IoT FINAL PROJECT a.y. 2022/2023

(1) Name: Capacci Tommaso (Person Code: 10654230, team leader)
(2) Name: Ginestroni Gabriele (Person Code: 10687747)

Project type: 1

ThingSpeak public channel URL: https://thingspeak.com/channels/2233511

In this project we were requested to implement a lightweight publish-subscribe application protocol using TinyOS and to test it over a star topology network in which we had 8 nodes (acting as MQTT motes) all connected to a single and unique PAN coordinator (acting as a MQTT broker).

Report outline:

1. Custom application layer message definition
2. Connections and subscriptions management
3. Retransmissions management
4. Messages queue implementation
5. Additional support code and assumptions
6. Node-RED logic
7. Notes on simulation

1) Custom application layer message definition

We decided to adopt a message structure that would allow us to reuse the same packet for any kind of message. We achieved this task using only 4 fields in our payload:

```
/*********** MESSAGE TYPE **********/

typedef nx_struct radio_route_msg {
    nx_uint16_t message_type;
    nx_uint16_t id;
    nx_uint16_t topic;
    nx_uint16_t payload;
} radio_route_msg_t;
```

We can now quickly present the meaning of all of these:

- *message_type*: represents the kind of application layer message the packet is carrying. This has to assume one of the following values, defined as constants:

```
CONNECT = 0,
CONNACK = 1,
SUB = 2,
SUBACK = 3,
PUBLISH = 4
```

- *id*: this field is used in CONNECT, SUB and PUBLISH messages to specify from which node the message is coming from
- *topic*: used in SUB messages to specifiy to which topic a node is requesting to be subscribed to, but also in PUBLISH messages to specify over which topic the message is being published
- *payload*: used in PUBLISH messages to represent the content of the message

2) Connections and subscriptions management

We used singly linked lists to keep track of all the connections and subscriptions the PAN coordinator must handle. To achieve this task, we used a very general node structure in such a way we could use it for both connections and subscriptions lists. The basic node structure has been defined in the "FinalProject.h" file in the following way:

```
/****** ADDITIONAL DEFINITIONS *****/

typedef struct Node{
    uint16_t id;
    struct Node* next;
} Node;
```

In total we used 4 linked lists, 1 to collect all the IDs of the nodes that have correctly connected to the coordinator by sending a CONNECT message and other 3 lists (1 for each topic) to keep track of the IDs of the nodes that have subscribed with a SUB message to that specific topic:

```
// Data structures for connections and subscriptions
Node* connections = NULL;
Node* subscriptions[3] = {NULL, NULL, NULL};
```

These lists have been managed through three simple methods:

```
bool searchID(Node* list, uint8_t node_id);
Node* addNode(Node* list, uint8_t node_id);
void printList(Node* list);
```

- *searchID*: a function that traverses the specified list to search for the requested id
- *addNode*: a function that first checks that the specified ID is not already present in the list exploiting the previous function and then, in case it is not already present, adds a new node with the correct id at the head of it
- *printList*: a function used for debugging purposes to print all the nodes inside a list, starting from the one placed at the head (so, the last added).

3) Retransmissions management

To handle retransmissions, we decided to use a timer (*Timer0*) and 2 additional variables to save the pointer to a copy of the message that needs to be acknowledged and the relative address to be used in case of retransmisssion:

```
// Variables for retransmission
message_t* request;
uint16_t requestAddress = 0;
```

The timer would be set (in one-shot mode) through a specific function called *"handleRetransmission"*:

```
void handleRetransmission(uint16_t address, message_t* message){
/*
* Handle retransmission of the specified packet
*/
  request = (message_t*) malloc(sizeof(message_t));
  memcpy(request, message, sizeof(message_t));
  requestAddress = address;
  call Timer0.startOneShot(ACK_TIMEOUT);
}
```

This should be called whenever needed, which means only in case of sending specific types of messages (CONNECT and SUB): this is performed directly by the "*sendDone*" event handler itself, in such a way we can be sure to start waiting exactly from when the message has been sent.

