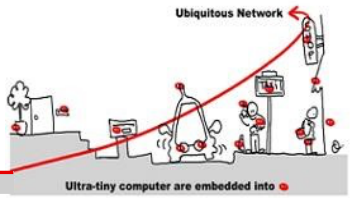
Tutorial: MQTT (Message Queuing Telemetry Transport)



J.-Y. Tigli, G. Rocher, I3S – University of Nice Sophia Antipolis, 2021-2022
Middleware for Internet of Things - MQTT

1 MQTT introduction

MQTT is a M2M lightweight, bandwidth and power-efficient publish/subscribe messaging protocol. It is useful for use with constrained infrastructures/devices (IoT) but is applicable to many scenarios.

1.1 Publish/Subscribe

The MQTT protocol is based on the principle of publishing messages and subscribing to topics, or "pub/sub". Multiple clients can connect to a broker and subscribe to topics that they are interested in (Figure 1). Clients also connect to the broker and publish messages to topics. Many clients can subscribe to the same topics and do whatever they want with the information. The broker acts as a simple, common interface for everything to connect to.

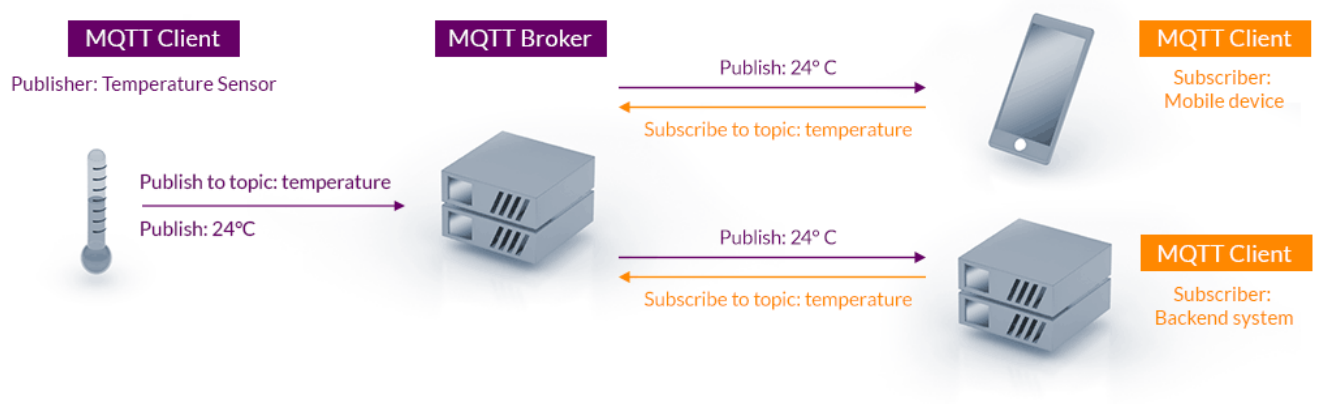


Figure 1 : source https://mqtt.org

1.2 Topics/Subscriptions

Messages in MQTT are published on topics. There is no need to configure a topic, publishing on it is enough. Topics are treated as a hierarchy, using a slash (/) as a separator. This allows sensible arrangement of common themes to be created, much in the same way as a filesystem. For instance, multiple computers may all publish their hard drive temperature information on the following topic, with their own computer and hard drive name being replaced as appropriate:

```
sensors/<COMPUTER_NAME>/temperature/<HARDDRIVE_NAME>
```

Clients can receive messages by subscripting to topics. A subscription may concern an explicit topic, in which case only messages relative to that topic will be received, or it may include wildcards. Two wildcards are available, + or #.

+ can be used as a wildcard for a single level of hierarchy. It could be used with the topic above to get information on all computers and hard drives as follows:

```
sensors/+/temperature/+
```

As another example, for a topic of "`a/b/c/d`", the following example subscriptions will match:

```
a/b/c/d     +/b/c/d     a/+/c/d     a/+/+/d     +/+/+/+
```

The following subscriptions will not match:

```
a/b/c     b/+/c/d     +/+/+
```

can be used as a wildcard for all the remaining levels of the hierarchy. **This means that it must be the final character in a subscription**. With a topic of "`a/b/c/d`", the following example subscriptions will match:

```
a/b/c/d     #     a/#     a/b/#     a/b/c/#     +/b/c/#
```

Zero length topic levels are valid, which can lead to some slightly non-obvious behaviours. For instance, a topic of "`a//topic`" would correctly match against a subscription of "`a/+/topic`". Likewise, zero length topic levels can exist at both the beginning and the end of a topic string, so "`/a/topic`" would match against a subscription of "`+/a/topic`", "`#`" or "`/#`", and a topic "`a/topic/`" would match against a subscription of "`a/topic/+`" or "`a/topic/#`".

