POLITECNICO
MILANO 1863

Node-RED Project

3rd IoT challenge

Author
Christian Confalonieri

Professors
Matteo Cesana
Fabio Palmese

2023/2024

Contents

1 Introduction

This document describes the project created for the 3rd IoT challenge. It was developed using Node-RED, a visual programming tool based on flow-based development principles.
The project consists of interconnected nodes working together to accomplish different objectives. The flow is designed to generate messages, process them, save them in a CSV file, and plot them in a chart. Additionally, it exchanges messages using the MQTT protocol and communicates with the Thingspeak server via GET requests, accessible at the provided link:

`https://thingspeak.com/channels/2504435`

2 Considerations and assumptions

- **Graphic design**: special consideration was given to organizing the flow graphically, making it easier to read and understand. In particular, efforts were made to clearly distinguish the different tasks required by the challenge.
- **Fully automated flow**: the flow was designed to operate autonomously without manual intervention. Upon deployment, it initiates the operation and only halts when explicitly stopped (e.g., breaking an arch and deploying). This also concerns the "Reset on restart" assumption.
- **Reset on restart**: it was assumed that the variables and charts reset at each restart of the flow. Context variables automatically reset with flow restarts. However, global variables require manual resetting since they persist across flow restarts. This is achieved through the "Reset ACK Counter" section (refer to 3.5.1). The chart is reset using an empty JSON array injected into the "Chart" section (refer to 3.4.1).
- **Stop after 80 msgs**: initially, it was interpreted from the provided PDF that a maximum of 80 messages should be generated in the first part as well. However, after clarifications on the forum, the design was adjusted to indefinitely generate messages in the first part and process a maximum of 80 messages for subsequent parts.

3 Design and implementation using Node-RED

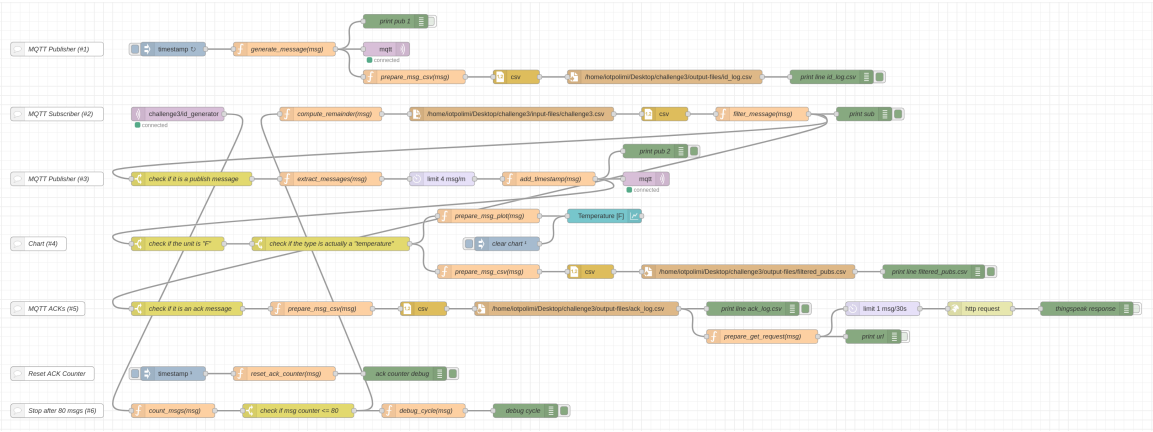

Figure 1: Flow diagram of the project

As mentioned earlier, we can clearly see the different tasks required by the challenge, each marked with a number between parantheses (e.g., "MQTT Publisher (#1)").

In the following paragraphs, we will describe the different parts of the flow, corresponding to the different tasks required by the challenge.

