IOT - Report Challenge 3

Giovanni De Lucia - Person Code: 10700658
Lorenzo Battiston - Person Code: 10618906

A.Y. 2022/2023

POLITECNICO
MILANO 1863

Contents

1 Introduction

This report presents the development of a TinyOS application and its simulation using TOSSIM. The application implements a simple routing protocol based on broadcasting. The objective is to create a network topology with 7 nodes and enable message transmission using routing tables. This report outlines the implemented routing protocol and the specific behavior of the nodes in response to various message types.

1.1 Header File and Definitions

In the header file, we have included the necessary specifications such as the routing message format and the structure of the routing table entry. We defined the routing table as an array of routing table entries.

To simplify the message handling process, we opted for a single routing message that contains all the fields required for different message types. The actual handling of the message is determined based on its type field.

```
typedef nx_struct radio_route_msg {
    nx_uint8_t Type;
    nx_uint16_t Sender;
    nx_uint16_t Destination;
    nx_uint16_t Value;
    nx_uint16_t NodeRequested;
    nx_uint8_t Cost;
} radio_route_msg_t;
```

Figure 1: Note that the structure shown can be further simplified by combining the Value and Cost fields into a single nx_uint16_t number, which is interpreted differently based on the message type. However, for the sake of code readability, we have chosen to retain the individual fields.

Regarding the routing table entry, we defined a structure that includes fields for Destination, Next Hop, and Cost. Initially, the Cost is initialized with a value of INFINITY to represent the absence of a valid route.

```
typedef nx_struct routing_table_entry {
    nx_uint16_t Destination;
    nx_uint16_t NextHop;
    nx_uint8_t Cost;
} routing_table_entry_t;

typedef routing_table_entry_t routing_table_t[MAX_NODES];
```

Figure 2: Although a fixed size array is not the most ideal choice for implementing a routing table, we opted for it in this case due to simplicity. While more advanced structures like a hash table could have been used for improved efficiency, the main focus of the challenge did not revolve around the routing table implementation.

1.2 High Level Code Logic

At a high level, the code follows the following logic:

Node1 Initialization:
- Nodes boot up and start the AMControl.
- Node1 attempts to send a DATA message to Node7, but it does not have a valid route in its routing table.

Sending ROUTE_REQ:
- Node1 triggers the sending of a ROUTE_REQ message in broadcast to discover a route to Node7.
- The ROUTE_REQ message propagates through the network until it reaches Node7.

Receiving ROUTE_REQ and Sending ROUTE_REPLY:
- When Node7 receives the ROUTE_REQ message, it generates a ROUTE_REPLY message and broadcasts it.
- The ROUTE_REPLY message eventually reaches Node1, containing the route information to Node7.

Updating Routing Table and Resending DATA:
- Upon receiving the ROUTE_REPLY, Node1 (and each node between Node1 and Node7) updates its routing table with the new route information.
- Node1 resends the DATA message to Node7 based on the newly obtained route.

To implement this behavior, a Timer is utilized. The Timer is started as a one-shot timer after the nodes boot up. It is responsible for triggering the message sending process from Node1 to Node7.

Additionally, the code includes a Receive function in the AMReceiver Component. This function extracts the payload from received messages and utilizes a switch statement based on the message type. For each message type, there are utility handling functions that handle the message appropriately and generate corresponding sends according to the logic of the broadcasting routing protocol explained in the Challenge Specification slides.

1.3 Message Handling

Whenever a message is received, the code updates the LED status to reflect the received message, providing visual feedback. After updating the LED status, the message is passed to the appropriate handling utility function based on its message type.

The utility functions, namely handleData(), handleRouteReq(), and handleRouteReply(), are responsible for implementing the logic described in the challenge slides.

```
event message_t* Receive.receive(message_t* bufPtr,
            void* payload, uint8_t len) {
/*
* Parse the receive packet.
* Implement all the functionalities
* Perform the packet send using the generate_send function if needed
* Implement the LED logic and print LED status on Debug
*/
dbg("receive","Message Received\n");
dbg_clear("receive","\t Payload Length: %hhu\n",call Packet.payloadLength(bufPtr));
  if(len != sizeof(radio_route_msg_t)) {
    return bufPtr;
  } else {
    radio_route_msg_t* rtm = (radio_route_msg_t*) payload;
    ledUpdate();
    switch(rtm->Type) {
        case DATA: handleData(rtm); break;
        case ROUTE_REQ: handleRouteReq(rtm); break;
        case ROUTE_REPLY: handleRouteReply(rtm); break;
    }
    return bufPtr;
  }
}
```

When a data message arrives, the node checks whether the destination node is reachable by consulting its routing table. If the destination node is not reachable, the node broadcasts a route request. However, in the final run, this scenario should not occur because when node 1 receives the route reply, it is assumed that all intermediate nodes have also received it.

If the route is present, the message is sent hop by hop, always setting the original sender as the sender. This ensures that the destination node knows who initially sent the message.

1.4 Led Update

In addition to the message handling and routing functionality, the code also includes LED status updates based on a person code array.
The LED status is updated using the modulo of the current digit of the person code. The person code is defined as an array of digits.
The current digit is incremented at each arriving message. If it overflows, it is reset back to 0.
The modulo 3 of the current digit determines which LED should be toggled.
If the node updating the LED status is Node 6, the LED status is also debugged on a dedicated channel using the bitmask provided by Leds.get().

```
void ledUpdate() {
  uint8_t led = person_code[led_iter++] % 3;
  switch(led) {
    case 0:
      call Leds.led0Toggle(); break;
    case 1:
      call Leds.led1Toggle(); break;
    case 2:
      call Leds.led2Toggle(); break;
  }
  if(led_iter == 8) led_iter = 0; // at the end return to the first
  if(TOS_NODE_ID == 6) dbg("led_update","Led Update: %hhu%hhu%hhu\n",
                                        call Leds.get() & LEDS_LED0,
                                        call Leds.get() & LEDS_LED1,
                                        call Leds.get() & LEDS_LED2);
}
```

1.5 Simulation with TOSSIM

For the simulation part, we followed these steps:

Compiling .nc files:

- We used the command "make micaz sim"
- This process also generated a TOSSIM.py script, which is required for running the simulation.

Adapting RunSimulationScript.py:

- We utilized the RunSimulationScript.py file as a template and made necessary adaptations.
- Specifically, we created channels for debugging purposes. Each important function had its own dedicated channel for debugging.

Updating Node Count:

- We incremented the number of nodes to 7 in both the topology and noise generation parts.
- The topology.txt file was also updated to reflect this change.

Running the TOSSIM simulation:

- The tossim simulation can be executed by running the command "python RunSimulationScript.py".
- By running the command "python RunSimulationScript.py | grep Led", you can view the status of the LED. It is important to note that the initial state (000) is not displayed as the state is only shown upon receiving an event.