Eric Haarlemmer
Lucas Grenèche

CS534 : TP Multi-Agent Systems

I - MQTT Basics

I.1 - First Contact

Pour utiliser la bibliothèque paho, il faut créer le client et se connecter au broker, puis indiquer les fonctions callback *on_connect* et *on_message* et lancer la boucle avec client.loop_forever ou loop_start.

La fonction *on_message* s'exécute à chaque message reçu et *on_connect* est exécutée au moment de la connexion du client au broker. Au moment de la connexion on peut *subscribe* aux topics désirés. Les topics sont au cœur de MQTT, il s'agit d'une structure arborescente hiérarchique auxquels les subscribers MQTT s'abonnent afin de recevoir les messages associés.

I.2 - Two clients

Le but de cette partie est de simuler un ping pong avec deux clients qui envoient tour à tour des messages. Un premier client envoie ping sur le topic ping et l'autre répond pong sur le topic pong. Nous récupérons le nom du topic grâce aux arguments passés au moment de l'exécution.

```python
def on_connect(client, userdata, flags, rc):
  print("Connected with result code "+str(rc))
  client.subscribe(sys.argv[1])

def on_message(client, userdata, msg):
    print("received message: ", end='')
    print(msg.payload.decode())
    time.sleep(1)
    if msg.payload.decode() == sys.argv[1]:
        client.publish(sys.argv[2], sys.argv[2])

client = mqtt.Client()
client.connect("localhost",1883,60)

client.on_connect = on_connect
client.on_message = on_message

client.publish(sys.argv[2], sys.argv[2])
client.loop_forever()
```

Figure 1: code pour le ping pong

Pour lancer ce code, on lance dans un terminal le programme avec les arguments ping pong et l'autre pong ping : *python3 TP1_exo2.py <topic_reception> <topic_emission>*
Si on passe par un seul topic, le client reçoit le message qu'il vient d'envoyer dessus. On est donc obligé de filtrer les messages reçus par le contenu. C'est donc plus pratique de passer par plusieurs topics.

Pour lancer le ping pong avec un seul processus, on le fait dans le script python *launch_pingpong.py* qui avec des subprocess lance chaque client avec les arguments appropriés.

II - Sensor Network

II.1 - Agents or nodes

Sensors

Les capteurs sont les composants essentiels de notre sensor network. Un capteur est lancé avec la commande suivante :
python3 sensor.py <room> <datatype> <appareil>
Il publie sur le topic *room/datatype/appareil* des données aléatoires comprises entre 20.0 et 25.0,et ce toutes les secondes, simulant un capteur IoT réel.

Averaging Agent

Les agents de moyenne rassemblent les données des capteurs qui sont présents dans une même pièce pour un même type de données. Il est lancé avec la commande suivante : *python3 averaging.py <room/datatype>*
Pour cela, l'agent s'abonne au topic *room/datatype/+* ce qui lui permet de recevoir les valeurs provenant de tous les capteurs correspondant à ce type dans la pièce. Ensuite il

stocke un nombre prédéfini de ces messages, ici 10, puis calcule la moyenne des valeurs de ceux-ci, et publie le résultat sur le topic *room/datatype/average.*

Interface Agent

Cet agent permet d'afficher les données produites par les agents de moyenne, et les affiche pour l'utilisateur. On peut suivre une seule moyenne en lançant l'agent avec : *python3 Interface.py <topic> ex: chambre/temperature/average*

Ou surveiller toutes les moyennes en s'abonnant à tous les topics average :
python3 Interface.py +/+/average

Ce mode permet d'observer l'ensemble des rooms et datatypes.

II.2 - Dynamics

Pour la suite du projet, et pour ne pas avoir à lancer à la main des dizaines de capteurs, nous avons créé le script python *sensors_sim.py.* Ce simulateur crée et gère automatiquement des capteurs dans un environnement dynamique.
Dans ce programme, nous avons spécifié des pièces (cuisine, chambre, sdb et salon), et des types de données (CO2, Température, Humidité). La simulation comporte plusieurs phases. La première phase est l'initialisation : 3 capteurs sont lancés pour chaque données et ce dans chaque pièce, soit un total de 3*3*4 = 36 capteurs.
Ensuite, c'est la phase de simulation, qui fonctionne par cycle. Au début de chaque nouveau cycle, on vient pour chaque capteur le supprimer avec une certaine probabilité (ici 30%) et on vient ajouter un nouveau capteur avec une certaine probabilité (ici 80%), dans une zone aléatoire.
Enfin, la simulation se termine lors d'une interruption manuelle de l'opérateur, avec un Ctrl-C dans le terminal, et tous les processus liés aux capteurs sont tués proprement.

