GETTING STARTED WITH MQTT GUIDE

Version Information
Version: 1.0
Release Date: 6/10/2019

Author Information
Justin Dean

CSE ICON, Inc.
Email: justin.dean@cse-icon.com
Website: www.cse-icon.com

Sahoma Controlware, LLC
Email: jdean@sahomacontrolware.com
Website: www.sahomacontrolware.com
Phone: 405.230.0415

WHY MQTT?

The Industrial Internet of Things (IIoT) has recently gained massive traction. IIoT challenges enterprises, small companies, and developers with new problems to solve. While HTTP is the de-facto protocol for the human web, communication between machines at scale requires a paradigm shift, steering away from request/response and leading towards publish/subscribe. This is where the ultra-lightweight, massively scalable, and easy-to-implement protocol MQTT enters the picture.

WHAT IS MQTT?

MQTT is a binary client-server publish/subscribe messaging transport protocol first standardized by OASIS. It is lightweight, open, simple, and easy to implement. Designed with a minimal protocol overhead, this protocol is a good choice for a variety of Machine-to-Machine (M2M) and Industrial Internet of Things applications, especially where a small code footprint is required and/or network bandwidth is at a premium. MQTT utilizes many characteristics of the TCP transport, so the minimum requirement for using MQTT is a working TCP stack, which is now available for even the smallest embedded systems.

The most recent version of MQTT is 5.0, which has many improvements over the second public MQTT release, MQTT 3.1.1.

USE CASES

MQTT excels in scenarios where reliable message delivery is crucial for an application but a reliable network connection is not necessarily available, i.e. mobile, radio, licensed & unlicensed networks.

Typical use cases of MQTT include:

- ✓ Telemetry/SCADA
- ✓ Embedded Systems
- ✓ Manufacturing
- ✓ Energy Monitoring
- ✓ Oil & Gas
- ✓ Notification Services
- ✓ Device Management Applications
- ✓ Etc.

PUBLISH / SUBSCRIBE

MQTT implements the brokered publish / subscribe pattern. The publish / subscribe pattern decouples a client (“publisher”), which is sending a particular message from other clients (“subscribers”), which are receiving the message. This means that the publisher and subscribers don’t know about the existence of one another. The clients do not know each other, but they know the message broker, which filters all incoming messages and distributes them to the correct subscribers as shown in the below figure:

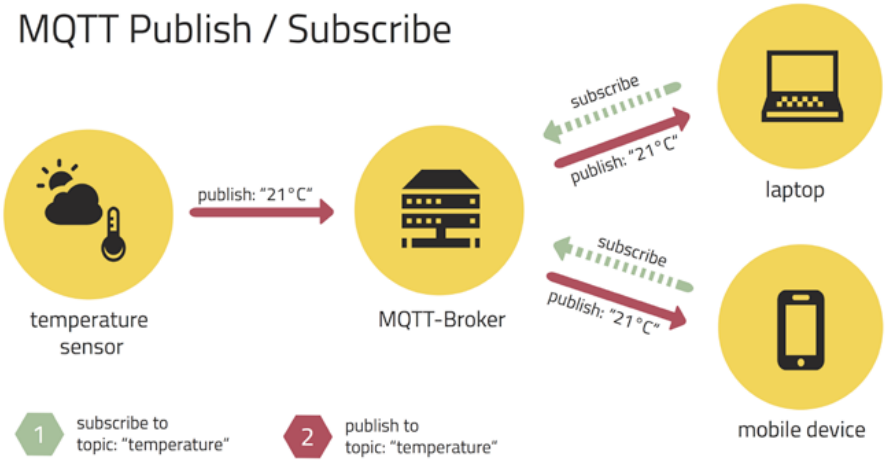

Figure 1: MQTT Publish / Subscribe

This decoupling of sender and receiver can be differentiated in three dimensions:

- Space decoupling: Publisher and subscriber do not need to know each other (for example, by IP address and port)
- Time decoupling: Publisher and subscriber do not need to be connected at the same time
- Synchronization decoupling: Operations on both components are not halted during publishing or receiving messages

MQTT MESSAGE TYPES

MQTT has 14 different message types. Typically, end users only need to employ the CONNECT, PUBLISH, SUBSCRIBE, and UNSUBSCRIBE message types. The other message types are used for internal mechanisms and message flows.

