Internet of Things

Smart CO monitoring

Adrien M.

July 2023

Contents

1 General introduction

The aim of the project was to create an IoT telemetry and control system. I decided to work on a solution to monitor the carbon monoxide level in factories. The aim of the document is to explain the implementation of the code in details, for more information about the resulting simulation, please refer on the README.md file.

1.1 What is CO ?

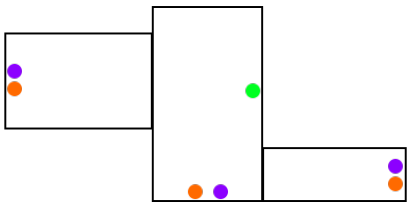
The carbon monoxide (CO) is a potentially **deadly toxic gas** that is highly dangerous due to its **colorless and odorless nature**. It is responsible for a significant number of preventable deaths each year.
All types of combustion appliances can be a potential source of carbon monoxide: boilers, water heaters, stoves, cookers, automotive engines, generators...
The production of CO results from an incomplete combustion of various fuels

due to insufficient ventilation , lack of appliance maintenance...
At concentrations **above 200 ppm** a human can have **headaches, dizziness, nausea**. At concentrations of **400 ppm and above**, headaches become intense, and there is a risk of death after only 3 hours of exposure. For a concentration of **1280 ppm death happens in 2 minutes**.
Measured using an **electrochemical sensor**: CO molecules attach to the sensor's surface, generating a measurable low electric current.

1.2 Use case: A factory

In many industrial sectors, workers are exposed to carbon monoxide when carrying out tasks such as pressure cleaning, warehousing, surface sanding and maintenance of combustion engines...

The factory is divided into 3 units: Unit A, Unit B, Unit C. Each unit is equipped with a pair of sensors - actuators. A border router is also deployed in the center.



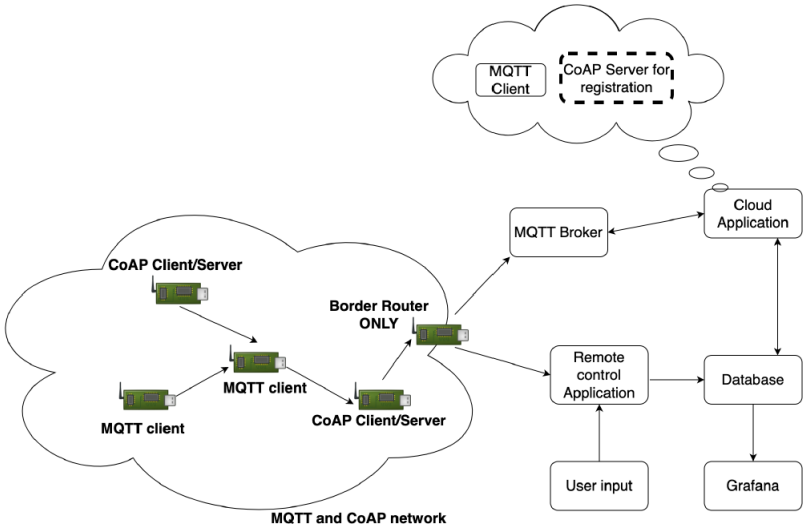
View of the factory from above

2 Requirements

To implement the project there was a given guideline to follow.

2.1 System scheme and structure

The deployment of the smart solution should have this architecture:



The system must comprise the following components:

- A network of IoT devices, including sensors collecting data from the environment and actuators. The sensors must use MQTT and the actuators must use CoAP. The devices that use CoAP are CoAP Server that expose their resources. In the network, a border router must be deployed in order to provide external access.
- The list of actuators is statically created and exploited by the User Application via a configuration file.
- The Cloud Application collects data from MQTT sensors, store them in a MySQL database.
- The Remote control Application reads information about actuators and sensors from the database and implements a simple control logic in order to apply some modifications to one or more actuators based on the data collected from the sensors.
- It is required to provide some User Input to implement the User logic for the IoT application. The User input can be implemented as a command line interface.
- A web-based interface deployed using Grafana must be developed in order to show the data collected and stored on the database.

3 MQTT sensors

The sensors are **measuring (fictitiously) the CO level** in the air of the Unit they are deployed in and then they **publish the value on the topic**

"CO_LVL" :

```
                if(state == STATE_SUBSCRIBED && (period%60==0)){
                        // Publish something
                        sprintf(pub_topic, "%s", "CO_LVL");
                        co_level =  co_level + random_rand()%(35-1+1);
                        printf("CO level is: %d ppm\n", co_level);
                        sprintf(app_buffer, "{\"sensor\": %d, \"CO level\": %d}",
node_id, co_level);

                        mqtt_publish(&conn, NULL, pub_topic, (uint8_t
*)app_buffer, strlen(app_buffer), MQTT_QOS_LEVEL_0, MQTT_RETAIN_OFF);
```

Then the sensors **subscribe to the topic "consistency"** because as the sensors are not real, it is mandatory to ensure data consistency when actions will be performed by the actuators to reduce the CO in unit:

```
                        // Subscribe to a topic
                        char topic[20];
                        sprintf(topic, "client%d/consistency", node_id);
                        strcpy(sub_topic, topic);

                        status = mqtt_subscribe(&conn, NULL, sub_topic,
MQTT_QOS_LEVEL_0);
```

