# FAITES COMMUNIQUER VOS PROJETS SIMPLEMENT AVEC MQTT

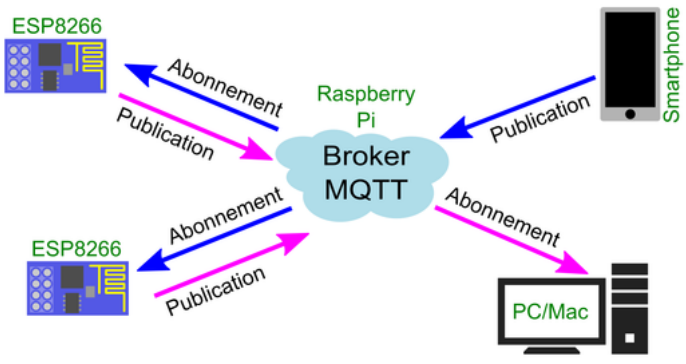
Hackable n° 026 (/Hackable/HK-026) | septembre 2018 | Denis Bodor (/auteur/view/8925-bodor_denis)

**Quel que soit le projet dans lequel vous vous lancez, s'il nécessite une communication entre plusieurs circuits, vous êtes amené à choisir une méthode pour échanger des informations, des messages, des instructions ou des valeurs. En fonction du medium utilisé pour la communication, ondes radios, câbles, infrarouge, etc., les choix sont plus ou moins nombreux. Mais dès lors qu'on parle de connectivité réseau, Ethernet ou Wifi, la liste des options n'en finit plus. Il en est pourtant une plus adaptée que les autres lorsqu'on parle de capteurs, de domotiques et d'IoT : MQTT !**

Imaginons une situation simple. Vous avez dans l'intention d'installer dans tout votre appartement ou maison une flopée de sondes de température vous informant régulièrement de leurs mesures. La construction coule de source puisqu'il s'agit d'un réseau complet et d'une taille potentiellement conséquente. Inutile de partir dans des considérations coûteuses avec des protocoles exotiques, le Wifi est la plus économique et la plus simple des solutions. Quelques dizaines d'euros de capteurs du type DS18B20 et de cartes ESP8266, un ou deux points d'accès Wifi pour couvrir toute la surface et voici votre réseau physiquement et logiquement construit : tous les ESP8266 se connectent au réseau local en Wifi et peuvent « remonter » régulièrement leurs mesures.

La question qui se pose alors est : « comment collecter ces données ? ». Plusieurs options sont envisageables, de celles consistant à demander aux ESP8266 d'envoyer les données à un serveur web en passant par un serveur central se connectant régulièrement à chaque ESP8266 pour l'interroger. Qu'il s'agisse de HTTP ou de tout autre protocole client/serveur de ce type, nous avons deux types d'acteurs : celui qui initie la communication, le client et celui qui l'attend, le serveur. L'un ou l'autre rôle peut être joué par les sondes de température, selon la manière donc on structure l'ensemble. On peut voir ce fonctionnement exactement comme une liaison filaire avec un protocole comme SPI, où il y a toujours un maître et un ou plusieurs esclaves.

Il existe cependant une troisième solution, faisant certes toujours intervenir techniquement un serveur et des clients, mais avec une approche sensiblement différente. Celle-ci se rapproche davantage de la notion utilisée dans le cas d'une liaison i2c où l'on n'a pas réellement d'entité qui contrôle d'une main de fer l'ensemble de la communication, c'est un bus. C'est exactement le même principe que pour un réseau informatique : vous avez plusieurs intervenants sur le réseau, œuvrant à diverses tâches, mais le réseau lui-même n'est pas entièrement orchestré.




Voici l'architecture typique de l'utilisation de MQTT. Nous avons au centre le broker qui gère les connexions et les échanges de messages. Les clients s'y connectent afin de recevoir les messages qui les intéressent et/ou publier des messages. L'ensemble est structuré sous la forme de « sujets d'échanges » appelés « topics » en MQTT.

## SOMMAIRE



## PAR LE MÊME AUTEUR

### SÉCURISEZ ET PROTÉGEZ VOTRE INSTALLATION MQTT (/HACKABLE/HK-026/SECURISEZ-ET-PROTEGEZ-VOTRE-INSTALLATION-MQTT)

Hackable n° 026 (/Hackable/HK-026) | septembre 2018 | Deni

(/auteur/view/8925-bodor_denis)

### MANIPULEZ ET ÉTUDIEZ LES FICHIERS BINAIRE (/OPEN-SILICIUM/OS-011/MANIPULEZ-ET-ETU LES-FICHIERS-BINAIRES)

Open Silicium n° 011 (/Open-Silicium/OS-011) | juin 2014 | De

Bodor (/auteur/view/8925-bodor_denis)

### ESSAI DE WINDOWS SUR INTEL GALILEO (/HACKABLE/HK-007/ESSAI-DE-WINDOWS-SU INTEL-GALILEO)

Hackable n° 007 (/Hackable/HK-007) | juillet 2015 | Denis Bod

(/auteur/view/8925-bodor_denis)

