TRAP debug tool for Transiently-Powered sensing systems

Department of Industrial Engineering
Mechatronics Engineering course

Laboratory of Internet of Things

Prof. Davide Brunelli
dept. Industrial Enginineering
University of Trento
Povo, Trento
davide.brunelli@unitn.it

Ph.D. Alessandro Torrisi
dept. Industrial Enginineering
University of Trento
Povo, Trento
alessandro.torrisi@unitn.it

M.S. Matteo Spadetto
dept. Industrial Enginineering
University of Trento
Povo, Trento
matteo.spadetto-1@studenti.unitn.it

***Abstract*—This report is treating the debug tool used for testing the TRAP (TRAnsiently-powered Protocol) for intermittent communication among transiently-powered devices. This technology, as it will briefly explained later, needs a monitoring application which is able to track and store the messages of an IoT (Internet of Things) network. This report will explain how the tool concept was tested and upgraded in order to achieve the goal, reach the highest possible level of scalability and make the TRAP test procedure much more user-friendly.**
As last thing, due to the fact that the application will be run on a remote device by different users, also a discursive explanation on how to use the scripts is proposed.

***Index Terms*—TRAP, Transiently-Powered, Debug, Tool, automation, scalable, Node-RED, Grafana, Python3, Bash, Linux**

I. Transiently-Powered sensing systems

With battery-less wireless sensors, the communication is guaranteed only if in the transmission instant both the sending node and the receiving one have enough energy to share data between each other [1]. This is not an obvious condition for such sensors, meaning a potentially high loss of messages and a waste of energy.
To better understand the Transiently-Powered communication problem, an example could be useful: if a node reaches the energy threshold level with which it is able to send the data but the receiving node is not at the same or higher level, the message will not be received. Moreover, this problem is not detectable from the sending node which cannot change its behaviour accordingly with the loss of information.
For this reason the new protocol TRAP (TRAnsiently-powered Protocol) for intermittent communication among transiently-powered devices was developed. Thanks to the energy status information, TRAP verify that both sender and receiver have enough energy before the transmission starts in order to ensure a successful communication without loss of data.
This kind of systems cannot use technologies such as active radio due to their high energy cost so back-scatter communication is used (the same implemented also in RFID tags) which enables an almost zero-power communication. Using this kind of technology, the nodes share the energy status between each others and, if it is considered higher enough for both sender and the receiver, the radio channel is used to transmit the message.
With a more practical example, what TRAP is able to do, is to know the energy state of the neighbour nodes and use this

information to decide if the message can be sent or must wait until another node has enough energy to receive data. Thanks to this approach, there is no more loss of information due to the power deficiency.
As it is possible to imagine, the system could be difficult to test without a proper tool condensing all these information, which are related to different nodes, in few and simple graphs and tables easy to understand. For this reason the debug tool treat in this report was born.

II. The goal

The Transiently-Powered technology is very interesting in network of battery-less devices. For this reason, tests require to monitor the state of a lot of this kind of sensors simultaneously. The requirements set at the beginning of this project were the following:

- the application has to monitor the energy level of each device;
- the application has to monitor the status (or control flags) of each device;
- at the beginning the application has to monitor just few devices but it should be easy scalable to multiple devices;
- the application should be easy to use in order to spend few time to setup each test;
- the user, due to the long time the test are intent to be, should have the possibility to monitor the system running in remote mode: for example from home when the system is running in the laboratory;
- the user should be able to import the collected data in an analysing software such like Matlab.

III. Test system

To completely understand how the goal was achieved, the development was divided into two phases:

- first of all, a first version of the system was designed, including just the the fundamental requirements;
- as upgrade, a second and definitive release was coded, thanks to two plugins wrote in *Python3* and Linux *bash*.

Analysing the first point, the one that will be discussed in the next sections, it is possible to understand how the system works. While, studying the upgrade features, the explanation moves on how the tool was improved in scalability and automation.
The basic elements of the debugging system are:

- the *Node-RED* flow which is involved in collecting, parsing and storing data;
- the InfluxDB database which stores the data and their relative timestamps while it also connects the *Node-RED* flow and the *Grafana* visualiser;
- the *Grafana* visualiser that gives an easy and efficient way to understand data;

In this section just one device as input was considered, then, in the upgrade section the scalability will be improved.

