IoT 2023 CHALLENGE 3

Name: Mirko Bitetto Person Code: 10928506

Name: Giulio Saulle Person Code: 10626444

Message Format and Routing Table Specification

The provided code defines the routing table that will be used to route the message across the network. The **`RoutingTableEntry`** structure represents each entry in the routing table and consists of fields such as **`destination`**, **`nextHop`**, and **`cost`**. The **`destination`** field holds the node identifier for a specific entry, while the **`nextHop`** field stores the neighboring node to which packets should be forwarded to reach the destination. The cost field represents the associated cost of the route. The overall routing table is an array of **`RoutingTableEntry`** structures, allowing for efficient storage and lookup of routing information.

```
// Routing table entry
typedef struct {
  uint16_t destination;
  uint16_t nextHop;
  uint8_t cost;
} RoutingTableEntry;

// Routing table
RoutingTableEntry routingTable[MAX_ROUTING_TABLE_SIZE];
```

The **`radio_route_msg`** structure defined in the **`radioroute`**.h file is used to represent messages that will be exchanged between the nodes. It serves as a unified message format that accommodates different types of messages, including data messages, route request messages, and route reply messages. The structure includes fields such as **`type`**, **`sender`**, **`destination`**, **`value`**, **`nodeRequested`**, and **`cost`**, which store relevant information for each message type. By utilizing a single message structure, the code simplifies the message handling process by eliminating the need for separate structures for each message type.

```
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;
```

The provided functions are designed to support the lookup and routing process within the project. The **`isRouteAvailable`** function checks whether a route to the specified destination exists in the routing table. The **`isCostLower`** function compares the provided cost value with the existing cost associated with the destination, determining if the provided cost is lower. The **`updateRoutingTable`** function allows for updating the routing table with new or modified information for a specific destination, including the next hop and cost. The **`getNextHop`** function retrieves the next hop node identifier from the routing table for the given destination. Finally, the **`getCost`** function retrieves the cost associated with the specified destination from the routing table.

```
bool isRouteAvailable(uint16_t destination);
bool isCostLower(uint16_t destination, uint16_t cost);
void updateRoutingTable(uint16_t destination, uint16_t nextHop, uint16_t cost);
uint16_t getNextHop(uint16_t destination);
uint16_t getCost(uint16_t destination);
```

Boot Sequence

The `Boot.booted()` event handler is triggered when the application finishes booting. It proceeds to start the radio using the `AMControl.start()` function. The `AMControl.startDone()` event handler is triggered when the radio start operation completes. If the radio starts successfully, it initializes the routing table by setting the destination, nextHop, and cost fields for each entry. Additionally, it starts a timer using the `Timer1.startOneShot()` function set to 5 seconds. In case the radio fails to start, the event handler attempts to start the radio again by recursively calling `AMControl.start()`.

`Timer1` it is responsible for implementing the logic to trigger Node 1 to send the first REQ (Route Request) packet. When it fires, it checks if the current node (`TOS_NODE_ID`) is equal to 1. If so, it proceeds with the packet generation and scheduling process. If the payload pointer is obtained successfully, the `type` field of the payload is set to 1 and the `nodeRequested` field is set to 7. The event handler then generates and schedules the transmission of the message by calling the `generate_send()` function. If any errors occur during the payload pointer retrieval or message transmission scheduling, appropriate debug error messages are printed.

The `generate_send()` function stores the packet and address in global variables and starts Timer0 to schedule the send operation. It allows sending only one message for each Route Request and Route Reply type.

`Timer0.fired()` is triggered when Timer0 fires. Its purpose is to calls the `actual_send()` function with the destination address and queued packet as parameters, which initiates the transmission of the packet to the specified destination.

The `actual_send()` function is responsible for performing the actual transmission of the packet. It takes the destination address and the packet as input parameters. The function first checks if the radio is locked and returns `FALSE` if it is. It then examines the type of the payload message and performs the appropriate send operation using the TinyOS interfaces. For a data message (type 0), the function sends the packet to the next hop based on the destination address. For a route request message (type 1), the function broadcasts the packet to all nodes. For a route reply message (type 2), the function also broadcasts the packet to all nodes. In the case of an invalid message type, the function returns `FALSE`.