**Embarqué** (/content/search/?filter[]=attr_category_lk:"embarqué"&activeFacets[attr_categor

### ALIMENTEZ VOTRE PROJET AVEC UN ACCU LI-I SANS VOUS RUINER (/HACKABLE/HK-022/ALIMENTEZ-VOTRE-PROJET-AVEC-UN-ACC ION-SANS-VOUS-RUINER)

Hackable n° 022 (/Hackable/HK-022) | janvier 2018 | Denis Bo

(/auteur/view/8925-bodor_denis)

### ACME ARIETTA G25 : UN NANO-ORDINATEUR D TAILLE D'UNE GOMME (/HACKABLE/HK-003/A ARIETTA-G25-UN-NANO-ORDINATEUR-DE-LA-TAILLE-D-UNE-GOMME)

Hackable n° 003 (/Hackable/HK-003) | novembre 2014 | Denis

(/auteur/view/8925-bodor_denis)

**Embarqué** (/content/search/?filter[]=attr_category_lk:"embarqué"&activeFacets[attr_categor

**Tests et prise en main** (/content/search/?filter[]=attr_category_lk:"tests et prise en main"&activeFacets[attr_category_lk:Domaines]=tests et prise main)

### DÉMARREZ VOTRE PC À DISTANCE AVEC UNE C ARDUINO OU ESP8266 (/HACKABLE/HK-018/DEMARREZ-VOTRE-PC-A-DISTANCE-AVEC CARTE-ARDUINO-OU-ESP8266)

C'est également le principe de fonctionnement des bus logiciels comme D-Bus, permettant aux applications d'un système comme GNU/Linux de communiquer entre elles et ceci est sans doute plus proche des concepts utilisés par MQTT. En effet, même si l'analogie avec un réseau informatique est parlante, la comparaison s'arrête au principe de fonctionnement lui-même, car contrairement à un tel réseau, MQTT nécessite le fonctionnement d'un programme central, à l'instar du *démon* **`dbus-daemon`** pour le D-Bus d'un système GNU/Linux.

# 1. LES PRINCIPES DERRIÈRE MQTT

MQTT est aujourd'hui souvent considéré comme l'acronyme de « *Message Queuing Telemetry Transport* » et ce protocole trouve ses origines en 1999 dans les travaux de Andy Stanford-Clark et Arlen Nipper, alors qu'ils travaillaient pour IBM. Comme le précise le site **mqtt.org (mqtt.org)**, MQTT signifie en réalité « *MQ Telemetry Transport* », sachant que MQ fait référence à *WebSphere MQ* (historiquement « *MQSeries* » puis « *IBM MQ* »), mais dans l'absolu, ce « MQTT » ne devrait plus être considéré comme un acronyme. Quoi qu'il en soit, IBM a quelque temps utilisé ce protocole en interne avant d'en diffuser une version libre de droits en 2010, conduisant, en 2014, à sa standardisation internationale OASIS.

Une Raspberry Pi 1 ou 2 équipée d'un adaptateur wifi USB, ou une Raspberry Pi 3 fera office ici à la fois de point d'accès wifi et de broker MQTT. Ce n'est pas la seule approche possible, le broker peut être n'importe quelle machine du réseau, mais autant proprement séparer tout cela du LAN et en particulier le trafic wifi des capteurs.

Ce bref historique est important, car non seulement nous avons là un standard et non simplement une technologie comme une autre, mais ceci nous montre également en quoi MQTT est spécialisé. Andy et Arlen ont initialement développé ce protocole pour une utilisation industrielle de télémétrie en lien avec la gestion des pipelines pétroliers. Lorsqu'on ramène le contexte technique à celui de la fin des années 90, on comprend rapidement pourquoi MQTT est parfait pour l'Internet des objets. MQTT a été pensé pour :

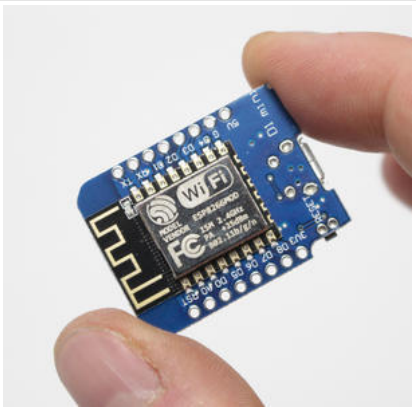
- être simple à implémenter ;
- disposer de fonctionnalités de gestion de qualité de service (QoS) ;
- être léger et pouvoir fonctionner à bas débit ;
- fonctionner indépendamment du type de données transmises (*data agnostic*) ;
- et intégrer une notion de session.

Le principe général de MQTT complètera le caractère parfaitement adapté au monde de l'IoT, impliquant généralement des systèmes optimisés pour l'autonomie et disposant donc de peu de ressources. MQTT fonctionne sur TCP/IP et fait intervenir deux types d'acteurs : des clients pouvant à la fois envoyer et recevoir des messages et un *broker* MQTT chargé de recevoir tous les messages et les dispatcher aux différents clients. Le *broker* ou « courtier » en français agit exactement comme un courtier pour des transactions bancaires, il sert d'intermédiaire entre un vendeur et un acheteur, ou ici, une entité envoyant un message et une autre souhaitant le recevoir.

