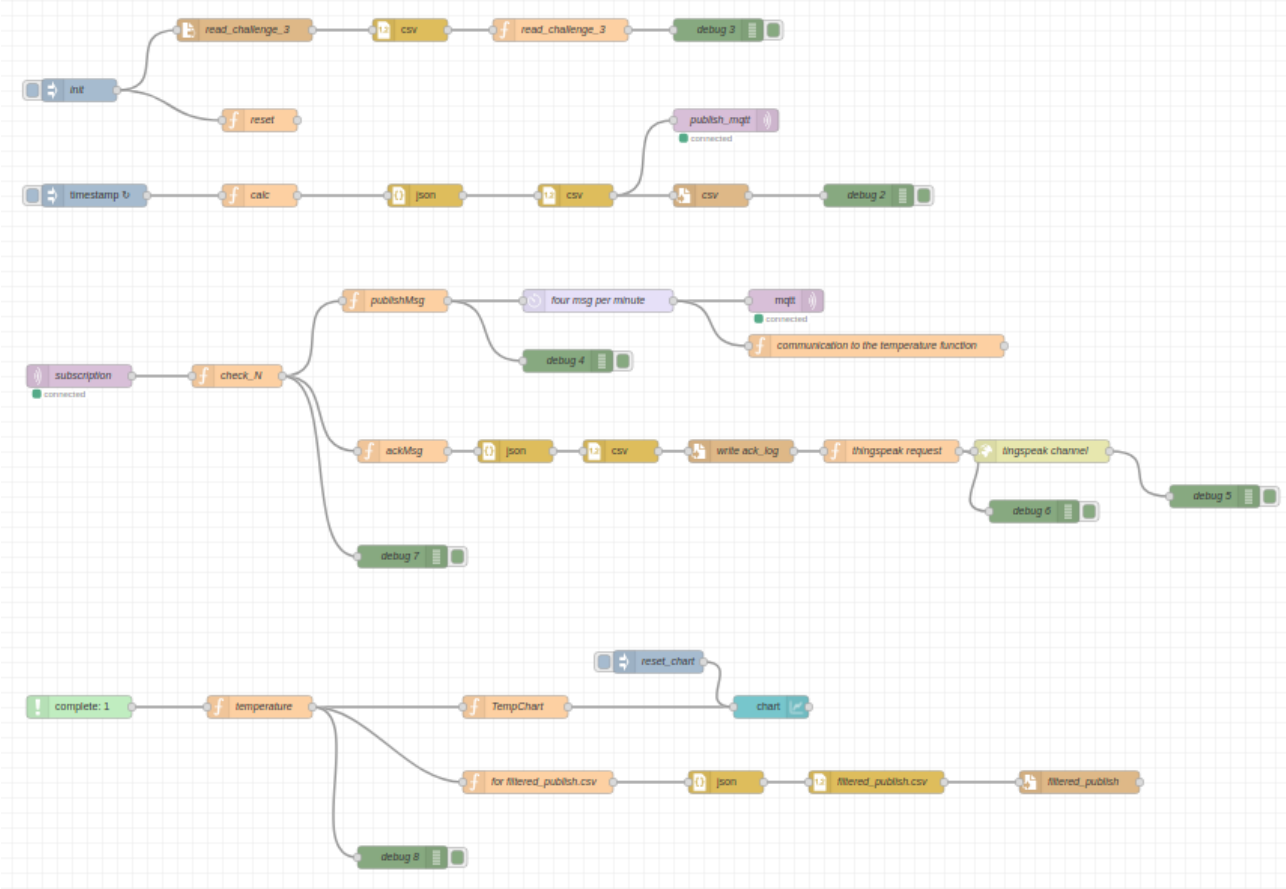
IoT Challenge 3
Challenge Part

Giovanni Ni 10831328 (Leader)
Xinyue Gu 10840236

Thingspeak channel ID: https://thingspeak.mathworks.com/channels/2931619

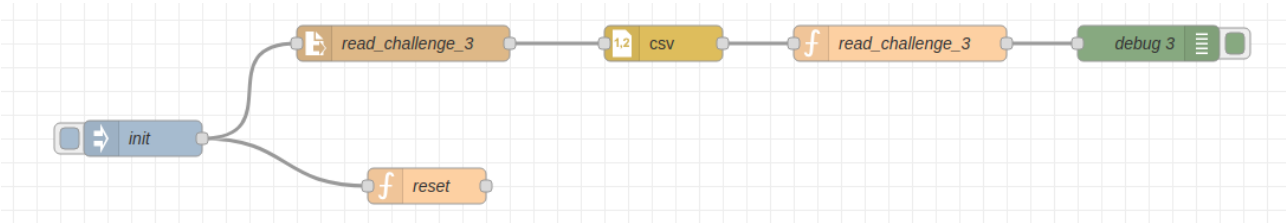
Node-red Flow



Note: all the reading and writing for csv files used local machine path.

Let's see now each branch:

Branch 1: INIZIALIZATION



Nodes:

- **Init**: Injection; pressing it will start the process of initialization.
- **read_challenge_3**: read the input file challenge3.csv
- **csv**: Parses the input CSV file.
- **read_challenge_3**: A function where we save all elements of the input file into a global variable. (Code reported below.)
- **reset**: A function that resets all global variables used in the flow. (Code reported below.)

Name read_challenge_3

Setup | On Start | **On Message** | On Stop

```
//store all rows in my global array for after
var dataArray = global.get("myData") || [];
dataArray.push(msg.payload);
global.set("myData", dataArray);

return msg;
```

Name reset

Setup | On Start | **On Message** | On Stop

```
//at every inizialization reset all global variables
global.set("Status", "RESET");
global.set("ackCount", 0);
