IoT 2023 FINAL PROJECT

(1) Name: Matteo Caccavale Person Code: 10693694

(2) Name: Aleix Stefano Prats Ferret Person Code: 10600613

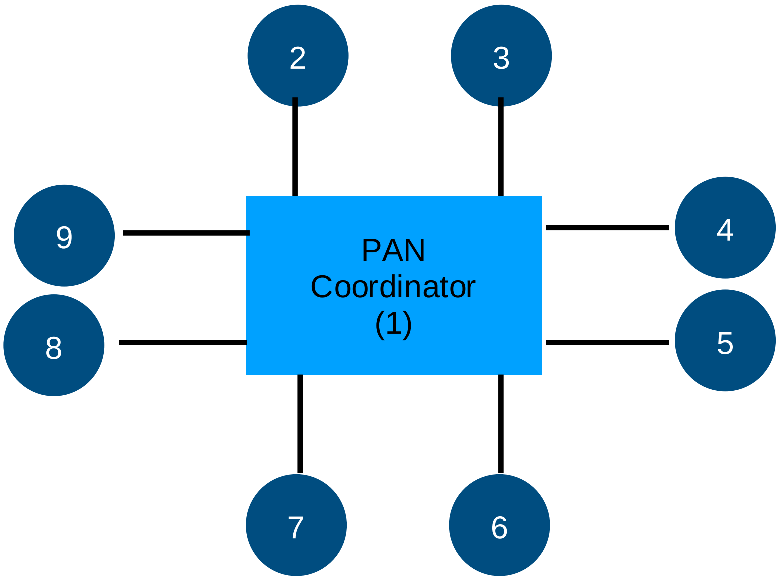
INTRODUCTION:

Among the 3 proposed projects we have chosen the first one: “Lightweight publish-subscribe application protocol”.

We had to create a MQTT-like application in TinyOs constituted by 9 nodes: a PAN Coordinator (PANC) and 8 clients. The application need the following features: connection (CONNECT message sent by the client node and following acknowledgment from the PANC, the CONNACK)

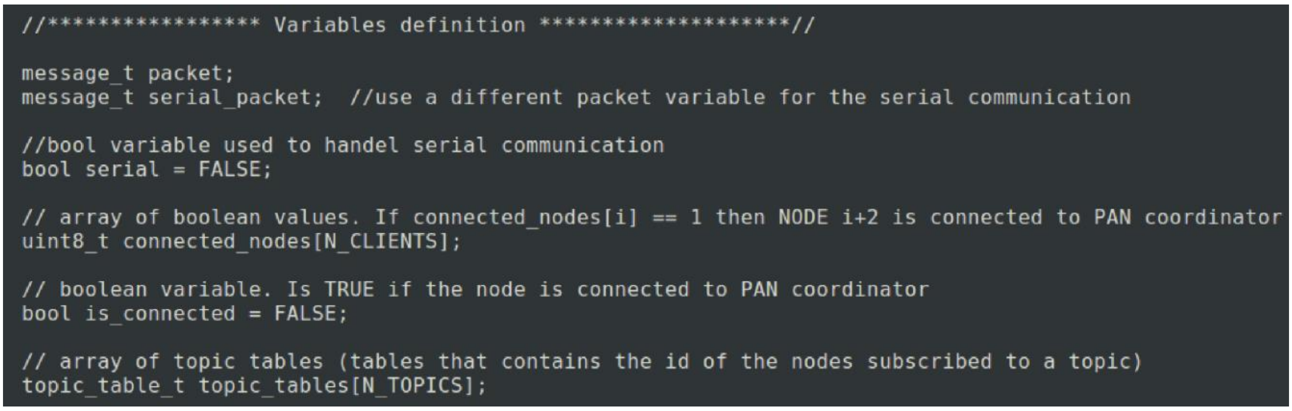
- subscription (SUBSCRIPTION message sent by the client node and following acknowledgment from the PANC, the SUBACK)
- publishing (with QoS=0).

Here it is the topology of the MQTT application we developed:



VARIABLES:

First of all, we declared the global variables used in the code.
The variables are the following:

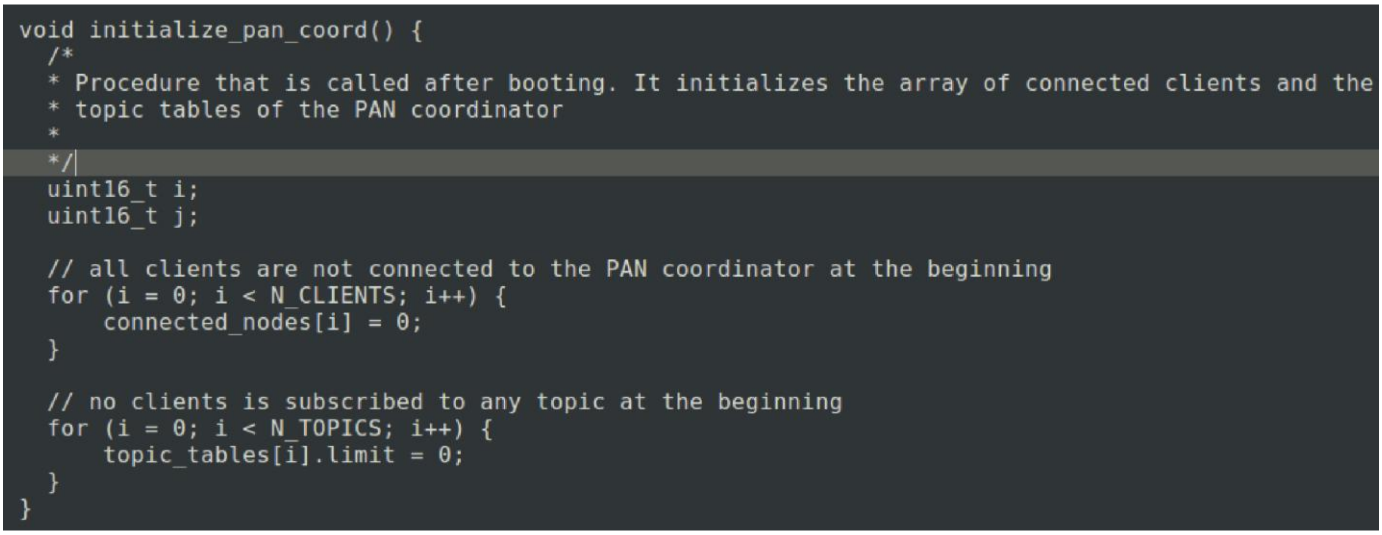


```
//***************** Variables definition ********************//

message_t packet;
message_t serial_packet;  //use a different packet variable for the serial communication

//bool variable used to handel serial communication
bool serial = FALSE;

// array of boolean values. If connected_nodes[i] == 1 then NODE i+2 is connected to PAN coordinator
uint8_t connected_nodes[N_CLIENTS];

// boolean variable. Is TRUE if the node is connected to PAN coordinator
bool is_connected = FALSE;

// array of topic tables (tables that contains the id of the nodes subscribed to a topic)
topic_table_t topic_tables[N_TOPICS];
```

FUNCTIONS:

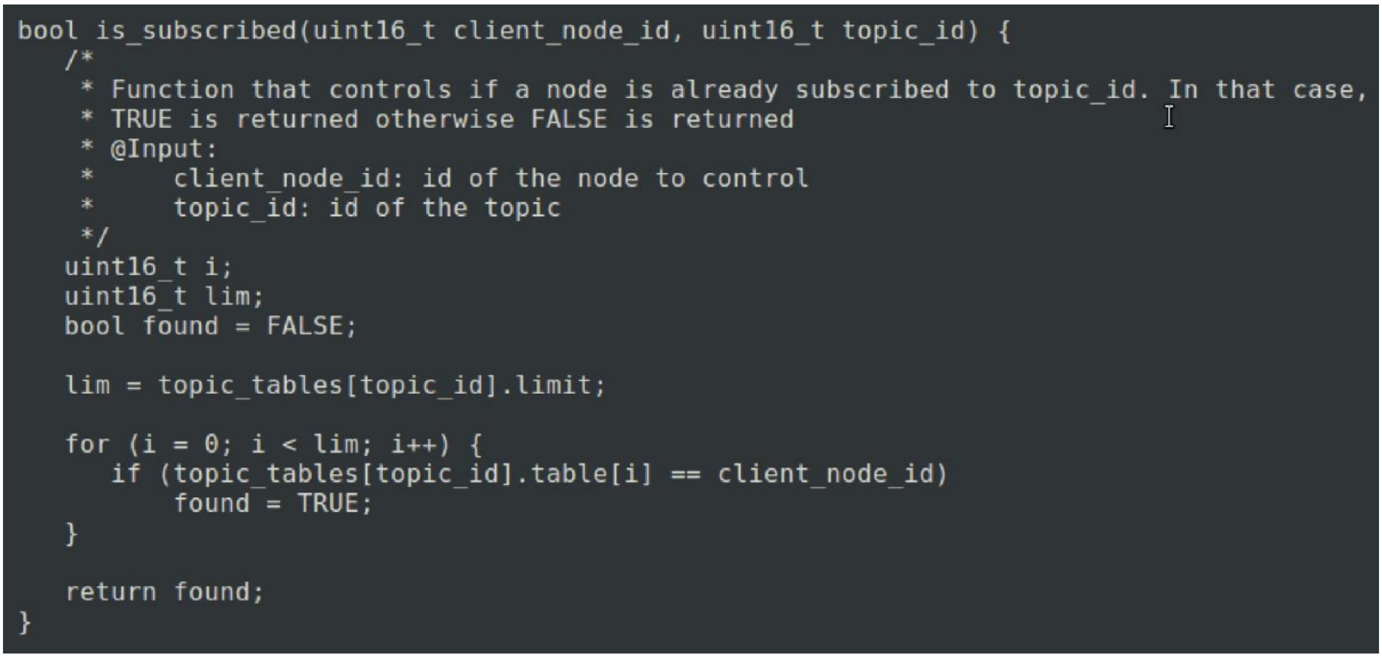
To implement our structure we developed the following functions:

- After booting, the function “initialize_pan_coord” is called to initialize an array, which contains all the connected clients, and a topic table, used by the PANC. At the beginning, all the nodes are not connected to the PANC and any client is subscribed to no topic.