A. Node-RED

Node-RED [2], is a browser based flow editing, with which it is possible to develop JS (JavaScript) code in a easy way. Thanks to the predefined nodes, it is possible to overcome the difficulties related to manually programming some JS functions. Very often, just a drag and drop is required. Even if this method could seem very useful, it implicitly leads not so much versatility: for this reason also function nodes are available giving the possibility to manually program in JS.

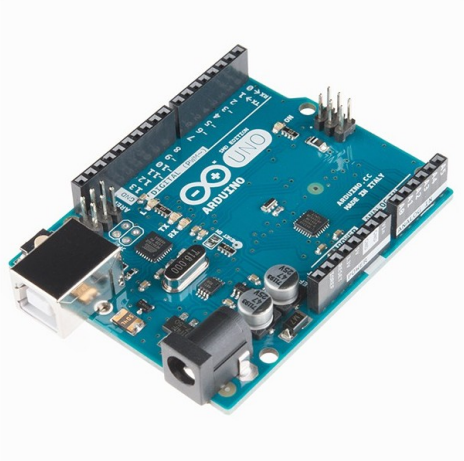

Fig. 1. Arduino UNO board used to simulate the sensor input

As it is possible to see in figure Fig.1 and Fig.2, the *Node-RED* system for this debug tool is composed by:

- the input node which is, in this case, just reading from an uart port connected to an Arduino UNO. This is sending a simulated sequence of messages collected during laboratory tests. It is important to remember that this is the earlier stage of the application, which just had to work correctly reading just simulated data to test the feasibility.

  The messages sent are one byte long, written as hexadecimal numbers, and they are summarising two different kind of information. They can be related to the energy percentage value (all integer numbers between 0 and 100), labeled as *engy*, or to predefined control labels:
  - 111: burst message received;
  - 112: the device transmitted the message;
  - 113: the device received the message;
  - 114: failed receiving the message;
  - 115: postponed message transmission;
  - 200: debug control.
- the *uart_id_parser* node which takes as input the byte of the message and, based on the integer number described

