



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

IoT Challenge #3, Node-RED Flow Report

INTERNET OF THINGS

Authors: **Kevin Zioldi - 10764177**
Matteo Volpari - 10773593

Professors: Alessandro Redondi, Fabio Palmese, Antonio Boiano
Academic Year: 2024-2025
Version: 1.0
Release date: 27-4-2025

Contents

Contents

1	Node-RED Flow	1
1.1	Introduction	1
1.2	Initial setup branch	1
1.2.1	Inject	2
1.2.2	Read challenge3.csv	2
1.2.3	csv	2
1.2.4	Save CSV	2
1.2.5	Prepare DELETE	2
1.2.6	http request	2
1.2.7	Clear CSV files	2
1.2.8	Initialization completed 5	2
1.2.9	Set initialization completed	2
1.3	MQTT Publisher branch	3
1.3.1	Inject (5 sec)	3
1.3.2	Create payload	3
1.3.3	challenge3/id_generator	4
1.3.4	Add row number	4
1.3.5	id_log.csv	4
1.3.6	Write id_log.csv	5
1.4	MQTT Subscriber branch	5
1.4.1	challenge3/id_generator	5
1.4.2	Compute N	5
1.4.3	Select message N	6
1.4.4	Handle PUBLISH message	7
1.4.5	Message Rate Limiter MQTT	10
1.4.6	Publish from CSV	10

1.4.7	Prepare Average temperature Fahrenheit	11
1.4.8	Average temperature (F) chart	11
1.4.9	Prepare filtered_pubs.csv	12
1.4.10	filtered_pubs.csv	13
1.4.11	Write filtered_pubs.csv	13
1.4.12	Handle ACK message	13
1.4.13	ack_log.csv	14
1.4.14	Write ack_log.csv	14
1.4.15	Prepare HTTP GET	15
1.4.16	Message Rate Limiter ThingSpeak	15
1.4.17	HTTP Request ThingSpeak	15

List of Figures	17
------------------------	-----------

List of Tables	18
-----------------------	-----------

1 | Node-RED Flow

1.1. Introduction

In this document there is the explanation of the entire Node-Red flow that aims to fulfil the requirements of Challenge 3. The flow is divided in 3 main sections: initial setup, MQTT publisher and MQTT subscriber. This document will explain how it works by analysing the individual nodes of the three sections mentioned above.