1.3 Quality of Service

MQTT defines three levels of Quality of Service (QoS). The QoS defines how hard the broker/client will try to ensure that a message is received. Messages may be sent at any QoS level, and clients may attempt to subscribe to topics at any QoS level. This means that the client chooses the maximum QoS it will receive. For instance, if a message is published at QoS 2 and a client is subscribed with QoS 0, the message will be delivered to that client with QoS 0. If a second client is also subscribed to the same topic, but with QoS 2, then it will receive the same message but with QoS 2. For a second example, if a client is subscribed with QoS 2 and a message is published on QoS 0, the client will receive it on QoS 0.

Higher levels of QoS are more reliable but involve higher latency and have higher bandwidth requirements.

0: The broker/client will deliver the message once, with no confirmation.

1: The broker/client will deliver the message at least once, with confirmation required.

2: The broker/client will deliver the message exactly once by using a four steps handshake.

2 MQTT Brokers providers in the Cloud

Like for Cloud services, some Cloud providers provide some more or less free MQTT brokers. The most famous ones are cloudMQTT, Flepsi, MaQiaTTo, IBM Bluemix, Heroku, HiveMQ and Microsoft Azure IoT.

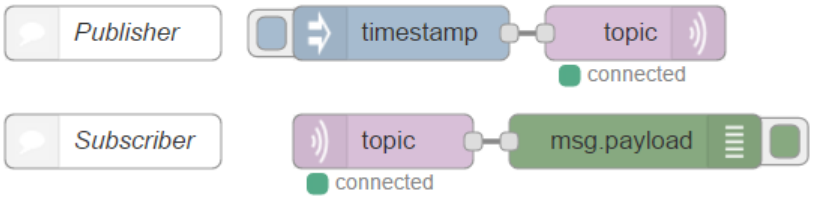
Exercise 1 : Use one of the below public free to use MQTT broker and test publish/subscribe with MQTT.fx https://mqttfx.jensd.de/ (username/password might not be necessary).

Broker	Serveur	Ports	Websocket	Versions
emqx.io	broker.emqx.io	1883 or 8883 (TCP / TLS)	8083 or 8084 (TLS)	
flespi	mqtt.flespi.io	**443 (SSL)** or **80 (non-SSL)**	**8883 (SSL)** ou **1883 (non-SSL)**	**3.1, 3.1.1, 5.0**
HiveMQ MQTT Dashboard	broker.hivemq.com broker.mqttdashboard.com	1883	8000	
MoQiatto	maqiatto.com Nécessite l'ouverture d'un compte gratuit	1883	3883 Certificat SSL	
Mosquitto	iot.eclipse.org test.mosquitto.org	1883 / 8883 1883 / 8883 / 8884	n/a 8080 / 8081	
Mosca	test.mosca.io	1883	80	

