Documentation technique  
Communication MQTT

La communication entre le superviseur et les 3 automates est un module indispensable afin de pouvoir contrôler et d’accéder aux données des robots à tout moment, principalement pour obtenir l’état du robot ainsi que sa position. Pour effectuer cette communication le plus efficacement possible, nous nous sommes dirigés vers le protocole de messagerie publish-subscribe MQTT (Message Queuing Telemetry Transport) basé sur le protocole TCP/IP.

Le client MQTT est conçu pour les connexions aux sites distants, ce qui nous est grandement utile dans le cadre de la fabrication de notre flotte de robots.

En ce qui concerne le broker, nous nous sommes dirigés vers eclipse-mosquitto qui est un broker open source utilisant le protocole MQTT.

1. **Mise en place de l’environnement MQTT**

Pour commencer, nous devons mettre en place l’environnement MQTT sur chacun des robots. Cela nous permettra ensuite de pouvoir transmettre des données et effectuer une connexion entre les différents opérateurs.

Pour cette partie du projet, nous travaillerons sur linux, plus précisément sur Ubuntu 16.04. Nous allons installer MQTT ainsi que Mosquitto sur les 3 robots, serveur, préparateur et accueil, ainsi que sur le superviseur. Pour ce faire, des simples lignes de commande suffisent1.

$ pip install paho-mqtt

$ sudo apt-get install mosquitto mosquitto-clients python-mosquitto

Une fois le client et le broker installé, nous pouvons commencer directement à envoyer et à réceptionner des messages.

1. **Envoi et réception**

Grâce à la communication effectuée à la différente partie, il ne nous reste plus qu’à créer nos programmes qui nous permettront de remplir cet objectif.

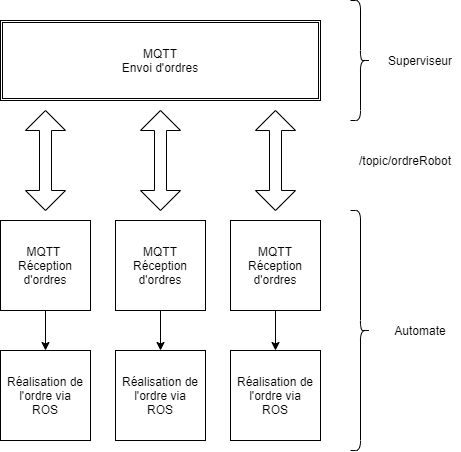
Le broker Mosquitto propose une librairie Python et C qui permettent d’implémenter le client MQTT.

Les programmes peuvent être programmés en Python ou en C, qui sont les langages principaux de Mosquitto, grâce aux librairies proposés par le broker. Nous nous concentrerons donc sur ces 2 langages.

Nous allons effectuer d’une part **la réception des données par le robot de la part du superviseur**, qui se caractérise par la réception et la réalisation d’un ordre par le robot, ainsi que **la réception des données par le superviseur de la part du robot**, qui se caractérise par la réception de l’état et du niveau de batterie du robot.

**A. Réalisation d’un ordre par le robot**

La communication entre le robot et le superviseur s’effectue de la façon suivante :



L’envoi et la réception de l’ordre s’effectue via un topic, ici */topic/ordreRobot*. Le superviseur est le *publisher*, qui envoie un message sur le topic, tandis que notre robot est un *suscriber* de ce topic, qui lit les informations reçues.

1. **Programme implémenté sur chacun des robots**

#!/usr/bin/env python3

import paho.mqtt.client as mqtt

import paho.mqtt.subscribe as subscribe

import os

def on\_connect(client, userdata, flags, rc):

print("Connected with result code "+str(rc))

if rc==0:

print("Connexion réussie")

else:

print("Connexion impossible")

client.subscribe("topic/ordreRobot", qos=2)

def on\_disconnect(client, userdata, rc):

print("Client Got Disconnected")

def on\_message(client, userdata, msg):

print("Le message suivant a bien été reçu : " , str(msg.payload))

print("message topic = ",msg.topic)

print("message qos = ",msg.qos)

print("message retain flag = ",msg.retain)

if ((str(msg.payload.decode("utf-8"))) == "1"):

print("Lancement de roslaunch superviseur superviseur.launch")

os.system("bash /home/<user>/Desktop/superviseur.sh")

if \_\_name\_\_ == '\_\_main\_\_':

client = mqtt.Client()

client.connect("192.168.1.4",1883,60)

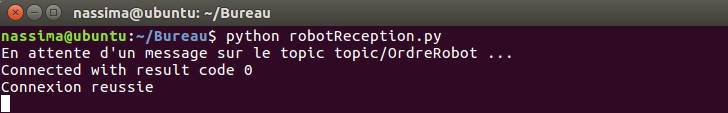
client.on\_connect = on\_connect

print("En attente d'un message sur le topic topic/OrdreRobot ...")

client.on\_message = on\_message

client.loop\_forever()

En lançant le programme Python sur le robot, on obtient un automate qui est en réception continu en attendant de lire un message sur le topic /topic/ordreRobot.