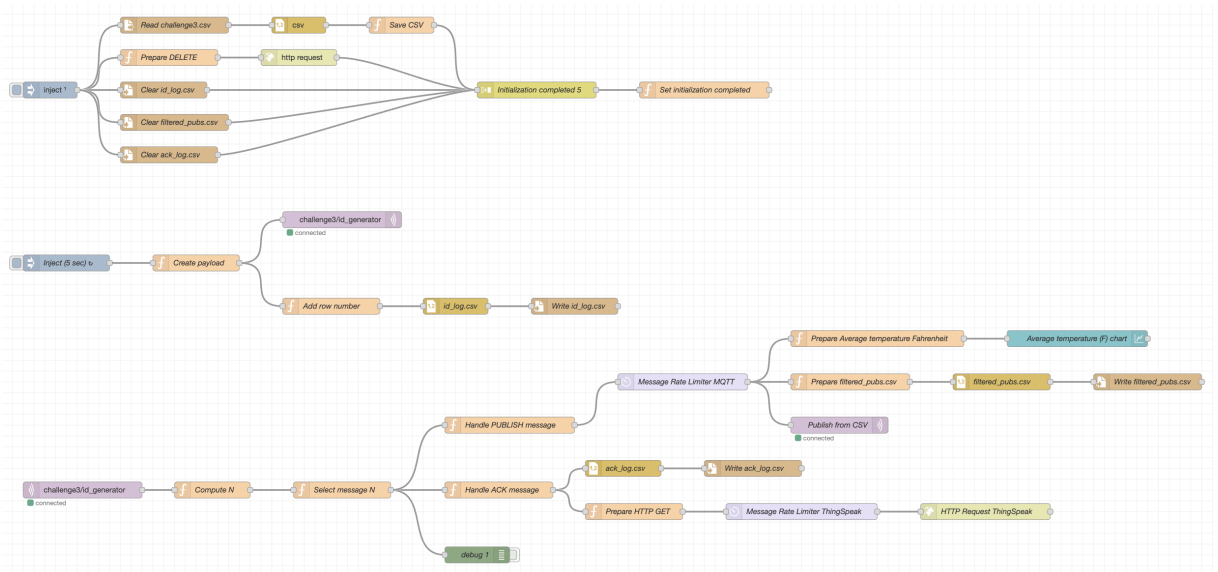


Figure 1.1: Complete Node-Red flow

1.2. Initial setup branch

In this branch, all operations considered preliminary to system start-up are carried out, such as loading the *challenge3.csv* file and cleaning the other CSV files that the system will have to write during execution.

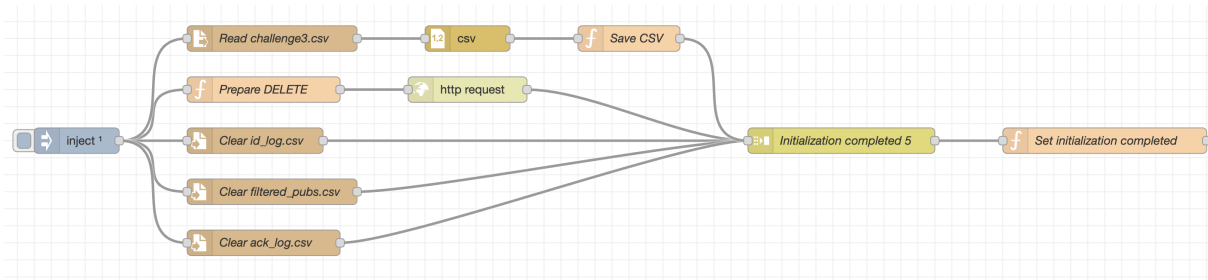


Figure 1.2: initial setup branch

1.2.1. Inject

When the system is deployed, this node starts this branch of the flow, which allows all preliminary operations to be carried out before starting the tasks of the other two branches of the flow.

1.2.2. Read challenge3.csv

1.2.3. csv

1.2.4. Save CSV

1.2.5. Prepare DELETE

1.2.6. http request

1.2.7. Clear CSV files

The three *Clear <file_name>.csv* nodes are used to clean up the three different CSV files that the system then has to write during execution. To clean up these files, an overwrite must be performed for each file.

1.2.8. Initialization completed 5

Questo nodo join permette di fermare il flow fin quando tutti e five i rami entranti non hanno terminato la loro esecuzione. Una volta che tutti e five i rami hanno terminato questo nodo fa andare avanti l'esecuzione passando a *Set initialization completed*.

1.2.9. Set initialization completed

1.3. MQTT Publisher branch

In the first branch of the flow, on intervals of five seconds, an MQTT publisher publishes a message onto the topic "challenge3/id_generator". In this section we will see in a detailed way the workflow of this part.

