POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Course: Internet of Things - 5 CFU

Professor: Matteo Cesana
Students: Bruno Arena, Matteo Cavalleri
Personal IDs: 10806605, 10768765

Challenge 3 – Part 1

POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Abstract

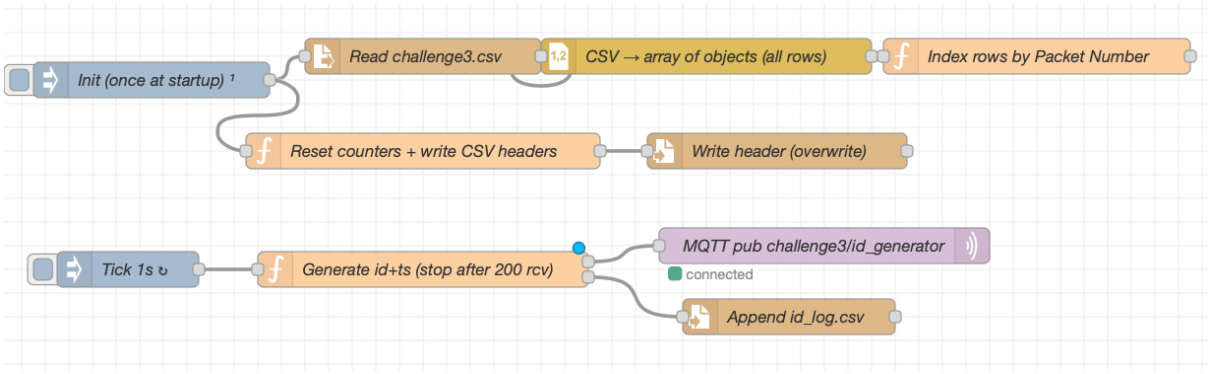
This report describes the development of a Node-RED flow designed for the real-time processing of ZigBee network packets stored in a CSV dataset. The system periodically generates random IDs and publishes them through MQTT to a local Mosquitto broker (which runs on port 1884). The same flow subscribes to the topic and selects a corresponding packet from the dataset whenever an ID is received using modular indexing.

The packets are then divided into three categories: ZBEE_ZCL frames generate an MQTT message containing a structured payload and, when relevant, selected power, current, and voltage attributes are filtered and saved both to a secondary CSV file and to a live dashboard. Link Status frames are instead used to maintain and continuously update a map of outgoing costs associated with each source node. All other frame types are ignored.

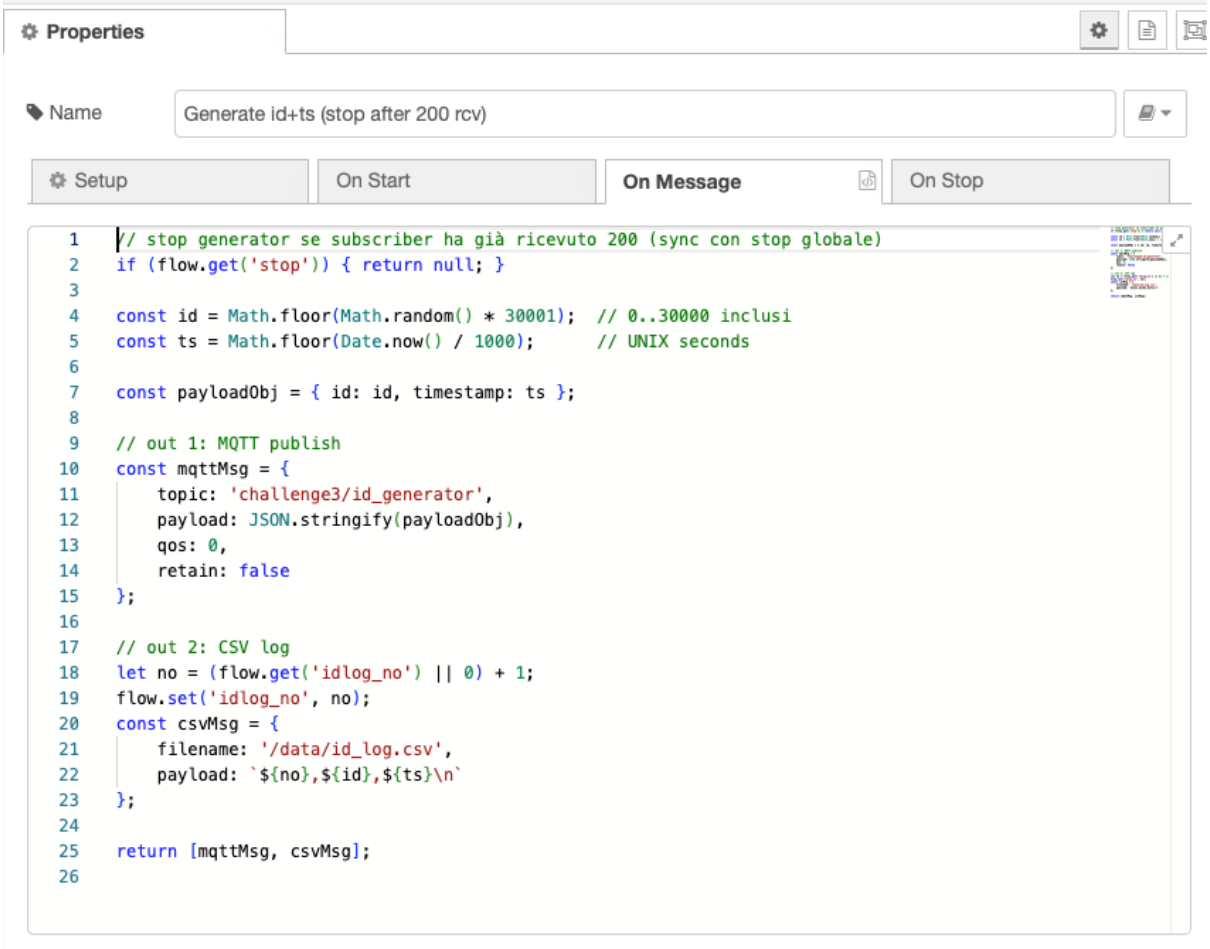
Once 200 messages have been processed, the flow automatically stops execution. At this point, the outgoing-cost map is exported to a CSV file, and the costs associated with the source address having the smallest hexadecimal value are sent to a ThingSpeak channel through its HTTP API, with transmissions spaced 20 seconds apart.

