Multi-Agent Systems

ESISAR 5A IR&C - CS534 - Multi-agent systems

MQTT Lab

This lab is done under Linux. There are two lab sessions. You can work on your personal computer but you must be able to install new software.

I - MQTT Basics

> "*MQTT is a lightweight, publish–subscribe, machine-to-machine network protocol for message queue/message queuing service. It is designed for connections with remote locations that have devices with resource constraints or limited network bandwidth, such as in the Internet of things (IoT).*"
>
> — Wikipedia

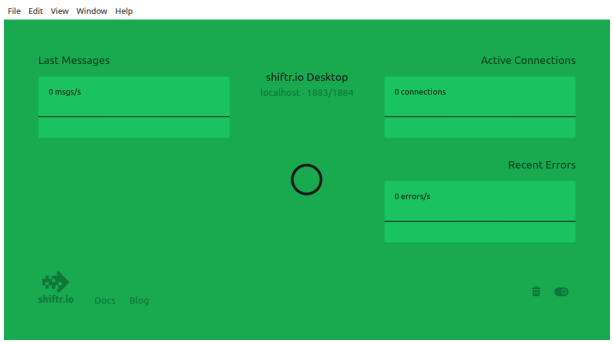
I.1 - First Contact

Setup - MQTT Broker

If you are working on the school's computer, you should be able to launch the "shiftr.io" application *et voilà.*

If you are working on your personal computer, you can install any MQTT broker you like.

Shiftr.io is a simple application that act as an MQTT broker and displays clients and messages in a simple and efficient way.



Git

Before starting to program your first client, create a Git repository[1] to store your code and give access to your repository to your teacher.

Put the code of each question in a dedicated directory in your repository.

First Client

To ensure your setup is working, you're going to program a first client and check that it's able to connect to the broker, subscribe and publish a message on a topic.

[1] Please, be aware that github is becoming less and less convenient and its owning company has started to migrate to its own infrastructure, making it more prone to unpleasant consequences in the future. Codeberg or other safer platforms are good alternatives

To program this first client, you can choose any langage that has an MQTT client lib. We can assist you with Python, C++, Java, and Rust clients. Any descent langage have an MQTT client lib with good starting examples.

> In the examples, the broker's address often points to some public broker on the internet. You should use the local broker you started implicitly when launching shiftr.io instead by connecting to **`localhost`**.

For **Python**, a good starting point is the Eclipse Paho MQTT package. It is recommended to use a virtual environment and to install `paho-mqtt` :

```
python -m venv .venv
source .venv/bin/activate
pip install paho-mqtt
```

Publish / Subscribe

This first client have to:

- successfully connect to the broker,
- subscribe to a topic of your choice, "`hello`" for example,
- print received messages on this topic on the console,
- publish several messages on this topic with delays between publishes.

Be sure to **commit and push** your work at this point.

I.2 - Two clients

To practice starting several clients, you'll begin with two clients playing the pong game.

You need first to program the two clients that will exchange messages, one responding `ping` to received `pong` messages, the other client doing the reverse.

You can use a command line argument to configure the same program to behave as a ping or pong agent.

Take the time to decide if it's more convenient to use one or two topics to implement this exercise. What is required if there is only one topic?

Find an automated way to start the two clients, using a bash script, a "master" process that spawn the clients, or any other solution, as long as it starts the clients in an automated fashion. In the following, there will be more clients and you don't want to start zillions of terminals.

II - Sensor Network

In this part, you will simulate a dynamic sensor network composed of several types of nodes, or agents:

- **Sensor agents**, emitting values at regular intervals.
- **Averaging agents**, collecting values from sensors and computing averages.
- **An Interface agent**, displaying the results produced by the network.

> Create a dedicated directory named "SensorNetwork" for this question.

II.1 - Agents or nodes

Sensors
Sensors belong to a "zone" corresponding to the area they are dispatched in. They publish their readings a regular intervals on a topic which includes the zone id, the type of measurement, and their individual id. The readings should change over time following a simple function, a sinusoid for example. You will start several sensors per zone with different ids. Your MQTT client code should therefore take parameters to be able to configure your system.

"Smart house" is an example of such sensor networks: several sensors are dispatched in a house, publishing on a topic composed of the room they are in, their measurement type, and their id.

A composite topic is a `/` separated list such as:

```
/living_room/temperature/STMicro_1a2b3c
/bathroom/humidity/TI_2f
```

You can choose other themes for your sensor network for example a warehouse, a plant, or a smart grid.

Averaging agents
These agents collect readings from different related sensors and compute an average over a predefined time window. At some predefined frequency, they publish their average on a dedicated topic.

Interface Agent
This agent displays the different averages, grouped by zone and measurement types.

You can make a simple agent that displays this information on the console or build a GUI to display the data in a more user-friendly way.

II.2 - Dynamics

To observe the ability of your setup to represent dynamic systems, ensure that some agents disappear and that other agents enter the system. You must automate this aspect of your simulation, either using a script or a master process spawning agents (or other solutions).

II.3 - Anomaly detection

Create a new directory in your Git repository for this question.

Detection agent
In the setup you have put in place in the previous section, add another agent which monitors the readings and averages and publish an alert on a dedicated topic when it detects an anomaly. The alert should contain the information about the sensors suspected to be in fault. An anomaly is constituted by a value significantly away from the system average. Implement a simple detection mechanism identifying readings that are two standard deviation off from the average. To test your detection, add a configuration in existing agents or a new type of agent that can send erroneous readings.

