IOT Final Project

Trace 2

(1) Name: Roberto Macaccaro - Person Code: 10615972
(2) Name: Roberto Molgora - Person Code: 10628824

Index:

1. Introduction

The aim of the project was to create a LoraWan like network in TinyOs, network nodes should periodically generate messages with random data and send them (through the network server) to Node-Red, whom have to extract the relevant data from them and send them via MQTT to a ThingSpeak channel. The ThingSpeak channel must finally plot the random values on 5 charts (one for each node of the network).

Our network has also an ack confirmation message functionality: when the server node receives a message, it must send an ack to the sender node, if the sender node doesn't receive the ack within a 1 second window, it sends again the message after a random amount of time (LoraWan-like). In addition, server node must discard duplicates.

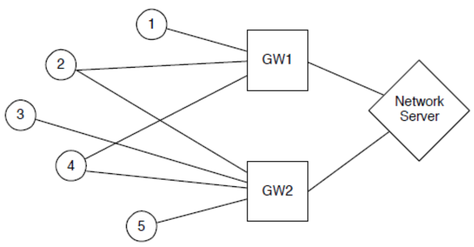
To connect TinyOs and Node-Red it is necessary to build and run the project using cooja, and send messages via TCP to Node-Red.

2. *TinyOs*

TinyOs is able to simulate a network and its functions, it works thanks to interconnections between some files. Those files are:

2.1. *Topology.txt*

This file contains the network topology (shown in figure).



As we can see the network is composed by 5 sensor nodes (nodes 1,2,3,4,5 in our topology file), whom have to send messages to the server node (node 8 in our topology file); passing through the two gateways (nodes 6 and 7 in our topology).
N.B. all connections are bidirectional.

2.2. *RadioRoute.h*

This file specifies which are the fields of the messages that go through the network. Fields are:

- Type: just specifies the type of the message, if it is a data message (1) or an ack message (2).
- Sender: this field specifies which is the original sender node (so between 1 and 5).
- Gateway: this field specifies which is the gateway from which the message arrives to the server node, useful for knowing where to send the ack back.
- Destination: specifies the final destination of the ack message (it's between 1 and 5).
- Value: contains the data, it must be a random generated data.
- ID: contains the message id, the message id is simply a generated number that is incremented by 1 every time a new message is sent.

2.3. *RadioRouteAppC.nc*

This module creates the interfaces and wires them to the RadioRouteC.nc file.
We can notice 3 timers, 2 used to send messages and one (ACK_timer) used to count how much time is passed from the last sending.
We also have a RandomC used to create the random value.
We got also SerialPrintfC and SerialStartC necessary to use printfs in cooja.

2.4. *RadioRouteC.nc*

In the first part of the file, we declared the libraries and the interfaces used.
Then we declared the variables we use in our application, and successively we start to specify functions and event trigger functions.
Practically in our application the program function in this way:

1) The Boot.booted event starts the radio components
2) The AMControl.startDone event start the Timer1 in the nodes from 1 to 5, i.e. the end nodes of our LoraWAN network, and do nothing in other three nodes
3) Timer1 creates and sets up the data messages, with a random value from 1 to 100, the id, that is the counter of the messages sent from the sending node, the type equal to 1 (data message in our application), and its id as sender.

Before sending, it stores locally the data message for retransmission purposes (case in which the ACK doesn't arrive).
After this it call the generate_send function with destination AM_BROADCAST_ADDR, in this way end nodes send to each connected nodes, so only to the two gateways or only one if the end node is connected only to one of these.

4) The generate_send function call the actual_send function after a small random delay, used to avoid more collisions that can be caused sending all in the same moment
5) After the sent, done by the actual_send, the event function AMSend.sendDone initializes the ACK_timer of 1 second, then after these there are 2 options:
   1) The ACK message for some reason doesn't come back to the end node within the ACK timeout time, so the end node retransmits it, and increment the counter of the retransmissions. It can happen maximum 3 times, after these the message will be discarded.
   2) The message arrives to the gateway/s (6,7), which after setting in the message him/they as gateway, forward/s it to the server node (8).
      The server node received the data messages store it in an array (until the next one from the same sender node arrives), and then respond with an ACK message to the gateway, which will forward the message to the interested end node.
      At this point when the end node receives the ACK message it restarts the Timer1 and restart the loop.