MESSAGE TYPE	DESCRIPTION
CONNECT	Client request to connect to Server
CONNACK	Connection Acknowledgement
PUBLISH	A message which represents a new/separate publish
PUBACK	QoS 1 Response to a PUBLISH message
PUBREC	First part of QoS 2 message flow
PUBREL	Second part of QoS 2 message flow
PUBCOMP	Last part of the QoS 2 message flow
SUBSCRIBE	A message used by clients to subscribe to specific
SUBACK	Acknowledgement of a SUBSCRIBE message
UNSUBSCRIBE	A message used by clients to unsubscribe from specific topics
UNSUBACK	Acknowledgement of an UNSUBSCRIBE message
PINGREQ	Heartbeat message
PINGRESP	Heartbeat message acknowledgement
DISCONNECT	Graceful disconnect message sent by clients before disconnecting

TOPICS

A topic is a UTF-8 string, which is used by the broker to filter messages for each connected client. A topic consists of one or more topic levels. Each topic level is separated by a forward slash (topic level separator).

In comparison to a message queue, a topic is very lightweight. There is no need for a client to create the desired topic before publishing or subscribing to it, because a broker accepts each valid topic without any prior initialization.

MQTT TOPIC WILDCARDS

MQTT Topic Wildcards can be used for topic filters when subscribing to MQTT messages. These wildcards are useful if a client wants to receive messages for different topics with similar structure at once.

Wildcards **are not allowed in topic names when publishing messages.** The wildcard characters are reserved and must not be used in the topic. These characters cannot be escaped.

WILDCARD	SYMBOL	DESCRIPTION
Single-Level Wildcard	+	A wildcard that matches one complete topic level. It must occupy an entire topic level. This wildcard can be used more than once in a topic subscription.
Multi-Level Wildcard	#	A wildcard that matches any number of levels within a topic. It must be the last character of a topic subscription.

VALID MQTT TOPIC EXAMPLES

- ✓ my/test/topic
- ✓ my/+/topic
- ✓ my/#
- ✓ my/+/+
- ✓ +/#
- ✓ #

QOS LEVELS

Each MQTT publish is sent with one of three Quality of Service (QoS) levels. These levels are associated with different guarantees with regards to the reliability of the message delivery. Both client and broker provide additional persistence and redelivery mechanisms to increase reliability in case of network failures, restarts of the application, and other unforeseen circumstances.

MQTT relies on TCP, which has reliability guarantees on its own. Historically QoS levels were needed to overcome data loss on older and unreliable TCP networks. This can still be a valid concern for mobile and low bandwidth networks today.

QOS LEVEL	DESCRIPTION
0	At most once delivery: The sender tries with best effort to send the message and relies on the reliability of TCP. No retransmission takes place.
1	At least once delivery: The receiver will get the message at least once. If the receiver does not acknowledge the message or the acknowledge gets lost on the way, it will be resent until the sender gets an acknowledgement. Duplicate messages can occur.
2	Exactly once delivery: The protocol makes sure that the message will arrive exactly once at the receiver. This increases communication overhead but is the best option when neither loss nor duplication of messages are acceptable.

LAST WILL AND TESTAMENT

A Last Will and Testament (LWT) message can be specified by an MQTT client when connecting to the MQTT broker. If that client does not disconnect gracefully, the broker sends out the LWT message on behalf of the client when connection loss is detected.

RETAINED MESSAGES

Each sent MQTT message can be sent as a retained message. A retained message is a last known good value and persists at the MQTT broker for the specified topic. Every time a new client subscribes to that specific topic, it will instantly receive the last retained message on that topic. This is also the case for matching wildcards.

CLEAN / PERSISTENT SESSIONS

When a client connects to an MQTT broker, it has the choice of requesting a persistent session. The broker is responsible for storing session information of the client if the client requested a persistent session. The session information of a client includes:

- ✓ All subscriptions of the client
- ✓ All QoS 1 / 2 messages which are not processed yet
- ✓ All QoS 1 / 2 messages the client missed while off line

Persistent sessions are often used for MQTT clients on constrained devices and clients who must not miss any messages for certain topics—not even when they are disconnected. When a client reconnects, the broker will send all missed messages for a subscription with a QoS Level of 1 or 2. Persistent sessions are most useful for clients that subscribe to topics; publishing-only clients don't profit from persistent sessions.

