POLITECNICO
MILANO 1863

Internet of Things
Challenge 3

Andrea Pesciotti
10715428

Teachers:
Redondi Alessandro Enrico Cesare
Fabio Palmese
Boiano Antonio

April 2025

1 MQTT Publisher and CSV Logger

1.1 Task Summary

The first task was to create a Node-RED flow that acts as an MQTT publisher. This flow needed to periodically generate messages containing a random ID and a timestamp, format them as JSON strings, and publish them to a specific topic on a local Mosquitto broker. Simultaneously, the generated ID, timestamp, and an incremental row number needed to be logged to a local CSV file named `id_log.csv`.

1.2 Nodes Used

The following Node-RED nodes were primarily used for this step:

- `Inject` node: To trigger the flow periodically.
- `Function` node ("Generate ID & Timestamp"): To create the random ID, get the timestamp, and manage the CSV row counter.
- `Change` node ("Format JSON String"): To convert the data object into a JSON string for MQTT.
- `MQTT Out` node ("Publish ID/Timestamp"): To publish the message to the MQTT broker.
- `Function` node ("Format CSV"): To format the data specifically for the CSV log file.
- `File` node ("Append to id_log"): To append the formatted string to the CSV file.

1.3 Node Configuration

1.3.1 Inject Node (”5 seconds”)

This node was configured to trigger the flow automatically every 5 seconds using the ‘interval‘ repeat mode.

1.3.2 Generate ID & Timestamp (Function Node)

This node generates the core data. It calculates a random ID between 0 and 30000, gets the current UNIX timestamp, retrieves and increments a row counter stored in flow context (`flow.csvRowCount`), and prepares the message object. It stores raw values for CSV logging (`msg.raw_id`, `msg.raw_timestamp`, `msg.rowNum`) and sets `msg.payload` to a JavaScript object containing the ID and timestamp for the MQTT path.

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

// Store the raw values on the msg object for later use (CSV)
msg.raw_id = randomId;
msg.raw_timestamp = timestamp;

// Create the payload object for MQTT path
msg.payload = {
    id: randomId,
    timestamp: timestamp
};

// Get and increment the row counter using flow context
let rowNum = flow.get("csvRowCount") || 0;
rowNum++;
flow.set("csvRowCount", rowNum);

// Store the row number for the CSV node path
msg.rowNum = rowNum;

return msg;
```

1.3.3 MQTT Publishing Path

- **Format JSON String (Change Node):** Converts the `msg.payload` object to a JSON string using the JSONata expression `JSON.stringify(payload)`.
- **Publish ID/Timestamp (MQTT Out Node):** Connects to `localhost:1884` and publishes the JSON string payload to the topic `challenge3/id_generator`. QoS 0, Retain false.

1.3.4 CSV Logging Path

- **Format CSV (Function Node):** This node takes the message branched from the ”Generate ID & Timestamp” node. It extracts the raw ID, times-

tamp, and row number, then formats them into the required CSV string. Simple string concatenation was used for reliability.

```
// Retrieve the values stored earlier by the Generate node
const rowNum = msg.rowNum;
const id = msg.raw_id;
const ts = msg.raw_timestamp;

// Check if values are valid (basic check)
if (typeof rowNum === 'undefined' || typeof id === 'undefined'
    || typeof ts === 'undefined') {
     node.error("Missing data for CSV formatting", msg);
     return null;
}

// Format the string for the CSV row using concatenation,
    including a newline character
// Format: No.,ID,TIMESTAMP
msg.payload = rowNum + "," + id + "," + ts + "\n";

return msg;
```

- **Append to id_log (File Node):** Configured to append data to the file named `id_log.csv`. The 'Action' was set to `Append to file` to ensure that all generated records during a single run are collected in the log. The 'Add newline' option was unchecked (as the newline was added in the function node). *Note: To get a clean log for each run, the file should be manually reset (e.g., using `echo "Header" > file.csv`) before starting the Node-RED flow.*

1.3.5 Flow Diagram Segment

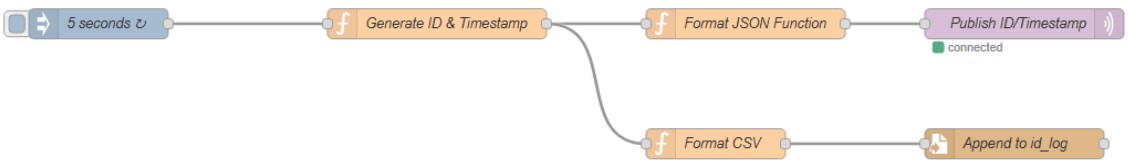

Figure 1: Complete Node-RED Flow Diagram. Step 1 includes the top-left Inject node and the two branches originating from "Generate ID & Timestamp".

2 MQTT Subscriber and CSV Lookup

2.1 Task Summary

The second step involved creating a new branch in the Node-RED flow to act as an MQTT subscriber. This branch listens for the messages published in Step 1 on the `challenge3/id_generator` topic. Upon receiving a message, it parses the JSON payload, extracts the `id`, and calculates a frame number `N` using the formula $N = \text{id} \pmod{7711}$. Finally, it reads and parses an external CSV file (`challenge3.csv`) to find and extract the specific row where the frame number in the 'No.' column matches the calculated value `N`.

2.2 Nodes Used

This part of the flow primarily utilized the following nodes:

- `MQTT In` node ("Subscribe ID/Timestamp"): To subscribe to the topic and receive messages.
- `Function` node ("Calculate N"): To parse the incoming message and calculate N.
- `File In` node ("Read challenge3.csv"): To read the external CSV file content.
- `CSV` node ("Parse CSV"): To parse the CSV text into usable data.
- `Function` node ("Find Row by N"): To search the parsed data for the matching row.

2.3 Node Configuration

2.3.1 Subscribe ID/Timestamp (MQTT In Node)

This node connects to the same local Mosquitto broker (`localhost:1884`) as the publisher. It subscribes to the topic `challenge3/id_generator`. Crucially, the 'Output' was configured as a parsed JSON object, ensuring the incoming JSON string payload is automatically converted into a JavaScript object for easier processing in subsequent nodes.

2.3.2 Calculate N (Function Node)

This node receives the parsed message object from the `MQTT In` node. It extracts the `id` field, calculates $N = \text{id} \pmod{7711}$ using the JavaScript modulo operator (`%`), and stores both the calculated `N` (as `msg.targetN`) and the original received `id` (as `msg.original_id`) onto the message object for use in later steps.

