Multi-Agent Systems

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

MQTT Lab

Students: Livio Dadone, Gabriel Bragança de Oliveira
Supervisor: Clément Raïevsky

1 Introduction

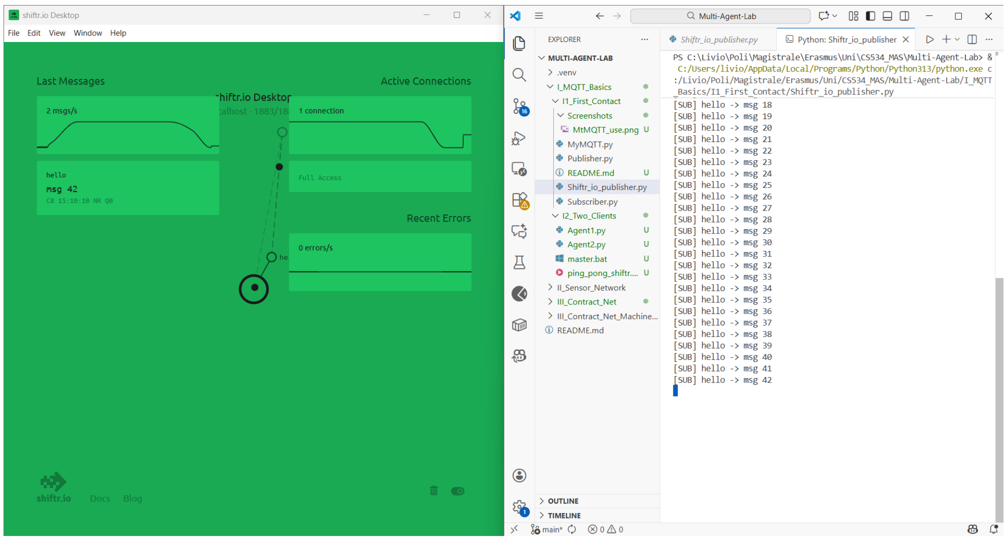
This report shows our work done fulfilling the requests of the CS534 MQTT lab in the following points: (I) a quick warm-up with MQTT basics, (II) a small sensor network that flags anomalies, and (III) a Contract Net scheduler where machines bid for jobs.

We used Python 3 with the `paho-mqtt`. In parts I–II we send numeric readings as plain text to make running and debugging easier with any MQTT viewer. In part III, messages carry richer content (job id, type, ETA, timestamps), so we switched to JSON for readability and portability. The repository is split by exercise (`I_MQTT_Basics/`, `II_Sensor_Network/`, and `III_Contract_Net/`) each with a short `README` explaining how to launch the agents and what to expect in the logs.

2 MQTT Basics

2.1 First Contact (Publish/Subscribe)

In `I_MQTT_Basics/I1_First_Contact`, we implement a minimal publish/subscribe workflow: a publisher periodically sends messages, while a subscriber prints received messages. We reuse a small abstraction layer `MyMQTT.py` on top of Paho to centralize connection, subscription, and callbacks.



```
PS C:\Livio\Poli\Magistrale\Erasmus\Uni\CS534_MAS\Multi-Agent-Lab> &
 C:/Users/livio/AppData/Local/Programs/Python/Python313/python.exe c
:/Livio/Poli/Magistrale/Erasmus/Uni/CS534_MAS/Multi-Agent-Lab/I_MQTT
_Basics/I1_First_Contact/Shiftr_io_publisher.py
[SUB] hello -> msg 18
[SUB] hello -> msg 19
[SUB] hello -> msg 20
[SUB] hello -> msg 21
[SUB] hello -> msg 22
[SUB] hello -> msg 23
[SUB] hello -> msg 24
[SUB] hello -> msg 25
[SUB] hello -> msg 26
[SUB] hello -> msg 27
[SUB] hello -> msg 28
[SUB] hello -> msg 29
[SUB] hello -> msg 30
[SUB] hello -> msg 31
[SUB] hello -> msg 32
[SUB] hello -> msg 33
[SUB] hello -> msg 34
[SUB] hello -> msg 35
[SUB] hello -> msg 36
[SUB] hello -> msg 37
[SUB] hello -> msg 38
[SUB] hello -> msg 39
[SUB] hello -> msg 40
[SUB] hello -> msg 41
[SUB] hello -> msg 42
```

Figure 1: Broker-side observation of publications using the Shiftr.io desktop interface.

