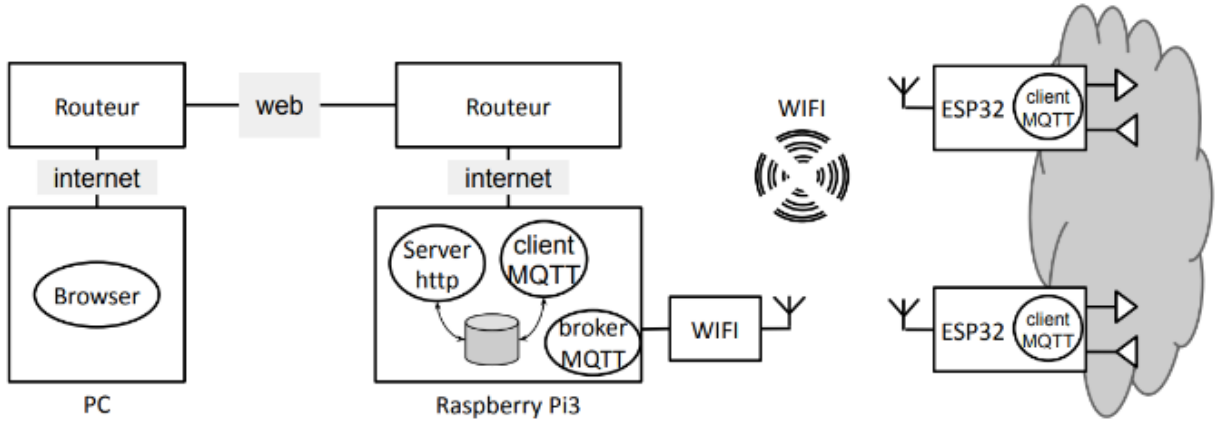
Projet IOC

Introduction et informations

Le but de ce projet est de faire un server HTTP sur une Raspberry Pi 3, qui communique avec plusieurs ESP32 via le protocole MQTT.



voici le sujet (https://largo.lip6.fr/trac/sesi-peri/wiki/IOC_mode_projet).

Auteurs

- Aggoun Tara
- Bordas Florent

La preuve de concepts

Avant de commencer à utiliser la Raspberrypi3 et les EPS32, nous allons faire une preuve de concept sur nos machines.

Etape 1 : Simulation de l'ESP32

Dans un premier temps nous allons écrire en python un programme pour simuler le programme de l'ESP32, pour cela nous allons faire un client MQTT qui publie un compteur.

1 - Installation des package MQTT

Pour installer le package `mosquitto` qui est un démon qui permet de communiquer en utilisant le protocol MQTT.

```
sudo (dnf | apt) install -y mosquitto
```

Pour lancer le serveur :

```
systemctl restart mosquitto
systemctl enable mosquitto
```

On peut maintenant tester que le serveur fonctionne en lançant dans un premier terminal :

```
mosquitto_sub -h localhost -t .
```

Et dans un deuxime terminal :

```
mosquitto_pub -h localhost -t . -m "un message"
```

`un message` devrais s'afficher sur le premier terminal.

2 - Ecrire le programme python

Maintenant que nous pouvons tester des programme qui utilisent pour communiquer un protocole mqtt, nous allons écrire un programme python qui envoie toute les seconde un compteur a un broker MQTT.

Premierement il faut installer la librairie python que nous allons utiliser :

```
pip3 install "paho-mqtt<2.0.0"
```

Installation Alternative (Ubuntu 23.04)

Installation de `virtualenv`

```
sudo apt install python3-virtualenv
```

Création d'un environnement virtuel isolé pour installer la librairie

```
virtualenv paho-mqtt
source paho-mqtt/bin/activate
pip install paho-mqtt
```

On installe une version inférieur a la version 2 car sinon il y a des problème, car certaines fonctions ont changées lors du passage a la deuxieme version.

Initialisation Broker

Une fois installée nous pouvons l'importer dans notre fichier python.
Ensuite nous allons initialiser les information du broker, ici on se connecte en localhost sur le port 1883.

```
    client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION1)
    client.connect('localhost', 1883, 60)
    client.loop_start()
    client.on_connect = on_connect
```

- `client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION1)` : crée une instance d'un client MQTT. str(message.payload)
- `client.connect('localhost', 1883, 60)` : se connecte en localhost (l'adresse du broker) sur le port 1883 et attend au maximum 60 seconde pour se connecter au broker.
- `client.loop_start()` : démarre une boucle de gestion d'evenement en arriere plan qui permet au client MQTT de fonctionner.
- `client.on_connect = on_connect` : permet d'executer la fonction une fois que le client mqtt est connecté au brocker, ici elle affiche juste que on est connecté.