global.set("rowNum", 1);
global.set("msgNum", 0);
global.set("rowNumTemp",0);
global.set("myData",[]);
global.set("message", null);

```

Branch2:

periodically publish MQTT messages to the local mosquitto broker (localhost, port 1884), to the topic challenge3/id_generator.

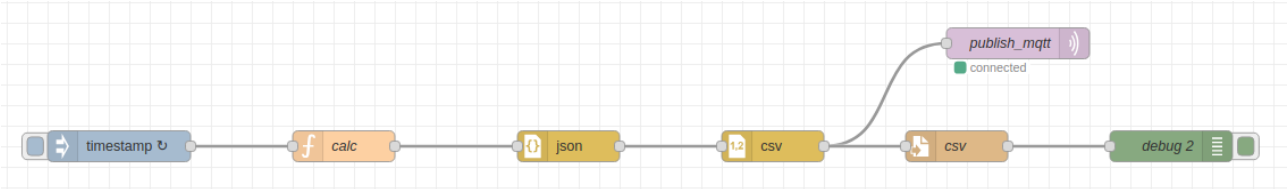
Messages should be sent with a rate of 1 message every 5 seconds.

Each message should contain, in the payload, a JSON-formatted string with:

- A random number (id) between 0 and 30,000, and
- The time the message was generated (UNIX timestamp).

Message payload example: {"id": 7781, "timestamp":1710930219}.

When sending the message, also save its field in a CSV (id_log.csv) with the form: No.,ID,TIMESTAMP where No. is the row number.



- **timestamp:** Triggers once every 5 seconds.
- **calc:** Function used to generate a random number (id) between 0 and 30,000, and the time when the message is generated. (Code reported below.)
- **json, csv:** Nodes used to prepare the writing process for the output file id_log.csv.
- **publish_mqtt:** Publishes MQTT messages to the local Mosquitto broker (localhost, port 1884) on the topic challenge3/id_generator.