Question #1 – ID Generator & MQTT Publisher

The first section of the flow handles the periodic generation and publication of random ID messages to the local MQTT broker, while also storing them in a CSV file. This part is organized into two sub-flows: an initialization sequence, executed once when the flow is deployed, and a periodic generation sequence, which runs every second.



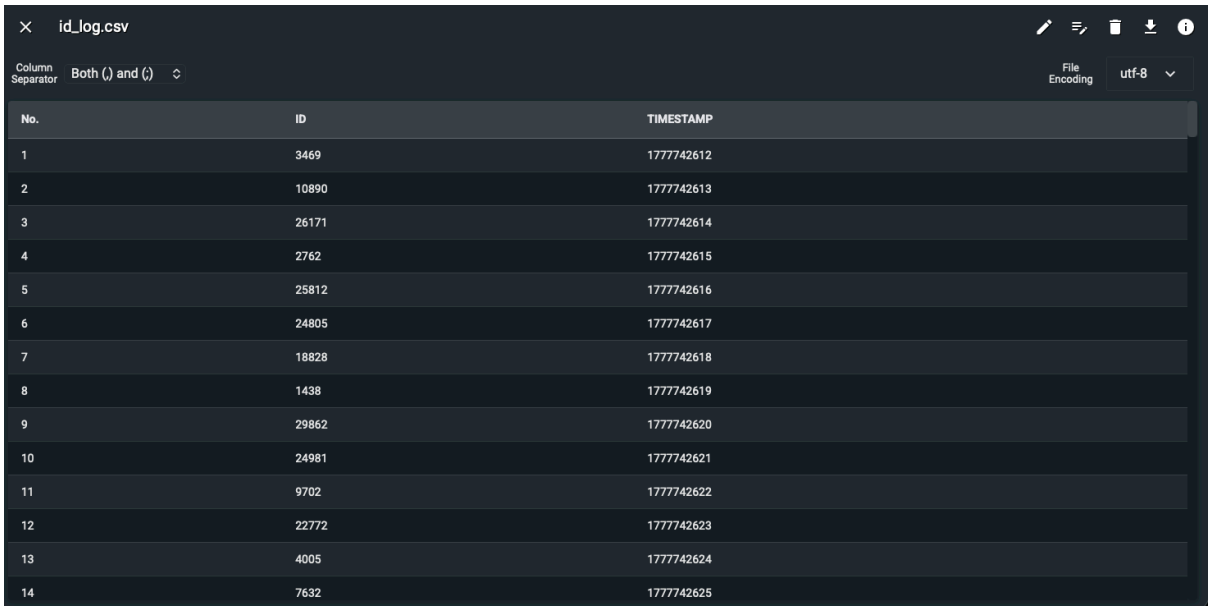
When the flow starts, the Init (once at startup) inject node is triggered once after a delay of 0.5 seconds, activating two parallel branches.
In the first branch, the file challenge3.csv is read from disk through a file in node and converted into an array of objects using the built-in CSV node. The resulting dataset is then stored in the flow context by the Index rows by Packet Number function node, which indexes the entries according to the Packet Number field. This allows constant-time, O(1), access to packets during the subscriber phase.
The second branch, managed by the Reset counters + write CSV headers function node, initializes all the persistent variables stored in the flow context (rcv, idlog_no, filt_no, stop, and linkmap). At the same time, it generates three separate messages, one for each output file, containing the corresponding CSV header and destination filename. After that, these messages are passed to the Write header (overwrite) file node, which writes the headers in overwrite mode so that every execution starts from a clean state.



The Tick 1s inject node is executed every second. Each tick enters the function node Generate id+ts (stop after 200 rcv), whose logic is shown in the image above. At the start of each invocation, the node checks the global stop flag (flow.get('stop')); if set, it returns null and suppresses all output.

The node has two outputs. Output 1 produces an MQTT message with `topic = 'challenge3/id_generator'` and `payload = JSON.stringify({id, timestamp})`, forwarded to the *MQTT pub* node which publishes to `localhost:1884` at QoS 0. Output 2 produces a file-append message containing the CSV row `no,{no}, no,{id},${ts}\n`, forwarded to *Append id_log.csv* which appends to `/data/id_log.csv`.

