Master's in Critical Computing Systems Engineering

# LAB 1 – The MQ Telemetry Transport or is it just MQTT?!

Ricardo Severino
António Barros

## Overview

**MQTT** (originally an initialism of **MQ Telemetry Transport**) is a lightweight, publish-subscribe, machine to machine network protocol for message queue/message queuing service. It is designed for connections with remote locations that have devices with resource constraints or limited network bandwidth. 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).

In this LAB you will be learning about MQTT, setting up a test environment and exploring, hands-on, an MQTT implementation. Particularly, you will be looking into the MQTT QoS levels and connection persistence settings.

At the end you will be asked to write a small (max. 2 pages) report on a topic related to MQTT.

## Setting up your environment

You will be setting up an Xubuntu or Ubuntu Virtual Machine.

1. Install latest stable Oracle VirtualBox in your host system (or similar virtualization software).
2. Download XUBUNTU fresh image or use the one that is handed to you in class.
3. Create a new Virtual Machine and install the OS. We recommend 25 GB disk space on your virtual hard disk.
4. You need to install tools for building the kernel moduleif you want to enjoy guest-additions.

```
$ sudo apt install virtualbox-guest-utils virtualbox-dkms
$ sudo ./VBoxLinuxAdditions.run
```

**While your virtual environment is downloading or being setup read through the next sections.**

# Overview of the MQTT

## Historical Background

Andy Stanford-Clark (IBM) and Arlen Nipper (then working for Eurotech, Inc.) authored the first version of the protocol in 1999. It was used to monitor oil pipelines within the SCADA industrial control system. The goal was to have a protocol that is bandwidth-efficient, lightweight and uses little battery power, because the devices were connected via satellite link which, at that time, was extremely expensive.

Historically, the "MQ" in "MQTT" came from the IBM MQ (then 'MQSeries') product line, where it stands for "Message Queue". However, the protocol provides publish-and-subscribe messaging (no queues, in spite of the name). In the specification opened by IBM as version 3.1 the protocol was referred to as "MQ Telemetry Transport". Subsequent versions released by OASIS strictly refers to the protocol as just "MQTT", although the technical committee itself is named "OASIS Message Queuing Telemetry Transport Technical Committee". Since 2013, "MQTT" does not stand for anything.

In 2013, IBM submitted MQTT v3.1 to the OASIS specification body with a charter that ensured only minor changes to the specification could be accepted. After taking over maintenance of the standard from IBM, OASIS released version 3.1.1 on October 29, 2014. A more substantial upgrade to MQTT version 5, adding several new features, was released on March 7, 2019.

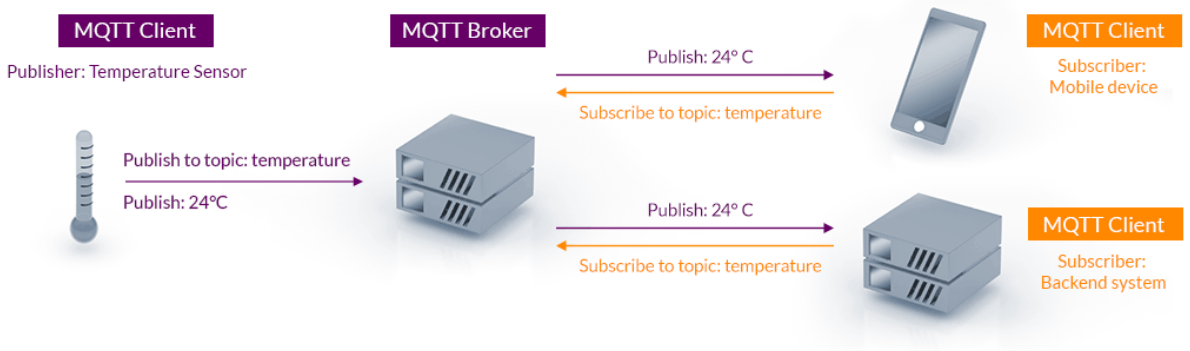
MQTT-SN (MQTT for Sensor Networks) is a variation of the main protocol aimed at battery-powered embedded devices on non-TCP/IP networks, such as Zigbee.

## Overview of MQTT innerworkings

Aimed at maximising the available bandwidth, MQTT's publish/subscribe (pub/sub) communication model is an alternative to traditional client-server architecture that communicates directly with an endpoint e.g. HTTP. By contrast, in the pub/sub model, the client that sends a message (the publisher) is decoupled from the client or clients that receive the messages (or the subscribers). Because **neither the publishers nor the subscribers contact each other directly**, third parties -- the message broker -- take care of the connections between them.

