# IoT 2023 - Project 2: LoraWAN-like sensor networks

## Group Members

Team Leader: Riaz Luis Ahmed.

Personal Code: 10372022

Second Member: Mario Cela.

Personal Code: 10685242

## Report

### SenseNet.h

First of all, in this file we can find the **structure** we used for the messages:

```
typedef nx_struct sense_msg {
  nx_uint8_t type;
  nx_uint16_t msg_id;
  nx_uint16_t data;
  nx_uint16_t sender;
  nx_uint16_t destination;
} sense_msg_t;
```

It follows a brief explanation of the attributes of the structure:

- **type**: this field can assume just the values 0 and 1. When a message is a data message, so a message that has been generated by the sensor, it will assume the 0 value. If it is an ack, so a message that has been generated by the sensor in response to a data message, it will assume the 1 value.
- **msg_id**: the pair <TOS_NODE_ID, msg_id> identifies uniquely a message. We decided to manage the packet sequence number by using a counter which is incremented every time a new message is sent by a sensor node. This means that

in the network there will be 5 messages with msg_id set to 0, but we uniquely identify a message by considering also who sent each message.

- **data**: it contains an integer vale inside the interval [0, 99]. The number is randomly generated using the Random interface.
- **sender**: when a message is sent by a sensor node, this value is equal to the TOS_NODE_ID of the node itself. It the message is sent by the server, we decided to set the sender value equal to the id of the gateway which forwarded the data message to the server.
- **destination**: when a message is sent by a sensor node, this value is equal to the gateway the message is being sent to. Then, when the gateway receives the message it checks the type, and if it is a data message automatically forwards it to the server. The benefit of the choice is that when the server receives a data message and generates the ack, it needs to know who is the gateway which forwarded the message. In this way, we include this information inside the destination field, since the topology of the network shows just one sink so the forwarding to the server can be done without any message parsing.

  Finally, if the message is being set by the server, the destination is equal to the sender of the data message, since the ack needs to reach the original sender of the message.

Then, we have other two structures used for re-transmissions and duplicate suppression, respectively.

They are:

1. **last_message_transmitted**:

   ```
   typedef struct last_message_transmitted {
     sense_msg_t sense_msg;
     bool ack_received;
   } last_message_transmitted;
   ```

   Each node has a variable declared as:

   ```
   last_message_transmitted msg_tx;
   ```

This variable is used just by the sensor nodes.

Once a sensor node is creating and transmitting a new data message, it will set the `sense_msg` attribute of the structure with the content of the new message.

After calling the `generate_send` function, the sensor node will wait 1000ms to receive the ack.

When Timer2 fires out, it checks `msg_tx.ack_received`: if it is equal to TRUE, it means that the ack has been received and that there is no need of re-transmission; if it is equal to FALSE, it puts inside the payload of `globalpacket` and generates the new send.

2. **last_message_received**:

```
typedef struct last_message_received {
  uint16_t msg_id;
  uint16_t gateway;
  bool retransmitted;
} last_message_received;
```

Each node has a variable declared as:

```
last_message_received msg_from_sensor[SENSOR_NODES];
```

This variable is used just by the server.

It is an array with length equal to the number of sensor nodes of the network. Inside each cell of the array (so for each sensor node), the server stores the `msg_id` of the last message received, the id of the gateway which forwarded the message, and a boolean called `retransmitted` which is used to re-transmit an ack just once, in case multiple requests of ack re-transmission come to the server.

When the server receives a message, checks the `msg_id` of the new message and the one saved in the array at the index corresponding to the same sensor node. If the ids are not the same, it sets again the values in the array at the corresponding cell. If the ids are the same, it checks who is the gateway which forwarded the two message (the one that has been just received and the one stored in the array): if they are the same it means that it is a message being re-transmitted, and if the boolean is set to FALSE the ack is sent back to the sensor; otherwise, if the gateways are not the same, the new message is considered a duplicate and it is discarded.

## Code - Interfaces

**Timer0.** Timer 0 is called by the `generate_send` function to call the `actual_send` function in order to effectively send the new packet to the radio interface. The timer emulates the **transmission delays**, defined for each node in the `time_delays` array.

**Timer1.** Timer1 fires out only for those nodes which were intended to send **data packets**, namely the Sensor nodes from 1 to 5. A further distinction is made between nodes which are meant to send packets to both gateways and nodes which are meant to send packets to only one gateway, according to the topology.

