# Internet of Things

## Challenge n. 3: Node-RED

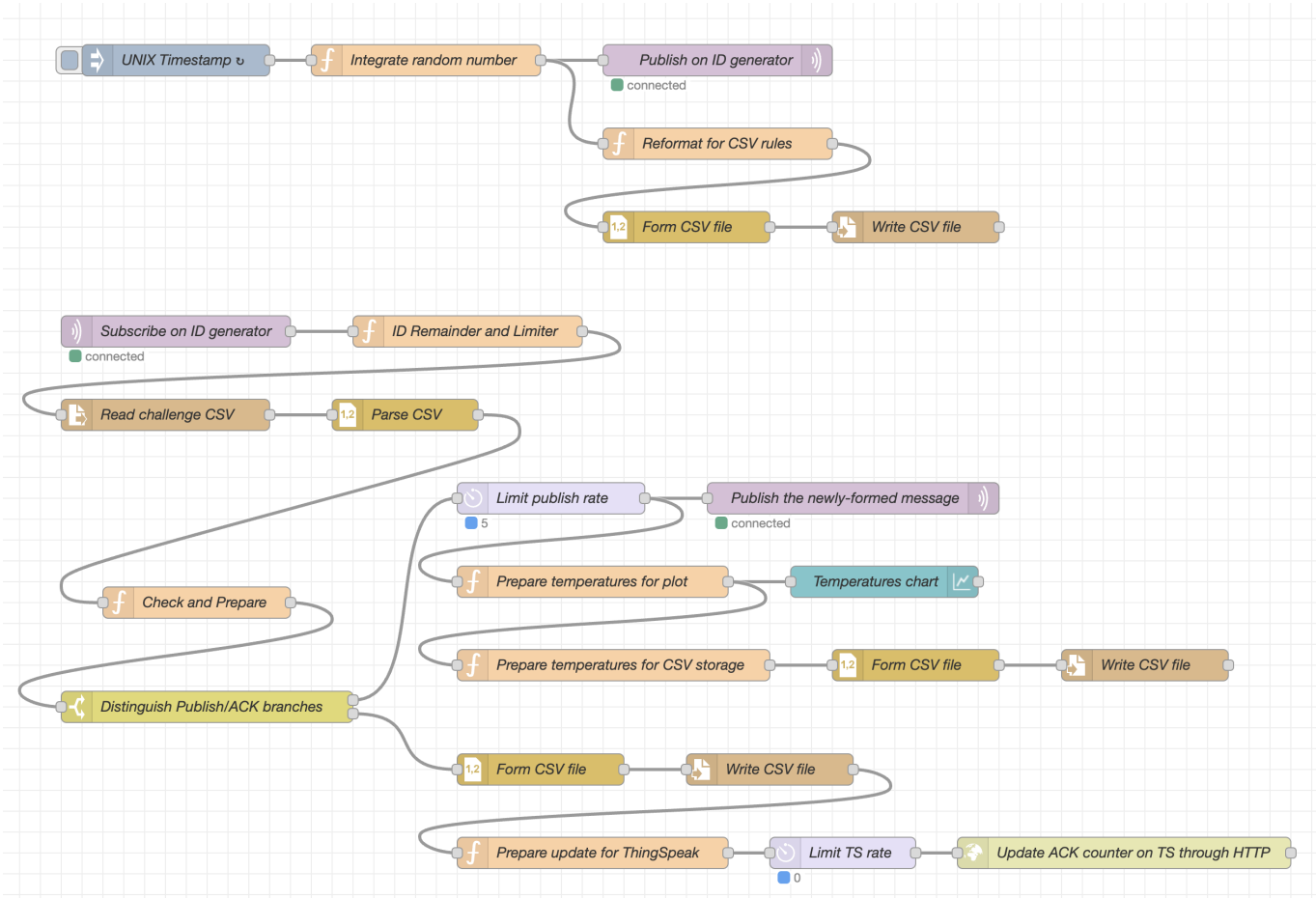
Andrea Caravano, Alberto Cantele

Academic Year 2024–25

## Contents



### Node-RED chart



### ThingSpeak public channel

`https://thingspeak.mathworks.com/channels/2922298`

# 1 Implementation and design choices

All the question have been answered by implementing a Node-RED flow that integrates all the functionalities required by the first part of the Challenge, in a compact way, which eases readability and clarity of the overall structure.
In the following description, each node has its own separate space, in which Input, Output and Processing details are provided.
A deeper explaination of the most meaningful points is then expanded and correlated by the implementation of the JavaScript function nodes, when used.