The following figure shows the first rows of the file that got produced. Each row contains an incremental row number, the generated random ID, and the UNIX timestamp at generation time, confirming the 1 Hz rate (timestamps increment by 1 second per row).



id_log.csv

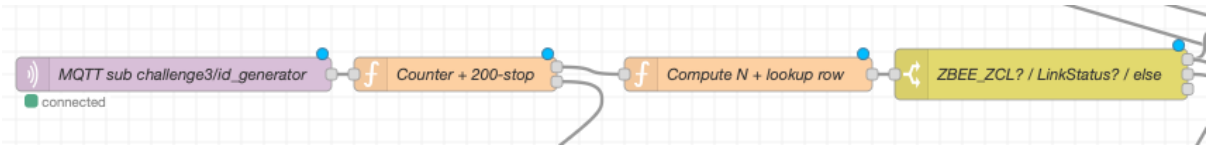
Column Separator: Both (,) and (;)

File Encoding: utf-8

No.	ID	TIMESTAMP
1	3469	1777742612
2	10890	1777742613
3	26171	1777742614
4	2762	1777742615
5	25812	1777742616
6	24805	1777742617
7	18828	1777742618
8	1438	1777742619
9	29862	1777742620
10	24981	1777742621
11	9702	1777742622
12	22772	1777742623
13	4005	1777742624
14	7632	1777742625

Question #2 – Subscriber, CSV Lookup & ZCL Processing

The second section of the flow listens to the same MQTT topic used by the generator. Each time an ID is received, the corresponding packet is retrieved from the dataset, classified based on its type, and then processed following the appropriate branch of the flow. This part is built around three function nodes, a switch node, and two downstream processing branches.



The MQTT sub node subscribes to the topic challenge3/id_generator on the local broker running on localhost:1884. Every incoming message is passed to the Counter + 200-stop function node, which increments the flow-context variable rcv. When the counter reaches 200, the function sets the stop flag using flow.set('stop', true) and sends a trigger through its second output to start the flush phase. In the other cases, the message continues through the normal processing chain.

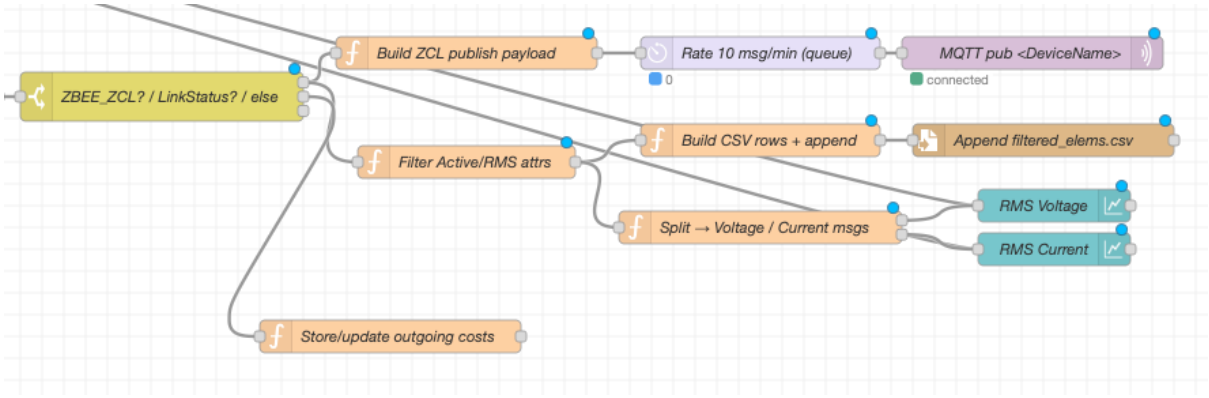
The Compute N + lookup row function node then extracts the id value from the JSON payload, computes the dataset index using the expression N = id % 5218, and retrieves the corresponding row from the flow-context map created during startup. The selected row is attached to the message as msg.row and msg.cmdStr.

Next, the switch node classifies the packet according to the content of msg.cmdStr. Messages are routed to output 1 if the Command String contains "Layer ZBEE_ZCL", to output 2 if it matches the regular expression associated with "Link Status (0x08)", while all remaining packet types are sent to output 3 and discarded.

When the switch node routes a message to output 1, corresponding to a ZBEE_ZCL packet, two parallel processing branches are activated:

1. The first one is the publish branch. Here, the Build ZCL publish payload function node creates a JSON object containing several fields extracted from the dataset and from the received message. These include the current UNIX timestamp, the received subscription ID, the source and destination WPAN addresses (wpan.src and wpan.dst), and the ZigBee source and destination addresses (zbee.src and zbee.dst). The function also generates a topic field formatted as "ZigBee/" followed by the value of Device Name

ZigBee Source, while the payload field contains the packet's Command String.
The outgoing MQTT topic is set directly to the value of Device Name ZigBee Source. Before publication, the message passes through a delay node configured in queue mode, which limits the transmission rate to a maximum of 10 messages per minute. Finally, the message is published to the local MQTT broker through the MQTT out node using the device name as the topic.