1. **Programme implémenté sur le superviseur**

#!/usr/bin/env python3

import paho.mqtt.client as mqtt

def on\_connect(client, userdata, flags, rc):

print("Connected with result code "+str(rc))

if rc==0:

print("Connexion réussie")

else:

print("Connexion impossible")

client.publish("topic/ordreRobot", "1", qos=2)

def on\_disconnect(client, userdata, rc):

print("Client Got Disconnected")

if \_\_name\_\_ == '\_\_main\_\_':

client = mqtt.Client()

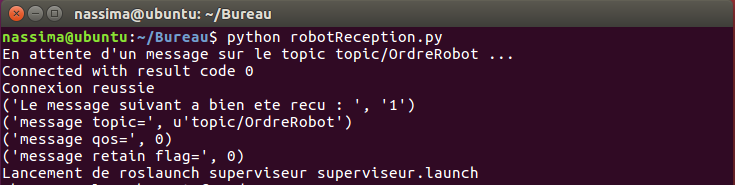
client.connect("192.168.1.4",1883,60)

client.on\_connect = on\_connect

client.on\_disconnect = on\_disconnect

Le lancement de ce programme permet la publication *unique* d’un message sur le topic ordreRobot. Nous avons configuré le programme de réception de tel sorte à ce qu’il réagisse lorsqu’il reçoit le message « 1 » de la part du superviseur.

Ainsi, une fois le message lancé, on obtient le résultat suivant dans le programme de la réception situé sur le robot :



1. **Explication détaillée des programmes**

Pour connecter les 2 automates, nous utilisons l’adresse IP 192.168.1.4, qui est l’adresse de notre superviseur. Il est indispensable que tous les robots se connectent à la même adresse IP afin de communiquer entre eux.

Une fois connecté, le robot se met en mode réception en continu grâce à la boucle *client.loop\_forever()*. Le message envoyé, quant à lui, cependant, ne se lance qu’une fois dans le cas de l’envoi d’un ordre, du fait que les robots n’effectuent pas toujours la même action.

Une fois le message lancé par le superviseur, le robot analyse si le message est le chiffre « 1 ». Si c’est le cas, il effectue l’action désirée par le superviseur. Ici, l’action désirée est le lancement du bash superviseur.sh. Dans ce fichier bash, nous avons le lancement du launcher *superviseur.launch*, qui est un outil permettant de lancer plusieurs nodes ROS localement.

Le contenu de notre launcher est le suivant :

<launch>

<node name="drive" pkg="robotino\_rest\_node" type="omnidrive.py" output="screen"/>

<node name="square" pkg="superviseur" type="square.py" output="screen"/>

</launch>

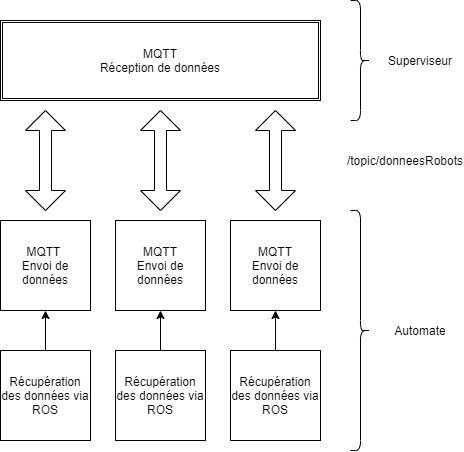
Pour nos tests en cours de réalisation, le launcher permet de lancer le node omnidrive.py, qui lit les vitesses du robot, ainsi que de lancer le script square.py afin de faire effectuer au robot un carré.

Enfin, nous avons décidé d’un QoS (Quality of Service) pour les messages envoyés et reçus via MQTT. Le QoS est un accord entre l’expéditeur et le destinataire sur la garantie de remise d’un message. Il existe 3 niveaux de QoS : 0, 1 et 2. Le niveau 0 correspond à la méthode d’envoi de message la plus simple et la moins lourde : le client publie le message sans accusé de réception. Le niveau 1 garantit que le message sera transféré au moins une fois. Dans le cas où l’accusé de réception est perdu, cela signifie que l’expéditeur renverra le message. Il est donc possible de recevoir plusieurs fois le message. Enfin, le niveau 2 est le niveau de service le plus élevé grâce à une séquence de 4 messages, afin de confirmer que le message a été envoyé, et que l’accusé de réception a bien été reçu. Le message sera donc bien envoyé une seule et unique fois.