# 2 ID Generator Publisher

## 2.1 UNIX Timestamp: injection node

### 2.1.1 Input

This is an injection node: no input is expected.

### 2.1.2 Output

`msg.timestamp` = current UNIX timestamp

The current UNIX timestamp is the number of milliseconds elapsed from the Epoch (defined on 01/01/1970) and provides a relative time measurement of the instant in which the message has been generated.
An example of the resulting output is provided in the following.

```
{
    "timestamp":1745243009801
}
```

### 2.1.3 Processing details

The processing phase is implicitly carried out by the node implementation.

## 2.2 Integrate random number: function node

### 2.2.1 Input

The input is provided by the injection node, which generates a message containing the current UNIX timestamp, to integrate further.

### 2.2.2 Output

`msg.payload.timestamp` = UNIX timestamp,
`msg.payload.id` = a random number between 0 and 30000 (included)

Note that the resulting format is different from the one that will be later used for CSV storage.

An example of the resulting output is provided in the following.

```
{
    "payload":
        {
            "id":9026,
            "timestamp":1745243009801
        }
}
```

#### 2.2.3 Processing details

The function node integrates the UNIX timestamp received in input with a random number between 0 and 30000 (included) that will then be used in successive computations.

The payload is then formed aggregating both of them.

#### 2.2.4 Implementation

```
// Random generator boundaries
const RAND_MAX = 30000;
const RAND_MIN = 0;

// Integer random
let generated = Math.floor(Math.random() * RAND_MAX) + RAND_MIN;
let timestamp = msg.timestamp;

// Prepare the payload format of the MQTT Publish message
// Note this format is different from the one used in the CSV file, as per specifications
msg = {
    payload: {
        id: generated, // the generated integer random number
        timestamp: timestamp // the left parameter being the payload field, the right one
        ↪  being the local variable
    }
}

return msg;
```

### 2.3 Publish on ID Generator: MQTT Publisher node

#### 2.3.1 Input

The aggregated Publish Message described earlier is injected.

#### 2.3.2 Output

This is a publish node: no output is expected.

#### 2.3.3 Processing details

The aggregated payload is published on the `challenge3/id_generator` topic, as per specifications.

All of the MQTT publishing parameters are statically set via the node parameters.

QoS and Retain values are set to their default (0 and False, respectively).

### 2.4 Reformat for CSV rules: function node

#### 2.4.1 Input

The aggregated Publish Message described earlier is injected.

#### 2.4.2 Output

`msg.payload."No."` = sequential counter of published IDs,
`msg.payload.ID` = the randomly generated ID,
`msg.payload.TIMESTAMP` = the UNIX timestamp

An example of the resulting output is provided in the following.

```
{
    "payload":
        {
            "No.":3,
            "ID":9026,
            "TIMESTAMP":1745243009801
        }
}
```


#### 2.4.3 Processing details

The published values are moved to a convenient position for CSV parsing and respecting the specifications for CSV formatting.

A local context counter is stored and updated to integrate the payload fields with it.

In a sense, it also acts as a change node and formatter for the publishing parameters, integrating their roles via code.

#### 2.4.4 Implementation

On node start (executed only once, at flow deployment):

```
// The ID counter starts from 0, as it is always incremented before usage
if (context.get("id_counter") === undefined) {
    context.set("id_counter", 0);
}
```

On message arrival (standard function node behaviour):

```
// Increment the ID counter (from local node context)
context.set("id_counter", context.get("id_counter") + 1);

// Form the payload content with the parameters required by the CSV formatting
// Starting from the ones provided as part of the Publish payload
// In a sense, this function node also serves as a change node for the already provided
↪  parameters
// For the CSV parsing to happen, the interested fields must be present in the output payload
msg.payload = {
                "No.": context.get("id_counter"),
                ID: msg.payload.id,
                TIMESTAMP: msg.payload.timestamp
};

return msg;
```

### 2.5 Form CSV file: CSV formatting node

#### 2.5.1 Input

The re-formatted PUBLISH MESSAGE is injected.

