POLITECNICO
MILANO 1863

SCUOLA DELL’INGEGNERIA INDUSTRIALE E DELL’INFORMAZIONE

COMPUTER SCIENCE AND ENGINEERING

Internet Of Things
3rd challenge

Simone Pio Bottaro - 10774229

Gabriele Lorenzetti – 10730455

A.Y 2024/2025

Introduction

The task involved creating a Node-Red flow capable of periodically generating JSON messages containing a random identifier and a timestamp and publishing them to a local MQTT broker.

At the same time, in another branch, we implemented a subscription to the same topic, allowing us to receive the published messages and process them based on information contained in a reference CSV file (***challenge3.csv***). Depending on the content associated with the received identifiers, the system performed different actions: publishing new MQTT messages, saving data into specific CSV files, or handling acknowledgment (ACK) messages.

Special attention was given to filtering data related to temperatures expressed in degrees Fahrenheit, which were visualized on a Node-Red chart and separately archived in a csv file (***filtered_pubs.csv***). Additionally, a mechanism was created to track the total number of ACK messages received, periodically sending the updated count to a ThingSpeak channel via HTTP API. Each ACK message was also stored in another csv file (***ack_log.csv***)

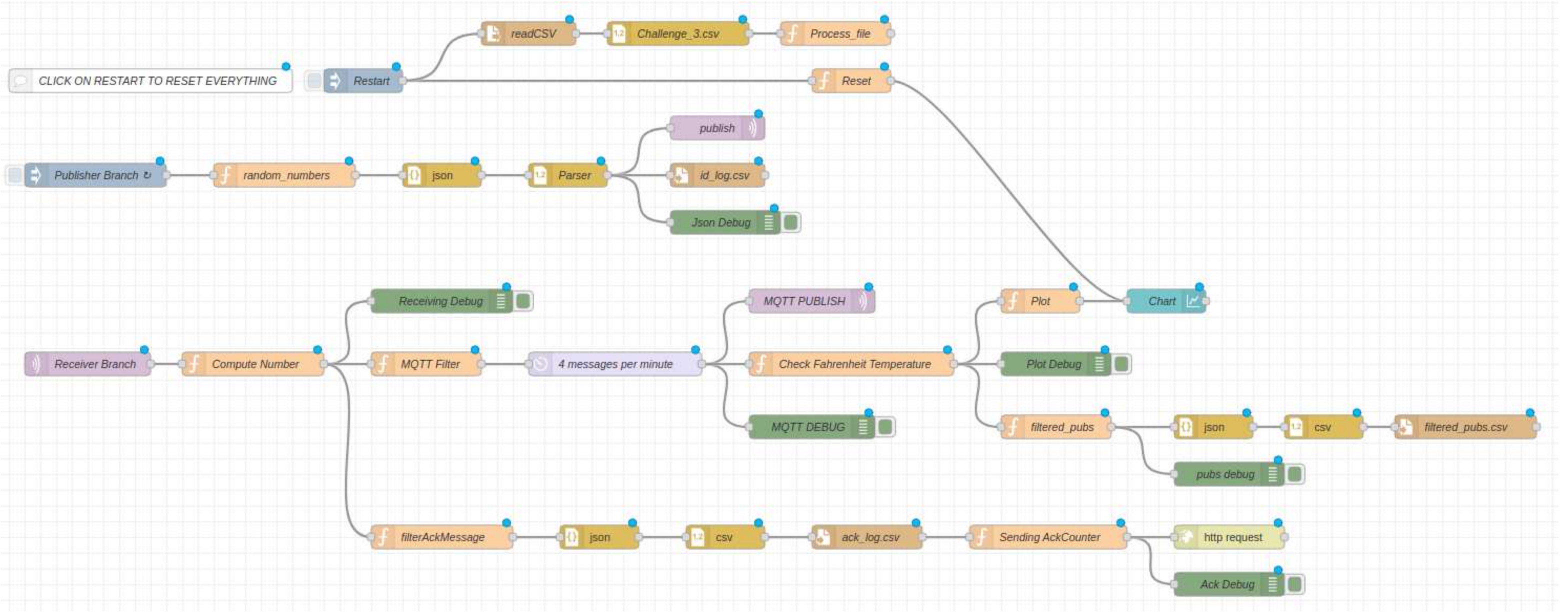
The receiving branch of the flow was designed to automatically stop after processing exactly 80 messages, strictly following all the conditions outlined in the Challenge description.

