IOT PROJECT - SMART BRACELETS

Federico Romeo - 10566536
Simone Sarti - 10580595

ProjectC.nc

When a mote boots up, some initializations take place. We set the radio of the mote to a non-locked state, then depending on the moteID a role for it is defined. The mote will be the PARENT if the moteID is even, CHILD otherwise. Also, using an integer division by 2, we associate the same key (string of 20 chars plus string terminator) to pairs of parent-child bracelets. The last step of the boot phase is the activation of the mote's radio; the operation is repeated until success.

At this point the pairing phase begins. A Pairing Timer is activated on the mote, which fires at intervals of 5 seconds. Every time the timer fires a PAIRING MESSAGE is sent, which is defined by a "pairing" type and whose key field is relevant for the pairing process. All other fields of the message take default values. This message is sent in broadcast. Other bracelets receive this message, see that it is of type "pairing", and compare the received key to their own. If the keys coincide, that mote responds by creating a PAIRING RESPONSE MESSAGE, which is sent in unicast to the sender of the original pairing message. We set the ack flag for this response message, as confirmation will be important later to conclude the pairing phase. In the *pairing response* type message, only the type is really relevant, other fields are set to default values.

At this point, the mote that sent the original pairing message should receive the corresponding pairing response from the mote with the same key. When the response is received, the mote sets the address of the mote who sent the response message as the one it is paired with, and changes its pairing status to true. At this moment the pairing timer is stopped as the mote has found the one it should be paired with. The pairing response message is acked to the sender because of the flag we set previously, therefore the sender will receive a pairing response ack. If the ack is not received, the pairing response message is sent again until success.

To conclude the pairing phase, two conditions must be satisfied:

- A mote must have received a pairing response message from its counterpart (I'm paired with it)
- A mote must have sent a pairing response message to its counterpart, and have received the ack (It is paired with me)

On reception of either the message or the ack we check which of these events took place up to this point; if both conditions are satisfied the motes move to the info-message exchange phase. We do this by activating the Info Timer on the child mote. This timer is once again a repeating timer, firing every 10s. Every time it fires, a call to a FakeSensor component is performed, which returns a custom data structure composed of three fields: *x ,y, kinematic_status*. These three values are random uint16_t values. The values for x and y are fine as they are, but the kinematic status is not as it should be one of *STANDING, WALKING, RUNNING, FALLING* (the first three happening with probability P=0.3, the last one with P=0.1). The transformation between integer and kinematic status is done by checking which interval of 0-65535 the number falls into, the intervals having sizes defined by the probability values. As for the message types and the mote types, an enum for the kinematic statuses exists.

Having the values for *x, y* and *kinematic status*, we fill the INFO MESSAGE and send it to the parent mote. The ack flag is set also for this type of message. In this case, a limit of two retransmission

attempts is set if the ack is not received. *x,y* and *kinematic status* are saved by the mote to be able to send them again in case a retransmission is needed. Every time the parent node receives an info message, it saves x and y. If the kinematic status is FALLING, an alert which includes the coordinates x and y is raised. A second type of timer is also activated on the parent on reception of an info message, an *outOfRange* Timer, which is a one-shot timer set to fire after 60s, which gets reset if an info message arrives from the child within this time interval. If the 60 seconds pass without receiving another info message, a MISSING alert is raised, containing the last saved values for x and y. We simulated the mote going out of range by turning off the child mote.

ProjectAppC.nc

In this file the components required to make the bracelets work are defined and wired together. Among them the radio components to send/receive messages and acks are found, together with the threed timers (Pairing, Info, OutOfRange) and the Fakesensor component.

Project.h

In this file we defined the enums and structs we used in our system.

Enums are used to define:

- mote type (*parent/child*)
- message type (*pairing / pairing response / info*)
- kinematic status (*none / standing / walking / running / falling*)

Structs are used for:

- message (*message type, sender ID, key, x, y, kinematic status*)
- fakeSensor data (*x, y, kinematic status*)

FakeSensorC.nc and FakeSensorP.nc

These two files were modified to account for our need to return the structure composed of three fields with random values (*x, y, kinematic status*) instead of the classical single random value.

Makefile and Topology.txt

In the Makefile, we just needed to change the component to be ProjectAppC.

Topology was modified to define all combinations of the 4 motes we used for the required use case.

RunSimulation.py and Simulation.txt

We adapted the script of the last challenge to work with 4 motes, to read the new debug channels, and to turn off a child mote (mote1) after a certain number of steps during the simulation. The result of the simulation can be read in the file *simulation.txt.*

In this file we also defined a serial forwarder using python serial package to send the ALERTS on the serial port (other messages are filtered out). Before running the simulation, follow the instructions in the next paragraph to make the serial work properly.

Running the simulation:

- start node-red: `node-red`
- import the flow (from *nodered_flow.txt*) and deploy
- run: `sudo python RunSimulation.py`

Serial

At first we added a *Serial* component in node-red, and tried to connect directly to that with the python *pyserial* package, but we didn't manage to make it work. So we decided to use the *TCP* component in the node-red flow instead, together with the *socat* command, a cmd command that creates virtual connections by simulating a bridge between serial ports and an http server. Thus we used *socat* to read the input from the serial port and forward in output with *TCP* to node-red.

Installations required to make the system work:

- install pyserial: `sudo pip install pyserial -t /usr/local/lib/python2.7`
- add user to *dialout* group: `sudo gpasswd --add ${USER} dialout`
- install socat: `sudo apt-get -y install socat`

Node-Red

The flow is composed of:

- TCP node: it receives on port *60001*, listening for strings (terminated by the character "\n")
- Clear Alert function node: simply used to reformat and clean the strings received
- Debug node: to be able to print the messages in the debug console

The log of node-red for our (last) simulation is in the picture, as only ALERTS were required to be sent via the serial port and there were only 2 of them.