```
// msg.payload is already a JSON object
const id = msg.payload.id;
const divisor = 7711;

if (typeof id === 'number') {
    const N = id % divisor ;
    msg.targetN = N;
    msg.original_id = id;

    return msg;
} else {
    node.error("Received payload did not contain a numeric ID.",
    msg);
    return null; // Stop the flow if ID is invalid
}
```

2.3.3 CSV Reading and Parsing

- **Read challenge3.csv (File In Node):** This node is triggered by the message coming from "Calculate N". It reads the entire content of the specified CSV file (`challenge3.csv`). It outputs the file content as `a single utf8 string` into `msg.payload`, overwriting previous payload content.
- **Parse CSV (CSV Node):** This node takes the CSV string from the previous node's `msg.payload`. It was configured with:
  - `First row contains column names`: Checked, to use headers like 'No.', 'Time', etc.
  - `Parse numerical values`: Checked, to help treat numbers as numbers.
  - `Output`: Set to `an array of objects`. This ensures the entire CSV content is passed as a single array in `msg.payload` to the next node, which is essential for searching.

2.3.4 Find Row by N (Function Node)

This node receives the array of CSV row objects in `msg.payload` and the target row number `N` in `msg.targetN` (passed along from the "Calculate N" node). It searches the array to find the object where the `'No.'` property matches `targetN`. It performs a numerical comparison for robustness. If a row is found, the original subscription ID (`msg.original_id`) is attached to the found row object, and this object becomes the new `msg.payload`. If not found, an error object (including the original ID) is set as the payload.

```
const csvData = msg.payload; // The array of rows from CSV node
const targetN = msg.targetN; // Get N calculated earlier
const original_id = msg.original_id; // Get original ID passed
    along

// Basic validation
if (!Array.isArray(csvData)) {
    node.error("CSV data is not an array.", msg);
    return null;
}
if (typeof targetN !== 'number') {
    node.error("Target N is not a number.", msg);
    return null;
}
if (typeof original_id === 'undefined') {
    node.warn("Original ID is missing entering Find Row", msg);
}

// Find the row where the 'No.' property equals targetN
const foundRow = csvData.find(row => {
  // Check if row and 'No.' property exist before comparing
  if (row && typeof row['No.'] !== 'undefined') {
      let rowNum = Number(row['No.']);
      let targetNum = Number(targetN);
      return rowNum === targetNum;
  }
  return false;
});

if (foundRow) {
    // Attach the original subscription ID to the found data object
    foundRow.original_id = original_id;
    msg.payload = foundRow; // Set the enhanced row object as the
    final payload
} else {
    // Set payload to an error object, include N and original_id
    for context
    msg.payload = { error: "Row not found for N=" + targetN, N:
    targetN, original_id: original_id };
}
return msg;
```

2.3.5 Flow Diagram Segment

Figure 2: Node-RED flow segment for Step 2: MQTT Subscription and CSV Lookup. This shows the path from "Subscribe ID/Timestamp" through "Calculate N", "Read challenge3.csv", "Parse CSV", and "Find Row by N".

3 Conditional MQTT Re-Publishing (Rate-Limited)

3.1 Task Summary

This step extends the subscriber branch. After finding the relevant row (`N`) in `challenge3.csv` (from Step 2), this logic checks if the found row's 'Info' field indicates one or more MQTT "Publish Message" commands. If it does, for each publish command identified within the row, it extracts its specific topic and payload string. A new JSON message is constructed containing the current timestamp, the original subscription ID (`SUB_ID`), the extracted topic, and the extracted payload string. These newly constructed messages are then rate-limited to a maximum of 4 per minute before being published to the MQTT broker on the dynamically extracted topic. This step also handles potentially multiple commands within a single CSV row and cases where the payload might be missing or malformed, treating such payloads as empty in the re-published message.

3.2 Nodes Used

The primary nodes involved in implementing this specific functionality are:

- `Function` node ("Process Row Data"): Central node performing checks, parsing, data extraction, and routing (utilizing its Output 2 for this path).
- `Delay` node ("Limit 4/min"): To enforce the message rate limit.
- `Function` node ("Stringify Final Payload"): To convert the prepared message object into a JSON string.
- `MQTT Out` node ("Publish CSV Data"): To publish the final message to the dynamic topic.

3.3 Node Configuration

3.3.1 Process Row Data (Function Node)

This node receives the found CSV row object (including `original_id`) from the "Find Row by N" node. It performs several crucial tasks:

1. Checks if the `Info` field contains "Publish Message".
2. Parses the `Info` field to extract potentially multiple destination topics.
3. Attempts to parse the `Payload` field, handling single JSON objects, multiple concatenated JSON objects ({...},{...}), and parsing errors.
4. For each valid topic/payload pair found, if the original row contained "Publish Message", it constructs a standard message object containing the current timestamp, the original subscription ID, the extracted topic, and the extracted payload string (or an empty string if the original payload was invalid/missing).
5. It sets the dynamic `msg.topic` property for the outgoing MQTT message.
6. It sends messages destined for re-publishing out via its **second output port**.

The code handles cases with multiple commands per row by potentially sending multiple distinct messages to the second output port.