Dans notre cas d’envoi d’ordres, nous avons décidé d’utiliser le niveau 2, soit le niveau le plus élevé. En effet, dans le cas où il nous faudrait envoyer un ordre à un robot, il est indispensable que le message soit bien reçu, donc l’accusé de réception est primordial, mais il est également très important à ce que le message ne soit reçu qu’une fois, afin d’éviter que le robot serveur serve plusieurs fois la même commande à la même table, par exemple.

**B. Réception des informations des robots**

Dans ce second cas, la communication s’effectue de la même manière que précédemment, à la différence étant que l’envoi s’effectue à présent fois de la part du robot, qui envoie son état (0 ou 1 en fonction de son état actuel) ainsi que son niveau de batterie au superviseur.



Ainsi, cela nous permettra de lire l’état et le niveau de charge de la batterie de chaque robot à une certaine fréquence que nous choisirons et de savoir à tout moment si le robot nécessite d’être chargé ou si un robot est libre d’utilisation.

Les résultats et la manière de faire sont exactement les mêmes, excepté le fait que :

1. Il n’est pas nécessaire cette fois-ci de lancer un launcher en fonction du message reçu ; la seule chose que nous désirons c’est récupérer les données du robot pour une utilisation ultérieure.
2. Le QoS n’a pas besoin d’être initialisé à 2 pour cette partie. En effet, nous pouvons utiliser un

QoS à 0, soit le niveau le plus faible, ce qui nous permettra d’une part d’alléger notre communication, et d’autre part parce que l’accusé de réception n’est pas indispensable étant donné que le message s’envoie en continu, à une fréquence que nous avons définie à 0.2 Hz pour un intervalle de 5 secondes.

1. **Programme implémenté sur le superviseur**

#!/usr/bin/env python

import paho.mqtt.client as mqtt

import paho.mqtt.subscribe as subscribe

import os

def on\_connect(client, userdata, flags, rc):

print("Connected with result code "+str(rc))

if rc==0:

print("ok")

else:

print("no")

client.subscribe("topic/donneesRobots", qos=0)

def on\_disconnect(client, userdata, rc):

print("Client Got Disconnected")

def on\_message(client, userdata, msg):

print("Yes! i receive the message :" , str(msg.payload))

print("message received " ,str(msg.payload.decode("utf-8")))

print("message topic=",msg.topic)

print("message qos=",msg.qos)

print("message retain flag=",msg.retain)

if \_\_name\_\_ == '\_\_main\_\_':

print("programme lance")

client = mqtt.Client()

client.connect("192.168.1.4",1883,60)

client.on\_connect = on\_connect

client.on\_message = on\_message

client.loop\_forever()

1. **Programme implémenté sur chacun des robots**

# !/usr/bin/env python

import paho.mqtt.client as mqtt

import time

def on\_connect(client, userdata, flags, rc):

print("Connected with result code "+str(rc))

if rc==0:

print("ok")

else:

print("no")

def on\_disconnect(client, userdata, rc):

print("Client Got Disconnected")

if \_\_name\_\_ == '\_\_main\_\_':

client = mqtt.Client()

client.connect("192.168.1.4",1883,60)

client.on\_connect = on\_connect

while True :

client.on\_disconnect = on\_disconnect

client.publish("topic/donneesRobots", "50% de batterie", qos=0)

time.sleep(5)

1. **Explication détaillée des programmes**

*A compléter une fois les variables de la batterie récupérées*

1. **Démarrage des scripts au boot grâce aux services systemd**

Une fois tous les scripts nécessaires à la communication MQTT programmés, il ne nous reste plus qu’à nous occuper du démarrage automatique des scripts au boot des robots ainsi que du superviseur.

Pour ce faire, nous utilisons systemd. Systemdest une suite logicielle qui possède des composants systèmes destinés aux systèmes d’exploitation Linux. Nous avons choisi de l’utiliser car il permet le chargement en parallèle des services au démarrage et de réduire les appels aux scripts shell.