Pour ancrer tout ceci dans quelque chose de plus concret, imaginez simplement nos capteurs de températures et une application quelconque chargée d'afficher de jolies jauges graphiques dans un navigateur ou sur un smartphone. Les sondes comme l'afficheur sont des clients MQTT et tout deux vont se connecter au broker. Tous sont des clients, mais les sondes publient des valeurs et l'afficheur souscrit à ce flux de données. Dans la terminologie MQTT, on parle précisément de publication et de souscription (ou d'abonnement) et ces deux actions sont généralement décrites sous les diminutifs *pub* et *sub*.

Le broker n'a que faire des données et de leur nature, son principal travail est uniquement de servir de relais. Pour ce faire, il doit également maintenir un répertoire de « qui-veut-quoi » qui prend la forme de sujets ou *topics*. Si nous partons du principe que nos sondes mesurent la température et l'hygrométrie relative, nous pouvons décider d'utiliser deux topics différents qui seront « temp » et « hygro ». Les sondes vont alors procéder aux mesures et publier régulièrement les deux valeurs, respectivement sous chaque topic. Si notre application graphique ne souhaite afficher que les données de température, celle-ci va alors s'abonner uniquement au topic « temp » et nous pouvons imaginer une autre application, s'abonnant au topic « hygro » pour, par exemple, déclencher une action.

Le principe est donc simple, tous les clients s'abonnant à un topic recevront les informations en relation et tous les clients souhaitant émettre des informations en rapport avec un topic publieront les données sous celui-ci. Mieux encore, l'adoption d'un rôle ou un autre n'est pas exclusif, pas plus que le choix d'un topic ou un autre. Un client, comme un montage à base d'ESP8266 peut parfaitement publier des données sous un ou plusieurs topics (température, hygrométrie, etc.) et s'abonner, en même temps à un ou plusieurs autres topics. De la même manière, notre hypothétique application d'affichage peut être abonnée à un topic lui permettant la collecte d'informations (températures) et publier sur un topic pour, par exemple, déclencher la mise en route d'une ventilation contrôlée par un ESP8266 servant également de sonde ou un autre, totalement distinct.



L'ESP8266 est, à mon sens, la plateforme par excellence pour créer des clients MQTT publiant des informations de façon récurrente, comme des mesures, mais aussi pour servir, par exemple, de point de contrôle de différents équipements, via un relais.

Un topic est une simple chaîne de caractères, mais qui peut être structurée hiérarchiquement. En reprenant notre exemple, avec « temp », il est possible de hiérarchiser cela, par exemple en fonction de l'emplacement du capteur. Nous pouvons ainsi avoir « salon/temp », « chambre/temp » et/ou « couloir/temp ». Ceci est laissé à votre entière discrétion et nous pouvons même pousser plus loin avec des choses comme « maison/rdc/salon/temp », « maison/etage1/chambre/temp », « maison/etage1/chambre/hygro », « maison/etage1/chambre/lux », etc.

Non seulement ceci peut mettre de l'ordre dans la nomenclature utilisée, mais surtout, côté abonnés, on peut souscrire à un ensemble de topics. Deux caractères jokers peuvent être utilisés pour cela, le **+** et le **#**. **+** est le joker pour un unique niveau de hiérarchie et un client souscrivant par exemple au topic « maison/+/temp » recevra les messages adressés par d'autres clients aux topics :

- « maison/salon/temp »
- « maison/garage/temp »
- « maison/couloir/temp »

mais pas

- « maison/salon/hygro »
- « jardin/temp ».

Le **#** est un joker multi-niveau s'utilisant toujours après un **/** et en dernier caractère. Il est destiné à remplacer n'importe quel niveau supérieur dans le topic. « maison/# » correspondra donc aux topics :

- « maison/salon/temp »
- « maison/salon/hygro »
- « maison/rdc/couloir/temp »

mais pas :

- « jardin/temp »
- « annexe/couloir/hygro ».

Enfin, un dernier caractère possède une signification particulière, mais les clients ne peuvent l'utiliser pour publier, c'est **$**. Celui-ci précède les topics concernant les statistiques internes du broker, mais le standard n'établit pas clairement son utilisation, qui est donc laissée à la discrétion de l'implémentation du broker. Cependant, par convention, le préfixe **$SYS/** est communément utilisé même si la hiérarchie des topics n'est pas forcément identique d'un broker à l'autre. Dans le cas du broker que nous allons mettre en œuvre, *Mosquitto*, voici quelques topics utilisables :

- « $SYS/broker/clients/connected » : le nombre de clients connectés au broker ;
- « $SYS/broker/clients/maximum » : le nombre maximum de clients connectés ayant été atteint ;
- « $SYS/broker/messages/received » : le nombre total de messages reçus depuis que le broker a été démarré ;
- « $SYS/broker/uptime » : le nombre de secondes écoulées depuis le démarrage ;
- « $SYS/broker/version » : la version du brocker.