```
// Input msg.payload is the row object from CSV, including
    original_id
const rowData = msg.payload;
const original_id = msg.original_id || rowData.original_id;

let tempFOutputs = []; // For path 1: Temp(F) data
let republishOutputs = []; // For path 2: Republish data

const infoString = rowData.Info || "";
const containsPublish = infoString.includes("Publish Message");

// Extract all topics
const topicRegex = /\[(.*?)\]/g;
let topics = [];
let match;
while ((match = topicRegex.exec(infoString)) !== null) {
    topics.push(match[1]);
}

// Process Payload(s)
const payloadString = rowData.Payload || "";
let payloadObjects = [];

if (payloadString.startsWith("{") && payloadString.endsWith("}") &&
    payloadString.includes("},{")) {
    try {
         let potentialJSONs = payloadString.split(/}\s*,\s*{/);
         for (let i = 0; i < potentialJSONs.length; i++) {
             let jsonStr = potentialJSONs[i];
```

```
            // Add potentially missing braces back after split
            if (i === 0 && !jsonStr.startsWith("{")) jsonStr = "{" + jsonStr;
            if (i === potentialJSONs.length - 1 && !jsonStr.endsWith("}")) jsonStr = jsonStr + "}";
            if (i > 0 && !jsonStr.startsWith("{")) jsonStr = "{" + jsonStr;
            if (i < potentialJSONs.length - 1 && !jsonStr.endsWith("}")) jsonStr = jsonStr + "}";
            try {
                payloadObjects.push({ raw: jsonStr, parsed: JSON.parse(jsonStr) });
            } catch(eP) {
                node.warn(`Failed to parse individual JSON part: ${eP.message} in row ${rowData.No}`);
                payloadObjects.push({ raw: jsonStr, parsed: null });
            }
        }
    } catch(eS) {
        node.warn(`Failed during split/parse of multi-JSON payload: ${eS.message} in row ${rowData.No}`);
        // Attempt single parse as fallback
        try { payloadObjects.push({ raw: payloadString, parsed: JSON.parse(payloadString) }); }
        catch (eS2) { payloadObjects.push({ raw: payloadString, parsed: null }); }
    }
} else if (payloadString.startsWith("{") && payloadString.endsWith("}")) {
    try { payloadObjects.push({ raw: payloadString, parsed: JSON.parse(payloadString) }); }
    catch (e1) {
        node.warn(`Failed to parse single JSON payload: ${e1.message} in row ${rowData.No}`);
        payloadObjects.push({ raw: payloadString, parsed: null });
    }
} else if (payloadString) {
    // Non-JSON payload, store raw string, parsed is null
    payloadObjects.push({ raw: payloadString, parsed: null });
}

// Process each Topic/Payload pair
let currentTimestamp = Math.floor(Date.now() / 1000);
let maxPairs = Math.min(topics.length, payloadObjects.length);
if (topics.length !== payloadObjects.length) {
    node.warn(`Topic/Payload count mismatch in row ${rowData.No}. Processing ${maxPairs} pair(s).`);
}

for (let i = 0; i < maxPairs; i++) {
    let currentTopic = topics[i];
    let currentPayloadStr = payloadObjects[i].raw;
    let parsedPayload = payloadObjects[i].parsed; // null if parse failed or non-JSON

```

```
    // 1. Check for Temp (F) for Output 1
    if (parsedPayload && parsedPayload.type === "temperature" &&
parsedPayload.unit === "F") {
        let meanValue = null;
        if (Array.isArray(parsedPayload.range) && parsedPayload.
range.length === 2) {
            let min = Number(parsedPayload.range[0]); let max =
Number(parsedPayload.range[1]);
            if (!isNaN(min) && !isNaN(max)) { meanValue = (min +
max) / 2; }
        }
        if (typeof meanValue === 'number') {
            tempFOutputs.push({ payload: { meanValue: meanValue,
long: parsedPayload.long, lat: parsedPayload.lat, type:
parsedPayload.type, unit: parsedPayload.unit, description:
parsedPayload.description } });
        }
    }

    // 2. Prepare data for Republish Path (Output 2)
     if (containsPublish) { // Only if the original row was a
publish type
         if (typeof original_id === 'undefined') { node.error(`
Original SUB_ID missing for republish in row ${rowData.No}`); }
         else {
            let payloadForRepublish = parsedPayload ?
currentPayloadStr : "";

            republishOutputs.push({
                payload: {
                    timestamp: currentTimestamp,
                    id: original_id,
                    topic: currentTopic,
                    payload: payloadForRepublish
                },
                topic: currentTopic
            });
        }
    }
}

// Send messages arrays to corresponding outputs
// Output 1: Temp(F) data; Output 2: Republish data
return [ tempFOutputs, republishOutputs ];
```

3.3.2 Limit 4/min (Delay Node)

This node receives messages from the second output of "Process Row Data". It is configured with:

- `Action`: `Rate Limit`.
- `Rate`: 4 message(s).
- `per`: `1 minute`.
- `Drop intermediate messages`: Checked.

This ensures that a maximum of 4 messages per minute proceed to the final publishing step, regardless of how many arrive from the processing node.

3.3.3 Stringify Final Payload (Function Node)

This node takes the prepared message object (containing timestamp, id, topic, payload) from the rate limiter and converts its `msg.payload` into a JSON string required for MQTT publication.

