**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

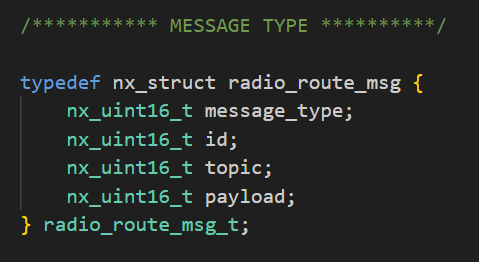
ThingSpeak public channel URL: <https://thingspeak.com/channels/2233511>

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
8. **Custom application layer message definition**

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



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 be one of the following values defined as constants:

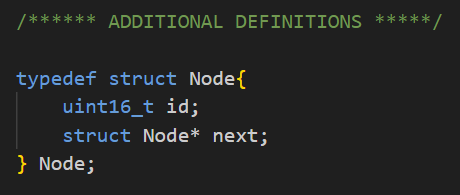
A black background with white text

Description automatically generated

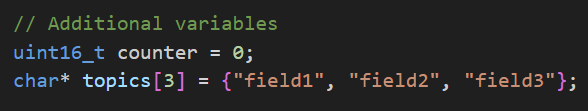
* *id*: this field is used in CONNECT and SUB messages to specify from which node the request is coming from
* *topic*: used in SUB messages to specifiy to which topic a node is requesting to be subscribed to, but also in PUBLISH messages to specify over which topic the message is being published
* *payload*: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

1. **Connections and subscriptions management**

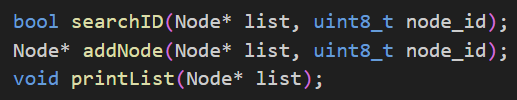
We used singly linked lists to keep track of all the connections and subscriptions the PAN coordinator has to handle. In order to achieve this task we used a very general node structure in such a way we could use it for both connections and subscriptions. The basic node structure has been defined in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



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:



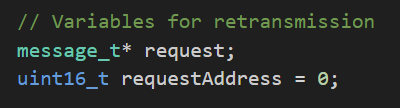
These lists have been managed through 3 simple methods:



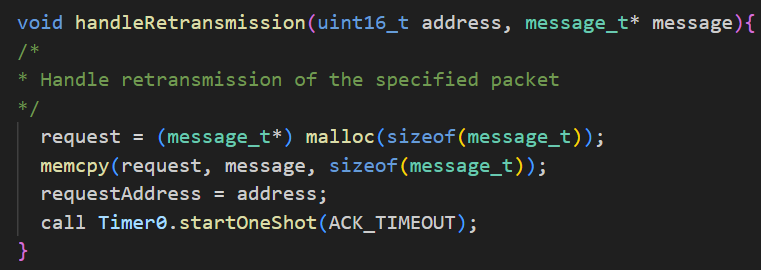
* *searchID*: a function that traverses the specified list in order to search for the requested id
* *addNode*: a function that first checks that the specified ID is not already present in the list then, in case, adds a new node with the correct id in the head of it
* *printList*: a function used for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

1. **Retransmissions management**

In order 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:



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

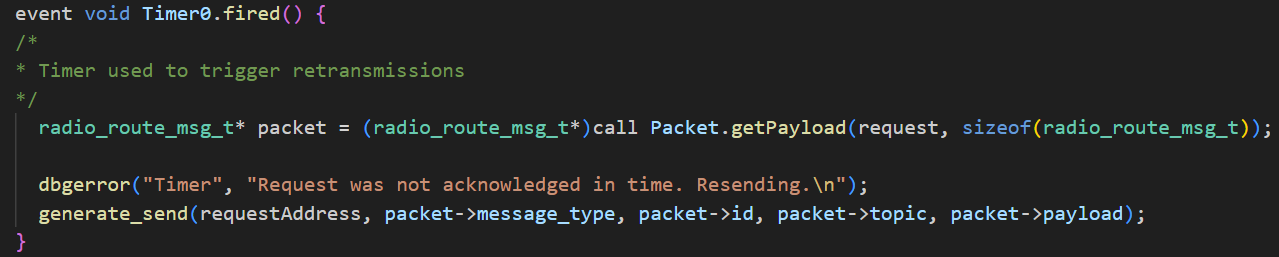


This should be called whenever needed (which means only in case of sending specific type of messages, which are CONNECT and SUB) by the “*sendDone*” event handler itself, in such a way we’ll be sure to start waiting exactly from when the message has been sent:

A computer screen with text on it

Description automatically generated

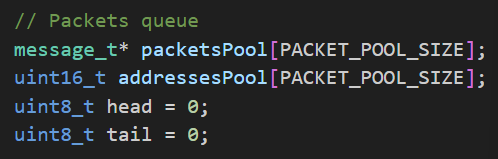
If the right ACK arrives to the node before the timer is fired, it is simply turned off and the involved variables are nullified otherwise, after a fixed amount of milliseconds defined by the constant named “*ACK\_TIMEOUT*”, the timer fires triggering the resend of the saved message:



1. **Messages queue implementation**

While the TinyOS nodes are able to 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 use a timer (*Timer2*) to periodically send the oldest message inside the queue at that moment. In this way the queue is managed with a First In First Out (FIFO) strategy.

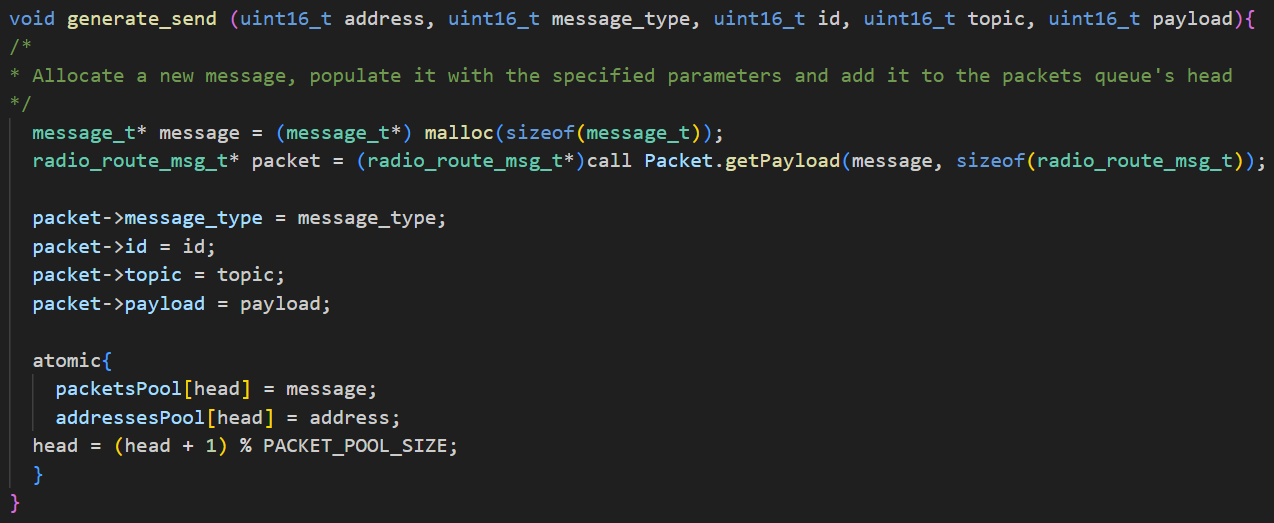
We decided to implement the queue with 2 circular buffers, in this case arrays, one for the messages and the other containing the address that the message in the same position of the first array has to be sent to. In order to correctly execute the FIFO strategy we needed 2 circular indexes:



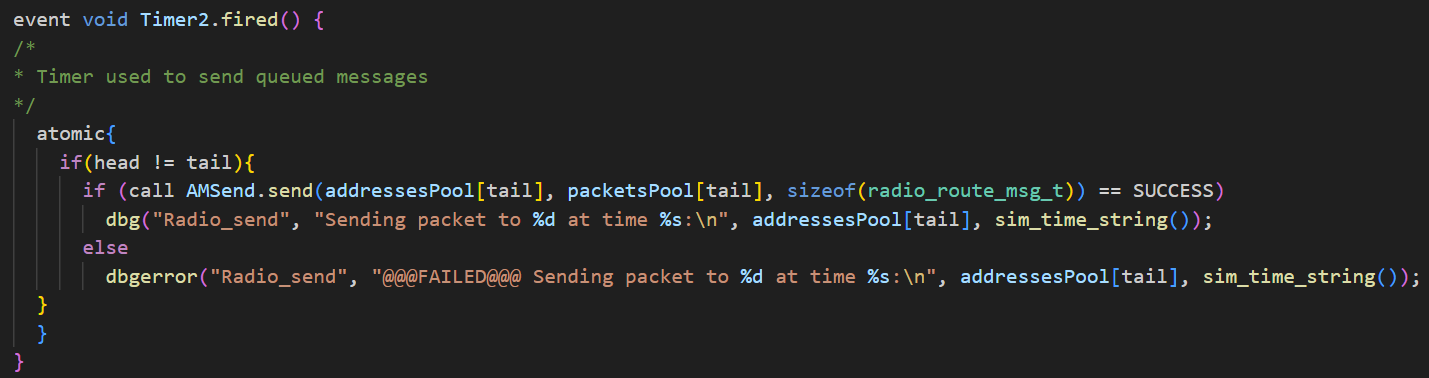
* *head*: points at the first empty slot of the arrays
* *tail*: points at the last message inside the queue, so the next to be sent

The queue is managed through 3 different portions of the code:

* *generate\_send*: this function is used to allocate 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

**

* *“fired” event handler*: when the right 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

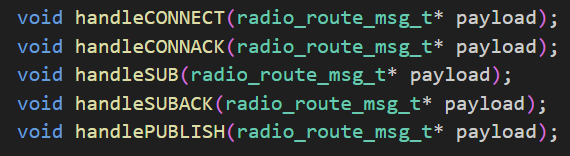


* *“sendDone” event handler*: this handler is also used to check that the the pointer to the message that has been sent corresponds to the one at the tail of our queue so that, in case it is, we can safely increment the “tail” index

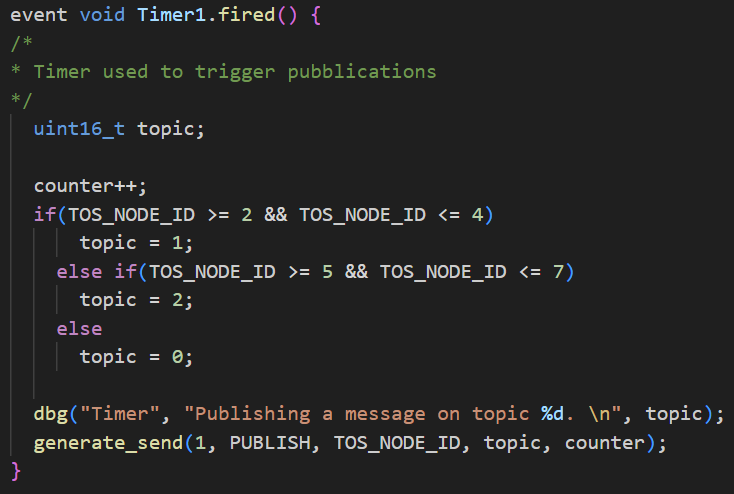
1. **Additional support code and assumptions**

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

* *printPacketDebug*: 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



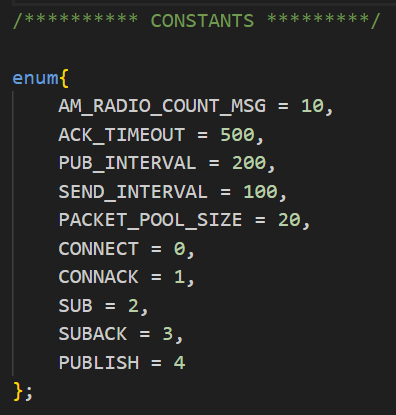
* *Timer1*: it used to trigger periodic transmissions