Properties

Name: calc

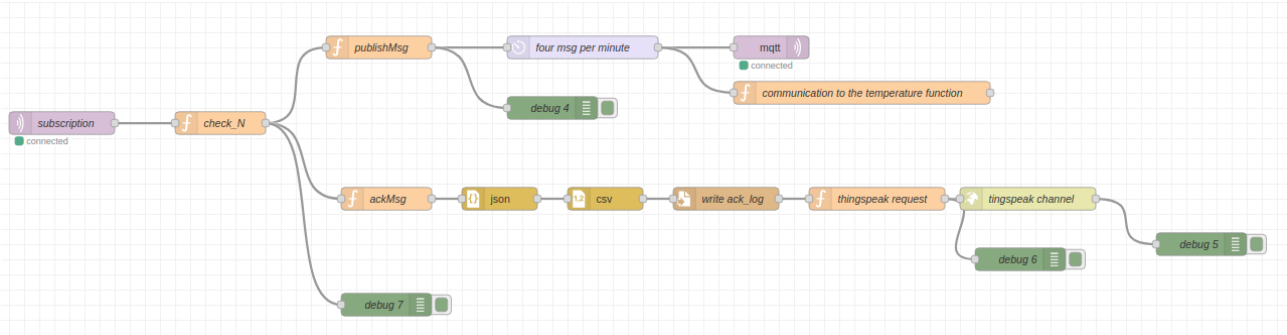
Setup | On Start | **On Message** | On Stop

```
//function for generating a random number
let rowNum = global.get("rowNum") || 1;

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

//prepare the format to publish on mqtt broker
msg.payload = JSON.stringify({
    "No.": rowNum,
    "ID": id,
    "TIMESTAMP": timestamp
});

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

return msg;
```

Each message randomly generates a number (id) between 0 and 30,000, records the generation time in UNIX timestamp format, and increments the row number.

Branch3:

Handles all behaviors after receiving different kinds of messages.



- **subscription:** Subscribes to the topic challenge3/id_generator on the local broker (localhost, port 1884).
- **check_N:** Function used to find the corresponding message inside the challenge3.csv file and to check how many messages have already been received. Detailed explanation below.
- **publishMsg:** Contains the logic for handling Publish messages. Detailed explanation below.
- **Four msg per minute:** A rate limiter node. It is set to allow four messages per minute (one message every 15 seconds). If too many Publish messages arrive, this node will enqueue the excess messages to maintain the rate limit and will publish them later (if total message number< =80)
- **mqtt:** MQTT Publish node that sends a Publish message to the local broker, using the same topic found in the MQTT Publish.
- **communication to the temperature function:** A function created for the next branch; it sets the global variable message with the newest MQTT published message. The next branch will read from this global variable.
- **ackMsg:** Contains the logic for handling ACK messages. Detailed explanation below.
- **json, csv, write ack_log:** Nodes used to write to the output file ack_log.csv.

- **thingspeak request:** Prepares the HTTP request to the ThingSpeak channel.
- **Thingspeak channel:** Sends the value of the global ACK counter to the ThingSpeak channel through the HTTP API.

Name: check_N

Setup | On Start | **On Message** | On Stop

```
//function to find out the corresponding message
//from challenge3.csv
let msgNum = global.get("msgNum");

if(msgNum >= 80) {
    global.set("Status", "DONE");
    node.warn("STOP WORKING")
    return;
}

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

global.set("SUB_ID", id);

var dataArray = global.get("myData");

var desiredID = id % 7711;


var message = dataArray.filter(function(item) {
    return parseInt(item["No."]) === desiredID;
});

//node.warn(message[0].Info);

msg.payload = message;

if (!message[0].Info.includes("Publish Message")) {
    msgNum++;
    global.set("msgNum", msgNum);
}

return msg;
```

It checks the status of the flow: if the node has not received more than 80 messages, it will compute the N value.

Specifically, it splits the string msg.payload by commas, storing the resulting parts in an array called parts. After extracting the id value, it searches for the message with the same ID in myData, which contains the information from the CSV file.

Generally, msgNum is incremented directly by 1. However, if there are Publish messages, the increment process is handled in the next node, because a single message could contain multiple Publish messages.

Name publishMsg

Setup | On Start | **On Message** | On Stop

```
//in case of publish message
if(global.get("Status") === "DONE") {
    return;
}

let msgNum = global.get("msgNum");
var message = msg.payload;
var dataArray = global.get("myData");
var sub_id = global.get("SUB_ID");

if(message && message[0].Info.includes("Publish Message")) {
    var topics = message[0].Info.match(/\[([^\]]+)\]/g);
    var payloads;
    if(message[0].Payload == null) {
        payloads= "";
    } else {
        payloads = message[0].Payload.match(/\{[^{}]+\}/g)||"";
    }

```

Name publishMsg

Setup | On Start | **On Message** | On Stop

```
    }

    topics.forEach(function(topic, i) {
        var publish = {
            payload: {
                "timestamp": Math.floor(Date.now() / 1000),
                "topic": topic,
                "id": global.get("SUB_ID"),
                "payload": payloads[i]
            },
            topic: topic
        };
        //node.warn(publish);
        msgNum++;
        global.set("msgNum", msgNum);
        node.send(publish);
    });
} else {
    global.set("message", null);
    return null;
}
```

First, it starts by checking the flow status.