3.1 What to do? (1): MQTT Publisher

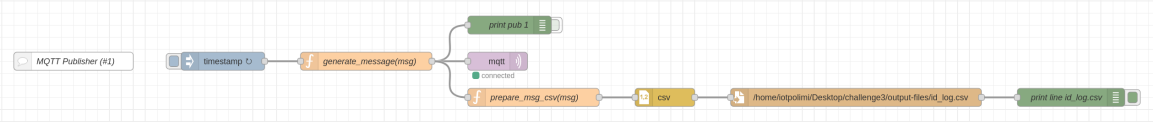

Figure 2: 1st MQTT Publisher part

3.1.1 Flow and nodes description

- **timestamp**: this node injects a message into the flow, specifically including the timestamp as the message payload. It triggers the flow every 5 seconds, maintaining a message sending rate of 1 message every 5 seconds. To ensure the flow triggers after the reset of the ack counter (explained in the following parts), the inject node is set to activate the flow every 5 seconds starting from 5 seconds after the flow begins (the "inject once after" option is unchecked).
- **generate_message(msg)**: this function node constructs the message sent to the broker, setting the topic ("challenge3/id_generator") and the quality of service (QoS=2). The message is created by combining a randomly generated ID (between 0 and 50000, both included) with a timestamp obtained from the inject node as msg.payload. The function code is the following:

```
msg.topic = "challenge3/id_generator";
msg.qos = 2;
msg.payload = {
   "id": Math.floor(Math.random()*(50000+1)),
   "timestamp": msg.payload
};

return msg;
```

- **print pub 1**: this debug node displays the message generated by the previous node, verifying its accuracy.
- **mqtt**: this MQTT Publisher node transmits the message to the broker. The topic and QoS are dynamically determined by the preceding node. The server (broker) is configured as 127.0.0.1:1884.
- **prepare_msg_csv(msg)**: this function node is used to prepare the message to be saved in the CSV file. The message is generated as requested by the challenge, with the No., the ID, and the TIMESTAMP.
  Note: the No. field increments with each execution of this function, starting from 1 (it is stored in a context variable "msg_no").

The function code is the following:

```
var msg_no = context.get('msg_no') || 0;
msg_no++;

context.set('msg_no', msg_no);

msg.payload = {
   "No.": msg_no,
   "ID": msg.payload.id,
   "TIMESTAMP": msg.payload.timestamp
};

return msg;
```

- **csv**: this CSV node converts the msg.payload (JSON object) into CSV format. It's configured to transmit the message header once, along with the first message.
- **/home/iotpolimi/Desktop/challenge3/output-files/id_log.csv**: this file node saves the msg in the CSV file named id_log.csv, located in the /home/iotpolimi/Desktop/challenge3/output-files/ directory.
- **print line id_log.csv**: this debug node prints the message stored in the CSV file, verifying its correctness.

3.2 What to do? (2): MQTT Subscriber

Figure 3: MQTT Subscriber part

To simplify the explanation, we can assume that the "challenge3/id_generator" node is directly connected to the "compute_remainder(msg)" node, as their behavior is the same as if they were linked. In reality, between these two nodes, there's the last part (the part limiting the processed messages to 80). A more detailed explanation of that part is provided in the respective section.

3.2.1 Flow and nodes description

- **challenge3/id_generator**: this MQTT Subscriber node subscribes to the topic "challenge3/id_generator" with QoS=2. The server (broker) is configured as 127.0.0.1:1884.
- **compute_remainder(msg)**: this function node computes the remainder of the ID when divided by 7711. The result is stored in a flow variable named N. Additionally, it stores the sub msg ID in a flow variable called sub_id, which is useful for the next parts of the challenge. The function code is the following:

```
flow.set("sub_id",msg.payload.id);
const N = msg.payload.id % 7711;
flow.set("N",N);

return msg;
```

- **/home/iotpolimi/Desktop/challenge3/input-files/challenge3.csv**: this file node reads the provided CSV file. The file is located in the /home/iotpolimi/Desktop/challenge3/input-files/ directory with the name challenge3.csv.