Notez que dans le cas d'un abonnement au topic « # », les topics débutants par « $SYS » ne sont pas automatiquement inclus. Pour s'abonner à l'ensemble de la hiérarchie en question, il faut utiliser « $SYS/# ».

Voici un vieil ami que les lecteurs de longue date connaissent sans doute. Il s'agit d'un module ESP-01 équipé d'un régulateur 3,3V et d'une sonde de température, le tout monté sur un adaptateur USB 5V. Son travail, et celui de ses compères : mesurer la température et la publier en MQTT.

Enfin, pour compléter davantage la description d'une architecture MQTT, il faut savoir qu'il est possible de mettre en place plusieurs brokers interconnectés à l'aide de ponts (*bridges*). Ceci permet de structurer plus avant une installation de taille conséquente avec, par exemple, un broker local en mesure de retransmettre les messages publiés ainsi que les souscriptions à un broker central distant, tout en permettant un filtrage. Ceci sort du cadre introductif de cette découverte de MQTT, mais est relativement bien décrit dans la documentation de Mosquitto (en anglais, bien sûr).

# 2. INSTALLATION DU BROKER SUR PI

Pour supporter notre architecture de démonstration, nous allons utiliser une Raspberry Pi comme élément central pour installer le broker. Ce système servira également de point d'accès wifi pour la connexion des ESP8266 au réseau. Ceux-ci seront des cartes/modules Wemos D1 mini, ou plutôt des clones, qu'on pourra très facilement se procurer sur eBay pour moins de 3 euros auprès de vendeurs chinois (~6€ chez un revendeur en Europe avec une livraison pas toujours plus rapide).

La création d'un point d'accès Wifi a déjà été traitée dans le magazine (numéro 14) et nous ne reviendrons pas sur le sujet. Notez cependant que ceci n'est pas une obligation et que vous pouvez parfaitement connecter vos ESP8266 à votre point d'accès domestique (ou box). Personnellement, je préfère bien séparer les choses sachant que la Pi sera également connectée au réseau local en Ethernet. Il est donc parfaitement possible de contacter le broker depuis le réseau local en filaire pour, par exemple, souscrire aux topics et afficher des données.

Le broker que nous utiliserons est sans le moindre doute le plus populaire et dispose de toutes les fonctionnalités adaptées à la dernière version du standard (3.1.1) : Mosquitto. Vous pourrez l'installer très simplement à l'aide de la commande **`sudo apt-get install install mosquitto`** et nous utiliserons également les utilitaires en ligne de commandes nous permettant de rapidement tester notre installation, via le paquet **`mosquitto-clients`**.

Immédiatement après l'installation du paquet, le broker MQTT est opérationnel et vous pouvez de suite utiliser un client localement pour vous en assurer, avec la commande :

```
$ mosquitto_sub -v -h localhost -t '$SYS/broker/version'
$SYS/broker/version mosquitto version 1.4.10
```

Le paquet **`mosquitto-clients`** fournit deux commandes, **`mosquitto_sub`** pour une souscription et **`mosquitto_pub`** pour une publication. Ici, nous nous connectons au broker fonctionnant sur la Pi elle-même et nous abonnons au topic « $SYS/broker/version » nous permettant d'obtenir le numéro de version du serveur Mosquitto. Notez l'utilisation des apostrophes autour du topic spécifié afin d'éviter que le shell n'interprète **`$SYS`** comme une variable et ne tente d'en utiliser le contenu. Autre point important, ici nous utilisons **`localhost`** comme hôte (option **`-h`**) pour nous connecter, ce qui correspond à l'adresse 127.0.0.1 et non à l'IP d'une des interfaces (Wifi ou Ethernet) de la Pi. Ceci n'a pas grande importance ici, mais plus tard, lorsque nous sécuriserons notre installation, il sera capital d'utiliser le nom d'hôte de la machine suivi de « .local ». Autant en prendre l'habitude de suite ou éviter des comportements problématiques par la suite.

Enfin, vous remarquerez que l'outil **`mosquitto_sub`** ne vous rend pas la main et reste connecté au broker. C'est le principe même du fonctionnement de MQTT lors d'un abonnement à un topic, rester à l'écoute. Vous pouvez stopper l'exécution en utilisant, tout simplement, le raccourci CTRL+C.

# 3. PREMIER CROQUIS ESP8266 ET BASE DE TRAVAIL

Utiliser MQTT avec un ESP8266, ou une carte Arduino avec un shield Ethernet, n'est pas bien compliqué à partir du moment où on a correctement assimilé le principe de fonctionnement de MQTT, des topics, des abonnements, etc. Une bibliothèque écrite par Nick O'Leary (« knolleary » sur GitHub) est directement installable depuis le

gestionnaire de bibliothèques Arduino : PubSubClient. Ce n'est, bien entendu, pas la seule solution, mais celle qui, je trouve, est à la fois la plus simple d'utilisation et la plus fonctionnelle. Celle-ci fonctionnera tout aussi bien sur Arduino (avec shield Ethernet), ESP8266, Arduino YUN, ESP32, etc.