```
try {
    // Convert the payload object into a JSON string
    msg.payload = JSON.stringify(msg.payload);
} catch (e) {
    node.error("Failed to stringify final payload: " + e.message,
    msg);
    msg.payload = undefined; // Avoid sending invalid data
}
return msg;
```

3.3.4 Publish CSV Data (MQTT Out Node)

This is the final node in this path. It connects to the Mosquitto broker at localhost:1884. Crucially, its `Topic` field is left **blank**. This allows it to use the topic set dynamically in `msg.topic` by the "Process Row Data" node. It publishes the final JSON string received in `msg.payload`. QoS was set to 0 and Retain to false.

3.3.5 Flow Diagram Segment

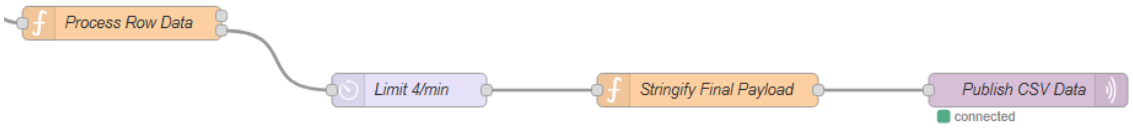

Figure 3: Complete Node-RED Flow Diagram. Step 3 corresponds to the path starting from the second output of "Process Row Data", through "Limit 4/min", "Stringify Final Payload", and ending at "Publish CSV Data".

4 Fahrenheit Temperature Plotting and Logging

4.1 Task Summary

Building upon the previous steps, this stage focuses on handling specific data found within the payloads of identified "Publish Message" rows from `challenge3.csv`. If a parsed payload indicates a temperature reading specifically in Fahrenheit (checking for 'Type=Temperature' and 'Unit=F'), two actions are performed:

1. The mean temperature value, calculated from the 'range' attribute $[min, max]$ as $(min + max)/2$, is plotted on a Node-RED Dashboard chart.
2. Detailed information about this specific Fahrenheit temperature reading (including an incremental row number, longitude, latitude, the calculated mean value, type, unit, and description) is appended to a separate log file, filtered_pubs.csv.

This processing happens in parallel for each Fahrenheit temperature payload identified, even if multiple occurred within the same original CSV row.

4.2 Nodes Used

This functionality relies on the following nodes, branching off from the "Process Row Data" node described in Step 3:

- `Function` node ("Process Row Data"): Its Output 1 initiates this path, sending only messages pre-identified as containing relevant Temp(F) data.
- `Function` node ("Prep Chart/CSV"): Extracts data fields, calculates the CSV row number, and formats outputs for the chart and the CSV file.
- `Chart` node (`ui_chart`) ("Avg Temp F"): Displays the mean temperature value on the dashboard.
- `Change` node ("Set Payload to CSV Row"): Prepares the message payload specifically for the CSV file writing node.
- `File` node ("Save Filtered Pub"): Appends the formatted data row to filtered_pubs.csv.

4.3 Node Configuration

4.3.1 Process Row Data (Function Node) - Output 1

As detailed in Step 3, the "Process Row Data" node internally checks each parsed payload from the CSV. If a payload matches 'type: "temperature"' and 'unit: "F"', it calculates the mean value and prepares an object containing ' meanValue, long, lat, type, unit, description '. This object is set as the `msg.payload` in a new message sent exclusively out of the node's **first output port**.

4.3.2 Prep Chart/CSV (Function Node)

This node receives the message object containing the filtered Fahrenheit temperature data via its input. It performs the final preparations for both the chart and the CSV log file. It retrieves/increments a dedicated counter (`flow.filteredPubsCounter`) for the `filtered_pubs.csv` file, extracts all necessary data fields from the incoming `msg.payload`, sets `msg.payload` to contain only the numeric `meanValue` (for the chart node), and creates the fully formatted CSV row string, storing it in `msg.csvRow`.

```
// Input msg.payload is like:
let data = msg.payload;

// Basic check for valid input data object
if (!data || typeof data !== 'object') {
     node.error("Prep Chart/CSV received invalid data object in msg.payload", msg);
     return null; // Stop flow if input is bad
}

// Extract mean value
let meanValue = data.meanValue;
if (typeof meanValue !== 'number' || isNaN(meanValue)) {
    node.warn("Invalid or missing meanValue in input data for Chart/CSV: " + meanValue, msg);
    return null; // Stop if invalid
}

// Set msg.payload to the numeric meanValue for the chart node
msg.payload = meanValue;

let rowNum = flow.get("filteredPubsCounter") || 0;
rowNum++;
flow.set("filteredPubsCounter", rowNum);

// Extract other fields for the CSV row from the input data object
let longitude = data.long || '';
let latitude = data.lat || '';
let type = data.type || '';
let unit = data.unit || '';
let description = data.description || '';

// Handle potential quotes within the description for CSV safety
```

```
let quotedDescription = '"' + String(description).replace(/"/g, '""') + '"';

// Construct the CSV row string using simple '+' concatenation
msg.csvRow = rowNum + "," + longitude + "," + latitude + "," + meanValue + "," + type + "," + unit + "," + quotedDescription + "\n";