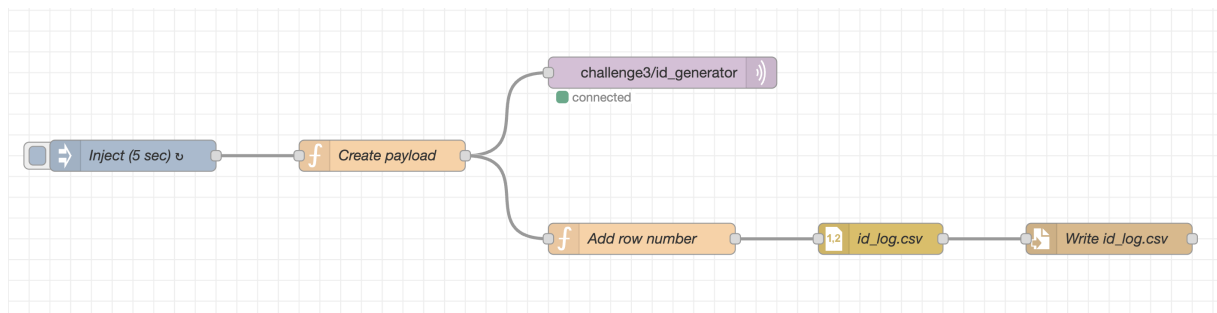


Figure 1.3: MQTT Publisher branch of the flow

1.3.1. Inject (5 sec)

This is the first node and the entry point for the whole system. It starts the process on a deploy and injects a new message every 5 seconds.

1.3.2. Create payload

This is a function node and its purpose is to generate the payload of messages that will be forwarded to the MQTT publisher.

The payload is composed by two fields:

- ID: this field is the Id of the message and is a random integer in the range of 0-30000.
- TIMESTAMP: this field represents the exact time the message was created. The code of the function node is reported here.

```

let initializationCompleted = flow.get("initializationCompleted") || false;
if (!initializationCompleted) {
    return null;
}
var id = Math.floor(Math.random() * 30000);
var timestamp = Math.floor(Date.now() / 1000);
msg.payload = {
    id: id,

```

```
        timestamp: timestamp
    };
    return msg;
```

1.3.3. challenge3/id_generator

This is the MQTT node that publishes the messages to the local MQTT broker running on port 1884. Messages are published to the topic "challenge3/id_generator", in this way the subscriber used in the second part of the flow could receive these messages.

1.3.4. Add row number

This is a function node and its goal is to add a row number to the message received from the node "Create payload".

This action is performed through the help of a context variable `row_count_id_log` that keeps the count of the processed messages. Then the row number is added to the message's payload and the message is forwarded.

```
var counter = context.get("row_count_id_log") || 0;
counter++;
context.set("row_count_id_log", counter);
var id = msg.payload["id"];
var timestamp = msg.payload["timestamp"];
msg.payload = {
    "No." : counter,
    "ID" : id,
    "TIMESTAMP": timestamp
}
return msg;
```

1.3.5. id_log.csv

This node is a csv node and is used to convert the JSON object contained in the payload into a CSV file. At the beginning it sends the header of the CSV that is composed by these fields: No., TIMESTAMP, SUB_ID, MSG_TYPE, then it sends all the rows to the node with the task of writing them.

1.3.6. Write id_log.csv

This is a write node and is used to write the actual CSV file `id_log.csv` saved to the disk in the path specified inside the node.

1.4. MQTT Subscriber branch

In the second branch of the flow, there is an MQTT subscriber that subscribes to the topic "challenge3/id_generator" and processes messages received from the MQTT broker. In this section, we will describe all nodes that are in this part of the flow.

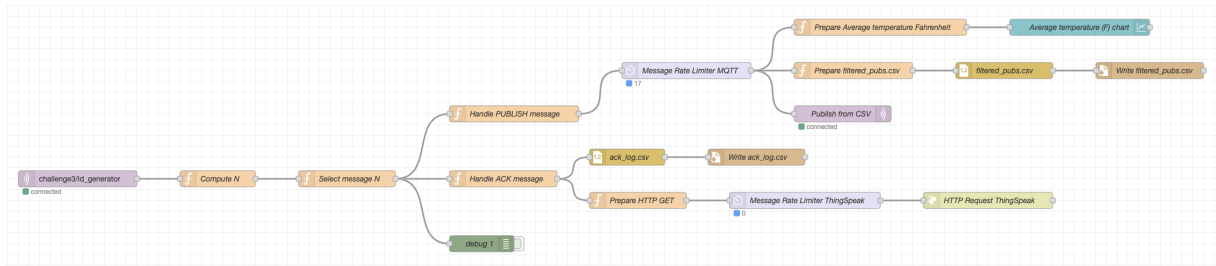


Figure 1.4: MQTT Subscriber branch of the flow

1.4.1. challenge3/id_generator

The first node is an mqtt in node, that is subscribed to the topic "challenge3/id_generator" to the local MQTT broker on port 1884. This node receives messages sent to the local broker by the publisher in the first branch of the flow and processes them with the following nodes.