To identify the sensors that send erroneous values, you can use the fact that readings are sent on topics containing clients' ids and that you can retrieve on which topic a message has been sent when handling it.

Identification agent
Add another agent that is responsible for asking the suspected agents to restart. Implement the reset of the sensor agent when it receive a dedicated message.

III - Contract Net - Machines and Jobs

In this question, you have to implement a classical coordination mechanism in multi-agent systems : the Contract Net protocol.

You have to implement a multi-agent system in charge of **scheduling jobs** on **machines**. Each agent is responsible for a machine. Each machine have a list of jobs it can do with an associated time to complete. Not all machines can do all jobs.

There is a “Supervisor” agent which, in principle, receive jobs from the user and which is responsible to assign jobs to machines. For the lab, it will be responsible to generate a sequence of jobs.

Figure 1 shows the sequence diagram of a round of the Contract Net protocol in the context of the question.

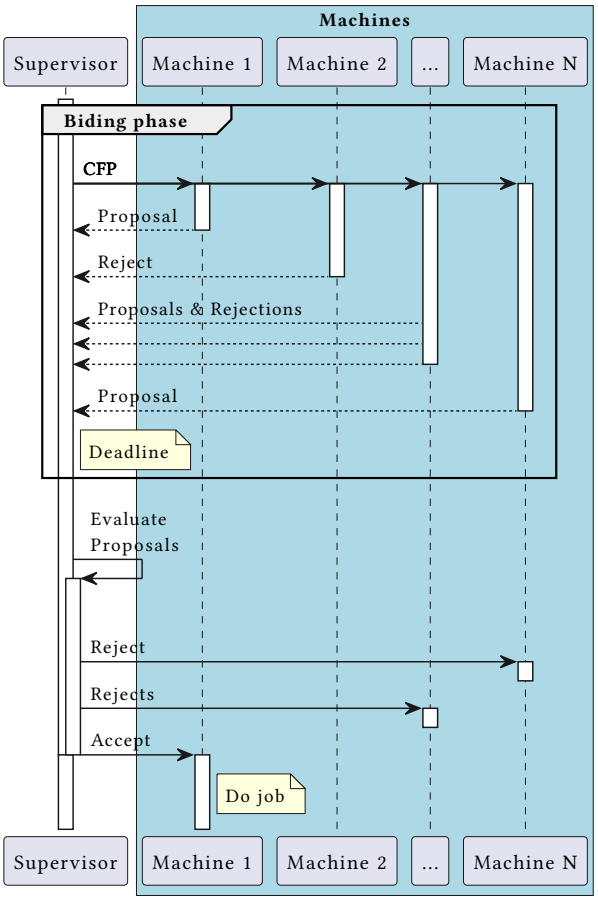


Figure 1: Contract Net protocol.

III.1 - Functional version

Implement this protocol in a multi-agent system using MQTT clients as agents. The supervisor will send a **Call for Proposal** (CfP) to machines with information about the requested job. Each agent will reply to these CfP with either a rejection message or a bid including the time associated with the requested job.

The supervisor will wait for some predefined time (deadline) for bids and eventually select a relevant machine to do the job.

When the selected machine receive the job, it can no longer answer to CfP until the job is done, that is to say after the elapsed time given as bid has passed, in seconds.

Your report should present relevant execution log showing that your implementation successfully dispatch jobs to machines in an efficient way.

You also should explain the choices you made during the implementation. For example:
- How did you manage to retrieve machines ids from biding messages?
- Did you send job dispatching messages to all machines?

III.2 - Optimisations - Optional Bonus

Once you have a functional job allocation mechanism and if you have time and interest, you can think about and propose optimisations.

For example, you can:
- Send CfP on dedicated topics, only soliciting machines able to do the job.
- Examine the next `n` jobs and try to optimize the allocation of jobs knowing only the proposals of current round.
- End the biding phase as soon as you received enough bids.

IV - Evaluation

IV.1 - Report

Your report should:
- Be synthetic: a maximum of 5 pages is expected. You can add annexes if you think it's required but your source code should complete the document by being well structured and documented.
- Present and **justify or explain** the choices you made (programming language, MQTT lib, threading or not, ids in topics or in messages, serialization lib, etc.)
- Show execution traces when relevant.
- Highlight relevant part of the code.
- Describe the main difficulties encountered and the solutions you implemented.
- Please, *pretty please*, no gpt bull***it.

You have to put your report inside your Git repository, in pdf format.

You can use the `typst` source code of this document as a starting point, it is available on this lab web page.

IV.2 - Source code

Your source code must be available on a Git platform such as GitLab and you have to **give your teacher adequate access** to it (maintainer for example)

You **have to** put the different exercises in different directories inside your repository.

Your code will be evaluated on:
- Its structure and decomposition, use of **classes, modules**, and other structuring tools available in the language of your choice.
- Good name choice for functions.
- A relevant user documentation in markdown format relative to each exercice. A **concise and precise** `Readme.md` in each directory is a frequent practice. This doc should describe briefly each type of clients and their parameters.
- Some comments or inline documentation when the function name is not explicit enough or when a piece of code is not trivial.