ioc-projet-cr

Projet : Contrôle ESP32 via navigateur WEB

réalisé par Amredin Batbout, Joachim Kponou.

Le but de ce projet est de concevoir et de mettre en œuvre un système permettant de contrôler et de surveiller des modules ESP32 (LED, photorésistance, buzzer, etc.) à distance via un navigateur web. Autrement dit, on souhaite créer une interface web qui permet de lire les données des capteurs de l'ESP32 et d'envoyer des commandes aux actionneurs en temps réel.

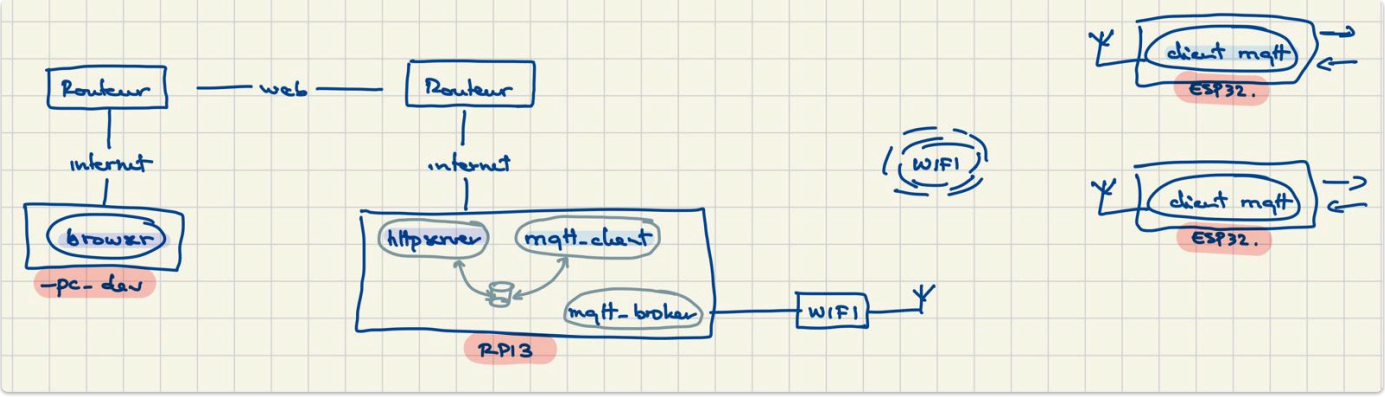


figure 1 : architecture globale du système.

Pour y parvenir, nous allons mettre en place un système composé de :

1. Un serveur HTTP sur une Raspberry Pi 3, qui servira d'interface entre le navigateur web et les modules ESP32.
2. Un broker MQTT sur la Raspberry Pi 3, qui permettra la communication entre les différents clients MQTT (les ESP32 et le serveur HTTP).
3. Des clients MQTT sur les ESP32, qui publieront les données des capteurs et souscriront aux commandes pour les actionneurs.
4. Une base de données (ou un simple fichier) sur la Raspberry Pi 3, qui stockera les données reçues des ESP32.

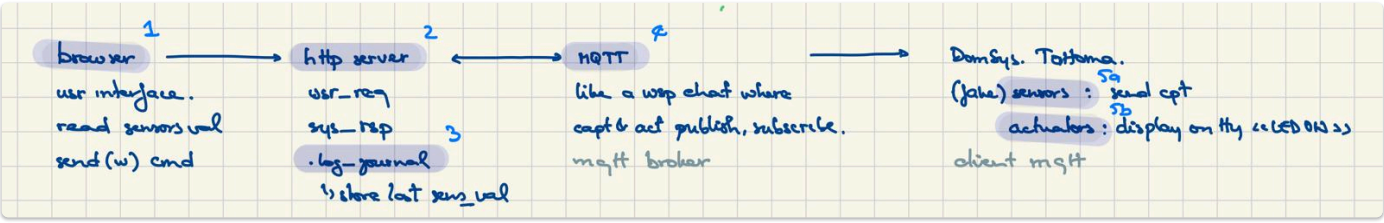


figure 2 : les différents composants de notre système (vu par moi, Joey)

Le projet se déroulera en deux grandes étapes : une preuve de concept sur une machine Linux (pour valider le fonctionnement du système) et la réalisation finale sur les composants matériels (Raspberry Pi 3 et ESP32).