1.4.2. Compute N

This node is a function node that performs two main tasks:

- It increases by 1 the value of a context variable, called `processed_messages`, that counts the number of messages processed up to a certain moment. The function node also checks if the value is greater than 80, in which case, the node discards the message.
- The second task is performed only if the message needs to be processed: the node computes the value of `N` as $N = id \% 7711$, which will be needed to select the correct MQTT message from the CSV.

The code of the function node is reported here.


```

var processedMessages = context.get("processed_messages") || 0;
processedMessages ++;
context.set("processed_messages", processedMessages);
if (processedMessages > 80) {
    return null;
}
var id = msg.payload["id"];
var N = id % 7711;
console.log(N);
// save id and N
msg.id = id;
msg.N = N;
return msg;

```

1.4.3. Select message N

This node uses the value of N previously computed and extracts the correct row from the CSV file "challenge3.csv". If N is 0, the messages is discarded, since entries start from N=1, otherwise a row will be extracted. The function node reads the CSV from the flow variable `csv_challenge3` and then filters the row with field "No." equal to N. The extracted message will be managed by the following two functions nodes, but at most one of them will actually use it to send a new message.

The code of the function node is reported here.

```

if (msg.N === 0) {
    return null;
}
let rows = flow.get("csv_challenge3");
if (!rows) {
    console.log("NULL ROWS ")
    return null;
}
let target = rows.find(r =>
    parseInt(r["No."], 10) === msg.N
);
msg.payload = target;
return msg;

```

1.4.4. Handle PUBLISH message

This node is a function node, it receives a message corresponding to the selected message from the CSV file "challenge3.csv", checks if it is a PUBLISH message and, if it is, prepares the new MQTT messages to publish. In order to prepare the messages to be published, we need to extract topics, extract payloads and map them together.

Topics

The topics are contained in the "Info" field of the message and we can extract them with a regex, since they are in the form "Publish Message [*topic*]".

Payloads

The payloads, instead, are contained in the "Payload" field of the message. They are contained in curly braces and are similar to a JSON array, but without square brackets. We extract them by adding the missing square brackets and parsing them as a JSON array. There are some messages in which the payload is malformed and cannot be parsed as previously explained. We analyzed all PUBLISH messages with a Python script and found out that there are 3 messages with malformed payload and all of them have a certain number of well-formed messages, formed by a last message malformed message, which is truncated.

Since we don't want to waste the correct information contained in the "Payload" field, we use a regex to extract the payloads one at a time from opening curly brace to the corresponding closing one and we parse the text contained between curly braces as a JSON object. We discard the last, malformed, payload.

The Python script and the results of the run are reported here.

```
import pandas as pd
import json

df = pd.read_csv("challenge3.csv")

# filter MQTT PUBLISH messages
publish_df = df[df['Info'].str.contains('Publish Message', na=False)]

# filter ACK messages
ack_df = df[df['Info'].str.contains("Connect Ack", na=False)] + df[df['Info'].str.contains("Publish Ack", na=False)] + df[df['Info'].str.contains("Subscribe Ack", na=False)]
```