return msg;
```

4.3.3 Avg Temp F (ui_chart Node)

This node provides the visual output on the Node-RED Dashboard. It receives the message from "Prep Chart/CSV" and automatically uses the numeric value in `msg.payload` (which was set to the 'meanValue') as the data point to plot.

- It was assigned to a specific `Group` and `Tab` within the dashboard configuration.
- The chart `Type` was set to `Line chart`.
- A descriptive `Label`, such as `Avg Temp (F)`, was assigned.
- The `X-axis` was configured to display a suitable time window (e.g., last 10 minutes).

The resulting chart can be viewed by accessing the Node-RED Dashboard UI, typically at `http://<node-red-ip>:1880/ui`.

4.3.4 CSV Logging Path (Change & File Nodes)

This parallel path ensures the Temp(F) data is logged to its dedicated file.

- **Set Payload to CSV Row (Change Node):** This node is placed after "Prep Chart/CSV". Its sole purpose is to prepare the message for the `File` node. It takes the formatted CSV string stored in `msg.csvRow` and moves it into `msg.payload`, overwriting the numeric value that was previously there. The rule is: Set `msg.payload` to the value of `msg.csvRow`.
- **Save Filtered Pub (File Node):** This node receives the message from the preceding `Change` node. It is configured to:
  - `Filename`: `filtered_pubs.csv`.
  - `Action`: `Append to file`.
  - `Add newline`: Unchecked.
  - `Create directory`: Checked.

4.3.5 Flow Diagram Segment

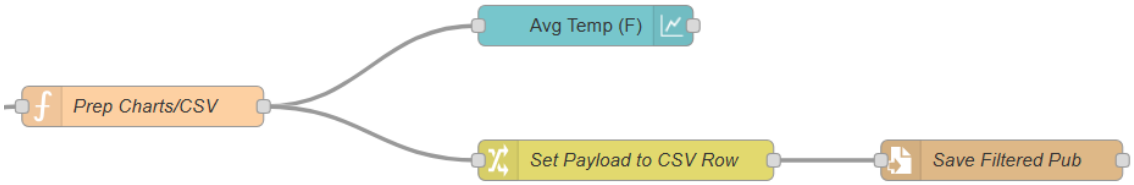

Figure 4: Complete Node-RED Flow Diagram. Step 4 corresponds to the path starting from the first output of "Process Row Data", through "Prep Chart/CSV", and then splitting to the "Avg Temp F" Chart node and the "Set Payload to CSV Row" / "Save Filtered Pub" nodes.

5 ACK Message Handling and ThingSpeak Update

5.1 Task Summary

This step introduces parallel processing for MQTT Acknowledgment (ACK) messages identified in the data retrieved from `challenge3.csv`. When a row found in Step 2 (corresponding to frame number `N`) contains an ACK message type (Connect Ack, Publish Ack, Sub Ack, or Unsub Ack) in its 'Info' field, two actions are triggered:

1. A global ACK counter is incremented.
2. A new entry is appended to `ack_log.csv` containing an incremental log number, the current timestamp, the original subscription ID (`SUB_ID`), and the specific ACK message type identified.
3. The current value of the global ACK counter is sent to a specified ThingSpeak channel using its HTTP API, updating 'field1'.

Messages that are not ACKs are ignored by this specific processing path.

5.2 Nodes Used

This functionality utilizes a distinct branch originating after the "Find Row by N" node, primarily using:

- `Switch` node ("Is ACK Msg?"): To filter messages based on ACK keywords in the 'Info' field.
- `Function` node ("Process ACK Data"): To increment counters, extract necessary data, and prepare outputs for CSV logging and ThingSpeak.
- `Change` node ("Set Payload for ACK CSV"): To prepare the message payload specifically for the ACK log file node.
- `File` node ("Save ACK Log"): To append the log entry to `ack_log.csv`.
- `Template` node ("Build ThingSpeak URL"): To construct the ThingSpeak API request URL.
- `HTTP Request` node ("Send to ThingSpeak"): To send the update request to ThingSpeak.

5.3 Node Configuration

5.3.1 Is ACK Msg? (Switch Node)

This node receives the message containing the found CSV row object from "Find Row by N". It filters the messages, allowing only those identified as ACKs to pass through to the next stage.

- `Property`: Checks `msg.payload.Info`.
- `Rules`: Configured with 4 rules to check if the property `contains` "Connect Ack", "Publish Ack", "Sub Ack", or "Unsub Ack", respectively.
- `Checking Logic`: Set to `stopping after first match`. This configuration ensures that if *any* of the rules match, the original message is passed forward.
- `Wiring`: As shown in the flow diagram, all relevant output ports of this Switch node are connected to the single input of the subsequent "Process ACK Data" node, implementing the "pass if any ACK matches" logic.

5.3.2 Process ACK Data (Function Node)

This node executes only for messages that passed the "Is ACK Msg?" switch. It performs the core logic for this step: increments the global ACK counter (`global.globalAckCounter`), gets the current timestamp, retrieves the original subscription ID (`SUB_ID`), extracts the specific ACK type from the `Info` string, calculates the row number for `ack_log.csv` (using `flow.ackLogCounter`), formats the CSV string and stores it in `msg.ackCsvRow`, and stores the current global ACK count in `msg.ackCountValue` for ThingSpeak.

```
// Input msg.payload is the row object from CSV, including original_id
let rowData = msg.payload;
let subId = msg.original_id || rowData.original_id;

if (typeof subId === 'undefined') {
    node.error("Original Sub ID missing in ACK processing", msg);
    return null;
}

// 1. Increment GLOBAL ACK counter
let ackCount = global.get("globalAckCounter") || 0;
ackCount++;
global.set("globalAckCounter", ackCount);

// 2. Get Current Timestamp (UNIX seconds)
let currentTimestamp = Math.floor(Date.now() / 1000);

// 3. Extract Message Type from Info string
let infoString = rowData.Info || "";
let msgType = "Unknown ACK";
const ackMatch = infoString.match(/(Connect Ack|Publish Ack|Sub Ack|Unsub Ack)/);
if (ackMatch) {
```

```
    msgType = ackMatch[0];
} else {
    // Fallback just in case contains worked but match didn't
    if (infoString.includes("Connect Ack")) msgType = "Connect Ack";
    else if (infoString.includes("Publish Ack")) msgType = "Publish Ack";
    else if (infoString.includes("Sub Ack")) msgType = "Sub Ack";
    else if (infoString.includes("Unsub Ack")) msgType = "Unsub Ack";
}

// 4. Get row number for ack_log.csv
let rowNum = flow.get("ackLogCounter") || 0;
rowNum++;
flow.set("ackLogCounter", rowNum);