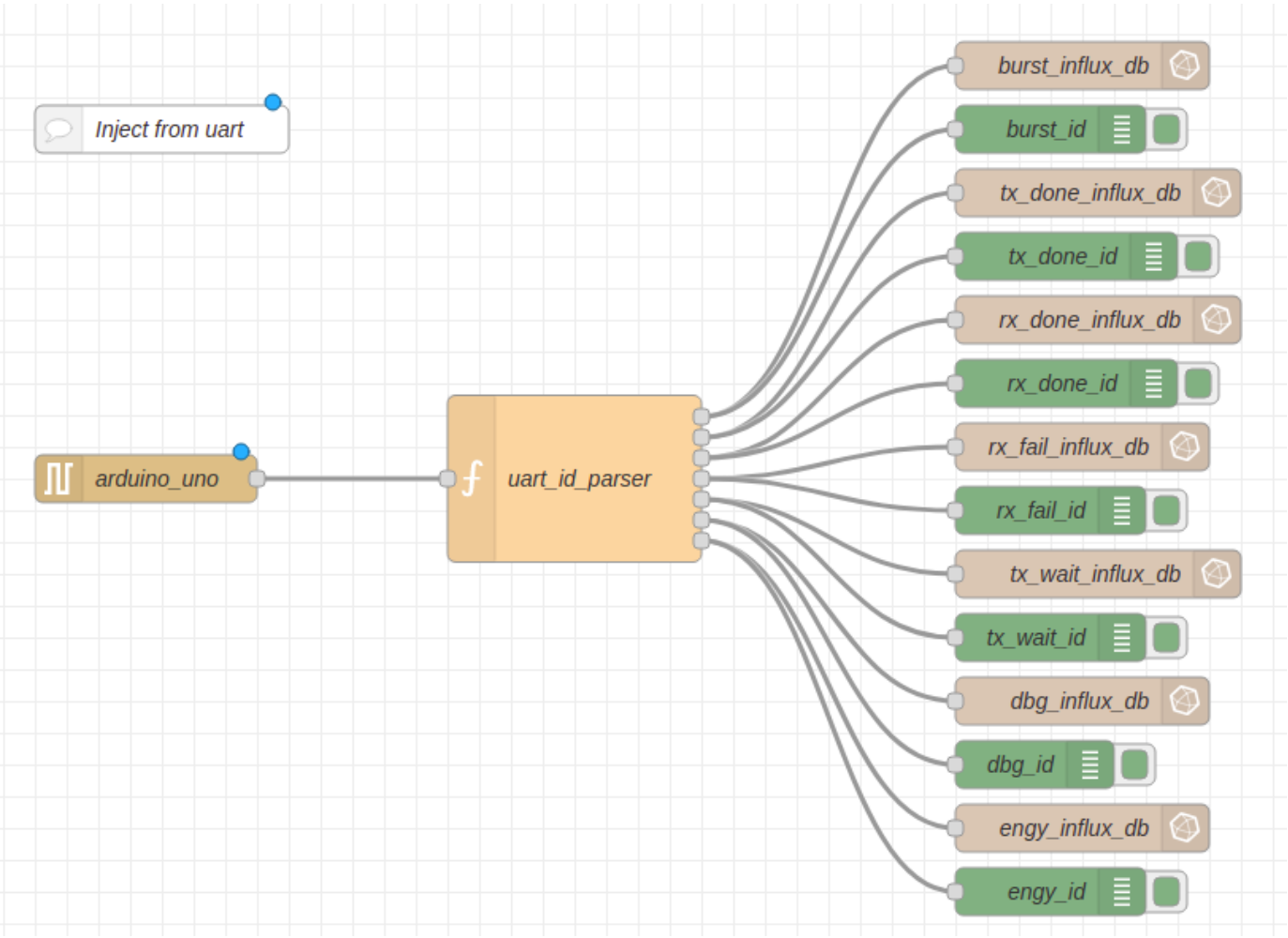

Fig. 2. *Node-RED* feasibility test flow with single input sensor simulated by an Arduino UNO

in the previous list, sends the byte to the relative InfluxDB node. This algorithm is based on a simple "switch-case" parsing all the received data.

- the InfluxDB nodes, storing all the data in the database. On the base of which node is fired by the *uart_id_parser*, the message is saved in a particular measurement and field of the database in the modality which will be shown later.

To make a simple example of the *Node-RED* system: if the Arduino UNO, reading the predefined test buffer, sends the byte $0x6F$ corresponding to the integer 111, the *uart_id_parser*, receiving this data from the uart node, sends the value 111 to the InfluxDB node relative to the burst messages.

B. InfluxDB

The InfluxDB [4] database role is to store the information in the correct measurement and field, with the relative timestamp, when the integer value is received from the *Node-RED* flow. To understand the structure chosen for the database the following table Fig.3 is useful.

As it is possible to see the measurements are:

- *engy* for energy value;
- *burst* for burst control flag received;
- *tx_done* for transmission done;
- *rx_done* for message received;
- *rx_fail* for failed receiving message;
- *tx_wait* for postponed message transmission;
- *debug* for debug control flag.

Then, in each measurement, a field relative to the device is created. In this way it is possible, for example, to find in the measurement *engy*, all the energy levels related to *device_0* in *engy_0* and the ones related to *device_1* in *engy_1*.

C. Grafana

Grafana [3] is used as database information visualiser. After accessing InfluxDB measurements, it displays, for each separate device a complete row. Each row is composed by a graph plotting all energy levels (and the threshold to overcome

InfluxDB database	iot_tool						
Mearuements	engy	burst	tx_done	rx_done	rx_fail	tx_wait	debug
Fields	engy_0	burst_0	tx_done_0	rx_done_0	rx_fail_0	tx_wait_0	debug_0

Fig. 3. InfluxDB database used for single input sensor

for the correct communication), collected from the specific device, and six single_stat counting the element of the field of the measurement relative to the device.
In addition to the information relative to each single device, also the totals are shown in the first row (but not the ones relative to the energy level because those are just percentage). In the following demo Fig.4 it is possible to see a preview of the *Grafana* interface.