In case the Sensor node has to send packets to both gateways we prepare the message that will be sent to gateway 1 (which is the node with TOS_NODE_ID equal to 6). Then, in order to avoid the possibility of concurrent access to the radio resources for the tranmission, we decided to use a boolean, called `transmitted_to_second_gateway`, which when we do the first transmission is set to FALSE. When the transmission to gateway 1 will be completed and the `sendDone` event will be triggered, we check the TOS_NODE_ID: if the node is 2 or 4, we check the value of the boolean `transmitted_to_second_gateway`: if it is set to FALSE means that we still need to prepare the packet and send the same message to gateway 2, so with destination address equal to 7; otherwise, if it is TRUE, we do nothing, since it means that we sent the two packets already.

In case the Sensor node has to send packets to only one gateway, we set the destination of the message and the address of the transmission to 6 or 7, depending on which is the gateway that will receive the message (gateway 1 or gateway 2).

**Timer2.** Once the `generate_send` function has been called to send the packet to the radio interface, the timer Timer2 is called with a timeout of 1 second to re-transmit the lost packets.

Timer2 fires out only for those nodes which sent data packets, namely the Sensor nodes from 1 to 5. How  transmission and retransmission work is explained in the **SenseNet.h** section of the report, in particular where the `last_message_transmitted` structure is presented.

**AMControl**. When the startDone event is triggered, if the radio start was successful, it is started a periodic timer (Timer1), whose period depends on the TOS_NODE_ID. That is the purpose of Timer1 is explained above.

**AMSend**. The `sendDone` event is triggered when the `AMSend.send` function terminates. As already explained, we decided to use this event for those sensor nodes that need to send the message to both gateway 1 and gateway 2. In particular, when the `sendDone` event returns successfully, we check the `transmitted_to_second_gateway` boolean: if it is FALSE, it means that we have just transmitted to node 6 (gateway 1), and we still need to send to node 7 (gateway 2); otherwise, it means that we sent to the second gateway. Obviously, this piece of code is executed just for those nodes that communicate with both gateways, which are node 2 and node 4.

**Receive**. The behavior of the Receive.receive event depends on the type of the node:

- **sensor node**: if the received message is of type 1 and the ID of the received message is equal to `msg_tx.msg_id`, it means that it is receiving the ack it was waiting for. To be sure we included these conditions in the if statement, however due to the periods we set for Timer1 and for the length of the 1000ms window for receiving the ack we are sure that a sensor node will always receive an ack, and never a data message, and also that when it receives the ack, it refers to the message saved in `msg_tx`. For example, if we consider node 1 which has a period of 2000ms we see that when it transmits a message we have space for 2 windows for the reception of the ack. This means that it also includes the window for the re-transmitted message. This shows that the ids of `msg_tx` and the ack will always be the same.
- **gateway node**: if the received message is of type 0, the message is forwarded to the server; if the message is of type 1, it is forwarded to the corresponding sensor node. In both cases, no field of the received message is modified.
- **server**: first of all, the `send_data_to_node_red` function is called, which will be explained in the next section. Then the server uses the `msg_from_sensor` array in order to check the ids of the message stored in the array and the one that has been received: if their ids are not the same, it means that the server is receiving a new message and stores it inside the corresponding cell of the array, so the ack can be created and sent; otherwise, if the ids are the same, the server checks whether they come from the same gateway (in order to discard duplicates) and checks if the `retransmitted` field is equal to FALSE, in order not to re-transmit the ack more than once. So, if conditions hold, the ack is transmitted again.

The assumption we made is that if the server receives a message that has been re-transmitted by a sensor node, it can receive it just from the same gateway that forwarded it to the server the first time. For example, let's suppose that node 2 transmits a new message to both gateway 1 and gateway 2. The server first receives the message forwarded by gateway 1, saves it into its array, creates the ack, sends it back to the sensor node, and then discards the duplicate coming from gateway 2. Suppose now that the ack is lost and node 2 re-transmits again the same message. In this case, the server will accept the message just from gateway 1 and will discard the re-transmitted message that will be forwarded by gateway 2.

## Code - Functions

**generate_send**. The function `generate_send` takes as input parameters the address to which the packet has to be sent and a pointer to the packet to be sent. It returns FALSE in case the function tries to access Timer0 while it's running, otherwise it calls Timer0 with a given delay for each node set in the array time_delays  and it assigns the pointer to the packet to a new `queued_packet` pointer and the input address to a new `queue_addr` variable, then it returns TRUE.

