POLITECNICO
MILANO 1863

Internet Of Things
A.A. 2022-2023

Project Report

Implementation of a 802.15.4 beacon-enabled personal area network with wokwi and node-red

Author

Matteo Bevacqua

Person code 10656748

Matricola 100819

Contents

1 Design Choices

1.1 Implementation Challenges

The following section highlights some of the challenges which have been faced during the implementation of the project and how they've been resolved. Most of them are due to wokwi's questionable choice of running in a browser.

1.1.1 Wokwi's simulation speed and random delays

Unfortunately, wokwi's simulation speed prevents us from using particularly accurate hardware timers, for example **hw_timer_t** with periods smaller than 1 millisecond. Due to this, the PWM handler of the actuators has been moved to the first core of the mote (number 0) which is dedicated to flipping on and off the led as per the pulse modulation period, which is 100 ms. The task waits (on a semaphore, the ones provided by **freeRTOS.h** library) until the first data packet coming from a sensor is received and will keep executing until the mote is switched off. [1]

1.1.2 Unpredictable latency from the broker

The MQTT messages might have unpredictable latency peaks, one of the reason for which QoS 0 has been preferred as the default quality of service level on the channel. This has had an impact on the size of one slot of the beacon, especially since in earlier versions the PAN coordinator featured a module that estimated the RTT to the motes and used said value to set the duration of a slot. However, due to said unpredictable peaks this module has been deprecated and the value of one slot set to a fixed value of **500ms** which is way more than enough in the average case. This value has been set experimentally and tested in around 20 simulations with a full network, it's both large enough to deal with additional latency and small enough to not cause a huge beacon interval (13s with 4 motes connected).

1.1.3 Implementation of the duty cycle

A real RFD participating in a 802.15.4 PAN would wakeup, send/receive messages and sleep for a certain amount of time regulated by its duty cycle. In order to keep the simulation time reasonable, two choices have been made in that regard:

1. No "real" sleep in the motes
   Wokwi cannot emulate the **light_sleep** mode of the ESP32 board and using **deep_sleep** (which would turn off everything radio-related and some more) would double the code size adding unneeded obscurity to the firmware of the motes while also causing a crash in about 20% of the simulation runs. As such sleep is "emulated" using the **delay** function which stalls the cpu for a certain amount of time. While far from being an optimal, nor correct solution in real world applications it works quite well for a simulation scenario. In order to avoid confusion, it should be noted that unfortunately **delay** is also used (rightly so this time) for busy waiting during the active part of the beacon.

2. A relatively high duty cycle
   The duty cycle is set to about 50% in the coordinator, in order to not have to wait minutes to see things happening in the simulation. Starting from the CFP, whose size can vary as motes associate, the amount of inactive slots is calculated to keep the 50% duty cycle goal.

1.1.4 Mote-Coordinator Synchronization

As explained in 1.1.1, wowki's simulation speed does not allow for an exact (up to the microsecond) accounting of the passing of time. As such all the time intervals are not measured in absolute terms (i.e. setting the date and time of the motes and using an exact date to account for the beacon interval) but relatively to the arrival of the beacon message. Given all the other additional latencies that have to be considered (JSON processing at the motes, network delay etc...), a tolerance period is added every time, which after some trial and error has been set at **the size of one slot**. The motes will "wakeup" (stop busy waiting) 1 slot before the beacon is set to arrive and run **mqtt.loop()** until it is received.

[1] The motes include some additional hardware which wasn't requested, however it's not the cause of the additional latencies in the simulation, all of these issues were present even when the hardware was the bare minimum and have not been exacerbated by it.

1.2 Known Issues

During long-running simulations [multiple hours] (especially with a full network) wokwi's motes will most likey run out of sync by a couple of seconds (highly dependant on the CPU)[2], however there is no way of fixing this on my part. Thankfully, using QoS 0 will cause the motes to re-sync themselves on the latest beacon without additional work needed on our part. Messages that are lost due these simulation latencies will just be retransmitted until they are acknowledged.
For similar reasons messages sent in the uplink slots of beacon n° i might not be transmitted during the downlink of the same beacon. This might happen because the downlink handler is triggered **exactly** when the CFP downlink phase starts and it is assumed that all the messages from the motes have already been received and stored in the message queue. Unfortunately wokwi's gateway or the HiveMQ broker might have random spikes of delay (measured up to 1.5s) that cause the downlink handler to see an empty queue since uplink messages haven't arrived yet. In order not to **violate the structure of the beacon** (and thus send messages at random times) this message will be stored and sent during the next beacon. To give a brief example of what this event might look like in the simulation here's a snippet of the PAN coordinator's output:

```
Sending beacon message - [2023-07-03 11:57:22] -  seq : [16]
PAN coord. received an uplink data message from mote 1 - seq : [2] - dest : [2]
Sending beacon message - [2023-07-03 11:57:27] -  seq : [17]
Relaying downlink message [2] to mote #2
```