Event functions:

- **Timer0.fired** -> fired after a delay, it calls actual send of the messages
- **Boot.booted** -> it starts the radio components
- **AMControl.startDone** -> if the radio is started well, it starts the timer1 (only in nodes from 1 to 5, i.e. the end nodes) with a single timeout of 2 seconds
- **AMControl.stopDone** -> it is triggered if the radio component stops to work for some reasons
- **Timer1.fired** -> when fired it creates and sends a data message with a random number from 1 to 100 as value
- **ACK_timer.fired** -> when fired it re-sends the message because the timeout for having an ACK for it is expired
- **Receive.receive** -> receive messages and elaborate them to extract information
- **AMSend.sendDone** -> triggered when it is sure that the send is done, in it we starts also the ACK_timer with a timeout of 1 second

Other functions:

- **actual_send** -> function that performs the send and lock the resource until it receives the event sent
- **generate_send** -> function that prepare the message to be sent in queue and after triggers the timer zero (with a delay), which will call the actual send of the message

2.5. *RunSimulationScript.py*

This file is the python file that contains the script to run all the project with the TOSSIM debugger, it uses the network topology file and the noise file to initiate all the nodes and their arcs. It initiates all the debug channels.
We used those during the development, in order to debug; in fact there are a lot of dbg calls in the RadioRouteC.nc code, they are commented because we had then to use Cooja and no more TOSSIM, in order to connect our nodes to Node-red.

2.6. *Cooja simulation*

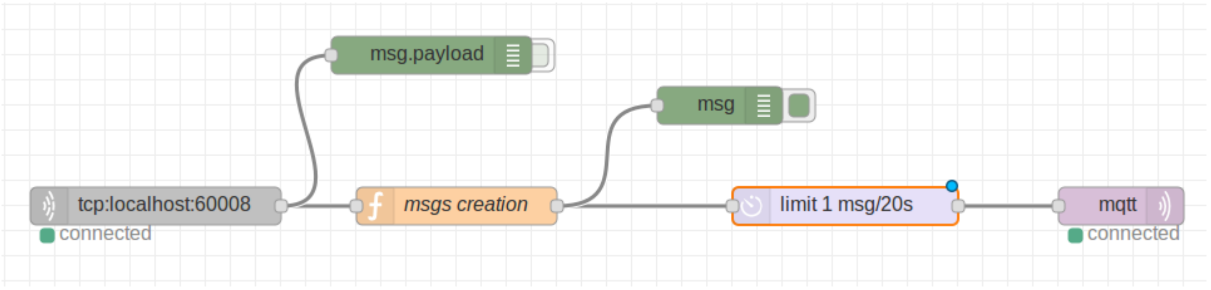
To run the project, it's necessary to compile it and open it in the Cooja runner simulator (Contiki). Before running it, we should ensure that the output node (8) is able to send his data through TCP, by opening a port on that server node.
Then we can start the simulation and see all the packets that flows through the network.

2.7. *Implementation choices*

Since we decided of sending data messages to broadcast, in the Cooja simulation also the nodes which are not connected looking at the topology receives the messages in case they are in range of the radio.
We threat this problem moving them in a way that they are more distanced possible, and moreover we add a control statement, in which if we receive a data message, but we are not gateway or server node, we do nothing.

3. *Node-Red*

The main functionalities of Node-Red (in our project) were to "parse" the received message obtained via TCP, create a new MQTT message and send it via MQTT.



It also limits the number of messages/s that ThingSpeak can receive, with a delay block, since a user with a free license on ThingSpeak can update a channel every 15 seconds.

3.1. *TCP block*

It receives messages sent by the node 8 from the Cooja simulation.

3.2. *Function block*

"msgs creation" take as input the TCP message and generate a new message to be sent via MQTT. In order to do so, it distinguish between debug messages (starts with dbg- in our implementation) and the ones we need to retransmit to ThingSpeak, from which in particular it extracts data about the sender node and the value of the message, and using them (and ThingSpeak credentials in order to transmit them successfully) it generates the new MQTT message, direct to the mqtt channel identified by the CHANNEL ID of the created channel in ThingSpeak.

3.3. *Delay block*

Used to limit the number of messages per second that ThingSpeak can receive (since in the free version we can make only 4 updates per minute).

3.4. *MQTT block*

Set in order to send data to out ThingSpeak channel.