Clean sessions are often used by publishing-only MQTT clients that are not interested in any state.

HEARTBEATS

An MQTT CONNECT message contains a keepAlive value in seconds where the client can specify the maximum timeout between message exchanges. This allows the broker to detect a half-open connection and close the connection to the (already disconnected) client if the keepAlive value is exceeded by more than 150% of the value.

So, if a connection between broker and client is still established, the client sends a PINGREQ message to the broker within the keepAlive interval if no other message exchange occurred. The broker responds with a PINGRESP message.

Every client specifies its keepAlive value when connecting and the maximum value is 65535 seconds (18h 12m 15s).

MQTT BROKER IMPLEMENTATIONS

A variety of high-quality MQTT brokers are available. The following table shows the most popular open source and commercial broker implementations.

BROKER	DESCRIPTION
mosquitto	**mosquitto is an open source MQTT broker written in C. It fully supports MQTT 3.1, 3.1.1, 5.0 and is very lightweight. Due to its small size, this broker can be used on constrained devices.**
Apache ActiveMQ	ActiveMQ is an open-source multi-protocol message broker with a core written around JMS. It supports MQTT and maps MQTT semantics over JMS.
HiveMQ	**HiveMQ is a scalable, high-performance MQTT broker suitable for mission critical deployments. It fully supports MQTT 3.1 and MQTT 3.1.1 and has features like WebSocket's, clustering, and an open-source plugin system for Java developers. Full support of MQTT 5.0 has not been validated by our work.**
RabbitMQ	RabbitMQ is a scalable, open-source message queue implementation, written in Erlang. It is an AMQP message broker but has an MQTT plugin available. Does not support all MQTT features (e.g. QoS 2).
mosca	mosca is an open-source MQTT broker written in Node.js. It can operate as standalone or be embedded into any Node.js application. Does not implement all MQTT features (i.e. QoS 2).
RSMB	RSMB is a message broker by IBM available for personal use. It is written in C and is one of the oldest MQTT broker implementations available.
WebSphereMQ / IBM MQ	WebSphere MQ is a commercial message-oriented middleware by IBM. Fully supports MQTT.

*Note: MQTT Brokers in **bold** is the most popular and widely used.*

MQTT CLIENTS

A variety of MQTT client implementations are available for most of the popular operating systems and programming languages. These lists give an overview of the most popular MQTT client libraries and MQTT client tools.

MQTT CLIENT LIBRARIES

LIBRARY	LANGUAGE	DESCRIPTION
Eclipse Paho	**C, C++, Java, JavaScript, Python, Go, C#**	**Paho clients are among the most popular client library implementations.**
M2MQTT	C#	M2MQTT is an MQTT client library for .NET and WinRT.
Fusesource MQTT Client	Java	The Fusesource MQTT client is a Java MQTT client with 3 different API styles: Blocking, Future-based, and Callback-based.
Machine Head	Clojure	Machine Head is an MQTT client for Clojure. It implements the basic MQTT 3.1 features.
MQTT.js	JavaScript	MQTT.js is an MQTT client library for Node.js and web applications, available as a npm module.
ruby-mqtt	Ruby	ruby-mqtt is an MQTT client available as a Ruby gem. It does not support QoS > 0.

*Note: MQTT Client Libraries in **bold** is the most popular and widely used due their ease of use and language support.*

MQTT CLIENT TOOLS

CLIENT TOOL	OS	DESCRIPTION
MQTT.fx	Windows, Linux, MacOS	MQTT.fx is a JavaFX application with a clean interface and advanced features like scripting, broker statistics, and templates.
mqtt-spy	Windows, Linux, MacOS	mqtt-spy is a JavaFX application that is easy to use and focused on analyzing MQTT subscriptions. There is also a CLI-based daemon application available, which does not need a graphic interface.
MQTT Inspector	iOS	MQTT Inspector is an iOS app that allows detailed analysis of MQTT traffic. Use of the publish/subscribe message types, and complex filtering's of received messages, are available.
HiveMQ WebSocket Client	Web browser	The HiveMQ WebSocket client runs on any modern browser and connects to MQTT brokers via WebSocket's. Very useful if it's not possible to install a client application on the machine in use, as well as for quick MQTT tests.
MyMQTT	Android	MyMQTT is an MQTT test application for Android devices. It allows the creation of templates for publishing, which makes it very useful for testing MQTT "on-the-go."
MQTTLens	Google Chrome	MQTTLens is a Chrome Webapp that can connect to MQTT brokers via TCP and over WebSocket's. This app is easy to grasp and equipped with all the basic MQTT features needed for quick tests.
mosquitto_pub / mosquitto_sub	Windows, Linux, MacOS	mosquitto_pub and mosquitto_sub are the best options for publish/subscribe on servers without GUI. It is also great for MQTT task automation.