**A. Comment créer un service sur Linux ?**

Nous allons implémenter divers services sur le superviseur ainsi que sur les différents robots. Voici un exemple de création de service2 :

**Nous commençons par créer un fichier test.service grâce à la commande suivante :**

$ sudo nano /etc/systemd/system/test.service

**Et nous y plaçons le contenu suivant :**

[Unit]

Description = Lancement du service de test

After = multi-user.targe

[Service]

Type = simple

Restart = always

ExecStart = /usr/bin/python /home/<user>/Desktop/test.py

[Install]

WantedBy = multi-user.target

**Explication détaillée du programme :**

**Un service est composé de 3 blocs : Unit, Service et Install.**

**Dans la partie *Unit*, nous avons la description du service.**

**Dans la partie *Service*, nous avons le type de service, la façon de relancer le service. Ici, nous avons réglé le flag Restart à *always* car nous souhaitons relacer le service si le serveur doit être redémarré ou s’il crashe. Nous avons également la ligne ExecStart qui prend en paramètre 2 arguments : le premier argument est le chemin de l’environnement. Par exemple, si nous désirons lancer un script Python, le chemin de l’environnement serait /usr/bin/python. Le deuxième argument est le chemin du script qui doit être exécuté, ici, un script python. Dans notre exemple, nous lançons un script python *test.py* présent sur le bureau.**

**Dans la partie Install, nous avons ce par quoi le service est demandé. Par exemple, si nous souhaitons lancer un node ROS, nous pouvons mettre dans cette ligne roscore, car sans roscore, le node ne peut pas être lancé. Pour l’instant, nous utilisons le paramètre par défaut.s**

**Il reste encore de très nombreux paramètres que nous pouvons implémenter dans notre service. Nous pouvons retrouver la liste des différentes configurations possibles facilement sur internet3.**

**A présent, nous redémarrons daemon grâce à la commande suivante. Cette étape est indispensable lorsque nous effectuons des modifications dans les fichiers de service.**

$ sudo systemctl daemon-reload

**Nous activons notre service afin qu’il ne se désactive jamais si nous redémarrons.**

$ sudo systemctl enable test.service

**Et, pour finir, nous démarrons notre service.**

$ sudo systemctl start test.service

Voilà, notre service est mis en place. Nous pouvons obtenir l’état d’un service ainsi que l’historique de ses actions :

$ sudo systemctl status test.service

Nous appliquons donc ce mini-tutoriel pour chacun des services qui suivent que nous allons implémenter.

**B. Mise en place des services sur les robots**

1. **Exemple d’un service roscore : roscore.service**

[Unit]

Description = Lancement de roscore

After = multi-user.targe

[Service]

Type = forking

User = <user>

Group = <user>

Restart = always

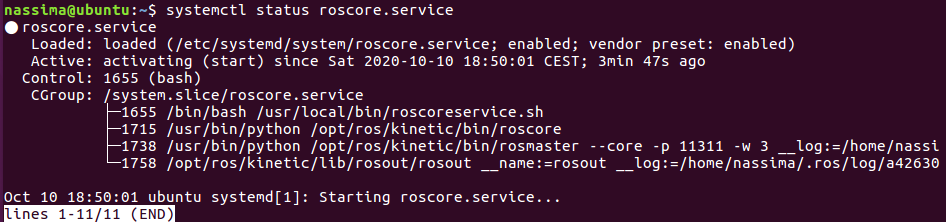
TimeOutSec = 5min

ExecStart = /bin/bash /usr/local/bin/roscoreservice.sh

[Install]

WantedBy = default.target

Voici le résultat obtenu en analysant le statut de notre service *roscore.service* au boot de notre ordinateur :



Nous pouvons voir que le service est bien actif et donc que *roscore* est en train de tourner en arrière-plan dès le démarrage.

1. **Exemple d’un service d’écoute : heronReception.service**

[Unit]

Description = Lancement de l’écoute sur le topic… En attente d’un ordre…

After = multi-user.target

[Service]

Type = simple

User = <user>

Group = <user>

Restart = always

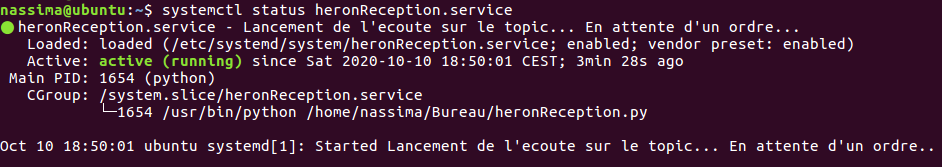
TimeOutSec = 5min

ExecStart = /usr/bin/python /home/<user>/Bureau/heronReception.py

[Install]

WantedBy = default.target

Voici le résultat du statut de notre service au boot :

****

Nous pouvons voir que notre service est bien en train de tourner et en attente de recevoir un message sur le topic, comme vu dans la partie précédente avec notre script python.