We can observe that message number 2 should have been sent during the downlink of beacon #16, the same one during which it was received, however it didn't arrive in time.
Needless to say, this delay can happen in the uplink slots too (message arrives after the uplink phase,i.e. during the downlink or inactive part). In order to avoid unnecessary complexity the message arriving out-of-order **won't be discarded** but kept and transmitted during the next beacon.
The impact of such issue can be attenuated by increasing even further the size of a slot, however these random delays might exceed the order of seconds and having slots this big would cause the beacon interval to take too much time.
Some "defensive" approaches have been employed to reduce the occurrence of such phenomenon such as allocating the sensors' uplink slots (the ones actually using the uplink phase) first in the GTS ordering to guarantee some additional "time cushion".

2 Structure of the network

The network follows a star-shaped topology whose shared medium is the mqtt topic over which messages are exchanged. Every node can, of course, overhear all the other transmission on the channel. Given that no direct transmission between the motes are allowed, messages coming from other motes or the coordinator which aren't meant for mote X will be ignored by said mote.

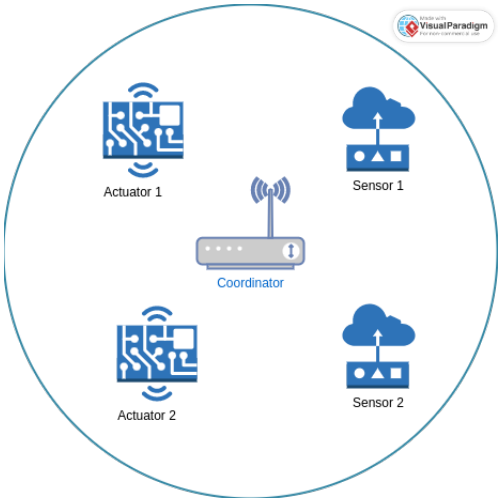

Figure 1: Shape of the network

[2]The simulation with 4 motes has been tested on a 11th gen i9, keeping all of wokwi's tabs open while running node-red on a different PC, and on average the motes had a 3-10 seconds difference in simulation time.

Field Name	Allowed Values	Purpose
msg_type	[0] Association req [1,2] RTT - deprecated [0x1e] Beacon [0x101,0x102] Data up/down [0x200] Ack	Identifies the type of message
CAP_START	Numeric [Ts]	Start of the CAP in number of slots, relative to the reception time of the beacon
CAP_DURATION	Numeric [Ts]	Duration of the Collision Access Part
SLEEP_DURATION	Numeric [Ts]	Duration of the inactive period, in number of slots
Timestamp	**Numeric [UNIX ms]**	**Time at which the beacon was transmitted**
SLOT_DURATION	Numeric [ms]	Duration of a single slot in milliseconds
GTS_UPLINK	Array of integers [AID]	Defines the order of the CFP uplink, during which motes can transmit to the PAN. Each number consists in the association id of the mote. If mote i's AID is in the j-th position it will transmit during the j-th slot.
GTS_DOWNLINK	Array of integers [AID]	Same as GTS_UPLINK but for the downlink phase. The order is exactly the same, it is a different field just to make things clearer.
AIDs	JSON array	For each associated mote in the PAN, this object contains a key-value pair in the form of MOTE_ID : AID It is used by motes to check their AID as well as detecting when actuators are online by sensors.
BEACON_SEQ_NUMBER	Integer	Sequence number of the beacon message
PAN_ID	Random Integer	Identifier of the PAN, selected randomly when the coordinator boots as explained in detail in the following paragraphs.

3 Structure of the beacon

The beacon's structure is similar but not identical to one of those found in real PAN networks. It is composed of a Collision Access part, used to register in the network and of a Collision Free Part, split in uplink and downlink slots to/from the PAN coordinator, in this order. It consists of **one uplink and downlink** slot for each of the motes in the network, in order to be able to send and receive messages from the coordinator.