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.[17]

Information is organized in a hierarchy of topics. When a publisher has a new item of data to distribute, it sends a control message with the data to the connected broker. The broker then distributes the information to any clients that have subscribed to that topic. The publisher does not need to have any data on the number or locations of subscribers, and subscribers, in turn, do not have to be configured with any data about the publishers.



If a broker receives a message on a topic for which there are no current subscribers, the broker discards the message unless the publisher of the message designated the message as a retained message. 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 the selected 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. This allows new subscribers to a topic to receive the most current value rather than waiting for the next update from a publisher.

When a publishing client first connects to the broker, it can set up a default message to be sent to subscribers if the broker detects that the publishing client has unexpectedly disconnected from the broker.

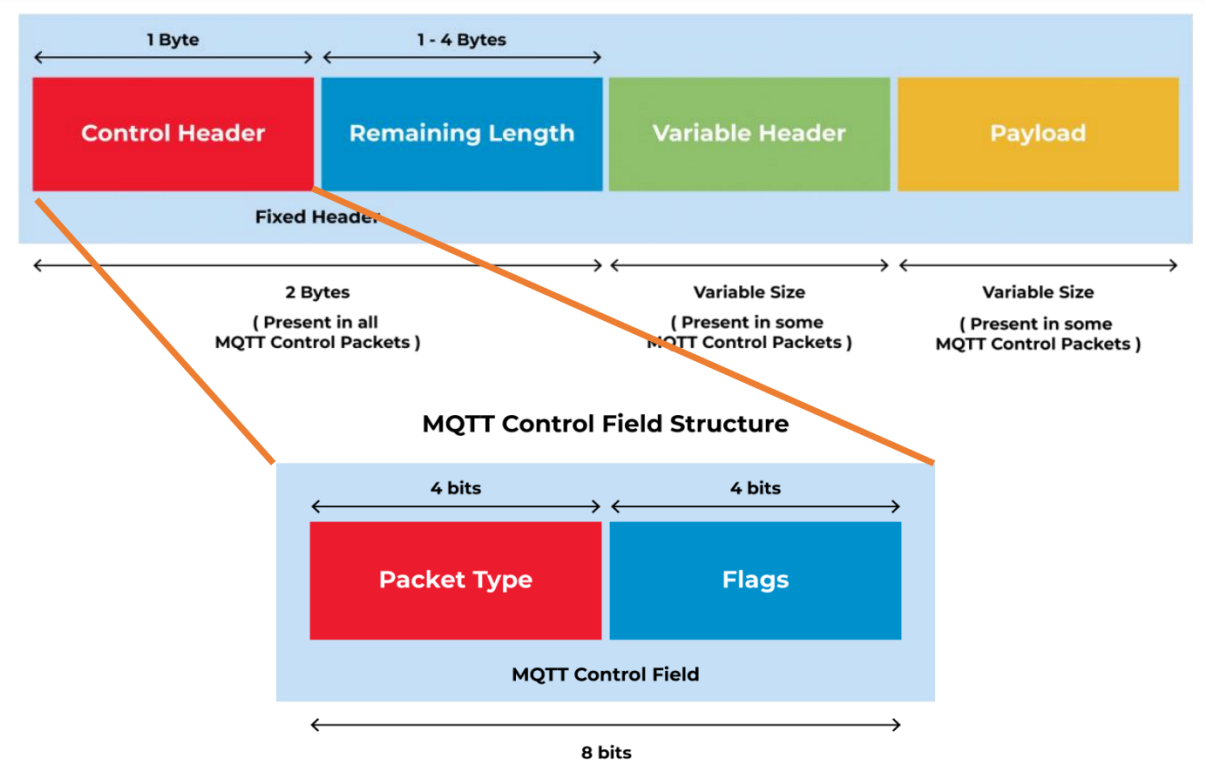
Clients only interact with a broker, but a system may contain several broker servers that exchange data based on their current subscribers' topics.

MQTT sends connection credentials in plain text format and does not include any measures for security or authentication. This can be provided by using TLS to encrypt and protect the transferred information against interception, modification or forgery.

The default unencrypted MQTT port is 1883. The encrypted port is 8883.

## Message Details

A minimal MQTT control message can be as little as two bytes of data. A control message can carry nearly 256 megabytes of data if needed.




However, MQTT supports message binary large objects (BLOBs) up to 256 MB in size. The format of the content will be application-specific. Topic subscriptions are made using a SUBSCRIBE/SUBACK packet pair, and unsubscribing is similarly performed using an UNSUBSCRIBE/UNSUBACK packet pair.