If it is a Publish message, it extracts the payloads and topics and sends them to the next node.

Attention: If the message contains multiple topics, it iterates through them using forEach, and uses node.send() to send each one separately.

During the forEach loop, it also increments the total number of messages. This means that it could happen that the ID generator produces fewer than 80 IDs, but the flow stops working because some messages contain multiple Publish messages.

Name communication to the temperature function

Setup | On Start | **On Message** | On Stop

```
if (global.get("Status") === "DONE") {
    node.warn("STOP WORKING");
    return;
}

//for every mqtt publish message, update the global variable, used by "temperature" function
global.set("message", msg);
```

The next branch will read the published temperature message from the global variable message (after each MQTT publish).

Name ackMsg

Setup | On Start | **On Message** | On Stop

```
if(global.get("Status") === "DONE") {
    return;
}

//in case of ack message
var dataArray = global.get("myData");
var ackCount = global.get("ackCount");
var message = msg.payload;
var sub_id = global.get("SUB_ID");
```

Name ackMsg

Setup | On Start | **On Message** | On Stop

```

if(message && message[0].Info.includes("Ack")) {
    ackCount++;
    var ackRegex = /(Connect|Publish|Subscribe|Unsubscribe)\sAck/;
    var msgTypeMatch = message[0].Info.match(ackRegex);
    var msgType =msgTypeMatch ? msgTypeMatch[0] : null;

    msg.payload = JSON.stringify({
        "No.": ackCount,
        "TIMESTAMP" : Math.floor(Date.now() / 1000),
        "SUB_ID": sub_id,
        "MSG_TYPE": msgType
    });

    global.set("ackCount", ackCount);
    return msg;
} else {
    return null;
}
```

This function checks if a global variable named Status is set to "DONE".
If it is, the function stops execution. Otherwise, it retrieves data from another global variable named myData. It also retrieves the current value of the global variable ackCount.
Then, it examines the payload received in the message. If the payload contains an object and the value of its Info property includes the string "Ack", it enters the if block and starts preparing the information for the next nodes.
It uses a regular expression to extract the type of the acknowledgment message from the Info property.

Next, it creates a JSON object containing the current timestamp, the message SUB_ID, and the acknowledgment message type.
This JSON object is added to the message payload.
Finally, it increments the ackCount variable by 1 (this updated value will be sent to the ThingSpeak channel) and returns the modified message.

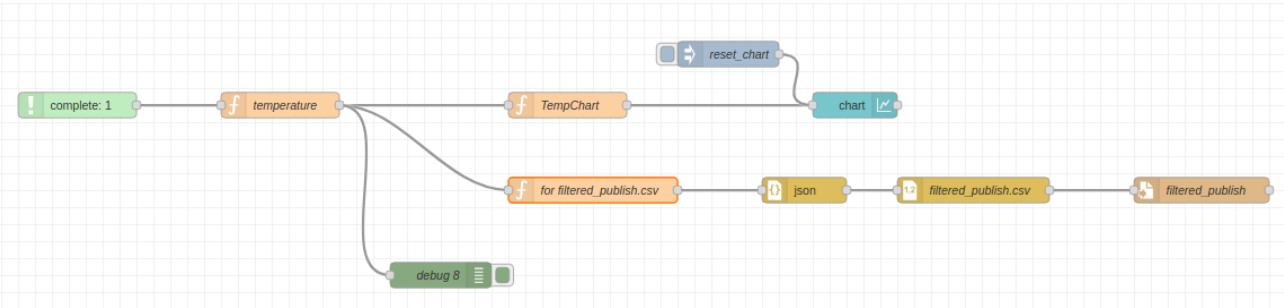
Name: thingspeak request

Setup | On Start | **On Message** | On Stop

```
msg.method = "GET";

var API_KEY = "Q6031M04L4LVMMVM";
var counter = global.get("ackCount");

msg.url = "https://api.thingspeak.com/update?api_key="+API_KEY+"&field1="+counter;

return msg;
```

It retrieves the current ackCount value and then sends it to the ThingSpeak channel.
Note: There could be a potential issue: since the ThingSpeak channel cannot receive messages too frequently, sometimes the HTTP request might fail.
As a result, the channel chart may show jumps — for example, going directly from 2 ACKs to 4 ACKs, skipping 3 ACKs — because when attempting to send the 3 ACKs update, the second request was sent too soon after the first.