3.1 Fields of the Beacon Frame

The following fields listed in subsection 3.1 are contained in the JSON representation of the beacon frames.
It is worth noting that the duration of the CFP is defined implicitly as the the sum of the slots in the uplink and downlink phase. The CFP is composed of the uplink phase first followed by the downlink.

3.2 ACK policy

To mimic a real world PAN, messages from exchanged from the PAN to the motes are acknowledged. This is needed as the **chosen QoS on the channel is 0**. Due to various reasons (wokwi dropping packets or pausing the simulation, delays, random disconnection from wokwi's gateway) the motes might become **out-of-sync with the beacon messages**. If Qos 1 or 2 (not supported by PubSubClient anyways) were to be used the broker would send first the messages that weren't acknowledged, i.e. **old and stale beacon messages** which would cause the motes to be essentially "living in the past", a condition that is unacceptable and cannot be allowed to happen.
The slot has been sized in order to allow to receive a DATA message and send back the ACK **as soon as the data has been received**. As such the ACKs aren't allocated to their own uplink/downlink slot but sent back as soon as possible. Since the size of the ACK is very small, the additional overhead of turning back on the

receiver, thus paying wakeup and RX time for less than 10 bytes of data has not been deemed acceptable.
If a mote doesn't receive an ACK it will keep re-sending the data message which hasn't been acknowledged in its uplink slot. The same holds for the coordinator.

3.3 PAN ID

This field represents the ID of the network, and it is selected randomly when node-red boots up as follows:

```
PAN_ID = Math.ceil(Math.random()*Math.pow(2,31)) ^ (Date.now() & 0x0FFFFFFF)
```

When motes first connect to the network they store the pan_id and match it against the one contained in the beacon at each interval. If it does not match the motes will reboot and reconnect to the network as it means that the coordinator has had an hard disconnect and has been rebooted, and as such it has no knowledge of the motes yet. [3]
The opposite might happen, i.e. a mote has been restarted from wokwi and attempts to associate again while the coordinator has already processed its request in the past. It will send an association request that will be ignored by the coordinator as it's already been associated before. In the next beacon it will find its own AID in the list of associated nodes and it will start working as normal. **This is allowed to happen** as the mote has virtually no status to keep coherent (it's just sensors producing data and actuators acting on it) while the coordinator has to manage the message queues and data/ack flows.

4 ESP-32 Motes

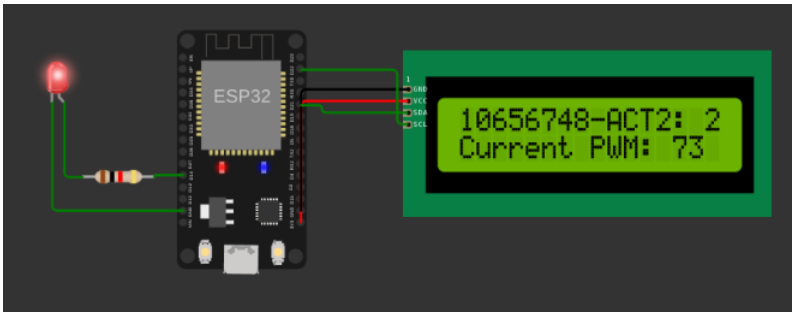

Figure 2: Actuator Mote

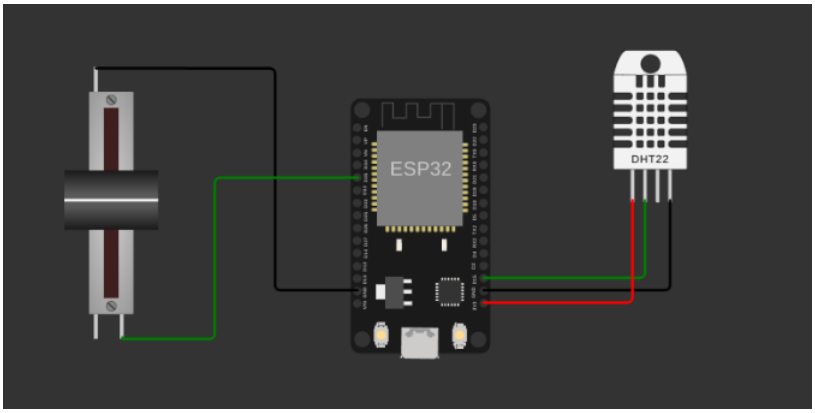

Figure 3: Sensor Mote

4.1 Description of the Motes

As per the project specification, there are 2 kinds of motes, sensors and actuators. Sensors are fitted with a DHT22 sensor along with a slide potentiometer, while actuators have a single LED and a LCD display addition