Ensuite nous envoyons notre compteur au broker :

```
counter = 0
while True:
  counter += 1
  client.publish("compteur", str(counter))
  print("Compteur publié : " + str(counter))
  time.sleep(1)
```

Toute les seconde nous envoyons la valeur du compteur au broker avec `client.publish(topic, str(counter))` qui envoie en chaine de charactere la valeur du compteur au broker qui attend le sujet `"compteur"` .

Pour tester notre code on lance dans un premier terminal:

```
mosquitto_sub -h localhost -t compteur
```

Et dans un deuxieme :

```
python3 fake_esp32.py
```

avec `fake_esp32.py` le nom du fichier python.

Etape 2 : Simulation de la Raspberry Pi 3

1 - Le Serveur HTTP

- Pour le serveur HTTP on va utiliser l'API Python : `http.server` qui nous offre toute l'implémentation de base sur laquelle on va pouvoir ajouter des modifications sur les requêtes de type `GET` et `POST` .
    - Pour cela on import la classe `BaseHTTPRequestHandler`
        - `from http.server import BaseHTTPRequestHandler`
    - On va définir note sous-classe `My_HTTP_Handler` basé sur `BaseHTTPRequestHandler` : `class My_HTTP_Handler(BaseRequestHTTPHandler)` et on peut ensuite rajouter nos propres implémentations dans les fonctions `do_POST` et `do_GET` .

- Dans la logique du serveur on va se baser sur le principe d'une API Rest où le client seul initie les échanges. On envoie les requêtes en utilisant un path spécifique pour soit formuler une requête d'input ou de demande de données dans la base de données.

```
def handle_data_exchange(self):
        match self.path:
            case "/input_user":
                data = self.read_json()
                send_esp32(data["topic"], data["val"])
                self.send_json(data)

            case "/request_user/send_button_1":
                data = db.fetch_data("send_button_1")
                self.send_json(data)

            case "/request_user/send_button_2":
                data = db.fetch_data("send_button_2")
                self.send_json(data)
            case "/request_user/send_photoresistance_1":
                data = db.fetch_data("send_photoresistance_1")
                self.send_json(data)
            case "/request_user/send_photoresistance_2":
                data = db.fetch_data("send_photoresistance_2")
                self.send_json(data)
            case _:
                self.handle_http(404, "text/html")

```

- La fonction handle_static sert à gérer les ressources static du serveur (affichage, style…) dans notre cas avec une seul page html cela n'est pas très utile mais nous l'avons rajouté dans l'optique d'avoir un code modulaire.

- La communication via les payload http se fait avec un format json qui est conçue pour faciliter les accès aux données envoyées.

```
    def send_json(self, data: dict):
        self.handle_http(200, "text/json")
        self.wfile.write(bytes(json.dumps(data), "UTF-8"))

    def read_json(self) -> dict:
        content_len = int(self.headers["Content-length"])
        data_string = self.rfile.read(content_len)
        return json.loads(data_string)
```

- Les fonctions `wfile.write` et `rfile.read` servent à lire le stream de données qu'on formattent par la suite sous forme d'une string json qu'on reconvertit objet json. On utilise les fonctions de la librairie suivante qui facilient encodage et le

décodage des objets json :
`https://docs.python.org/3/library/json.html`

2 - Le Client MQTT

Maintenant nous allons faire un client MQTT qui reçois les donnée envoyé sur le topic : `compteur` . Nous initialisons notre client comme dans l'etape1, en ajoutant les choses suivante :

- Dans la fonction `on_connect` on se connecte au topic :

```
def on_connect(client, userdata, flags, rc):
    print("Connecté")
    client.subscribe("compteur")
```

- Dans le main on rajoute :

```
    client.on_message = on_message
```

Cette ligne sert pour quand nous allons recevoir un message, nous allons exécuté la fonction on_message.

- La création de la fonction de reception d'un message :

```
def on_message(client, userdata, message):
    print("Message reçu : " + str(message.payload))
```

Quand on reçoit un message, on l'affiche.

Pour tester notre code on lance dans un premier terminal:

```
python3 client_mqtt.py
```

avec `client_mqtt.py` le nom du fichier python.

Et dans un deuxieme :

```
mosquitto_pub -h localhost -t compteur -m "test"
```

3 - Le Broker MQTT

Le démon `mosquitto` joue le broker. L'installation est décrite dans la partie 1 de l'étape 1.