In the following sections of the report, we will provide a detailed explanation of the implemented components, the functioning of the Node-Red flow, and the results obtained.

Node-RED

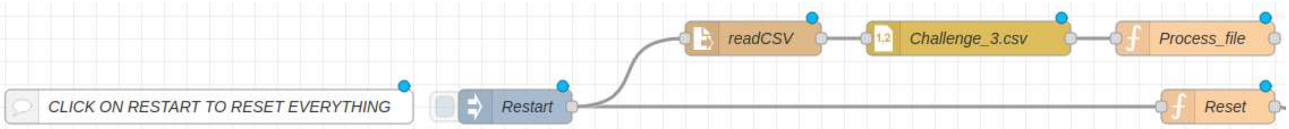
This is the all Node-Red flow

Thingspeak channel ID: https://thingspeak.mathworks.com/channels/2929937

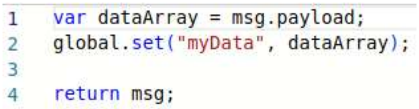


Setup Branch

This branch allows to restart the computation from the beginning, clearing the plot and setting the global variables to the default values.

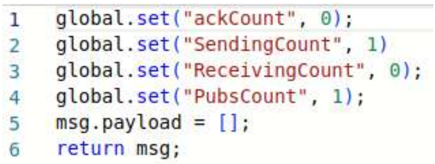


- **readCSV**: read the input file challenge3.csv
- **Challege3.csv**: parser the input csv file
- **Process_file**: this stores CSV data from msg.payload into a global variable"myData"



```
var dataArray = msg.payload;
global.set("myData", dataArray);

return msg;
```

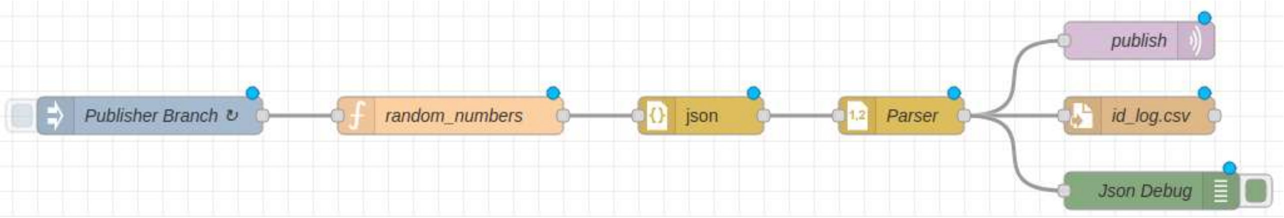
- **Reset**: this function is used to initialize the global variables and to clean the chart graph



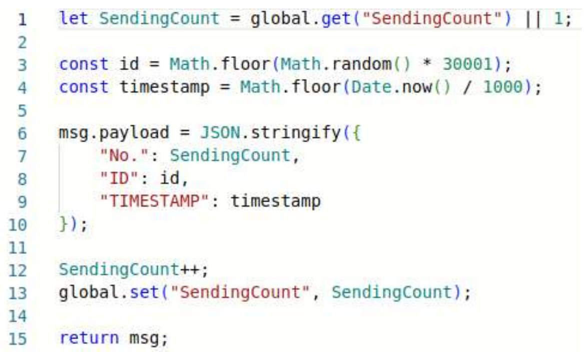
```
global.set("ackCount", 0);
global.set("SendingCount", 1)
global.set("ReceivingCount", 0);
global.set("PubsCount", 1);
msg.payload = [];
return msg;
```

Publisher Branch

This part deals with creating the ID and posting a message every 5 seconds in a specific topic