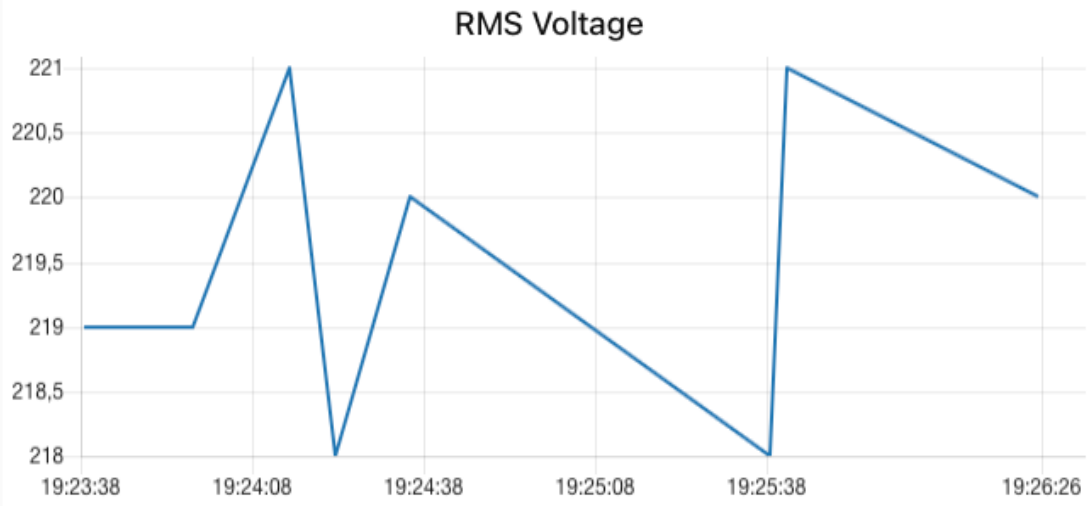
1. The second branch that gets activated by output 1 is the filter branch. The Filter Active/RMS attrs function node parses the Command String field (normalising Python-dict syntax — single quotes, None, True, False — to valid JSON) and iterates over the Attribute list. It retains only entries whose attribute name contains "Active Power", "RMS Current", or "RMS Voltage", pairing each with its corresponding Status, Data Type, and value. The filtered entries are stored in msg.filtered and forwarded to two further nodes in parallel. The Build CSV rows + append function formats each entry as a CSV row (columns: No., Timestamp, Sequence Number, Attribute, Status, Data Type, Data Value) using the persistent filt_no counter for row numbering, and the downstream file node appends the rows to /data/filtered_elems.csv.
   At the same time, the Split → Voltage / Current msgs function fans the filtered entries into two arrays — one for RMS Voltage readings and one for RMS Current readings — which are forwarded to the respective ui_chart nodes (RMS Voltage and RMS Current) for live display on the Node-RED dashboard.

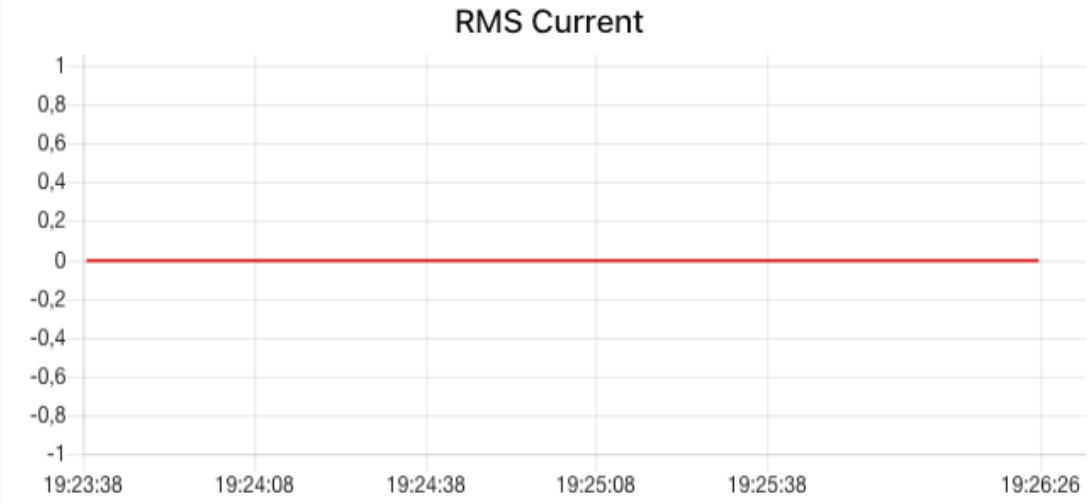
filtered_elems.csv

Column Separator: Both (,) and (;) | File Encoding: utf-8

No.	Timestamp	Sequence Number	Attribute	Status	Data Type	Data Value
1	1777742618	236	Active Power	Success	16-Bit Signed Integer	0
2	1777742618	236	RMS Current	Success	16-Bit Unsigned Integer	0
3	1777742618	236	RMS Voltage	Success	16-Bit Unsigned Integer	219
4	1777742637	132	Active Power	Success	16-Bit Signed Integer	0
5	1777742637	132	RMS Current	Success	16-Bit Unsigned Integer	0
6	1777742637	132	RMS Voltage	Success	16-Bit Unsigned Integer	219
7	1777742654	105	Active Power	Success	16-Bit Signed Integer	0
8	1777742654	105	RMS Current	Success	16-Bit Unsigned Integer	0
9	1777742654	105	RMS Voltage	Success	16-Bit Unsigned Integer	221
10	1777742662	24	Active Power	Success	16-Bit Signed Integer	0
11	1777742662	24	RMS Current	Success	16-Bit Unsigned Integer	0
12	1777742662	24	RMS Voltage	Success	16-Bit Unsigned Integer	218
13	1777742675	122	Active Power	Success	16-Bit Signed Integer	0
14	1777742675	122	RMS Current	Success	16-Bit Unsigned Integer	0

