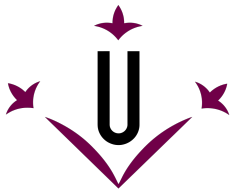

Universitat de Lleida

Cloud Service

Cloud Service - Grau en Enginyeria Informàtica

Artur Cullerés Cervera
Roger Fontova Torres

December 19, 2022

Contents

List of Figures

1 Introduction

This project attempts to create a whole network of IoT. This network is composed of:

- **MQTT broker:** It enables the communication between the temperature sensors and the cloud service.
- **Kafka broker:** It enables the communication between the cloud service and the AI agent.
- **Temperature sensors:** Simulated sensors that read temperatures from a CSV file and send them to the MQTT broker.
- **Cloud Service:** Reads the data sent by the sensors to the MQTT broker, stores the data, sends it to a Kafka broker, receives the AI predictions sent to the Kafka broker, and stores all the data.
- **AI agent:** Makes predictions with the given temperatures.
- **Non relational database:** Stores all the processed data by the cloud service.

In the following sections it will be explained the main decision that we have taken in order to implement the whole network.

However, we will not explain the MQTT publishers (the temperature sensors), and the ai-agent since it has been provided by the professor.

2 Cloud service

As it can be seen on the figure 1 the cloud service is divided by Subscriber, Producer, Consumer, DatabaseController and Queues.

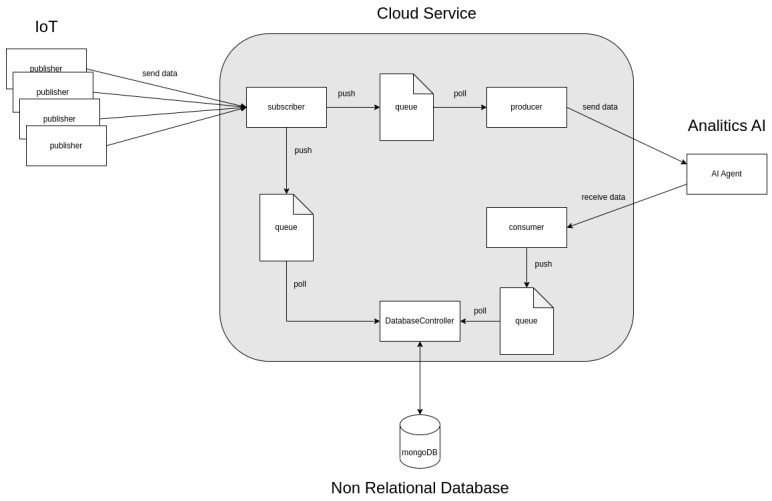

Figure 1: Cloud Service Architecture

2.1 Subscriber

This class defines an Subscriber thread that listens for incoming messages on a specified MQTT topic from a specified broker. When a message is received, it is deserialized and added to two queues, one for Kafka and one for a database. The Subscriber implements the MqttCallback interface to handle the various MQTT events.

It's worth noting that the run() method of the Subscriber class is calling itself if a connection is lost or an error occurs in order to try to reconnect to the specified broker.

2.2 Producer

This class defines a Kafka Producer thread that sends messages to a Kafka broker. It uses a PasiveWaitQueue to retrieve messages from a queue and sends them to the Kafka broker using a KafkaProducer.

The KafkaProducer is created using a set of properties that specify the configuration for the producer, such as the address of the Kafka broker and the serializers to use for the keys (StringSerializer) and values (Temperature-

Serializer, which is a custom serializer) of the messages.

The run() method of the DataProducer thread continuously polls the PasiveWaitQueue for new messages and sends them to the Kafka broker. The message is a Temperature object, which is wrapped in a ProducerRecord and then sent to the Kafka broker using the send method of the KafkaProducer.

2.3 Consumer

This class defines a Kafka Consumer thread that consumes messages from a Kafka topic. It user a PasiveWaitQueue to send the messages received from the Kafka Broker to the KafkaDatabase controller.

The run() method is overridden to provide the behavior of the thread. It has a loop that polls the Kafka consumer for messages with a timeout of 100 milliseconds. For each message received, it pushes the value of the message (a TemperaturePrediction object) onto the PasiveWaitQueue object and prints some information about the message to the console.

The Kafka consumer is subscribed to the ”analytics_results” topic, so it will receive all messages published to that topic. The consumer is configured to start reading from the earliest record in the stream, and to use a custom deserializer for the values of the messages (TemperaturePredictionDeserializer). The group.id property is set to ”predictions-results”, which identifies the consumer group that this consumer belongs to.