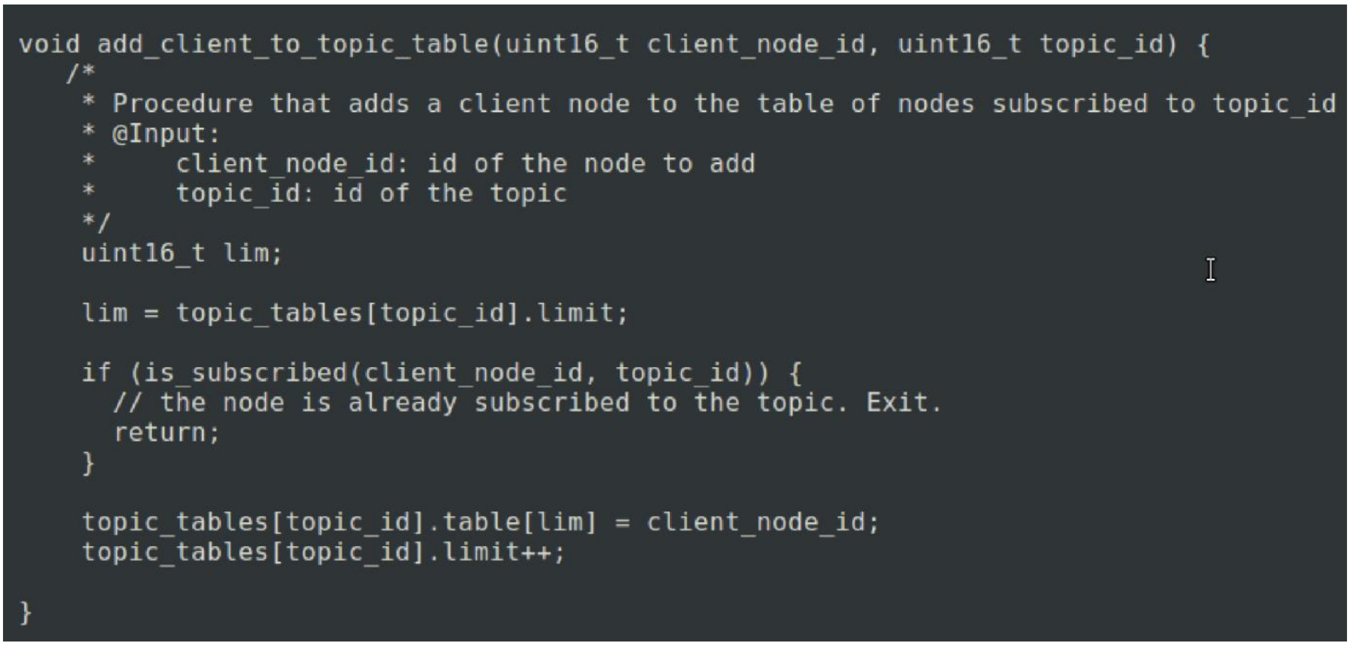
```
void initialize_pan_coord() {
  /*
  * Procedure that is called after booting. It initializes the array of connected clients and the
  * topic tables of the PAN coordinator
  *
  */
  uint16_t i;
  uint16_t j;

  // all clients are not connected to the PAN coordinator at the beginning
  for (i = 0; i < N_CLIENTS; i++) {
      connected_nodes[i] = 0;
  }

  // no clients is subscribed to any topic at the beginning
  for (i = 0; i < N_TOPICS; i++) {
      topic_tables[i].limit = 0;
  }
}
```

- With a function, called “is_subscribed”, we checked if a node is already subscribed to a specific topic: if it is, returns TRUE, otherwise returns FALSE.



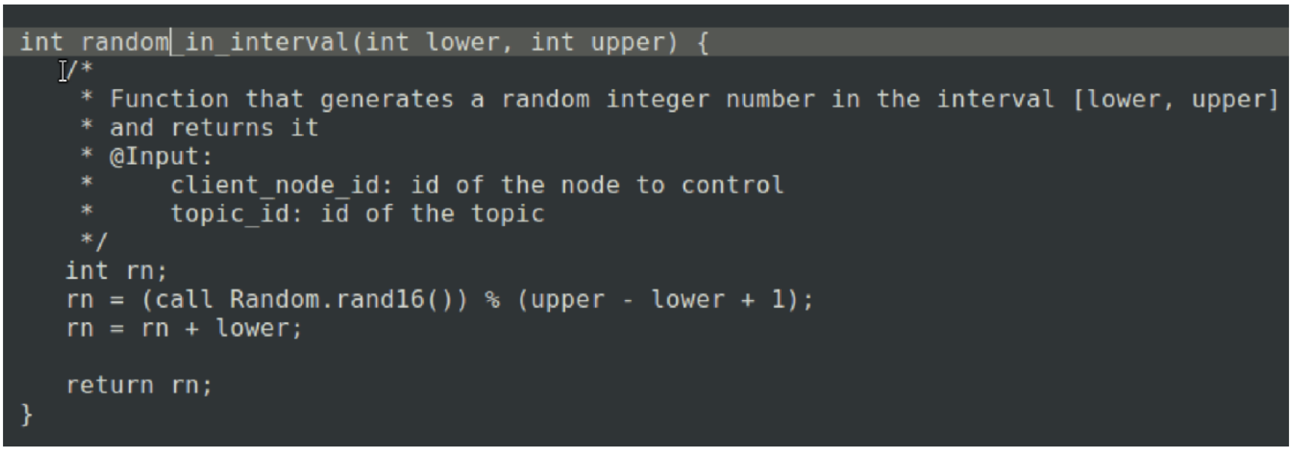
```
bool is_subscribed(uint16_t client_node_id, uint16_t topic_id) {
   /*
    * Function that controls if a node is already subscribed to topic_id. In that case,
    * TRUE is returned otherwise FALSE is returned
    * @Input:
    *     client_node_id: id of the node to control
    *     topic_id: id of the topic
    */
   uint16_t i;
   uint16_t lim;
   bool found = FALSE;

   lim = topic_tables[topic_id].limit;

   for (i = 0; i < lim; i++) {
      if (topic_tables[topic_id].table[i] == client_node_id)
          found = TRUE;
   }

   return found;
}
```

- With the function “add_client_to_topic_table”, we add a client to the table which contains the nodes subscribed to a specific topic.



```
void add_client_to_topic_table(uint16_t client_node_id, uint16_t topic_id) {
   /*
    * Procedure that adds a client node to the table of nodes subscribed to topic_id
    * @Input:
    *     client_node_id: id of the node to add
    *     topic_id: id of the topic
    */
    uint16_t lim;

    lim = topic_tables[topic_id].limit;

    if (is_subscribed(client_node_id, topic_id)) {
      // the node is already subscribed to the topic. Exit.
      return;
    }

    topic_tables[topic_id].table[lim] = client_node_id;
    topic_tables[topic_id].limit++;

}
```

- The function “random_in_interval” generates a random integer number in a specific interval ([lower,upper]) and returns it.



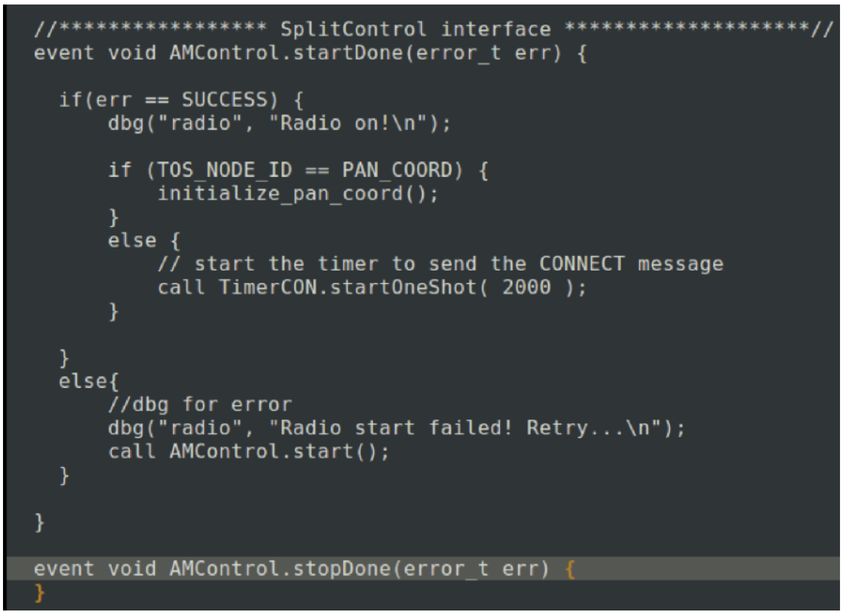
```
int random_in_interval(int lower, int upper) {
   /*
    * Function that generates a random integer number in the interval [lower, upper]
    * and returns it
    * @Input:
    *     client_node_id: id of the node to control
    *     topic_id: id of the topic
    */
   int rn;
   rn = (call Random.rand16()) % (upper - lower + 1);
   rn = rn + lower;

   return rn;
}
```

CONNECTION PHASE:

After the booting, the nodes are initialized, so as the topic tables, and the clients start the connection procedure to the PAN Coordinator.
To establish a connection, each node has to send a connect message to the PANC, which contains 2 fields: the type of the message and the sender node of that message. Note that the type message of the connection messages is defined as “0” in the “PubSub.h” file.
In order to establish a successful connection, every client needs to wait for an acknowledgment by the PANC and, if it doesn’t receive any ACK after 2 seconds, it will try to send the CONNECT message again.