Branch4:
After publishing the Publish message, if the payload contains a temperature in Fahrenheit (i.e., it has the attributes Type=Temperature and Unit=F), take this message and plot its value in a Node-RED chart.
At the same time, save the payload of these messages (only those with temperature in Fahrenheit) into a CSV file (filtered_pubs.csv), with one message payload per row.
filtered_publish.csv format:
No., LONG, LAT, MEAN_VALUE, TYPE, UNIT, DESCRIPTION
This branch explains how Fahrenheit temperature messages are processed.



- **Complete:** Monitors the MQTT node from Branch 3. It triggers after each Publish message from that node.
- **Temperature:** Function that checks if the Publish message contains a Fahrenheit temperature. If so, it forwards the message to the next nodes. (Code below.)
- **TempChart:** Function that forwards the message to the chart. To keep the chart cleaner, we decided to use a single topic for all Publish messages.

- **for filtered_publish.csv:** Function that prepares the message to be saved into the output file filtered_publish.csv.
- **chart:** Creates a Node-RED chart displaying all Fahrenheit temperatures (after the MQTT Publish step in the previous branch).
- **reset_chart:** Node used to reset the chart whenever we want to reinitialize it.
- **json, csv, filtered_publish:** Nodes that write the output to the file filtered_publish.csv.

Name: temperature

Setup | On Start | **On Message** | On Stop

```
if(global.get("Status") === "DONE") {
    node.warn("STOP WORKING")
    return;
}

//after every mqtt publish message, control if there is a F temperature
var message = global.get("message");
var data;

//node.warn(message);

if(message) {
    if(message.payload.payload != null) {
        data = JSON.parse(message.payload.payload);
    } else {
        data = "";
    }

```

Name: temperature

Setup | On Start | **On Message** | On Stop

```
//node.warn(message);

if(message) {
    if(message.payload.payload != null) {
        data = JSON.parse(message.payload.payload);
    } else {
        data = "";
    }

    //node.warn(data);

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

First, it checks if the flow has already terminated.
If not, it verifies whether the message is a Publish message containing a Fahrenheit temperature.
If so, it forwards the message; otherwise, it does nothing.

Name TempChart

Setup | On Start | **On Message** | On Stop

```
//creating a msg with the temp mean value for the ui chart
var data = msg.payload;

if(data) {
    let tempRange = data.range;
    let temp = (tempRange[0] + tempRange[1])/2;

    //we are setting a unic topic for all tmp received
    //otherwise the chart is not very clean
    msg.topic = " ";
    msg.payload = temp;
    return msg;

} else {
    return;
}
```

Prepares the temperature information for the chart, setting the topic as an empty string ("") for all messages to keep the chart clean.
The temperature value reported is the mean value.

Name for filtered_publish.csv

Setup | On Start | **On Message** | On Stop

```
//for each F temperature read, add that also in filtered_publish.csv
let rowNum = global.get("rowNumTemp");

var data = msg.payload;

var long = data.long;
var meanValue = (data.range[0] + data.range[1])/2;
var lat = data.lat;
var type = data.type;
var unit = data.unit;
var descr = data.description;
```

Name: for filtered_publish.csv

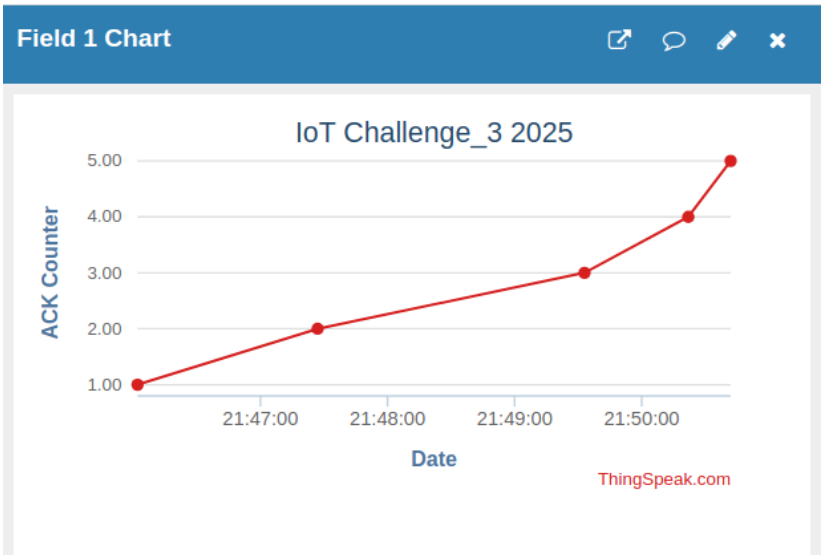
Setup | On Start | **On Message** | On Stop