There are fourteen defined message types used to connect and disconnect a client from a broker, to publish data, to acknowledge receipt of data, and to supervise the connection between client and server, as follows:

Message Type:

| Message type | Value | Description | Fixed Header |
|---|---|---|---|
| Reserved | 0 | Reserved | Present |
| CONNECT | 1 | Client connect request to server or broker | Present |
| CONNACK | 2 | Connect request acknowledgment | Present |
| PUBLISH | 3 | Publish message | Present |
| PUBACK | 4 | Publish acknowledgment | Present |
| PUBREC | 5 | Publish receive | Present |
| PUBREL | 6 | Publish release | Present |
| PUBCOMP | 7 | Publish complete | Present |
| SUBSCRIBE | 8 | Client subscribe request | Present |
| SUBACK | 9 | Subscribe request acknowledgment | Present |
| UNSUBSCRIBE | 10 | Unsubscribe request | Present |
| UNSUBACK | 11 | Unsubscribe acknowledgment | Present |
| PINGREQ | 12 | PING request | Present |
| PINGRESP | 13 | PING response | Present |
| DISCONNECT | 14 | Client is disconnecting | Present |
| Reserved | 15 | Reserved | Present |

Topic strings form a natural topic tree with the use of a special delimiter character, the forward slash (/). A client can subscribe to -- and unsubscribe from -- entire branches in the topic tree with the use of special wildcard characters. There are two wildcard characters: a single-level wildcard character, the plus character (+); and a multilevel wildcard character, the hash character (#). A special topic character, the dollar character ($), excludes a topic from any root wildcard subscriptions. Typically, $ is used to transport server-specific or system messages.

Another operation a client can perform during the communication phase is to ping the broker server using a PINGREQ/PINGRESP packet sequence. This packet sequence roughly translates to ARE YOU ALIVE/YES I AM ALIVE. This operation has no other function than to maintain a live connection and ensure the TCP connection has not been shut down by a gateway or router.

When a publisher or subscriber wants to terminate an MQTT session, it sends a **DISCONNECT** message to the broker and then closes the connection. This is called a graceful shutdown because it gives the client the ability to easily reconnect by providing its client identity and resuming where it left off.

Should the disconnect happen suddenly without time for a publisher to send a DISCONNECT message, the broker may send subscribers a message from the publisher that the broker has previously cached. The message, which is called a last will and testament, provides subscribers with instructions for what to do if the publisher dies unexpectedly.

MQTT relies on the TCP protocol for data transmission. A variant, MQTT-SN, is used over other transports such as UDP or Bluetooth.

## Benefits and Drawbacks of MQTT

What are the benefits of using MQTT?

The lightweight properties and minimum overhead of the MQTT protocol architecture help ensure smooth data transfer with low bandwidth and reduce the load on the CPU and RAM. Among MQTT's advantages over competing protocols are the following:

- efficient data transmission and quick to implement, due to its being a lightweight protocol;
- low network usage, due to minimized data packets;
- efficient distribution of data;
- successful implementation of remote sensing and control;
- fast, efficient message delivery;
- uses small amounts of power, which is good for the connected devices; and
- optimizes network bandwidth.

What are the drawbacks of MQTT?

Potential downsides to MQTT include the following:

- MQTT has slower transmit cycles compared to Constrained Application Protocol (CoAP).
- MQTT's resource discovery works on flexible topic subscription, whereas CoAP uses a stable resource discovery system.
- MQTT is unencrypted. Instead, it uses TLS/SSL (Transport Layer Security/Secure Sockets Layer) for security encryption.
- It is difficult to create a globally scalable MQTT network.
- Other MQTT challenges relate to security, interoperability and authentication.

## Competing protocols

Other transfer protocols that compete with MQTT include the following:

- Constrained Application Protocol (CoAP). Well suited for IoT, it uses a request/response communication pattern.
- Advanced Message Queuing Protocol (AMQP). Like MQTT, it uses a publish/subscribe communication pattern.
- Simple/Streaming Text Oriented Messaging Protocol (STOMP). It is a text-based protocol. However, STOMP does not deal with queues and topics; it uses a send semantic with a destination string.
- Simple Media Control Protocol (SMCP). A CoAP stack that is used in embedded environments, is C-based.
- SSI (Simple Sensor Interface). This is a communications protocol for data transfer between a combination of computers and sensors.
- Data Distribution Service (DDS). For real-time systems, it is a middleware standard that can directly publish or subscribe communications in real time in embedded systems.