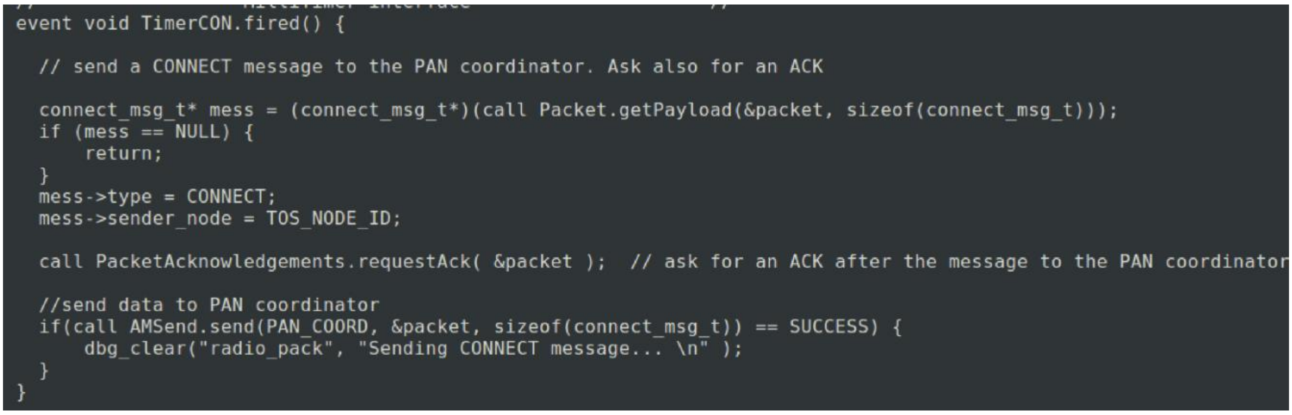
The following figures show the events and the procedures used to perform the connection phase:

- The initialization:



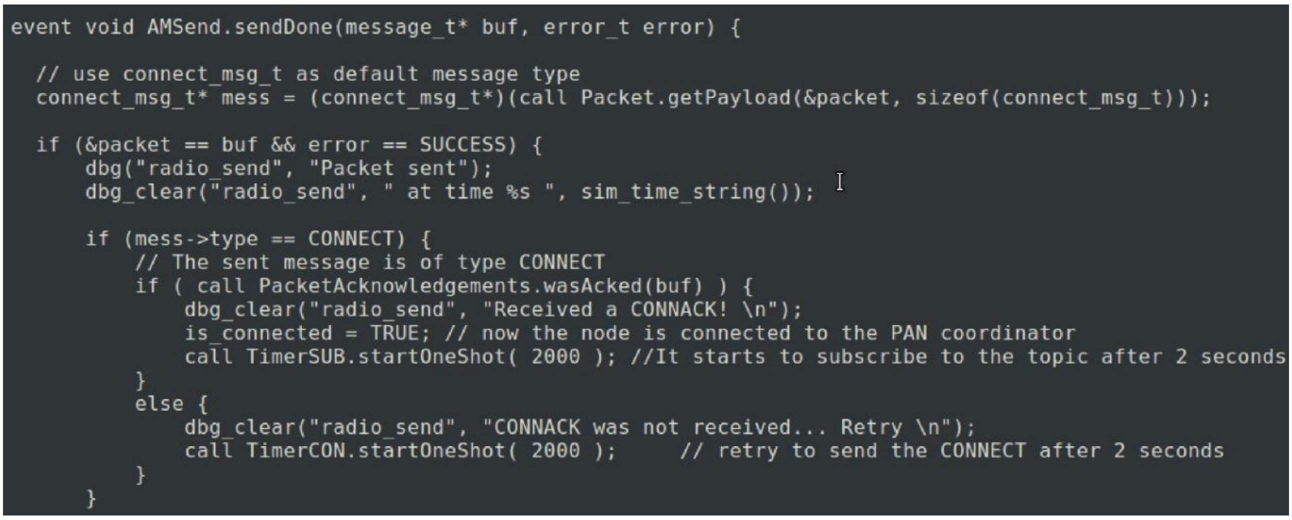
```
//***************** SplitControl interface ********************//
event void AMControl.startDone(error_t err) {

  if(err == SUCCESS) {
      dbg("radio", "Radio on!\n");

      if (TOS_NODE_ID == PAN_COORD) {
          initialize_pan_coord();
      }
      else {
          // start the timer to send the CONNECT message
          call TimerCON.startOneShot( 2000 );
      }

  }
  else{
      //dbg for error
      dbg("radio", "Radio start failed! Retry...\n");
      call AMControl.start();
  }

}

event void AMControl.stopDone(error_t err) {
}
```

- The event “Timer.CON”, which sends a connect message to the PANC and asks for an acknowledgment:



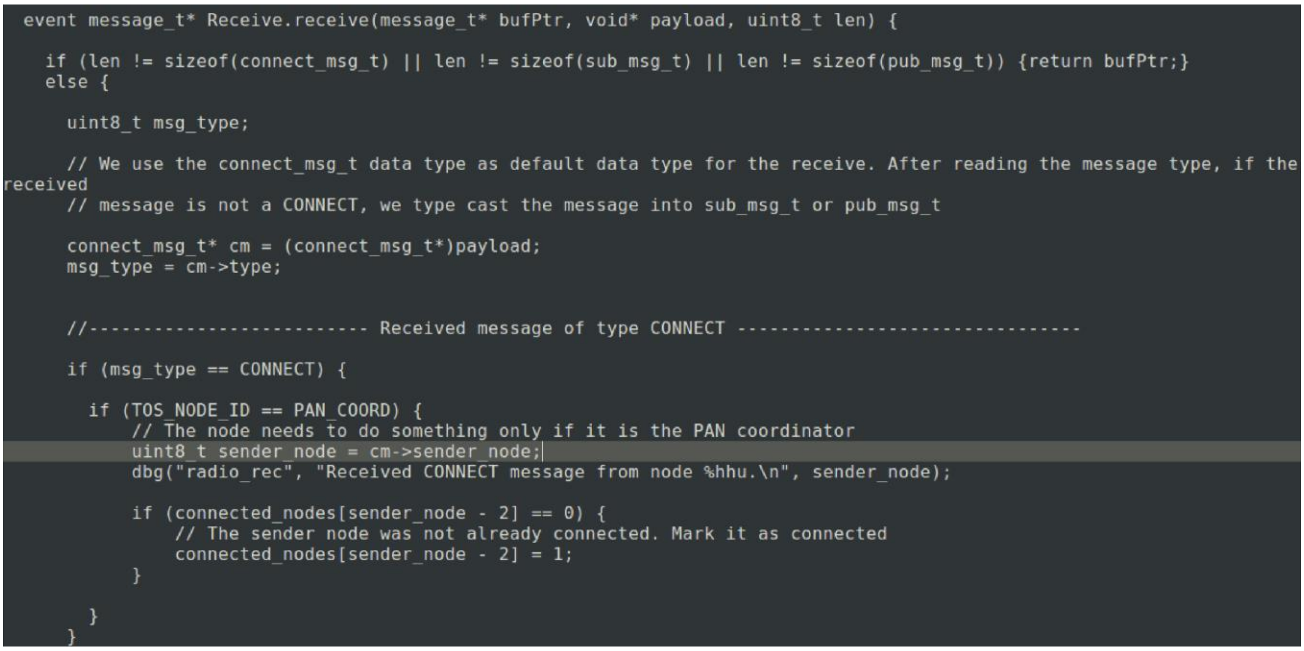
```
//                MilliTimer interface                //
event void TimerCON.fired() {

  // send a CONNECT message to the PAN coordinator. Ask also for an ACK

  connect_msg_t* mess = (connect_msg_t*)(call Packet.getPayload(&packet, sizeof(connect_msg_t)));
  if (mess == NULL) {
      return;
  }
  mess->type = CONNECT;
  mess->sender_node = TOS_NODE_ID;

  call PacketAcknowledgements.requestAck( &packet );  // ask for an ACK after the message to the PAN coordinator

  //send data to PAN coordinator
  if(call AMSend.send(PAN_COORD, &packet, sizeof(connect_msg_t)) == SUCCESS) {
      dbg_clear("radio_pack", "Sending CONNECT message... \n" );
  }
}
```

- The event “send.done”, which is called whenever a send of a message of any type is performed (in this case a connection message) and which performs the request of an acknowledgment or, otherwise, triggers the “Timer.CON” in order to send another connection request:



```
event void AMSend.sendDone(message_t* buf, error_t error) {

  // use connect_msg_t as default message type
  connect_msg_t* mess = (connect_msg_t*)(call Packet.getPayload(&packet, sizeof(connect_msg_t)));

  if (&packet == buf && error == SUCCESS) {
      dbg("radio_send", "Packet sent");
      dbg_clear("radio_send", " at time %s ", sim_time_string());

      if (mess->type == CONNECT) {
          // The sent message is of type CONNECT
          if ( call PacketAcknowledgements.wasAcked(buf) ) {
              dbg_clear("radio_send", "Received a CONNACK! \n");
              is_connected = TRUE; // now the node is connected to the PAN coordinator
              call TimerSUB.startOneShot( 2000 ); //It starts to subscribe to the topic after 2 seconds
          }
          else {
              dbg_clear("radio_send", "CONNACK was not received... Retry \n");
              call TimerCON.startOneShot( 2000 );     // retry to send the CONNECT after 2 seconds
          }
      }
```

- The event “Receive.receive”:



```
 event message_t* Receive.receive(message_t* bufPtr, void* payload, uint8_t len) {

   if (len != sizeof(connect_msg_t) || len != sizeof(sub_msg_t) || len != sizeof(pub_msg_t)) {return bufPtr;}
   else {

     uint8_t msg_type;

     // We use the connect_msg_t data type as default data type for the receive. After reading the message type, if the
received
     // message is not a CONNECT, we type cast the message into sub_msg_t or pub_msg_t

     connect_msg_t* cm = (connect_msg_t*)payload;
     msg_type = cm->type;


     //------------------------- Received message of type CONNECT -------------------------------

     if (msg_type == CONNECT) {

       if (TOS_NODE_ID == PAN_COORD) {
           // The node needs to do something only if it is the PAN coordinator
           uint8_t sender_node = cm->sender_node;
           dbg("radio_rec", "Received CONNECT message from node %hhu.\n", sender_node);

           if (connected_nodes[sender_node - 2] == 0) {
               // The sender node was not already connected. Mark it as connected
               connected_nodes[sender_node - 2] = 1;
           }

       }
     }
```

SUBSCRIPTION PHASE:

In our application, every node has to be subscribed to at least one topic (chosen randomly) in order to receive and/or send publish messages.

