Lab : MQTT

Filière : IR&C

Realised by : AL ABOUDI Imane   BENABDALLAH Redouane

1 Introduction

The objective of this lab is to discover the MQTT communication protocol through practical experiments using Python and the Eclipse Paho MQTT client library. The lab starts with basic publish/subscribe communication and simple interactions between clients. It then explores a multi-agent sensor network, including data aggregation, anomaly detection, and recovery mechanisms. Finally, the lab implements the Contract Net Protocol to illustrate decentralized coordination between autonomous agents using MQTT.

2 MQTT Basics

2.1 First Contact

The goal of this first experiment was to understand the basic principles of the MQTT publish/subscribe communication model and to verify the correct operation of the broker and clients. To achieve this, we used shiftr.io as a local MQTT broker and implemented simple clients in Python using the Eclipse Paho MQTT library.

A first Python script was developed as a publisher. This client connects to the broker and periodically sends text messages on a topic named `Hello`. Each message is published at a fixed time interval, illustrating the asynchronous nature of MQTT communication. In parallel, a second script was implemented as a subscriber. This client subscribes to the same topic and continuously listens for incoming messages, displaying them as soon as they are received.

This initial setup allowed us to validate several key aspects of MQTT : successful connection to the broker, correct message publication, and reliable message reception through subscriptions. It also demonstrates one of the main advantages of MQTT, namely the decoupling between publishers and subscribers, which communicate only through topics without any direct knowledge of each other.

2.2 Two Clients Interaction

To further explore bidirectional communication using MQTT, we implemented an interactive scenario involving two clients, named `ping` and `pong`. Both clients are based on the same Python script but are started with different roles using a PowerShell automation script(we worked with Windows). Each client acts simultaneously as a publisher and a subscriber.

The communication relies on two distinct topics : `ping` and `pong`. When the ping client starts, it publishes a message on the `ping` topic and then waits for a response on the `pong` topic. Conversely, the pong client subscribes to the `ping` topic and replies by publishing a message on the `pong` topic. This exchange continues in a loop, creating a simple request response interaction.

This experiment highlights the use of topics to structure interactions.

3 Sensor Network

3.1 Overview

In this part, we implemented a small multi agent system simulating a sensor network using MQTT. Each agent is implemented as an independent MQTT client, communicating exclusively through topics.

The system is composed of three types of agents :

— **Sensor agents**, which generate and publish measurements,

— **An averaging agent**, which collects sensor data and computes averages,

— **An interface agent**, which displays aggregated results.

3.2 Sensor Agents

Sensor agents simulate temperature sensors deployed in a specific area. In our implementation, all sensors belong to the `living_room` zone and periodically generate temperature values. The measurements evolve over time according to a sinusoidal function, which provides realistic variations.

Each sensor publishes its readings on a topic structured as `/zone/measurement/sensor_id`.This topic hierarchy embeds information directly into the topic name, making it easy for other agents to filter messages based on location, measurement type, or sensor identity. Sensors operate independently and publish data at regular intervals.

3.3 Averaging Agent

The averaging agent is responsible for collecting sensor readings and computing aggregated values. It subscribes to all temperature sensor topics in the living room using a wildcard subscription `/living_room/temperature/+`. Each received value is stored temporarily, and once a predefined number of samples is reached, the agent computes the average temperature.

The computed average is then published on a separate topic, `/living_room/temperature/average`. Using a dedicated topic for aggregated data ensures a clear separation between raw sensor measurements and processed information, simplifying the design of other agents that rely on these results.

