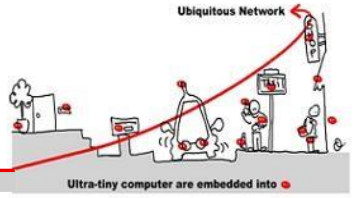
Tutorial: MQTT (Message Queuing Telemetry Transport)



J.-Y. Tigli, G. Rocher, I3S – University of Nice Sophia Antipolis, 2018-2019
Middleware for Internet of Things - MQTT

1 MQTT introduction :

MQTT is a lightweight publish/subscribe messaging protocol. It is useful for use with low power sensors 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 connect to a broker and subscribe to topics that they are interested in. Clients also connect to the broker and publish messages to topics. Many clients may subscribe to the same topics and do with the information as they please. The broker and MQTT act as a simple, common interface for everything to connect to.

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 example, 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 creating subscriptions. A subscription may be to an explicit topic, in which case only messages 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 remaining levels of 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 behavior. For example, 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 example, 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 step handshake.

2 MQTT Brokers providers in the Cloud

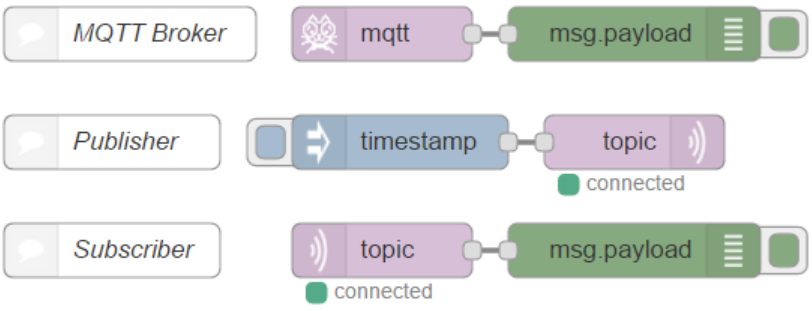
Like for Cloud services, some Cloud providers provide some more or less free MQTT brokers. Most famous are ThingMQ, ThingStudio, cloudMQTT, IBM Bluemix, Heroku, Hivemq, Microsoft Azure IoT, MQTT.io.

Exercice 1 : Create an account on CloudMQTT https://www.cloudmqtt.com/ and test your access with MQTT.fx https://mqttfx.jensd.de/ .

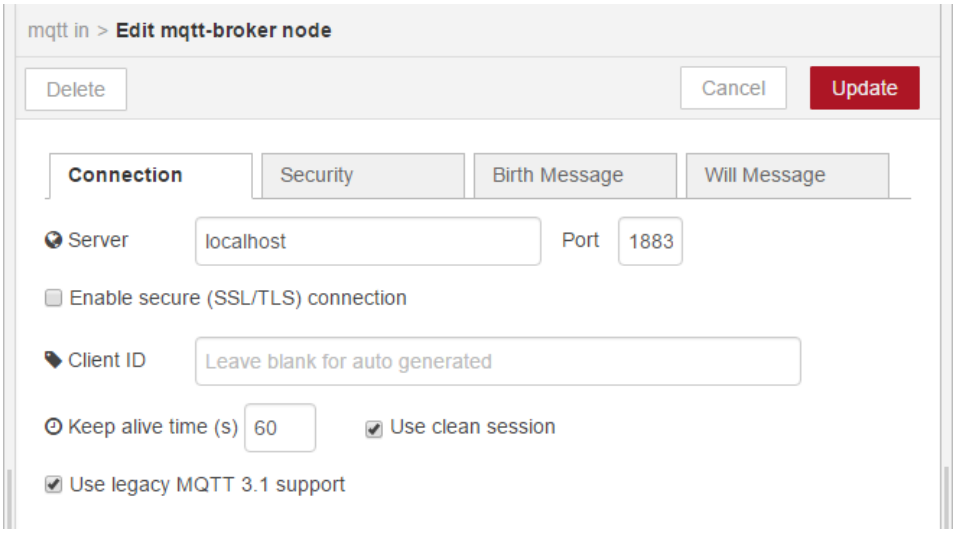
2 MQTT Programming with NodeRed

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 set of the MQTT node, you must at least configure the server and port of the broker you want to reach, the username and password for authorization.