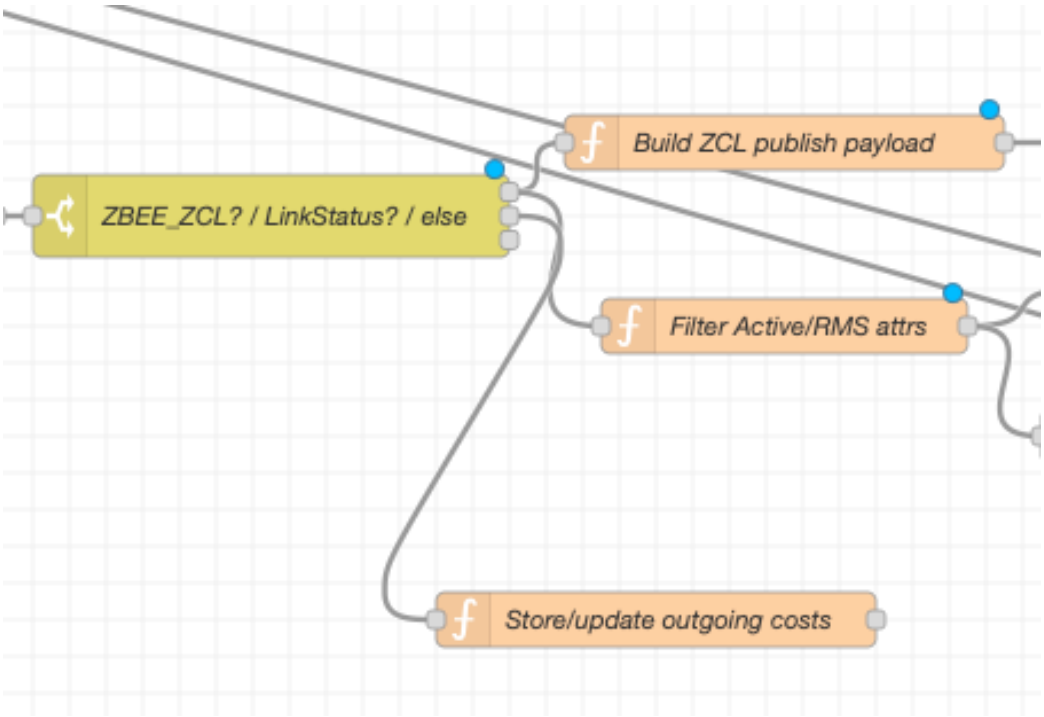
RMS


RMS Voltage
221
220,5
220
219,5
219
218,5
218
19:23:38
19:24:08
19:24:38
19:25:08
19:25:38
19:26:26


RMS Current
1
0,8
0,6
0,4
0,2
0
-0,2
-0,4
-0,6
-0,8
-1
19:23:38
19:24:08
19:24:38
19:25:08
19:25:38
19:26:26

Question #3 – Link Status Processing & ThingSpeak Upload

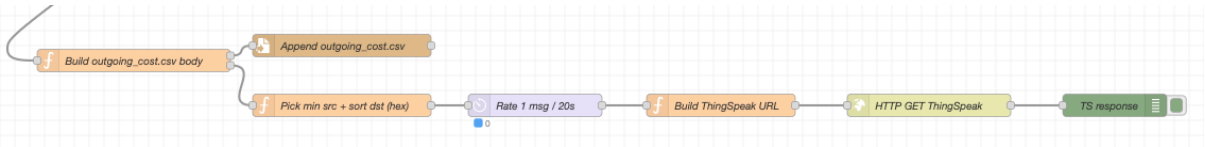
When the switch node sends a packet to output 2, the packet gets recognized as a Link Status frame. At this point, the Store/update outgoing costs function node processes the selected row and updates a persistent cost map stored in the flow context.



The function node parses the Command String of the Link Status packet (converting Python-dict syntax to valid JSON) and reads the Links array. Each element in the array contains a destination address and an outgoing cost. For each link entry, the node updates the flow-context map (linkmap) keyed by the source ZigBee address of the packet: linkmap[src][dst] = cost. If an entry for a given destination already exists, its cost is overwritten with the latest value. The map is thus a continuously updated table of outgoing costs, indexed first by source and then by destination address.

Question #3 – Stop Condition

After exactly 200 messages have been received and processed, the Counter + 200-stop function node sets the stop flag and emits a trigger on its second output. This trigger activates the flush pipeline, which serialises the accumulated linkmap and forwards cost records to ThingSpeak.



