Smart Bracelets

IOT Project

2022

Michele Lucio

10490074

Sommario

Introduction

The project requires to develop a pair of devices that at the boot begins a pairing phase, and after found the device to couple to, start to send periodic info messages containing information about the position coordinates and the kinematic status of the person who is wearing it (Standing, Walking, Running, Falling).

The project has been developed simulating the communication between the two couple of devices in the simulation environment TOSSIM.

Project Files

For this project I've created the following files:

- **`SmartBracelets.h`** contains the struct of the message sent by the two types of motes, and some constant useful to determine the status of the application
- **`SmartBraceletsApp.nc`** where all the used components were declared
- **`SmartBraceletsC.nc`** contains the logic of all the application
- **`RunSimulationScript.py`** that contains the python code needed to execute the TinyOS SIMulator (TOSSIM),
- **`topology.txt`** contains configuration parameters for the 2 motes antenna
- **`meyer-heavy.txt`** contains data to use in TOSSIM to simulate communication noises.

SmartBracelets.h

In this file I've defined the ***`my_msg_t`*** type for the messages sent by each type of node. It's composed by:

- **`msgType`** (type nx_uint8_t): the type of the message (PAIRING, CLOSE_PAIRING, OPERATION_MODE, INFO)
- **`key`** (type nx_uint8_t): an array containing the device key, used for the pairing phase (in SmartBraceletsC.nc the key it's defined as array of char, here I've considered the corresponding int of the char according to the ASCII table)
- **`sendingDeviceID`** (type nx_uint8_t): contains the id of the device which is sending the message
- **`pos_X`**, **`pos_Y`** (type nx_uint16_t): x, y values of the child device's position
- **`kinematicStatus`** (type nx_uint16_t): the status of the child device (can be Standing, Walking, Running, Falling).

A ***`serial_msg_t`*** type for the message sent in the serial communication, containing **`pos_X`**, **`pos_Y`** and **`kinematicStatus`** used in the same way of the previous message.

I've also defined some constant variables for the message type and for the kinematic status.

SmartBraceletsApp.nc

In addition to the main components needed to boot the application (***MainC***, ***SmartBraceletsC*** as App), I've used the ***AMSenderC***, ***AMReceiverC***, ***ActiveMessageC*** components to manage the communication with messages, ***SerialActiveMessageC*** to manage the serial communication, ***RandomC*** component to generate random numbers, 3 ***TimerMilliC*** components to use a timer for the pairing-pashe, the operation-phase, and the missing-phase, and ***FakeSensorC*** component to have a sensor that gives the data about the kinematic status of the child device according to the project's given probability (P(STANDING) = P(WALKING) = P(RUNNING) = 0,3 ; P(FALLING) = 0,1).

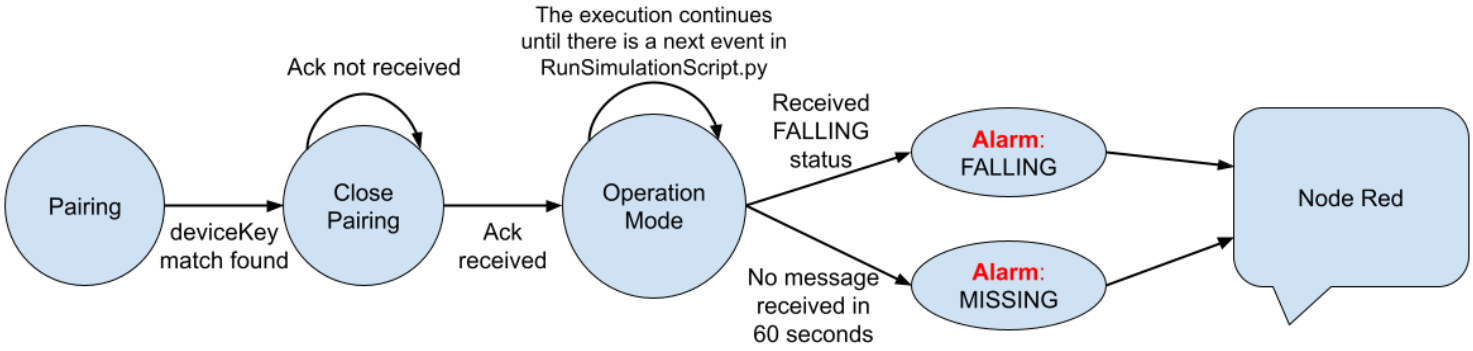
SmartBraceletsC.nc

Here I used an array of 20 char (***`deviceKey`***) to store the unique-key needed for the parent-child pairing process, two uint8_t variables to store: (***`linkedDevice`***) the id of the coupled device, (***`currentMode`***)

the identification of the current working mode and two uint16_t variables (***last_pos_x***, ***last_pos_y***) to store the last x and y position received from the child device.

This application is designed to boot in the following way: in the RunSimulationScript.py all the new nodes must be added and turn on so that the first node is a parent node and has an id=1, immediately after, his child must turn on and need to have id=2, and so on for all the new other couple of devices, that must turn on in the order: parent, child, parent, child, …, each one with id incremented by 1 respect to the previous one (in this project implemented for 4 devices but it's possible to add other couple of devices).