If the right ACK arrives to the node before the timer is fired, it is simply turned off and the involved variables are nullified. Otherwise, after a fixed number of milliseconds (defined by the constant named "*ACK_TIMEOUT*"), the timer is fired, triggering the resend of the saved message:

```
event void Timer0.fired() {
/*
* Timer used to trigger retransmissions
*/
  radio_route_msg_t* packet = (radio_route_msg_t*)call Packet.getPayload(request, sizeof(radio_route_msg_t));

  dbgerror("Timer", "Request was not acknowledged in time. Resending.\n");
  generate_send(requestAddress, packet->message_type, packet->id, packet->topic, packet->payload);
}
```

4) Messages queue implementation

While the TinyOS nodes can receive multiple messages by default, we needed to come up with some way to allow them to correctly handle the send of multiple messages. The strategy that worked best for us included using a queue to collect all the outgoing messages and then exploit a timer (*Timer2*) to periodically send the oldest message present inside the queue. In this way the queue is managed with a First In First Out (FIFO) strategy.

We decided to implement the queue with 2 circular buffers (in this case implemented by means of arrays), one for the messages and the other to hold the address that the message in the same position of the first array must be sent to. To correctly execute the FIFO strategy, we needed 2 circular indexes:

```
// Packets queue
message_t* packetsPool[PACKET_POOL_SIZE];
uint16_t addressesPool[PACKET_POOL_SIZE];
uint8_t head = 0;
uint8_t tail = 0;
```

- *head*: points at the first empty slot of the queue
- *tail*: points at the oldest message inside the queue (so, the next to be sent)

The queue is managed through 3 different portions of code:

- *generate_send*: this function is used to allocate (in heap) the space required for a new message, set its payload with all the fields specified through the function's arguments and add pointer/address to the queue before incrementing the "head" index in a circular way

```
void generate_send (uint16_t address, uint16_t message_type, uint16_t id, uint16_t topic, uint16_t payload){
/*
* Allocate a new message, populate it with the specified parameters and add it to the packets queue's head
*/
  message_t* message = (message_t*) malloc(sizeof(message_t));
  radio_route_msg_t* packet = (radio_route_msg_t*)call Packet.getPayload(message, sizeof(radio_route_msg_t));

  packet->message_type = message_type;
  packet->id = id;
  packet->topic = topic;
  packet->payload = payload;

  atomic{
    packetsPool[head] = message;
    addressesPool[head] = address;
  head = (head + 1) % PACKET_POOL_SIZE;
  }
}
```

- *"Timer2.fired" event handler*: when the timer fires, it checks that the queue is not empty and, only in case it is not, performs the actual send of the message pointed by the "tail" index

```
event void Timer2.fired() {
/*
* Timer used to send queued messages
*/
  atomic{
    if(head != tail){
      if (call AMSend.send(addressesPool[tail], packetsPool[tail], sizeof(radio_route_msg_t)) == SUCCESS)
        dbg("Radio_send", "Sending packet to %d at time %s:\n", addressesPool[tail], sim_time_string());
      else
        dbgerror("Radio_send", "@@@FAILED@@@ Sending packet to %d at time %s:\n", addressesPool[tail], sim_time_string());
  }
  }
}
```

- *"sendDone" event handler*: this handler is also used to check that the the pointer to the message that has been sent corresponds to the one at the tail of our queue so that, in case it is, we are sure the right message has been sent and we can safely increment the "tail" index

```
event void AMSend.sendDone(message_t* bufPtr, error_t error) {
/*
*  Check if the RIGHT packet has been sent
*/
radio_route_msg_t* packet;

  atomic{
    if (bufPtr == packetsPool[tail]){
      packet = (radio_route_msg_t*)call Packet.getPayload(packetsPool[tail], sizeof(radio_route_msg_t));
      if (packet->message_type == CONNECT || packet->message_type == SUB)
        handleRetransmission(addressesPool[tail], packetsPool[tail]);

      free(packetsPool[tail]);
      tail = (tail + 1) % PACKET_POOL_SIZE;
    }
  }
}
```

5) Additional support code and assumptions

Inside the code we can find some additional code used for support purposes:

- *printPacketDebug*: function used to print on a specific TOSSIM debug channel ("*Data*") the content of a packet in a structured way
- *handlers*: for readability and atomicity we decided to decouple the protocol logic from the message parsing one. To implement this we left the parsing of the message type to the "*Receive*" event handler and then we used a separate function (one for each message type) for the remaining handling procedures

```
void handleCONNECT(radio_route_msg_t* payload);
void handleCONNACK(radio_route_msg_t* payload);
void handleSUB(radio_route_msg_t* payload);
void handleSUBACK(radio_route_msg_t* payload);
void handlePUBLISH(radio_route_msg_t* payload);
```