- **csv**: this CSV node converts the CSV format into an object (JSON), with the first row serving as the header.
- **filter_message(msg)**: this function node filters the message received from the MQTT Subscriber node. It verifies if the remainder of the ID (N) matches the "No." field from the message extracted from the CSV file. If the condition is met, the function returns the message. The function code is the following:

```
if(msg.payload && msg.payload["No."] == flow.get("N")) {
   return msg;
}
```

  Note that in the if statement, we also verify if msg.payload is not null. This is likely necessary because the last empty line in the CSV file could cause an error.

- **print sub**: this debug node prints the message received from the MQTT Subscriber node that meets the condition.

3.3 What to do? (3): MQTT Publisher

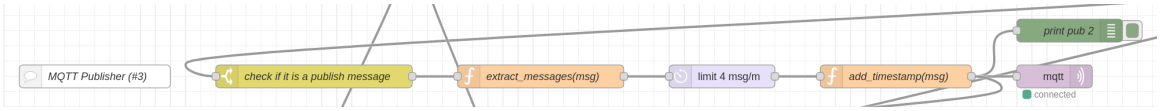

Figure 4: 2nd MQTT Publisher part

3.3.1 Flow and nodes description

- **check if it is a publish message**: this switch node verifies if the payload.Info field of the message filtered in the subscriber part contains the string "Publish Message," indicating that it is a publish message. If the condition is met, the message is forwarded to the next node.
- **extract_messages(msg)**: this function node extracts multiple messages from a single payload, as each publish message can contain several messages. It gathers these messages into an array, ensuring that each message is sequentially passed on to the next node in the flow.
  The original payload typically contains strings like "Publish Message," along with topics and payloads that need extraction. Here's how the function works:

  1. It divides the original payload into arrays of topics and payloads.
  2. For each topic and payload pair, it creates a new message.
  3. These messages are stored in an array, which is then returned.

  Additionally, the function assigns a Quality of Service (QoS) for each message and incorporates the ID of the extracted message from the subscriber part.

The function code is the following:

```
const info = msg.payload.Info;
const n = info.split("Publish Message").length - 1;

var topics = info.split('Publish Message ').join('');
topics = topics.substring(topics.indexOf('[')).split('[').join('').split(']'.join('')
.split(',');

const payloads = (msg.payload.Payload || "").split(',{').join('').split('}');

var msgs = [];

for (var i = 0; i < n; i++) {
   var newMsg = JSON.parse(JSON.stringify(msg));

   newMsg.topic = topics[i];
   newMsg.qos = 2;

   if (payloads[0].length > 0) {
      if(i == 0) {
         var payload_str = payloads[i] + "}";
      }
      else {
         var payload_str = "{" + payloads[i] + "}";
      }
      var payload = JSON.parse(payload_str);
   }
   else {
      var payload = JSON.parse("{}");
   }
   newMsg.payload = {
      "id": flow.get("sub_id"),
      "payload": payload,
   };

   msgs.push(newMsg);
}

return [msgs];
```

- **limit 4 msg/m**: this rate limit node restricts the number of messages sent to 4 per minute, as requested by the challenge requirements.
- **add_timestamp(msg)**: this function node appends a timestamp to each message. The timestamp is generated using the Date.now() function and stored in the msg.payload.timestamp field. The function code is the following:

```
msg.payload = {
   "timestamp": Date.now(),
   "id": msg.payload.id,
   "payload": msg.payload.payload
};

return msg;
```

- **mqtt**: this MQTT Publisher node transmits the message to the broker. The topic and QoS are dynamically determined by the previous node (extract_messages). The server (broker) is configured as 127.0.0.1:1884.
- **print pub 2**: this debug node displays the message sent to the broker.