4 - La Base de donnée

- Pour la base de donnée nous avons opté pour un format Sqlite3 et nous avons utilisé la librairie python3 : `https://docs.python.org/3/library/sqlite3.html` qui nous permet de facilement créer une base de données sous forme d'un fichier `.db` et d'effectuer de simples requêtes sql : `CREATE / INSERT INTO / SELECT FROM .`

5 - Code côté client et gestion des events

Notre interface utilisateur se base sur une page html avec des options d'input pour la communication avec les esp32, et du code javascript pour dynamiser l'affichage et pour récupérer les entrées de l'utilisateur.

```
<div class="inputBox">
        <p class="parag">Led Period</p>
        <input type="text" id="ledField" maxlength="5">
        <button class="button" onclick="setLed()">Set Led Period
</div>
```

Nos 'inputBox' et 'requestBox' vont toutes utiliser le même principe :

- l'attribut `onclick` nous permet de déclencher une réaction dans le code javascript ce qui a pour effet d'appeler une fonction pour gérer la demande de l'utilisateur.
    - Dans notre exemple on a appelé `setLed()` qui va elle même appeler la fonction `apiPost` dans le cas d'une entrée utilisateur ou la fonction `apiGet` dans le cas d'une demande de donnée.

    ```
    async function setLed() {
      apiPost("led", 'ledField');
    }
    ```

- La fonction apiPost est appelée avec les noms des topics correspondant aux périphériques que l'on souhaite utiliser sur les esp32.

- Dans la fonction `apiPost` on utilise l'API `fetch` de javascript qui nous permet de récupérer des données d'une ressource distante sur le principe. On précise si on utilise une méthode POST ou GET et dans le cas d'un POST on fournit un objet json dans le body de la requête que le serveur va récupérer et transmettre via le client MQTT et le broker MQTT aux esp32.

```
async function apiPost(topic, id) {
  const html_element = document.getElementById(id);
  if (target === "") {
    console.log("No Target");
    document.getElementById("error").style = "color: firebrick;";
    return document.getElementById("error").innerHTML = "Please select
  }
  if (target === "esp1") {
    const res = await fetch("/input_user", {
      method: "POST",
      body: JSON.stringify({ topic: topic + "_1", val: html_element.va
    });
    console.log(res);
    const data = await res.json();
    return data;
  }
```

```
async function apiGet(topic) {
  if (target === "esp1") {
    const res = await fetch("/request_user/" + topic + "_1", {
      method: 'GET',
      headers: {
        'Content-Type': 'text/json'
      }
    });
    const data = await res.json();
    return data;
  }
```

- Le fonctionnement de l'API fetch est asynchrone c'est pourquoi on utilise le keyword `async` dans les signatures des fonctions, ainsi que le keyword `await` pour stopper l'exécution en attendant la valeur de retour de la fonction fetch.

- Dans notre architecture on utilise un topic pour déclencher l'écriture des valeurs récupérées par les esp32 via la photorésistance et le bouton. Ainsi, dans les fonctions de demande de données par l'utilisateur, on passe d'abord par un

`apiPost` pour déclencher une réaction puis par un `apiGet` pour aller récupérer l'information dans la base de données.

```
async function buttonGet() {
  data = await apiPost("recv_button", "buttonField");
  if (target === "esp1&esp2") {
    esp1 = await apiGet("send_button_1");
    esp2 = await apiGet("send_button_2");
    document.getElementById("result_esp1_b").innerHTML = esp1.value;
    return document.getElementById("result_esp2_b").innerHTML = esp2.v
  }
  else {
    esp = await apiGet("send_button");
    console.log(esp);
    return document.getElementById("result_"+target+"_b").innerHTML =
  }
}
```

La réalisation sur le matériel

Etape 1 : Client MQTT sur l'ESP32

Nous allons nous baser sur le client MQTT que nous avons fait dans La preuve de concepts. Pour faire un client MQTT sur ESP32 il faut le faire en `Arduino` qui se base sur le C++.

Problème possible : Si en essayant d'éxecuter le code l'erreur `Could not open /dev/ttyUSB0, the port doesn't exist` alors que la device est bien reconnu par l'IDE, la problème est que l'IDE n'a pas les droit. Il faut donc les changé avec la commande

```
sudo chmod 666 /dev/ttyUSB0
```

Configuration de la connection WIFI

Pour communiquer avec la Rasberry Pi, il faut que les deux device soit connecté au même réseau.