#### 2.5.2 Output

`msg.payload` = CSV payload,
`msg.columns` = CSV headers

An example of the resulting output is provided in the following.


```
{
    "payload":"3,9026,1745243009801",
    "columns":"No.,ID,TIMESTAMP"
}
```

#### 2.5.3 Processing details

The chosen line separators (Linux, `\n`) are injected through the CSV formatting node and not via the Write File one, so they should not be repeated later.

Empty and null fields are included in the parsing procedure.

Headers are instead included only at flow deployment (so they behave as expected, being present only at the first line of the output file).

### 2.6 Write CSV file: file output node

#### 2.6.1 Input

The CSV payload formatted for CSV output writing.

#### 2.6.2 Output

The output of this node replicates the input, but integrates the file name on which the CSV payload is stored (`msg.filename = "/data/challenge/id_log.csv"`).

On the machine's disk, of course, it also writes (and updates) the CSV output file.

#### 2.6.3 Processing details

New content is, naturally, appended.

Line separators are already injected by the CSV parsing node (see related notes).

## 3 ID Generator Subscriber and parser

### 3.1 Subscribe on ID generator: MQTT Subscriber node

#### 3.1.1 Input

This is a subscribe node: no input is expected.

#### 3.1.2 Output

The received resulting Publish Message is returned.

The expected format, topic and parameters are the same described at the Publish side.

An example of the resulting output is provided in the following.

```
{
    "topic":"challenge3/id_generator",
    "payload":
        {
            "id":28784,
            "timestamp":1745242246354
        },
    "qos":0,
    "retain":false
}
```

#### 3.1.3 Processing details

The aggregated payload is received on the `challenge3/id_generator` topic, as per specifications.

All of the MQTT subscribe parameters are statically set via the node parameters.

QoS value is set to its default (0).

## 3.2 ID Remainder and Limiter: function node

### 3.2.1 Input

The received PUBLISH MESSAGE by means of the ID generator subscription.

### 3.2.2 Output

The expected output matches the received PUBLISH MESSAGE, integrates the remainder (via the modulo operation) and stores a copy of the PUBLISH payload (`id` and `timestamp` values), for successive computations.

`msg.remainder` = the result of the modulo operation,
`msg.sub` = the received PUBLISH MESSAGE by means of the subscription

An example of the resulting output is provided in the following.


```
{
    "remainder":5651,
    "sub":
        {
            "id":28784,
            "timestamp":1745242246354
        }
}
```


### 3.2.3 Processing details

The ID process limit is implemented trough a local context counter, described in the following, alongside the remainder computation.

### 3.2.4 Implementation

On node start (executed only once, at flow deployment):

```
// The ID process counter starts from -1, as it is always incremented before usage
if (context.get("process_counter") === undefined) {
    context.set("process_counter", -1);
}
```

On message arrival (standard function node behaviour):

```
const REMAINDER_NUM = 7711; // constant (number of CSV entries in the provided input file)
const ID_PROCESS_LIMIT = 80; // maximum number of subscription-side ID messages to process

let payload = msg.payload;

// Reset the message structure, for clarity
msg = {};
// Modulo operation
msg.remainder = parseInt(payload.id) % REMAINDER_NUM;
// Storage of the received subscription payload (it will be useful in the successive
↪  computation)
msg.sub = payload;

// The local processing counter is updated
context.set("process_counter", context.get("process_counter") + 1);

// Checks if the resulting message is 0 (not existing, since counting starts from 1 in the CSV
↪  file, but note that it is still counted in the process limit)
// or the processing counter has been reached and successive messages are to be discarded (no
↪  return)
```

```
if (msg.remainder !== 0 && context.get("process_counter") < ID_PROCESS_LIMIT) // <= if instead
↪  counting from 0
    return msg;
```

## 3.3 Read challenge CSV: file input node

### 3.3.1 Input

The integrated Publish Message flowing as part of the subscription.

In this case, it serves as an injection node, to let the activity of the file input read start (its fields will be replicated for successive computations).

### 3.3.2 Output

The unformatted input file, served as a complete flattened string, for successive CSV parsing, alonside the rest of the remainder and subscription fields, being replicated.