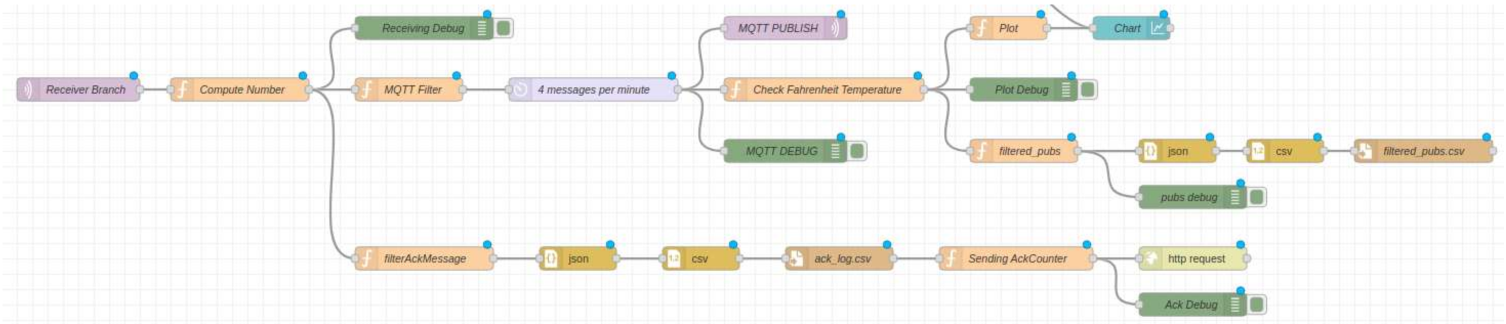
- **Publisher Branch**: repeatedly with a rate of 1 every 5seconds
- **random_numbers**: randomly generates number (id) between 0 and 30000, the timestamp and increments the ***SendingCount*** global variable



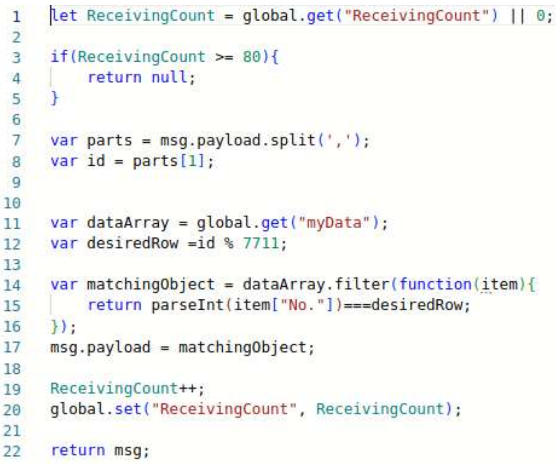
```
let SendingCount = global.get("SendingCount") || 1;

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

msg.payload = JSON.stringify({
    "No.": SendingCount,
    "ID": id,
    "TIMESTAMP": timestamp
});

SendingCount++;
global.set("SendingCount", SendingCount);

return msg;
```

- **id_log.csv**: writes the file id_log.csv
- **publish**: publish the various Publish Messages Payload to the topic ***“challenge/id_generator”*** (localhost, port 1884)

Receiver Branch

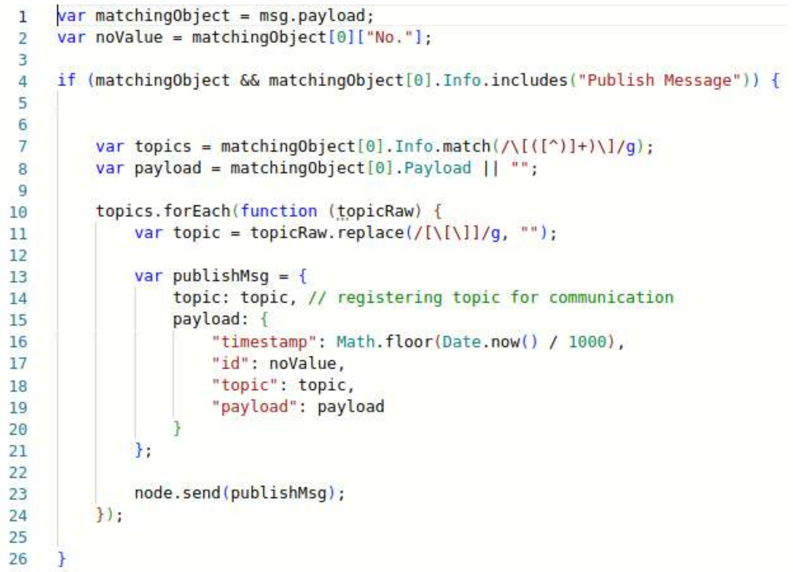


- **Receiver Branch**: Subscribed to the topic challenge3/id_generator in the local broker (localhost, port 1884)
- **Compute Number**: The function processes up to 80 messages. For each message, it extracts an ID from the payload, reduces it modulo 7711, searches the global array myData for an object whose "No." field matches the result, replaces the message's payload with the found object, increments the ReceivingCount counter by 1, and passes the modified message forward.