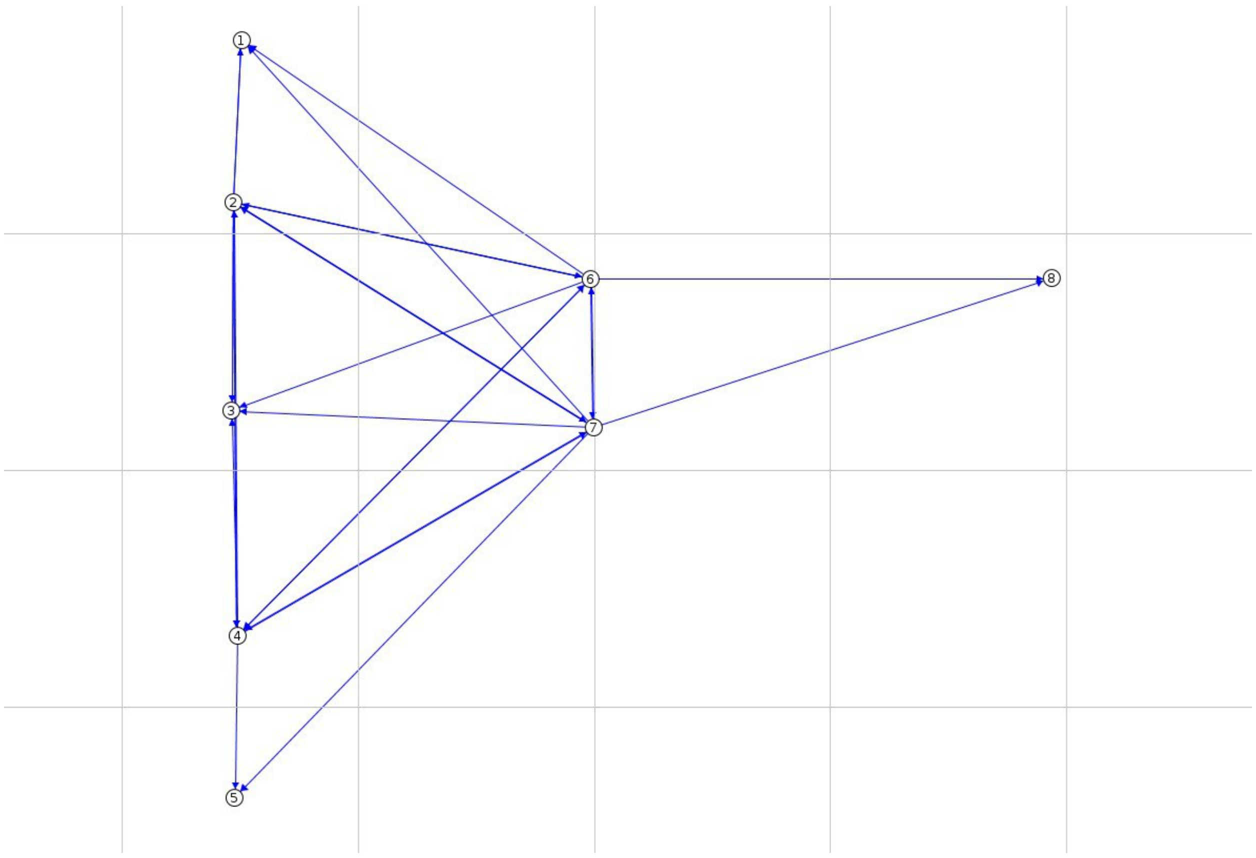
**actual_send**. The function `actual_send` takes as input parameter the queued address and the pointer to the queued packet. If locked variable has been put to TRUE (another instance is using the radio interface) it returns FALSE, otherwise it accesses the radio interface by calling `AMSend.send` function with input parameters the address of the node to which we are sending the message and a pointer to the message and it sets the locked variable to TRUE.