II.3 - Anomaly detection

Detection Agent

Cet agent permet de détecter une anomalie dans une moyenne de capteurs. Il s'abonne au topic */room/datatype/*+ pour accéder aux informations des capteurs d'un certain type dans une certaine pièce, et récupère dans un buffer avec un principe de fenêtre glissante les 5 dernières valeurs envoyées par le capteur. Ensuite, on part du principe qu'un capteur a un comportement qui est anormal si la moyenne des valeurs du buffer dépasse de plus de 2 écarts-types la moyenne. Si un tel capteur est détecté, un message d'alerte est publié sur le topic zone/datatype/sensor_id/alert. Dans ce message d'alerte on y met l'id du capteur, la zone, le type de donnée, la valeur anormale relevée ainsi que la moyenne et l'écart-type utilisé pour détecter le dépassement.

Identification Agent

Cet agent permet de détecter le capteur responsable de l'anomalie relevée par l'agent de détection. Il a deux mode de fonctionnement, soit il s'abonne au topic alerte d'une certaine donnée d'une certaine piece avec *python3 identification_agent.py salon/temperature/+/alert,* Mais il peut aussi s'abonner à tous les topics alert si on lui donne en paramètre *+/+/+/alert*.
L'agent extrait l'id du capteur qui pose problème à partir du topic, et publie le message "*reset"* de restart sur le topic zone/datatype/sensor_id/reset. Il faut donc modifier le code de sensor.py pour que chaque capteur s'abonne à ce topic, et en cas de réception du message reset, renvoie le message suivant : "Capteur {sensor_id} : comportement anormal détecté par l'identification agent, redémarrage…" : une fois le message "reset" envoyé, le capteur se déconnecte, se reconnecte, et se remet à publier normalement, simulant la remise en état du capteur.

III - Contract Net - Machines and Jobs

III.1 - Functional version

Pour résoudre ce problème, l'émission d'un CPF, les proposals, et les acceptations se font de la manière suivante:

- D'abord le superviseur fait une proposition sur un topic CPF/x ou x est le numéro de la tâche à réaliser.
- Les machines répondent Proposal ou Reject sur le topic proposal/xxx ou xxx est le nom de la machine (les machines déjà sur une tâche refusent la nouvelle tâche proposée, les autres ont une chance sur deux d'accepter ou de refuser).
- Le superviseur mémorise les noms de machines qui ont répondu à Proposal dans une liste.
- Dès que la deadline est atteinte, le superviseur sélectionne au hasard dans les machines disponibles et envoie accept sur le topic acceptation/{machine_acceptée}. Il envoie aussi reject sur acceptation/{machine_rejetée}
- La machine qui reçoit l'acceptation commence la tâche et refuse toutes propositions de tâches tant que sa tâche n'est pas finie.

Nous séparons donc les propositions et les réponses aux machines en ayant un topic par machine sur les propositions et les acceptations. Le contenu des messages ensuite donne donc la réponse si la machine est acceptée ou si la machine se propose pour une tâche.

```
machine2 proposal for job n° 2
machine1 proposal for job n° 2
selected machine: machine2
machine2 rejected job n° 3
machine1 proposal for job n° 3
selected machine: machine1
```

Figure: Sortie du superviseur

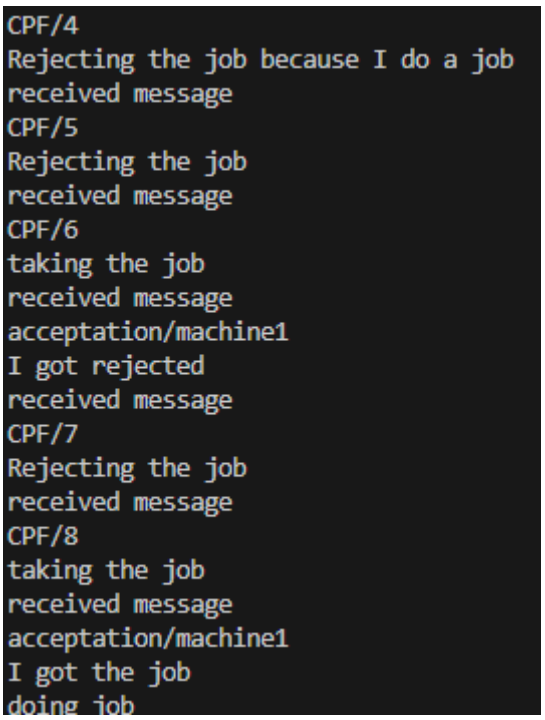

Figure: Sortie pour la machine 1

Grâce aux figures ci-dessus, on observe que le comportement est bien celui voulu. Quand une machine a déjà un job, elle le refuse, quand elle se propose pour une tâche, elle est tirée au sort et s'il n'y a pas de machines proposées, la tâche n'est pas faite.

IV Conclusion

Dans le projet, nous n'avons pas utilisé de threading mais le module subprocess de python, qui nous permet de lancer un processus par capteur, ce qui se rapproche plus du comportement d'un vrai système distribué. Nous avons choisi le langage python pour sa simplicité et du fait que le sujet du TP nous guidait vers ce langage avec la librairie paho-mqtt.
Enfin, durant le TP nous avons eu un problème avec des variables d'état des capteurs dans les fonctions *on_message* et *on_connect*, que nous n'avions pas passées en global.