[3] The restart might seem a bit overkill but starting from a clean slate in wokwi is the best as reconnecting the wifi and the broker will cause wokwi's own gateway to lose the connection most of the time for unknown reasons. Unfortunately, even the restart doesn't work 100% of the times.

to the board itself. The display will report the current association id of the mote in the first row and the PWM value that is being modulated (if any) in the second one. It's been added to make it easier to follow what's going on in the sensors without having to keep looking at node red's terminal output. In addition to that it gives a quick reference on the mote's AID and makes it easier to send custom data messages to the mote from node red's UI as will be explained later.
The potentiometer in the sensor is instead used to modify on the fly the probability of capturing a measurement for each beacon interval. The sensor motes will draw a random number between 0 and 100 each time and check whether it's contained in the interval defined by the value read by the potentiometer pin as such:

```
((esp_random() & 0x0FF) % 100) in [0,map(analogRead(POT_PIN),0,4095,0,100)]
```

If said condition holds, the sensor will buffer a measurement with the current values returned by the DHT sensor. The potentiometer is placed so that the value sampled is higher when the slider is as the top of the range.

The LED is the one that will modulated according to the values sampled by the sensors. Both boards have common modules which regulate their operations (i.e. the way they handle messages or the mqtt connection to the broker) and differ only in specific parts as is explained next.

4.2 Common operations of the motes

4.3 Duty Cycling

As specified in the previous paragraphs, sleep is only "emulated" in the nodes by the delay function which is also (rightly so) used to idle during the active part of the beacon. As such, duty cycling is implemented not in a single function but distributed between the mqtt callback and the main loop function of the motes.

4.4 Message Processing Callback

MQTT messages are handled in the function:

```
void message_received(char* topic, byte * message, unsigned int length)
```

which is set as the default mqtt callback during the inizialization of **PubSubClient**. The versions in sensors and actuators are slightly different only in the handling of data messages and ACKs. Their common behavior can be summarized as follows in image 4.
Callbacks usually aim to be as short as possibile in order to be able to process multiple inbound messages and thus split the logic of parsing the message from the one of receiving it. This is clearly not the case for two reasons. First of all, to keep code size to a minimum and make it easier to read, second due to the structure of the message/response interaction between the coordinator and the motes. There aren't supposed to be multiple (meaningful) messages to be exchanged on the channel at the same time, and even if heard, the motes should just ignore those and idle during the other mote's active intervals. Moreover, all the code paths of the function are all relatively short and thus won't take much time to execute.

4.5 Sensor-specific Modules

4.5.1 Data Sampling

At every iteration of the **loop()** function, the mote will draw a random number and buffer a measurement if said number is within a specific interval as explained in 4.1. The values will be collected and stored until the sensors detect (by looking at the AID vector of the beacon) at least one of the actuators online. Once that happens they will begin emptying the buffer by sending one data message to one of the online actuators selected randomly.
The buffer can hold up to 200 measurements and is managed as a FIFO queue by the functions:

```
add_reading(chosen_actuator,sampled_value);
get_first(* to,* value, *seq_number);
```

which respectively add a reading to the queue and get the first available sample. The queue is controlled by two variables, **head** which points to the first available measurement and **tails** which points to the first free slot. The **head** is moved forward **only when a message has been acknowledged**.