The topics are 3 and are defined as follows in the “PubSub.h” file:

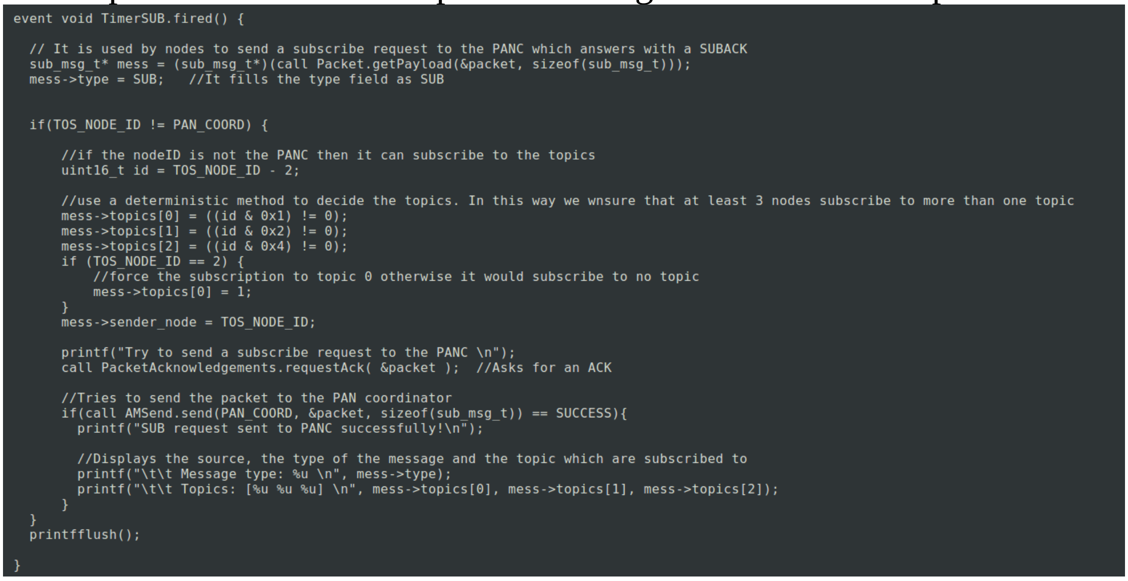
- TEMP = 0
- HUM = 1
- LUM = 2

We structured the subscription messages in 3 fields: type, sender_node and an array of boolean values containing 3 rows, named “topics”.
Each position of the array corresponds to one topic and it can assume 2 different values, “0” and “1”, where “0” means that the node doesn’t want to subscribe to that specific topic and “1” that it want to subscribe to it.
For example the array “[0 1 1]” means that the client node is subscribed to the second and third topic (i.e. is subscribed to “HUM” and “LUM”), but not to the first one (i.e. not to “TEMP”).

As in the connection phase, also in the subscription phase every subscription message needs to be acknowledged from the PAN Coordinator, otherwise the client node tries to send it again after a timeout of 2 seconds.

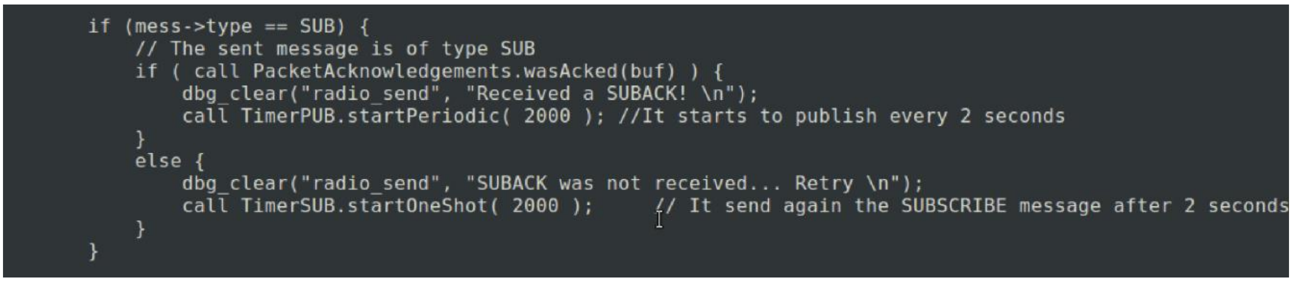
Once a node is subscribed to a topic, it can publish messages on that topic and receive messages about the topics, which is subscribed to.

- We develop this event in order to set a subscribe message and we make the subscription to the various topics according to a deterministic procedure:



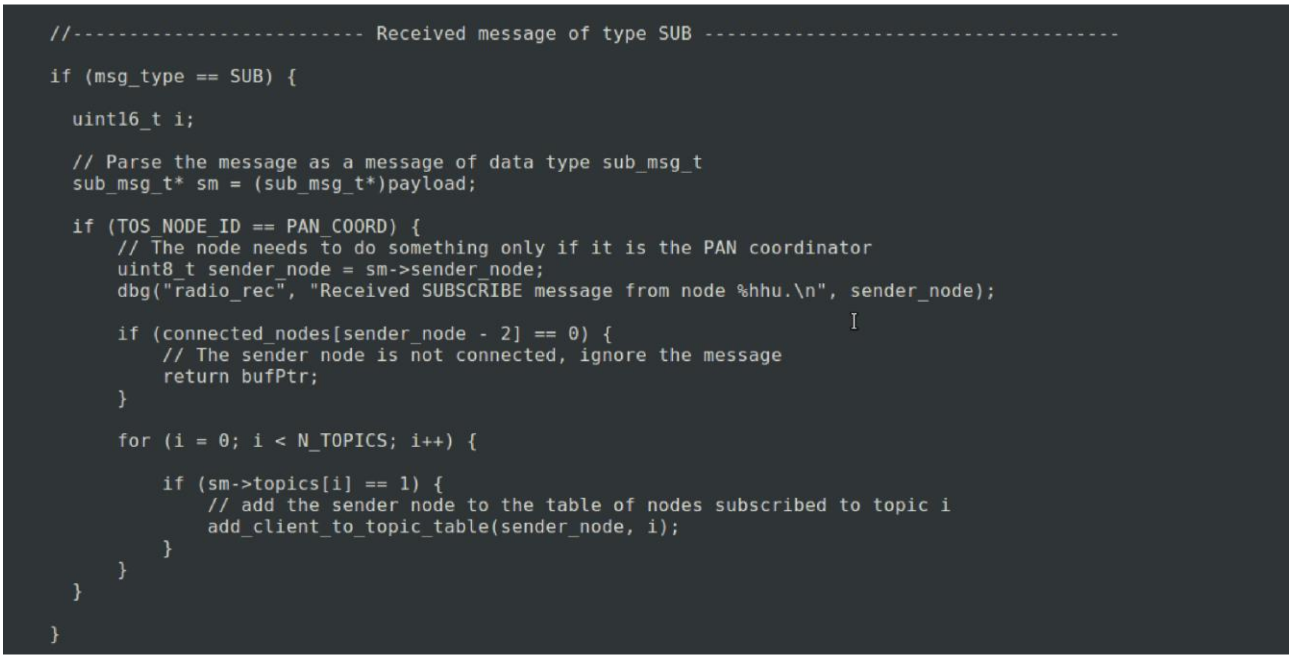
```
event void TimerSUB.fired() {

  // It is used by nodes to send a subscribe request to the PANC which answers with a SUBACK
  sub_msg_t* mess = (sub_msg_t*)(call Packet.getPayload(&packet, sizeof(sub_msg_t)));
  mess->type = SUB;   //It fills the type field as SUB


  if(TOS_NODE_ID != PAN_COORD) {

      //if the nodeID is not the PANC then it can subscribe to the topics
      uint16_t id = TOS_NODE_ID - 2;

      //use a deterministic method to decide the topics. In this way we wnsure that at least 3 nodes subscribe to more than one topic
      mess->topics[0] = ((id & 0x1) != 0);
      mess->topics[1] = ((id & 0x2) != 0);
      mess->topics[2] = ((id & 0x4) != 0);
      if (TOS_NODE_ID == 2) {
          //force the subscription to topic 0 otherwise it would subscribe to no topic
          mess->topics[0] = 1;
      }
      mess->sender_node = TOS_NODE_ID;

      printf("Try to send a subscribe request to the PANC \n");
      call PacketAcknowledgements.requestAck( &packet );  //Asks for an ACK

      //Tries to send the packet to the PAN coordinator
      if(call AMSend.send(PAN_COORD, &packet, sizeof(sub_msg_t)) == SUCCESS){
        printf("SUB request sent to PANC successfully!\n");

        //Displays the source, the type of the message and the topic which are subscribed to
        printf("\t\t Message type: %u \n", mess->type);
        printf("\t\t Topics: [%u %u %u] \n", mess->topics[0], mess->topics[1], mess->topics[2]);
      }
  }
  printfflush();

}
```

- If the sent message is a subscribe message, the PANC should reply with an acknowledgment, otherwise it will send it again after 2 seconds:



```
    if (mess->type == SUB) {
        // The sent message is of type SUB
        if ( call PacketAcknowledgements.wasAcked(buf) ) {
            dbg_clear("radio_send", "Received a SUBACK! \n");
            call TimerPUB.startPeriodic( 2000 ); //It starts to publish every 2 seconds
        }
        else {
            dbg_clear("radio_send", "SUBACK was not received... Retry \n");
            call TimerSUB.startOneShot( 2000 );     // It send again the SUBSCRIBE message after 2 seconds
        }
    }
```

- The received message is now a subscribe message and it is received by the PANC. If the node isn’t connected, the PANC ignore that message; otherwise, the client node will be added to the tables of nodes subscribed to a specific topic:



```
//------------------------- Received message of type SUB -------------------------------------

if (msg_type == SUB) {

  uint16_t i;

  // Parse the message as a message of data type sub_msg_t
  sub_msg_t* sm = (sub_msg_t*)payload;

  if (TOS_NODE_ID == PAN_COORD) {
      // The node needs to do something only if it is the PAN coordinator
      uint8_t sender_node = sm->sender_node;
      dbg("radio_rec", "Received SUBSCRIBE message from node %hhu.\n", sender_node);

      if (connected_nodes[sender_node - 2] == 0) {
          // The sender node is not connected, ignore the message
          return bufPtr;
      }

      for (i = 0; i < N_TOPICS; i++) {

          if (sm->topics[i] == 1) {
              // add the sender node to the table of nodes subscribed to topic i
              add_client_to_topic_table(sender_node, i);
          }
      }
  }

}
```