`msg.payload` = complete flattened CSV string

### 3.3.3 Processing details

The provided input file is read and returned as payload to the parser node.

## 3.4 Parse CSV: CSV formatting node

### 3.4.1 Input

The unformatted challenge CSV file, served as a complete flattened string, for CSV parsing.

### 3.4.2 Output

The CSV input payload formatted for JSON/JavaScript computations, as a JavaScript data structure.

An example of the resulting output is provided in the following.

```
{
    "payload":"(omitted, the resulting array containing all the parsed messages)",
    "remainder":5651,
    "sub":{
        "id":28784,
        "timestamp":1745242246354
    },
    "filename":"/data/challenge/challenge3.csv",
    "columns":"No.,Time,Source,Destination,Protocol,Length,Source Port,Destination
    ↪  Port,Info,Payload"
}
```

### 3.4.3 Processing details

Empty and null fields are included in the parsing procedure.

## 3.5 Check and Prepare: function node

### 3.5.1 Input

The CSV input payload formatted for JSON/JavaScript computations, as a JavaScript data structure.

### 3.5.2 Output

A Publish Message, which is then eventually re-formatted for temperature analysis or an Acknowledgement Message.

They are later distinguished by a Switch node.

An example of the resulting output is provided in the following.

A Publish Message:


```
{
    "topic":"factory/room1/room6/hydraulic_valve",
    "payload":
        {
            "timestamp":1745249897928,
            "id":27562,
            "topic":"factory/room1/room6/hydraulic_valve",
            "payload":{
                "unit": "F",
                "long": 87,
                "description": "Room Temperature",
                "lat": 91,
                "range": [10, 37],
                "type": "temperature"
            }
        }
}
```


An Acknowledgement Message:


```
{
    "payload":
        {
            "No.":12,
            "TIMESTAMP":1745259353585,
            "SUB_ID":296,
            "MSG_TYPE":"Publish Ack"
        }
}
```


### 3.5.3 Processing details

In the provided CSV, numbering starts from 1, unlike JavaScript data structures, that are instead numbered starting from 0: the input reimainder is adapted to the CSV format.

Packets that apply the Piggy-Backing technique (transporting more than one description and payload in their structure) are assumed to have a well-formed separation architecture.

The number of descriptions and related payloads are therefore assumed to be the same, but can be null or not validly defined.

The edge case, in which an aggregated message carries disomogeneous types (therefore disaligning payloads indexes) is also managed, through a support counter for any other kind of message type encountered.

A FIFO data structure, in the form of a Stack, is used to form a network buffer that stores the packets that will be returned by the node.

JSON parsing is then applied to the remaining fields, in a coherent format with respect to the specifications.

In Node-RED, there are two possible strategies implementing a network buffer:

- Flattening: An arbitrarily complex and convoluted data structure is flattened in output, multiplying the number of returned messages by the cardinality of the data structure used.

The implementing syntax requires indication of the interested data structure between brackets: `return [buffer];`.

This is the chosen approach.

- Pull-based: A FIFO data structure is enforced: in the buffering node, messages are pushed, while in the receiving node, messages are pulled singularly.

  This is an unnatural approach in Node-RED, being it based on a continuous message flow, that is clearly represented by its implementation chart.

### 3.5.4 Implementation

On node start (executed only once, at flow deployment):

```
// The global ACK counter starts from 0, as it is always incremented before usage
// Specifications require the global counter is reset at each flow deployment
// This code will be executed only once, when deploying/updating the flow
global.set("ack_counter", 0);
```