// 5. Format CSV String: No.,TIMESTAMP,SUB_ID,MSG_TYPE
msg.ackCsvRow = `${rowNum},${currentTimestamp},${subId},"${msgType}"\n`;

// 6. Store the counter value needed for the URL
msg.ackCountValue = ackCount;

return msg;
```

5.3.3 ACK CSV Logging Path (Change & File Nodes)

This path logs the ACK information.

- **Set Payload for ACK CSV (Change Node):** Receives the message from "Process ACK Data". It sets `msg.payload` to the value of `msg.ackCsvRow`, preparing the correct string for the File node.
- **Save ACK Log (File Node):** Appends the received `msg.payload` (which now contains the formatted CSV string) to the file `ack_log.csv`. Configuration includes `Action: Append to file` and ensuring `Add newline` is unchecked.

5.3.4 ThingSpeak Update Path (Template & HTTP Request Nodes)

This parallel path sends the global ACK count to ThingSpeak.

- **Build ThingSpeak URL (Template Node):** Receives the message from 'Process ACK Data'. It generates the required HTTP GET request URL.
  - `Set property: msg.url`
  - `Format: Mustache template`
  - `Template`: Configured to generate the ThingSpeak URL using the Mustache template format.
    The generated URL follows this pattern: `https://api.thingspeak.com/update?api_key=WRITE_API_KEY&field1=\{\{ackCountValue\}\}`

- **Send to ThingSpeak (HTTP Request Node):** Receives the message with `msg.url` set.
  - `Method`: `GET`
  - `URL`: Left blank (uses `msg.url`).
  - `Return`: `ignore response`.

  This node executes the request to update the ThingSpeak channel's field1 with the value from `msg.ackCountValue`.

5.3.5 Flow Diagram Segment

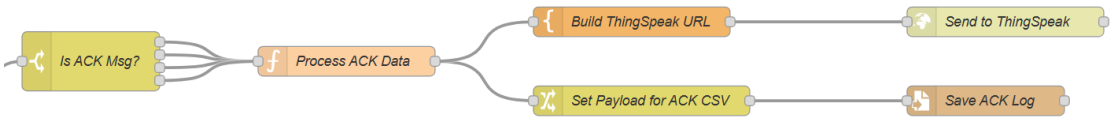

Figure 5: Complete Node-RED Flow Diagram. Step 5 corresponds to the path starting after "Find Row by N", filtering through "Is ACK Msg?", processing in "Process ACK Data", and then splitting to the "Save ACK Log" File node and the "Send to ThingSpeak" HTTP Request node.

6 Message Processing Limit

6.1 Task Summary

The final requirement was to limit the processing performed by the subscriber branch of the flow. The flow needed to be configured to stop processing new incoming messages from the `challenge3/id_generator` subscription after exactly 80 messages had been received. This count includes all messages received by the subscriber, regardless of whether they subsequently resulted in finding a matching row in the CSV or triggering further actions (like re-publishing or logging ACKs/Temps).

6.2 Nodes Used

This functionality was implemented by adding one primary node at the beginning of the subscriber branch:

- `Function` node ("Count & Limit Msgs"): Inserted immediately after the `MQTT In` node ("Subscribe ID/Timestamp") to count messages and conditionally stop the flow path.

6.3 Node Configuration

6.3.1 Count & Limit Msgs (Function Node)

This node acts as a gatekeeper for the entire subscriber processing logic. It is placed directly after the `MQTT In` node ("Subscribe ID/Timestamp"), ensuring every received message passes through it first.

- It utilizes a counter stored in global context (`global.subscriberMsgCount`) to keep track of how many messages have been received since the counter was last reset.
- Upon receiving a message, it increments this counter.
- It then checks if the incremented count exceeds the defined limit (80).
- If the count is within the limit (1 to 80), it passes the original message (`msg`) unchanged to its output, allowing the rest of the subscriber flow ("Calculate N", etc.) to proceed.
- If the count exceeds the limit (81 or higher), it returns `null`. Returning `null` from a function node effectively stops the message flow; the message is discarded, and no subsequent nodes in that path are executed.

This ensures that only the first 80 messages received trigger the full subscriber processing logic.

```
1 // Define the maximum number of messages to process
2 const MESSAGE_LIMIT = 80;
3
```

```
// Get the current count from global context, default to 0 if not set
let currentCount = global.get("subscriberMsgCount") || 0;

// Increment the counter
currentCount++;

// Store the updated count back into global context
global.set("subscriberMsgCount", currentCount);