MQTT ON THE COMMAND LINE

Trying MQTT on the command line is very easy. Install either mosquitto or HiveMQ as the MQTT broker and start it. Download HiveMQ at hivemq.com/download and download the mosquitto client tools with the package manager of choice or via mosquitto.org.

To try MQTT without installing a broker, the following hosted brokers are available for free:

ADDRESS	PORT	BROKER
broker.mqttdashboard.com	1883	HiveMQ
test.mosquitto.org	1883, 8883, 8884, 8885	mosquitto
iot.eclipse.org	1883	mosquitto

Open two terminal windows, one for publishing and one for subscribing.

Subscriber

```
# Subscribing to an MQTT topic with QoS 2 and debug output

mosquitto_sub -h broker.mqttdashboard.com -t 'my/topic' -q
2 -d
```

Publisher

```
# Publishing an MQTT message with QoS 2

mosquitto_pub -h broker.mqttdashboard.com -t 'my/topic' -m
'my_message' -q 2
```

Now you should receive the message the publisher sent with the subscribing client.

Note: This MQTT commands has been tested with Linux AND Mac Operation Systems.

PUB / SUB WITH PAHO

Eclipse Paho is an umbrella project which provides scalable open-source MQTT client implementations for various languages. The following examples use the Eclipse Paho Java library for the MQTT client.

OBTAINING THE LIBRARY

With Maven: pom.xml

```
.....
<repositories>
    <repository>
        <id>Eclipse Paho Repo</id>
        <url>https://repo.eclipse.org/content/repositories/
paho-releases/</url>
    </repository>
</repositories>
....
<dependencies>
    <dependency>
        <groupId>org.eclipse.paho</groupId>
        <artifactId>org.eclipse.paho.client.mqttv3</
artifactId>
        <version>1.0.2</version>
    </dependency>
</dependencies>
```

With Gradle: build.gradle

```
repositories {
    maven { url 'https://repo.eclipse.org/content/
repositories/paho-releases/' }
}

dependencies {
    compile( [group: 'org.eclipse.paho', name: 'org.eclipse.
paho.client.mqttv3', version: '1.0.2'] )
}
```

PUBLISH A MESSAGE

Publishing messages is straightforward. After connecting, publishing is a one-liner with the publish() method.

```
MqttClient mqttClient = new MqttClient(
        "tcp://broker.mqttdashboard.com:1883",  //1
        "refcard-client");                              //2
mqttClient.connect();
mqttClient.publish(
        "topic",                                //3
        "message".getBytes(),                           //4
        0,                                              //5
        false);                                         //6
mqttClient.disconnect();
```

1. The server URI
2. The MQTT client ID
3. The MQTT topic
4. The payload as byte array
5. The QoS Level
6. Retained Flag

SUBSCRIBE TO TOPICS

To subscribe to topics, an MqttCallback must be implemented. This callback is triggered every time an event (like messageArrived) occurs. This callback must be implemented before connecting to the broker.

```
mqttClient.setCallback(new MqttCallback() {     //1
    @Override
    public void connectionLost(Throwable throwable) {
        //Called when connection is lost.
    }

    @Override
    public void messageArrived(String topic, MqttMessage
         mqttMessage) throws Exception {
        System.out.println("Topic: " + topic);
        System.out.println(new String(mqttMessage.
                getPayload()));
        System.out.println("QoS: " + mqttMessage.
                getQos());
        System.out.println("Retained: " + mqttMessage.
isRetained());
    }

    @Override
    public void deliveryComplete(final IMqttDeliveryToken
iMqttDeliveryToken) {
        //When message delivery was complete
    }
});


mqttClient.connect();

mqttClient.subscribe("my/topic", 2);            //2
```