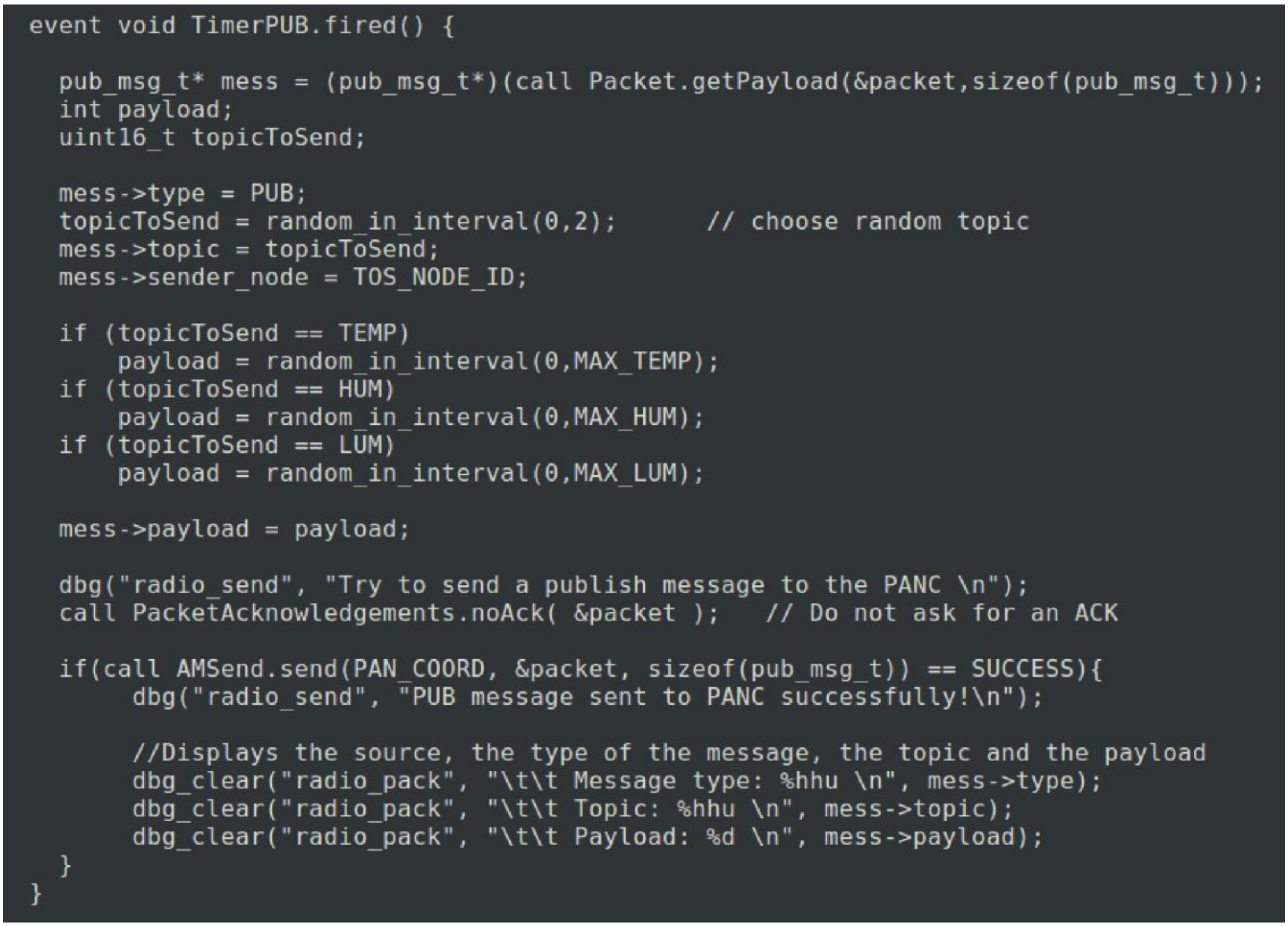
PUBLICATION PHASE:

In the publication phase a client node sends a publish message to the PANC; this message is composed by 4 fields: the message type, the sender node, the topic of the publication and the payload.

Once a client node sends a publish message, it will be chosen a random topic through the function “random_in_interval” and a random value for the payload is created (its value is set in different ranges of possible values, in dependance on the chosen topic).

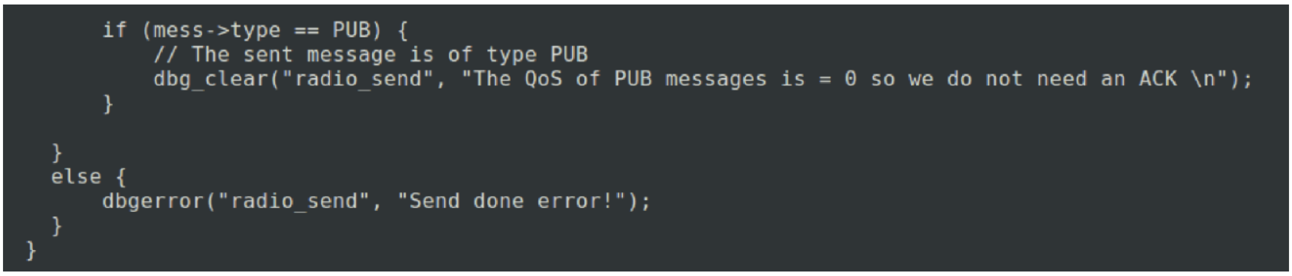
For our implementation the QoS (the quality of service) for the publication messages is asked to be = 0, i.e. the message is sent and it needs no acknowledgment of being received. So, there is no certainty that the message is sent correctly but the procedure is the most simple and the less expensive.

- When the event “TimerPUB.Fired” is called, a random topic is chosen and a random value for the payload is generates, as said above:



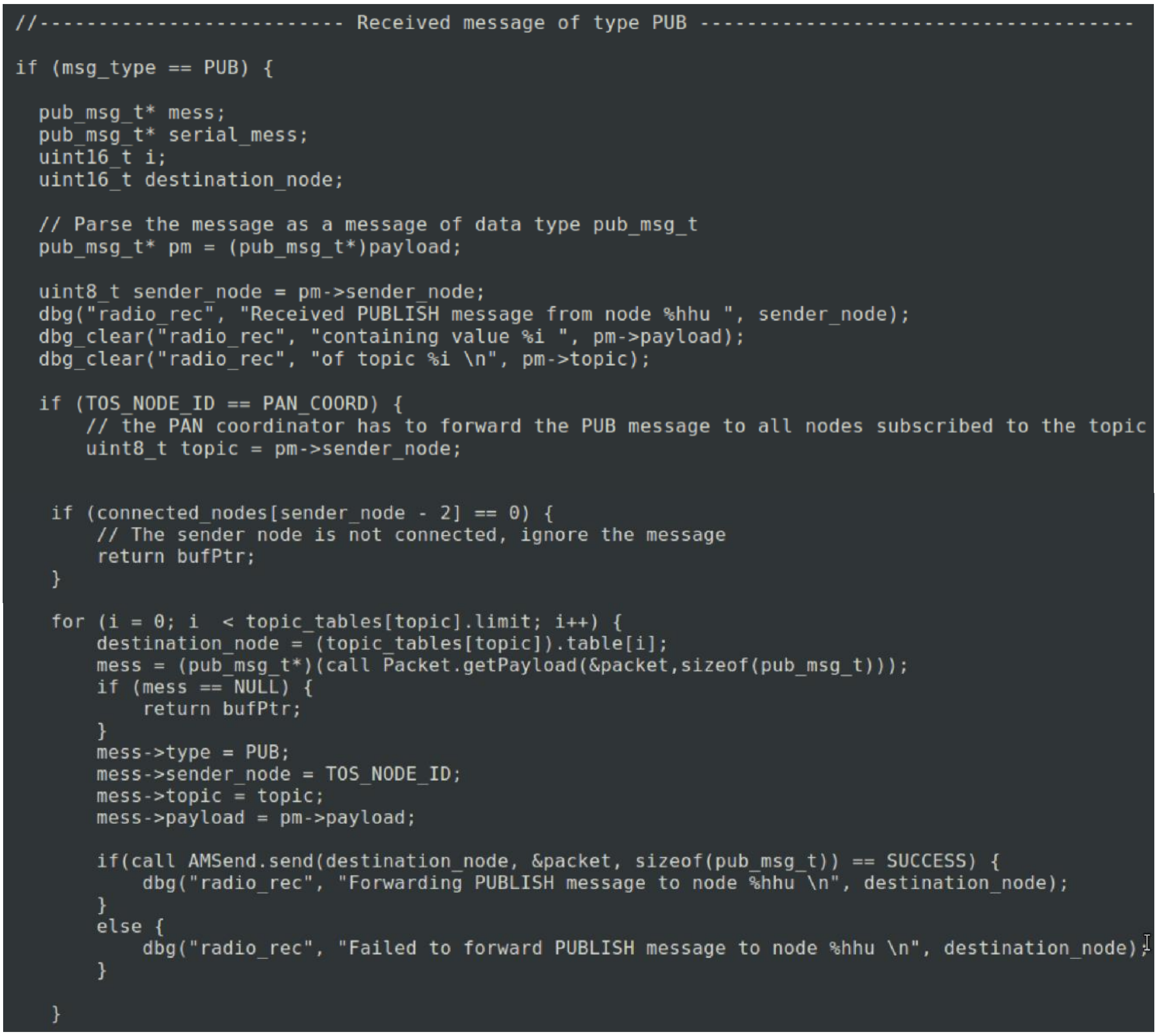
```
event void TimerPUB.fired() {

  pub_msg_t* mess = (pub_msg_t*)(call Packet.getPayload(&packet,sizeof(pub_msg_t)));
  int payload;
  uint16_t topicToSend;

  mess->type = PUB;
  topicToSend = random_in_interval(0,2);      // choose random topic
  mess->topic = topicToSend;
  mess->sender_node = TOS_NODE_ID;

  if (topicToSend == TEMP)
      payload = random_in_interval(0,MAX_TEMP);
  if (topicToSend == HUM)
      payload = random_in_interval(0,MAX_HUM);
  if (topicToSend == LUM)
      payload = random_in_interval(0,MAX_LUM);

  mess->payload = payload;

  dbg("radio_send", "Try to send a publish message to the PANC \n");
  call PacketAcknowledgements.noAck( &packet );   // Do not ask for an ACK

  if(call AMSend.send(PAN_COORD, &packet, sizeof(pub_msg_t)) == SUCCESS){
       dbg("radio_send", "PUB message sent to PANC successfully!\n");

       //Displays the source, the type of the message, the topic and the payload
       dbg_clear("radio_pack", "\t\t Message type: %hhu \n", mess->type);
       dbg_clear("radio_pack", "\t\t Topic: %hhu \n", mess->topic);
       dbg_clear("radio_pack", "\t\t Payload: %d \n", mess->payload);
  }
}
```