## QoS Levels

Each connection to the broker can specify a quality of service (QoS) measure. These are classified in increasing order of overhead:

- At most once – the message is sent only once and the client and broker take no additional steps to acknowledge delivery (fire and forget).
- At least once – the message is re-tried by the sender multiple times until acknowledgement is received (acknowledged delivery).
- Exactly once – the sender and receiver engage in a two-level handshake to ensure only one copy of the message is received (assured delivery).

This field does not affect handling of the underlying TCP data transmissions; it is only used between MQTT senders and receivers.

The simplest QoS level is unacknowledged service. This QoS level uses a PUBLISH packet sequence; the publisher sends a message to the broker one time, and the broker passes the message to subscribers one time. There is no mechanism in place to make sure the message has been received correctly, and the broker does not save the message. This QoS level may also be referred to as at most once, QoS0 or fire and forget.

The second QoS level is acknowledged service. This QoS level uses a PUBLISH/PUBACK packet sequence between the publisher and its broker, as well as between the broker and subscribers. An acknowledgment packet verifies that content has been received, and a retry mechanism will send the original content again if an acknowledgment is not received in a timely manner. This may result in the subscriber receiving multiple copies of the same message. This QoS level may also be referred to as at least once or QoS1.

The third QoS level is assured service. This QoS level delivers the message with two pairs of packets. The first pair is called PUBLISH/PUBREC, and the second pair is called PUBREL/PUBCOMP. The two pairs ensure that, regardless of the number of retries, the message will only be delivered once. This QoS level may also be referred to as exactly once or QoS2.

## Persistency & Retained Messages Persistence

Using a persistent connection, the broker will store subscription information, and undelivered messages for the client. In order for the broker to store session information for a client, a client id must be used. When a client connects to the broker, it indicates whether the connection is persistent or not, by setting the **'clean-session' flag to FALSE or TRUE** respectively, in the CONNECT packets.

In command-line, clean session is the default. In the mosquitto_sub command, add flag '-c' ('--disable-clean-session') for a persistent connection.

It is important to realise that not all messages will be stored for delivery, as the quality of service, of the subscriber and publisher has an effect.

Retained Messages

If a publisher publishes a message to a topic and no one is subscribed to that topic, the message is simply discarded by the broker. However the publisher can tell the broker to keep the last message on that topic by setting the "retained" message flag. **Only one message is retained per topic. The next message published on that topic replaces the last retained message for that topic.**

## Working with MQTT

Go back to your virtual machine and install mosquitto broker and client.

$ sudo apt install mosquitto mosquitto-clients

Configure MQTT Broker (do not forget to sudo):

In /etc/mosquitto/mosquitto.conf, add

allow_anonymous true

listener 1883

Restart the service

$ sudo service  mosquitto stop

$ sudo pkill mosquitto

$ sudo mosquitto -v -c /etc/mosquitto/mosquitto.conf

open another terminal to monitor the service, and do:

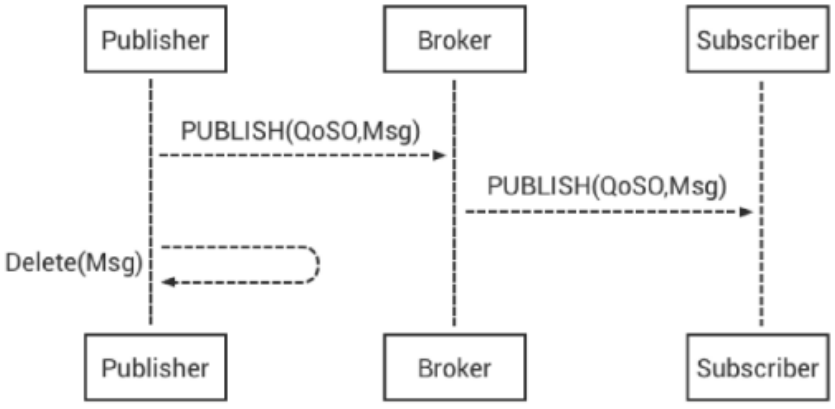
$ sudo tail –f /var/log/mosquitto/mosquitto.log

### Test different persistence settings and its relation to QoS level.

**Common steps: 1. Initialise the client, requesting or not a 'clean_session'. 2. Subscribe to a topic with specific QoS. 3. Disconnect. 4. Publish to the topic that the client subscribed to with QOS set. 5. Reconnect client 6. Make note of any messages received.**

Open wireshark and configure it to use the loopback device. Apply the "mqtt" filter. Start capturing.

Analyse the following protocol diagram.