Pour se faire nous avons besoin d'importer la libraire `WiFi.h` et définir les deux chaines de characteres `ssid` et `password` qui sont respectivement le nom et le mot de passe du réseau.

Une fois ces deux constante définie, on utilise `WiFi.begin(ssid, password)` pour démarrer la connection, et `WiFi.status()` en véfifiant si le resultat est égale a `WL_CONNECTED` pour s'assurer qu'on est bien connecté au réseau.

Configuration de la connection au broker

Une fois que la connection internet fonctionne il faut configurer la connection au broker. Pour se faire il faut installer et importer la librairie `PubSubClient` .

Il faut de plus définir :

- la chaine de charactere `mqtt_broker` : avec l'adresse ip de la Raspberry Pi ou se trouve le broker.
- les chaine de characteres `mqtt_username` et `mqtt_password` qui peuvent être vide, mais qui serve de sécurité pour pas que n'importe quel client MQTT puisse se connecté et envoyé de fausse information.
- l'entier `mqtt_port` qui contient le port sur lequel on se connecte pour se connecter au broker, par défault il vaut 1883.
- `WiFiClient espClient` qui sert a initialisé `PubSubClient client(espClient)` , l'objet client qui est utilisé pour se connecter au broker MQTT et publier ou s'abonner à des messages sur des topic MQTT.

Gestion des taches

Nous avons deux ESP32 sur chacune d'entre elle, il y a un écran une led une photo-resitance et un bouton poussoire. Pour rendre facile notre communication entre les ESP32 et notre Raspberry Pi, nous avons décidé de faire un `topic` par element.

Pour l'ESP32 i on a donc comme topic (avec i vallant 1 ou 2) :

- const char * `topic_led` = "led_i" : messages venant de la Raspberry Pi vers l'ESP32 pour allumé/étaindre/faire clignoter la led.
- const char * `topic_RPR` = "recv_photoresistance_i" : messages venant de la Raspberry Pi vers l'ESP32 pour demandé la valeur de la photoresistance.
- const char * `topic_SPR` = "send_photoresistance_i" : message venant de l'ESP32 vers la Raspberry Pi repondant la valeur de la photoresistance.
- const char * `topic_SPK` = "speaker_i" : que sur une ESP32, messages venant de la Raspberry Pi vers l'ESP32 pour demander de jouer une musique
- const char * `topic_RBTN` = "recv_button_i" : messages venant de la Raspberry Pi vers l'ESP32 demandant la valeur du bouton.

- const char * `topic_SBTN` = "send_button_i" : messages venant de l'ESP32 vers la Raspberry Pi repondant la valeur du bouton.
- const cher * `topic_OLED` = "oled_i" : messages venant de la Raspberry Pi vers l'ESP32 qui affiche un message sur l'ecran.

Le client MQTT publie sur topic_SPR et topic SBTN et s'abonne a tout les autres.

Pour l'affichage sur l'ecran, la récupération de la valeur de la photo-résistance, la valeur du bouton poussoir ou pour faire clignoter la led nous procedons de la même façon que nous avons fait dans le TME4.

Nous avons une boite au lettre par taches, quand on reçoit un message sur un topic, on rempli la boite au lettre, et on éxécute toute les taches les une apres les autres.

Pour la led, on récupère une période en deciseconde (10^-1 s) qui va faire clignotter la led. On fait clignoter la led en utilisant un timer, toute les période on inverse la valeur de la led et on l'écrit.

Pour la photoresistance, quand on reçoit une demande de la valeur de la photoresistance, on la lit puis l'envoie a la Raspberry Pi.

Pour l'ecran Oled, on affiche des qu'il y a une demande d'ecriture sur l'ecran.

Pour le bouton poussoir, comme pour la photoresistance quand on reçoit une demande de la valeur, on l'a lit et l'envoie au server HTTP via le client mqtt qui se trouve sur la Raspberry Pi.

Enfin pour le buzzer, nous avons une liste de musique, le server http, envoie une requete de demande de musique et l'esp32 la joue.
On a trouver toute nos musique sur ce lien https://github.com/hibit-dev/buzzer/tree/master (https://github.com/hibit-dev/buzzer/tree/master).
On modifie les musique de tel sorte a qu'elle joue une note par une note pour ne pas rester trop de temps sur la tache qui joue la musique, et que les autres taches puissent aussi s'éxécuter en même temps.

