receive duplicate copies of the Message.

If the subscribing Client has been granted maximum QoS 0, then an Application Message originally published as QoS 2 might get lost on the hop to the Client, but the Server should never send a duplicate of that Message. A QoS 1 Message published to the same topic might either get lost or duplicated on its transmission to that Client.

**Non normative comment**

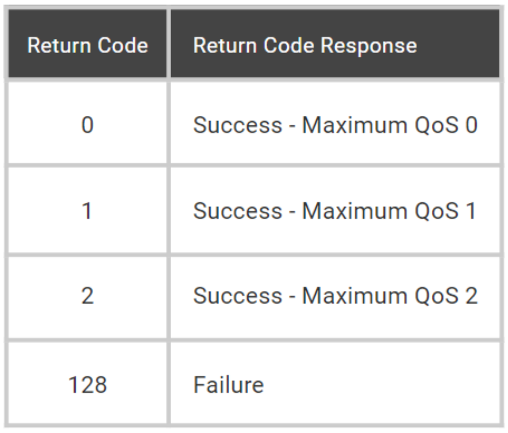
Subscribing to a Topic Filter at QoS 2 is equivalent to saying "I would like to receive Messages matching this filter at the QoS with which they were published". This means a publisher is responsible for determining the maximum QoS a Message can be delivered at, but a subscriber is able to require that the Server downgrades the QoS to one more suitable for its usage.

The overall QOS is always equal to the lowest QOS of the publish or subscribe, as shown in the table below. 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.

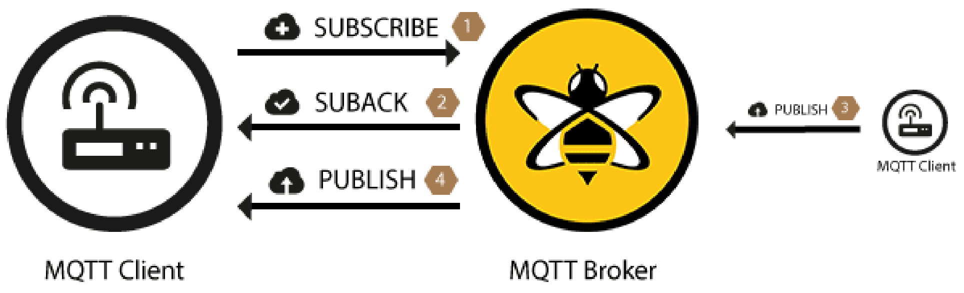


| QOS PUBLISH | QOS SUBSCRIBE | OVERALL QOS |
|---|---|---|
| 0 | 0 or 1 or 2 | 0 |
| 1 | 0 | 0 |
| 1 | 1 or 2 | 1 |
| 2 | 0 | 0 |
| 2 | 1 | 1 |
| 2 | 2 | 2 |

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.
**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. The return code acknowledges each topic and shows the QoS level that is granted by the broker. If the broker refuses a subscription, the SUBACK message conains 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 |



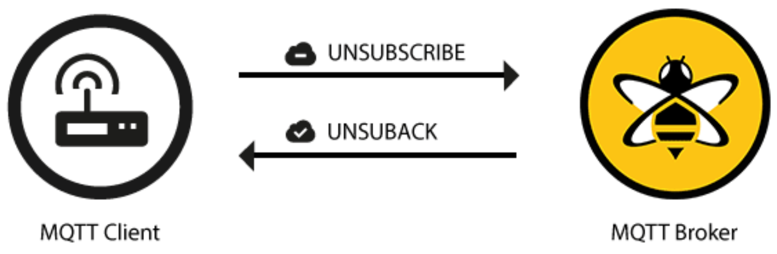
After a client successfully sends the SUBSCRIBE message and receives the SUBACK message, it gets every published message that matches a topic in the subscriptions that the SUBSCRIBE message contained.

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.

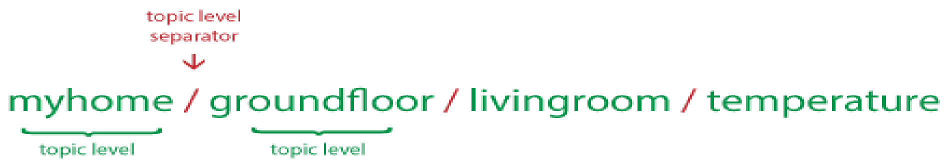
Unsuback

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).



## MQTT 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).

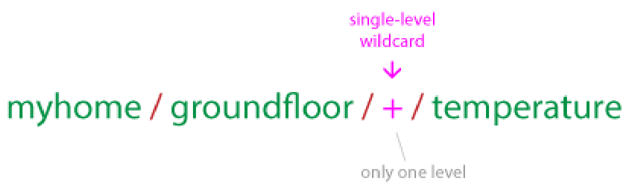


MQTT Wildcards

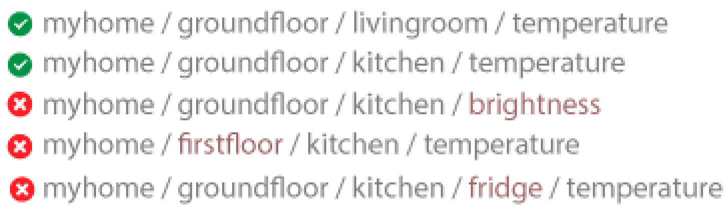
When a client subscribes to a topic, it can subscribe to the exact topic of a published message or it can use wildcards to subscribe to multiple topics simultaneously. A wildcard can only be used to subscribe to topics, not to publish a message. There are two different kinds of wildcards: *single-level*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

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.

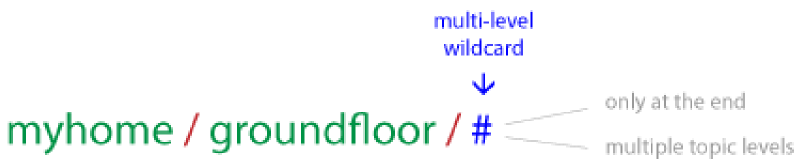


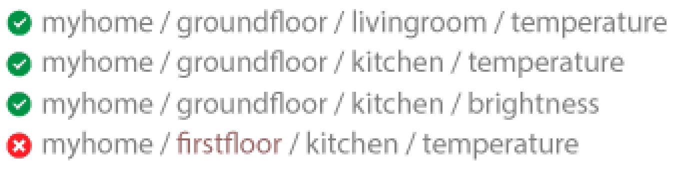
Any topic matches a topic with single-level wildcard if it contains an arbitrary string instead of the wildcard. For example a subscription to *myhome/groundfloor/+/temperature* can produce the following results:


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

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.




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

When a client subscribes to a topic with a multi-level wildcard, it receives all messages of a topic that begins with the pattern before the wildcard character, no matter how long or deep the topic is. If you specify only the multi-level wildcard as a topic (#), you receive all messages that are sent to the MQTT broker.