Mangano Davide - 10627074
Panzeri Matteo - 10615045

REPORT SMART BRACELETS PROJECT

Introduction and general overview

First of all, we decided to implement the **Smart Bracelet** project since it was the most appealing in our eyes thanks to its applicability in everyday lives. Secondly, after selecting the project, we had to establish which ambient simulator to choose between TOSSIM and Cooja: since the professor during classes said that Cooja could have problems with many motes, we decided to use a more immediate command line alternative like **TOSSIM**, that is both faster to run and easier to debug. Thirdly, we chose to manage the pairing phase with the use of 2 global variables, endedPairing1/2 that are marked as FALSE until the unicast confirmation message is acked.

Headers

Now, our objective was to define parameters in the headers file; we knew that messages had different purposes and content across phases, so we opted to keep two separate kinds of messages for the two distinct phases: pairing and operational mode. In the pairing one, we had just to keep the key that has to be communicated between motes, while in the operational mode we needed more parameters like *data* and *X* and *Y* coordinates.

Configuration

The next step was to define interfaces: we kept the usual ones, with the addition of **3 timers** (used as pairing, info and missing timeouts) and the **PacketAcknowledgements** component for the second step of the pairing phase (the confirmation unicast message).

Pairing phase: Design and choices

This is the first part of the project, the one that is called immediately upon booting. After calling SplitControl in the Boot function, two precise actions are executed: starting the pairing timer (**PairTimer**) and calling the **sendPairing** function. An important note here is that the only purpose of the PairTimer is to call again the sendPairing function when expired and the pairing phase is still not completed.

In the aforementioned sendPairing function, a broadcast message is sent of 2 different random keys; this sending will immediately issue the **sendDone** and **receive** events that will operate as follows.

The sendDone event will not generate anything interesting in this moment, while the receive event is the main focus since after a few checks on variables (pairing not ended, corresponding correct TOS_NODE_IDs, equal random key), some debug messages are sent and the **sendResp** function is called.

This is the final passage, in which a unicast message is sent to the corresponding mote in order to complete the pairing; in particular, we decided to implement this last part with an ack request so to not make incomplete sequences of messages or to have to send another confirmation message. After such operations, since a message is being sent (and in the code it is checked that it is sent correctly), as before, **sendDone** and **receive** events are issued.

In this case, the receive event won't do anything while the sendDone is the one that completes the sequence: after checking the correctness of the TOS_NODE_ID, it sets the corresponding global variable endedPairingX (with "X" being either 1 or 2) to TRUE, such to end the pairing phase for that couple of nodes. The execution of these phases is done in parallel for both nodes.

Operation phase: Design and choices

As said before, during the operation mode we defined a new message type different from the one used during the paring phase.

The **info_message** struct mimics the structure of the object produced by the fake sensor, it contains the two spatial coordinates X and Y and the data which represent the child status.

Despite the data structure difference in the two project phases: paring and operation, the defined methods are the same. Now we will discuss the design choices we made while programming the methods from the operation phase perspective.

The **AMSend.sendDone** event, in operation mode, simply checks that no errors occurred while sending the packet and in the positive case frees the channel.

The **Receive.receive** event handles the reception of a message in our case a message from the child bracelet.
If the paring phase has ended (one of the two boolean flags is true) the method checks if the received message does contain a paring string, in this case it ignores the message. (This may happen right after the pairing phase has been completed and the delays of the channel cause the generation of additional pairing messages).
If this does not happen then the method checks if the message has been sent by the right mote and in the affirmative case it prints the information contained in the message and restarts the missing timeout.
In case of a falling state the method also generates the alert.

The event **FakeSensor.readDone** is used by the child motes and ensures that no message is sent before the reading of the position and action of the bracelet has been done. After this control the method send_info_message is called .

The **send_info_message** function checks if the channel is free and, if it is free, it composes the message and sends it to the proper mote.