3.4 What to do? (4): Chart

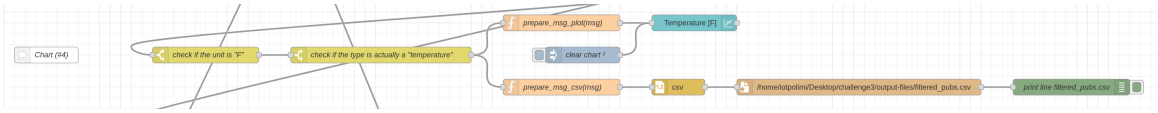

Figure 5: Chart part

3.4.1 Flow and nodes description

- **check if the unit is "F"**: this switch node examines if the msg.payload.payload.unit generated by the add_timestamp function (from the previous part) is "F". If the condition holds true, the message is forwarded to the next node.
- **check if the type is actually a "temperature"**: this switch node verifies if the msg.payload .payload.type generated by the add_timestamp function (from the previous part) is "temperature". If the condition holds true, the message is forwarded to the next node.
  Note: while this node might not be strictly necessary, it serves as an additional verification step. Using two switch nodes enables the handling of AND conditions, as the switch node doesn't seem to support this directly.
- **prepare_msg_plot(msg)**: this function node prepares the message for plotting in the chart. Each value is computed following the challenge specifications, which involve averaging the elements of the vector msg.payload.payload.range. The computed value is then stored in the msg.payload field. The function code is the following:

```
msg.payload = (msg.payload.payload.range[0] + msg.payload.payload.range[1])/2;
return msg;
```

- **clear chart**: This inject node clears the chart each time the flow is restarted to ensure it starts empty. It achieves this by injecting an empty JSON array.
- **Temperature [F]**: This chart node plots the message received from the previous node. The chart is configured to display the message with a horizontal axis of 1 hour.

  The following page shows the image of the temperature graph generated along with the attached filtered_pubs.csv file.

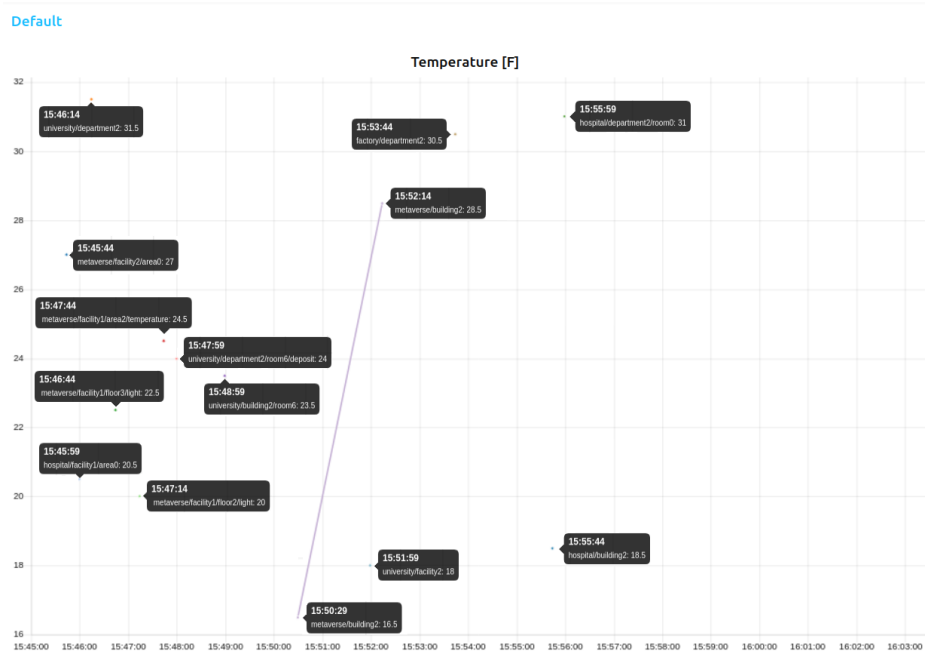

Figure 6: Temperature chart generated

- **prepare_msg_csv(msg)**: this function node prepares the message for storage in the CSV file. The message is formatted as required by the challenge, including the fields: No., LONG, RANGE, LAT, TYPE, UNIT and DESCRIPTION.
  Additionally, it increments the counter (msg_no = No.) and stores it in the respective context variable. The function code is the following:

```
var msg_no = context.get('msg_no') || 0;
msg_no++;

context.set('msg_no', msg_no);

msg.payload = {
   "No.": msg_no,
   "LONG": msg.payload.payload.long,
   "RANGE": msg.payload.payload.range,
   "LAT": msg.payload.payload.lat,
   "TYPE": msg.payload.payload.type,
   "UNIT": msg.payload.payload.unit,
   "DESCRIPTION": msg.payload.payload.description
};

return msg;
```