At the boot of the node, the first process called it's ***getDeviceKey()*** that manages the unique device key: for that part I used an external file to store the random generated key. First, the node tries to open the *"DeviceKey.txt"* file in read mode, if the file doesn't exist, it means that, that node it's the first one and it's a parent node, so it creates the file, open it in write mode and store on it 20 random char. If the file already exists, it means that the node which is trying to access it, it's a child or a new parent: if it is a child, it only needs to read the last line of the *"DeviceKey.txt"* file, otherwise if it is a new parent node, it needs to append a new random generated key at the end of the file.



Finite-state machine of the program

After the assignment of the random generated key, the node sets its **currentMode** to **PAIRING**, and after the start of the radio, the **PairingTimer** is started with a periodicity of 1 second.

Each time the **PairingTimer** is triggered, it's called the ***sendReq()*** function where the pairing message is prepared to be sent in broadcast and the payload of the message it's configured setting the **msgType=PAIRING,** the **deviceKey** with the key assigned at the boot of the node and **sendingDeviceID** with the id of the node that it's sending the message.

In the pairing phase, when a node receives a message, the event ***Receive.receive()*** it's triggered, here the node parses the **deviceKey** contained in the massage, and if it is different from the one of the node, it simply discards the message, otherwise, if it is equal, the node stores the **sendingDeviceID** variable contained in the message in the global variable **linkedDevice**, and because the device to couple to, has now been found, set also the **currentMode** to the status **CLOSE_PAIRING,** and call the **sendReq()** function.

In the **sendReq()** function, because the **currentMode** is now **CLOSE_PAIRING,** the payload of the message it's prepared, setting the message type to **CLOSE_PAIRING** and the destination of the message is set using the node id stored previously in the global variable **linkedDevice,** also the ack request it's activated for this response message, and it's finally sent with a unicast message to the coupled node. The coupled node that will receive this message, within the event ***Receive.receive(),*** because it is in **CLOSE_PAIRING** mode, will stop his **PairingTimer** and will set its **currentMode = OPERATION_MODE.** The other node (that will be the child node) in the ***AMSend.sendDone()*** will check if the sent message has been acked, if it has been acked correctly, the node will also stop his **PairingTimer**, will set its **currentMode = OPERATION_MODE** and finally it will start the **OperationTimer** with a frequency of 10 seconds. Otherwise,

if the message hasn't been acked correctly, the node will remain in **CLOSE_PAIRING** mode and the pairing timer will fire again calling the **sendReq()** function and the procedure will be repeated, so that the other node will receive another **CLOSE_PAIRING** message.

Now both the nodes are in **OPERATION_MODE** and if they receive a **PAIRING** message sent in broadcast by other nodes, they will simply discard them checking if the sendingDeviceID contained in the message is different by the value contained in the global variable **linkedDevice**.

In **OPERATION_MODE** when the **OperationTimer** is fired, the child device calls the ***sendResp()*** function where the **Read.read()** event it's called to obtain the kinematic status from the sensor, that will be received in the ***Read.readDone()*** event, where an INFO message is prepared to be sent to the parent node and the payload of the message it's configured setting: **msgType = INFO**, **kinematicStatus** = data obtained from the sensor, **pos_X** and **pos_Y** are initialized with the data obtain calling the **Random.rand16()** component. Then the ack it's requested for the packet, and it's sent in unicast to the linked device using as destination address the id of the linked node stored in the **linkedDevice** global variable.

When the parent node receives the INFO message in the ***Receive.receive()*** event, first it stops the **MissingTimer** (at the first INFO message there is no **MissingTimer** active), because a message has been received from the child device and the MissingTimer can be reset, then it's called the function **MissingTimer.startOneShot**() to start a new MissingTimer with a duration of 60 seconds and finally it stores the x and y position of the child device in two local variables.

The MissingTimer it's used to count the time elapsed of two received message from the child node, if the parent node doesn't receive any message from the child node in a minute, the MissingTimer it's triggered, and an alarm message is sent using the position received from the child node in the last sent message, that has been saved in the 2 local variables, as said before.

If the message that the parent node receives from the child contains a **kinematicStatus** of type **FALLING ,** an alarm message is sent by the parent with the x and y position contained in the received message. The same message it's also sent using the ***serialSend*** function.

Node-red (alarm system)

In node red, the flow starts with an **exec node** that executes the command:

cd "/home/user/Desktop/Smart Bracelets" ; make micaz sim; python RunSimulationScript.py

that execute 3 commands in one line: first change the directory to the folder where the project it's located, then compile the project, and finally executes the python script, then with 2 **function node** I've parsed the output of the program searching for MISSING or FALLING message, and in case they are found, the are also showed in the debug console in node red. After the exec node, the output of the python script it's also written in TossimLog.txt file to have a complete log file of the Tossim simulation.

RunSimulationScript.py

Here the simulation environment is setting up: radio channels are activated, some debug channels are created, node 1,2,3,4 are created and activated at time 0,2,4,6 respectively from the beginning of the simulation. The radio channels are then setting up using the "topology.txt" and "meyer-heavy.txt" that also adds noise to the communication in the radio channel. Finally, the simulation is started running event in a for loop. To simulate the MISSING status of one node and allow the **missingTimer** to be fired, the node 2 it's manually turned off with the function **node2.turnOff().**