iot_project

Marco D'Amico, Paolo Moser

June 2023

IoT 2023 PROJECT

(1) Name: Marco D'Amico Person Code: 10661846
(2) Name: Paolo Moser Person Code: 10884678

Contents

1 TinyOS

The project required to emulate in TinyOS a lightweight publish/subscribe protocol of communication, with two actors:

- PAN Coordinator, a self elected node that acts as the broker of the protocol of communication;
- Motes, 8 different mote that can connect to the broker, subscribe to some topics (in our case they were only 3) and publish (one topic per mote, with random payload).

As PAN Coordinator we decided to give the TOS_NODE_ID of 1, while motes have TOS_NODE_ID from 2 to 9. This decision was hard-coded in our application but it can easily be modified at need.

1.1 Application

The application is divided into two parts: one about the mote and one about the PAN Coordinator (each of them explain in details in successive subsections). The reason of this choice is that the behaviour of the two components is deeply different (even with different requirements in term of memory usage and radio usage, i.e., in terms of energy consumption in case of real world scenarios). Still there are some common portion to be shared between this two type of components. In fact we have a common LwPubSubMsg.h file, an header files that contains shared variables and definitions:

- *TIME_TO_LOSS* parameters specifies the amount of time after which a message is considered to be lost during transmission;
- *pub_sub_msg_t* is the struct which defines the structure of our messages. We decided to use only one struct for all the different kind of message for sake of simplicity. So we have 4 fields:
  - type, the type of the message;
  - sender, the sender of the message;
  - topic, the topic of the message (used for subscriptions and publications);
  - payload, the payload of the message.
- proper enums for sake of clairty, e.g., *msg_type*, *topics* ecc...

Per each message it is specified the sender, a type and accordingly to it the remaining fields might be used or ignored:

- the CONNECT message (type 0);
- the SUBSCRIBE message (type 1) specifies the topic to subscribe to;
- the PUBLISH message (type 2) specifies the topic and the payload of the publication.

1.1.1 Mote

The portion of the application that refer to Mote needs to handle connection to the broker, subscription to the topics and publications.

The first thing each mote does after boot is to start the radio (required for communication) and then tries to connect to the PAN coordinator: once the radio was turned on correctly it sends a CONNECT message to the broker, waiting for a CONNACK. This behaviour is obtained by using the Acknowledgment interface of TinyOS and starting a timer (which fires after *TIME_TO_LOSS*) which has the duty of signaling that something went wrong (either the message was never received by the PAN coordinator or the CONNACK was lost), therefore a new CONNECT message has to be sent. If connection is successful the timer is stopped.

After connection each mote starts a periodic timer that is responsible for new PUBLISH message (the period is chosen at random, but eventually could be hard-coded with some policy). Also it starts handling subscriptions (here topics' subscriptions were hard-coded depending on the *TOS_NODE_ID* of the mote, but it could have been randomized or a different policy could have been put in place), sending one message per topic subscription, in order to keep the Pan Coordinator logic as much simple as possible. The same behaviour as for CONNACK message is put in place for the SUB and SUBACK message.

1.1.2 PAN Coordinator

The PAN Coordinator acts as the broker of the publish/subscribe protocol, so it has the task of receiving and managing all messages sent over the radio by the motes.

PAN Coordinator has one more header file *PANCoordinator.h* in which are defined two structures:

- *node_info*: represents the information that the broker needs about a mote. It is made up of 2 boolean variables:
  - *connected*, TRUE if a mote is connected to the broker, FALSE otherwise;
  - *topics*, an array of dimension 3, where each position is TRUE if the mote is subscribed to corresponding topic, FALSE otherwise;
- *queue_msg_t*: used to manage the queue of publication messages to be processed, made by the same fields of the struct for the messages with an additional field, destination.

The PAN Coordinator manages information about nodes via an array of type *node_info* and of length 8 (one per each possible connected client); on boot, every field is initialized to FALSE. We will call this array *nodes*.

After booting the Pan Coordinator need to handle the following events:

- **CONNECTION**, when it receives a CONNECT message, it sets to TRUE the value of *nodes[i].connected*, where i is the sender of the CONNECT message.

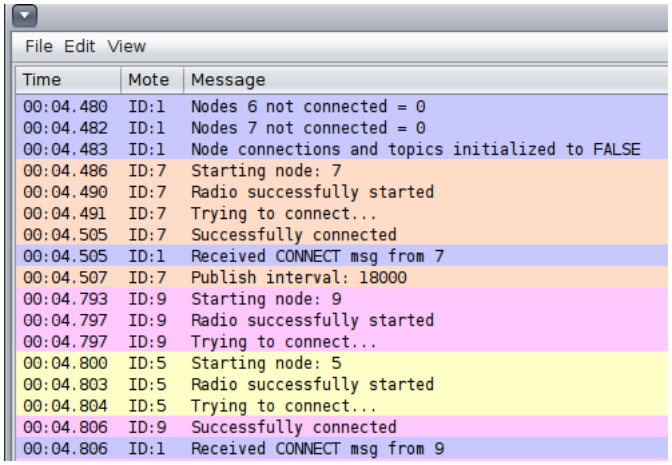


Figure 1: connection of motes

- **SUBSCRIBE**, when it receives a SUBSCRIBE message, it sets to TRUE the value of *nodes[i].[topics]* corresponding to the topic to which the mote i wants to subscribe.