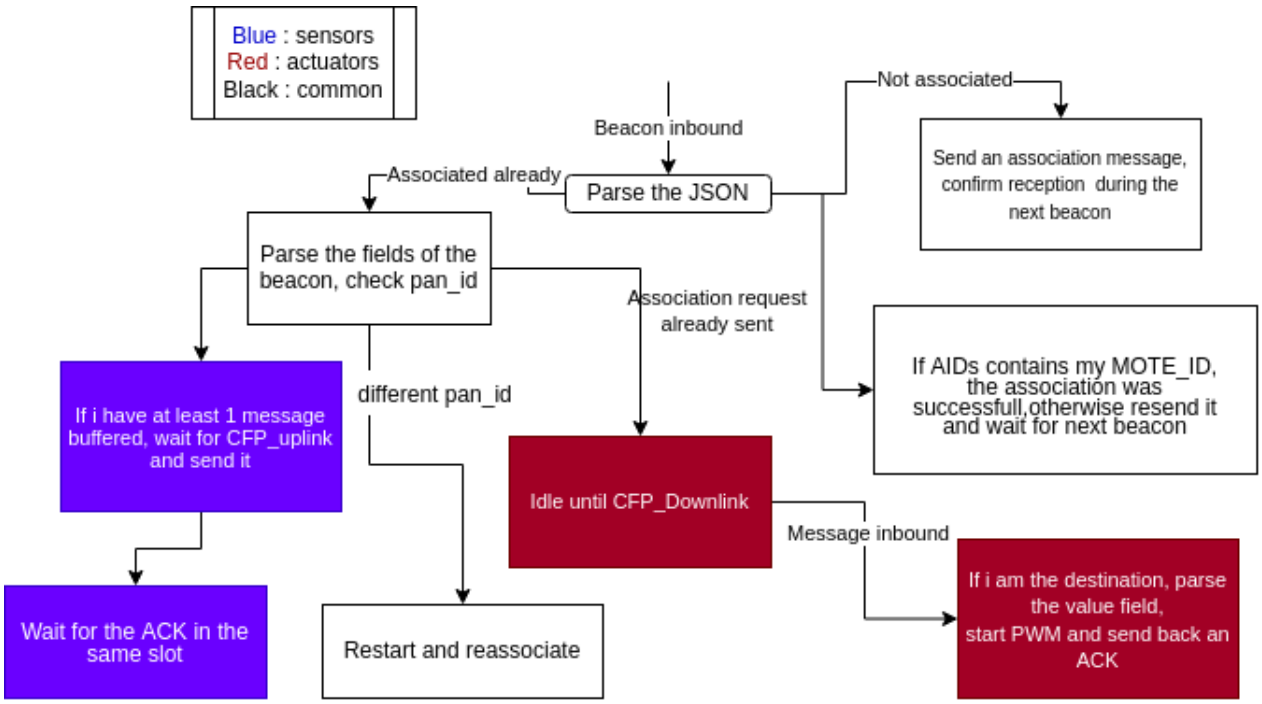

Figure 4: Flow of the MQTT callback

4.6 Actuator-specific Modules

4.6.1 PWM task

The pulse-width modulation of the actuators is managed on a dedicated core (number 0) due to the issues presented in 1.1.1. The task in charge of this action is created in the setup and assigned to core #0 as follows:

```
semaphore = xSemaphoreCreateBinary();
xTaskCreatePinnedToCore(
        do_pwm_stuff,          // Task function
        "pwm_handler",         // Task name
        10000,                 // Stack size (bytes)
        NULL,                  // Task parameter
        1,                     // Task priority
        NULL,                  // Task handle
        0                      // Core number (0 or 1)
  );
```

It is held by a semaphore until the first sample is delivered by one of the sensors. The actual code of the task can be briefly summarized as such:

```
#DEFINE INCLUDE_vTaskSuspend 1 //block indefinitely until data arrives
if (xSemaphoreTake(semaphore, (TickType_t)portMAX_DELAY) == pdTRUE)
   Serial.println("Semaphore obtained, starting PWM of the LED");

 while (true) {
       if(new_value_inbound){
         flash_led(4);
         new_value_inbound = false;
         //LCD stuff
       }
       digitalWrite(PWM_LED_PIN, HIGH);
       delay(pwm_value);
       digitalWrite(PWM_LED_PIN, LOW);
       delay(PWM_PERIOD_TIME-pwm_value);
  }
```

When a message is received the new_value_inbound variable is set to true and the PWM led is flashed 4 times to signal that it's about to change value. After that's been done (only once per new value) the PWM cycle begins. The PWM period has been set to 100 millisecond (there is really no alternative as the minimum granularity needs to be 1 ms) and as such there is no need to use the **map()** function as the **humidity value is rounded** and already belongs to the [0-100] interval.

5 MQTT channel

5.1 Channel name

The chosen channel for the project is named

/iot/polimi/c22b520197ff50c6ae38d0536a41a53d17e7e7ed9ece4ed815b74b9459c20238

which is the sha256 hash of my person code **10656748**, chosen this way in order to avoid random people publishing on the channel.