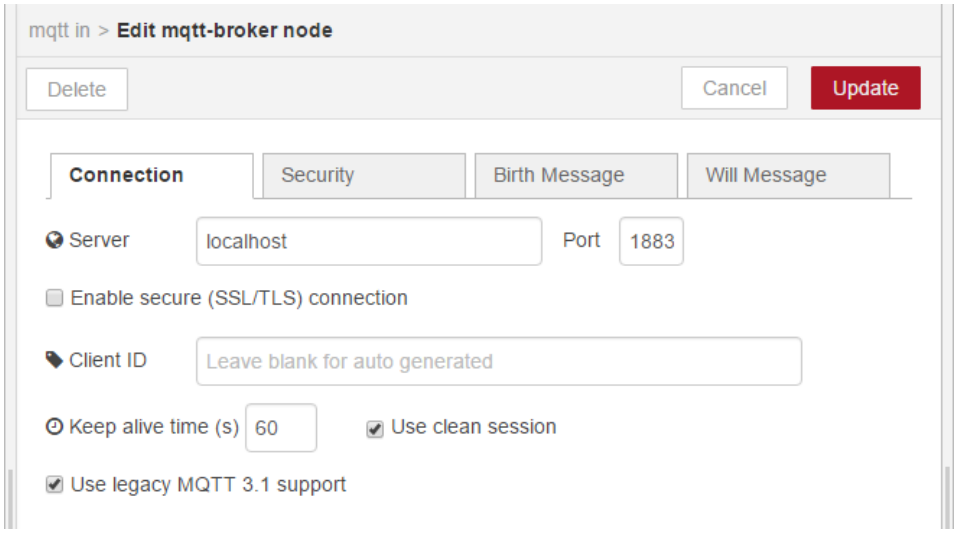
Figure 2 source : https://projetsdiy.fr/8-brokers-mqtt-objets-connectes-cloud/

2 MQTT Programming with NodeRed

Node-Red provides MQTT Client nodes to enable access to MQTT Broker in a node assembly. Once you just put this node on Node-RED and hit deploy button, MQTT Broker will run on your Node-RED.



You can set "localhost" in MQTT-in and MQTT-out properties as follows.

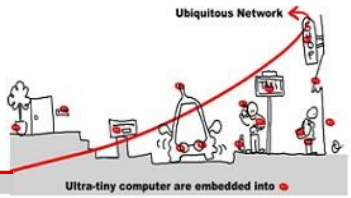


In the properties of the MQTT node, you must at least configure the server and port of the broker you want to reach (as you did when using MQTT.fx), the username and password for authorization.

Exercise 2 : Publish and Subscribe on the public free to use MQTT broker used in the previous exercise from a Node-Red application. Details are provided in [Configuring the MQTT Publish and Subscribe Nodes in Node-Red].

3 Install your own MQTT broker : Mosquitto

Mosquitto is an open source (BSD licensed) message broker that implements the MQTT protocol version 3.1. MQTT provides a lightweight method of carrying out messaging using a publish/subscribe model. This makes it suitable for "machine-to-machine" messaging such as with low power sensors or mobile devices such as phones, embedded computers or microcontrollers like the Arduino.



3.1 Linux Ubuntu Computer

In the first part of this tutorial. You need to boot your computer on Linux or use a VMware workstation with Ubuntu (See Appendix)

3.2 Installing Mosquitto

Mosquitto can be simply installed like a linux/ubuntu package

Exercise 3 : Install mosquitto from your package manager.

```
sudo apt-get install mosquitto
```

Don't forget to install also shell script commands clients

```
sudo apt-get install mosquitto-clients
```

3.3 The server

The server listens on the following ports:

1883 : MQTT, unencrypted

8883 : MQTT, encrypted

8884 : MQTT, encrypted, client certificate required

8080 : MQTT over WebSockets, unencrypted

8081 : MQTT over WebSockets, encrypted

The encrypted ports support TLS v1.2, v1.1 or v1.0 with x509 certificates and requires client support to connect. In all cases you should use the certificate authority file mosquitto.org.crt to verify the server connection. Port 8884 requires clients to provide a certificate to authenticate their connection. If you wish to obtain a client certificate, please get in touch.

3.4 MQTT/Mosquitto Man pages and commands

For more information on MQTT, see http://mqtt.org/ or the Mosquitto MQTT man page: http://mosquitto.org/man/

2.5.1 mosquitto — an MQTT broker

```
mosquitto [-c config file] [ -d | --daemon ] [-p port number] [-v]
```

Exercise 4 : Test your Mosquitto broker using MQTT.fx and Node-Red programming (Mosquitto broker can be started with `sudo service mosquitto start`).