IV. Automation and scalability issues

One of the goal of the project is to give the possibility to monitor not just one node at time but several of them in parallel.
To do this, the first approach was to manually replicate the system explained in the previous chapter for four different devices. This was possible implementing some new features:

- more than one uart node are used. These ones are connected to a function which set the proper labels used for storing the data related to the specific device: for example *device_0* will have the labels set as *rx_done_0*, *tx_done_0*, etc. Then, the data is sent to the parsing function which, as in the single input case, transmits the messages in the right measurement and field of the database.
- the database has the same structure of the single device state but each measurement contains different fields (one for each device): for example measurements *engy* will contatin *engy_0*, *engy_1*, etc. Fig.5.
- in *Grafana* the row containing the totals is maintained but the one related to the single device is replicated for each input and connected to the correspondent field of the InfluxDB database.

As it is possible to read, this procedure requires that the user changes manually a lot of parameters and implement new channels every time a new device is added to the network. To avoid all these recurrent tasks during the tests and give to the user a better experience, reducing the experiment setup time, some upgrades were implemented as it is possible to read in the next two sections of this report.

A. Python3 code

The concept behind the automation system is to give the possibility to do the long and repetitive work of the setup phase to the software. This will set all the necessary structures just receiving as input all the ports to which the devices are connected.
The use of *Python3* was very useful: from the moment the script must just set the parameters of the system, it has not to respect some time constraints as the overall system should. As second reason *Python3* is a very easy and versatile programming language giving good results with few lines of code and so less developing time.
The script is divided in several functions for which a brief description is provided here but more detailed comments are written on the `constructor.py` script itself:

- *custom_split* function is used to rearrange the strings of the `.txt` files, in an array of strings. The deliminator of the strings are "!" symbols which are placed in the file used as templates. To better understand, both *Node-RED* and *Grafana* are just running a `.json` file containing all the information the user sets in the relative browser graphic tools. In other words, it is possible to directly modify the `.json` files to produce different *Node-RED* flows and *Grafana* visualisers. To do this, the *custom_split* function create the array of strings that the following functions are modifying to add new nodes to the flow or new rows to *Grafana*;
- *node_opt_mode* creates a new uart settings object for the node uart that will be generated next;
- *node_uart* creates a new uart node with the settings specified by the *node_opt_mode* function;
- *node_func* creates the function node that is responsible to set the correct labels for the specific device that will be used to store the data in the measurements;
- *new_device_node* creates a full *Node-RED* sequence composed by the uart node and its settings and the function related to the specific device;
- *row_build* function creates a new *Grafana* row (without tables in it);
- *table_build* function creates a new *Grafana* table which can be a graph or a single_stat;
- *new_device_grafana* creates a full new *Grafana* device row with one graph table for *engy* measurement and six single_stat for all the others. Then, it relates all the tables to the field correspondent to the device.
- *new_device* creates a full new device calling both *new_device_node* and *new_device_grafana*.

It is important to understand that, particularly in *Node-RED*, the concept is to build flows. For this reason it is also important to set and pass the node IDs to each subsequent component. It is also fundamental to use the `.json` structure used by the *Node-RED* and *Grafana* architecture: for this reason templates are used. The first section of this report explained how to get one complete flow and row to debug one single device and, thanks to the possibility of exporting the `.json` of those

Fig. 4. *Grafana* visualiser used for feasibility test with single input sensor simulated by an Arduino UNO

InfluxDB database	iot_tool							
Mearuements	engy		burst		tx_done		rx_done	
Fields	engy_0	engy_1	burst_0	burst_1	tx_done_0	tx_done_1	rx_done_0	rx_done_1

Mearuements	rx_fail		tx_wait		debug	
Fields	rx_fail_0	rx_fail_1	tx_wait_0	tx_wait_1	debug_0	debug_1

Fig. 5. InfluxDB database structure, case with two sensors

nodes and visualiser, it was possible to understand and manage the basic structure. In fact, thanks to the "!" symbols placed in the correct points of the `.txt` templates files (the `.txt` file is just a conversion of the exported `.json`) it is possible to generate two new `.json` to import in *Node-RED* and *Grafana* for running.