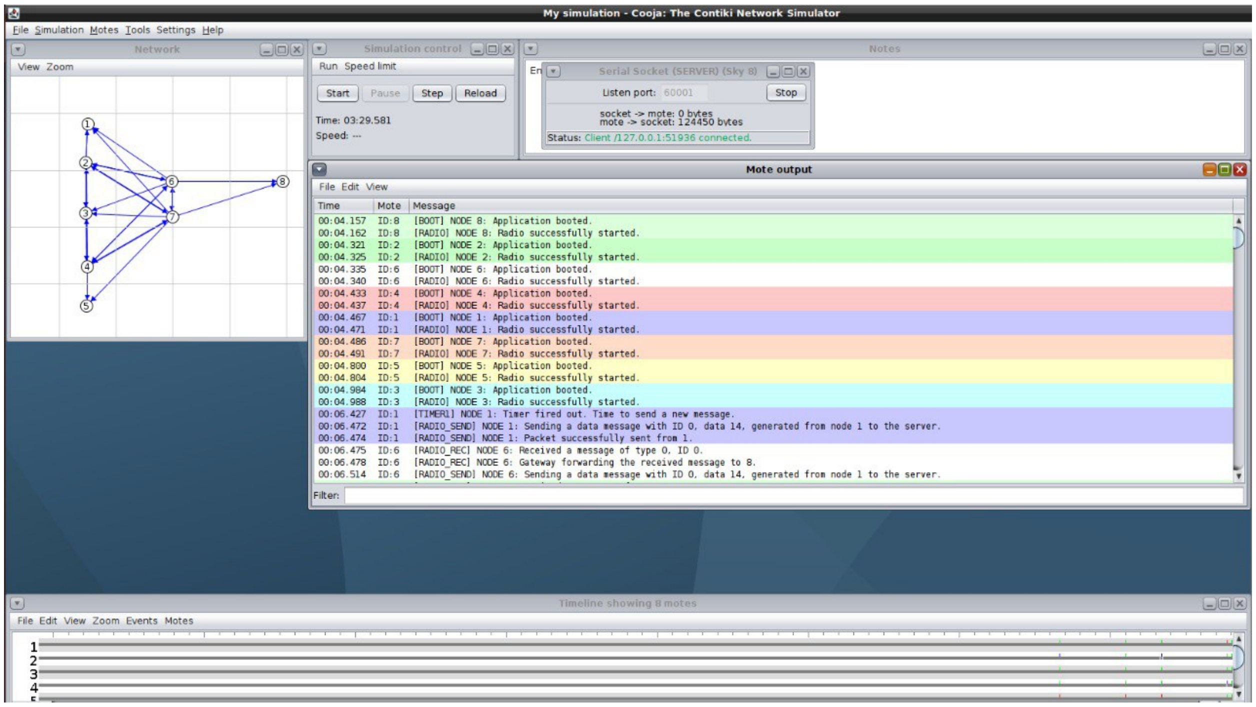
**send_data_to_node_red**. The function `send_data_to_node_red` creates a string buffer and prints in it the desired field of Thingspeak in which it wants the data value printed and the corrrespondent data value (for example for field1 with value 5 to be printed it prints *fieldone:5* ). A decision has been taken to put in the three different fields of Thingspeak the data coming from the sensors 1, 3 and 5. The buffer string is then printed in the output terminal using the `printf` function from the SerialPrintfC component; the output will then be printed in Cooja and sent to node-red through a TCP connection (see Connection to Thingspeak through MQTT Server in node-red).

## Cooja

**Connection with Thingspeak through MQTT Server in node-red**. Data values coming from the sensor nodes 1,3,5 of the topology are visualized in three different charts of a Thingspeak public channel. The target objective is achieved by implementing a node-red flow to parse the packets coming from a TCP connection opened on Cooja and by using the incoming packets' fields to craft messages in the correct format to be sent to an MQTT Server to be displayed in Thingspeak charts. Outputs in the Cooja simulation are printed in the terminal and a TCP connection to port 60001 is opened so to send the printed outputs also to node-red, which listens to the same port and receives all the printed outputs.