3.4.2 mosquitto_pub

```
mosquitto_pub [-A bind_address] [-d] [-h hostname] [-i client_id] [-I client id
prefix] [-k keepalive time] [-p port number] [-q message QoS] [--quiet] [-r] [-S] {
-f file | -l | -m message | -n | -s } [ [-u username] [-P password] ] [ --will-topic
topic [--will-payload payload] [--will-qos qos] [--will-retain] ] [[ { --cafile
file | --capath dir } [--cert file] [--key file] [--ciphers ciphers] [--tlsversion
version] [--insecure] ] | [ --psk hex-key --psk-identity identity [-ciphers
ciphers] [--tls-version version] ]] [--proxy socks-url] [-V protocolversion] -t
message-topic
```

Exercise 5 : Test these examples using 2 console terminals: one for publishers and one for subscribers.

1. Publish temperature information to localhost with (with the broker up & running on localhost or using online broker e.g., `mosquitto_pub -h broker.emqx.io -p 1883…`):

   ```
   mosquitto_pub -t -h 127.0.0.1 -p 1883 sensors/temperature -m 32
   ```

2. Publish timestamp and temperature information on a nonstandard port (here 1885) and QoS 1:

   ```
   mosquitto_pub -h 127.0.0.1 -p 1885 -t sensors/temperature -m "1266193804 32" -q 1
   ```

3. Publish light switch status. Message is set to retain (option r) because there may be a long period of time between light switch events or intermittent connections to the broker:

   ```
   mosquitto_pub -h 127.0.0.1 -p 1883 -r -t switches/kitchen_lights/status -m "on"
   ```

   You can try disconnecting subscriber from the broker and publish different values. Then, resubscribe to the topic. What happens?

4. Send the contents of a file in two ways:

   ```
   mosquitto_pub -t my/topic -f ./data
   mosquitto_pub -t my/topic -s < ./data
   ```

3.4.3 mosquitto_sub

```
mosquitto_sub [-A bind_address] [-c] [-C msg count] [-d] [-h hostname] [i
client_id] [-I client id prefix] [-k keepalive time] [-p port number] [-q
message QoS] [-R] [-S] [-N] [--quiet] [-v] [ [-u username] [-P password] ]
[ --will-topic topic [--will-payload payload] [--will-qos qos] [--will-
retain] ] [[ { --cafile file | --capath dir } [--cert file] [--key file]
[--tls-version version] [--insecure] ] | [ --psk hex-key -psk-identity
identity [--tls-version version] ]] [--proxy socks-url] [-V protocol-
version] [-T filter-out...] -t message-topic...
```

Exercise 6 : Test these examples :

Subscribe to temperature information on localhost with QoS 1:

```
mosquitto_sub -t sensors/temperature -q 1
```

Subscribe to hard drive temperature updates on multiple machines/hard drives. This expects each machine to be publishing its hard drive temperature to `sensors/machines/HOSTNAME/temperature/HD_NAME`.

```
mosquitto_sub -t sensors/machines/+/temperature/+
```

Subscribe to all broker status messages:

```
mosquitto_sub -v -t \$SYS/#
```

4 Advanced MQTT Tutorial

4.1 MQTT Client in C# (can be done in Python or java or …)

First we need to install M2Mqtt for .Net https://m2mqtt.wordpress.com/. The better way to do so is to use the "Nuget package manager console" in Visual Studio to download M2Mqtt as a nugget package. See https://www.nuget.org/packages/M2Mqtt/.

See https://github.com/eclipse/paho.mqtt.m2mqtt for examples and setup a Visual Studio Project: "Visual C#", "Application Console Win32" (empty project) to test the API.

BE CAREFUL!
Your console project on Visual studio must be developed for .Net Framework 4.5 because of the m2mqtt package dependencies.

Exercise 7 : Implement a C# publisher client and test it through the Mosquitto MQTT broker
Exercise 8 : Implement a C# subscriber client and test it on your Mosquitto MQTT broker

4.2 MQTT and Complex event processing (CEP)

Event processing is a method of tracking and analysing (processing) streams of information (data) about things that happen (events) and deriving a conclusion from them. Complex event processing (CEP) is an event processing that combines data from multiple sources to infer events or patterns that suggest more complicated events. The goal of complex event processing is to identify meaningful events (such as opportunities or threats) and respond to them as quickly as possible.

Figure 3 : https://fr.wikipedia.org/wiki/Traitement_des_événements_complexes

For these exercises, one can use the C# MQTT clients you developed or develop a Node-RED flow.