5.2 Types of Messages

Each of the messages, given that MQTT is a text-based protocol and node-red implemented in JavaScript, is formatted in JSON and manipulated with the aid of the **ArduinoJson** library on the ESP-32 motes. Needless to say, in a real PAN using JSON or any text-based message protocol for the beacon would be an incredible waste of resources, which is however deemed acceptable in this instance given the didactic purpose of the project. The following types of messages are thus defined to enable motes to interact with the PAN coordinator. Being a star topology, each of the messages is first relayed to the coordinator in the uplink slot of the CFP, that will then forward it to the destination in the downlink slot of the appropriate mote.

1. Association Message [Value = 0]

```
{
    "msg_type" : 0,
    "mote_id" : "MOTE_ID"
}
```

2. DATA uplink message [Value = 0x100]

```
{
    "msg_type" : 256,
    "from" : "SOURCE_MOTE_ID",
    "to" : "DEST_MOTE_ID",
    "value" : "humidity_value",
    "sequence_number" : "SEQ_N"
}
```

3. DATA downlink message [Value = 0x101]

```
{
    "msg_type" : 257,
    "from" : "SOURCE_MOTE_ID",
    "to" : "DEST_MOTE_ID",
    "value" : "humidity_value",
    "sequence_number" : "SEQ_N"
}
```

4. BEACON message [Value = 0x1E]

```
{
    "msg_type":30,
    "slot_duration":500,
    "cap_start":1,
    "GTS_uplink":[1,2],
    "GTS_downlink":[1,2],
    "cap_duration":4,
```

```
    "sleep_duration":7,
    "AIDs":{
        "10656748-SENS1" : 1,
        "10656748-ACT1" : 2
    },
    "beacon_interval":14,
    "pan_id":947582242,
    "beacon_seq_number":70,
    "timestamp": 1687698575913
}
```


5. ACK message [Value = 0x200]


```
{
    "msg_type" : 512,
    "from" : "SOURCE_MOTE_ID",
    "sequence_number" : "SEQ_N"
}
```