1. Implement the MqttCallback to process messages which match the subscription

2. Subscribe to a topic with Quality of Service level 2

CONNECTING WITH ADDITIONAL OPTIONS

For more sophisticated MQTT applications, there are additional options for establishing a connection to the broker available.

```
MqttConnectOptions options = new MqttConnectOptions();
options.setCleanSession(true);                          //1
options.setKeepAliveInterval(180);                      //2
options.setMqttVersion(MqttConnectOptions.MQTT_
VERSION_3_1_1);                                         //3
options.setUserName("username");                        //4
options.setPassword("mypw".toCharArray());              //5
options.setWill(
        "will/topic",                                   //6
        "message".getBytes(),                           //7
        1,                                              //8
        true);                                          //9

mqttClient.connect(options);
```

1. If a clean or persistent session should be used
2. Interval in seconds for heartbeats
3. MQTT version (3.1, 3.1.1, or 5.0)
4. Username for authentication
5. Password for authentication
6. Topic for Last Will and Testament
7. Last Will and Testament message
8. Last Will and Testament QoS
9. Last Will and Testament Retained Flag

MQTT OVER WEBSOCKETS

HTML5 WebSocket's provide a full-duplex communication over a TCP connection. Most modern web browsers implement this specification, even on mobile devices. MQTT can be used in conjunction with WebSocket's to allow any web application to behave like a full-featured MQTT client. A library that utilizes WebSocket's for MQTT like the **Paho JavaScript Client** is needed.

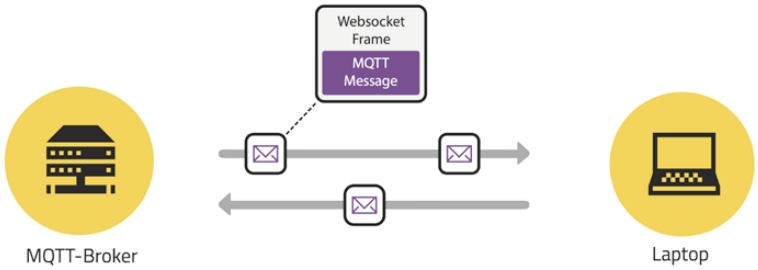

Figure 2: MQTT Over WebSocket's

The advantages of using MQTT in web applications are:

- **Quality of Service semantics:** With QoS 1 and 2, there's an assurance that a message arrives on the client or broker at least once/exactly once, even if the Internet connection dropped in the meantime.
- **Queuing:** When using QoS 1 or 2 and a persistent session, the broker will queue all messages a client misses from its subscriptions when it is not connected. On reconnect, all messages are delivered instantly to that client.
- **Retained messages:** Messages that are retained on the server are delivered instantly when a web application subscribes to one of these topics.
- **Last Will and Testament:** If a client doesn't disconnect gracefully, it's possible to publish a message to a topic to notify all subscribers that the client went off line.

CONNECTING WITH PAHO JAVASCRIPT

A web application can be connected to an MQTT broker easily by using the Paho JavaScript library. Typically, the following code is executed as soon as the page is loaded.

```
var client = new Messaging.Client(hostname, port, clientid);

var options = {
     //connection attempt timeout in seconds
     timeout: 3,

     /*Called if the connection has successfully been
established*/
     onSuccess: function () {
         alert("Connected");
     },
     /*Gets Called if the connection could not be
established*/
     onFailure: function (message) {
         alert("Connection failed: " + message.
errorMessage);
     }
 };

//Connect the MQTT client
client.connect(options);
```

PUBLISHING WITH PAHO JAVASCRIPT

After a connection is established, the client object can be used to publish messages.

```
var message = new Messaging.Message(payload);
     message.destinationName = topic;
     message.qos = qos;
     client.send(message);
```

SUBSCRIBING WITH PAHO JAVASCRIPT

To process messages, a callback is needed for handling each arriving message. After assigning the callback, subscribing to concrete topics is possible.

```
/*Gets called whenever a message is received for a
subscription*/
 client.onMessageArrived = function (message) {
     //Do something with the push message you received
 };
client.subscribe("testtopic", {qos: 2});
```

SCALING MQTT

In a brokered architecture it's critical to avoid a single point of failure and to think about scaling out, since typically only one broker node is used. In the context of MQTT there are two different popular strategies applicable:

BRIDGING

Some brokers implement an unofficial bridging protocol which makes it possible to chain brokers together. Bridging allows forwarding messages on specific topics to other MQTT brokers. Bridge connections between brokers can be uni or bidirectional. Technically, a bridge connection to another broker is a connection where the broker behaves like an MQTT client and subscribes to specific topics.