Partie 1 : proof of concept.

Voici ce qu'on souhaite mettre en place ici :

1. un site web (`index.html`) où l'on peut voir les informations des capteurs et envoyer des commandes (ex : "allume la LED")
2. un serveur http (`app.py`) qui reçoit les demandes du site web et les transmet aux capteurs / actionneurs.
3. le broker mqtt pour gérer la communication entre le serveur et la carte.
4. de faux capteurs et actionneurs (des clients mqtt). Vu qu'on n'a pas encore les ESP32, on va les simuler avec un programme qui fait semblant d'envoyer des données. Par exemple, un compteur à la place de la photorésistance (`simulate_sensor.py`), et un simple affichage "LED allumée" dans le terminal pour simuler les LED (`simulate_actuator.py`).
5. un fichier `sensor_data.txt` qui stocke les valeurs envoyées par notre capteur factice.

Step 1 : installer et configurer le broker MQTT (`Mosquitto`)

Le broker MQTT est le cœur du système, étant donné qu'il permet la communication entre les différents clients (serveur HTTP, clients MQTT simulés). On commence donc par installer et configurer `Mosquitto`, un broker MQTT léger et facile à utiliser.

Pour installer `mosquitto` :

```
sudo apt update
sudo apt install mosquitto mosquitto-clients
```

lance le broker MQTT :

```
mosquitto
```

Pour tester le broker, on ouvre un terminal et on tape :

```
mosquitto_sub -t "topic_test"
```

dans un autre terminal, on envoie un message :

```
mosquitto_pub -t "topic_test" -m "Hello MQTT"
```

On peut voir le message "Hello MQTT" apparaître dans le premier terminal. Cela confirme que le broker fonctionne correctement.

Step 2a : créer un client MQTT simulé (capteur)

Maintenant que le broker MQTT fonctionne, on va créer un client MQTT qui simule un capteur (un compteur qui simule la photorésistance). Ce client va publier des données sur un topic MQTT à intervalles réguliers. Pour ce fait :

1. on installe la bibliothèque MQTT pour Python :

   ```
   sudo apt install python3-paho-mqtt
   ```

2. on crée un script Python pour simuler le capteur (`simulate_sensor.py`) qui ne fait que :
   1. créer un client
   2. connecter ce client au broker mqtt

    3. publier sur le topic 'sensor/photoresistor' la valeur du compteur qu'il incrémente à chaque publication
3. on exécute le script

```
python3 simulate_sensor.py
```

    - on a des messages comme "Publication de la valeur : 0", "Publication de la valeur : 1", etc., toutes les 5 secondes, comme attendu.
4. on vérifie que les données sont bien publiées :
    - on ouvre un autre terminal et on s'abonne au topic `sensor/photoresistor` :

```
mosquitto_sub -t "sensor/photoresistor"
```

    - on a les valeurs (0, 1, 2, etc.) apparaître dans ce terminal. yay!

Step 2b : créer un client MQTT simulé (actionneur)

Ensuite, on va créer un client MQTT qui simule un actionneur (celle d'une LED). Ce client sera abonné à un topic MQTT et réagira aux commandes reçues.

1. on crée un script Python (`simulate_actuator.py`) qui :
    1. crée un client, se connecte au broker
    2. s'abonne au topic 'actuator/led'
    3. avec une fonction `on_message` qui gère les commandes reçues (écrire dans le terminal "LED allumée" s'il reçoit "ON" et inversement)
2. on exécute le script :

```
python3 simulate_actuator.py
```

   Le script est maintenant en attente de commandes sur le topic `actuator/led`.
3. on ouvre un autre terminal et publier une commande sur le topic `actuator/led`

```
mosquitto_pub -t "actuator/led" -m "ON"
```

    - On peut voir "Commande reçue : ON" et "LED allumée" dans le terminal où le script `simulate_actuator.py` est en cours d'exécution.