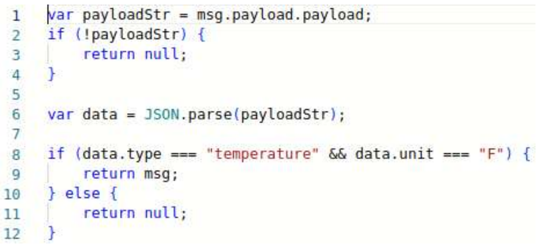
```
let ReceivingCount = global.get("ReceivingCount") || 0;

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

var parts = msg.payload.split(',');
var id = parts[1];


var dataArray = global.get("myData");
var desiredRow =id % 7711;

var matchingObject = dataArray.filter(function(item){
    return parseInt(item["No."])===desiredRow;
});
msg.payload = matchingObject;

ReceivingCount++;
global.set("ReceivingCount", ReceivingCount);

return msg;
```

- **MQTT Filter**: The code checks if the incoming data mentions "Publish Message". If so, it extracts all topics listed inside square brackets from the Info field. For each topic, it builds a new message containing the timestamp, the ID, the topic and the payload, and then sends that message.



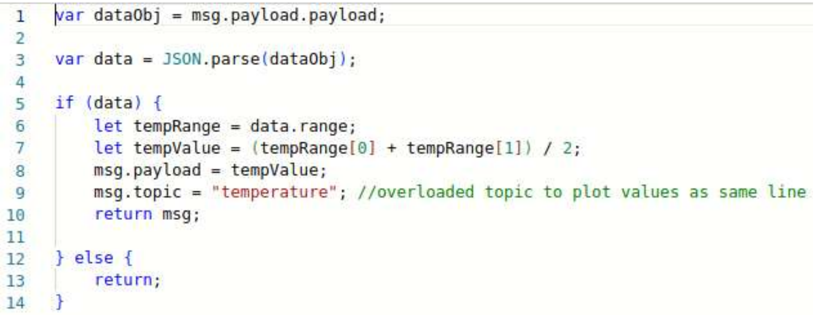
```
var matchingObject = msg.payload;
var noValue = matchingObject[0]["No."];

if (matchingObject && matchingObject[0].Info.includes("Publish Message")) {


    var topics = matchingObject[0].Info.match(/\[([^)]+)\]/g);
    var payload = matchingObject[0].Payload || "";

    topics.forEach(function (topicRaw) {
        var topic = topicRaw.replace(/[\[\]]/g, "");

        var publishMsg = {
            topic: topic, // registering topic for communication
            payload: {
                "timestamp": Math.floor(Date.now() / 1000),
                "id": noValue,
                "topic": topic,
                "payload": payload
            }
        };

        node.send(publishMsg);
    });

}
```

- **4 messages per minute**: only passes 4 messages per minute
- **MQTT Publish**: publish the various Publish Messages Payload to the respectively topics dynamically
- **Check Fahrenheit Temperature**: the code checks if the incoming message has a valid payload. If it does, it parses the payload as JSON and checks if it's a temperature reading in Fahrenheit ("F"). If both conditions are met, it lets the message through; otherwise, it discards it

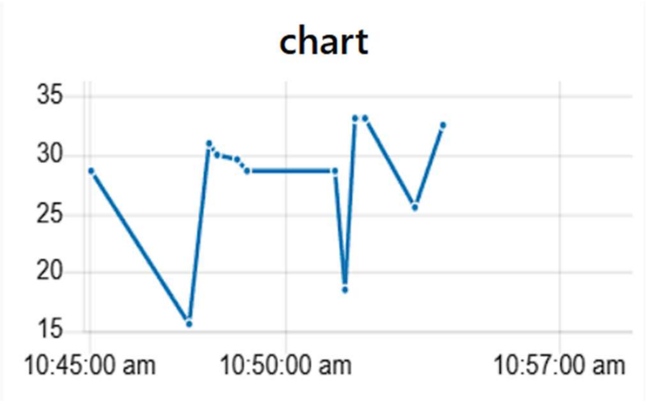


```
var payloadStr = msg.payload.payload;
if (!payloadStr) {
    return null;
}

var data = JSON.parse(payloadStr);

if (data.type === "temperature" && data.unit === "F") {
    return msg;
} else {
    return null;
}
```