Bien qu'on puisse parfaitement utiliser MQTT avec un seul montage faisant office de sonde, avec la Pi comme broker et client et un ESP8266 comme client, tout ceci prend réellement toute son importance lorsqu'il s'agit d'avoir une tripotée de clients, et donc d'ESP8266, communiquant via le broker. Il est donc généralement judicieux de prévoir, dès les premiers essais, quelque chose de fonctionnel et qu'il sera possible de rapidement faire évoluer en taille. Quelque chose de *scalable* comme aiment à le dire certains joueurs de business loto qui s'ignorent...

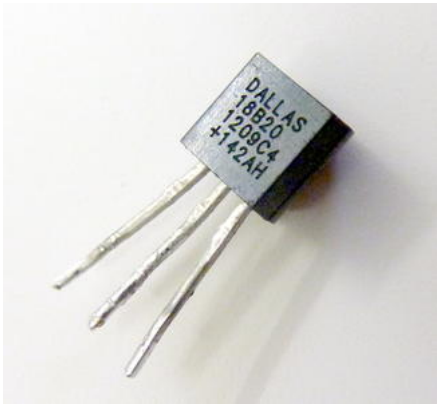
Comme je l'avais évoqué dans des articles passés, dans ce genre de situations, avec un unique croquis qui sera installé sur maintes ESP8266, il est préférable de stocker certaines informations en dehors du croquis afin qu'elles ne nécessitent pas de changement dans le code pour chaque carte : le SSID du point d'accès, la phrase de passe associée et le nom d'hôte utilisé afin de ne pas reposer sur les adresses IP, mais ici également, de donner un nom au client MQTT. Ces informations seront enregistrées dans l'EEPROM émulée des ESP8266 et ne seront donc pas affectées ensuite lors d'une mise à jour du croquis.

Pour stocker ces informations, nous utiliserons le code suivant :

```
#include <ESP8266WiFi.h>
#include <EEPROM.h>
// structure pour stocker les infos
struct EEconf {
  char ssid[32];
  char password[64];
  char myhostname[32];
};
void setup() {
  // déclaration et initialisation
  EEconf myconf = {
    "monSSID",
    "phrase2passe",
    "nomhote"
  };
  // seconde variable pour la (re)lecture
  EEconf readconf;

  Serial.begin(115200);
  // initialisation EEPROM
  EEPROM.begin(sizeof(myconf));
  // enregistrement
  EEPROM.put(0, myconf);
  EEPROM.commit();
  // relecture et affichage
  EEPROM.get(0, readconf);
  Serial.println("\n\n\n");
  Serial.println(readconf.ssid);
  Serial.println(readconf.password);
  Serial.println(readconf.myhostname);
}
void loop() {
}
```

Ce croquis se contente de stocker dans une structure les trois informations qui nous intéressent, les enregistre dans l'EEPROM puis les relit pour enfin les afficher. Dès son exécution sur l'ESP8266 ces données seront stockées indépendamment des enregistrements de croquis qui suivront et nous n'avons donc qu'à l'utiliser une seule fois sur chaque carte pour « embarquer » les paramètres de connexion qui seront relus et utilisés dans un unique croquis commun à tous les ESP8266.




Le 18B20 de Dallas est un classique. Il s'agit d'un capteur de température avec une interface 1-Wire, très facile à utiliser à l'aide d'une simple résistance, dans n'importe quel montage Arduino, ESP8266 ou ESP32.

Nous pouvons à présent nous pencher sur notre premier vrai croquis utilisant MQTT. La première chose à faire pour ce genre de projet est de déterminer clairement et à tête reposée quels sont vos besoins et comment structurer vos topics. Ici, nous n'avons pas de cas pratique n'impliquant pas la mise en œuvre d'autres composants comme un capteur de température ou autre, nous sommes donc libres comme l'air quant à cette structure. Nous choisissons donc totalement arbitrairement les fonctionnalités souhaitées :

- Les ESP8266 doivent pouvoir, tous ensemble, répondre à une commande permettant l'allumage ou l'extinction de la led intégrée.
- Chaque ESP8266 doit régulièrement envoyer un message avec une valeur variable tout en permettant de savoir d'où vient chaque message.

Pour rapprocher cela d'un cas concret, il est parfaitement imaginable que chaque ESP8266 puisse relever une mesure rapportée régulièrement et qu'une commande agisse comme un interrupteur global. Pourquoi pas une serre avec des mesures de température et la commande d'un ensemble de ventilation type VMC. On peut aussi imaginer un scénario où mesures et commandes sont dissociées, comme le relevé de température et la commande de luminaires aux mêmes endroits.

Nous allons donc établir le topic « ctrlled » permettant, si la valeur accompagnant le message (le *payload*) est à 1, d'allumer la led et l'éteindre si elle est à 0. Les ESP8266 vont tous s'abonner à ce topic et réagir en conséquence. Inversement, chaque ESP8266 va régulièrement publier une valeur s'incrémentant sur le topic « maison/XXX/valeur » où « XXX » sera l'identité de chaque ESP8266 (son nom d'hôte, mais en situation il pourra s'agir, par exemple, du nom d'une pièce).