Thanks to this method it is possible to add, theoretically, infinite devices. The only input arguments needed are the ports to which the devices are connected.

Until now just the *Node-RED* and *Grafana* upgrade was treated but to make the overall system working autonomously also some upgrades related to the InfluxDB database are needed. For this reason, in the *Node-RED* structure some nodes were added as it is possible to see in the next figure Fig.6:

- execution node which is responsible to drop the old database used for the previous running;
- execution node that creates a new database in InfluxDB to store the new data;
- execution node to export, in `.csv` files, all the data stored in InfluxDB. The script generates one `.csv` file for each measurement of the database. This sub-flow is present also in *Node-RED* but used there only for tests. The more efficient way to generate the `.csv` files is to type the command in the terminal window once the experiment is finished.

All these execution nodes was needed from the moment that the *node-red-contrib-influxdb* plugin is limited. This nodes are used, as it will be shown soon, by the device in which the scripts are running (for example the Raspberry Pi collecting the data from the external sensors).

B. Linux bash code

Due to the fact that is not only necessary to set the proper *Node-RED* and *Grafana* `.json` files but it is also fundamental to access and manage the InfluxDB database in an automated and user-friendly way, a Linux *bash* file is used. In particular, the *manage_influx_db.sh* script is used to:

- show all databases;
- select the database to use;
- drop the selected database;
- create a new database;
- show all the measurements of the selected database;
- show all fields of the specified measurement;
- drop the selected measurement.

The last and most important command the user can use is:

```
$ ./manage_influx_db.sh -r
↪  '/dev/ttyACM3 /dev/ttyACM4'
```