3.4 Interface Agent

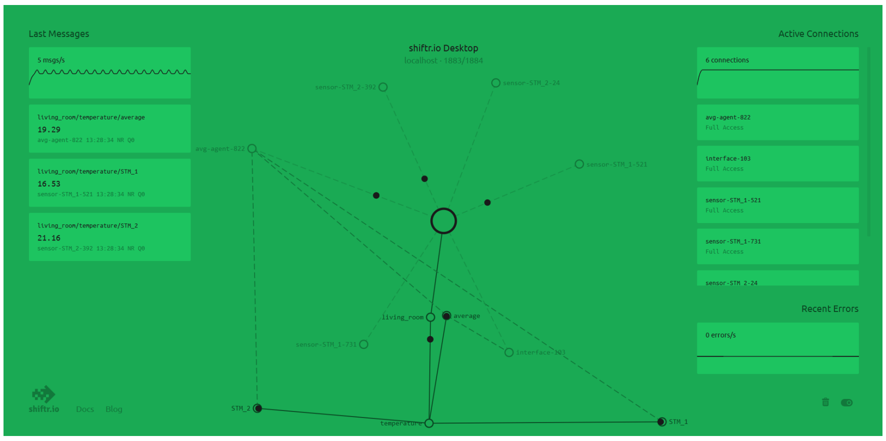
The interface agent provides a simple visualization of the aggregated data produced by the averaging agent. It subscribes to all average topics using the wildcard `/+/+/average`, which allows it to receive averages from any zone or measurement type.

When a message is received, the agent extracts contextual information such as the zone and measurement type directly from the topic name and displays the result in a readable format. This agent does not need any knowledge about the sensors or the averaging process, making it fully independent and easily reusable.

3.5 Anomaly Detection

3.5.1 Detection Agent

To extend the sensor network with fault detection capabilities, we introduced a detection agent responsible for monitoring sensor readings in real time and identifying abnormal behavior. This agent subscribes to all temperature sensor topics using the wildcard subscription


Last Messages

5 msgs/s

```
living_room/temperature/average
19.29
avg-agent-822 13:28:34 NR Q0

living_room/temperature/STM_1
16.53
sensor-STM_1-521 13:28:34 NR Q0

living_room/temperature/STM_2
21.16
sensor-STM_2-392 13:28:34 NR Q0
```

Active Connections

6 connections

```
avg-agent-822
Full Access

interface-103
Full Access

sensor-STM_1-521
Full Access

sensor-STM_1-731
Full Access

sensor-STM_2-24
```

Recent Errors

0 errors/s

FIGURE 1 – Results

/living_room/temperature/+, allowing it to receive measurements from all sensors without prior knowledge of their identities.

Each received value is associated with its sensor identifier, which is extracted directly from the topic name. The agent maintains a history of recent readings for each sensor and computes the global mean and standard deviation across all collected values. An anomaly is detected when a sensor produces a value that deviates from the system average by more than two standard deviations. When such a situation occurs, the detection agent publishes an alert message on a dedicated topic /living_room/temperature/alert, explicitly listing the identifiers of the suspected sensors.

This approach leverages both the hierarchical topic structure of MQTT and a simple statistical method to identify abnormal readings while remaining lightweight and fully decentralized.

3.5.2 Identification Agent

Following anomaly detection, an identification agent is used to react to alerts and initiate corrective actions. This agent subscribes to the alert topic /living_room/temperature/alert and waits for anomaly notifications emitted by the detection agent. Upon receiving an alert, it extracts the list of suspicious sensor identifiers contained in the message payload.

For each identified sensor, the agent publishes a reset command on a dedicated control topic of the form /{sensor_id}/reset. This mechanism allows the identification agent to directly address faulty sensors without interacting with unaffected ones. The separation between detection and identification responsibilities improves modularity and mirrors real-world monitoring architectures where analysis and control are handled by distinct components.

3.5.3 Sensor Update

To support the identification and recovery process, sensor agents were modified to handle reset commands. Each sensor now subscribes to its own reset topic and implements a callback function that reinitializes its internal state when a reset message is received. In our implementation, this reset operation consists of restarting the internal time variable used to generate temperature values.

This modification allows sensors to dynamically react to control messages while continuing to publish measurements autonomously. As a result, the system can not only detect faulty behavior but also actively recover from it, demonstrating a complete monitoring, diagnosis, and correction loop implemented entirely through MQTT communication.