The `AMSend.sendDone()` event handler is triggered when a message has been sent using the Active Message (AM) interface. It checks if the message was successfully sent by examining the error parameter. If the message was sent successfully, the function unlocks the radio and logs a debug message indicating that the packet was sent at the current simulation time. Additionally, if the sent message was a data message and the current node is Node 1, the `data_sent` flag is set to `TRUE` to indicate that no more data messages should be sent from Node 1.

The `Receive.receive()` event handler is responsible for processing received packets. It first checks the length of the received packet to ensure it matches the expected size. If the length is incorrect, the function returns the buffer pointer without further processing. Otherwise, it proceeds to parse the received packet and extract the message type, sender, and destination. Based on the message type, different actions are taken. For data messages (type 0), if a route is available to the destination, it schedules the message transmission using the `generate_send()` function. For

route request messages (type 1), it checks if the node is the requested node and sends a route reply message accordingly. If the node knows the route to the requested node, it also sends a route reply message. If none of these conditions are met, the route request message is forwarded. For route reply messages (type 2), it updates the routing table if necessary, forwards the message, and may schedule a data message transmission if certain conditions are met (Node 1 receives the REP with node requested 7). Unknown message types are handled as errors.

The simulations ends when Node 7 receives the data message containing value 5, since no more message exchange are scheduled between the nodes.

Led Logic

The provided code snippet represents a function that implements the LED logic based on received messages and the leader person code. The **`person_code_digit`** is calculated by taking the modulo of the **`received_msg_counter`** with 8, ensuring it stays within the range of 0 to 7. The function uses a switch statement to determine the LED to activate based on the remainder of **`person_code[person_code_digit]`** divided by 3. Depending on the case, the corresponding LED is toggled, and a debug message is printed to indicate the received packet, person code digit, and the activated LED. After the switch statement, a debug message displays the current status of the LEDs by retrieving their values using the **`Leds.get()`** function. Finally, the **`received_msg_counter`** is incremented to track the number of received messages.

```
    person_code_digit = recived_msg_counter % 8;

    switch (person_code[person_code_digit] % 3) {
      case 0:
        call Leds.led0Toggle();
        dbg("led_0","Received packet %hu, person code digit %hu, activating LED 0\n", recived_msg_counter,
person_code[person_code_digit]);
        break;
      case 1:
        call Leds.led1Toggle();
        dbg("led_1","Received packet %hu, person code digit %hu, activating LED 1\n", recived_msg_counter,
person_code[person_code_digit]);
        break;
      case 2:
        call Leds.led2Toggle();
        dbg("led_2","Received packet %hu, person code digit %hu, activating LED 2\n", recived_msg_counter,
person_code[person_code_digit]);
        break;
    }

    dbg("led_status","LED status: %d%d%d\n", (call Leds.get() & LEDS_LED0) , (call Leds.get() & LEDS_LED1)/2,
(call Leds.get() & LEDS_LED2)/4);

    recived_msg_counter++;
```

To retrieve the led status history of Node 6 the simulation was run piped with grep. The status is the following:

```
user@user-iot:/mnt/hgfs/folder$ python RunSimulationScript.py | grep "DEBUG (6): LED status"
DEBUG (6): LED status: 010
DEBUG (6): LED status: 110
DEBUG (6): LED status: 010
DEBUG (6): LED status: 011
DEBUG (6): LED status: 010
```

RadioRouteApp

The configuration, named `RadioRouteAppC.nc`, defines the components used in the application. These components include `MainC` and `RadioRouteC`, which are responsible for the main functionality and the `RadioRoute` module, respectively. Additionally, the `LedsC` component is declared to handle the LEDs in the application. The code also includes components for the ActiveMessage sender and receiver and for Timers. By defining and wiring these components, the configuration sets up the necessary infrastructure for implementing the desired functionality in the `RadioRouteC.nc`.

Simulation With TOSSIM

In the simulation script a few debug channels were added like `led_status`.

The topology has been set accordingly to the specification, with all links bidirectional and -60dBm as gain for all entries.