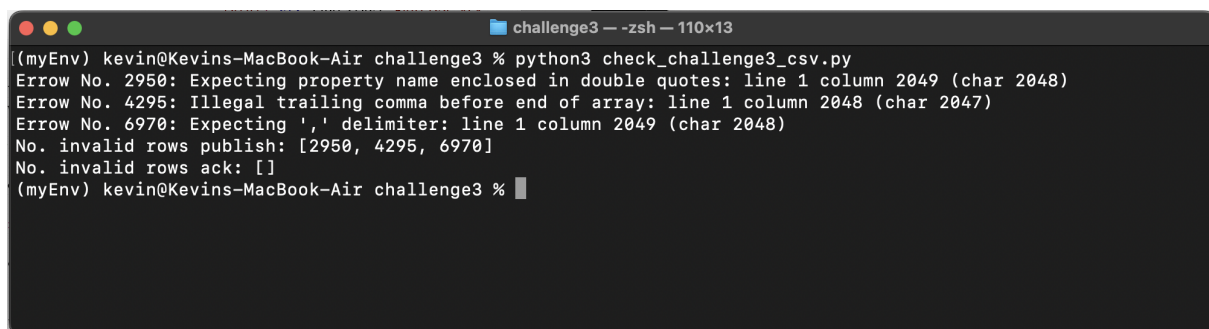
```

invalid_rows_publish = []
invalid_rows_ack = []

# find invalid PUBLISH messages
for idx, raw_payload in zip(publish_df.index, publish_df['Payload']):
    payload_str = raw_payload if isinstance(raw_payload, str) else ''
    if not payload_str.strip():
        continue
    try:
        json.loads(f'[{payload_str}]')
    except json.JSONDecodeError as e:
        print(f"Error No. {idx+1}: {e}")
        invalid_rows_publish.append(idx + 1)
print("No. invalid rows publish:", invalid_rows_publish)

# find invalid ACK messages
for idx, raw_payload in zip(ack_df.index, ack_df['Payload']):
    payload_str = raw_payload if isinstance(raw_payload, str) else ''
    if not payload_str.strip():
        continue
    try:
        json.loads(f'[{payload_str}]')
    except json.JSONDecodeError as e:
        print(f"Error No. {idx+1}: {e}")
        invalid_rows_ack.append(idx + 1)
print("No. invalid rows ack:", invalid_rows_ack)

```



```

challenge3 - zsh - 110x13
(myEnv) kevin@Kevins-MacBook-Air challenge3 % python3 check_challenge3_csv.py
Error No. 2950: Expecting property name enclosed in double quotes: line 1 column 2049 (char 2048)
Error No. 4295: Illegal trailing comma before end of array: line 1 column 2048 (char 2047)
Error No. 6970: Expecting ',' delimiter: line 1 column 2049 (char 2048)
No. invalid rows publish: [2950, 4295, 6970]
No. invalid rows ack: []
(myEnv) kevin@Kevins-MacBook-Air challenge3 %

```

Figure 1.5: Python script malformed messages run

Complete messages

Finally, we need to combine topics and payloads, we do that by using the map operation. If the number of topics is greater than the number of payloads, we assign an empty payload

to the topics without a payload. Beyond topic and payload, the produced messages will have the current timestamp and the id, which was saved in the message from the Compute N function node. The produced MQTT message will have the following form:

```
'{
  "timestamp": "CURRENT_TIMESTAMP",
  "id": "SUB_ID",
  "topic": "MQTT_PUBLISH_TOPIC",
  "payload": "MQTT_PUBLISH_PAYLOAD"
}'
```

In order to have the timestamp and id quoted, we convert them to strings inside the function node.