We show now some screenshots made during the simulation:




My simulation - Cooja: The Contiki Network Simulator
File Simulation Motes Tools Settings Help
Network
View Zoom
Simulation control
Run Speed limit
Start Pause Step Reload
Notes
Serial Socket (SERVER) (Sky 8)
Listen port: 60001
Stop
Timeline showing 8 motes
File Edit View Zoom Events Motes

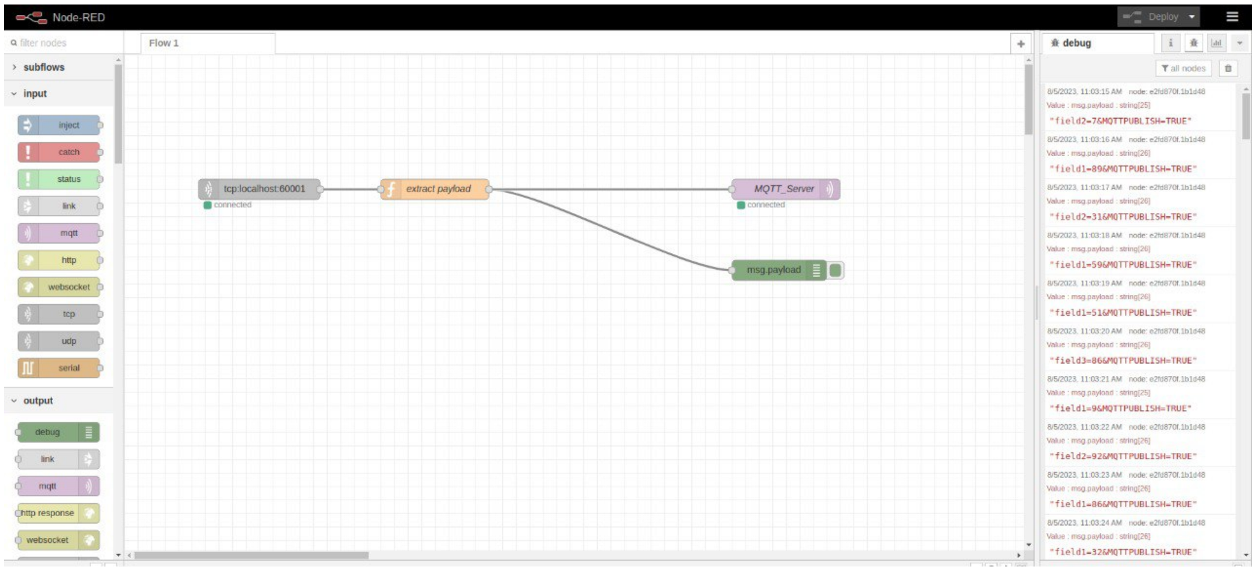
```
Time: 03:29.581
Speed: ---

socket -> mote: 0 bytes
mote -> socket: 124450 bytes
Status: Client /127.0.0.1:51936 connected.
```

```
Time       Mote  Message
00:04.157  ID:8  [BOOT] NODE 8: Application booted.
00:04.162  ID:8  [RADIO] NODE 8: Radio successfully started.
00:04.321  ID:2  [BOOT] NODE 2: Application booted.
00:04.325  ID:2  [RADIO] NODE 2: Radio successfully started.
00:04.335  ID:6  [BOOT] NODE 6: Application booted.
00:04.340  ID:6  [RADIO] NODE 6: Radio successfully started.
00:04.433  ID:4  [BOOT] NODE 4: Application booted.
00:04.437  ID:4  [RADIO] NODE 4: Radio successfully started.
00:04.467  ID:1  [BOOT] NODE 1: Application booted.
00:04.471  ID:1  [RADIO] NODE 1: Radio successfully started.
00:04.486  ID:7  [BOOT] NODE 7: Application booted.
00:04.491  ID:7  [RADIO] NODE 7: Radio successfully started.
00:04.800  ID:5  [BOOT] NODE 5: Application booted.
00:04.804  ID:5  [RADIO] NODE 5: Radio successfully started.
00:04.984  ID:3  [BOOT] NODE 3: Application booted.
00:04.988  ID:3  [RADIO] NODE 3: Radio successfully started.
00:06.427  ID:1  [TIMER1] NODE 1: Timer fired out. Time to send a new message.
00:06.472  ID:1  [RADIO_SEND] NODE 1: Sending a data message with ID 0, data 14, generated from node 1 to the server.
00:06.474  ID:1  [RADIO_SEND] NODE 1: Packet successfully sent from 1.
00:06.475  ID:6  [RADIO_REC] NODE 6: Received a message of type 0, ID 0.
00:06.478  ID:6  [RADIO_REC] NODE 6: Gateway forwarding the received message to 8.
00:06.514  ID:6  [RADIO_SEND] NODE 6: Sending a data message with ID 0, data 14, generated from node 1 to the server.
```

Finally, in the delivered folder, we also included the file Cooja_log_file.txt inside the resources folder containing the console output of Cooja, so it shows all the debug messages.