6 Node-Red's PAN coordinator

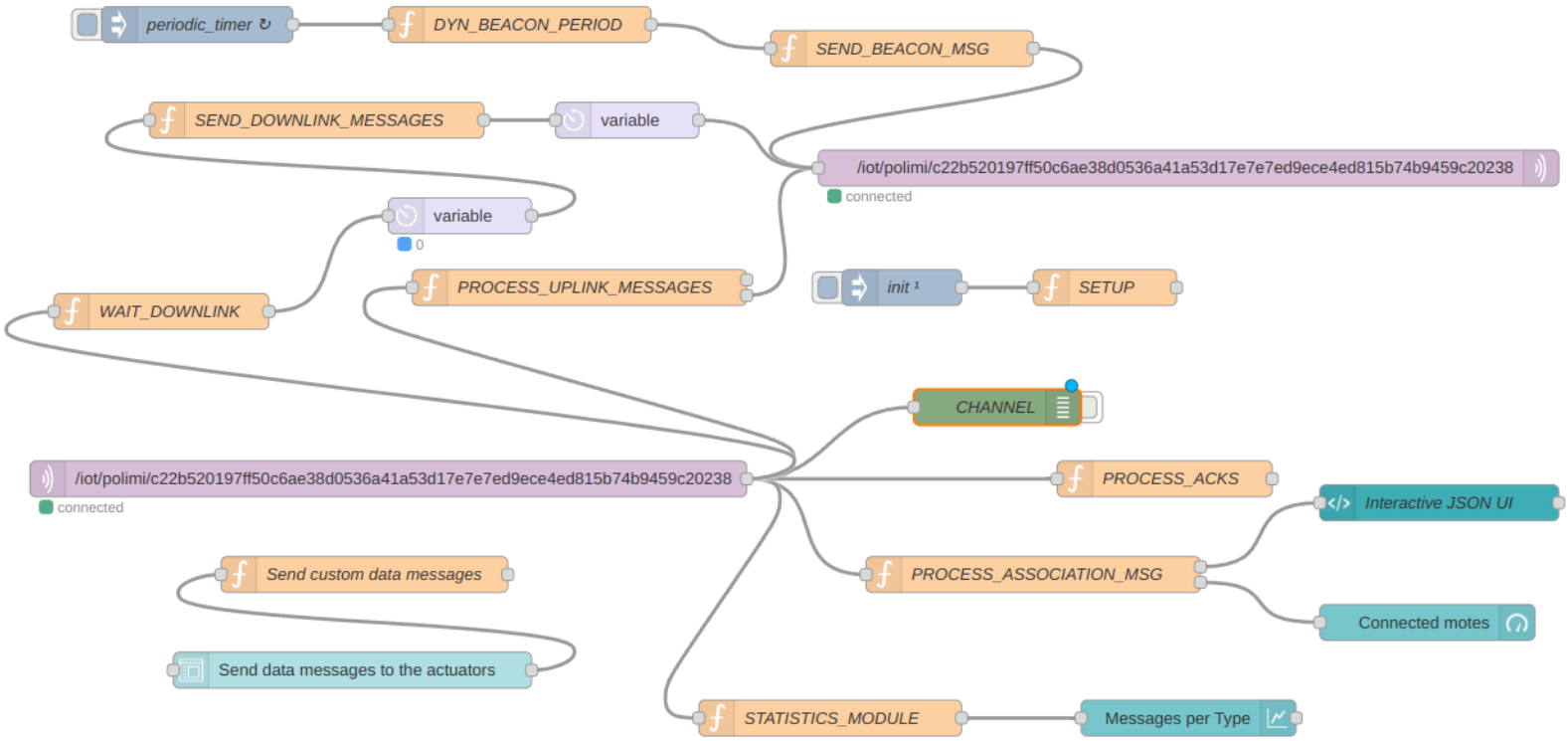

Figure 5: Structure of the PAN coordinator

6.1 Node red Variables

The following are the main (global) variables that encapsulate the status of the network.

1. message_buffer
   It's the array that holds all the messages to be sent during the downlink phase of the beacon. It is managed as a FIFO queue, sending the messages at the top during each beacon. Said message will be deleted only when acked by the destination mote.

2. beacon_message
   A JSON object which encapsulates the current state of the beacon, modified only when new nodes connect to the network. For example:

3. associated_nodes
   A JSON object that keeps track of the nodes associated to the network, it has a key-value pair field for each mote in the form of

   ```
   { "MOTE_ID" : "AID"}
   ```

   and an additional fields "number_of_nodes" which counts how many nodes are connected to the network.

6.2 Node-red Nodes

6.2.1 SETUP

This node is triggered once at startup and initializes all the variables needed to keep track of the network to their default values.

6.2.2 SEND_BEACON_MESSAGE

It's the heart of the coordinator, it sends the beacon message on the channel after each beacon period.

6.2.3 PROCESS_ASSOCIATION_MESSAGE

This node, as the name implies, is in charge of processing all the association messages sent by the motes during the CAP. Once a new one arrives, the mote updates the list of connected nodes and updates the value of the **CFP** of the beacon, adding a slot in uplink and downlink for the new mote.
The output of this mote is connected to two UI nodes, the first allows to browse the connected nodes (it's a JSON array) while the second is a gauge which shows in real time how many of the 4 motes are currently connected to the network.

6.2.4 PROCESS_UPLINK_MESSAGES

It handles the DATA messages sent by the motes during their uplink phase and does two things:

1. Sends back the ACK to confirm that the message has been received.
2. Puts the message in the downlink queue, a buffer that holds the messages that will be sent during the downlink phase.

6.2.5 WAIT_DOWNLINK

Once the beacon arrives over the channel (done this way to minimize coordinator/mote delays) it will send a message that will switch on the downlink handler exactly when the CFP downlink period starts.

6.2.6 SEND_DOWNLINK_MESSAGES

This node is triggered by the arrival of the beacon over the shared channel once per beacon interval (it is delayed by the previous node so that it arrives exactly when the CFP starts). Once that happens, it iterates the message queue and picks the first message (FIFO ordering) for each mote. It calculates the delay (in order to send the message once the destination's CFP downlink period starts) and sets the delay field so that it is sent at the right time. The msg.delay is used by the subsequent delay node that will hold the message until it's time to send it.

6.2.7 PROCESS_ACKS

Connected directly to the MQTT out node, it snoops for ACK messages, once they arrive it matches the source node of the message and the sequence number, if they match it removes the message from the downlink queue.

6.2.8 STATISTICS_MODULE

It provides some basic statistics to be visualized in the UI sections. It mostly aggregates messages for each type and sends them over to the bar chart of the UI sections to have a look on what's going on in the network.