The code of the function node is reported here.

```
if (msg.payload["Protocol"] === "MQTT" &&
    msg.payload["Info"].includes("Publish Message")) {
  // save info in a variable
  let info = msg.payload["Info"];
  // save messages payloads in a variable
  let messagesPayloads = msg.payload["Payload"] ?? "";
  // extract topics
  let topicMatches = [...info.matchAll(/Publish Message \[([^\]]+)\]/g)];
  let topics = topicMatches.map(m => m[1]);
  // extract payloads
  let jsonArray;
  if (messagesPayloads.trim() === "") {
    jsonArray = [];
  } else {
    try {
      jsonArray = JSON.parse("[ " + messagesPayloads + " ]");
    } catch (e) {
      // extract well formed messages

      const objs = [];
      const objRegex = /\{[^\}]*\}/g;
      let m;
      while ((m = objRegex.exec(messagesPayloads)) !== null) {
```

```

        try {
            objs.push(JSON.parse(m));
        } catch (_) {
            // skip malformed object
        }
    }
    jsonArray = objs;
}
}
// create messages
var currTimestamp = Math.floor(Date.now() / 1000);
let messages = topics.map((topic, index) => {
    return {
        topic: topic,
        payload: {
            timestamp: currTimestamp.toString(),
            id: msg.id.toString(),
            topic: topic,
            payload: jsonArray[index] ?? {}
        }
    };
});
return [messages];
}
return null;

```

1.4.5. Message Rate Limiter MQTT

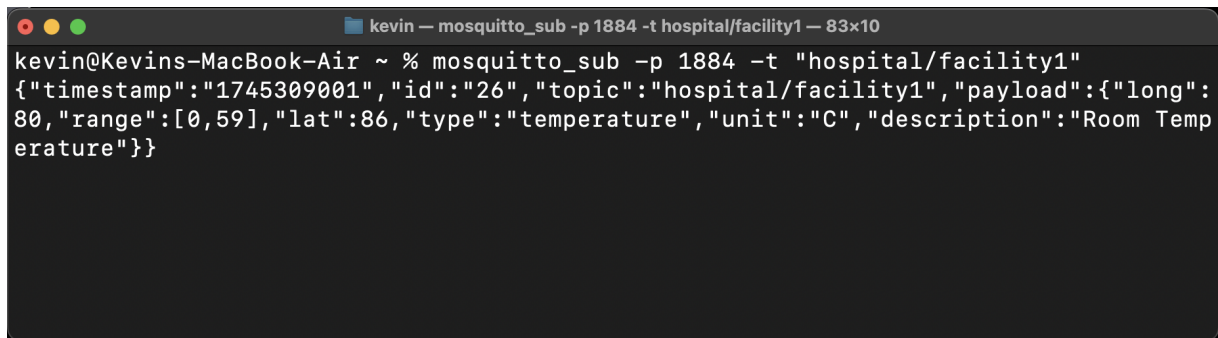
Among all extracted messages, we need to publish up to 4 messages per minute. We do that thanks to a delay node, that is set to work with action "Rate Limit" and limits the rate to 4 messages per minute, queueing intermediate messages.

1.4.6. Publish from CSV

This node is a mqtt out node that publishes messages that manage to go through the Message Rate Limiter. They are published to the local broker, on a dynamic topic, that was extracted from the CSV file and is saved in the message itself.

We show the correct functioning by subscribing to a topic from a terminal and showing

the message arriving to the subscriber.

A terminal window titled 'kevin — mosquitto_sub -p 1884 -t hospital/facility1 — 83x10'. The terminal shows a JSON message received from an MQTT broker: {"timestamp": "1745309001", "id": "26", "topic": "hospital/facility1", "payload": {"long": 80, "range": [0, 59], "lat": 86, "type": "temperature", "unit": "C", "description": "Room Temperature"}}.

```
kevin@Kevins-MacBook-Air ~ % mosquitto_sub -p 1884 -t "hospital/facility1"
{"timestamp": "1745309001", "id": "26", "topic": "hospital/facility1", "payload": {"long": 80, "range": [0, 59], "lat": 86, "type": "temperature", "unit": "C", "description": "Room Temperature"}}
```

Figure 1.6: MQTT client subscribed to topic "hospital/facility1"

1.4.7. Prepare Average temperature Fahrenheit

This node is a function node that receives messages that went through the rate limiter and were published. The function checks if the value contained in the payload of the message is a temperature in Fahrenheit and, if it is, computes the average temperature starting from the range contained in the payload. The prepared message will be used to plot the average temperature in Fahrenheit.