- **csv**: this CSV node converts the msg.payload (JSON object) into CSV format. It's configured to transmit the message header once, along with the first message.
- **/home/iotpolimi/Desktop/challenge3/output-files/filtered_pubs.csv**: this file node saves the message in the CSV file named filtered_pubs.csv, located in the /home/iotpolimi/Desktop/challenge3/output-files/ directory.
- **print line filtered_pubs.csv**: this debug node prints the message saved in the CSV file, verifying its correctness.

3.5 What to do? (5): MQTT ACKs

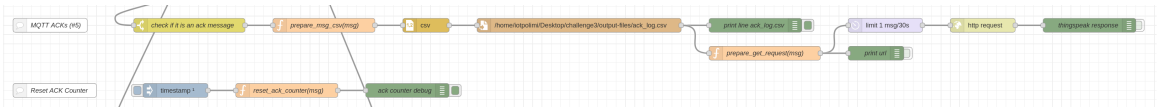

Figure 7: MQTT ACKs part

3.5.1 Flow and nodes description

- **check if it is an ACK message**: this switch node verifies if the payload.Info field of the message filtered in the subscriber part contains the string "ACK," indicating that it is an ACK message. If the condition holds true, the message is forwarded to the next node.
- **prepare_msg_csv(msg)**: this function node prepares the message to be saved in the CSV file. The message is generated as requested by the challenge, with the fields: TIMESTAMP, SUB_ID and MSG_TYPE. The function code is the following:

```
var ack_counter = global.get('ack_counter') || 0;
ack_counter++;

global.set('ack_counter', ack_counter);

var msg_type = msg.payload.Info.match(/(\S+\s+Ack)/);



msg.payload = {
   "TIMESTAMP": Date.now(),
   "SUB_ID": flow.get("sub_id"),
   "MSG_TYPE": msg_type[0]
};

return msg;
```

  Additionally, it increments the ack counter and stores it in the respective global variable. This variable resets when the flow restarts, thanks to the "Reset ACK Counter" section that is composed of:

  - **timestamp**: an inject node used solely to trigger the flow at each restart.
  - **reset_ack_counter(msg)**: this function node resets the ack counter by setting the global variable ack_counter to 0 and preparing the debug string for printing. The function code is the following:

```
global.set('ack_counter', 0);
msg.payload = global.get('ack_counter');
msg.payload = "ack counter reset correctly"
return msg;
```

  - **ack counter debug**: a debug node that prints the message generated by the previous node.

- **csv**: this CSV node converts the msg.payload (JSON object) into CSV format. It's configured to transmit the message header once, along with the first message.

- **/home/iotpolimi/Desktop/challenge3/output-files/ack_log.csv**: This file node saves the message in the CSV file named ack_log.csv, located in the /home/iotpolimi/Desktop/challenge3/output-files/ directory.
- **print line ack_log.csv**: this debug node prints the message saved in the CSV file, verifying its correctness.
- **prepare_get_request(msg)**: this function node prepares the message to be sent as a GET request to the Thingspeak server, setting up the msg.url with the write API and the ack counter. The function code is the following:

```
var write_api_key = "OSQEBQNP8IYBQCWN";

msg.url = "https://api.thingspeak.com/update?api_key="+write_api_key+"&field1="
+global.get('ack_counter');

return msg;
```

- **print url**: this debug node prints the URL of the GET request.
- **limit 1 msg/30s**: this rate limit node restricts the message sending rate to 1 message every 30 seconds. It's necessary because the Thingspeak server has a limit of 1 request every 20/30 seconds (I'm not sure about the exact value).
- **http request**: this HTTP Request node sends the message to the Thingspeak server with the method set to GET.
- **thingspeak response**: this debug node prints the response received from the server.