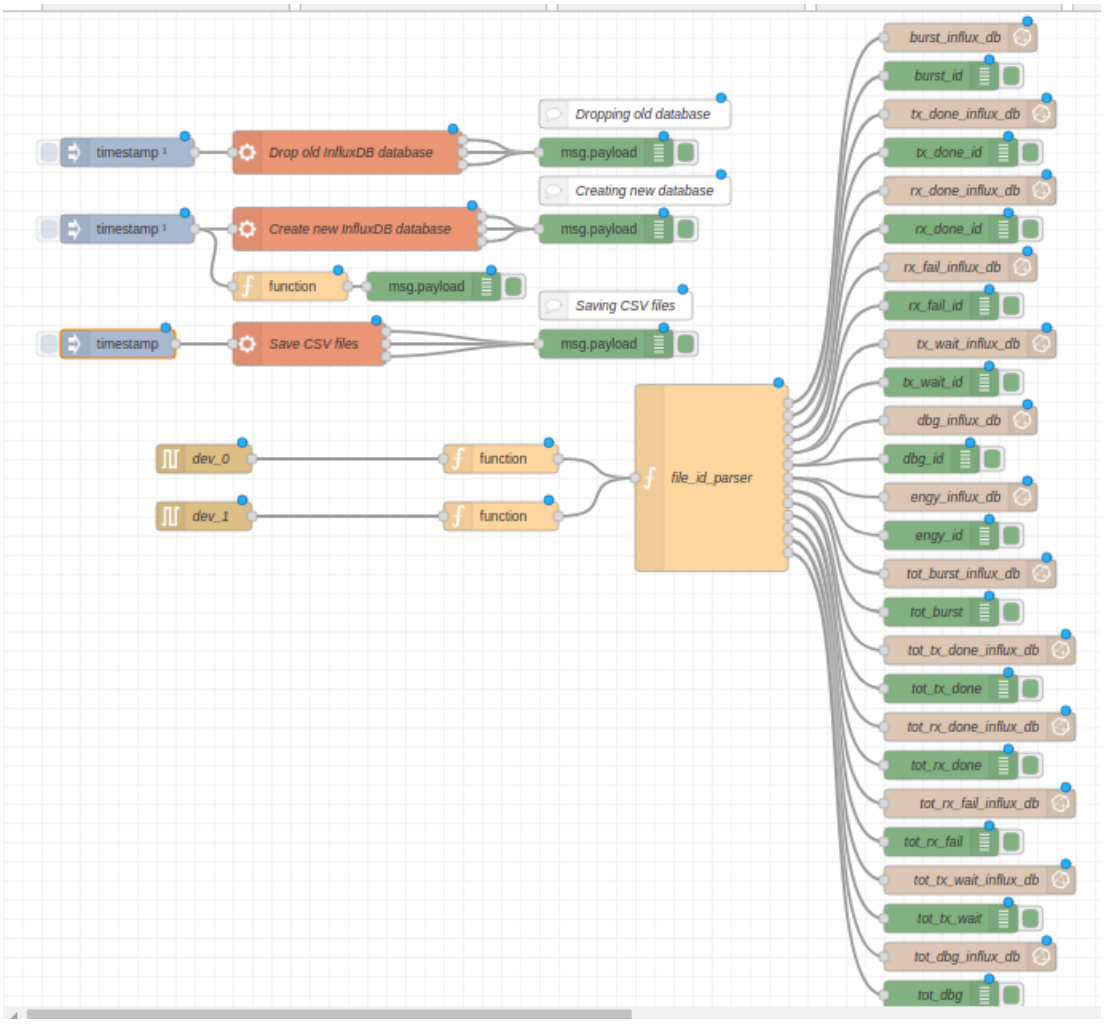

Fig. 6. *Node-RED* flow, case with two sensors

This is the command used to run the entire system just passing as argument all the ports on which the devices are connected. With this command the *bash* file is performing the following actions:

- it stops the grafana-server opened session, if it exists;
- it calls the *Python3* script `constructor.py` which generates the `.json` files for *Node-RED* and *Grafana* with the requested ports to debug;
- it starts again the grafana-server session. Thanks to a change in the `grafana.ini` file at the line

  ```
  default_home_dashboard_path =
  ↪  /path/to/use_grafana.json
  ```

  the visualiser, when opened, will automatically start the `.json` file generated by the *Python3* script;
- it starts the *Node-RED* session running the new `.json` file generated by the `constructor.py`

When the test is ended it is also possible to download all the `.csv` files, one for each measurement contained in the database. This is important to do before starting a new test because the new session will drop the old information. It is also possible to not overwrite the old database just modifying the name of the database in the *Node-RED* `.json` file before running it. The choice was to overwrite the database because, when the `.csv` files are saved, it also possible to recreate the old structure. Furthermore, it makes the overall system more versatile from the moment it can be run with just few arguments.

V. Overall system: Upload, setup and run

From the moment this application will run on a remote device, as a Raspberry Pi Fig.7, it is important to understand how to setup it and use it with `ssh` commands. The user-experience is divided in three different parts that will be treated now: upload, setup and run.

As initial assumption, the tests and the *bash* script was developed for Linux based system: this means that they will run only on those systems unless the *bash* file described before will not be write also for Windows and Macintosh.
Starting from the first step, if not already done, `ssh` must

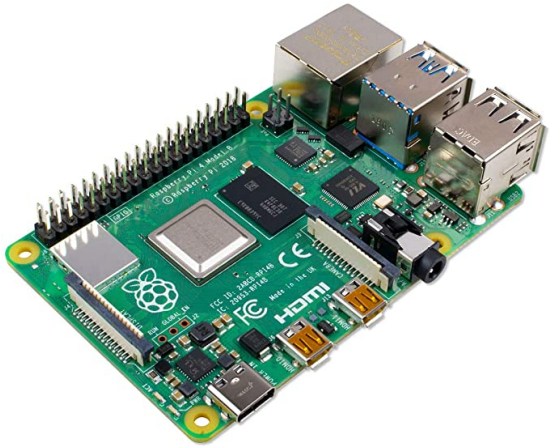

Fig. 7. Raspberry Pi board

be installed to be able to connect the personal computer to the remote device, in this case the Raspberry Pi. Also on the remote device it could be necessary to enable the `ssh` option: for example in Raspbian [5] this is done by typing on terminal:

```
sudo raspi-config
```

and searching for `SSH` in `Interfacingoptions`. Then, connecting via SSH to the Raspberry Pi, the following dependencies should be installed:

- *Node-RED* and the plugins: *node-red-contrib-influxdb* and *node-red-node-serialport*;
- *Grafana*;
- *Python3*.