Le croquis générique débutera, tout naturellement, par l'inclusion des bibliothèques nécessaires :

```
// connectivité
#include <ESP8266WiFi.h>
// mDNS pour les nom d'hôtes
#include <ESP8266mDNS.h>
// émulation EEPROM
#include <EEPROM.h>
// MQTT
#include <PubSubClient.h>
```

On passe ensuite à la déclaration des variables globales :

```
// nom de la machine ayant le broker (mDNS)
const char* mqtt_server = "raspibase.local";
// structure pour la configuration
struct EEconf {
  char ssid[32];
  char password[64];
  char myhostname[32];
} readconf;
// objet pour la connexion
WiFiClient espClient;
// connexion MQTT
PubSubClient client(espClient);
// pour l'intervalle
long lastMsg = 0;
// valeur à envoyer
byte val = 0;
```

Notez que nous utilisons ici la résolution de nom mDNS (voir *Hackable n°21*) et que la désignation du serveur MQTT se fait sur la base du nom d'hôte de la Pi suivi de « .local ». Je profite de cette occasion pour rappeler qu'il faut toujours changer le nom d'hôte d'une Pi fraîchement installée, car mDNS (service Avahi) est actif par défaut et que toutes les Pi s'appellent « raspberrypi ». Si vous en avez plus d'une sur votre réseau avec ce nom d'hôte vous risquez d'avoir des problèmes…

Habituellement, la connexion au point d'accès wifi se fait dans la fonction **`setup()`**, mais nous préférons ici alléger celle-ci et réunir la partie connexion et la configuration mDNS dans une fonction dédiée :

```
void setup_wifi() {
  // mode station
  WiFi.mode(WIFI_STA);
  Serial.println();
  Serial.print("Connexion ");
  Serial.println(readconf.ssid);
  // connexion wifi
  WiFi.begin(readconf.ssid, readconf.password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  // affichage
  Serial.println("");
  Serial.println("Connexion wifi ok");
  Serial.println("Adresse IP: ");
  Serial.println(WiFi.localIP());
  // configuration mDNS
  WiFi.hostname(readconf.myhostname);
  if (!MDNS.begin(readconf.myhostname)) {
    Serial.println("Erreur configuration mDNS!");
  } else {
    Serial.println("répondeur mDNS démarré");
```

```
    Serial.println(readconf.myhostname);
  }
}
```

Notez que les données de configuration sont ici supposées déjà valides. Il conviendra donc de peupler la structure **`readconf`** avant d'appeler **`setup_wifi()`**. Voici pour la partie générique, nous pouvons maintenant nous pencher sur les fonctions propres à MQTT. Le principe de fonctionnement de la bibliothèque *PubSubClient* repose sur un mécanisme de *callback* (ou fonction de rappel en bon français). Une fonction de callback est passée en paramètre d'une autre fonction et est appelée automatiquement dans certaines situations. Ceci n'est en rien un mécanisme propre à MQTT, ni même au domaine du réseau, mais utilisé de longue date un peu partout. Dans notre cas, la bibliothèque demande que vous déclariez et spécifiez cette fonction de callback qui sera appelée en cas de réception d'un message sur n'importe quel topic souscrit :

```
void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message [");
  Serial.print(topic);
  Serial.print("] ");
  // affichage du payload
  for (int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.println();
  // le caractère '1' est-il le premier du payload ?
  if ((char)payload[0] == '1') {
    // oui led = on
    digitalWrite(BUILTIN_LED, LOW);
  } else {
    // non led = off
    digitalWrite(BUILTIN_LED, HIGH);
  }
}
```

Le prototype de cette fonction, le nombre et le type d'arguments qui lui sont passés, sont imposés par la bibliothèque. Nous avons ici un pointeur sur une chaîne de caractères pour le topic et pour les données associées (le *payload*) et un entier spécifiant la taille de ces dernières. Une simple boucle nous permet de directement envoyer ces données pour vérification au moniteur série et nous testons ensuite, très simplement, si le premier caractère de ces données est « 1 ». Si tel est le cas, nous mettons la broche désignée par **`BUILTIN_LED`** (la led installée sur l'ESP8266) à la masse. Notez que la led étant reliée entre le port et l'alimentation via une résistance, la logique est inversée (**`LOW`** = mise à la masse = led allumée). N'importe quel autre caractère que « 1 » provoque l'extinction de la led.