## Node-Red

The flow is:



```
8/5/2023, 11:03:15 AM  node: e2fd870f.1b1d48
Value : msg.payload : string[25]
"field2=7&MQTTPUBLISH=TRUE"
8/5/2023, 11:03:16 AM  node: e2fd870f.1b1d48
Value : msg.payload : string[26]
"field1=89&MQTTPUBLISH=TRUE"
8/5/2023, 11:03:17 AM  node: e2fd870f.1b1d48
Value : msg.payload : string[26]
"field2=31&MQTTPUBLISH=TRUE"
8/5/2023, 11:03:18 AM  node: e2fd870f.1b1d48
Value : msg.payload : string[26]
"field1=59&MQTTPUBLISH=TRUE"
8/5/2023, 11:03:19 AM  node: e2fd870f.1b1d48
Value : msg.payload : string[26]
"field1=51&MQTTPUBLISH=TRUE"
8/5/2023, 11:03:20 AM  node: e2fd870f.1b1d48
Value : msg.payload : string[26]
"field3=86&MQTTPUBLISH=TRUE"
8/5/2023, 11:03:21 AM  node: e2fd870f.1b1d48
Value : msg.payload : string[25]
"field1=9&MQTTPUBLISH=TRUE"
8/5/2023, 11:03:22 AM  node: e2fd870f.1b1d48
Value : msg.payload : string[26]
"field2=92&MQTTPUBLISH=TRUE"
8/5/2023, 11:03:23 AM  node: e2fd870f.1b1d48
Value : msg.payload : string[26]
"field1=86&MQTTPUBLISH=TRUE"
8/5/2023, 11:03:24 AM  node: e2fd870f.1b1d48
Value : msg.payload : string[26]
"field1=32&MQTTPUBLISH=TRUE"
```

**Node-red nodes and functions.**

**tcp_localhost:60001**. This node listens to the TCP port 60001 of the localhost and receives the strings from Cooja output to forward them to the function node **extract payload**.

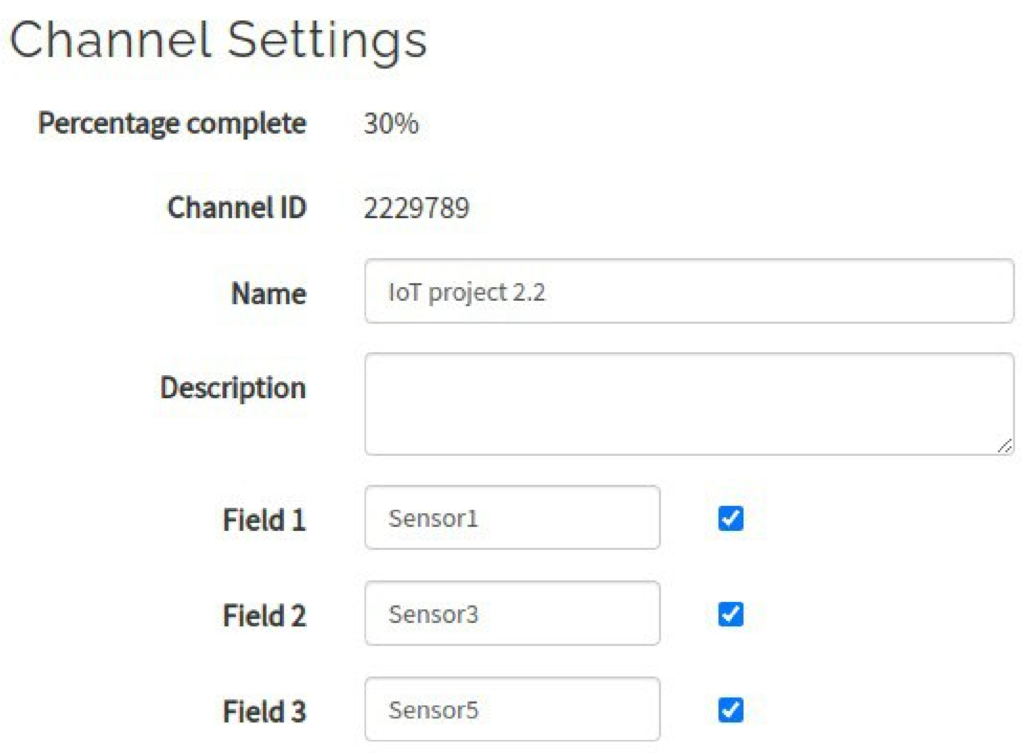
**Extract payload**. This node uses a script in javascript to edit the `msg.payload` field only in the messages whose destination is Thingspeak. In fact, since we used the `printf` function also to print debug messages, they are sent through the TCP port to node-red, too. However, in this node we consider just those messages that are intended to be sent to Thingspeak, so all the debug messages are ignored.