4. *ThingSpeak*

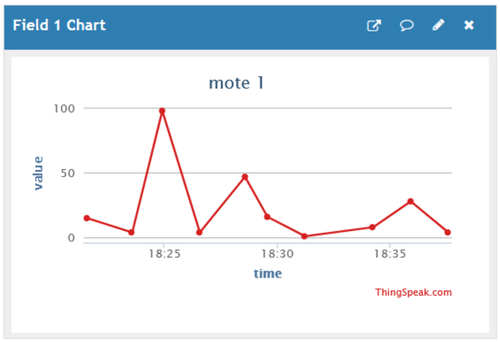
ThingSpeak is an online portal used to connect some kind of peripherals and read data. In our case we connected via MQTT and read data from Node-Red.

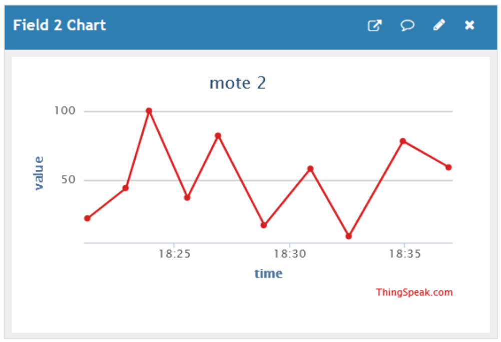
4.1. *Channel creation and settings*

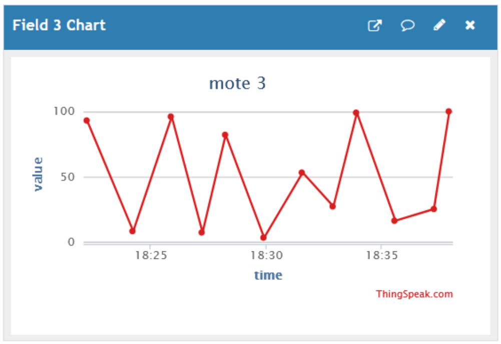
We created a channel with 5 fields (charts), in which each one contains the temporized value sent by each sensor node, so in the end we can see a diagram.

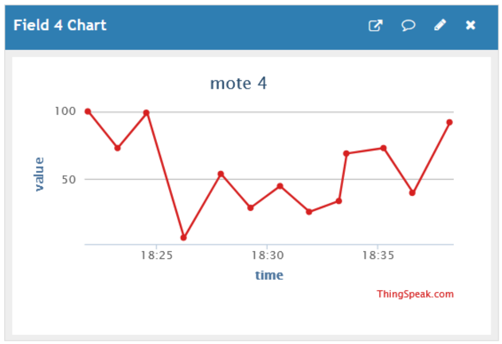
4.2. *Working flow*

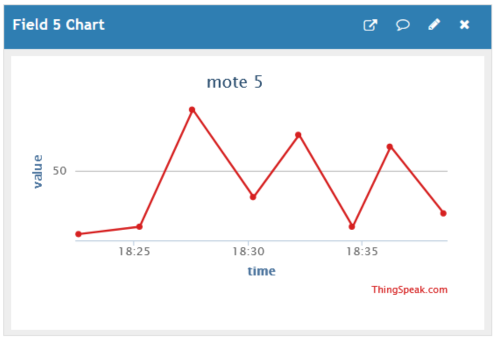
ThingSpeak listens on an MQTT channel for incoming messages.
The channel can have up to 8 fields, we use only 5 (one for each sensor node), ThingSpeak reads the messages and assigns the new values of each field to a corresponding new value in the corresponding chart.











5. *Observations*

5.1. *Run instructions*

In order to successfully run the project it's necessary that, while we are running the TinyOs project using Cooja with Contiki, we enable the node 8 (server node) to communicate via a port specified by us, and then we have to specify that port in the TCP module of NodeRed.

In order to grant to NodeRed the access to ThingSpeak and to send it values, it's necessary to specify the account and channel's credentials int the MQTT module.

It is possible to see the live plots using this link to the ThingSpeak channel:
https://thingspeak.com/channels/2205556

5.2. Delay in communication between Node-Red and ThingSpeak

Due to the fact that we add a delay in the transmission from Node-Red to ThingSpeak we can notice that in the delay block some message will be accumulated, in fact messages arrives enough fast from the LoraWan network server, but they need to be sent to ThingSpeak with a rate slower than a message every 15 seconds (because it updates every 15 seconds in the free version).