2.2 Two Clients (Ping/Pong)

In `I_MQTT_Basics / I2_Two_Clients`, the objective is to achieve bidirectional communication between two autonomous MQTT clients using a simple request–response pattern. One client periodically publishes a `ping` message on a predefined topic, while the second client subscribes to this topic and immediately answers by publishing a `pong` message on a complementary topic.

Therefore, both agents act simultaneously as publishers and subscribers, illustrating the symmetry of the MQTT communication model. We have done this using two separate topics for publishing and subscribing.

The exchanged payloads are intentionally kept as plain strings in order to focus on the communication logic rather than on data serialization. Message timing is controlled with simple sleep intervals, to make the interaction easy to observe and debug. The correct behavior is verified through console logs, where each received `ping` triggers exactly one `pong` response.

Furthermore, python scripts are written in such a way that it is enough to change a variable in the *main* (which can easily be passed as input when running the script) to change the role of each agent between "*pong*" and "*ping*"

3 Sensor Network

3.1 Agents and Topic Design

We designed the sensor network as a realistic monitoring system for a *mountain refuge* (`refuge_Monviso`). The monitored data include both environmental measurements (temperature, humidity, wind speed) and technical quantities related to energy management (produced and consumed power). The topic hierarchy strictly follows the lab specification based on *zones*, *measurement types*, and *sensor identifiers*. Each sensor publishes its raw measurements on topics of the form:

```
refuge_Monviso/<room>/<measurement_type>/<sensor_id>
```

Two implementations of the sensor network are provided. In **V1**, each sensor, averaging agent, and interface agent is implemented as a separate script. This approach is straightforward (it clearly separates the roles of the agents), but it quickly becomes cumbersome: adding a new sensor or averaging agent requires duplicating or creating new scripts and manually launching multiple processes.

In **V2**, the architecture is refactored using **Python classes** to represent the different agent types (sensors, averaging agents, interface agent). A single script can therefore instantiate an arbitrary number of agents as objects, without writing additional code files. This design improves modularity and scalability: the number of sensors, their parameters (room, measurement type, publishing period), and the number of averaging agents can all be configured programmatically. It also makes the system easier to extend and closer to a real deployment, where agents may be created, destroyed, or restarted dynamically.

Averaging agents subscribe to a set of sensor topics corresponding to a given room and measurement type. They maintain a sliding window of the most recent sensor values and periodically compute an aggregate value (mean). This aggregated measurement is then published on a dedicated topic:

```
refuge_Monviso/<room>/<measurement_type>/<AA_id>
```

By separating raw data production from aggregation, averaging agents reduce the amount of data consumed by higher-level agents and provide more stable, noise-reduced values to the interface and to anomaly detection components.

3.2 Dynamics (Agents joining/leaving)

The dynamic version of the sensor network (`II_Sensor_Network/II2_Dynamics`) extends the static architecture by modeling agents that can dynamically join and leave the system. This behavior is more similiar to the one that takes place in the real-world IoT deployments, where sensors may disconnect due to power constraints, network issues, or maintenance, and later reconnect.

A dedicated master script (`main.py`) orchestrates the system: it instantiates sensor agents and averaging agents as Python objects and controls their lifecycle by alternating ON/OFF phases with randomized durations. During an OFF phase, an agent stops publishing (or unsubscribes), while during an ON phase it resumes normal operation. These cycles are intentionally asynchronous, ensuring that agents do not all disconnect or reconnect simultaneously.

The interface agent continuously listens to the aggregation topics and therefore observes a live system in which data streams may temporarily disappear and reappear. Importantly, no special recovery mechanism is required: thanks to MQTT's decoupled publish/subscribe model, the system remains stable even when publishers are temporarily absent. This experiment validates that the architecture is robust to dynamic membership and that the averaging and interface agents can tolerate partial and transient data availability without crashing or producing inconsistent behavior.

3.3 Anomaly Detection and Recovery

In `II_Sensor_Network/II3_Anomaly_Detection`, we add:

- a **Detection Agent** subscribing to both sensor readings and averages, publishing alerts on `refuge_Monviso/alert/anomaly`;
- an **Identification Agent** that receives alerts and asks suspected sensors to restart via a dedicated reset topic;
- a sensor "fault" mode (configurable) to generate erroneous readings and validate detection.