On message arrival (standard function node behaviour):

```
// The input flowing in the function node is the output of the CSV read-file parsing
let csv_entries = msg.payload;
// In the provided CSV, numbering starts from 1, unlike JavaScript data structures that are
↪  numbered from 0
// As described earlier, this motivates the exclusion of the remainder 0 by dropping a
↪  possible resulting packet
let remainder = msg.remainder - 1;

// Parsing of the packet description (type of packet)
// Packets applying piggy-backing (transporting more than one description+payload in their
↪  structure)
// are separated by a comma (,) and a space ( )
// This produces an array data structure in which each packet information and payload is
↪  separated
let info = String(csv_entries[remainder].Info);
let infos = info.split(", ");

// Parsing of the packet payload
// Packets applying piggy-backing (transporting more than one description+payload in their
↪  structure)
// are separated by a comma (,) and a curly bracket ({)
// This produces an array data structure in which each packet information and payload is
↪  separated
// The curly bracket ({) is restored back since it is already part of the JSON payload inner
↪  structure
let payload = String(csv_entries[remainder].Payload);
let payloads = payload.split(",{");
for (let i = 1; i < payloads.length; i++)
    payloads[i] = '{' + payloads[i];

// A Publish Message has its own description separated by the topic through a couple of
↪  brackets (Publish Message (eventual id) [topic])
// Therefore, the interesting part for identification is before the opening bracket ([),
↪  translating to the first position of the split result
let types = [];
for (let i = 0; i < infos.length; i++)
    types[i] = infos[i].split(' [')[0];

// Explicit assertion: the number of packet descriptions and payloads is the same.
// A payload being null or undefined still passes this check (being it still a payload, but
↪  with a non-meaningful value).
```

```
// A non formatted payload in a piggy-backed message (containing more than one) can't be
↪  incoherent in size.
// The case in which the number of descriptions does not match the number of payloads is
↪  therefore not valid, in an unformatted environment.
// Instead, the case in which the message is well formatted but one or more payloads (or its
↪  components)
// are missing is coherent and valid parsing is performed in the following.
let publish_checker = 0;
types.forEach(t => {
    if (t.startsWith("Publish Message"))
        publish_checker++;
});
console.assert(publish_checker == payloads.length);

// A support variable, used to gather the right payload match later
let non_publish_counter = 0;

// A Stack (FIFO) data structure, acting as a network buffer
// Its behaviour is ideal both for the flattening and FIFO cases (see notes)
let buffer = [];
for (let i = 0; i < infos.length; i++) {
    let type = types[i];
    // As shown, a publish message is declared with "Publish Message" at the beginning of its
    ↪  description
    if (type.startsWith('Publish Message')) {
        // Publish Messages are of course interleaved with any other piggy-backed message,
        ↪  potentially not being a Publish one
        // (and therefore not carrying a payload)
        payload = payloads[i - non_publish_counter];
        // As described earlier, the next part of the split result is now interesting for the
        ↪  topic identification
        // Next, it is also split for the closing bracket (]) being the separator for the
        ↪  topic identifier ending
        let topic = infos[i].split('[')[1].split(']')[0];
        // manage the case of null or undefined payload
        if (payload === null || payload === undefined || payload.length === 0) {
            payload = "{}"; // replace with empty JSON payload, as per specifications
        }

        // Final publishing content setup, as per specifications
        let content = {
            timestamp: Date.now(), // UNIX timestamp (number of milliseconds elapsed from the
            ↪  standard Epoch time discriminant (01/01/1970))
            id: msg.sub.id, // ID of the message received through the ID generator
            ↪  subscription (see previous description)
            topic: topic, // Identified topic
            payload: payload // n-th payload in the message
        }

        let result = {
            topic: topic, // the MQTT topic on which to publish can also be set via the topic
            ↪  property
            // But must be replicated in this field, since is required by the MQTT Publisher
            ↪  node to operate correctly
            payload: content // Content described earlier
        }

        // FIFO data structure behaviour: the resulting publish message is pushed
        // to be popped by successive computations
        buffer.push(result);
    } else if (type.includes('Ack')) {
        non_publish_counter++;
```

```
        // The global ACK counter is updated among (possibly) the whole Node-RED installation
        global.set("ack_counter", global.get("ack_counter") + 1);

        // Final ACK content setup, as per specifications
        // The chosen format is already the one that will be parsed and stored in the CSV file
        let content = {
            "No.": global.get("ack_counter"), // Global ACK counter (can be seen by the whole
            ↪  installation)
            TIMESTAMP: Date.now(), // UNIX timestamp (number of milliseconds elapsed from the
            ↪  standard Epoch time discriminant (01/01/1970))
            SUB_ID: msg.sub.id, // ID of the message received through the ID generator
            ↪  subscription (see previous description)
            // An Acknowledgement message has its own description separated by the ID through
            ↪  a couple of brackets (Publish Ack (eventual id) [eventual notes])
            // Therefore, the interesting part for identification is before the opening
            ↪  bracket ((), translating to the first position of the split result
            // Then, it must be separated by the following opening bracket ([) for notes
            MSG_TYPE: (type.split(' (')[0]).split('[')[0]
        }

        let result = {
            // ACKs, of course, do not have a topic, as they will not be published!
            // This is a sufficient characteristic to let the Switch node differentiate among
            ↪  the two message kinds!
            payload: content
        }

        // FIFO data structure behaviour: the resulting publish message is pushed
        // to be popped by successive computations
        buffer.push(result);
    } else non_publish_counter++;
}

// Node-RED operates flattening on the return structure,
// so there will be separate messages being returned for both single and piggy-backed
↪  aggregated messages
return [buffer];
```