4 Contract Net - Machines and Jobs

In this section, we implemented the classical Contract Net Protocol using MQTT to coordinate the execution of jobs among multiple machine agents. Each machine is modeled as an autonomous MQTT client capable of bidding for jobs according to its capabilities, while a supervisor agent is responsible for generating jobs, collecting bids, and assigning each job to the most suitable machine. All coordination is achieved through asynchronous message exchanges over MQTT topics, without any direct communication between machines.

4.1 Machine Agent

Each machine is implemented as an independent MQTT client responsible for executing jobs. A machine is characterized by a unique identifier and a set of job capabilities, where each supported job type is associated with a required execution time. When the machine receives a Call for Proposal message on the topic `/cnp/cfp`, it evaluates whether it is currently available and whether it can perform the requested job type.

If the machine is capable and not busy, it responds by publishing a bid containing the estimated execution time. Otherwise, it sends a rejection message. The bid is published on a topic of the form `/cnp/bid/{machine_id}`, which embeds the machine identifier directly into the topic. This design choice allows the supervisor to retrieve the machine ID from the topic name itself, without requiring additional metadata in the message payload.

Once a machine receives an assignment message on its dedicated topic `/cnp/assign/{machine_id}`, it switches to a busy state and simulates job execution by sleeping for the agreed duration. During this time, the machine ignores incoming CFP messages, ensuring that it cannot be assigned multiple jobs simultaneously. After completion, the machine becomes available again.

4.2 Supervisor Agent

The supervisor agent is responsible for coordinating the job allocation process using the Contract Net protocol. It generates a sequence of jobs, each defined by a unique identifier and a job type, and initiates a negotiation round by broadcasting a CFP message on the shared topic `/cnp/cfp`. This message is sent to all machine agents simultaneously, allowing every available machine to evaluate the request independently.

To collect responses, the supervisor subscribes to the wildcard topic `/cnp/bid/+`. Using this topic structure, the supervisor retrieves the machine identifier directly from the topic name on which each bid is received. This eliminates the need to include machine IDs inside the message payload and simplifies bid processing.

During a predefined deadline period, the supervisor gathers all incoming bids and stores the execution times proposed by each machine. Rejection messages are simply ignored in the selection phase. Once the deadline expires, the supervisor selects the machine offering the shortest execution time, ensuring an efficient assignment strategy.

The job assignment is sent only to the selected machine on its dedicated topic `/cnp/assign/{machine_id}`. Job dispatch messages are therefore not broadcast to all machines, but targeted to a single agent, preventing unnecessary processing by non selected machines. This choice reinforces the decentralized nature of the system while maintaining efficient coordination.

4.3 Execution Logs and Observations

To evaluate the Contract Net implementation, multiple machine agents are launched first, each with a distinct identifier and different job execution capabilities. The supervisor agent is started afterward, which triggers a sequence of job allocations. Upon execution, the logs show that each job initiates a Call for Proposal broadcast, followed by proposals or rejections from the machines depending on their capabilities and availability.

For each job, the supervisor receives multiple bids and selects the machine offering the shortest execution time. The selected machine logs the start of the job execution and remains unavailable during processing, while non selected machines remain idle and ready for subsequent negotiations. Once the execution time passes, the machine logs the completion of the job and becomes available again.

These logs demonstrate that jobs are dispatched efficiently, that machines do not execute concurrent jobs, and that the Contract Net protocol is correctly enforced through decentralized negotiation and targeted task assignment.

5 Webography

— EMQX Blog – *How to Use MQTT in Python*
`https://www.emqx.com/en/blog/how-to-use-mqtt-in-python`

— HiveMQ Blog – *MQTT Client Library : Paho Python*
`https://www.hivemq.com/blog/mqtt-client-library-paho-python/`

— Université de Pau – *MQTT avec Python*
https://munier.perso.univ-pau.fr/tutorial/iot/2022/20220510-mqtt$_{p}ython/$

— Medium – *A Beginner's Guide to MQTT*
`https://medium.com/@potekh.anastasia/a-beginners-guide-to-mqtt-understanding-mqtt`

— PyPI – *Paho MQTT Python Package*
`https://pypi.org/project/paho-mqtt/`