The detection logic is a direct implementation of the statement "two standard deviations away from the mean" over a sliding window (`window_size`, default 50 samples) for each measurement type. The alert payload includes the topic metadata (room, sensor ID), the current value, the window mean and standard deviation, and the selected $k = 2$.

Listing 1: Two-sigma anomaly test (detection_agent.py)

```python
mean = stat.mean(window)
std  = stat.stdev(window) if len(window) >= 2 else 0.0
threshold = self.k_sigma * std

if std > 0.0 and abs(value - mean) > threshold:
    alert = { "measurement_type": mtype, "room": room, "sensor_id": sid,
              "value": value, "mean": mean, "stdev": std, "k_sigma": self.k_sigma,
              "timestamp": time.time() }
    client.publish(self.topic_alert, json.dumps(alert), qos=0)
```

4 Contract Net Implementation

We implement the Contract Net Protocol (CNP) as three modules: *common* (data types and topics), *machine* (one agent per machine), and *supervisor* (job generator and allocator). Agents are MQTT clients communicating with JSON payloads for portability and easy inspection. Machines expose a capability map (job $\rightarrow$ ETA in seconds) and remain *busy* while executing, ignoring CfPs as required by III.1. We use lightweight threading on machines so the MQTT loop stays responsive during job execution.

4.1 Topic and Message Design

We use JSON messages for CNP because each message carries several fields (job ID, type, timestamps, ETA, machine ID). Using JSON also makes debugging straightforward with any MQTT inspector. The topics are:

- `lab/cnp/cfp/<job_type>` for CfP (multicast, filtered by job type);
- `lab/cnp/proposals` for all proposals (machines $\rightarrow$ supervisor);
- `lab/cnp/accept/<machine_id>` for Accept (unicast to the winner);
- `lab/cnp/done` for completion notifications.

Dedicated CfP topics are already an optimization: machines subscribe only to job types they can do, reducing unnecessary traffic.

Listing 2: Message dataclasses and topic helpers (common.py)

```python
BASE = "lab/cnp"

def t_cfp(job_type: str) -> str:      return f"{BASE}/cfp/{job_type}"
def t_proposals() -> str:             return f"{BASE}/proposals"
def t_accept(machine_id: str) -> str: return f"{BASE}/accept/{machine_id}"
def t_done() -> str:                  return f"{BASE}/done"

@dataclass
class Proposal:
    job_id: str; job_type: str; machine_id: str; eta_s: float; at: float
```

4.2 Bidding and Allocation Process

Machine IDs are carried in the proposal JSON (`machine_id`) and therefore available to the supervisor without relying on MQTT client identifiers. The supervisor stores received proposals per `job_id` and selects the minimum ETA at deadline.

Only CfPs are multicast. The Accept message is sent to *one* topic `lab/cnp/accept/<machine_id>` so that only the winner executes the job.

Listing 3: Supervisor selection and unicast accept (supervisor.py)

```python
ps = proposals[jid]
win = min(ps, key=lambda p: float(p["eta_s"]))
client.publish(t_accept(win["machine_id"]), Accept(jid, jt).to_msg(), qos=0)
```

4.3 Execution Traces (What to Observe)

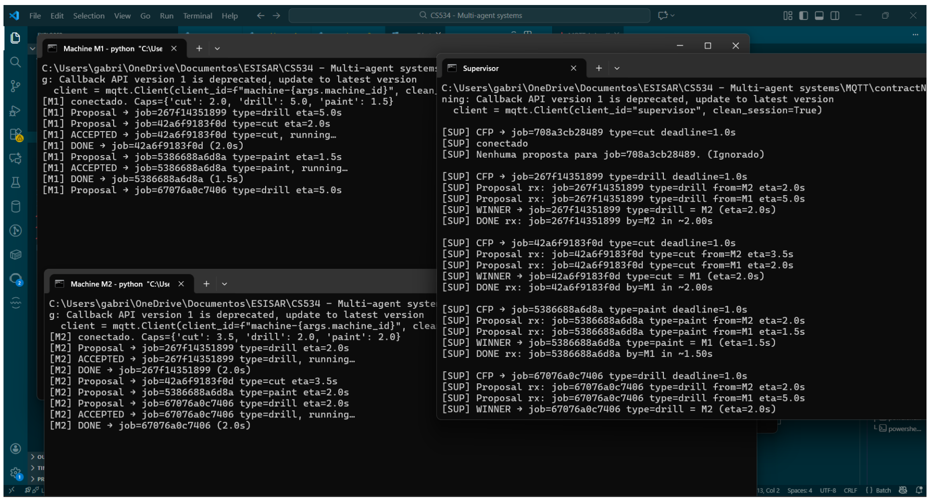
The programs print structured logs that allow validation of the protocol sequence: `CfP` → `Proposal(s)` → `WIN` → `DONE`. The following excerpt shows the *exact* log format produced by `supervisor.py` and `machine.py` during our tests (values depend on the machine capabilities and generated job IDs):