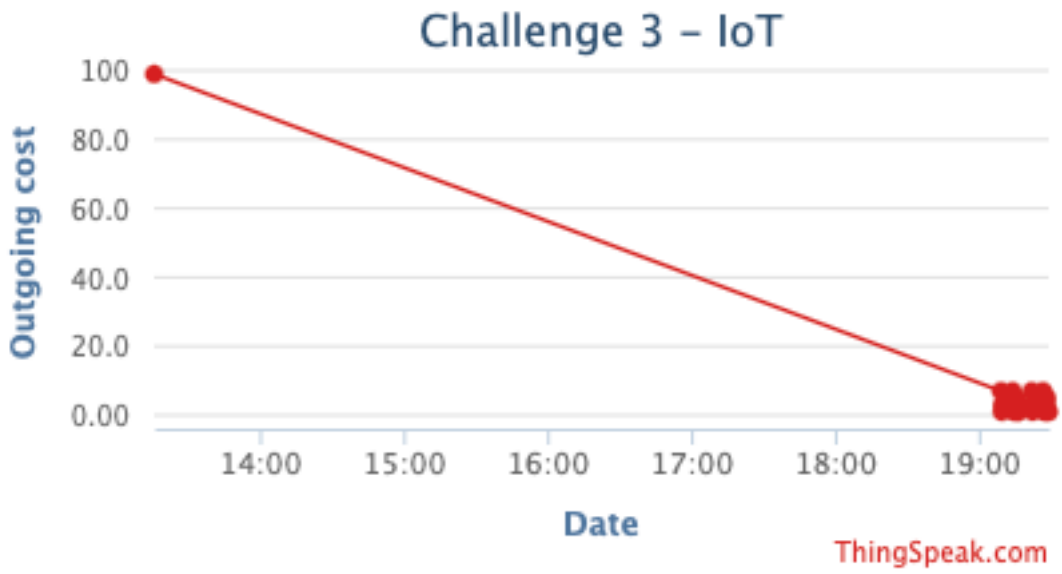
On receiving the trigger, the Build outgoing_cost.csv body function iterates over all entries in linkmap and serialises them into CSV rows (with the following columns: No., Source Address ZigBee, Destination Address ZigBee, Outgoing Cost).
It emits two messages: output 1 sends the CSV body to the Append outgoing_cost.csv file node, which appends the rows to /data/outgoing_cost.csv; output 2 forwards a trigger to the ThingSpeak pipeline. The Pick min src + sort dst function node selects the source address with the lowest hexadecimal value from all keys in linkmap. For that source, it sorts all associated destination entries in ascending hexadecimal order and generates one message per destination. Each message includes the source address, destination address, and link cost in its payload.
These messages are then passed through a delay node configured in rate-limit mode, releasing one message every 20 seconds to comply with the ThingSpeak free-tier restrictions. After this throttling step, the Build ThingSpeak URL function node constructs an HTTP GET request URL using the channel's write API key, encoding the cost value into field1.
The resulting request is sent via the http request node, which forwards the data to ThingSpeak. Finally, a debug node logs the response from the server, allowing verification that the update was successfully received.

outgoing_cost.csv

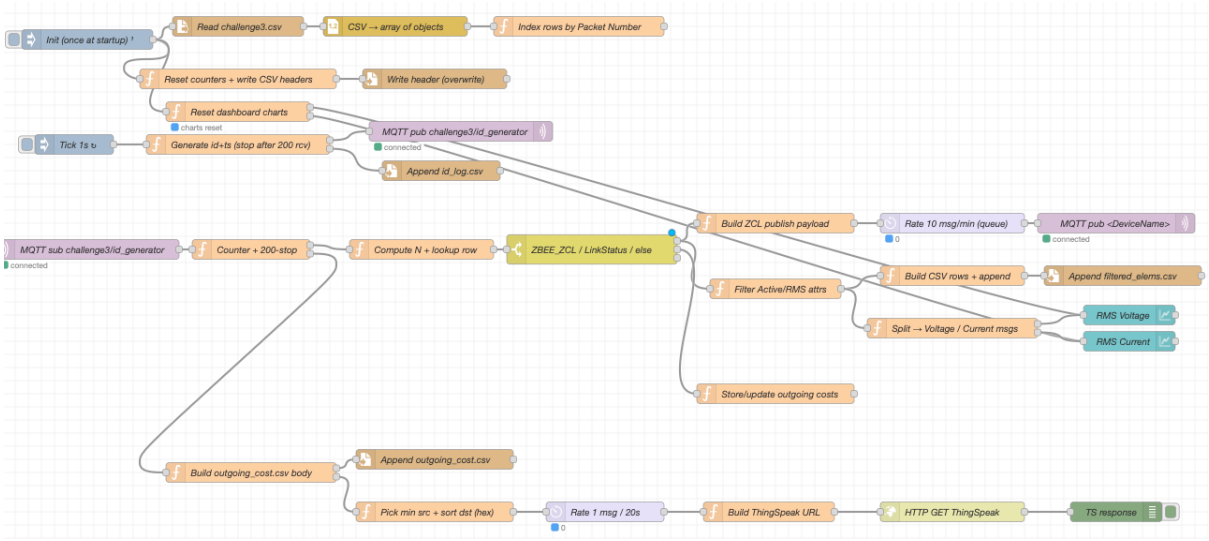
Column Separator: Both (,) and (;) | File Encoding: utf-8

No.	Source	Destination	Cost
1	0xe1a6	0x0000	1
2	0xe1a6	0x0112	7
3	0xe1a6	0x1cd8	1
4	0xe1a6	0x1e15	1
5	0xe1a6	0x3d95	1
6	0xe1a6	0x4e11	1
7	0xe1a6	0xa706	5
8	0xe1a6	0xec7f	1
9	0x0000	0x0112	7
10	0x0000	0x1cd8	1
11	0x0000	0x1e15	3
12	0x0000	0x3d95	3
13	0x0000	0x4e11	3
14	0x0000	0xa706	5