Pour que tout ceci fonctionne, nous devons nous connecter au broker et maintenir cette connexion. Là encore, plutôt que de surcharger une fonction qui doit rester simple, nous ne placerons pas le code nécessaire directement dans **`loop()`**, mais dans une fonction dédiée :

```
void reconnect() {
  // Connecté au broker ?
  while(!client.connected()) {
    // non. On se connecte.
    if(!client.connect(readconf.myhostname)) {
      Serial.print("Erreur connexion MQTT, rc=");
      Serial.println(client.state());
      delay(5000);
      continue;
    }
    Serial.println("Connexion serveur MQTT ok");
    // connecté.
    // on s'abonne au topic "ctrlled"
    client.subscribe("ctrlled");
  }
}
```

Cette approche nous permettra également d'ajouter les éléments nécessaires à la sécurisation de l'ensemble par la suite. Pour l'heure, nous nous contentons de vérifier l'état de la connexion et, si elle est n'est pas déjà établie, nous nous connectons au broker puis souscrivons au topic « ctrlled » afin de recevoir les messages destinés à contrôler la led embarquée. Tant que la connexion est maintenue avec le broker cet abonnement restera actif et nous permettra de recevoir les messages. Dans le cas contraire, il conviendra de se reconnecter puis de se réabonner au topic.

Dès lors que cet abonnement est en place, l'arrivée d'un message sera automatiquement prise en charge par la bibliothèque et la fonction callback sera appelée en passant en argument le topic, la donnée associée (payload) et la longueur de cette dernière.

Notez l'argument utilisé pour **`client.connect()`**. Il ne s'agit ni de l'hôte du broker, ni réellement du nom d'hôte du client, mais d'une chaîne de caractères permettant au broker d'identifier la connexion (un *clientid*). J'utilise ici tout simplement le nom d'hôte de l'ESP8266, car il est unique, mais n'importe quelle chaîne de caractères fera l'affaire, à partir du moment où chaque client en utilise une différente des autres. Gardez cela à l'esprit, car deux clients ayant le même identifiant vont perturber le broker et provoquer des déconnexions/reconnexions intempestives.


Les modules capteurs peuvent combiner plusieurs mesures en un seul composant afin d'éviter une ignoble salade de câbles. À droite, nous avons un BMP180 (température et pression) et à gauche un BME280, (température, pression et hygrométrie).

Il ne nous reste plus, maintenant, qu'à appeler ces fonctions depuis **`setup()`** et **`loop()`**, et commençons, tout naturellement, par configurer le tout au démarrage du croquis :

```
void setup() {
  // configuration led
  pinMode(BUILTIN_LED, OUTPUT);
  // configuration moniteur série
  Serial.begin(115200);
  // configuration EEPROM
  EEPROM.begin(sizeof(readconf));
  // lecture configuration
  EEPROM.get(0, readconf);
  // configuration wifi
  setup_wifi();
  // configuration broker
  client.setServer(mqtt_server, MQTT_PORT);
  // configuration callback
  client.setCallback(callback);
}
```

Nous avons déjà détaillé les fonctions et les commentaires devraient être suffisants. Notez cependant l'utilisation de **`MQTT_PORT`**, qui est une macro valant 1883 et correspondant au port TCP/IP par défaut des broker MQTT. Nous reviendrons sur ce point par la suite.

Enfin, voici notre boucle principale :

```
void loop() {
  // array pour conversion val
  char msg[16];
  // array pour topic
  char topic[64];
  // Sommes-nous connectés ?
  if (!client.connected()) {
    // non. Connexion
    reconnect();
  }
  // gestion MQTT
  client.loop();
  // temporisation
  long now = millis();
  if (now - lastMsg > 5000) {
    // 5s de passé
    lastMsg = now;
    val++;
    // construction message
    sprintf(msg, "hello world #%hu", val);
    // construction topic
    sprintf(topic, "maison/%s/valeur", readconf.myhostname);
    // publication message sur topic
    client.publish(topic, msg);
  }
}
```

Les messages envoyés en MQTT sont typiquement des chaînes de caractères, tout comme les topics et nous prévoyons des tableaux destinés à les stocker. Le topic est ici composé en fonction du nom d'hôte de l'ESP8266 sur lequel le croquis fonctionne et nous utilisons la fonction **`sprintf()`** afin d'intégrer cette chaîne dans le topic. Bien entendu, en situation réelle, il conviendra d'adapter cela en fonction de vos besoins, ou de tout simplement choisir un nom d'hôte s'intégrant dans votre nomenclature. Ceci, comme l'ID client, pourrait également être stocké en EEPROM avec le reste des paramètres de chaque ESP8266. La création de la chaîne de caractères du message se fait de la même manière, en convertissant simplement un entier (**`val`**) en chaîne.

Le DS18B20 n'est pas seulement disponible sous la forme d'un composant (paquet TO-92), mais est également vendu en version étanche et équipée d'un câble d'un mètre pour moins de 2€ (eBay). Ce format est absolument parfait pour la construction de capteurs communiquant en MQTT puisque la sonde peut se glisser n'importe où alors que l'ESP8266 reste à l'abri et à portée de wifi.

La gestion de MQTT par la bibliothèque se fait via des appels répétés à **`client.loop()`**. C'est pour cette raison que nous ne pouvons pas gérer la temporisation de 5 secondes via un simple **`delay(5000)`** qui bloquerait l'exécution du croquis, empêchant une réception correcte d'un message sur le topic auquel nous avons souscrit.