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).
* Same for pubblications: each node should publish on the “next” topic with respect to the one it is subscribed to (0 -> 1, 1 -> 2, 2 -> 0)
* Although we know that, in general, it is not necessary to be subscribed to any topic to publish, we preferred to delay the start of publications for each node to after the moment it has subscribed to its topic
* Instead of printing random payloads we preferred to keep a counter for each node (initialized to 0) that has been 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:



1. **Node-RED logic**

Inside Node-RED we implemented the following logic:

A group of tags with text

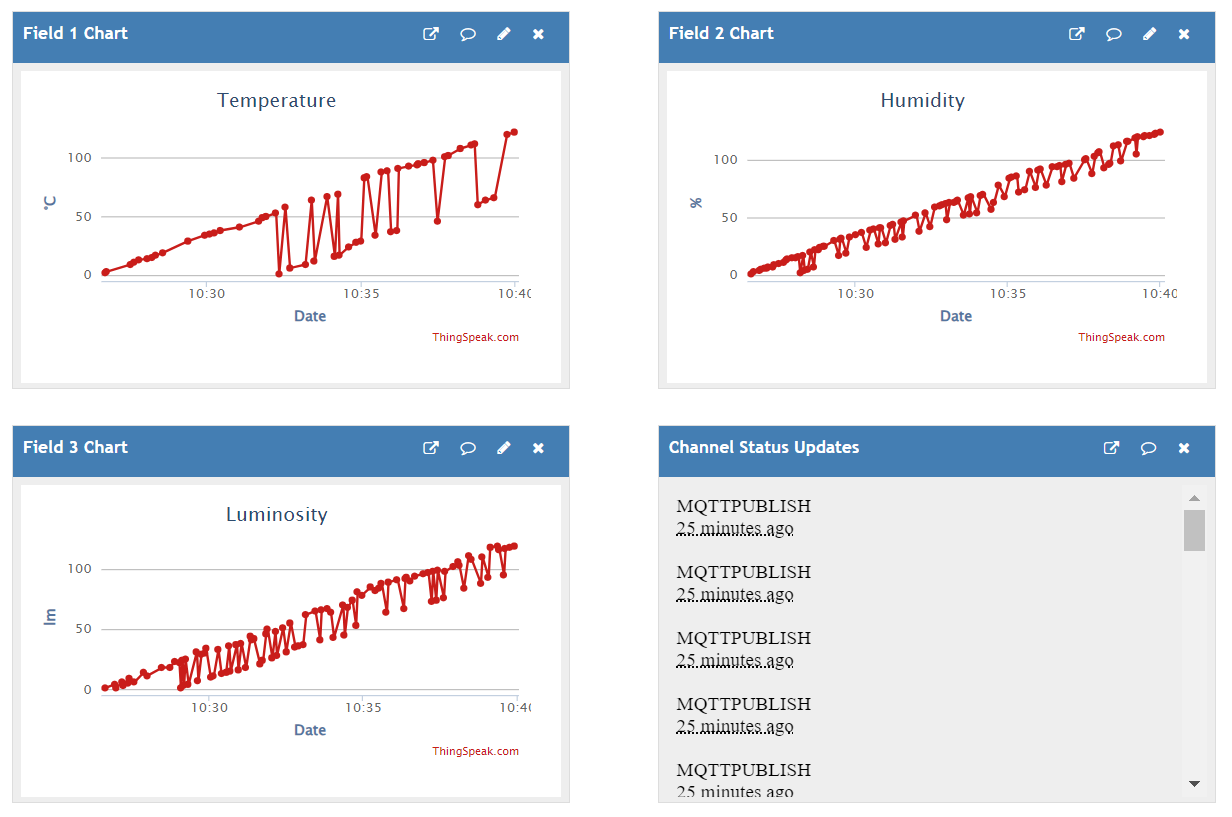
Description automatically generated with medium confidence

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 in order 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 other ones to JSON objects
* *limit 1 msg/3s*: this delay node is used to limit the message rate to 1 message every 3 seconds in a way to not overload the ThingSpeak receiving MQTT broker
* *Reset*: node that contains very simple code used to reset the status of the previous node to “empty”

1. **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 notify the fact that it seems 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 each group of nodes, but this can be due to the effect of noise over the communication channels.

A screenshot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generated