- *Timer1*: it is used to trigger periodic publish transmissions

```
event void Timer1.fired() {
/*
* Timer used to trigger pubblications
*/
  uint16_t topic;

  counter++;
  if(TOS_NODE_ID >= 2 && TOS_NODE_ID <= 4)
      topic = 1;
    else if(TOS_NODE_ID >= 5 && TOS_NODE_ID <= 7)
      topic = 2;
    else
      topic = 0;

  dbg("Timer", "Publishing a message on topic %d. \n", topic);
  generate_send(1, PUBLISH, TOS_NODE_ID, topic, counter);
}
```

Some assumptions we used for our implementation:

- For debugging purposes we preferred to use deterministic subscriptions rules: nodes from 2 to 4 subscribed to topic 0 (Temperature), from 5 to 7 to topic 1 (Humidity) and nodes 8 and 9 to topic 2 (Luminosity). We did the same for pubblications: each node should publish on the "next" topic with respect to the one it is subscribed to (so 0 -> 1, 1 -> 2, 2 -> 0)
- Although we know that, in general, it is not necessary to be subscribed to any topic to publish over the network, we preferred to delay the start of publications for each node to after the moment it has subscribed to its topic so that we could easily see form the log of the simulation how much time does it take for each node to complete the connection/subscription cycle
- Instead of printing random payloads we preferred to keep a counter for each node (initialized to 0) to then be included as payload in PUB messages and incremented after each send: in this way, in the last visualization, we were able to graphically check if the log of the simulation was coherent with what we were seeing

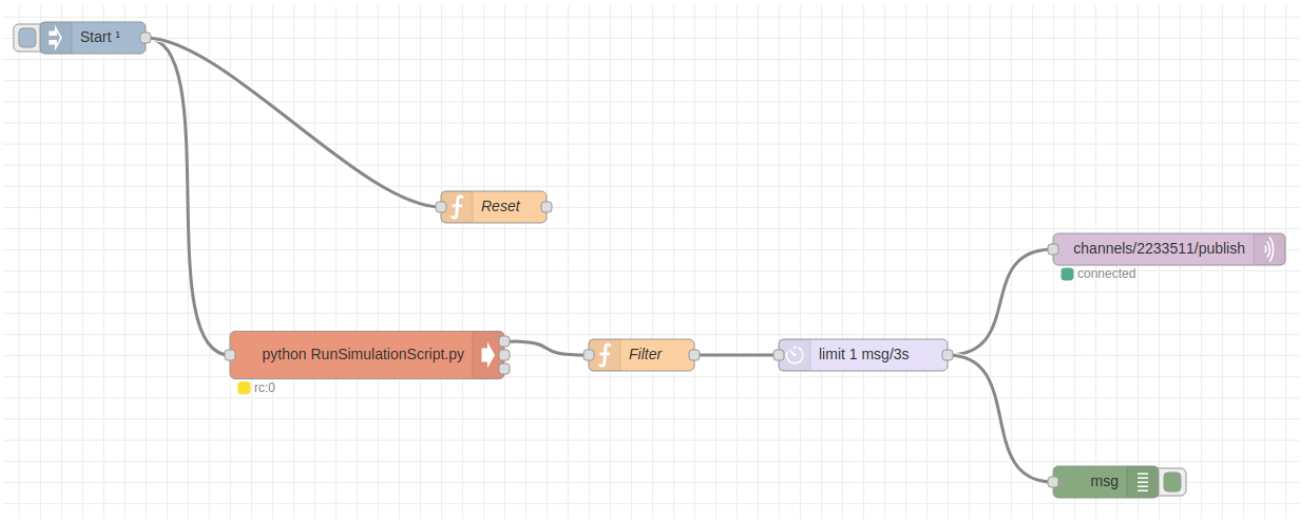
Constants used:

```
/********* CONSTANTS ********/

enum{
    AM_RADIO_COUNT_MSG = 10,
    ACK_TIMEOUT = 500,
    PUB_INTERVAL = 200,
    SEND_INTERVAL = 100,
    PACKET_POOL_SIZE = 20,
    CONNECT = 0,
    CONNACK = 1,
    SUB = 2,
    SUBACK = 3,
    PUBLISH = 4
};
```