- The QoS is set to 0, so we need no acknowledgment:



```
      if (mess->type == PUB) {
          // The sent message is of type PUB
          dbg_clear("radio_send", "The QoS of PUB messages is = 0 so we do not need an ACK \n");
      }

  }
  else {
      dbgerror("radio_send", "Send done error!");
  }
}
```

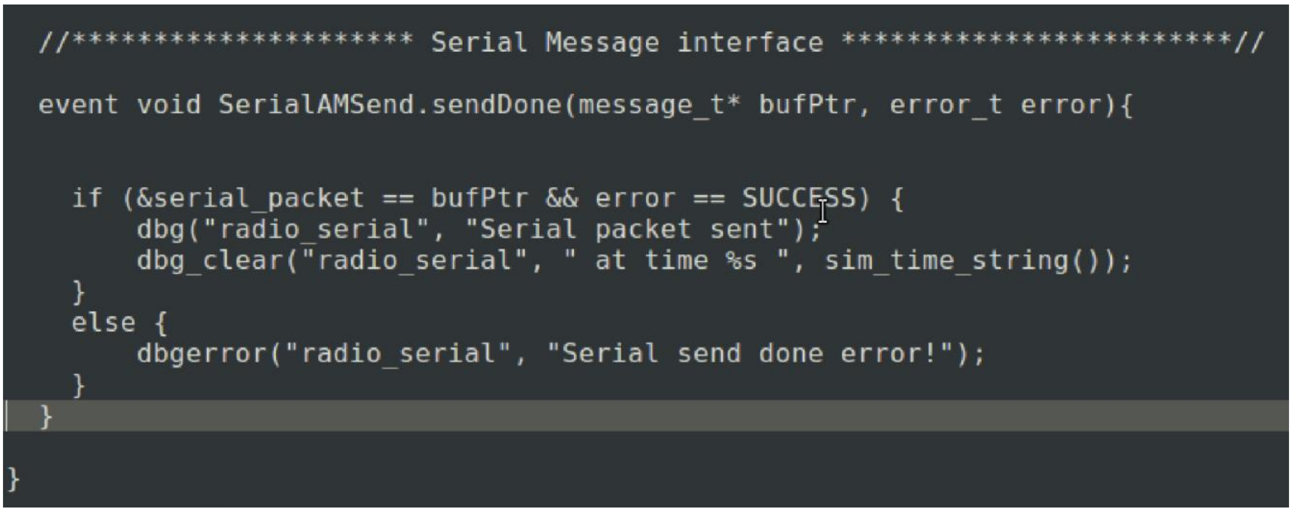
- In this case, the message needs to be sent to the PANC and then needs to be forwarded to all the clients who are subscribed to the topic of publication.
  If the sending fails, the procedure will retry to send the message again (at the PANC and/or at the clients nodes).



```
//------------------------- Received message of type PUB ------------------------------------

if (msg_type == PUB) {

  pub_msg_t* mess;
  pub_msg_t* serial_mess;
  uint16_t i;
  uint16_t destination_node;

  // Parse the message as a message of data type pub_msg_t
  pub_msg_t* pm = (pub_msg_t*)payload;

  uint8_t sender_node = pm->sender_node;
  dbg("radio_rec", "Received PUBLISH message from node %hhu ", sender_node);
  dbg_clear("radio_rec", "containing value %i ", pm->payload);
  dbg_clear("radio_rec", "of topic %i \n", pm->topic);

  if (TOS_NODE_ID == PAN_COORD) {
      // the PAN coordinator has to forward the PUB message to all nodes subscribed to the topic
      uint8_t topic = pm->sender_node;


   if (connected_nodes[sender_node - 2] == 0) {
       // The sender node is not connected, ignore the message
       return bufPtr;
   }

   for (i = 0; i  < topic_tables[topic].limit; i++) {
       destination_node = (topic_tables[topic]).table[i];
       mess = (pub_msg_t*)(call Packet.getPayload(&packet,sizeof(pub_msg_t)));
       if (mess == NULL) {
           return bufPtr;
       }
       mess->type = PUB;
       mess->sender_node = TOS_NODE_ID;
       mess->topic = topic;
       mess->payload = pm->payload;

       if(call AMSend.send(destination_node, &packet, sizeof(pub_msg_t)) == SUCCESS) {
           dbg("radio_rec", "Forwarding PUBLISH message to node %hhu \n", destination_node);
       }
       else {
           dbg("radio_rec", "Failed to forward PUBLISH message to node %hhu \n", destination_node);
       }

   }
```

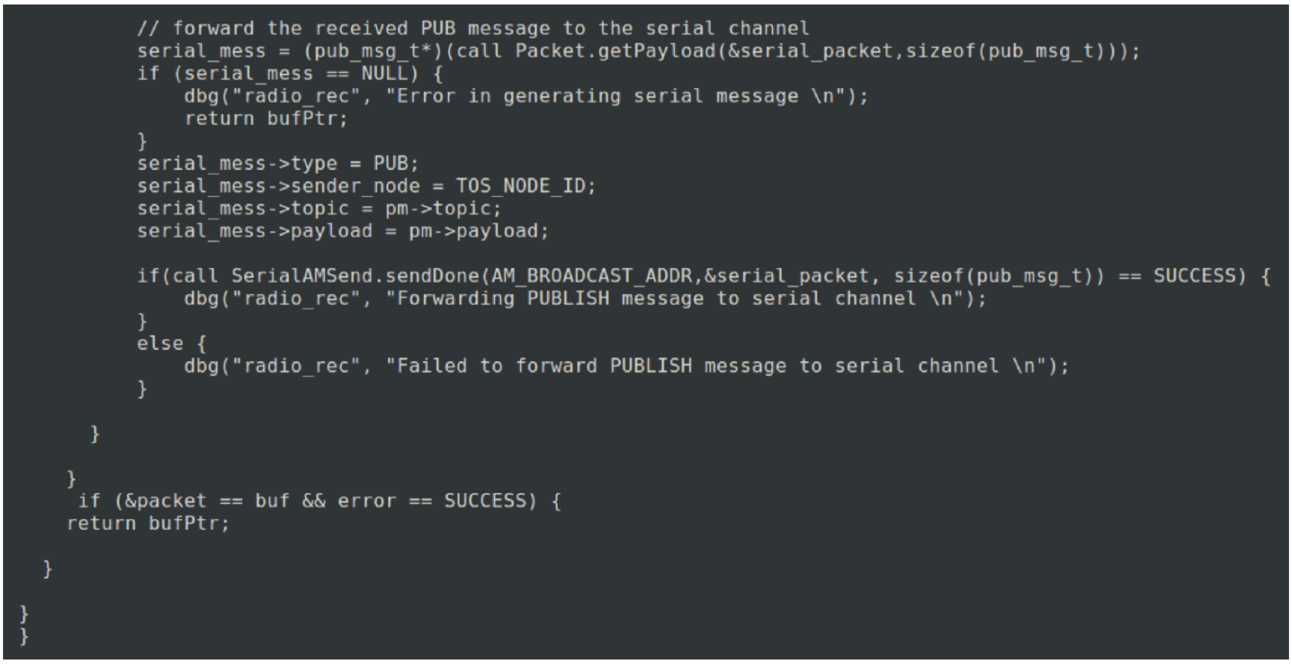
We also developed the Serial messages interface and implemented the receive event in order to forward the received publish messages to the serial channel.
Moreover, we used another packet variable for the serial communication, called in the variables definition as “serial_packet”, but has the same 4 fields as the publication message packet and it maintains the same type (i.e. the publication type).

- Here it is the Serial message interface:



```
  //******************** Serial Message interface ***********************//

  event void SerialAMSend.sendDone(message_t* bufPtr, error_t error){


    if (&serial_packet == bufPtr && error == SUCCESS) {
        dbg("radio_serial", "Serial packet sent");
        dbg_clear("radio_serial", " at time %s ", sim_time_string());
    }
    else {
        dbgerror("radio_serial", "Serial send done error!");
    }
  }

}
```

- Here it is the implementation of the receive event, which calls the Serial message interface and forwards the messages to the serial channel:



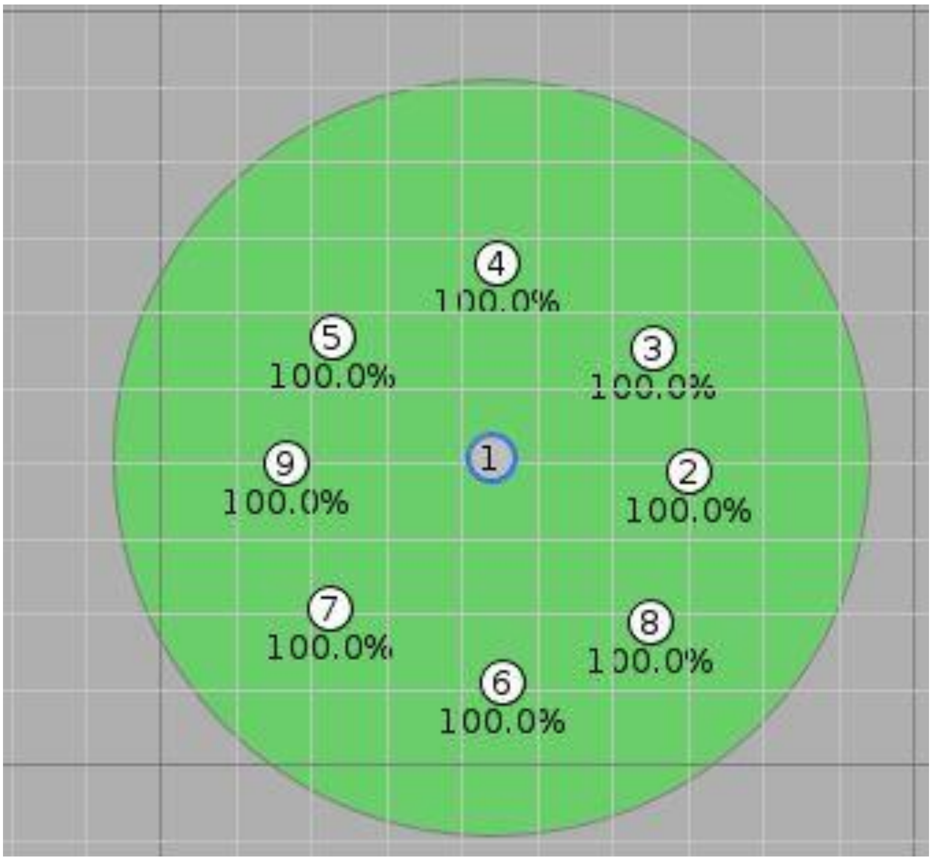
```
            // forward the received PUB message to the serial channel
            serial_mess = (pub_msg_t*)(call Packet.getPayload(&serial_packet,sizeof(pub_msg_t)));
            if (serial_mess == NULL) {
                dbg("radio_rec", "Error in generating serial message \n");
                return bufPtr;
            }
            serial_mess->type = PUB;
            serial_mess->sender_node = TOS_NODE_ID;
            serial_mess->topic = pm->topic;
            serial_mess->payload = pm->payload;

            if(call SerialAMSend.sendDone(AM_BROADCAST_ADDR,&serial_packet, sizeof(pub_msg_t)) == SUCCESS) {
                dbg("radio_rec", "Forwarding PUBLISH message to serial channel \n");
            }
            else {
                dbg("radio_rec", "Failed to forward PUBLISH message to serial channel \n");
            }

        }

      }
       if (&packet == buf && error == SUCCESS) {
      return bufPtr;

    }

  }
  }
```