2.4 Database Controller

This class defines a DatabaseController thread that starts two threads, one for the Kafka temperature predictions database controller, and one for the MQTT temperature data database controller. Both use the same InfluxDB client in order to store the data to the same database.

The token used to connect to the database is fix since it is specified like this on the docker-compose.yaml of the database.

2.4.1 Kafka Database Controller

This class defines the Kafka database controlled thread mentioned before. It continuously polls the PasiveWaitQueue given by the Database Controller for a TemperaturePrediction object, and then saves it to the database using the InfluxDBClient and its writeMeasurement method.

2.4.2 MQTT Database Controller

This class does the same as the Kafka Database Controller, but storing temperatures from the MQTT broker

2.5 PasiveWaitQueue

This is a thread-safe queue implementation that uses a ConcurrentLinkedQueue as the underlying data structure to store the elements.

There are two public methods (push and poll) and two private methods (isEmpty and waitData). The push method adds an element to the queue and then sends a notification to all waiting threads to wake up and check the queue for new data. The poll method waits for data to be available in the queue if it is empty, and then returns and removes the first element in the queue. The isEmpty method checks if the queue is empty. The waitData makes the execution wait to be notified.

One thing to note is that the waitData method calls wait on the queue object itself, which means that any thread that calls wait or notifyAll on the queue must hold the lock on the queue object. This is because the wait and notifyAll methods can only be called from within a synchronized block.

This implementation enables the usage of while(true) to poll data from the queue without doing an active wait.

3 Database

For the database we decided to use InfluxDB since it is a time series database. These means that it is optimized to store data that the time when it is collected is important.

To access the database a token is needed, so it is specified on the docker-compose.yaml to always use the same one.

4 Curiosities

4.1 Java Instant

In Java, Instant objects cannot be serialized or deserialized, so a custom Serializer/Deserializer is needed. So our custom serializer converts an the Instant object to milliseconds and the deserializer converts milliseconds to Instant object. In orther to specify to the entity that has an Instant attribute how to serialize/deserialize it, we used the following tags:

```
@Column(timestamp = true)
@JsonProperty("timestamp")
@JsonDeserialize(using = MyInstantDeserializer.class)
@JsonSerialize(using = MyInstantSerializer.class)
public Instant time;
```

- @Column(timestamp = true): used for the database to use it as the timestamp.
- @JsonProperty("timestamp"): name to match on the json object when serializing or deserializing.
- @JsonDeserialize(using = MyInstantDeserializer.class): specifies the deserializer class
- @JsonSerialize(using = MyInstantSerializer.class): specifies the serializer class

4.2 MQTT fault tolerance

When the MQTT subscriber looses its connection, the thread dies, so in order to avoid that, we catched the exception and run again the run() method, with a waiting time. This may cause an infinite loop, however, it is necessari to keep trying to reconect to the broker.

```
try {
    client.connect();
    client.subscribe(topic, qos);
    System.out.println("Subscriber UP!");
} catch (MqttException e) {
    System.out.println(e.getMessage());
    try {
        Thread.sleep(5000);
        run();
    } catch (InterruptedException ex) {
        throw new RuntimeException(ex);
    }
}
```

4.3 Docker compose fault tolerance

If the project was executed with docker–compose.yaml only using the depends_on option, it wouldn't execute, since it only depends on the creation of the container, not the execution of its code.

To avoid any kind of problem, it is necessary to use the option healthcheck. This allows to check if the server running on another container is already up (healthy). If that's the case, the depending container can run. The following code shows an example of how to set the healthcheck option:

```
healthcheck:
    test: curl --fail http://localhost:8086/health || exit 1
    interval: 10s
    retries: 5
    start_period: 5s
```

```
timeout: 10s
```

5 Conclusions

With this project we have learnt a lot about MQTT and Kafka. Even though it is all simulated with docker containers, it would only need a little more of efford to deploy it with real sensors and with a server for the cloud service and the MQTT and Kafka brokers. Not only we have learnt about MQTT and Kafka, we also have learnt a lot about docker and serialization of objects.

It also has been a lot of fun designing and implementing the cloud service with this particular architecture. Of course it has a lot of margin of improvement, for example, it could be implemented commands to start or stop different parts of the cloud service, for example, if the kafka database thread dies, be able to rerun it manually with a command.

Finally, we will leave the link to the github repository of the project so anyone can access the source code.

The README.md has an explanation of how to run the project.