The code of the function node is reported here.

```
if (msg.payload.payload.type === "temperature"
    && msg.payload.payload.unit === "F") {
  let minTemp = msg.payload.payload.range[0];
  let maxTemp = msg.payload.payload.range[1];
  let avgTemp = (minTemp + maxTemp)/2;
  msg.topic = "TempF";
  msg.payload = avgTemp;
  return msg;
}
return null
```

1.4.8. Average temperature (F) chart

This node is a chart node and is used to plot in Node-RED the average temperature in Fahrenheit of the published messages.

An example of produced chart is reported here.

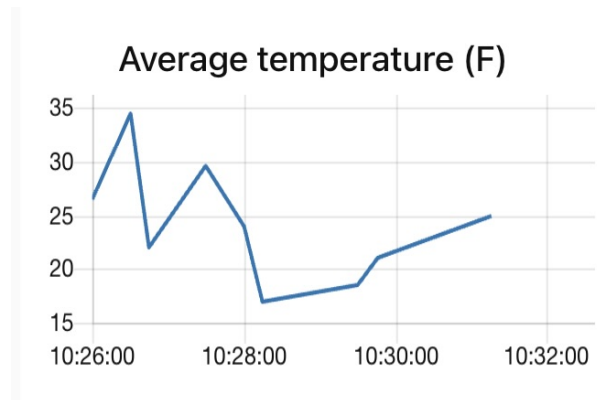


Figure 1.7: Average Fahrenheit temperature chart in Node-RED

1.4.9. Prepare filtered_pubs.csv

This function node takes all PUBLISH messages that were published and filters the same ones as the other function node, i.e. all the ones containing a temperature in Fahrenheit. The function increases by one the counter for the number of rows in the CSV file `filtered_pubs.csv`, contained in the context variable `row_count_filtered_pubs`. Moreover, the function produces the new row of the CSV using the data contained in the payload of the message and the value of the row counter. The payload must be a JSON object with fields matching the ones of the CSV, in order to work seamlessly with the csv node.

The code of the function node is reported here.

```
if (msg.payload.payload.type === "temperature"
    && msg.payload.payload.unit === "F") {
  var counter = context.get("row_count_filtered_pubs") || 0;
  counter++;
  context.set("row_count_filtered_pubs", counter);
  let data = msg.payload.payload;
  let meanValue = (data.range[0] + data.range[1]) / 2;
  let long = data.long;
  let lat = data.lat;
  let type = data.type;
  let unit = data.unit;
  let desc = data.description;
  msg.payload = {
    "No." : counter,
```

```

        "LONG": long,
        "LAT": lat,
        "MEAN_VALUE": meanValue,
        "TYPE": type,
        "UNIT": unit,
        "DESCRIPTION": desc
    }
    return msg;
}
return null;

```

1.4.10. filtered_pubs.csv

This node is a csv node and is used in the "Object to CSV" way to save the JSON object contained in the payload to a row of the CSV. At first, the node produces the header: No.,LONG, LAT, MEAN_VALUE, TYPE, UNIT, DESCRIPTION. Then it produces the following lines based on the JSON object it receives, which has the same fields as the CSV.

1.4.11. Write filtered_pubs.csv

This node is a write file node as is used to write the actual CSV file filtered_pubs.csv, which is saved to a location on the disk specified in the path inside the node.

1.4.12. Handle ACK message

This node is a function node that is used to handle ACK messages retrieved from the challenge3.csv file. The function checks if the message is an MQTT ACK message, looking at the "Info" field: if it contains the word "Ack", it is an ACK message. The function increases a row counter for the CSV file ack_log.csv, contained the context variable row_count_ack_log and prepares the JSON object to be written in the CSV. The produced messages will have in the payload all fields of the CSV file ack_log.csv: the incremental row counter in "No.", the current timestamp, the id of the original message and the message type, which the function finds in the "Info" field, together with the "Ack" keyword.