Pros:

- Great for forwarding messages on specific topics
- Different broker products can be chained
- Hierarchical broker architectures possible

Cons:

- No shared state between brokers
- Bridge protocol is not officially specified

Brokers which implement bridging: HiveMQ, mosquitto, RSMB, WebSphere MQ / IBM MQ.

CLUSTERING

Many enterprise MQTT brokers implement clustering, which supports high availability configurations and allows for scaling out by adding more broker nodes. When a cluster node is no longer available, other cluster nodes can take over so that no data or messages are lost. Often brokers implement elastic clustering, and nodes can be added or removed any time.

Pros:

- High availability and scalability
- MQTT branches across cluster nodes

Cons:

- No standard
- Broker-specific

Brokers which implement clustering: Apache ActiveMQ, HiveMQ, RabbitMQ.

Note: If broker implementation allows, clustering and bridging can be used together, enabling messages from one broker cluster to be forwarded to another isolated cluster.

MQTT AND SECURITY

Security is a very important part of any communication architecture. MQTT itself keeps everything as simple as possible and relies on other proven technologies for safeguards instead of reinventing the wheel.

USERNAME / PASSWORD AUTHENTICATION

An MQTT CONNECT message can contain a username and password. The broker can authenticate and authorize with this information if such a mechanism is implemented. Many open-source brokers rely on Access Control Lists while other enterprise brokers allow coupling with user databases and/or LDAP (Lightweight Directory Access Protocol) systems.

TRANSPORT SECURITY: TLS

A best practice when using MQTT is to add transport layer security if possible. With TLS, the complete communication between client and broker is encrypted, and no attacker can read any message exchanged. If feasible, X509 client certificate authentication adds an additional layer of security to the clients: trust. Some MQTT brokers, like *mosquitto and* HiveMQ, allow the use of X509 certificates in the plugin system for further processing (i.e. authorization).

Note: Cloud based IOT Hubs such as Microsoft Azure and AWS require MQTT data to be packaged with TLS or data will be rejected. Based on my experience, this will also cause errors at the broker and client.

OTHER SECURITY MECHANISMS

Most enterprise MQTT brokers add additional security mechanisms, i.e. a plugin system where concrete logic can be connected. Additional security for MQTT communications can be gained when adding the following to clients / brokers:

- **Payload encryption:** This is application-specific. Clients can encrypt the payload of their PUBLISH messages. The shared secret must be provisioned to all communication participants beforehand.

- **Payload signing:** If the MQTT broker of choice supports intercepting MQTT messages (i.e. with a plugin system), every received message payload can be intercepted and signed with a private key before distributing. The distributed messages can then be verified by the MQTT clients to make sure no one has modified the message.

- **Complex authentication protocols:** For many enterprise MQTT brokers, additional authentication methods can be implemented (i.e. OAuth 2, Kerberos, OpenID Connect, etc.).

- **Authorization / Topic Permissions:** Securing access to topics is often done with a permission concept. Some brokers offer restricting publish / subscribe permissions with a plugin system. This makes sure no one can subscribe to more information than needed, and that only specific clients can publish on specific topics.

ENHANCED EXISTING FEATURES WITH MQTT 5.0 STANDARD

MQTT 5.0 is the most recent MQTT release and was published in December 2017 I believe. While most popular MQTT brokers and MQTT client libraries support MQTT 5.0, some older implementations still use 3.1 and 3.1.1. While 3.1 and 3.1.1 are mainly backwards-compatible, the two versions have subtle differences. However, version 5.0 is not backwards compatible with 3.1.1 or below.

The following features were added to MQTT 3.1.1 and enhanced in 5.0:

- **Session present flag:** If a client connects with a persistent session (which means it doesn't use a clean session), an additional flag was introduced in the CONNACK message to indicate that the broker already has prior session information of the client like subscriptions and queued messages.