```
var descr = data.description;

rowNum++;
msg.payload = JSON.stringify ( {
    "No.": rowNum,
    "LONG": long,
    "LAT": lat,
    "MEAN_VALUE": meanValue,
    "TYPE": type,
    "UNIT": unit,
    "DESCRIPTION": descr

});

global.set("rowNumTemp", rowNum);

return msg;
```

Prepares the message to be written into the output file filtered_publish.csv.

The two chart created:

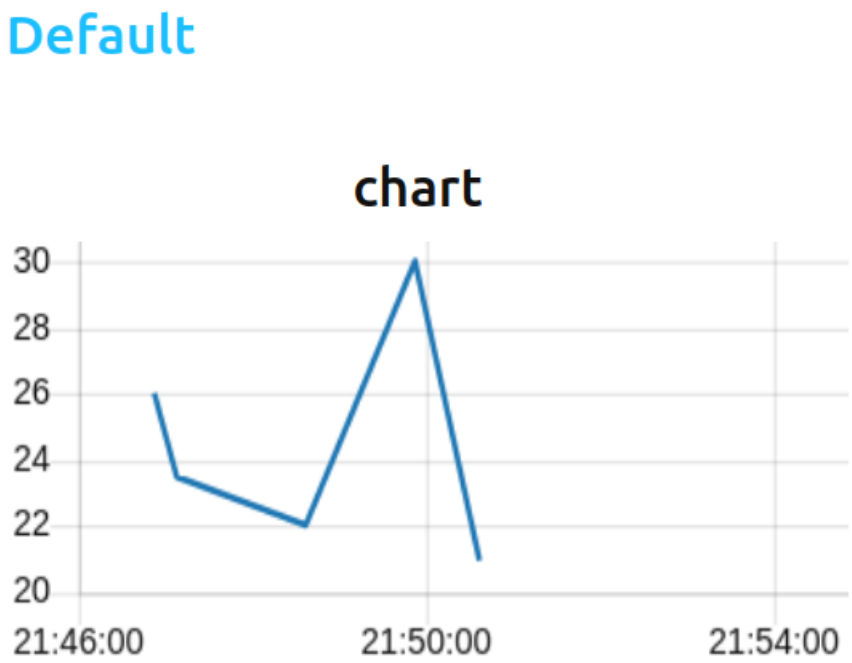
- ACK Counter



Correspond to the ack_log.csv

A	B	C	D	E
1	1745783161	15515	Subscribe Ack	
2	1745783246	348	Connect Ack	
3	1745783372	189	Subscribe Ack	
4	1745783422	29361	Publish Ack	
5	1745783442	23224	Subscribe Ack	

Node-red chart: Fahrenheit temperature.



Correspond to the filtered_pubs.csv:

A	B	C	D	E	F	G	H
1	75	74	26	temperat	F	Room Temperature	
2	94	97	23.5	temperat	F	Room Temperature	
3	54	64	22	temperat	F	Room Temperature	
4	51	91	30	temperat	F	Room Temperature	
5	77	77	21	temperat	F	Room Temperature	

<u>Final consideration</u>:

We observe that id_log.csv contains more than 80 IDs. This happens because an ID is generated every 5 seconds and the process does not automatically stop. However, the remaining branches actually stop working after around 65 IDs(in this specific case) are generated. We can confirm this because we added a node.warn("STOP WORKING") message in some branches when the status becomes "DONE". Some messages contain multiple publish messages, which leads to around 80 messages being processed in total. This behavior is exactly as expected.