Exercise 9 : Implement a simple example that uses one publisher that publishes events as integer values X, one subscriber waiting for the events followed by a complex event processing that transform value X to an event sin(X) and publish it.

One of the applications of CEP is CED (Composite Event Detection). CED is a way to implement multiple conditions on the basis of different events to emit new events. For instance, Event Condition Action (ECA) is a short-cut referring to the structure of active rules in event driven architecture and active database systems. Such rules are defined as follows:

- The EVENT part specifies the signal that triggers the invocation of the rule,
- The CONDITION part is a logical test that, if satisfied, causes the action to be carried out,
- The ACTION part consists of updates, invocations of particular methods, etc.

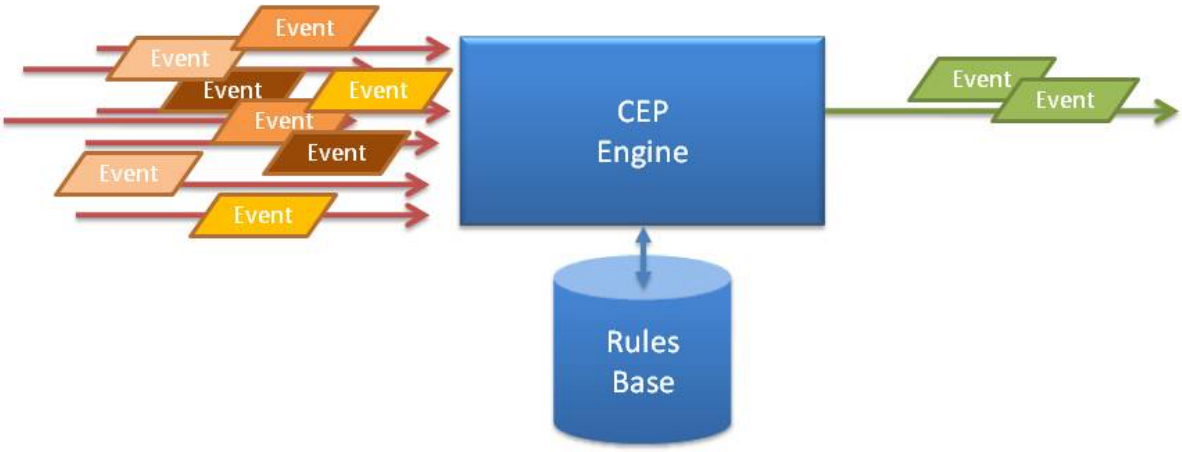
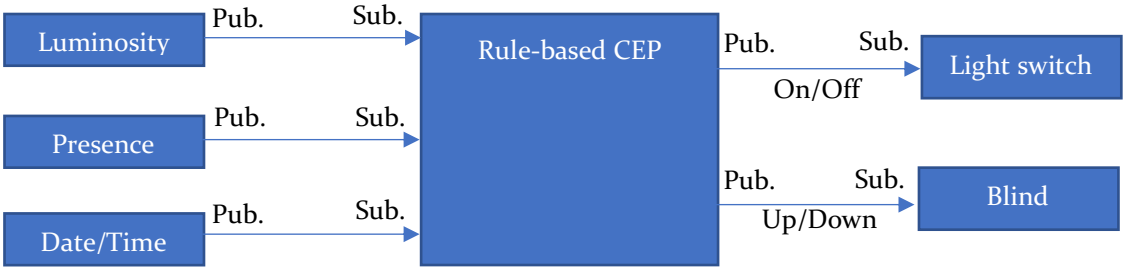


Figure 4 : https://fr.wikipedia.org/wiki/Traitement_des_événements_complexes

Exercise 10 : Implement a CED process between publisher and subscriber clients to apply a set of ECA rules on input event occurrences and, after evaluation of the conditions, emit new events as actions (it could be commands,...). One can think implementing ECA rules as a Finite State Machine (FSM). For instance, one can get information on luminosity, presence, hours, etc... and may need to send commands to light switches and blinds in order to ensure a constant luminosity value to users in the room. For those of you using Node-RED, there is a node dedicated to specify FSM (https://flows.nodered.org/node/node-red-contrib-fsm).



This is an example; one can propose any other (more complicated/relevant) one...

Exercise 11: What are the advantages of using an Event-based middleware here?