SIMULATION (WITH TOSSIM OR COOJA) AND NODE-RED:

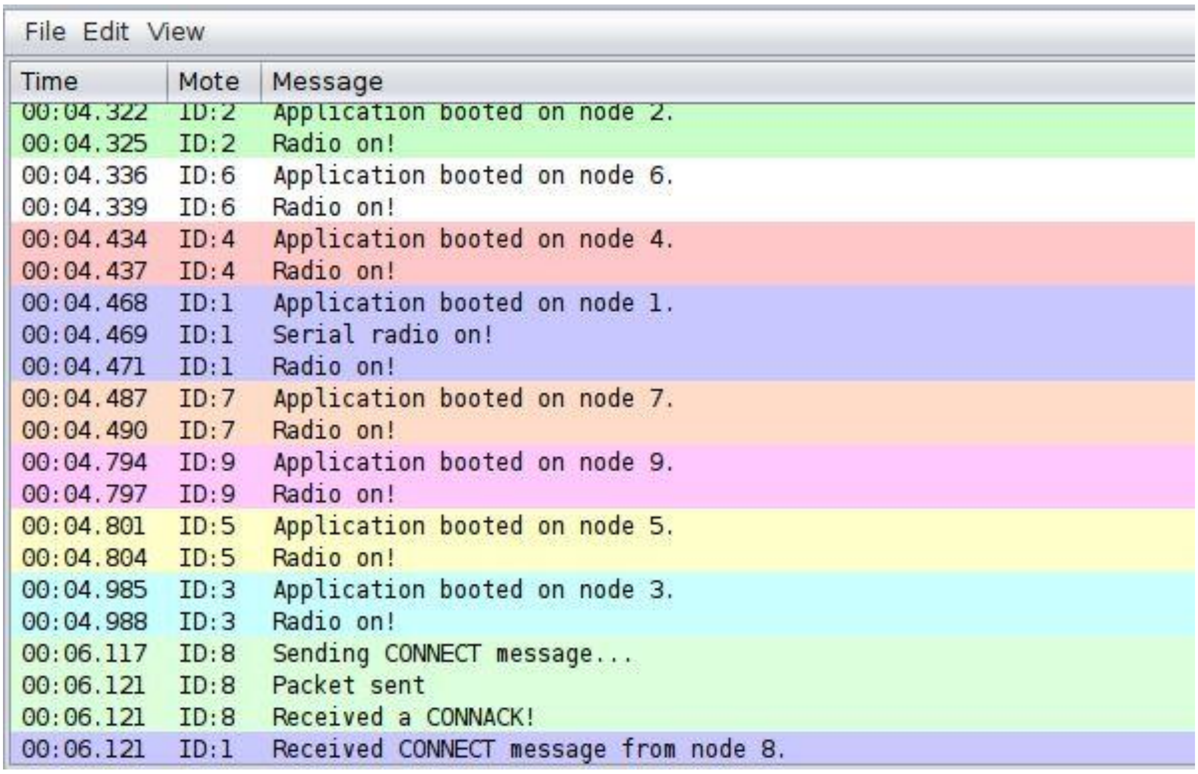
Firstly, for the simulation we thought on using MicaZ and TOSSIM, but we had to change our idea and choose Cooja, because we weren't able to connect TOSSIM with NODE-Red through a serial port. Moreover, we were forced to use TelosB, and not MicaZ, because MicaZ doesn't work well with Cooja.
We faced soon another problem: since the debug statements are not supported by Cooja, we decided to use the Printf statements, which allowed us to verify the correctness of the procedure. But, once we started to send the messages to Node-RED, we had to cancel them because they seemed to be received as serial messages.

We insert here some screenshots from the simulation on Cooja.

Here we put the topology of the network:

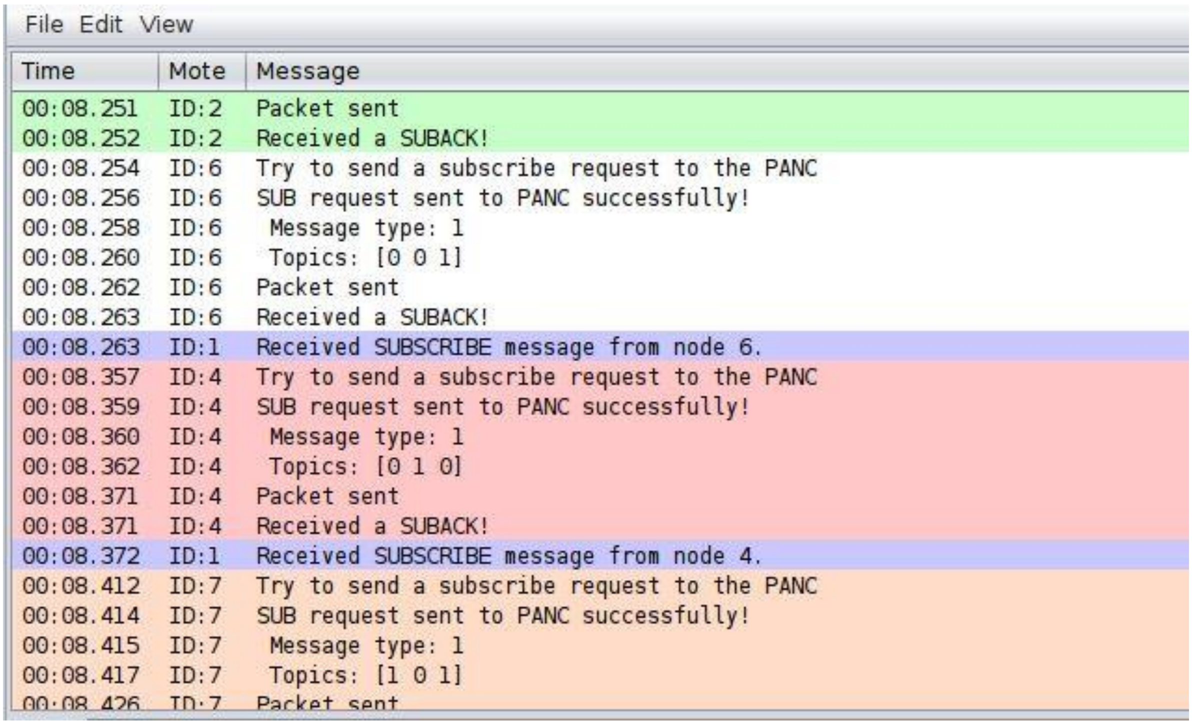


Here we put an example of CONNECT messages:



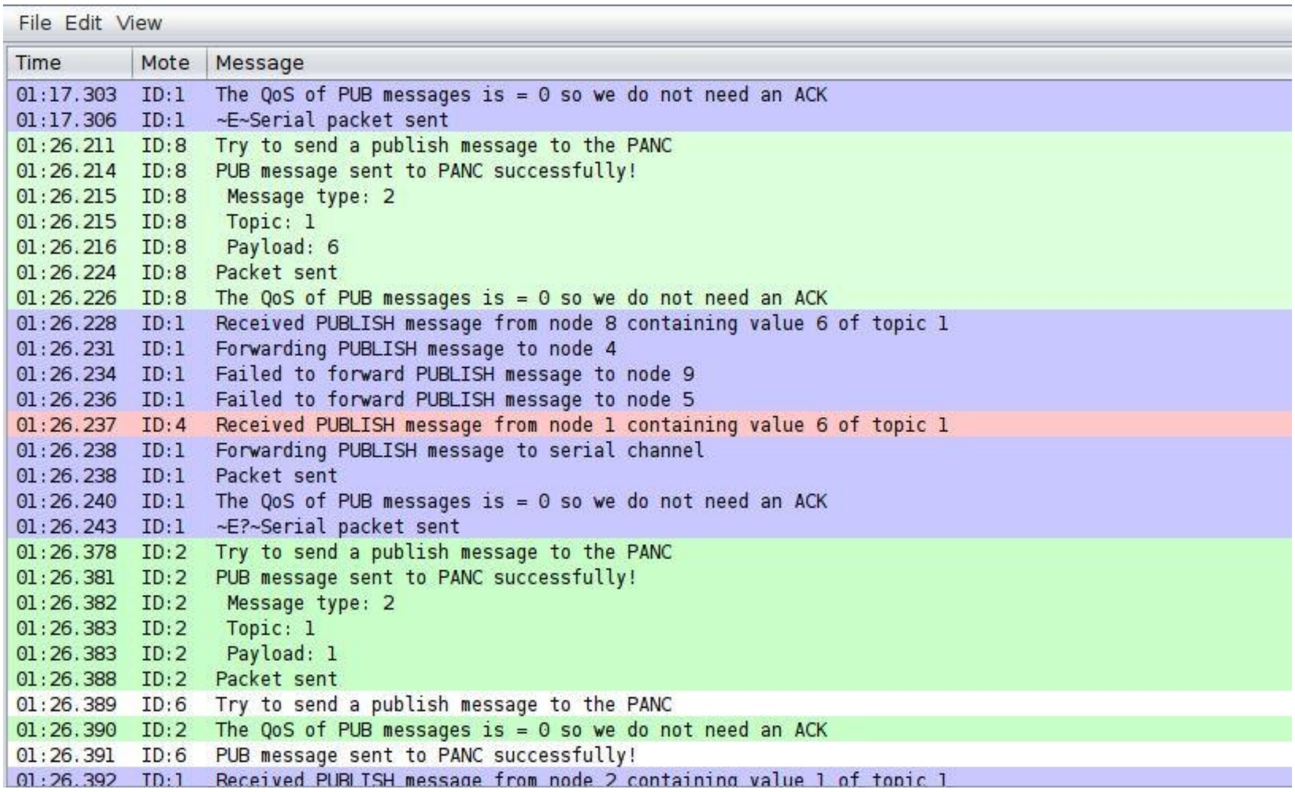
```
Time        Mote   Message
00:04.322   ID:2   Application booted on node 2.
00:04.325   ID:2   Radio on!
00:04.336   ID:6   Application booted on node 6.
00:04.339   ID:6   Radio on!
00:04.434   ID:4   Application booted on node 4.
00:04.437   ID:4   Radio on!
00:04.468   ID:1   Application booted on node 1.
00:04.469   ID:1   Serial radio on!
00:04.471   ID:1   Radio on!
00:04.487   ID:7   Application booted on node 7.
00:04.490   ID:7   Radio on!
00:04.794   ID:9   Application booted on node 9.
00:04.797   ID:9   Radio on!
00:04.801   ID:5   Application booted on node 5.
00:04.804   ID:5   Radio on!
00:04.985   ID:3   Application booted on node 3.
00:04.988   ID:3   Radio on!
00:06.117   ID:8   Sending CONNECT message...
00:06.121   ID:8   Packet sent
00:06.121   ID:8   Received a CONNACK!
00:06.121   ID:1   Received CONNECT message from node 8.
```

Here we put an example of SUBSCRIBE messages:



```
Time        Mote   Message
00:08.251   ID:2   Packet sent
00:08.252   ID:2   Received a SUBACK!
00:08.254   ID:6   Try to send a subscribe request to the PANC
00:08.256   ID:6   SUB request sent to PANC successfully!
00:08.258   ID:6    Message type: 1
00:08.260   ID:6    Topics: [0 0 1]
00:08.262   ID:6   Packet sent
00:08.263   ID:6   Received a SUBACK!
00:08.263   ID:1   Received SUBSCRIBE message from node 6.
00:08.357   ID:4   Try to send a subscribe request to the PANC
00:08.359   ID:4   SUB request sent to PANC successfully!
00:08.360   ID:4    Message type: 1
00:08.362   ID:4    Topics: [0 1 0]
00:08.371   ID:4   Packet sent
00:08.371   ID:4   Received a SUBACK!
00:08.372   ID:1   Received SUBSCRIBE message from node 4.
00:08.412   ID:7   Try to send a subscribe request to the PANC
00:08.414   ID:7   SUB request sent to PANC successfully!
00:08.415   ID:7    Message type: 1
00:08.417   ID:7    Topics: [1 0 1]
00:08.426   ID:7   Packet sent
```

Here we put an example PUBLISH and serial messages:

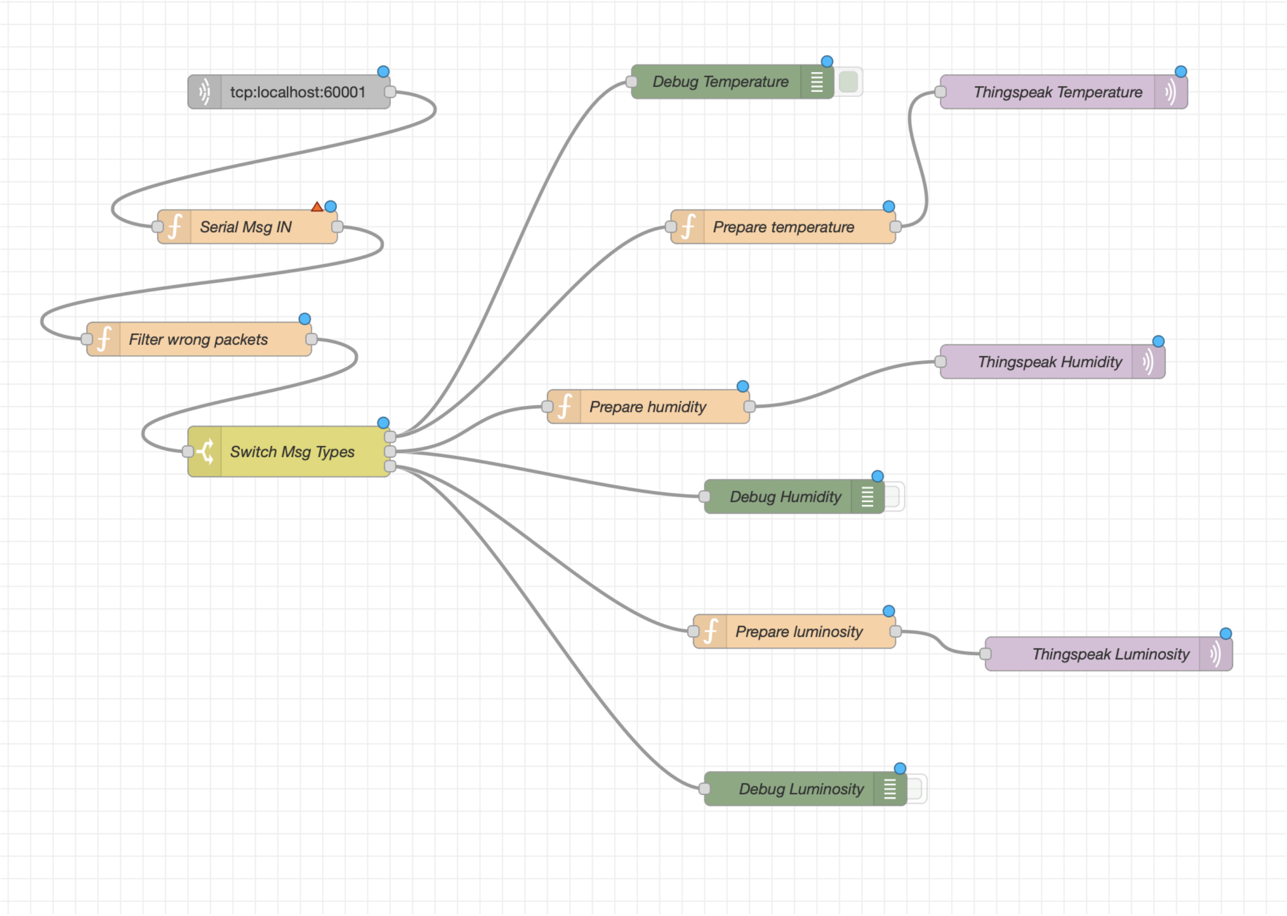


```
Time       Mote  Message
01:17.303  ID:1  The QoS of PUB messages is = 0 so we do not need an ACK
01:17.306  ID:1  ~E~Serial packet sent
01:26.211  ID:8  Try to send a publish message to the PANC
01:26.214  ID:8  PUB message sent to PANC successfully!
01:26.215  ID:8   Message type: 2
01:26.215  ID:8   Topic: 1
01:26.216  ID:8   Payload: 6
01:26.224  ID:8  Packet sent
01:26.226  ID:8  The QoS of PUB messages is = 0 so we do not need an ACK
01:26.228  ID:1  Received PUBLISH message from node 8 containing value 6 of topic 1
01:26.231  ID:1  Forwarding PUBLISH message to node 4
01:26.234  ID:1  Failed to forward PUBLISH message to node 9
01:26.236  ID:1  Failed to forward PUBLISH message to node 5
01:26.237  ID:4  Received PUBLISH message from node 1 containing value 6 of topic 1
01:26.238  ID:1  Forwarding PUBLISH message to serial channel
01:26.238  ID:1  Packet sent
01:26.240  ID:1  The QoS of PUB messages is = 0 so we do not need an ACK
01:26.243  ID:1  ~E?~Serial packet sent
01:26.378  ID:2  Try to send a publish message to the PANC
01:26.381  ID:2  PUB message sent to PANC successfully!
01:26.382  ID:2   Message type: 2
01:26.383  ID:2   Topic: 1
01:26.383  ID:2   Payload: 1
01:26.388  ID:2  Packet sent
01:26.389  ID:6  Try to send a publish message to the PANC
01:26.390  ID:2  The QoS of PUB messages is = 0 so we do not need an ACK
01:26.391  ID:6  PUB message sent to PANC successfully!
01:26.392  ID:1  Received PUBLISH message from node 2 containing value 1 of topic 1
```

As regards Node-RED, with the TCP node, we were able to receive the serial messages coming from the Cooja simulation of our TinyOs application.
We had to make a “filter” function, called “Filter wrong packets”, in order to eliminate some strange with impossible values appearing during the simulation, which were not sent by the PANC (so their sender node wasn’t the node 1). With this function we discarded all the messages whose sender node wasn’t the PANC and whose values are over the maximum imposed in the “PubSub.h” file.
Firstly, we tried to represent all the datas in one single chart, but it seems confused and difficult to read easily.
So we thought useful to separate the datas concerning different topics, in this way the graphs result more clear and readable.
Due to this necessity, we separated the messages by their topic thanks to a “switch” node and prepare them to be sent to Thingspeak through a “mqqt out” node.

The link to the public Thingspeak channel is the following:
https://thingspeak.com/channels/2235049

Here it is the Node-RED scheme:


tcp:localhost:60001
Serial Msg IN
Filter wrong packets
Switch Msg Types
Debug Temperature
Prepare temperature
Thingspeak Temperature
Prepare humidity
Thingspeak Humidity
Debug Humidity
Prepare luminosity
Thingspeak Luminosity
Debug Luminosity