After that, it is necessary to download the zip folder of the project and copy the archive to the remote device with an `scp` command. After it is unzipped, the user should check in the template files, used from `constructor.py`, for wrong paths due to the fact they are now imported in a new device.

Once this steps are done, it is possible to set in the file `grafana.ini` the lines:

- `default_home_dashboard_path = /path/to/use_grafana.json`
  to start the right `.json` file just opening *Grafana* at the port 3000 of the device IP address;
- `min_refresh_interval = 200ms`
  to set the visualiser refresh rate to 200ms.

Now it is possible to connect to the remote device all the sensors to the debug them via USB port, or a different UART connection, remembering to give all permissions to the user for reading the inputs with the command:

```
sudo chmod -R 777 /dev/
```

It is possible to say that the upload and setup phase is ended and the run command can be executed via ssh:

```
ssh pi@xxx.xxx.x.xxx:
 ↪  /path/to/manage_influxdb.sh -r
 ↪  '/dev/ttyACM3 /dev/ttyACM4'
```

which returns as output Fig.8.

This command is, as explained in previous chapter, running both *Node-RED* and *Grafana* on the remote device at, respectively, `localhost:1880` and `localhost:3000`. For this reason, just connecting to the remote device IP address, at those ports, it is possible to see the flow running and the behaviour of the sensors in an online mode.
It is important to remember that, as mentioned previously, before running again the application, the command:

```
./manage_influx_db.sh -w
 ↪  /path/to/folder/
```

must be used to export the `.csv` files, otherwise the data will be overwritten.

VI. Conclusion

This report explained all the process used to develop the debug tool for testing the TRAP protocol.
The first chapters were focused on how to satisfy the basic requirements, such as monitor the energy level and all the control flags of a single battery-less sensor thanks to *Node-RED* and *Grafana*.
After that, it was fundamental to increase the scalability level to make this tool able to satisfy the experiment upgrades of the following months. To do this also the automation level was increased with a *Python3* script and a *bash* file that are responsible of managing the generation of new `.json` objects in order to handle new sensors, both in *Node-RED* and *Grafana*.
With this application, the time to setup the test environment is decreasing significantly but the first time the dependencies must be installed and the files have to be imported into the remote device. This process could be simplified in future developing an installation script which is responsible of downloading all the required files and programs on the Raspberry Pi.
The tool is dependent just by the fact it must run on a Linux based OS (Operative System). This limitation, as mentioned before could be overcame just writing a *bash* file `manage\_influx\_db.*` for Windows or Machintosh.
Thanks to the possibility to download the `.csv` files of all the data stored in the database, it is also possible to copy the entire folder from the remote device to the local personal computer to import it in some analysis tool as Matlab.
In conclusion, the goals set were met but there are also some helpful features that can be added in order to:

- make the first setup easier and faster;
- make no need of arguments to run `constructor.py` and so the `manage\_influx\_db.sh` leaving to the software the detection of the port on which the sensors are connected;
- give the possibility to set every time a different database in order to not overwrite the old one;
- give the possibility to run the application also with different OS.

These are not difficult tasks to implement but they require an evaluation of the trade-off they creates between the efficiency and complexity.