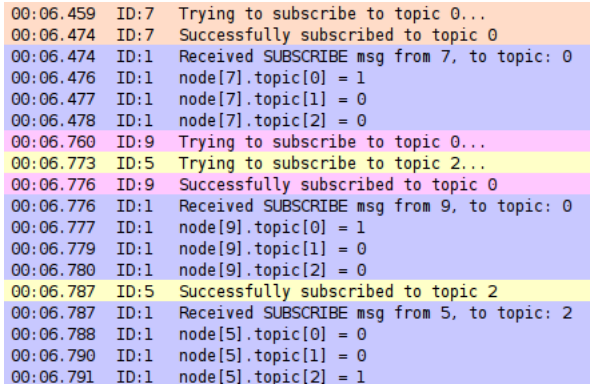


Figure 2: subscription of mote to a topic

- **PUBLISH**, when it receives a PUBLISH message, it checks to which nodes to forward the message, by scanning the array *nodes*. If a mote is subscribed to that topic, the payload of the message to be sent is created and enqueued. The queue is required in order to handle multiple concurrent publications that reach the PAN Coordinator while it is still finishing

forwarding messages. Then if the radio is free (variable *locked* is FALSE), it starts sending messages.

```
00:44.376  ID:2   Publishing on topic: 2, payload: 69, with QoS=0
00:44.390  ID:1   Received PUBLISH msgfrom 2topic:2payload:69
00:44.394  ID:1   Enqueued message, with dest:2, message_payload: type: 2, sender: 2, topic: 2, payload: 69
00:44.399  ID:1   Enqueued message, with dest:3, message_payload: type: 2, sender: 2, topic: 2, payload: 69
00:44.404  ID:1   Enqueued message, with dest:5, message_payload: type: 2, sender: 2, topic: 2, payload: 69
00:44.409  ID:1   Enqueued message, with dest:6, message_payload: type: 2, sender: 2, topic: 2, payload: 69
00:44.414  ID:1   Enqueued message, with dest:8, message_payload: type: 2, sender: 2, topic: 2, payload: 69
00:44.419  ID:1   Enqueued message, with dest:9, message_payload: type: 2, sender: 2, topic: 2, payload: 69
00:44.423  ID:1   Sending message, with dest:2, message_payload: type=2, sender=2, topic=2, payload=69
00:44.426  ID:1   Locking the radio, sending msg...
00:44.427  ID:1   Send done, unlocking the radio
00:44.431  ID:2   Received message, type: 2, sender: 2, topic: 2, payload: 69
00:44.432  ID:1   Sending message, with dest:3, message_payload: type=2, sender=2, topic=2, payload=69
00:44.434  ID:1   Locking the radio, sending msg...
00:44.437  ID:1   Send done, unlocking the radio
00:44.441  ID:3   Received message, type: 2, sender: 2, topic: 2, payload: 69
00:44.442  ID:1   Sending message, with dest:5, message_payload: type=2, sender=2, topic=2, payload=69
00:44.444  ID:1   Locking the radio, sending msg...
00:44.452  ID:1   Send done, unlocking the radio
00:44.455  ID:5   Received message, type: 2, sender: 2, topic: 2, payload: 69
00:44.456  ID:1   Sending message, with dest:6, message_payload: type=2, sender=2, topic=2, payload=69
00:44.459  ID:1   Locking the radio, sending msg...
00:44.460  ID:1   Send done, unlocking the radio
00:44.463  ID:6   Received message, type: 2, sender: 2, topic: 2, payload: 69
00:44.465  ID:1   Sending message, with dest:8, message_payload: type=2, sender=2, topic=2, payload=69
00:44.467  ID:1   Locking the radio, sending msg...
00:44.475  ID:1   Send done, unlocking the radio
00:44.478  ID:8   Received message, type: 2, sender: 2, topic: 2, payload: 69
00:44.480  ID:1   Sending message, with dest:9, message_payload: type=2, sender=2, topic=2, payload=69
00:44.482  ID:1   Locking the radio, sending msg...
00:44.489  ID:1   Send done, unlocking the radio
00:44.493  ID:9   Received message, type: 2, sender: 2, topic: 2, payload: 69
```

Figure 3: mote 2 publish a message

1.2 Simulation

The simulation of our application was performed by Cooja (instead of Tossim) where we created a star topology (Figure4), with random positions per each mote, but checking that every mote was able to communicate with the broker.