ThingSpeak Channel ID: 3366185
Channel link: https://thingspeak.mathworks.com/channels/3366185

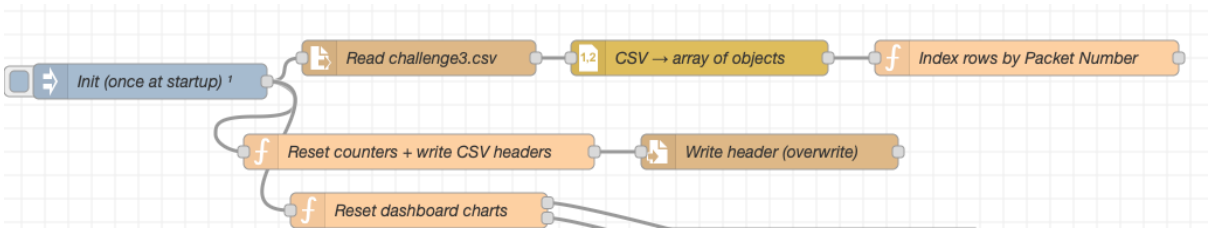


Complete flow description



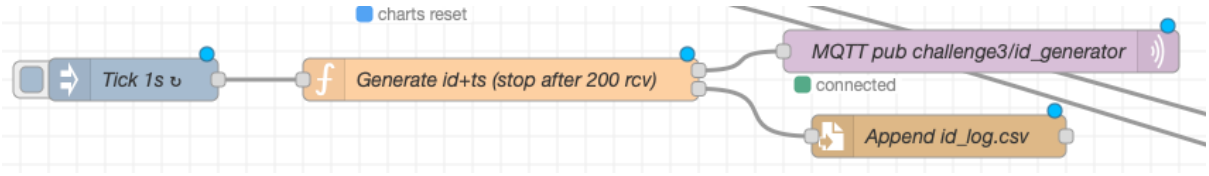
The figure above shows the complete Node-RED flow as deployed. Each node is described below in execution order.

1. ***Initialisation block***



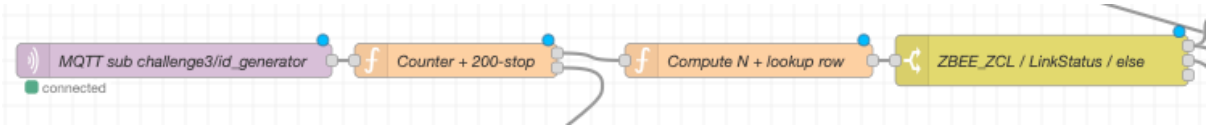
Init (once at startup) — Inject node configured to fire once with a 0.5 s delay at deploy time. It triggers all three initialisation branches simultaneously.
Read challenge3.csv — File-in node that reads /data/challenge3.csv from disk as UTF-8 text and passes the raw string downstream.
CSV → array of objects — Built-in CSV node that parses the raw string into an array of row objects, using the first line as column headers.
Index rows by Packet Number — Function node that iterates the row array and builds a flow-context dictionary rows_by_pkt keyed by Packet Number, enabling O(1) lookup. Also stores the total row count in rows_count.
Reset counters + write CSV headers — Function node that zeroes rcv, idlog_no, filt_no, clears linkmap and the stop flag, then emits three messages (one per output file) each carrying the CSV header string and the target filename.
Write header (overwrite) — File node set to overwrite mode that writes each header message to the corresponding file (/data/id_log.csv, /data/filtered_elems.csv, /data/outgoing_cost.csv), ensuring a clean file on every deploy.
Reset dashboard charts — Function node that emits two empty-array messages (one per output) to clear the history of the RMS Voltage and RMS Current ui_chart nodes so the dashboard starts blank.

2. ***ID Generator block***



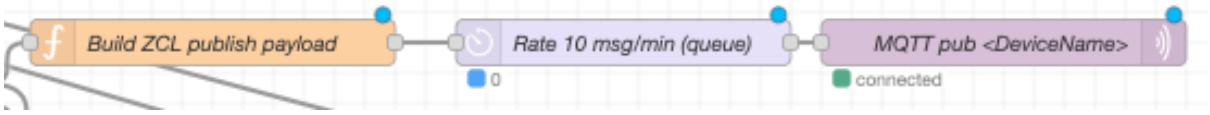
Tick 1s — Inject node configured to repeat every 1 second (no initial fire). Drives the generator at 1 Hz.
Generate id+ts (stop after 200 rcv) — Function node that first checks the stop flag; if set, returns null. Otherwise it draws a random integer in [0, 30000] and captures the current UNIX epoch in seconds. Output 1 emits an MQTT message with topic challenge3/id_generator and payload JSON.stringify({id, timestamp}). Output 2 emits a file-append message with the CSV row (no., id, timestamp).
MQTT pub challenge3/id_generator — MQTT-out node that publishes the message from Output 1 to localhost:1884 at QoS 0.
Append id_log.csv — File node in append mode that writes the CSV row from Output 2 to /data/id_log.csv.