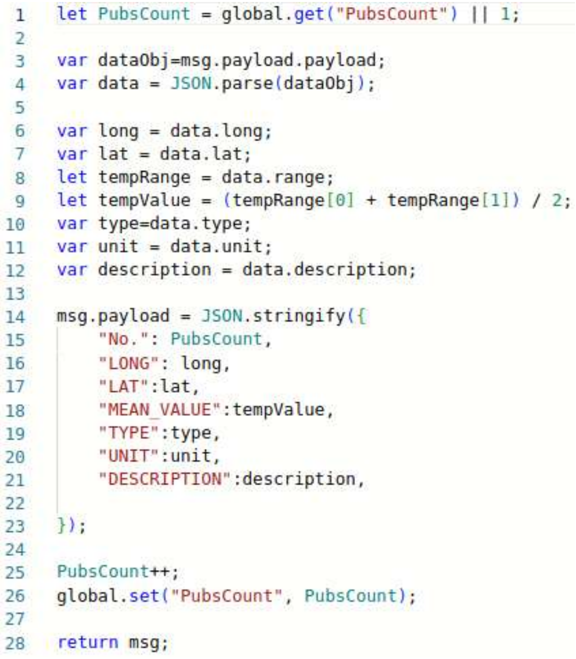
- **Plot**: the code parses a JSON payload from the incoming message. If successful, it calculates the average value between two numbers in the range field, sets that average as the new payload, assigns the topic "temperature", and sends the updated message forward. If parsing fails, it stops and returns nothing.



```javascript
var dataObj = msg.payload.payload;

var data = JSON.parse(dataObj);

if (data) {
    let tempRange = data.range;
    let tempValue = (tempRange[0] + tempRange[1]) / 2;
    msg.payload = tempValue;
    msg.topic = "temperature"; //overloaded topic to plot values as same line
    return msg;

} else {
    return;
}
```

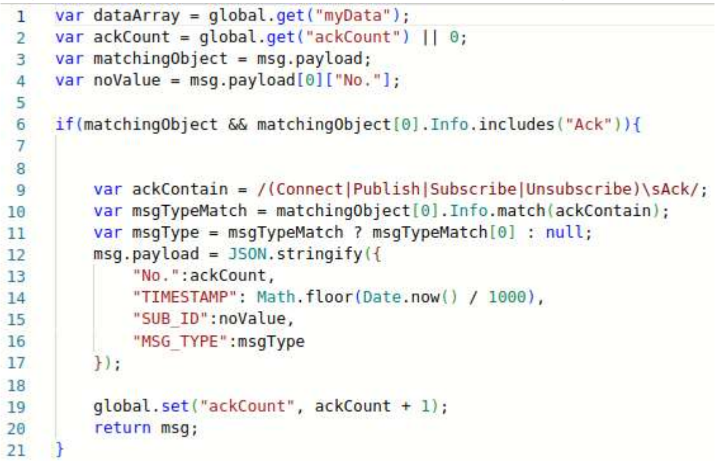


- **Filtered_pubs**: The code retrieves the current value of the global variable "***PubsCount***", defaulting to 1 if it hasn't been set yet. It then parses the incoming payload into a JSON object and extracts specific fields, assigning them to individual variables. Using these values, it builds a new JSON object that includes the extracted fields along with an incremented row number. Finally, it updates the " PubsCount " global variable and returns the modified message.



