IoT 2023 Challenge 3

Group Members

Team Leader: Riaz Luis Ahmed.

Personal Code: 10372022

Second Member: Mario Cela.

Personal Code: 10685242

Report

In this section a sequential description of the flow of execution of the implemented code is provided, focusing on the most important details like used variables and implemented functions.

radio_route_msg. In the RadioRoute.h file, we defined the structure of the message that has been used for communicating between the nodes of the network. The struct is:

```
typedef nx_struct radio_route_msg {
  nx_uint8_t type;
  nx_uint16_t sender;
  nx_uint16_t destination;
  nx_uint16_t node_requested;
  nx_uint16_t value;
  nx_uint16_t cost;
} radio_route_msg_t;
```

A general description of the meaning of the fields is provided:

- type: it can assume three values: 0 if we want to send a data message, 1 if we want to send a ROUTE_REQ message, 2 if we want to send a ROUTE_REPLY message.
- sender: when a node sends a message, it will set this field to its `TOS_NODE_ID`, which is its identifier in the network.

- destination: for the purpose of this network, this field is used just by data messages, since requests and replies are sent in broadcast. However, is specified the destination of the message that is being sent.
- node_requested: this field is used by ROUTE_REQ and ROUTE_REPLY messages. For the requests, it specifies which is the node we are asking information about, while in the replies, it is used to specify the node for which a reaching cost is being communicated to the other nodes.
- value: this field is used just by data messages, and it is the actual content of the message, what a node wants to communicate to the destination. In our example, node 1 wants to communicate the value “5” to node 7.
- cost: defines the number of hops that needs to be done in order to reach the requested node. This value is set just in ROUTE_REPLY messages.

Routing table. Each node of the network has its own routing table.

It is declared as follow:

```
uint16_t routing_table[6][3]
```

The matrix is statically initialized when the devices is booted. We decided to not handle dynamical updates of the table during the execution since the network is known already, and it is of small dimensions.

So, when device is booted, we used the `initialize_routing_table()` function to insert all the nodes of the network inside the matrix except for the node itself, and we initialized the next hop and the cost of each row of the matrix to `UINT16_MAX`, which is the maximum value that a 16-byte integer may assume.

Every time a ROUTE_REPLY will be received, the routing table will be updated.

For simplicity, it has been implemented also a function that returns the row of the matrix in which it is stored information about a desired node, which is passed in input to the function itself. It is:

```
uint16_t get_row_index_by_node_id(uint16_t node_id) {...}
```

It will be used in the `receive` event in order to check values inside the routing table.

actual_send. This function transmits a message in the radio channel using the send function from AMSend interface. It takes two inputs: address, which is the packet destination address
and packet, which is the packet to be sent. It generates as output a boolean variable which is true when the message could be sent, false otherwise.

The message is sent only if there aren't other messages being transmitted by the same node. To ensure the latter condition a boolean variable called locked is used : locked is assigned a `TRUE` value when a message is already being sent and `FALSE` when a message has not been sent or the last message has completed the sending procedure. Function `actual_send` firstly checks the condition on locked so that it effectively sends the message only if no other messages have called the `AMSend.send` function before without having triggered the `AMSend.sendDone` event. If locked is `FALSE` (no other messages being sent) it sends the message and it assigns locked a `TRUE` value, otherwise it returns a FALSE value.

Boot interface. When the device is booted, we do two things:

- we initialize the routing table of the node calling the `initialize_routing_table()` function
- we start the radio using `call AMControl.start()`

AMControl interface. Two events of this interface are implemented:

- `startDone(error_t err)`: if the radio failed to start, we call again `call AMControl.start()`; otherwise, if the radio has been started correctly, we start once

timer 1 for 5 seconds, as specified in the requirements of the challenge. This is done just for node 1 through the following check:

```
if (TOS_NODE_ID == 1) {...}
```

- `stopDone(error_t err)`: this has been written because necessary for the execution of the code, otherwise the compiler would return an error.

Timer 1.This function is triggered by event `Timer1.fired()` which is started by `startDone` event (after the start of the node). Its task is to fill all the fields of the first request packet which is then sent to generate_send to be transmitted.

The function gets a pointer rrm to the message_t type variable `globalpacket`, which is then used to fill the fields of interest: field type is assigned value 1 as it is the type indicated for requests, field node_requested is filled with the requested value id, which is 7; sender is assigned the node id of the sender (TOS_NODE_ID) and destination is assigned the `UINT16_MAX` constant which is 65535, equivalent to `AM_BROADCAST_ADDR` which is the broadcast address. The value and cost fields are filled with `UINT16_MAX` constant but they are not of interest in this context.