# 4. ESSAIS ET VALIDATION DU CROQUIS

Une fois le croquis chargé dans la mémoire de l'ESP8266, nous pouvons très facilement vérifier son fonctionnement et celui de l'ensemble du système. Pour cela, nous n'avons qu'à nous tourner vers les outils disponibles pour Mosquitto côté Pi. Nous pouvons, dans un premier temps, utiliser **`mosquitto_pub`** afin de publier un message que l'ESP8266 va alors utiliser pour changer l'état de sa led :

```
$ mosquitto_pub -h raspibase.local -t ctrlled -m 0
$ mosquitto_pub -h raspibase.local -t ctrlled -m 1
```

L'option **`-h`** permet de spécifier l'hôte sur lequel fonctionne le broker et non l'hôte ESP8266, c'est là tout l'intérêt de MQTT et du broker. Nous spécifions ici le nom d'hôte de la Raspberry Pi, mais comme il s'agit de la même machine que celle sur laquelle nous exécutons la commande, nous aurions tout aussi bien pu utiliser **`localhost`**, **`127.0.0.1`** ou l'adresse IP de la Pi. L'utilisation de mDNS simplifie toutefois grandement les choses et, comme nous le verrons plus loin, est la bonne approche à adopter pour ensuite ajouter plus de sécurité (SSL/TLS).

Le topic sur lequel publier est spécifié via l'option **`-t`** et la « charge utile » (*payload*), ou en d'autres termes le contenu du message à envoyer, est spécifié avec l'option **`-m`**. Étant donné le fonctionnement du croquis, **`"0"`** fonctionnera de la même manière que n'importe quel caractère n'étant pas **`"1"`**. Inversement, **`"1"`** aura le même effet que **`"1coucou"`** puisque nous ne considérons que le premier caractère de la chaîne. Comprenez bien que ceci dépend entièrement de l'écriture de votre fonction de callback et n'est en rien lié au fonctionnement même de MQTT.

Inversement, nous pouvons également nous abonner au topic sur lequel notre (ou nos) ESP8266 publie :

```
$ mosquitto_sub -h raspberrypiled.local -t maison/mqtttaf1/valeur
hello world #24
hello world #25
hello world #26
hello world #27
hello world #28
[...]
```

Notez que le fait que le topic existe ou non ne change rien au fonctionnement de **`mosquitto_sub`**. La notion même d'existence d'un topic est sans signification en MQTT, un topic n'existe finalement que lorsqu'un message est publié sous celui-ci. « mqtttaf1 » est ici le nom d'hôte choisi pour notre ESP8266, mais il pourrait s'agir de n'importe quoi. En cas d'utilisation de plusieurs ESP8266, il est possible de s'abonner aux messages de l'ensemble avec « maison/+/valeur ». Nous verrons alors apparaître la totalité des messages sur l'écran. L'option **`-v`** pourra être ajoutée pour afficher le provenance de chaque message :

```
$ mosquitto_sub -v -h raspberrypiled.local -t maison/+/valeur
maison/mqtttaf1/valeur hello world #63
maison/mqtttaf2/valeur hello world #42
maison/mqtttaf1/valeur hello world #64
maison/mqtttaf2/valeur hello world #43
[...]
```

# 5. ON A BIEN AVANCÉ

Nous venons de voir que MQTT est une approche radicalement différente d'une simple communication bilatérale comme on peut le voir avec un client et un serveur HTTP. Cette architecture particulière permet toutes sortes de structurations et énormément de choses sont laissées au bon vouloir du programmeur. Mais avec beaucoup de liberté arrive également souvent beaucoup de confusion et il est donc très important, avant d'écrire la moindre ligne de code, d'établir un cahier des charges précis afin de pouvoir structurer correctement ses topics.

Dans notre exemple, tous les ESP8266 s'abonnent au topic « ctrlled » et réagissent aux messages afin de contrôler, ensemble, leur led. Ceci est bien plus confortable dans le cas d'un contrôle de luminaires puisqu'une publication suffirait à allumer ou éteindre chaque éclairage, mais quid du contrôle individuel ? Si ce besoin se fait jour par la suite, ce n'est pas seulement le code lui-même qu'il faudra revoir, mais l'intégrité du concept et donc des topics utilisés.


Un ESP8266 peut tout aussi bien publier en MQTT que s'abonner à un topic. De ce fait, rien ne vous empêche alors d'en utiliser un exemplaire pour afficher physiquement une valeur reçue (d'un autre ESP8266) avec un ampèremètre analogique comme celui-ci...

Cependant, après deux ou trois essais, frustrations et révisions, on apprend à être plus prudent et réfléchi, et la leçon est ainsi vite retenue. Ce qui est plus problématique et ne peut être ignoré en revanche est l'absence totale de sécurité dans notre premier exemple. Absolument rien n'empêche une machine du réseau de s'abonner à tous les topics pour en étudier le contenu puis à publier selon son bon vouloir des directives ou fausses mesures qui ne pourront être différenciées des données légitimes. En d'autres termes, il n'y a absolument aucune sécurité dans une installation aussi basique et c'est précisément ce que nous allons corriger dans le prochain article...