6.3 UI Components

What follows is how the UI section of the coordinator might look like after all nodes are connected and have exchanged messages for a while:

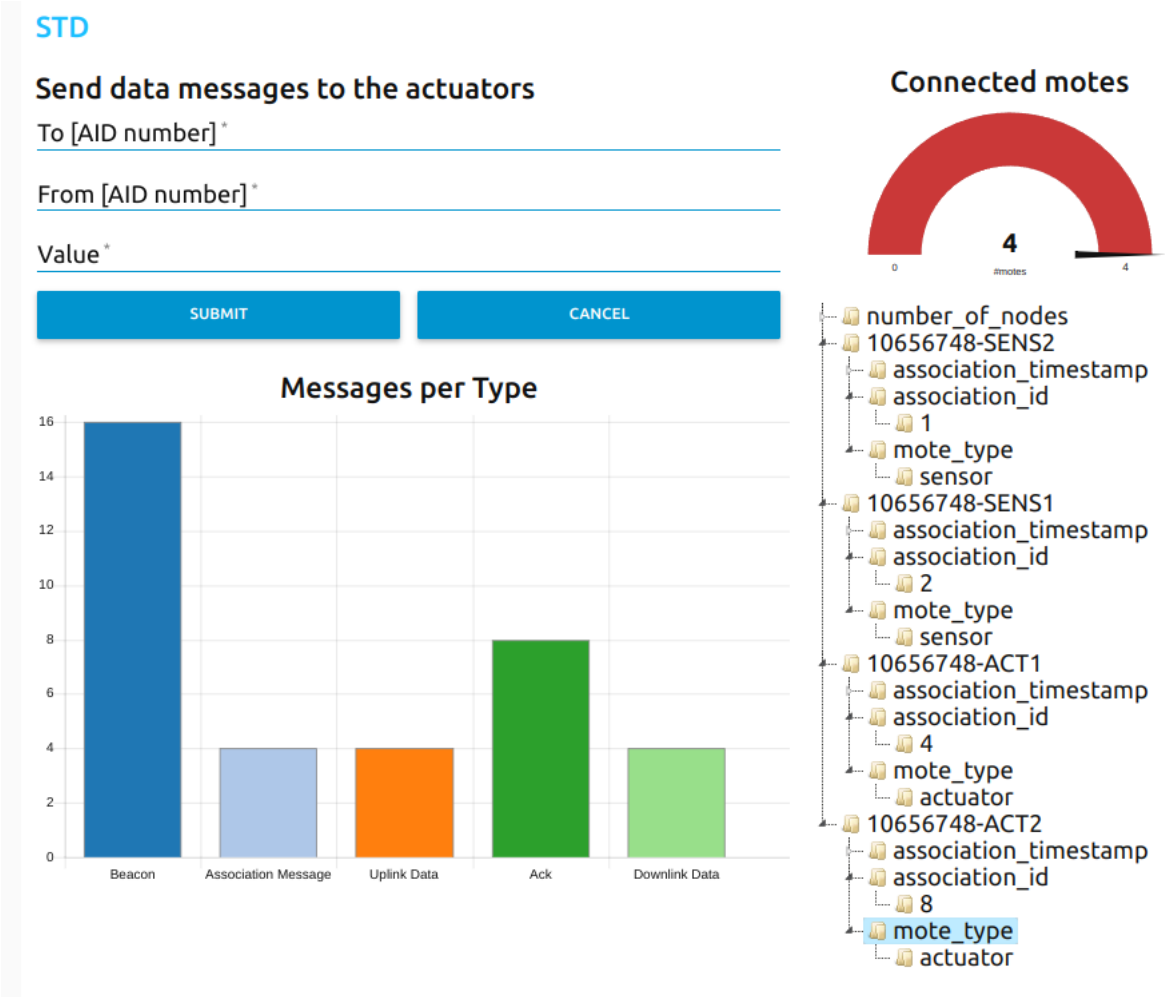

Figure 6: UI section of Node Red

The form can be used to send messages to the actuators to test the PWM without having to wait for the sensors to produce data. It takes an input two AIDs, the source and destination node (the source can be fake, the nodes won't check if it belongs to the network to keep things simple).
The graph instead is a bar chart showcasing the number of messages exchanged over the channel per type.
On the right there is a JSON interactive UI element that shows basic information of the motes currently connected to the network.