AMSend interface. One event of this interface is implemented:

- `sendDone(message_t* bufPtr, error_t error)` : this event is triggered when a message has been sent, just to check that it was sent.

Receive interface. The receive interface implements can be divided in 4 parts:

- Receiving of a message of **type 0** (data message). When a node receives a data message, we are interested in two cases:

    - The receiving node is node 7: it means that the destination of the message has been reached, so we print using the `dbg(…)` function, and no other messages will be sent into the network.
    - The receiving node is any other node: it means that the current node is seen as the next hop by the previous node in order to reach the destination (node 7). So, it is generated a new message of type 0 (data message) that will be forwarded to the next hop specified in the routing table.

- Receiving of a message of **type 1** (ROUTE_REQ). In this part, three checks are done:
    - If the current node is not the requested node and if the values corresponding to the requested node inside current node's routing table are not initialized, we broadcast a new ROUTE_REQ message to the reachable nodes.

      In the following code it is shown how the check is done. Notice that `mess` is the variable that contains the message that has been received by the current node and that is being analyzed, and that the `row` variable stores the index in the routing table in which we can find information about the requested node.

      ```
      if (mess->node_requested != TOS_NODE_ID &&
                                  routing_table[row][1] == UINT16_MAX)
      ```

    - Otherwise, if the current node is the requested node, we generate a ROUTE_REPLY with cost set to 1 and we broadcast it to the reachable nodes.

      The condition is implemented as follows:

      ```
      if (mess->node_requested == TOS_NODE_ID)
      ```

    - Otherwise, if the current node has reasonable values to inside the routing table in order to reach the requested node (which means, that the values are not `UINT16_MAX` that have been set in the `initialize_routing_table()` function), we generate a ROUTE_REPLY and we broadcast it to the reachable nodes.

      The condition is implemented as follows:

      ```
      if (routing_table[row][1] != UINT16_MAX)
      ```

- Receiving of a message of **type 2** (ROUTE_REPLY). In this part we followed what have been specified in the text of the challenge. So, the checks that are done are:
  - If the current node is the requested node, we do nothing. The condition is implemented as follows:

    ```
    if (mess->node_requested == TOS_NODE_ID)
    ```

  - Otherwise, if inside the current node's routing table we have values still set to UINT16_MAX or if we have a cost that is greater than the one specified in the received ROUTE_REPLY message, then we update the routing table.

    The condition is implemented as follows:

    ```
    if (routing_table[row][1] == UINT16_MAX ||
                               mess->cost < routing_table[row][2])
    ```

    Once we updated the routing table, we check which is the receiving node:

    - If it is node 1, we can generate the data message with destination node 7 and value equal to 5 and we send it to the next hop specified in the routing table (is the sent flag is FALSE, flag that is used to send the data message just once);
    - Otherwise, if we are not node 1, we generate a new ROUTE_REPLY with cost increased by one with respect to the reply that has been received, and we broadcast it to the reachable nodes.
- **Led implementation.** Every node has the availability of three Leds. Led status update is performed by assigning two global variables

  ```
  uint16_t i_start = 7;
  uint32_t pcode = 10372022;
  ```

The types are handled with care to avoid overflow and truncation: variable `pcode` is an unsigned 32 bytes integer (to avoid truncation) and `tmp` is of the same type (to avoid overflow). The global variables were assigned so that their value could be kept through different receive events. Local variables are then declared:

```
uint16_t i;
uint32_t tmp;
uint16_t c;
```

Variable `i` is used to keep track of the number of times the receive event has been triggered, `tmp` is a dummy variable for the computation of the digit and c is the variable defined for the digit itself. For every value of `i` from 7 to 0, `tmp` is assigned the value of person code and then divided by 10 for i times. Afterwise the digit of interest (variable `c`) is computed by taking the rest of the division of the `tmp` value by 10 and by taking the rest of the division of the computed value by 3. Then the `c` variable value is used to toggle the corresponding LED by calling the functions ledxToggle (where x is the number of the LED), with the calls

```
call Leds.led0Toggle(); // toggle LED0
call Leds.led1Toggle(); // toggle LED1
call Leds.led2Toggle(); // toggle LED2
```

Finally the value of `i_start` is updated by restoring it to 7 if previously it was 0 (since the sequence of the person code must repeat itself) otherwise it is diminished by one unit, with the commands

```
if(i_start == 0)
   i_start = 7;
else
   i_start--;
```