## 3.6 Distinguish Publish/ACK branches: switch node

### 3.6.1 Input

The aggregated data structure carrying a PUBLISH or an ACKNOWLEDGEMENT message, for later analysis.

### 3.6.2 Output

The input message is replicated on the chosen branch.

### 3.6.3 Processing details

Uses the topic presence as a discriminant among PUBLISH and ACKNOWLEDGEMENT messages.
The processing flow continues in two separate output branches.

## 3.7 Publish the newly-formed message: MQTT Publisher node

### 3.7.1 Input

The aggregated PUBLISH MESSAGE formed by the function node is retrieved from the FIFO buffer (in our case, automatically flattened by NODE-RED).
It is filtered by a rate limiter node, that, per specifications, is set to a limit of 4 messages per minute.

#### 3.7.2 Output

This is a publish node: no output is expected.

#### 3.7.3 Processing details

Reliability MQTT publishing parameters are statically set via the node parameters.

The topic used is the one received by the PUBLISH MESSAGE, as per specifications (therefore, it is dynamic).

QOS and RETAIN values are set to their default values (0 and FALSE, respectively).

### 3.8 Prepare temperatures for plot: function node

#### 3.8.1 Input

The aggregated PUBLISH MESSAGE formed by the function node is retrieved from the FIFO buffer (in our case, automatically flattened by NODE-RED).

It is filtered by a rate limiter node, that, per specifications, is set to a limit of 4 messages per minute.

#### 3.8.2 Output

The average temperature measured by the input message range, alongside the parsed message.

The payload (average temperature measured) is used to draw the temperatures chart.

It is then available through the NODE-RED DASHBOARD (at `/ui`).

`msg.payload` = average temperature measurement,
`msg.parsed` = JSON parsed message payload

An example of the resulting output is provided in the following.

```
{
    "payload":23.5,
    "parsed":
        {
            "description":"Room Temperature",
            "long":87,
            "range":[10, 37],
            "lat":91,
            "type":"temperature",
            "unit":"F"
        }
}
```

#### 3.8.3 Processing details

Some parsing checks are needed when filtering out unparsable or incomplete payloads.

Moreover, some explicit assumptions need to be checked: payload type is `temperature` and unit is `F` (Fahrenheit).

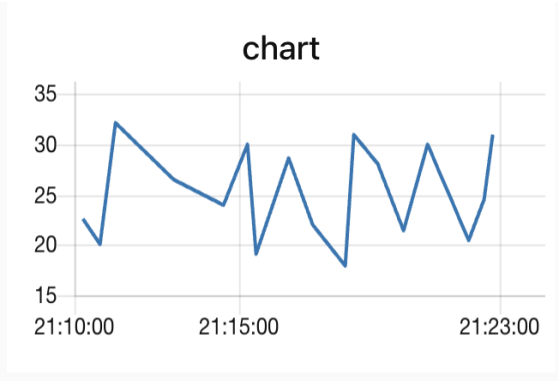
The range is also composed by exactly 2 (possibly floating-point) numbers that are then parsed when computing the final average value.