The code of the function node is reported here.

```

if (msg.payload["Protocol"] === "MQTT"

```



```

    && msg.payload["Info"].includes("Ack")) {
var counter = context.get("row_count_ack_log") || 0;
counter++;
context.set("row_count_ack_log", counter);
var currTimestamp = Math.floor(Date.now() / 1000);
let idMatch = msg.id;
var type = "";
if (msg.payload["Info"].includes("Connect Ack")) {
    type = "Connect Ack";
} else if (msg.payload["Info"].includes("Publish Ack")){
    type = "Publish Ack";
} else if (msg.payload["Info"].includes("Subscribe Ack")) {
    type = "Subscribe Ack";
}
msg.payload = {
    "No.": counter,
    "TIMESTAMP": currTimestamp,
    "SUB_ID": idMatch,
    "MSG_TYPE": type
}
return msg;
}
return null;

```

1.4.13. ack_log.csv

This node is a csv node and is used in the "Object to CSV" way to save the JSON object contained in the payload to a row of the CSV ack_log.csv. The csv node produces a header No., TIMESTAMP, SUB_ID, MSG_TYPE and all the rows of the CSV, matching the fields of the CSV with the ones of the message payload.

1.4.14. Write ack_log.csv

This node is a write file node as is used to write the actual CSV file ack_log.csv, which is saved to a location on the disk specified in the path inside the node.

1.4.15. Prepare HTTP GET

This node is a function node that prepares the HTTP message to send to the ThingSpeak channel. It sets the method to GET and the URL to the one exposed by the API, passing the API KEY and the value of the global ack counter, contained in the message payload and computed in the Handle ACK message function node.

The code of the function node is reported here.

```
var API_KEY="SECRET_API_KEY"
var globalCounter = msg.payload["No."]
msg.method = "GET";
msg.url = "https://api.thingspeak.com/update?api_key="+
        API_KEY+"&field1="+globalCounter

return msg;
```

1.4.16. Message Rate Limiter ThingSpeak

This node is a a delay node, that is set to work with action "Rate Limit" and limits the rate to 1 messages every 20 seconds, queueing intermediate messages. We introduced this node to cope with the free plan limit of ThingSpeak that imposes a limit of 1 message every 15 seconds; by sending one message every 20 seconds and queueing messages, we don't loose any message.

1.4.17. HTTP Request ThingSpeak

This node is an http request node that sends the HTTP request to ThingSpeak, using the method and URL set in the message. The link of the public ThingSpeak channel is <https://thingspeak.mathworks.com/channels/2924504/>.

An example of produced chart is reported here.

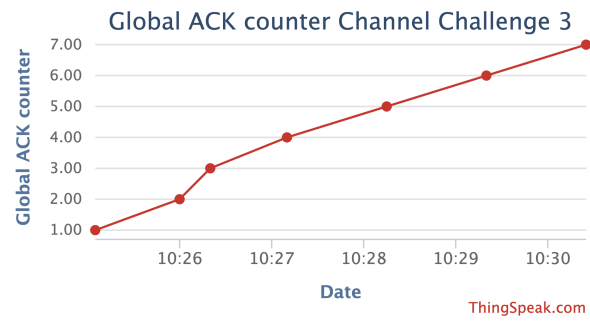


Figure 1.8: Global ACK counter chart in ThingSpeak

List of Figures

1.1	Complete Node-Red flow	1
1.2	initial setup branch	2
1.3	MQTT Publisher branch of the flow	3
1.4	MQTT Subscriber branch of the flow	5
1.5	Python script malformed messages run	8
1.6	MQTT client subscribed to topic "hospital/facility1"	11
1.7	Average Fahrenheit temperature chart in Node-RED	12
1.8	Global ACK counter chart in ThingSpeak	16

List of Tables