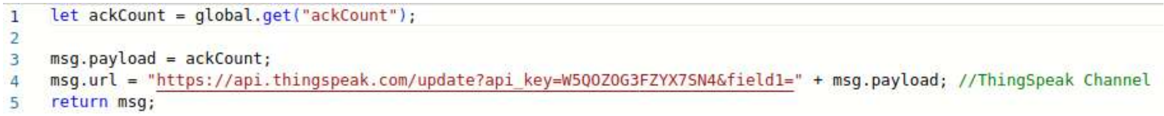
```javascript
let PubsCount = global.get("PubsCount") || 1;

var dataObj=msg.payload.payload;
var data = JSON.parse(dataObj);

var long = data.long;
var lat = data.lat;
let tempRange = data.range;
let tempValue = (tempRange[0] + tempRange[1]) / 2;
var type=data.type;
var unit = data.unit;
var description = data.description;

msg.payload = JSON.stringify({
    "No.": PubsCount,
    "LONG": long,
    "LAT":lat,
    "MEAN_VALUE":tempValue,
    "TYPE":type,
    "UNIT":unit,
    "DESCRIPTION":description,

});

PubsCount++;
global.set("PubsCount", PubsCount);

return msg;
```

- **Filtered_pubs.csv**: writes the file filterd_publish.csv
- **FilterAckMessage**: The code retrieves a global acknowledgment counter (ackCount). It checks if the incoming message contains an "Ack". If so, it extracts the type of acknowledged message (Connect, Publish, Subscribe, Unsubscribe), builds a new JSON object, increments the counter, updates the global state, and sends the new message forward.



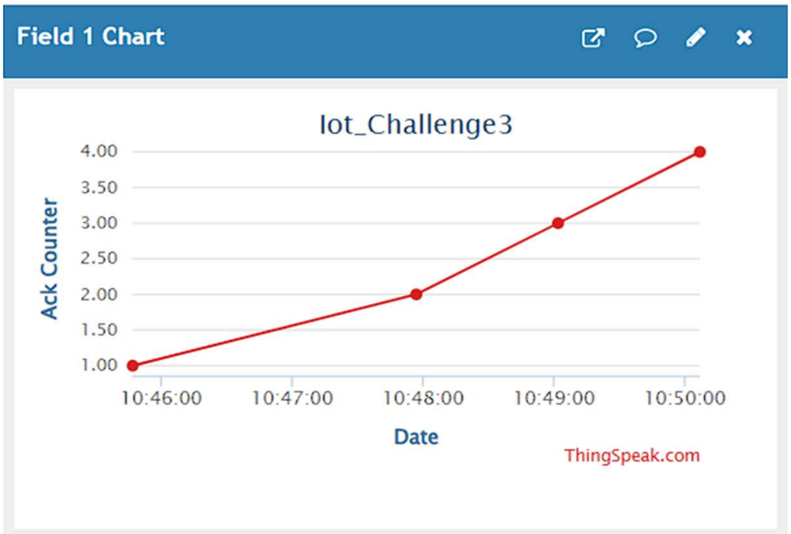
```
var dataArray = global.get("myData");
var ackCount = global.get("ackCount") || 0;
var matchingObject = msg.payload;
var noValue = msg.payload[0]["No."];

if(matchingObject && matchingObject[0].Info.includes("Ack")){


    var ackContain = /(Connect|Publish|Subscribe|Unsubscribe)\sAck/;
    var msgTypeMatch = matchingObject[0].Info.match(ackContain);
    var msgType = msgTypeMatch ? msgTypeMatch[0] : null;
    msg.payload = JSON.stringify({
        "No.":ackCount,
        "TIMESTAMP": Math.floor(Date.now() / 1000),
        "SUB_ID":noValue,
        "MSG_TYPE":msgType
    });

    global.set("ackCount", ackCount + 1);
    return msg;
}
```

- **Ack_log.csv**: write the file ack_log.csv
- **Sending AckCounter**: The code retrieves ackCount from the global state. It sets the message payload to ackCount, builds a ThingSpeak update URL including this value as field1, and returns the updated message.
  Thingspeak channel ID: https://thingspeak.mathworks.com/channels/2929937



```
let ackCount = global.get("ackCount");

msg.payload = ackCount;
msg.url = "https://api.thingspeak.com/update?api_key=W5Q0Z0G3FZYX7SN4&field1=" + msg.payload; //ThingSpeak Channel
return msg;
```

- **http request**: SEND the value of the global ACK counter to thingspeak channel through HTTP API


Field 1 Chart
Iot_Challenge3
Ack Counter
10:46:00
10:47:00
10:48:00
10:49:00
10:50:00
Date
ThingSpeak.com