The function in this node first searches for a match in the incoming string to one of the fields of the Thingspeak chart (eg. the incoming string contains the string “fieldone” and it matches to field1 of the Thingspeak charts); if the incoming string checks positive for one of the three fields then a number (the data value) is parsed from the string and both the value and the correspondent field are assigned to a modified payload which is then forwarded to both the MQTT Server and the debug node. Only the strings with the data intended to be shown in Thingspeak are forwarded to the MQTT Server.

**MQTT Server**. This node sends the message to an MQTT Server configured in Thingspeak with address mqtt3.thingspeak.com, port 1883 and correspondent Client Id, Username and Password. The public channel is accessible at address https://thingspeak.com/channels/2229789 and displays the three charts correspondent to the data sent by Sensor nodes 1, 3, 5 of the topology.

## Thingspeak

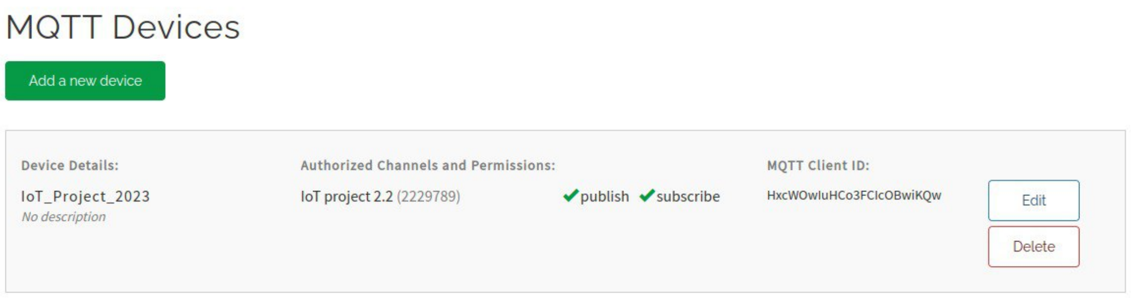
Thingspeak channel is configured by selecting the new channel in the my channels field, then name, description and fields are set in the new channel and the settings are saved.



Channel Settings

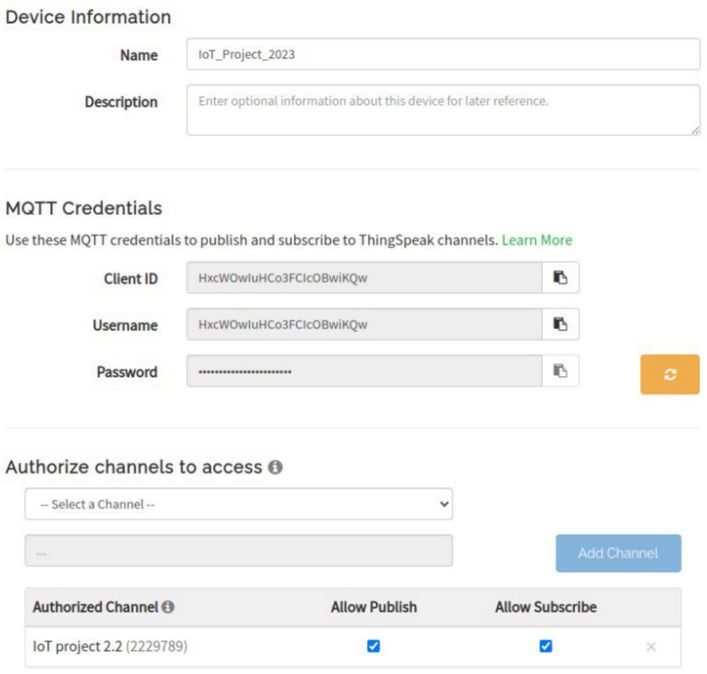
| | | |
|---|---|---|
| Percentage complete | 30% | |
| Channel ID | 2229789 | |
| Name | IoT project 2.2 | |
| Description | | |
| Field 1 | Sensor1 | ✓ |
| Field 2 | Sensor3 | ✓ |
| Field 3 | Sensor5 | ✓ |

A new MQTT device has to be configured to be used for forwarding messages from node-red; this is achieved through selecting the devices field in Thingspeak, selecting the MQTT option and selecting add new device. The new device is configured by assigning it a name which in our case is **IoT_Project_2023**, optionally a description and by selecting the channels for which the new device is authorized to write and read, the settings are then saved.



