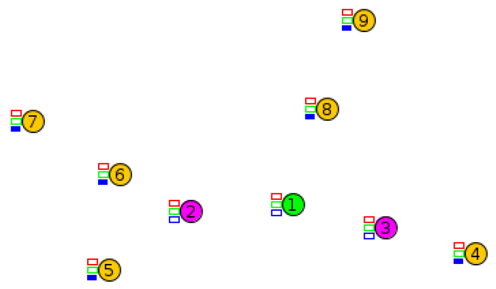
LINGI2146 - Project 1

Florentin Delcourt - 5329 1500
Alexandre Halbardier - 5843 1400
Elias Oumouadene - 8121 1400

https://github.com/eoumouadene/LINGI2146

29 May 2020

1 Introduction



For this project we have implemented a code corresponding to the following (fictive) scenario: We have a building management system with *yellow nodes* that sense the current temperature and can open/close a valve (represented by a green LED), a *green node* that acts as a root and sends data to a server for the computation of a least-squares fit on the data. Then we have *pink nodes* that can compute the data of maximum 5 yellow nodes so that it does not go all the way to the server. All of this solution is computed on Contiki with Z1 motes with Rime modules.

2 Global Approach

As we could only use Rime modules for single-hop reliable unicast and best effort local area broadcast, our sensor network needed both of these protocols to work properly. To do that we used 2 processes on each node to manage Broadcast and Runicast message transmission independently. (And a third one on the root node for the root-server communication.)

Broadcast message are used when destination addresses are unknown to the sender or when the messages need to be forwarded to a sub-part of the tree without caring for the next-hop addresses.

On the other side, Runicast message are used when destination addresses are well known in order to target a specific node. This node is most of the time the parent of the sender node but as we also needed Runicast message to be able to go down in the tree to reach specific nodes without using Broadcast (as it would overflow the network with useless information) we had to implement a routing table on each node. (For now we fixed the size of this routing table but it limits the maximum number of nodes we can have in our network.)

The routes in our routing tables are structured as follows :

- **int TTL :**

  Time-to-Live value used as a timer for a route expiration and reset each time its route is actualised. In our project we used TTL = 3 for our initial value, we decrease it by 1 each minute and if the node receives a message corresponding to this route while its TTL value is greater than 0 it puts it back to 3. This means that our routes can stay alive without actualisation for 3 minutes. (We can choose another TTL initial value depending of the network stability and as long as it is greater than the time between sensors data transmission)

- **int addr_to_find[2] :**

  Address of the node we are looking for.

- **int next_node[2] :**

  Address of the next node we need to send runicast message to in order to reach the destination node.

There is 3 kind of nodes in our network as asked in the instruction :

- *Basic Sensor Node :* Get data, send them for computation and receive instructions when needed.
- *Computation Node :* Do the computation for maximum 5 sensor nodes at the same time or behave like a basic node to transmit messages.
- *Root Node :* Send data to the server to do the computation for the rest of the sensor nodes and act as a root for the tree creation.

Another important aspect of this project was to avoid as much collisions as possible during transmissions. To achieve this we added random waiting time on some keys points of the execution of our network such as the hop-by-hop broadcasting during the tree creation or the first runicast communication used to carry data. Even if this help prevent problems if collisions still happen the nodes are forced to reset themselves on detection. (If there is too much collisions in the network a good way to prevent them would be to add more computation nodes to distribute the communication load among them)

3 Message structure

We have only one message structure in our network but some of its elements are only useful in specific situation. Our messages elements are structured as follows :

- **int msg_type :**

  Define the type of the message, we used 6 different message types :

  0. Node Down : Message sent when a node is down/disconnected from its parent. By sending this message to its neighbors we can find which node had this one as parent and reset them. This helps adapting the rank in case of a node failure.
  1. Discovery : Message sent by broadcast at the initialisation of the network and periodically after, to set nodes ranks, parent address and adapt the tree in case of modification. It helps to check for signal strength between sender of same rank to make sure that the parent chosen by the node is the best. (Also acts as a starting point for the data transmission of the sensor nodes)
  2. Data Up by Runicast : Message sent by runicast and going up the tree (parent after parent) with the sensor value of a node to reach the nearest computation node or the root node. It is also used to manage the routing table of each node.
  3. Action Down by Runicast : Message sent by runicast and going down the tree (using each node's routing table) while carrying the valve opening instruction for a specific node.
  4. Action Up by Runicast for general Broadcast : Message sent by runicast and going up the tree (parent after parent) with the address of a node whose valve needs to be open. Triggered when a type 3 message failed to reach its destination to try to ensure that a node receive its instructions even after moving in the network.
  5. Action Down by general Broadcast : Message sent by broadcast in the whole tree to maximise the chance of finding the correct node after a valve opening message failed. (similar to flooding = worst-case scenario)

- **int sender_rank :** (used for message type 1, 5)

  Give the rank of the message sender, as our network is built as a tree we often need to take into account the rank of the sender to process a message correctly.

- **int origin_addr[2] :** (used for message type 2, 3, 5)

  Give the address of the sensor node which is sending its data for computation.