Exercice 2 : Publish and Subscribe on the CloudMQTT broker 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 MQ Telemetry Transport 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

Exercice 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 require 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 it touch.

You are free to use it for any application, but please do not abuse or rely upon it for anything of importance. You should also build your client to cope with the broker restarting.

Please don't publish anything sensitive, anybody could be listening.

3.4 Caveats

This server is provided as a service for the community to do testing, but it is also extremely useful for testing the server. This means that it will often be running unreleased or experimental code and may not be as stable as you might hope. It may also be. Finally, not all of the features may be available all of the time, depending on what testing is being done. In particular, websockets and TLS support are the most likely to be unavailable.

In general, you can expect the server to be up and to be stable though.

3.5 MQTT/Mosquitto Man pages and commands

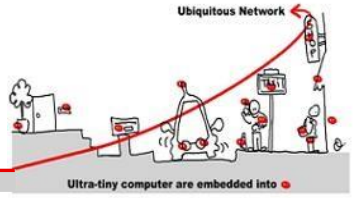
For more information on MQTT, see http://mqtt.org/ or the Mosquitto MQTT man page: http://mosquitto.org/man/

2.5.1 mosquitto — an MQTT broker

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

3.5.2 mosquitto_pub

Tutorial: MQTT (Message Queuing Telemetry Transport)




```
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
```

Exercice 3 : Test your Mosquitto broker using MQTT.fx and Node-Red programming

Exercice 4 : Test these examples using to console terminals: one for publishers and one for subscribers.

1. Publish temperature information to localhost with QoS 1:
   - mosquitto_pub -t sensors/temperature -m 32 -q 1
2. Publish timestamp and temperature information to a remote host (here localhost for test) on a nonstandard port (here the standard one: 1883 for test !) and QoS 0:
   - mosquitto_pub -h 127.0.0.1 -p 1883 -t sensors/temperature -m "1266193804 32"
3. Publish light switch status. Message is set to retain because there may be a long period of time between light switch events:
   - mosquitto_pub -r -t switches/kitchen_lights/status -m "on"
4. Send the contents of a file in two ways:
   - mosquitto_pub -t my/topic -f ./data
   - mosquitto_pub -t my/topic -s < ./data

Exercice 5 : intermittent network connections management

Because the mosquito broker is local for these tests, we can illustrate the management of intermittent connection with MQTT stop and start. To test that you can use these shell commands:

```
Sudo service mosquitto stop
```

```
Sudo service mosquitto start
```

What happens?

3.5.3 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...
```

Exercice 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/#

Exercice 7 : Subscribe to the "temp/random" and see what happen as soon temp/random changes :

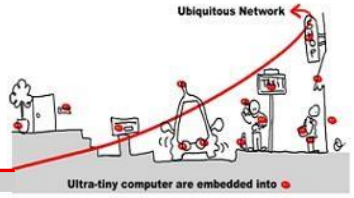
Exercice 8 : Write a temperature profile in a file and publish it with time stamps. Trace in another file, the change of the temperature through the client/subscribe.

Exercice 9 : Display and compare both files with gnuplot (install gnuplot if necessary).

BE CAREFUL!

To access to a standard remote server broker like test.mosquitto.org, the Mosquitto port mustn't be filtered by your Internet Gateway.

If you don't find better solution, use a Smart Phone as an internet access point 4G/Wi-Fi. That works!



BE CAREFUL!

If you are working on Mac OS X, you must run mosquito explicitly like that:

```
/usr/local/sbin/mosquitto  -c /usr/local/etc/mosquitto/mosquitto.conf
```

4 Advanced MQTT Tutorial

Everybody from MQTT beginner to expert needs a handy tool to try out stuff or for debugging. The site (http://www.hivemq.com/blog/seven-best-mqtt-client-tools) gives a brief overview of the best MQTT client tools for different platforms and highlight special features.

Exercice 10 : Install the last stable version of MQTT.fx on Windows and configure it to connect the remote Mosquitto MQTT broker that you installed. Use the graphical user interface to publish and subscribe events.

This tool will be useful for the following.

4.1 Docker Architecture for MQTT

To facilitate MQTT application and brokers deployment, all of these can be implemented using Docker containers.

Exercice 11 : Install in docker containers, a MQTT broker (Eclypse Mosquitto) container, and a Node-Red Container. Test this deployment with past examples.

4.2 MQTT Client Programming

4.2.2 MQTT Client in C# programming language

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

After reading http://www.hivemq.com/blog/mqtt-client-library-encyclopedia-m2mqtt on M2qtt API, implement 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.

Exercice 12 : Implement an event publisher and test it on your Mosquitto MQTT broker
Exercice 13 : Implement an event subscriber and test it on your Mosquitto MQTT broker

* MQTT CLIENT IN C PROGRAMMING LANGUAGE AND POSIX LIBRARIES (OPTIONAL)

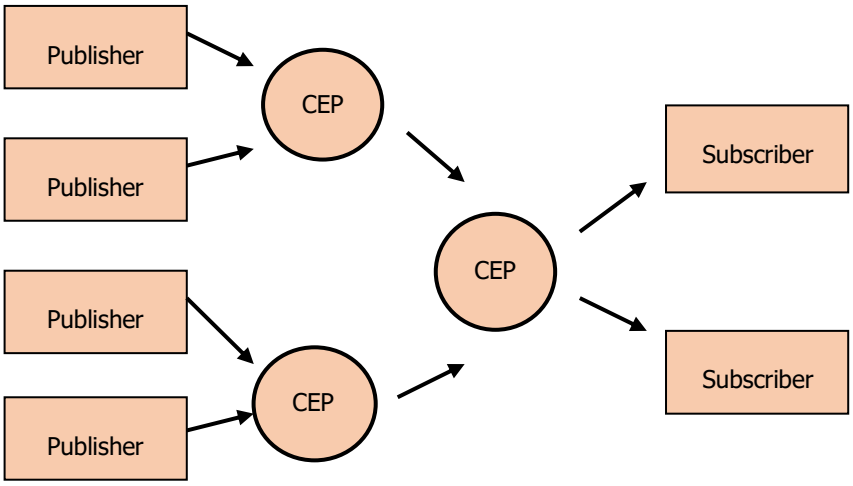
For the student that doesn't feel comfortable with C# programming ("Java like" language), here you can find sample POSIX / C code to test MQTT API in the Paho Eclipse project (https://projects.eclipse.org/projects/technology.paho)

See for libraries:

https://www.eclipse.org/downloads/download.php?file=/paho/1.0/m2mqtt/M2Mqtt_4.0.0.0_bins.zip

See for sample codes: https://www.eclipse.org/downloads/download.php?file=/paho/1.1/eclipse-paho-mqtt-c-windows-1.0.3.zip and explanations at https://eclipse.org/paho/clients/c/

4.3 MQTT and Complex event processing (CEP)



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

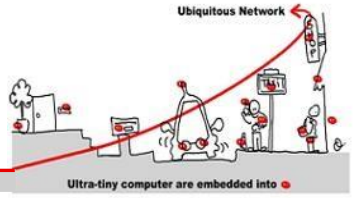
If you met some difficulties in programming MQTT client, you can use shell script with Mosquitto shell commands to do these exercises.

Exercice 14 : Propose a CEP architecture using MQTT. To test it, i mplement a simple example that uses with one publisher of an event with an integer value X, one subscriber CEPwaiting for an event with the value sin(X) and a complex event processing that transform event X in event sin(X).

One of the applications of CEP is CED (Composite Event Detection). CED is a way to implement multiple conditions on multiple events to emit new events. For example Event condition action (ECA) is a short-cut for referring to the structure of active rules in event driven architecture and active database systems. Such a rule traditionally consisted of three parts:

- The event part specifies the signal that triggers the invocation of the rule
- The condition part is a logical test that, if satisfied or evaluates to true, causes the action to be carried out
- The action part consists of updates or invocations on the local data

Tutorial: MQTT (Message Queuing Telemetry Transport)



J.-Y. Tigli, G. Rocher, I3S – University of Nice Sophia Antipolis, 2018-2019

Middleware for Internet of Things - MQTT

Exercice 15 : Implement a CED process between publisher and subscriber client to apply a set of ECA rules on input events occurrences, after evaluation of the conditions on separate data, to emit new events as actions.