Internet of things

IoT Challenge #3
Node-Red

Authors:
Daniel Shala - 10710181
Jurij Diego Scandola - 10709931

Academic Year:
2024 - 2025

1 Introduction

The objective of this document is to present and describe a Node-RED flow developed to manage and monitor the lifecycle of sensor data. This includes reading from CSV files, publishing and receiving messages via MQTT, logging acknowledgments, filtering data, and visualizing results both locally and through a cloud platform.

This flow represents a comprehensive IoT data pipeline designed for scalability, modularity, and integration with external services such as Thingspeak.

1.1 General Overview

The flow is composed of multiple functional blocks that are logically structured and interconnected. Overall, the flow follows the challenge rules that are:

- Generating and publishing random IDs to a broker every 5 seconds.
- Logging each published ID to a CSV file (`id_log.csv`).
- Subscribing to the same MQTT topic to receive messages.
- Mapping received IDs to rows in the CSV based on modulo 7711.
- Depending on the message type in the matched CSV row, forwarding messages, plotting temperature data in Fahrenheit, or logging acknowledgments to `ack_log.csv`.
- Limiting the number of processed messages to 80.
- Sending filtered temperature data to a chart and saving them in `filtered_pubs.csv`.
- Reporting ACK counts to Thingspeak.

1.2 Flow Breakdown

1. **Init read CSV**: At startup, a CSV file is read, parsed, and saved in memory using a flow variable for fast access during processing. Also, the function node contains a JS script to save the content parsed by the CSV file.

   ```
       flow.set("csvParsed", msg.payload);
       return msg;
   ```

2. **Trigger Send 5s**:
   - Every 5 seconds, a new message is created with a random ID (between 0 and 30000) and a timestamp.
   - The message is published to the topic `challenge3/id_generator`.

- Simultaneously, the message is parsed through a parsing node, from JSON to CSV and is logged into `id_log.csv` with row number, ID, and timestamp (using a storage node) The function node contains the following JS script. This code generates up to 80 MQTT messages, each containing a random ID and a timestamp, and numbers them progressively. Once it reaches 80 messages, it stops generating further messages.

```
        let count = flow.get("IdRowCount") || 0;

        if(count >= 80) {
            return null;
        }

        count += 1;
        flow.set("IdRowCount", count);

        let id = Math.floor(Math.random() * 30001);
        let timestamp = Math.floor(Date.now() / 1000);

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

        return msg;
```

3. **Read from topic**:

   - Subscribes to the same MQTT topic and read from it.
   - Upon reception, the ID is extracted and reduced modulo 7711 to compute an index N.
   - The CSV data is scanned for the row with frame number N. The JS script utilized in this section is the following one. It still checks for the 80 messages limit.

```
        let msgs = flow.get("CountMsgs") || 0;

        if(msgs >= 80) {
            return null;
        }

        flow.set("CountMsgs", msgs+1);

        let N = msg.payload.ID % 7711;
        flow.set("N",N);
        let rows = flow.get("csvParsed");

        let result = rows.find(r => r["No."] == N);

        msg.payload = result;

        return msg;
```

   - **If the matching row contains a `Publish Message`:**
     - The function node processes a received ”Publish” message by extracting all MQTT topics from the Info field and splitting the raw Payload into valid JSON objects. It

safely parses each payload and associates it with its corresponding topic. The result is an array of properly formatted messages, each containing the timestamp, topic, ID (N), and payload, ready to be published individually.

  – If the payload is of type `temperature` and unit `F`, the message is plotted and logged in `filtered_pubs.csv`. This is the JS script related to the node:

```
let str = msg.payload["Info"];
let matches = [...str.matchAll(/\[([^\]]+)\]/g)
    ];
let topics = matches.map(m => m[1]);
let N = flow.get("N");
let raw = msg.payload["Payload"] || "";
let rawParts = raw
    .split(/(?<=\})\s*,\s*(?=\{)/g)
    .map(p => p.trim());

let payloads = [];
for (let part of rawParts) {
    try {
        payloads.push(JSON.parse(part));
    } catch (err) {
        payloads.push("");
    }
}

let out = [];

for (let i = 0; i < topics.length; i++) {
    out.push({
        topic: topics[i],
        payload: {
            timestamp: Math.floor(Date.now() /
                1000).toString(),
            topic: topics[i],
            id: N,
            payload: payloads[i] || ""
        }
    });
}

return out;
```

  – The flow applies a rate limit of 4 messages/minute for this branch.

- **If the row contains an `ACK` message (e.g., Connect Ack, Sub Ack), an ACK counter is incremented.**
- The ACK information is saved in `ack_log.csv`, and the updated count is sent to Thingspeak via HTTP API. The function node contains the following JS script:

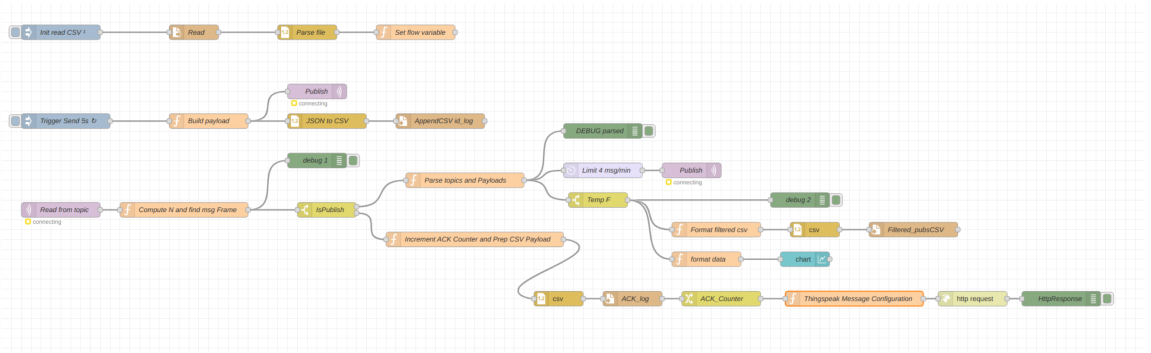
```
let ack_counter = flow.get("ACK_Counter") || 0;
flow.set("ACK_Counter", ack_counter+1);
let ack_row = ack_counter+1;
let N = flow.get("N");
let str = msg.payload["Info"];
let msg_type = "";
```

```
        let endIndex = str.indexOf("Ack");
        if (endIndex !== -1) {
            msg_type = str.substring(0, endIndex + 3);
        }

        let payload = {
            "No.": ack_row,
            TIMESTAMP: Math.floor(Date.now() / 1000).toString()
                ,
            SUB_ID: N,
            MSG_TYPE: msg_type
        }
        msg.payload = payload;

        return msg;
```

4. **Flow Termination Logic**:
    - A message counter ensures the flow processes only 80 subscription messages total.
    - Even discarded or unprocessable messages are counted toward this limit.

1.3 Chart and Visualization

Messages containing valid Fahrenheit temperature data are parsed to compute the average of the min and max range. These values are visualized in a real-time Node-RED chart. An example of the chart can be seen in the following figure.



This Node-RED flow demonstrates a complete and modular approach to handling IoT message lifecycles — from data ingestion, ID generation, and MQTT communication to data filtering, visualization, and cloud integration. The logic enforces message limits, differentiates between message types, and applies contextual actions such as storing, visualizing, or forwarding data.

The system is designed with scalability and clarity in mind, offering a strong foundation for future expansion. Additional improvements, such as better error handling, enhanced data persistence, and increased dashboard interactivity, could make the flow even more robust and suitable for production-level IoT applications.