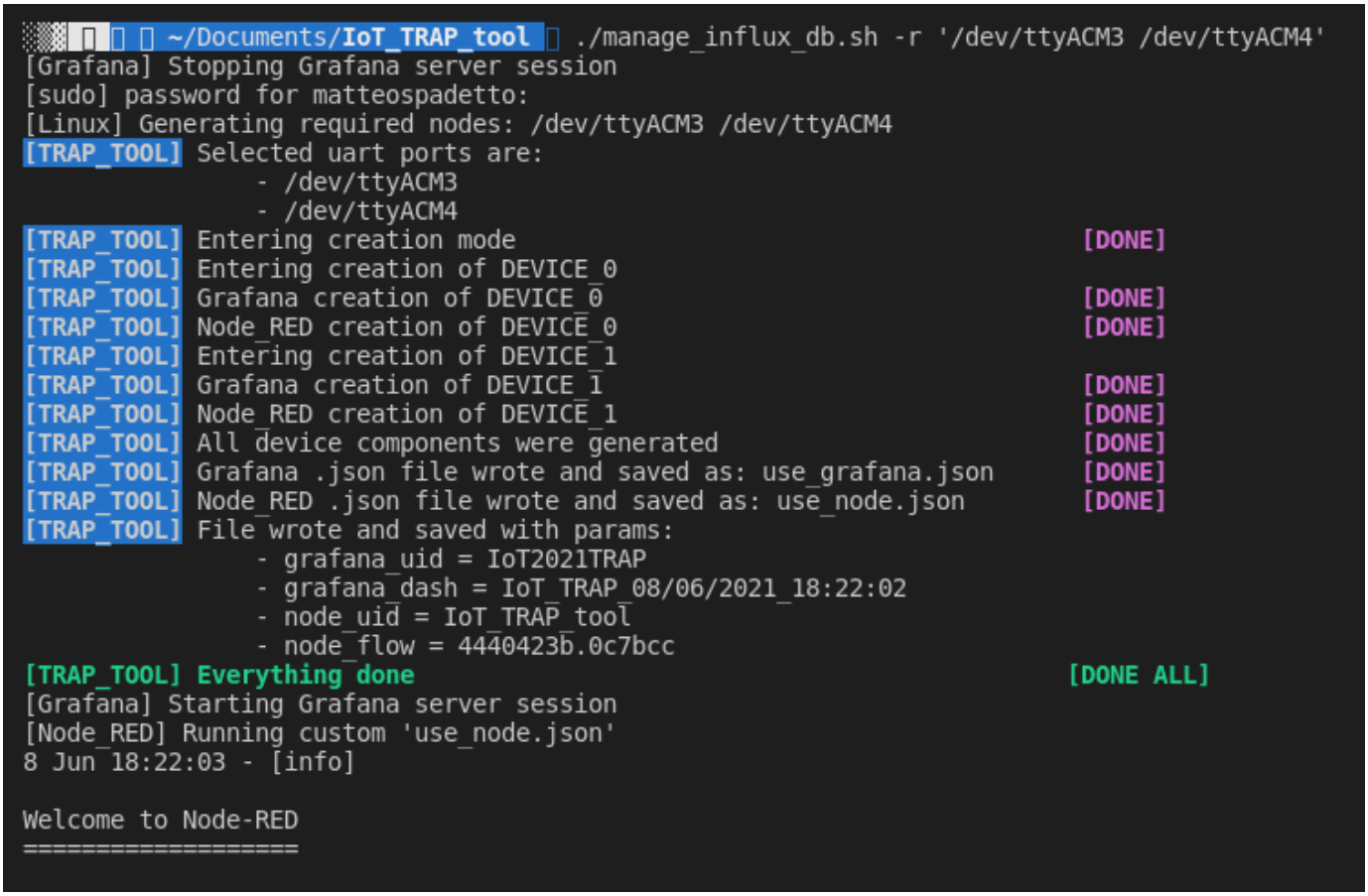

Fig. 8. Output of the application run command

REFERENCES

[1] A.Torrisis, D. Brunelli, K. Sinan Yildirim, Zero Power Energy-Aware Communication for Transiently-Powered Sensing Systems. Trento, IT: University Trento, November 2020.
[2] https://nodered.org.
[3] https://grafana.com.
[4] https://www.influxdata.com.
[5] https://raspberrypi.org.