3. ***Subscriber and routing block***



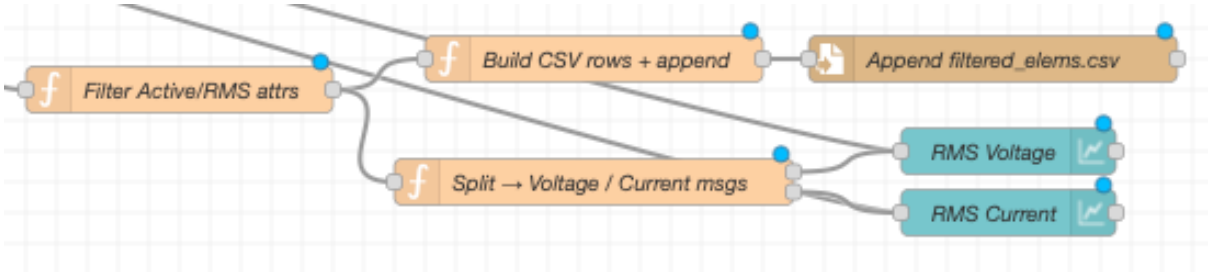
MQTT sub challenge3/id_generator — MQTT-in node subscribed to challenge3/id_generator on localhost:1884, datatype JSON. Receives every message published by the generator.
Counter + 200-stop — Function node that increments the flow-context counter rcv on every message. If rcv > 200 the message is dropped. On the 200th message it sets stop = true and emits a flush trigger on Output 2; for all other valid messages it forwards the message on Output 1.
Compute N + lookup row — Function node that extracts id from the payload, computes N = id % 5218, and retrieves the matching row from rows_by_pkt (falling back to N+1 if not found). Attaches the row as msg.row and its Command String as msg.cmdStr.
ZBEE_ZCL / LinkStatus / else — Switch node that routes on msg.cmdStr: Output 1 if the string contains "Layer ZBEE_ZCL"; Output 2 if it matches the regex for "Link Status (0x08)"; Output 3 (discarded) for all other packet types.

4. ***ZCL publish branch***



Build ZCL publish payload — Function node that assembles a JSON object containing timestamp, id, wpan.src, wpan.dst, zbee.src, zbee.dst, a topic field set to "ZigBee/<DeviceName>", and the Command String as payload. Sets msg.topic to the Device Name ZigBee Source value.
Rate 10 msg/min (queue) — Delay node in rate-limit mode that queues messages and releases at most 10 per minute, preventing broker flooding.
MQTT pub <DeviceName> — MQTT-out node that publishes the throttled payload to the local broker using the device name as topic.

5. *ZCL filter and dashboard branch*

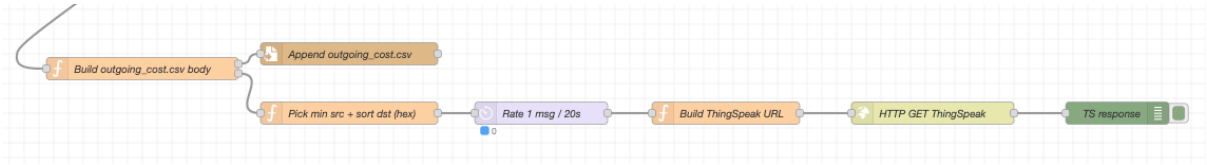


Filter Active/RMS attrs — Function node that parses the Command String (normalising Python-dict syntax to valid JSON) and iterates the Attribute array, retaining only entries for Active Power, RMS Current and RMS Voltage with their Status, Data Type and numeric value. Stores results in msg.filtered.
Build CSV rows + append — Function node that formats each filtered entry as a CSV line (No., Timestamp, Sequence Number, Attribute, Status, Data Type, Data Value) using the persistent filt_no counter.
Append filtered_elems.csv — File node in append mode that writes the formatted lines to /data/filtered_elems.csv.
Split → Voltage / Current msgs — Function node that fans msg.filtered into two output arrays: Output 1 for RMS Voltage readings, Output 2 for RMS Current readings.
RMS Voltage / RMS Current (ui_chart) — Two dashboard chart nodes that display the live readings as line charts on the Node-RED dashboard tab "Challenge3".

6. *Link Status branch*

Store/update outgoing costs — Function node that parses the Command String of the Link Status packet, reads the Links array, and for each entry writes linkmap[src][dst] = cost into flow context, overwriting any previous value. The map is continuously updated throughout the 200-message run.

7. ***Flush and ThingSpeak pipeline***



Build outgoing_cost.csv body — Function node triggered after the 200th message. Iterates linkmap and serialises all entries into CSV rows (No., Source, Destination, Cost). Output 1 sends the body to the file node; Output 2 triggers the ThingSpeak sub-pipeline.
Append outgoing_cost.csv — File node in append mode that writes the CSV body to /data/outgoing_cost.csv.
Pick min src + sort dst (hex) — Function node that identifies the source address with the lowest hexadecimal value in linkmap, sorts its destinations in ascending hex order, and emits one message per destination carrying {src, dst, cost}.
Rate 1 msg / 20s — Delay node in rate-limit mode that releases one message every 20 seconds, complying with ThingSpeak's free-tier limit.
Build ThingSpeak URL — Function node that constructs the HTTP GET URL with the channel write API key and the cost value encoded in field1.
HTTP GET ThingSpeak — HTTP-request node that executes the GET request against the ThingSpeak API.
TS response (debug) — Debug node that logs the ThingSpeak HTTP response to the sidebar for verification.