IoT 2023 CHALLENGE 3

(1) Name: Fontana Nicolò Person Code: 10581197

(2) Name: Gerosa Andrea Person Code: 10583298

The TinyOS application we implemented uses the TOSSIM tool to simulate the sending of a data packet through a network consisting of 7 nodes, using a simple routing protocol. It is composed of a coded part (nesC and Python) where we implemented the whole logic of the application and two text files used to specify the topology of the network and to add the underlying noise. Here we briefly present the code logic of the main nesC and Python files and the content of the main log file from TOSSIM.

RadioRoute.h

This header file contains the definition of the three different types of messages (data, request, reply), and the definition of the data structures for the messages and for the routing table. Only one structure for all types of messages is used: not all fields are relevant for all messages. Two simple structs of integer fields, plus one custom *message_type* field for the message, were enough to describe the network and the messages.

RadioRouteApp.nc

This configuration file defines all the components and the interfaces needed by the program. These include components for communication, for the LEDs and for the timers. Interfaces are wired to the components to make them work.

RadioRoute.nc

It contains the declaration of all useful variables and the instantiation of data structures. It is comprehensive of all the main basic procedures that simulate the communication between the nodes of the network:

- Booting the nodes
- Sending, receiving and parsing a packet
- Firing the timers
- Updating the LEDs

Most of them are event functions triggered by some events and so they are automatically called. They have been implemented mainly with basic constructs (if-else statements, switch cases, loops) due to the relative simplicity of the network behavior, so no particular design pattern has been used.

- *update_led* uses a round counter and the 8-digits person code to toggle the LEDs upon receiving a message. A switch case construct is used for discriminating the led to toggle.
- Event *Timer1.fired()* starts after the timer of 5 seconds for node1 fires and checks if the routing table contains the destination of the message (node 7) and decides which message to send between the

data and the route request, setting it up with the correct information to carry. In the case of a data message, it sets up the next hop, while in the case of the route request, the message is sent in broadcast.

- Event *Receive.receive* parses the received message with the help of a switch case to discriminate between the three different available message types.
  Upon receiving a data message the function checks whether the destination is the current node, stopping the procedure, or it is another node, forwarding the message to the next hop or reporting an error if it is not present in the routing table.
  The error is reported because to receive a data message means to be in some other node routing table as a next hop, this is possible only if the current node has notified the other nodes of knowing the route for the destination of the data message, thus it must have the destination in its routing table.
  Upon receiving a route request, the function does a similar check, either sending a route reply in case the current node was the destination (with cost 1) or the information of such node is present in the routing table (at an increased cost of 1), or broadcasting the route request in case the routing table doesn't offer information on the destination.
  Upon receiving a route reply the function checks if the current node needs to update the values in its routing table. In case the current node is not the destination, the routing table is updated accordingly and a new route reply with incremented cost (+1) and a new sender (the current node) is sent.
- *Actual_send* handles concurrency by checking the state of the lock variable which is used to avoid sending the packet when the radio is already transmitting. It also sets this variable to *true* in case of a successful send.
- Event *AMSend.sendDone* checks if the packet is sent and sets the lock variable to *false*.

Utils.h

It's a small library containing some support functions for the main procedures of *RadioRoute.nc*. All the operations to handle the routing table (initialization, update, query) are implemented here and so are the functions that create the actual messages of all the three types, as well as other short functions for debugging purposes.

- *print_msg_type* and *print_rt* are functions used for debugging.
- *is_dst_in_rt* consists of a simple loop that checks whether a destination is present in the routing table of a node.
- *init_rt* and *update_rt* perform the initialization of the routing table and its update, including both value update and creation of new entries.
- *get_next_hop* and *get_cost* query the routing table to obtain the next hop and cost, respectively, for a certain destination.
- *create_data_msg*, *create_route_req_msg* and *create_route_rep_msg* create the three different types of messages.

RunSimulationScript.py

Python script which is used for the simulation. It sets up the debug channels, instantiates the nodes with their links (from the file *topology.txt*) and it manages the number of events and the noise trace (from the file *meyer-heavy.txt*) for the network.

TOSSIM_log.txt

After the simulation, we obtained the *TOSSIM_log.txt* log file, which contains all the useful debug prints we chose to make the reader better understand what happened during the simulation.
We can see the application booting for all the nodes, the LEDs being initialized to 0 and the radio starting for each node. Timer1 then fires after 5 seconds and the communication through the nodes of the network starts with node 1 trying to send the message of value 5 to node 7.
We can immediately see that all the routing tables are in fact initialized to 0 because node 7 is not found in the routing tables of any node. A route request is sent from node 1 and the network gets flooded with its rebroadcasting that happens every time an intermediate node doesn't find the destination node in its routing table (we only print this event once for every node to keep the log readable). This process goes on until the destination is reached.
When reached, node 7 answers with a route reply. This message travels back through the network and the routing tables of the intermediate nodes get updated with the new values for the next hop and for the cost (also in this case the network is flooded with route replies, but we decided to print only the first occurrence for each node to improve readability).
Once node 1 receives a route reply with the next hop for node 7, it is ready to send again the data message, which follows the path built during the discovery phase, hop by hop, and finally gets to the destination. The simulation ends.
Finally, we also print the changes in the status of the LEDs for each node as requested and we save the history of these changes at node 6 in a separate log file (*Node6_leds_log.txt*). The changes happen every time a node receives a message of any kind (we decided not to print the receiving of the message for nodes different from 6 just to keep the TOSSIM log readable).