École nationale supérieure en informatique, systèmes avancés et réseaux - Valence

Filière : IR&C

CS534 - Multi-agent systems MQTT Lab

Réalisé par :
CHAKIR Abderrahmane
EL AZZOUZI Soukaina

Enseignant :
Mr. Annabelle Mercier

Année Universitaire 2025-2026

0.1 Introduction

MQTT is a lightweight publish-subscribe messaging protocol designed for resource-constrained devices and low-bandwidth networks. In this lab, we explore how MQTT can serve as the backbone for multi-agent systems, where autonomous agents communicate and coordinate through message exchange.

The lab is divided into three parts :

1. **MQTT Basics** : Understanding publish/subscribe patterns through a simple client and a ping-pong game.
2. **Sensor Network** : Building a dynamic network of sensor agents, averaging agents, and an interface agent.
3. **Contract Net Protocol** : Implementing a coordination mechanism for job scheduling across multiple machines.

0.1.1 Technology Choices

We chose **Python** as the programming language for several reasons :

— The `paho-mqtt` library provides a mature and well-documented MQTT client.

— Python's threading support makes it easy to handle asynchronous message callbacks.

— Quick prototyping allows us to focus on the multi-agent concepts rather than low-level details.

All agents run as separate processes, spawned by a master script. This approach simulates a realistic distributed system where each agent operates independently.

0.2 Part I : MQTT Basics

0.2.1 First Client

The first exercise establishes a basic MQTT workflow. Our client connects to a local broker, subscribes to a topic called `hello`, and publishes messages with delays between them.

```
client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
client.on_connect = on_connect
client.on_message = on_message
client.connect('localhost', 1883, 60)
client.subscribe('hello')
```

Listing 1 – Key connection and subscription code

The client successfully receives its own published messages, confirming that the publish-subscribe loop works correctly. This simple test validates our development environment.

0.2.2 Ping-Pong Game

For the ping-pong exercise, we implemented a single configurable client that can play as either "ping" or "pong" based on a command-line argument.

Topic Design Decision

We chose to use **two separate topics** : `game/ping` and `game/pong`. This design is cleaner than a single topic because :

- Each player only subscribes to the opponent's channel.
- No need to filter out self-sent messages.
- Clear separation of concerns.

With a single topic, the client would need to ignore messages it sent itself, adding unnecessary complexity.

Automated Startup

A master script (`start_game.py`) spawns both players as subprocesses and displays their interleaved output. This approach scales well for simulations with many agents.

```
[GAME] Starting Ping-Pong Game...
[PONG] Connected to broker!
[PING] Sent: PING
[PONG] Received: PING
[PONG] Sent: PONG
[PING] Received: PONG
[PING] Sent: PING
...
[PONG] Game finished after 10 rounds!
```

Listing 2 – Execution trace of the ping-pong game

0.3 Part II : Sensor Network

This section simulates a smart home sensor network with dynamic behavior.

0.3.1 Architecture

The system consists of three types of agents :

Agent Type	Role	Topic Pattern
Sensor	Publish readings	`<zone>/<type>/<id>`
Averaging	Compute statistics	Subscribe to `<zone>/<type>/+`
Interface	Display dashboard	Subscribe to `averages/#`

Table 1 – Agent types and their topic patterns

0.3.2 Topic Structure

We designed a hierarchical topic structure that mirrors the physical organization :

```
living_room/temperature/sensor_001
living_room/humidity/sensor_002
averages/living_room/temperature
```

This structure allows averaging agents to subscribe to all sensors of a specific type in a zone using wildcards (e.g., `living_room/temperature/+`).

0.3.3 Sensor Implementation

Each sensor generates readings following a sinusoidal pattern to simulate realistic variations :

```python
def generate_reading(self):
    elapsed = time.time() - self.start_time
    value = self.base_value + self.amplitude * math.sin(elapsed * 0.1)
    value += random.uniform(-0.5, 0.5)  # Add noise
    return round(value, 2)
```

Listing 3 – Sinusoidal reading generation

0.3.4 Dynamic Behavior

The master process demonstrates dynamic system behavior by :

- Spawning new sensors at random intervals.
- Removing sensors randomly (keeping a minimum count).
- Adding averaging agents when new zone/type combinations appear.