The following is the image of the ack counter chart generated at the Thingspeak page along with the attached ack_log.csv file.

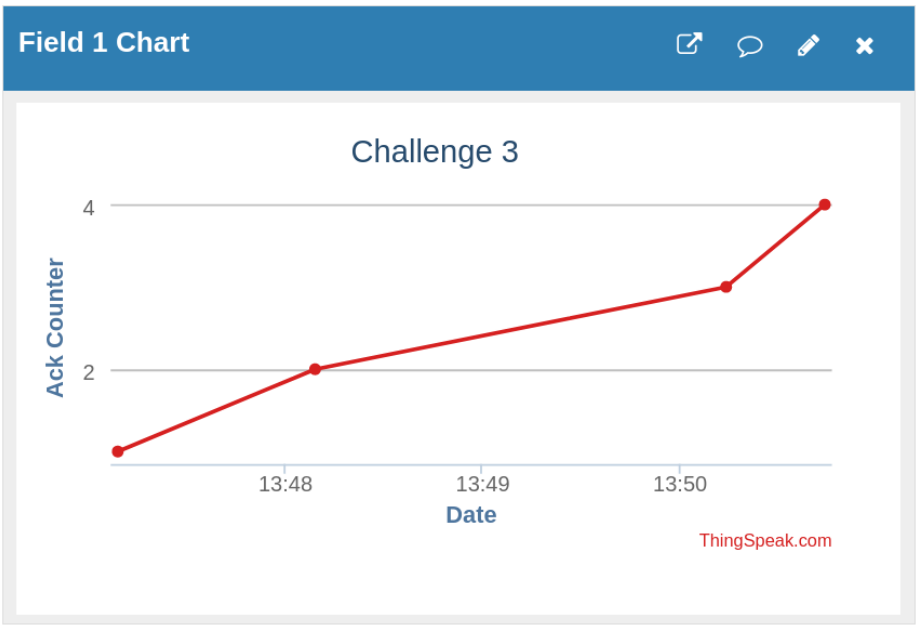

Figure 8: Ack Counter chart at the Thingspeak page

This chart can be accessed at the following link: `https://thingspeak.com/channels/2504435`.

3.6 What to do? (6): Stop after 80 msgs

Figure 9: Stop after 80 msgs part

3.6.1 Flow and nodes description

- **count_msgs(msg)**: this function node counts the number of messages received from the MQTT Subscriber node. This value is stored in a context variable named msg_no. Additionally, the counter is set in the msg.payload.counter field to preserve other fields needed in the compute_remainder function in the Subscriber part (#2). The function code is the following:

```
var msg_no = context.get('msg_no') || 0;
msg_no++;

context.set('msg_no', msg_no);

msg.payload.counter = msg_no;

return msg;
```

- **check if msg counter <= 80**: this switch node verifies if the counter is less than or equal to 80. If the condition is met, the message is forwarded to the next nodes, including the compute_remainder function node in the Subscriber part (#2).
- **debug_cycle(msg)**: this function node prepares the string utilized in the debug node, containing the current cycle number (which saturates at 80, as only the subscriber part is executed a limited number of times). The function code is the following:

```
switch(msg.payload.counter) {
   case 1:
      msg.payload = "1st subscriber cycle";
      break;
   case 2:
      msg.payload = "2nd subscriber cycle";
      break;
   case 3:
      msg.payload = "3rd subscriber cycle";
      break;
   default:
      msg.payload = msg.payload.counter +"th subscriber cycle"
}

return msg;
```

- **debug cycle**: this debug node prints the message generated by the previous node.

4 Conclusions

In conclusion, the project was successfully implemented using Node-RED, in a simple and intuitive way. It was a good exercise to understand the potential of this tool and to make in practice the concepts learned during the course.