#### 3.8.4 Implementation

```
let parsed;

// The payload received through the published message is parsed.
// If no parsing is possible (the payload is therefore invalid),
// the payload is nulled and the whole message discarded by the successive check
try {
```

```
    parsed = JSON.parse(msg.payload.payload);
} catch (e) { parsed = null; }

if (parsed !== null && // the parsed payload is null or converted to null being it unparsable
    parsed !== undefined && // the parsed payload, even if parsed, has no useful content
    ↪ detected
    parsed.type === "temperature" && // the message being transported is a temperature
    ↪ measurement
    parsed.unit === "F" && // the used unit is Fahrenheit degrees
    parsed.range !== null && // the temperature range is invalid or empty
    parsed.range !== undefined &&
    parsed.range.length === 2) { // temperature range is (as expected) composed by both the
    ↪ minimum and maximum values

    let range = {
        min: parseFloat(parsed.range[0]), // the minimum value is the first part of the range
        max: parseFloat(parsed.range[1]) // the maximum value is the second part of the range
    };

    // The range is prepared for plotting
    let ret = {
        // Average calculation
        payload: (range.min + range.max) / 2,
        // Storage of the parsed payload for successive computations
        parsed: parsed
    }

    return ret;
}
```

#### 3.8.5 Temperatures chart: example of resulting plot




### 3.9 Prepare temperatures for CSV storage: function and CSV formatting node

#### 3.9.1 Input

The parsed JSON message, with all its fields and the computed temperature average.

#### 3.9.2 Output

`msg.payload` = CSV-formatted fields (`No.,LONG,LAT,MEAN_VALUE,TYPE,UNIT,DESCRIPTION`)

An example of the resulting output is provided in the following.

```
{
    "payload":
        {
            "No.":1,
            "LONG":87,
```

```
            "LAT":91,
            "MEAN_VALUE":23.5,
            "TYPE":"temperature",
            "UNIT":"F",
            "DESCRIPTION":"Room Temperature"
        }
}
```

#### 3.9.3 Processing details

The temperatures counter is a local node context variable, updated before usage and prepared at flow initialization.

The CSV-formatted style is then prepared gathering all of the parsed parameters.

#### 3.9.4 Implementation

On node start (executed only once, at flow deployment):

```
// The stored temperatures counter starts from 0, as it is always incremented before usage
if (context.get("temp_counter") === undefined) {
    context.set("temp_counter", 0);
}
```

On message arrival (standard function node behaviour):

```
// The stored temperatures counter is incremented before usage
context.set("temp_counter", context.get("temp_counter") + 1);

// The parsed payload has been piggy-backed earlier by the input node
let parsed = msg.parsed;
// The average temperature value has been set as payload for plot display
let mean = msg.payload;

msg = {};

msg.payload = {
                "No.": context.get("temp_counter"), // Stored temperatures counter
                LONG: parsed.long, // Longitude
                LAT: parsed.lat, // Latitude
                MEAN_VALUE: mean, // Average temperature
                TYPE: parsed.type, // Expected type is "temperature" (measurement)
                UNIT: parsed.unit, // Expected unit is "F" (Fahrenheit)
                DESCRIPTION: parsed.description // A description of the payload content
};

return msg;
```

### 3.10 Prepare update for ThingSpeak: function node

#### 3.10.1 Input

The CSV-parsed Acknowledgement structure is provided to the function node after storage.

Its format has already been defined earlier, when preparing for branching, and is not modified after, since its only usage destinations are compatible with the former format style.

#### 3.10.2 Output

An HTTP Request is built up imposing the GET method and the ThingSpeak API Writing Endpoint URL, correlated by the private API Key and updated global Acknowledgement counter.

It is then filtered by a rate limiter node, imposing a message limit of one per 20-second period, as per ThingSpeak free plan requirements.

`msg.method` = the HTTP Request method (GET),
`msg.url` = the ThingSpeak API Writing Endpoint URL, appended to the API Key and updated global ACK counter value

An example of the resulting output is provided in the following.

```
{
    "method":"GET",
    "url":"https://api.thingspeak.com/update?api_key=API_KEY&field1=4"
}
```

### 3.10.3 Processing details

Messages flowing outside the rate limit are queued.

The global Acknowledgement counter is stored for the whole Node-RED installation while active.

### 3.10.4 Implementation

```
// The API Key is stored as a local node variable and used as part of the URL contacted
↪  through HTTP
let API_KEY = "<omitted>";

msg = {};
msg.method = "GET";
// A GET request to the ThingSpeak API endpoint for data writes is prepared, packing up also
↪  the global ACK counter
msg.url = "https://api.thingspeak.com/update?api_key=" + API_KEY + "&field1=" +
↪  global.get("ack_counter");

return msg;
```