MQTT Devices

| Device Details: | Authorized Channels and Permissions: | | MQTT Client ID: |
|---|---|---|---|
| IoT_Project_2023<br>*No description* | IoT project 2.2 (2229789) | ✓publish ✓subscribe | HxcWOwluHCo3FClcOBwiKQw |


Device Information

| | |
|---|---|
| Name | IoT_Project_2023 |
| Description | Enter optional information about this device for later reference. |

MQTT Credentials

Use these MQTT credentials to publish and subscribe to ThingSpeak channels. Learn More

| | |
|---|---|
| Client ID | HxcWOwIuHCo3FCIcOBwiKQw |
| Username | HxcWOwIuHCo3FCIcOBwiKQw |
| Password | •••••••••••••••••••••••• |

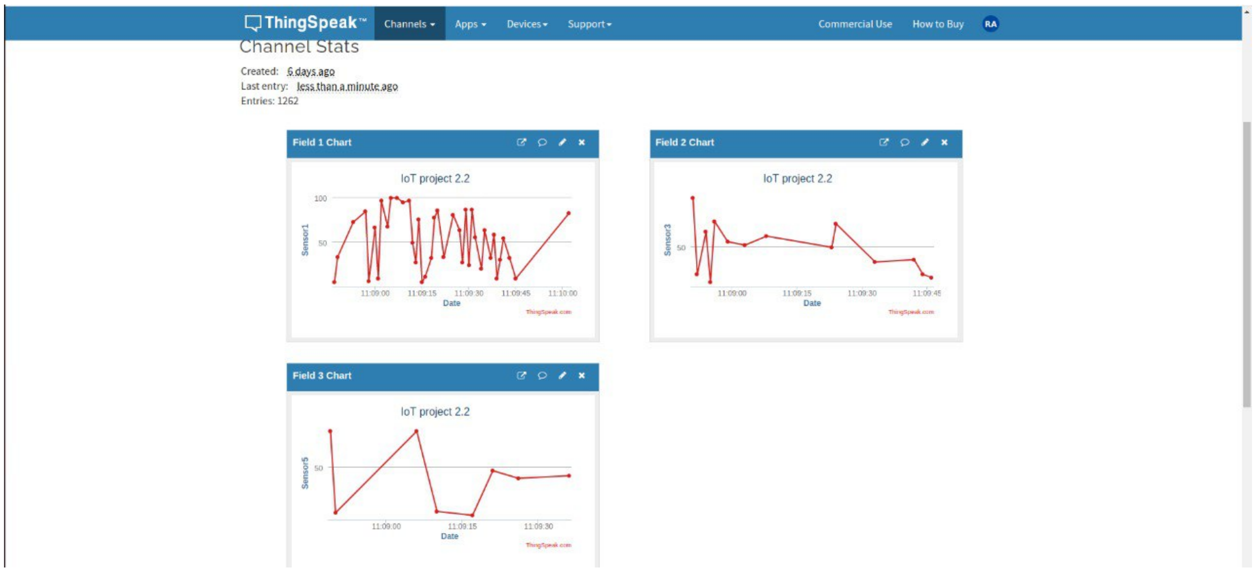
Authorize channels to access

-- Select a Channel --

Add Channel

| Authorized Channel | Allow Publish | Allow Subscribe |
|---|---|---|
| IoT project 2.2 (2229789) | ✓ | ✓ |

Once Cooja simulation is started and node-red starts transmitting to the MQTT Server, by going to the public view of our channel in Thingspeak it's possible to observe how the data values from Sensors 1,3,5 in the chart change over time, at a rate varying from 10% to 100% which is set in Cooja. It's possible to notice that the rate at which the values in the charts are updated is different from that of the Cooja simulation; this is due to the fact that Thingspeak free license imposes a rate limit of 1 message every 15 seconds.



Channel Stats

Created: 6 days ago

Last entry: less than a minute ago

Entries: 1262