// Check if the limit has been exceeded
if (currentCount > MESSAGE_LIMIT) {
    // If limit exceeded, log it and stop processing by returning null
    node.warn(`Message limit (${MESSAGE_LIMIT}) reached. Further messages will be ignored by this flow branch.`);
    return null;
} else {
    return msg;
}
```

6.3.2 Flow Diagram Segment

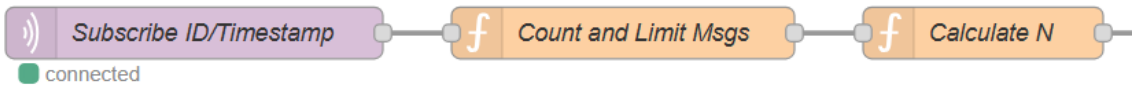

Figure 6: Complete Node-RED Flow Diagram. Step 6 involves the insertion of the "Count & Limit Msgs" Function node immediately after the "Subscribe ID/Timestamp" MQTT In node.

7 Results of Full Run

This section presents the final results obtained after performing a clean run of the complete Node-RED flow, configured to stop processing after receiving exactly 80 messages from the initial MQTT subscription.

7.1 Final Flow Diagram

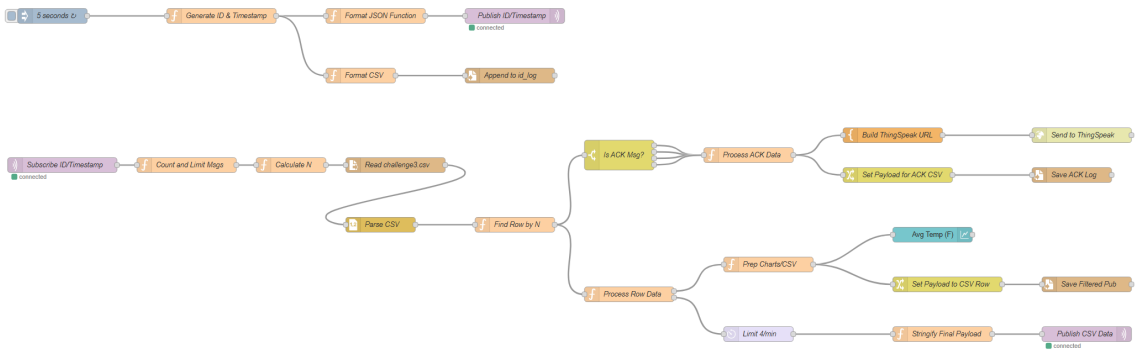

Figure 7: Complete Node-RED Flow Diagram Implementing All Steps.

7.2 Generated CSV Files

The following subsections show the content of the three CSV files generated during the 80-message run. The content is included directly from the uploaded files.

7.2.1 `id_log.csv` Content

This file logs the Row Number, generated random ID, and UNIX timestamp for every message published to the `challenge3/id_generator` topic (up to the 80 processed messages).

```
No.,ID,TIMESTAMP
1,1830,1745531691
2,24096,1745531696
3,22645,1745531701
4,3173,1745531706
5,25866,1745531711
6,16343,1745531716
7,22381,1745531722
8,19783,1745531727
9,6485,1745531732
10,21713,1745531737
11,23327,1745531742
12,1576,1745531747
13,4939,1745531753
14,13655,1745531758
15,2716,1745531763
16,13984,1745531768
17,28165,1745531773
18,23582,1745531778
```

```
19,8102,1745531783
20,26144,1745531788
21,621,1745531793
22,22241,1745531798
23,10367,1745531803
24,6964,1745531808
25,26318,1745531813
26,28817,1745531819
27,7736,1745531824
28,13523,1745531829
29,4851,1745531834
30,18553,1745531839
31,16507,1745531844
32,10348,1745531849
33,24920,1745531854
34,23002,1745531859
35,2561,1745531864
36,6928,1745531869
37,26176,1745531874
38,2733,1745531879
39,24037,1745531884
40,7530,1745531889
41,10083,1745531894
42,10484,1745531899
43,24718,1745531904
44,1602,1745531909
45,9708,1745531915
46,16128,1745531920
47,19318,1745531925
48,12836,1745531930
49,18137,1745531935
50,7194,1745531940
51,13006,1745531945
52,13369,1745531951
53,28280,1745531956
54,3729,1745531961
55,7391,1745531966
56,6231,1745531971
57,247,1745531976
58,10034,1745531981
59,3832,1745531986
60,4990,1745531991
61,19781,1745531996
62,17158,1745532001
63,18945,1745532006
64,12120,1745532011
65,5279,1745532017
66,13146,1745532022
67,9011,1745532027
68,3992,1745532032
69,21672,1745532037
70,17088,1745532042
71,2156,1745532048
72,3356,1745532053
73,19927,1745532058
74,29134,1745532063
```

```
75,13279,1745532068
76,12591,1745532073
77,15138,1745532078
78,13756,1745532083
79,7361,1745532088
80,4819,1745532093
```

7.2.2 filtered_pubs.csv Content

This file logs entries only for messages corresponding to Fahrenheit temperature readings found in challenge3.csv. It includes the specific log's row number, location data, calculated mean temperature, type, unit, and description.

```
No.,LONG,LAT,MEAN_VALUE,TYPE,UNIT,DESCRIPTION
1,51,74,20.5,temperature,F,"Room Temperature"
2,49,48,19.5,temperature,F,"Room Temperature"
3,41,63,29,temperature,F,"Room Temperature"
4,50,46,24.5,temperature,F,"Room Temperature"
5,96,90,26.5,temperature,F,"Room Temperature"
6,44,45,22.5,temperature,F,"Room Temperature"
7,43,72,23.5,temperature,F,"Room Temperature"
8,41,48,29.5,temperature,F,"Room Temperature"
9,96,70,32.5,temperature,F,"Room Temperature"
10,54,57,21,temperature,F,"Room Temperature"
11,66,98,26.5,temperature,F,"Room Temperature"
```

7.2.3 ack_log.csv Content

This file logs entries only when an MQTT ACK message type is identified in the corresponding row from challenge3.csv. It includes the specific log's row number, the timestamp when the ACK was processed, the original subscription ID (SUB_ID), and the type of ACK message.