Then the sensors have to adapt their behaviour depending on what they received on the topic "consistency" to simulate air conditioning:

```
static void
pub_handler(const char *topic, uint16_t topic_len, const uint8_t *chunk,
            uint16_t chunk_len)
{
  printf("Pub Handler: topic='%s' (len=%u), chunk_len=%u\n", topic,
         topic_len, chunk_len);
  char expected_topic[20];
  sprintf(expected_topic, "client%d/consistency", node_id);
  if(strcmp((const char *)topic, expected_topic) == 0){
    printf("Received Actuator command\n");

    if (strcmp((const char *)chunk, "moderate")==0){
        co_level = co_level - random_rand()%(35-20+1);
    }else if (strcmp((const char *)chunk, "high")==0){
        co_level = co_level - random_rand()%(95-55+1);
    }else{
        printf("No data.\n");
    }
    return;
  }
}
```

Simulate air conditioning

4 CoAP Actuators

For the CoAP actuators, **two resources** were defined:

- LED

```
RESOURCE(led,
         "title=\"LED Color: ?color=r|g|y, POST mode=low|moderate|high\"
```

Depending on the "mode" posted by the CoAP application, the **leds of the actuators will change** from green to yellow to red to easily alert the workers on the CO concentration in the unit:

```
if(strncmp(mode, "low", len) == 0) {
  leds_set(LEDS_NUM_TO_MASK(LEDS_GREEN));
} else if(strncmp(mode, "moderate", len) == 0) {
  leds_off(LEDS_ALL);
  leds_on(LEDS_RED);
  printf("Activation of air conditioning.\n");
} else if(strncmp(mode, "high", len) == 0) {
  leds_set(LEDS_NUM_TO_MASK(LEDS_RED));
  printf("Setting air conditioning at maximum speed.\n");
}else {
  success = 0;
```

- Factory

```
RESOURCE(factory,
         "title=\"Factory: ?unit=0..\" POST/PUT name=<name>&co_value=<co_value>\"
```

Then the 3 different units are defined:

```
static char units_avl[6][15] = {
        "unit_A ",
        "unit_B ",
        "unit_C ",
    };
static int actual_units = 3;
static int max_units = 7;
```

Then the values of the CO level are put in the corresponding unit:

```
len = coap_get_post_variable(request, "name", &text);
if(len > 0 && len < 15) {
    memcpy(unit, text, len);
    success_1 = 1;
}

len = coap_get_post_variable(request, "co_value", &text);
if(len > 0 && len < 32 && success_1 == 1) {
        memcpy(co_level, text, len);
        char msg[500];
    memset(msg, 0, 500);
        sprintf(msg, "CO level in %s set to %s", unit, co_level);
        int length=sizeof(msg);
        coap_set_header_content_format(response, TEXT_PLAIN);
        coap_set_header_etag(response, (uint8_t *)&length, 1);
        coap_set_payload(response, msg, length);
        success_2=1;
        coap_set_status_code(response, CHANGED_2_04);
```

5 Applications

The applications need to be implemented to **define the behaviour of the sensors and actuators and perform small actions**.

5.1 MQTT Application

The MQTT application's aim is to retrieve the data and store it in the database and also ensure the consistency. First, the MQTT collector, implemented with Python, must subscribe to the topic "CO_LVL" to receive the data from the MQTT sensors:

```
# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, flags, rc):
    print("connected with result code " + str(rc))
    client.subscribe("CO_LVL")

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))
    data = json.loads(msg.payload)
    sensor = data["sensor"]
    co_level = data["CO level"]
    now = datetime.now()
    timestamp = now.strftime("%d/%m/%Y %H:%M:%S")
```

Then, the application will **write the value received by the sensor in the database**, with the following structure: node ID — *co_level* — timestamp:

```
    mycursor.execute("INSERT INTO CO_MONITORING (`sensor`, `co_level`, `timestamp`) VALUES (%s, %s, %s)", (sensor, co_level,
timestamp))
    mydb.commit()
    print(mycursor.rowcount, "record inserted.")
```

Finally, the application will **publish on the topic "consistency"** to modify the behaviour of the sensor depending on the values of the CO level received:

```
if sensor == 2:

  if co_level > 100 and co_level <= 300:
      client.publish("client2/consistency", payload="moderate")
  elif co_level > 300:
      client.publish("client2/consistency", payload="high")
```

5.2 CoAP Application

The CoAP application's aim is to **retrieve the data from the database** and to **modify the behaviour of the CoAP actuators depending on the values**. The thresholds are fixed to 100 ppm and 300 ppm. Under 100 ppm, the LED in the unit is green and **no actions are required** from the actuator, between 100 ppm and 300 ppm, the LED is yellow and the actuator in the unit is asked to **trigger the air conditioning** with the post request in mode "moderate" and if the value is above 300 ppm, the LED is red and the actuator is asked to **set the air conditioning at maximum speed** with the post request in mode "high":

```
# Actuator 2:
#update sensor2
mycursor.execute("SELECT co_level FROM CO_MONITORING WHERE sensor = 2")
result2 = mycursor.fetchall()
sensor2 = result2[-1][-1]
print("CO level in Unit A is: %d ppm" % sensor2)

if sensor2 < 100:
    client2.post(res_led, payload="mode=low")
elif sensor2 > 100 and sensor2 <= 300:
    client2.post(res_led, payload="mode=moderate")
elif sensor2 > 300:
    client2.post(res_led, payload="mode=high")

client2.put(factory, payload="name=unit_A&value={}".format(sensor2))
```

Retrieving data from the database

Posting the mode corresponding to the CO level for the LED resource

Setting the CO level in the corresponding unit