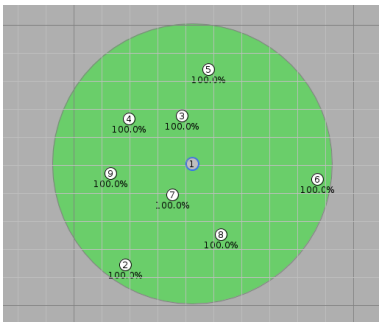


Figure 4: Star topology adopted in Cooja. All positions are randomized but within the range of the Pan Coordinator.

Every mote is modeled as a sky-mote:

- the PAN Coordinator is a sky-mote with a binded executable file derived from the PAN Coordinator TinyOS files

- motes are 8 identical sky-mote devices with executable file derived from the Mote TinyOS files.

Also in order to be compliant with the project's requirement the PAN Coordinator opens up a tcp connection with the Node-RED flow: the mote acts a tcp client, connect to a proper tcp component of Node-RED on the socket (localhost:9000, Figure5). In this way all the serial debug-print of the broker are sent to the Node-RED flow, including the PUBLISH message to be forwarded to Thingspeak.

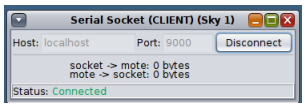
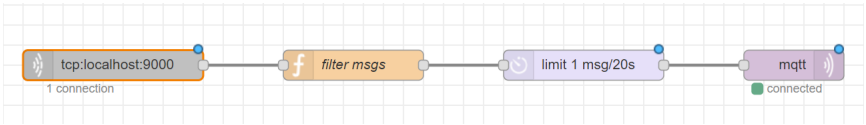


Figure 5: Client connection of PAN Coordinator.

2 Node-RED

In this section we will discuss the Node-RED and Thingspeak implementation for the project.

2.1 Node-RED Flow



Node-RED flow starts with a *TCP_in* node which listens on port 9000, and outputs a stream of String (payloads of received messages). This stream of string is managed by a function node that filter the messages received, keeping only the ones properly marked (PUBLISH messages are the only ones containing the string "PUBLISH"). In fact because of the serial connection between the TinyOS Pan Coordinator and the *TCP_in* node, every debugging print from Cooja is sent to the Node-RED flow which is in charge of filtering it and also periodically transmit data received (obtained with a *delay_node* which set the flow throughput to 1 message every 20 seconds.

When a publish message is received, it is splitted to extract the topic and the payload. At last they are used in order to be compliant with ThingSpeak API of mqtt publication on a single topic (e.g., the topic is concatenated with a fixed string representing the ThingSpeak pattern to publish on a single topic). The output is sent to a *mqtt-out-node* connected to a ThingSpeak channel.

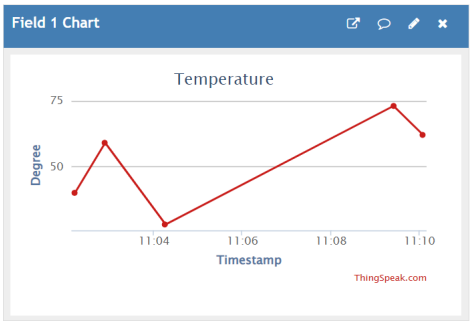
2.2 ThingSpeak

In Thingspeak we created a mqtt-device (ID: CxwDGwoJEzojOC8zHT0XKzk) that would work as receiver, in such a way that the output of the Node-RED flow could be properly received and then published on the associated channel.

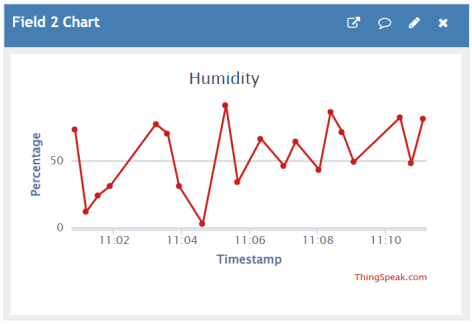
We also created the aforementioned channel, which has three charts, one per topic (TEMPERATURE, HUMIDITY, LUMINOSITY), which are periodically updates by publications.

3 Simulated example

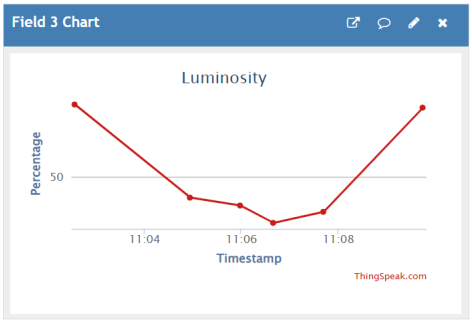
The final simulation last 10 minutes and we set the message interval of 20 seconds (1 message every 20 second on random topic). Log file for all debug print gotten from Cooja is loglistenr.txt; a csv file containing all data publication of the simulation is feed.csv.



(a) Temperature chart



(b) Humidity chart



(c) Luminosity chart