0.3.5 Anomaly Detection

We extended the sensor network with anomaly detection capabilities :

1. **Detection Agent** : Monitors all sensor readings and computes rolling statistics. When a reading exceeds 2 standard deviations from the mean, an alert is published.
2. **Identification Agent** : Counts alerts per sensor. After 3 alerts, it sends a reset command to the faulty sensor.
3. **Sensor Reset** : Sensors subscribe to `control/reset/<id>` and reset their state when commanded.

The faulty sensor simulation uses a 30% chance to send an anomalous reading :

```
if self.faulty and random.random() < 0.3:
    value += random.choice([-1, 1]) * self.amplitude * 4
```

Listing 4 – Faulty reading injection

0.4 Part III : Contract Net Protocol

The Contract Net protocol is a coordination mechanism where a supervisor allocates tasks to worker agents through a bidding process.

0.4.1 Protocol Implementation

Our implementation follows the standard Contract Net flow :

1. **Call for Proposal (CfP)** : The supervisor broadcasts a job request on `cfp/jobs`.
2. **Bidding** : Machines respond on `bids/<job_id>` with either a proposal (including completion time) or a rejection.
3. **Deadline** : The supervisor waits for a configurable deadline (3 seconds in our tests).
4. **Evaluation** : The supervisor selects the machine with the lowest completion time.
5. **Award** : The winner receives the job on `awards/<machine_id>`.
6. **Execution** : The machine becomes busy and cannot bid until the job completes.

0.4.2 Design Decisions

Machine ID Retrieval

Machine IDs are included in the bid message payload rather than extracted from topics. This makes the supervisor's logic simpler and more robust :

```
bid = {
    'type': 'proposal',
    'machine_id': self.machine_id,
    'time': bid_time,
    ...
}
```

Targeted Awards

Awards are sent to machine-specific topics (`awards/<machine_id>`) rather than broadcast. This ensures only the winner receives the assignment and reduces unnecessary message processing.

0.4.3 Execution Results

A typical simulation with 4 machines and 10 jobs shows the protocol working correctly :

```
[CFP] CfP sent for job 15412d4e (painting)
[WAIT] Waiting 3s for bids...
[BID] Received bid from machine_D: 2.8s
[REJECT] Rejection from machine_A
[BID] Received bid from machine_C: 4.03s
[WINNER] Selected machine_D (time: 2.8s)
[AWARD] Job 15412d4e awarded to machine_D
...
FINAL STATISTICS
Jobs Completed: 9
Jobs Failed:    1
```

Listing 5 – Contract Net execution trace (excerpt)

The one failed job (packaging) occurred because no machine had the required capability, demonstrating that the protocol correctly handles edge cases.

0.5 Difficulties and Solutions

0.5.1 Process Management

Problem : Spawned subprocesses didn't always terminate cleanly on Ctrl+C.

Solution : We implemented proper signal handling with `signal.SIGINT` and added cleanup code that terminates all child processes before exiting.

0.5.2 Message Timing

Problem : In the ping-pong game, if ping started before pong was ready, messages were lost.

Solution : Pong is started first, with a small delay before ping begins. Additionally, ping waits 0.5 seconds after connecting before sending its first message.

0.6 Conclusion

This lab demonstrated how MQTT can effectively enable communication in multi-agent systems. The publish-subscribe pattern provides a flexible and decoupled architecture where agents can join or leave the system dynamically. **Key takeaways :**

— **Topic design** is crucial for scalability and clarity.
— **JSON payloads** make messages self-describing and easy to debug.
— **Process isolation** (each agent as a separate process) provides realistic distribution simulation.
— The **Contract Net protocol** effectively coordinates task allocation in a distributed system.

The complete source code is available in the Git repository, organized by exercise with README files documenting each component.

Repository Structure

```
mqtt/
 I_FirstClient/      # Basic publish/subscribe
 I_PingPong/         # Ping-pong game
 SensorNetwork/      # Dynamic sensor network
 AnomalyDetection/   # Anomaly detection extension
 ContractNet/        # Contract Net protocol
```