QoS0 – *Fire and Forget*

*While carrying out the following tests, look at wireshark output to identify the protocol transactions.*

**Test 1 – QoS=0 & no Persistence (Publish / Subscribe QoS: 0)**

Notice that since '-c' flag is omitted, default behaviour is non-persistent connection.

Open two new terminals:

Start Sub:

mosquitto_sub -h localhost -i Sub1 -t "test/message" -q 0

Disconnect.

Start Pub:

mosquitto_pub -h localhost -t "test/message" -m "MESSAGE 1" -q 0

Restart Sub. Did you receive anything? Why?

**Test 2 – QoS=0 & Persistence** ('-c' flag indicates persistent connection)

Start Sub:

mosquitto_sub -h localhost -i Sub1 -t "test/message" -q 0 **-c**

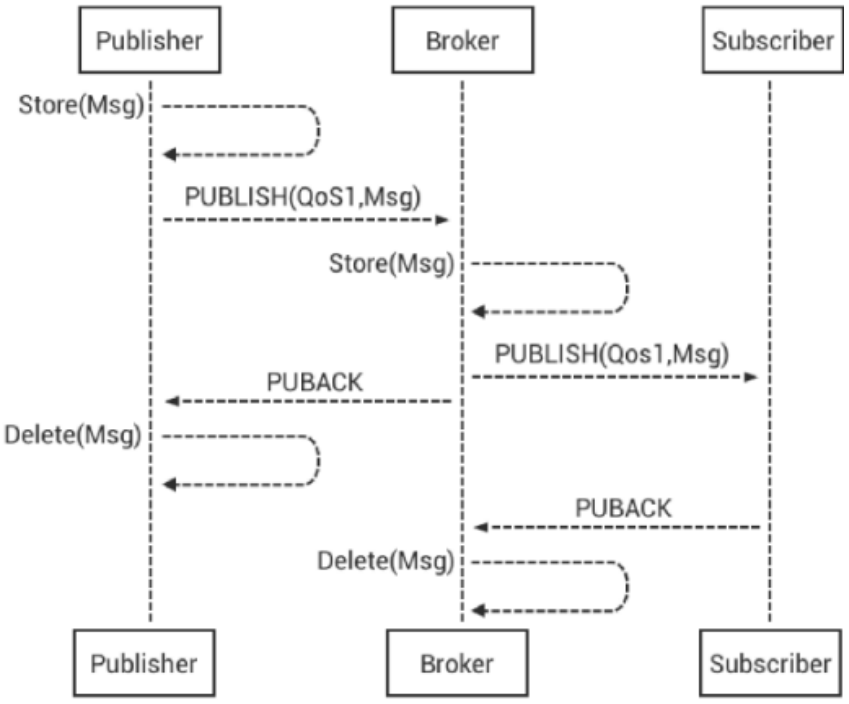
Disconnect.

Start Pub:

mosquitto_pub -h localhost -t "test/message" -m "MESSAGE 2" -q 0

Did it work now? What can be happening?

**Analyse the following diagram.**




QoS1 - At least once

**Test 3 – QoS=1 & Persistence**

Start Sub: mosquitto_sub -h localhost -i Sub1 -t "test/message" -q 1 -c

Disconnect.

Start Pub: mosquitto_pub -h localhost -t "test/message" -m "MESSAGE 3" -q 1

Reconnect Sub. Analyse the result. What changed and why?

**Test 4 – Different Pub/Sub QoS**

Start Sub: mosquitto_sub -h localhost -i Sub1 -t "test/message" -q 0 -c

Disconnect.

Start Pub: mosquitto_pub -h localhost -t "test/message" -m "MESSAGE 4" -q 1

Reconnect Sub. Analyse the results.

**Test 5 – Different Pub/Sub QoS**

Start Sub: mosquitto_sub -h localhost -i Sub1 -t "test/message" -q 0 -c

Disconnect.

Start Pub: mosquitto_pub -h localhost -t "test/message" -m "MESSAGE 4" -q 1

Reconnect Sub. Analyse the results.

## Going beyond

Security was not deeply addressed in this LAB. However, it is critical for most systems. To go beyond this LAB, explore the security details of the protocol and write a short report (max. 2 pages). Reason about security considerations. Understand if security of the MQTT protocol has been compromised by checking CVE. Discuss any exploit you may find.

**Credits**

**Wikipedia on MQTT,** https://en.wikipedia.org/wiki/MQTT

MQTT.org https://mqtt.org/

Techtarget MQTT, https://www.techtarget.com/iotagenda/definition/MQTT-MQ-Telemetry-Transport

Steves internet guide on MQTT,
http://www.steves-internet-guide.com/mqtt-clean-sessions-example/