Listing 4: Typical log lines printed by the machine

```
[M12] Connected. Caps={'cut': 1.7, 'drill': 3.9, 'paint': 1.7}
[M12] Proposal -> job=9d55387c5ecd type=cut eta=1.7s
[M12] ACCEPTED -> job=9d55387c5ecd type=cut, running...
[M12] DONE -> job=9d55387c5ecd (1.7s)
```

Listing 5: Typical log lines printed by the supervisor

```
[SUP+] CFP: job=9d55387c5ecd type=cut deadline=1.00s
[SUP+] Connected
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M09 eta=2.7s
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M10 eta=3.2s
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M05 eta=1.9s
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M01 eta=1.8s
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M04 eta=2.2s
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M07 eta=3.6s
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M12 eta=1.7s
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M06 eta=2.8s
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M02 eta=2.4s
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M11 eta=2.5s
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M03 eta=3.0s
[SUP+] Proposal: job=9d55387c5ecd type=cut from=M08 eta=2.1s
[SUP+] WIN: job=9d55387c5ecd type=cut -> M12 (eta=1.7s)
```


File Edit Selection View Go Run Terminal Help
CS534 - Multi-agent systems
Machine M1 - python "C:\Use
Machine M2 - python "C:\Use
Supervisor
13, Col 2 Spaces: 4 UTF-8 CRLF { } Batch

```
C:\Users\gabri\OneDrive\Documentos\ESISAR\CS534 - Multi-agent systems
g: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client(client_id=f"machine-{args.machine_id}", clean_
[M1] conectado. Caps={'cut': 2.0, 'drill': 5.0, 'paint': 1.5}
[M1] Proposal → job=267f14351899 type=drill eta=5.0s
[M1] Proposal → job=42a6f9183f0d type=cut eta=2.0s
[M1] ACCEPTED → job=42a6f9183f0d type=cut, running…
[M1] DONE → job=42a6f9183f0d (2.0s)
[M1] Proposal → job=5386688a6d8a type=paint eta=1.5s
[M1] ACCEPTED → job=5386688a6d8a type=paint, running…
[M1] DONE → job=5386688a6d8a (1.5s)
[M1] Proposal → job=67076a0c7406 type=drill eta=5.0s
```

```
C:\Users\gabri\OneDrive\Documentos\ESISAR\CS534 - Multi-agent syste
g: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client(client_id=f"machine-{args.machine_id}", clea
[M2] conectado. Caps={'cut': 3.5, 'drill': 2.0, 'paint': 2.0}
[M2] Proposal → job=267f14351899 type=drill eta=2.0s
[M2] ACCEPTED → job=267f14351899 type=drill, running…
[M2] DONE → job=267f14351899 (2.0s)
[M2] Proposal → job=42a6f9183f0d type=cut eta=3.5s
[M2] Proposal → job=5386688a6d8a type=paint eta=2.0s
[M2] Proposal → job=67076a0c7406 type=drill eta=2.0s
[M2] ACCEPTED → job=67076a0c7406 type=drill, running…
[M2] DONE → job=67076a0c7406 (2.0s)
```

```
C:\Users\gabri\OneDrive\Documentos\ESISAR\CS534 - Multi-agent systems\MQTT\contractN
ning: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client(client_id="supervisor", clean_session=True)

[SUP] CFP → job=708a3cb28489 type=cut deadline=1.0s
[SUP] conectado
[SUP] Nenhuma proposta para job=708a3cb28489. (Ignorado)

[SUP] CFP → job=267f14351899 type=drill deadline=1.0s
[SUP] Proposal rx: job=267f14351899 type=drill from=M2 eta=2.0s
[SUP] Proposal rx: job=267f14351899 type=drill from=M1 eta=5.0s
[SUP] WINNER → job=267f14351899 type=drill = M2 (eta=2.0s)
[SUP] DONE rx: job=267f14351899 by=M2 in ~2.00s

[SUP] CFP → job=42a6f9183f0d type=cut deadline=1.0s
[SUP] Proposal rx: job=42a6f9183f0d type=cut from=M2 eta=3.5s
[SUP] Proposal rx: job=42a6f9183f0d type=cut from=M1 eta=2.0s
[SUP] WINNER → job=42a6f9183f0d type=cut = M1 (eta=2.0s)
[SUP] DONE rx: job=42a6f9183f0d by=M1 in ~2.00s

[SUP] CFP → job=5386688a6d8a type=paint deadline=1.0s
[SUP] Proposal rx: job=5386688a6d8a type=paint from=M2 eta=2.0s
[SUP] Proposal rx: job=5386688a6d8a type=paint from=M1 eta=1.5s
[SUP] WINNER → job=5386688a6d8a type=paint = M1 (eta=1.5s)
[SUP] DONE rx: job=5386688a6d8a by=M1 in ~1.50s

[SUP] CFP → job=67076a0c7406 type=drill deadline=1.0s
[SUP] Proposal rx: job=67076a0c7406 type=drill from=M2 eta=2.0s
[SUP] Proposal rx: job=67076a0c7406 type=drill from=M1 eta=5.0s
[SUP] WINNER → job=67076a0c7406 type=drill = M2 (eta=2.0s)
```