Dans notre projet final, les commandes `mosquitto_sub` et `mosquitto_pub` seront gérées par le serveur http et les scripts `simulate_.py` seront eux gérés par l'esp32 avec du Arduino.

Step 3 : créer un serveur HTTP simple (en Python avec Flask)

Enfin, on passe au serveur HTTP qui servira d'interface entre l'utilisateur (via un navigateur web) et le système MQTT. Ce serveur devra être capable de :

- recevoir les données des capteurs via MQTT. (sub au topic du capteur)
- afficher ces données dans une page web (envoyer directement à l'index.html avec du JavaScript ou lire la dernière valeur dans la database, variable `sensor_value`)
- envoyer des commandes aux actionneurs via MQTT (pub sur le topic de l'actionneur)

1. on installe Flask.

```
sudo apt install python3-flask
```

2. on crée un serveur HTTP assez basique (`server.py`) qui va juste :
    1. créer un client mqtt
    2. s'abonner au topic du capteur, traiter la valeur reçue (enregistrer dans un fichier `sensor_data.txt`)
    3. publier sur le topic de l'actionneur la commande de l'utilisateur (@app.route, POST)
    4. afficher la dernière valeur reçue (stockée dans la pseudo base de données) sur le site (grâce à JavaScript)
3. on crée une interface web assez simple (.html) et un petit simple JavaScript pour gérer l'envoi des commandes au serveur et la mis à jour de la valeur du capteur sur le site.
4. on peut démarrer le serveur :

    ```
    python server.py
    ```

    Le serveur est accessible à l'adresse `http://localhost:5000`.
5. pour tester l'interface web :
    - ouvrir un navigateur et accéder à `http://localhost:5000`.
    - on peut voir la dernière valeur reçue du capteur (publiée par `simulate_sensor.py`, attention, il ne faut pas oublier de 'démarrer' le capteur en exécutant le script dans autre terminal).
    - cliquer sur les boutons "Allumer la LED" et "Éteindre la LED" pour envoyer des commandes à l'actionneur simulé.

Petite note : on a modifié l'interface de site entre temps et on a oublié de prendre un capteur de la v0. Néanmoins, tout ce qui vient d'être dit reste toujours valable.

Structure du projet (proof-of-concept) :

```
ioc_project/
├── server.py             # Fichier principal Flask
├── sensor_data.txt       # Fichier pour stocker les données des capteurs
├── templates/            # Dossier pour les templates HTML
│   └── index.html        # Template pour l'interface web
└── requirements.txt      # Fichier pour les dépendances
```

À cette étape, on a tous les composants qui communiquent correctement entre eux :

- Les clients MQTT (capteurs et actionneurs) publient et reçoivent des messages correctement.
- Le serveur HTTP reçoit les données des capteurs et les affiche dans l'interface web.
- Les commandes envoyées depuis l'interface web sont bien reçues par les actionneurs.

On peut passer à la réalisation sur du matériel.

Partie 2 : prototype.

Liste des tâches à accomplir :

- A. Configuration Matérielle (RPi)

- [x] Installer Raspberry Pi OS sur la RPi
- [x] Configurer le WIFI, l'accès SSH
- [x] Installer le broker MQTT sur la RPi
- [x] Installer les librairies nécessaires sur l'ESP32

- B. ESP32
  - [x] Tester la communication ESP32 ⟷ RPi
  - [x] Porter le code Python (`simulate_.py`) vers l'ESP32 (Arduino C++)
    - [x] Connexion WiFi
    - [x] Publication des données capteurs (topic `sensor/photoresistor`)
    - [x] Abonnement aux commandes (topic `actuator/led`)
    - [x] Ajouter la gestion des vrais pins GPIO
- C. Serveur HTTP sur RPi
  - [x] Installer les dépendances (Flask, Paho)
  - [x] Adapter le code Flask pour le matériel :
    - Changer `localhost` par l'IP locale
- D. Interface Web Finale
  - Design (CSS, JavaScript)
- E. Déploiement
  - Service Mosquitto
  - Script Flask
  - Programme ESP32

Étape 1 : Configuration matérielle.