6) Node-RED logic

Inside Node-RED we implemented the following logic:



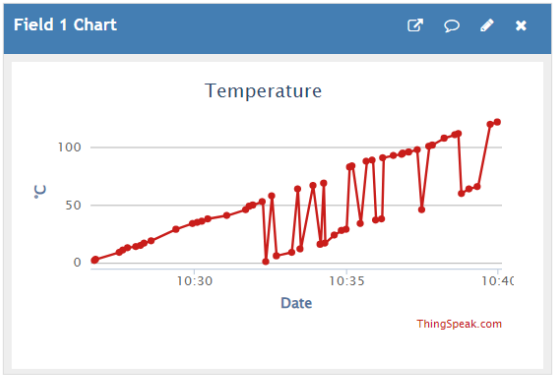
The main nodes are:

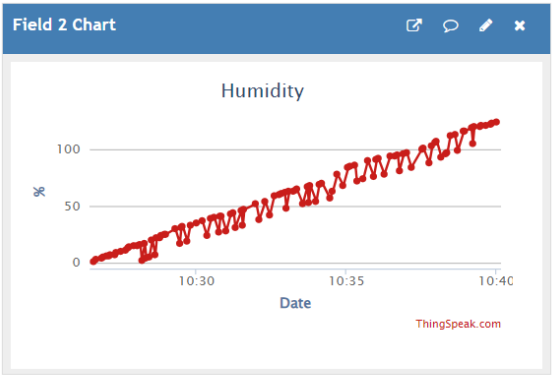
- *python RunSimulationScript.py*: this node allowed us to execute the TOSSIM simulation directly inside the Node-RED environment and redirect the results published from the executable on *stdout* to the following node. We used relative paths, so it is necessary to start Node-RED from inside the same folder that contains the python script for this to work
- *Filter*: this node contains a Javascript function that is used to filter out those printed lines that are not interesting to our means and convert the remaining ones to JSON objects

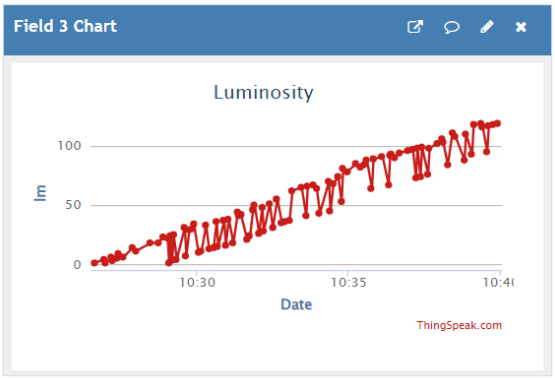
- *limit 1 msg/3s*: this delay node is used to limit the message rate to one message every 3 seconds in a way to not overload the ThingSpeak receiving MQTT broker
- *Reset*: node that contains quite simple code used to reset the status of the previous node to “empty”. It appears like it is not linked to any other node because it has been used only when needed by manually wiring it to the “limit 1 msg/3s” node and restarting the execution.

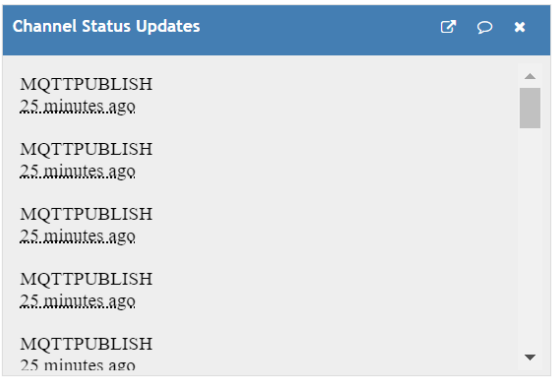
7) Notes on simulation

As we can see from the ThingSpeak charts, thanks to our choice for the publications timing and payload, we can easily recognize the moment in which different devices started publishing on our topics (in the Temperature chart this is particularly notable):









Moreover, we can observe that, in every execution, not all the devices are able to connect on the first attempt: usually 2-3 rounds are needed to complete connection and subscription for all groups of nodes but this can be due to the effect of noise over the communication channels.