Etape 2 : Broker MQTT sur Raspberry Pi 3

Pour avoir un broker sur la Raspberry Pi nous avons décidé d'installer le démon `mosquitto` . Pour l'installer, le démarrer ou le stopper il nous faut les droit root, malheuresement nous n'avions pas le mot de passe de la raspberry pi, on a donc du réinstaller l'os.

Installation de Raspbian

Pour installer Raspbian sur la Raspberry pi il suffit juste d'installer l'outil `rpi-imager` grâce a la commande

```
1 sudo dnf install rpi-imager
```

Une fois installer il faut juste choisir le type de device, ici Raspberry Pi 3, puis l'os conseillé et enfin ou on veut l'installer, ici la carte sd de la raspberry pi.

Après on attend que l'outil flash la carte.
Une fois fini on remet la carte dans la raspberry pi et on la configure avec comme identifiant : `pi` et comme mot de passe `raspberry` .

Installation de Mosquitto

Mosquitto est donc un démon qui va joué dans notre broker mqtt.
On l'installe avec la commande :

```
1 sudo apt install mosquitto
```

Par défault mosquitto n'accepte pas les connection distante (autre que en localhost), il faut donc les configurer. Pour cela il faut rajouter dans le fichier `/etc/mosquitto/mosquitto.conf` :

```
1 listener 1883 0.0.0.0
2 allow_anonymous true
```

Une fois ce modifié il faut lancer/relancer le démon :

```
systemctl restart mosquitto.service
systemctl restart mosquitto
```

Ces deux commandes s'éxecute avec les droits de root.
Le broker MQTT est enfin utilisable, Si on veut voir les log pendant les échanges entre l'ESP32 et la raspberry pi on peut lancer la commande

```
    mosquitto
```

Etape 2 : Client MQTT sur Raspberry Pi 3

Le client MQTT sur Raspberry Pi ressemble beaucoup a ce que on avait fait dans la preuve de concept car c'est un client en python.
Nous avons 14 topics, ils sont tous doubler car nous avons deux ESP32. Nous avons donc deux topic pour les leds, deux pour la reception et deux pour l'envoie de messages par rapport a la photoresistance de même pour le bouton poussoir, deux pour l'affichage de message sur l'ecran Oled et enfin un pour le speaker.

Le client MQTT s'abonne a 4 topic, ceux pour récupéré les valeur des photoresistance et des boutons poussoir.

Quand il reçoit une valeur il l'écrit dans une base de donnée pour que le server HTTP puisse récupéré les information.

Etape 3 : Serveur HTTP sur Raspberry Pi 3

Pour le server HTTP et la base de donnée, on reprend ce que on fait dans la preuve de concepte.

Problème possible : Dans le server http on utilise un match qui est disponible depuis la version 3.10 de python, par défault c'est la version 3.9.2 qui est installé sur la Raspberry. On a du mettre à jour la version de python.

Connexion en SSH à la Raspberry Pi :

- Pour faire plus simple on a utilisé l'interface graphique mais on aurait pu faire tous ces paramétrages avec rpi imager au préalable et après flasher la carte.

- Première étape :

    - Autoriser les connexions ssh sur la raspberry : icône raspberry -> Préférences -> Configuration du Raspberry Pi -> Interfaces -> Cocher l'option SSH.
    - `$sudo raspi-config` -> interfacing options -> ssh -> yes -> finish.

- Deuxième étape trouver l'adresse IP de la raspberry :
    - `$hostname -I` : IP de la raspberry sur le réseau.

- Troisième étape se connecter en ssh :
    - `$ssh pi@Adresse_IP` puis rentrer le mot de passe.

On peut également utiliser la raspberry comme un point d'accès Wi-Fi lorsqu'on n'a pas de routeur à disposition.

- Il faut au préalable se connecter à la raspberry via SSH
- On run la commande suivante: `$sudo nmcli device wifi hotspot ssid <nom de l'AP> password <mot de passe de l'AP> ifname wlan0`
    - Les arguments `ifname` et `wlan0` spécifient qu'il faut utiliser le module Wi-Fi intégré qui supporte le mode de point d'accès pour broadcast un réseau wifi. Si on avait pas de module Wi-Fi on aurait pu spécifier une interface USB qui supporte le mode AP.
- Une fois qu'on a passé toutes les étapes on peut fermer le terminal qui host la connexion ssh et le réseau de la raspberry est disponible.
  (Bien sûr il faut mettre la ligne de commande dans le bashrc au préalable et avoir noté les adresses IP puisque pour se connecter en ssh il faut que la raspberry soit connecté à un réseau).