Cette partie nous a pris pratiquement la moitié du temps de réalisation.

1.1. Installation de la Raspberry Pi OS sur la RPi

1. Télécharger & Installer Raspberry Pi Imager :
   - https://www.raspberrypi.com/software/ : rpi_imager_1.8.5_amd64.deb
   - Installer le fichier .deb : `sudo dpkg -i rpi_imager.deb`
   - Lancer l'outil : rpi-imager
2. Préparer la carte SD
   - Dans Raspberry Pi Imager :
     - Choisir l'OS : *Raspberry Pi OS 3* dans notre cas.
     - Sélectionner le stockage : la carte SD (`/dev/sdb`).
   - Configurer avant écriture
     - Dans les Settings : modifier ce que vous voulez s'il le faut
       - mot de passe, wifi, etc.
   - Écrire l'image : Cliquer sur *"Write"* → Confirmer l'effacement de la carte.
3. Booter la Raspberry Pi
   - Insérer la carte SD dans la Raspberry Pi.
   - Brancher : Alimentation USB-C
   - Attendre 2-3 minutes que le système termine la configuration.

1.2. Connexion WIFI et accès SSH.

Pour la connexion internet, on a décidé de faire un partage de connexion avec notre 4G pour plus de flexibilité et de mobilité.
On récupère l'adresse IP de la rpi (`hostname -I`) et on vérifie sur le `pc_dev` connecté au même réseau avec `ping <IP_RPI>`.

Pour l'accès SSH (en vue de simplifier la programmation), normalement l'accès est enable lors de l'installation de l'OS. Si ce n'est pas le cas, il suffit le configurer dans les paramètres de l'OS.

1.3. Installation du broker MQTT sur la RPi.

```
sudo apt install mosquitto mosquitto-clients
sudo apt install python3-paho-mqtt
```

Pour rendre le service persistent :

```
sudo systemctl enable mosquitto
```

Pour tester le broker (communication pc_dev ⟷ rpi), on lance :
sur le `pc_dev` : `mosquitto_sub -h <IP_RPI> -t "test/topic"`
sur rpi : `mosquitto_pub -h <IP_RPI> -t "test/topic" -m "Message from RPi"`

À ce niveau, on a eu quelques soucis de firewall qu'on a réglé avec un petit script (source)

```
# Installer ufw
sudo apt update
sudo apt install ufw -y

# Configurer les règles de base
sudo ufw default deny incoming  # Bloquer toutes les connexions entrantes par défaut
sudo ufw default allow outgoing  # Autoriser toutes les connexions sortantes

# Autoriser les services essentiels
sudo ufw allow 22/tcp  # SSH
sudo ufw allow 1883/tcp  # MQTT
sudo ufw allow 5000/tcp  # Flask

# Activer le pare-feu
sudo ufw enable

# Vérifier l'état
sudo ufw status verbose
```

1.4. Configuration Arduino IDE

1. **Installer les bibliothèques** :
   - `PubSubClient` (Gestion MQTT)
   - `WiFi` (Connexion réseau)
     Via :
     `Croquis` → `Inclure une bibliothèque` → `Gérer les bibliothèques` → Rechercher et installer.
2. **Sélectionner le bon board** :

- `Outils` → `Board` → `ESP32 Dev Module`
- `Port` → Sélectionner le port USB de l'ESP32

On a là aussi eu quelques soucis avec Arduino et la connexion USB (problèmes d'autorisations sur Linux). On a pu régler avec : `sudo chmod 777 /dev/ttyUSB0` (à lancer à chaque branchement de l'esp32 pour pouvoir téléverser du code.)

Étape 2 : ESP32.