Figure 2: Contract Net round - using 2 machines

4.4 Optimizations

We provide an optional optimized supervisor (`supervisor_opt.py`) implementing:

- **Early stop** of bidding: stop waiting before the deadline when enough *distinct* machine proposals have arrived (`-min-bids`) or after a quiet period without new proposals in milliseconds (`-quiet-ms`).
- **Lookahead** ($n = 1$): if the next job has the same type, optionally keep the fastest machine available by selecting the second-best proposal when it is within a factor α (`--guard-fast --alpha 1.15`).

Listing 6: Early-stop conditions (supervisor_opt.py)

```
# stop when we already have enough distinct bidders
if args.min_bids and len({p["machine_id"] for p in proposals[jid]}) >= args.min_bids:
    break
# or when no new proposals arrived for quiet_ms milliseconds
if args.quiet_ms and (now_s() - last_rx[jid]) * 1000 >= args.quiet_ms:
    break
```

Listing 7: Lookahead n=1 with alpha guard (supervisor_opt.py)

```
ps = sorted(proposals[jid], key=lambda p: float(p["eta_s"]))
winner = ps[0]  # lowest ETA by default
same_next = (idx + 1 < len(job_types) and job_types[idx + 1] == jt)
if args.guard_fast and same_next and len(ps) >= 2:
    best_eta   = float(ps[0]["eta_s"])
    second_eta = float(ps[1]["eta_s"])
    if second_eta <= args.alpha * best_eta:
        winner = ps[1]  # keep the fastest for the next same-type job
```

4.5 Main Difficulties and Solutions

While executing jobs, keeping agents responsive was essential. We simulate execution with `sleep(ETA)` in a daemon thread and protect the shared `busy` flag with a lock; this prevents the MQTT loop from blocking and avoids race conditions. To avoid double acceptance, machines ignore CfPs whenever `busy=True`, and `on_accept` rechecks the state before starting any new job, ensuring a single job runs at a time.

Regarding traffic and scalability, we partition CfP topics by job type so each machine subscribes only to relevant requests, reducing unnecessary processing. Finally, to handle stragglers or temporarily offline agents, we enforce deadlines in the baseline supervisor and support early-stop in the optimized version, guaranteeing progress even when some machines are slow or disconnected.

5 Reproducibility (How to Run)

All exercises assume an MQTT broker available at `localhost:1883`.

Listing 8: Contract Net: supervisor + two machines (example)

```
# Terminal 1 - Starting the supervisor
# Baseline:
python supervisor.py --jobs "cut,drill,cut,paint,drill" --deadline 1.0 --wait-done
# or Optimized:
python supervisor_opt.py --jobs "cut,drill,cut,paint,drill" --deadline 1.0 --min-bids
    2 --quiet-ms 200 --guard-fast --alpha 1.15

# Terminal 2
python machine.py --machine-id M1 --caps "cut:2,drill:5,paint:1.5"

# Terminal 3
python machine.py --machine-id M2 --caps "cut:3.5,drill:2,paint:2"
```