- **Error codes on failed subscriptions:** Prior to MQTT 3.1.1, it was impossible for clients to find out if the MQTT broker didn't approve a subscription, which could be the case when using fine-grained permissions for MQTT topics. The new specification changes that and adds a new error (0x80) in the MQTT SUBACK message, so clients can react on forbidden subscriptions.

- **Anonymous MQTT clients:** The MQTT client identifier can be set to zero-byte length. The MQTT broker will assign a random client identifier to the client temporarily.

- **Immediate publishes:** MQTT clients now can send MQTT PUBLISH messages before waiting for a CONNACK response of the MQTT broker.

- **No client identifier restrictions:** MQTT 3.1 had a limit of 23 bytes per client identifier. With the removal of this restriction, client IDs can now use up to 65535 bytes.

Major objectives for MQTT 5.0 were:

- Enhanced scalability and large-scale systems in respect to setup with 1000's and millions of devices.
- Improved error reporting (Reason Code & Reason String).
- Formalized common patterns including capability discovery and request response.
- Extensibility mechanisms including **user properties, payload format, and content type**.
- Performance improvements and improved support for small clients.

The following features were added to MQTT 5.0:

- **User Properties**
  - **User properties on PUBLISH:** are forwarded with the message and are defined by the client applications. They are forwarded by the server to the receiver of the message.
  - **User properties on CONNECT and ACKs:** are defined by the sender, are unique to the sender implementation, and are not defined by MQTT. An unlimited number of user properties can be added!

- **Payload Format Indicator & Content Type:** Another identifier/value pair is available for use when sending a PUBLISH message. This is the Payload Format indicator. If present and set to **1, this indicates that the PUBLISH payload is UTF-8 encoded data**. If set to **0, or if the indicator is not present, then the payload is an unspecified byte format**, exactly as with MQTT v3.1.1.
  - Optional part of the PUBLISH message
  - Reason Codes for ACK messages available inf payload format is invalid
  - Receiver may validate the format indicator
  - "Content Type" optional header can carry a MIME type
  - "Payload Format" indicator can be binary or UTF-8
- **Shared Subscriptions:** Client Load Balancing is now included in MQTT 5.0. The message load of a single topic is distributed amongst all subscribers as shown in the below figure.

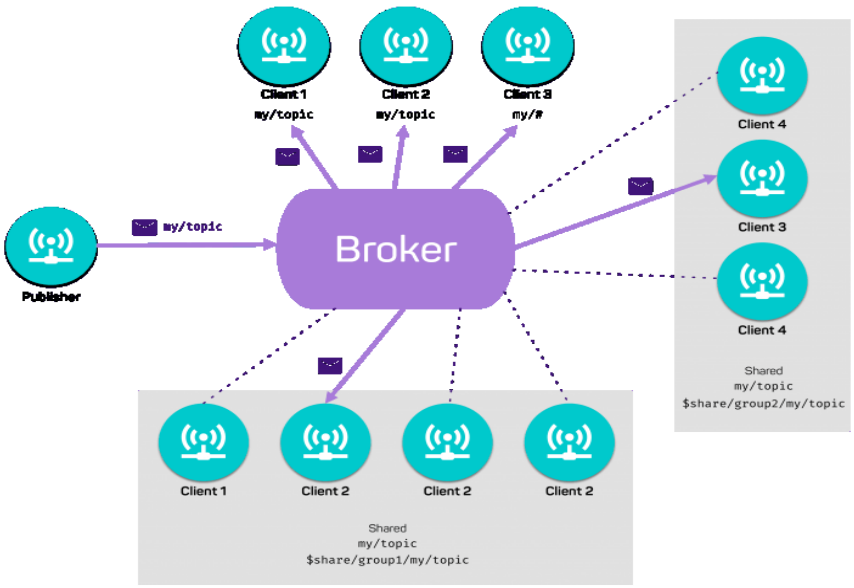

Figure 3: MQTT Shared Subscriptions

Note: All the above information is based on experiences and notes captured during design & development of MQTT Projects within the energy sector. Also, this the information in this guide is directed towards engineers and developers implementing MQTT as part of platform or solution.