- **int sender_data_value :** (used for message type 2)

  Give the value of the data generated by the node of origin_addr for computation.

- **char sender_data[64] :**

  Give extra information as a string, was used for debugging.

4 Sensor Network building and adaptation

In our network important value were added on each node to ensure reliable communication : its rank, its parent's address and the Received Signal Strength Indication (RSSI) from its parent.
These are the main steps in our network creation :

1. At first the Root node broadcasts its rank and address to all its neighbors using discovery message,
2. Each node set its parent's address, its parent's RSSI and its own rank as received rank + 1, broadcasts its rank and address to all its neighbors by discovery messages and discard messages received from nodes with a greater or equal rank, (and so on until we reach the bottom of the tree)
3. If a node receive a discovery message from another node with rank equals to its own - 1 and better RSSI than the one from its parent it updates its parent's address to the new one (same without checking for RSSI if the discovery message came from an even lower ranked node) then it broadcast the message down with its own rank,
4. After receiving first information about its parent, a node waits for a random time for adjustment before trying to reach them and send them data periodically by runicast.

The first step is repeated from time to time to adapt the network to modifications. (such as new nodes added in the network and the disappearances or displacement of old ones)

5 Data management & Computation

5.1 Formula

We use a first degree least squares fit that uses the last 30 data received from a sensor node. The compute slope is then compared to a certain threshold that we have set to 1 here.

$$y = ax + b$$

$$a = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2}$$

$$\text{if } a > \textit{threshold} \text{ then OpenValve}$$

5.2 Computation Node

We use a specific structure to maintain the state of the 5 nodes that we will do the computation for. This structure is stored in a global variable **static struct data taken_list[5]**.

- **struct route *route :**

  We take advantages of coding in C by using a pointer to the route maintained in our route_table. This will provide that the TTL will always be up-to-date.

- **int data[30] :**

  This is a simple array list of int which contains the data. We use it with the index next_slot_data.

- **int next_slot_data :**

  This variable is used to know which is the next position in our array list where we should put the next data. It's value is always between 0 and 29 as we apply a *modulo* 30 every time we increment, therefore we go back to the beginning of the table when it is filled out

- **int full :**

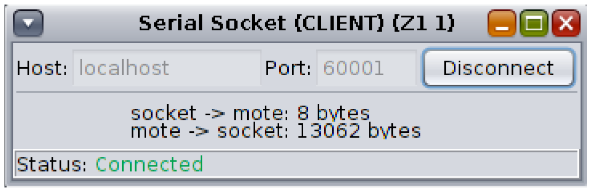
  This is a boolean like variable. Its role is to tell us if there are already 30 values in the list

The computation node behaves like a basic node, without sending new data, but when it receives runicast messages of **type 2** it will keep a list of 5 nodes for which it will compute the least squares fit and send the correct response (as the server would have done). When this message is received we check if the node is already in our routing_table and our taken_list.

If the node was not found in the route_table then we launch the function *add_to_routing_table* that will create a new route, will check if a position is available (old node's TTL = 0 or no route is linked) in the taken_list and put it there. If no position are available we have to pass the value to the parent.

If the node is in our taken list and still alive, we update its data values and check if we have enough data to do the computation. Once we have at least 30 values we verify that the threshold we gave wasn't exceeded. If it's the case we send a **type 3** to ask a specific node to open its valve. (A similar behaviour as the root node will happen if the **type 3** message fails to reach its destination as it will try to do a general broadcast from the root node)

5.3 Server



Serial Socket (CLIENT) (Z1 1)

Host: localhost Port: 60001

socket -> mote: 8 bytes
mote -> socket: 13062 bytes

Status: Connected

To connect our server to the border node, we first create a socket with localhost as host and a port number we chose. This port number (60001) should be the same as the one used to connect the border node. We launch our server with python3.

Once connected, the server will scan everything that is printed on the border node (with printf command). In order to take only the data intended for the server, we decided to print these messages with the following format: @:id1:id2:valueToCompute:@. (There are two id's because the id of a node is composed of two parts, ex: 2.0)

When the server receives the second @, it will know that the filled buffer ends with a message addressed to it. Using a split function we catch the node id and the value to compute and add it to our list (memorydata). If this id hasn't sent data yet, we create a new location for it in list memorydata, it's made easy with python's append function. In order to know where the datafor a certain id is located in our list, we use another list, whichindex, that matches the locations.

Once a node has sent 30 data, we can perform the leastsquare calculation. If the slope of our data exceeds a threshold that we set at 1, the server will send the id of this node to the border node to tell it to open the valve of this id. When a node has already sent 30 data and sees more, the oldest data is deleted.

6 Conclusion

In conclusion we programmed in C three different kinds of nodes that can communicate together and manage a routing table so that everyone can use runicast messages after the first round of broadcasting. The parent-child relation makes it very clear to see how the messages are exchanged in our network. The small addition of computation nodes in the network does not modify the behaviour of our route and sensor node but it allow us the decrease the load of messages send to the server. We found this project really close to a real life IoT case.