2.1. Connexion WIFI

La première étape sur l'ESP32 a été de connecter la carte au même réseau que la RPi pour permettre les communications MQTT.
(source)

2.2. Connexion MQTT et test de la communication ESP32 <-> RPi.

(source)

On teste la communication ESP32 ⟷ RPi avec un programme simple :

```
client.publish("test", "ESP32 connecté");
```

Et on vérifie dans le terminal RPi :

```
mosquitto_sub -t "test"
```

voir le code (`arduino.ino`) pour plus de détails.

Dans la fonction callback de mqtt dans Arduino, qui traite les messages reçus de mqtt, on rajoute le changement d'état de la LED selon la commande reçu.

On peut donc directement depuis un terminal RPI publié le message ON ou OFF sur le topic 'esp32/led' auquel est abonné l'ESP32.

Étape 3 : Server HTTP sur RPi.

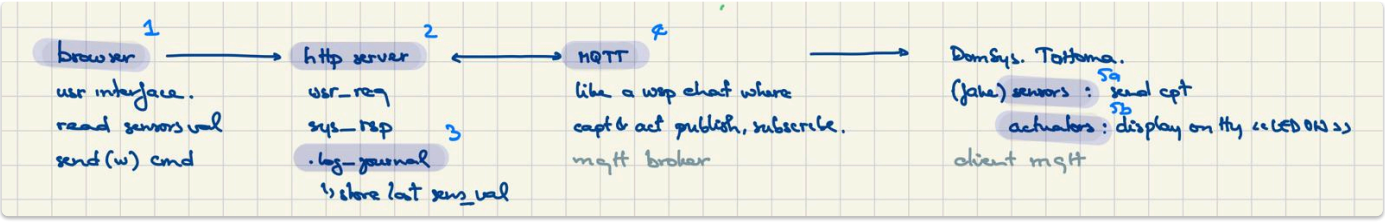
1. Installer les dépendances nécessaires pour executer le code du serveur et communiquer avec mqtt.

```
sudo apt install mosquitto python3-pip
sudo apt install python3-flask
sudo apt install python3-paho-mqtt
```

2. Adapter le code `app.py` (version optimisée pour le matériel)
   - l'essentiel du code est déjà fait dans le proof-of-concept
   - il faut juste modifier les bonnes adresses ip, les bons ports.

On peut ensuite envoyer le code et le template sur la RPi, lancer le script app.py avec l'ESP32 actif.

On peut dire que le projet est terminé. On a bien notre architecture de départ avec toutes les différentes parties qui communiquent entre elles et réagissent par conséquent.



PS : Pour le traitement des valeurs de la photorésistance, on a pensé à 3 différents scénarios :

1. envoi périodique du niveau de luminosité dans la base de données et mis à jour sur l'interface web toutes les 5s (c'est qu'on a finalement fait, car plus intuitif comme scénario). L'avantage, c'est qu'on observe sur le site l'état qui capteur en temps réel. Le hic, c'est ce que ça fait beaucoup de données à stocker dans la base de données. Et aussi quid de la consommation d'énergie.
2. envoi sur demande de l'utilisateur : la valeur de la luminosité est transmise uniquement lorsqu'une requête explicite est effectuée par l'utilisateur. On aura juste à modifier le code un tantinet : l'utilisateur envoie une requête READ_PHOTORES, le serveur transmet à l'ESP32 via un topic `get_photores`, dans le callback, on lit la valeur actuelle de la photorésistance et on fait un `client.publish` sur un autre topic `post_photo_res`. Le serveur lit le message de ce dernier topic, rajoute la valeur à la base de données et réponds à l'utilisateur avec la dernière valeur de la base de données (met à jour l'affichage).
3. envoi lors de variations significatives : le capteur envoie la nouvelle valeur uniquement lorsqu'un changement important de luminosité est détecté, avec un seuil de tolérance paramétrable (par exemple, une variation supérieure à 10%). Cela permet d'éviter les transmissions inutiles en cas de fluctuations mineures (exemple : envoi si la luminosité passe de 50% à 40% ou moins, ou de 50% à 60% ou plus).