1. **Exemple d’un service d’envoi**

[Unit]

Description = Envoi des donnees du robot vers le superviseur…

After = multi-user.target

[Service]

Type = simple

User = <user>

Group = <user>

Restart = always

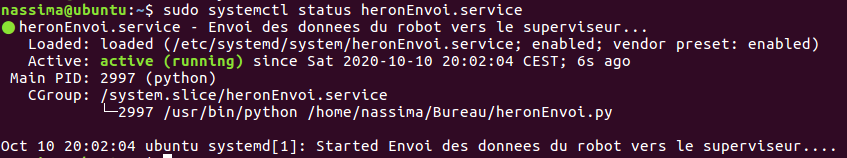
TimeOutSec = 5min

ExecStart = /usr/bin/python /home/<user>/Bureau/heronEnvoi.py

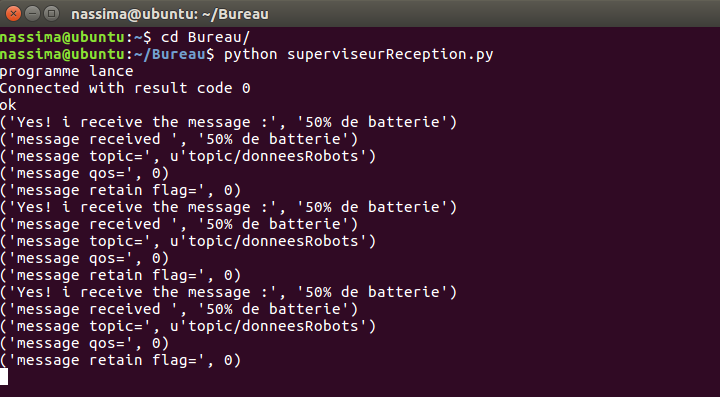
[Install]

WantedBy = default.target

Notre service est bien en train de tourner dès le démarrage du robot et lance le script heronEnvoi :

****

Nous pouvons tester en activant l’écoute du topic donneesRobots sur lequel est censé publier le script *heronEnvoi.py*. On reçoit bien le message correspondant toutes les 5 secondes :

****

**C. Mise en place des services sur le superviseur**

1. **Exemple d’un service d’écoute : superviseurReception.service**

[Unit]

Description = Ecoute des donnees du robot…

After = multi-user.target

[Service]

Type = simple

User = <user>

Group = <user>

Restart = always

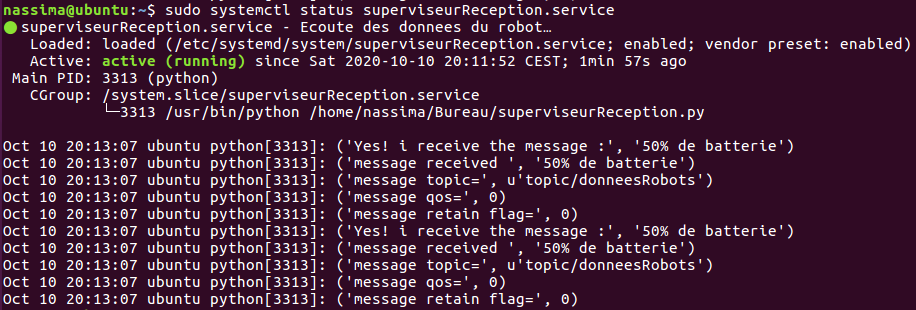
TimeOutSec = 5min

ExecStart = /usr/bin/python /home/<user>/Bureau/superviseurReception.py

[Install]

WantedBy = default.target

On reçoit bien automatiquement les messages (publiés eux aussi automatiquement) sur le superviseur.



Bibliographie

1 Installation MQTT : <https://pypi.org/project/paho-mqtt/>

2 Services :

<https://medium.com/@wasiullah.khan21/setup-a-python-script-as-a-service-through-systemctl-systemd-f0cc55a42267>

<https://blog.roverrobotics.com/how-to-run-ros-on-startup-bootup/>

<https://answers.ros.org/question/290599/how-to-run-roscore-or-roslaunch-when-i-boot/>

<https://www.rosehosting.com/blog/how-to-list-all-services-in-linux/>

<https://unix.stackexchange.com/questions/579264/systemd-service-ends-up-in-inactive-dead-after-boot>

<https://linuxconfig.org/how-to-use-systemctl-to-list-services-on-systemd-linux>

3 Configuration des services :

<https://www.freedesktop.org/software/systemd/man/systemd.unit.html>