```
No.,TIMESTAMP,SUB_ID,MSG_TYPE
1,1745531778,23582,"Publish Ack"
2,1745531889,7530,"Publish Ack"
3,1745531940,7194,"Publish Ack"
4,1745531971,6231,"Publish Ack"
```

7.3 Generated Charts

7.3.1 ThingSpeak ACK Counter Chart

The global counter for ACK messages processed was sent to ThingSpeak field1. Figure 8 shows the chart from the public channel. The counter increments each time an ACK message was identified in the challenge3.csv data during the run. The public ThingSpeak channel can be viewed at: https://thingspeak.mathworks.com/channels/2930201

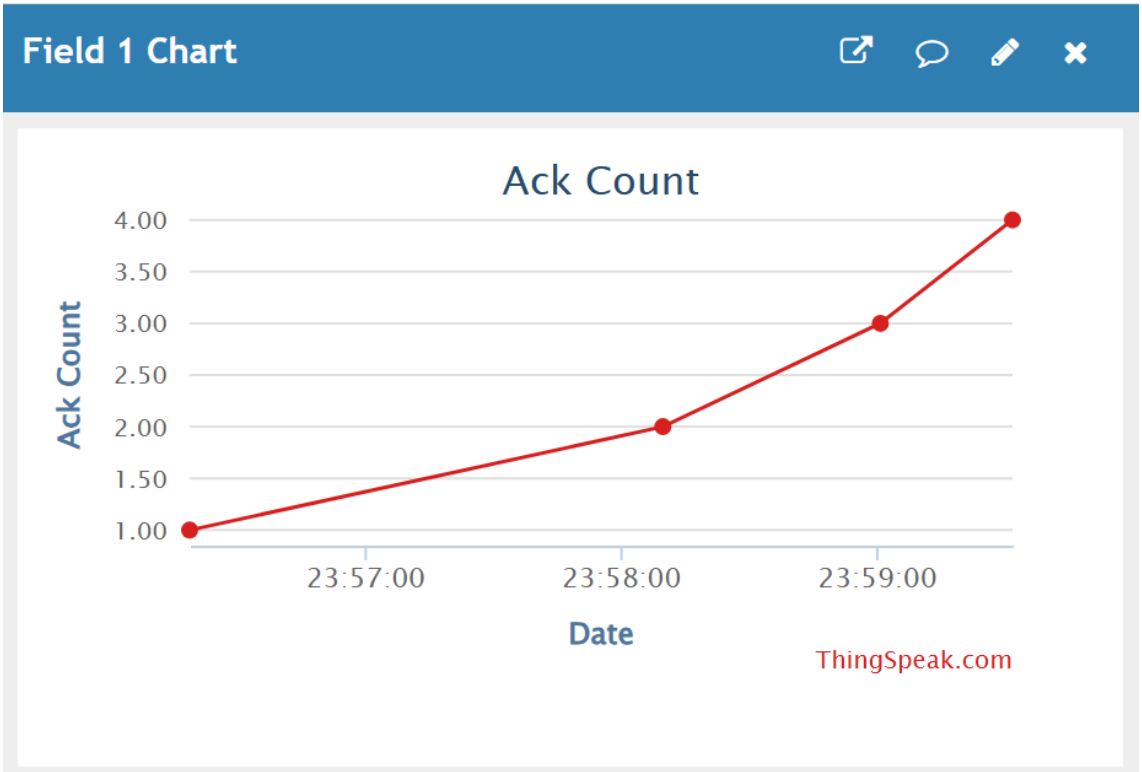

Figure 8: ThingSpeak Chart showing the Global ACK Counter (Field 1) over time.

7.3.2 Node-RED Dashboard Temperature Chart

The Node-RED Dashboard chart displayed the mean temperature value for payloads identified as Fahrenheit temperature readings. Figure 9 shows the resulting chart after the 80-message run.

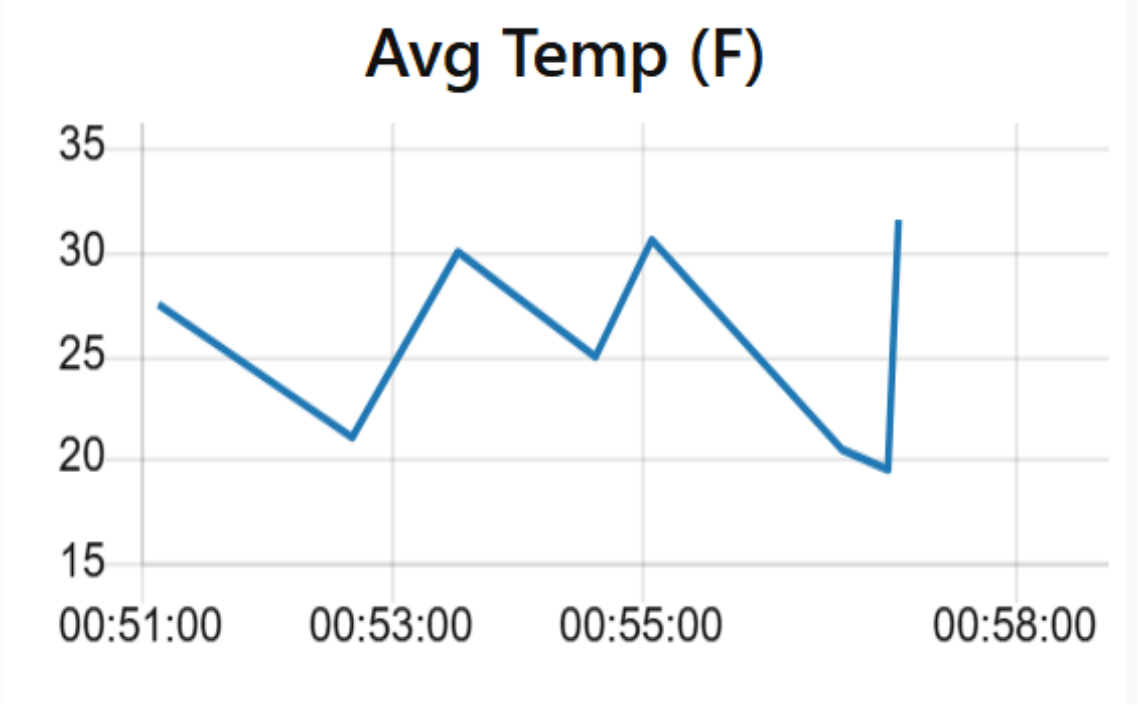

Figure 9: Node-RED Dashboard Chart plotting Mean Temperature (°F) values. Note: The specific values shown in this chart screenshot may differ from the data in `filtered_pubs.csv`, as they originate from different simulation runs.