Final Project

Matteo Bordignon,
Massimo Terzi

We decided to complete the project 1: Keep your distance.

The assignment asks to simulate motes that are able to alert if the social distancing is not respected for more than 5 seconds. To do so, each mote broadcasts a message with its ID every $500\ ms$. When a second mote receives ten consecutive broadcast messages from the first mote, an alert is sent.

The first thing to do is to decide where to place the broadcast messages processing in order to check if ten consecutive messages are received from the same mote. We think of two scenarios:

1. To minimize the memory requirements and the complexity of the motes, we can place the processing in the online server, represented by Node-Red in this project. In particular, what each mote does is to simply *forward* the broadcast message received, attaching its own ID. Then, the online server is able to process these messages and fire an alert.

   The downside of this approach, from an IoT point of view, is that the mote forwards every message it receives, and it can be very energy consuming.

2. The second option is to place the processing inside the mode and, when an alert needs to be fired, the mote sends it to the online server. This requires to allocate some memory to store counters and, in general, it makes the mote a bit more complex to program.

   On the other hand, the number of forwarded messages (the broadcasts are the same) is reduced by a factor 10 in the worst case scenario (if no streaks are interrupted, the mote forwards one every ten received broadcasts). Since the transmission is the main cause of energy consumption in an IoT device, such upside is very beneficial.

Overall, we think that the trade-off is strongly leaning toward the second option, therefore we decided to follow it and to implement the counter **inside** each mote, forwarding a message to the online server only if an alert needs to be sent.

Note: Unfortunately, since the `dbg()` function does not work with TinyOS, we must use `printf()` to both create a proper log file and send information to Node-Red, generating a lot more traffic then needed from motes to server; in an ideal situation, we would only send alert messages, gaining every benefit of our implementation.

TinyOS Implementation

Our idea to implement the check inside the mote is the following:
Besides the ID, an **incremental counter** is inserted inside each broadcast message by the mote.
Then, each mote stores in the memory **two arrays**, one to save the last counters received called `last_counter[]`, and the other to save the number of consecutive messages received called `streak[]`(*i.e.* the active streak between the mote ego and all the other motes). We assume that the mote ID (namely `TOS_NODE_ID`) is an integer that can be used as index for such arrays; if this is not the case, the problem can be simply solved using a different data structure, like a dictionary. Now that we have all the components, we can see how the check works. Basically, when a mote receives a broadcast message, it extracts the sender ID and the message counter. Using the ID as index, the mote can access the last counter received by the same sender and the active streak of consecutive messages with the sender. If the difference between the new counter and the last counter is *one*, the mote is receiving **consecutive** messages, so the active streak number is increased. Otherwise, if the difference is greater than one, this means that *at least* one message has been missed, so the active streak counter is reset to one. Actually, there is an exception of this case: when the incremental counter goes in overflow, the difference between two consecutive messages is different than zero. We can easily solve this problem, though, by including in the if statement the condition that the last counter is 65536 (maximum value for UINT16) and the new one is 0.
In any case, following the aforementioned logic, we are able to keep track of the length of the active streak. Then, if the length is equal to 10, the alert message containing the two IDs (ego_ID and sender_ID) is sent to the online server, and also the active streak is reset to one.
Note: We decide to interpret the assignment as "Every ten consecutive messages received, an alert needs to be sent" instead of "After ten consecutive messages received, send an alert each new message received".

Note: Since we need to create a TCP connection for each mote, a **maximum** number of motes must be hard-coded; we defined it as a global variable MAX_MOTES and initialized it to 8, adding 8 TCP sockets in Node-Red accordingly. As far as we know, Cooja doesn't create a variable that stores the number of motes created for a specific simulation, hence we unfortunately must suppose it a priori.
The program can be easily adapted to deal with more then 8 motes by changing the value of MAX_MOTES and increasing the number of TCP connection in Node-Red.
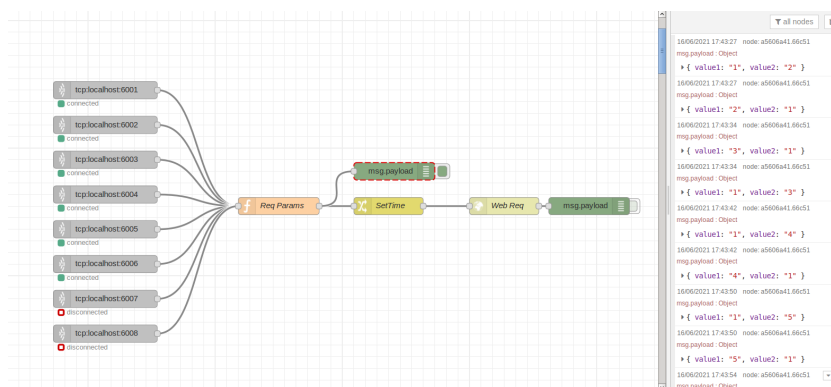
Node-Red Implementation

Figure 1: Node-Red scheme

In Node-Red, the online server of this project, we create the TCP ports to receive the alert messages from each mote. Then, we connect these ports to a simple function block in which we split the alert message in the two IDs and assign them to the corresponding values in the dictionary created in the payload of the object. Finally, we forward such object to the applet created with IFTTT in order to receive the alert message by e-mail (Figure 2).

(a) Alert received from Mote 3

(b) Alert received from Mote 4

Figure 2: Messages received when Motes 3 and 4 are too close for too long

Cooja Simulation

The project was simulated with *Cooja* connected to *Node-Red*. We start the simulation with 6 motes not in range with each other, all connected to their respective socket in Node-Red (Figure 3); only Mote 1 is moved around during the test in order to trigger the Closeness alert from the others.
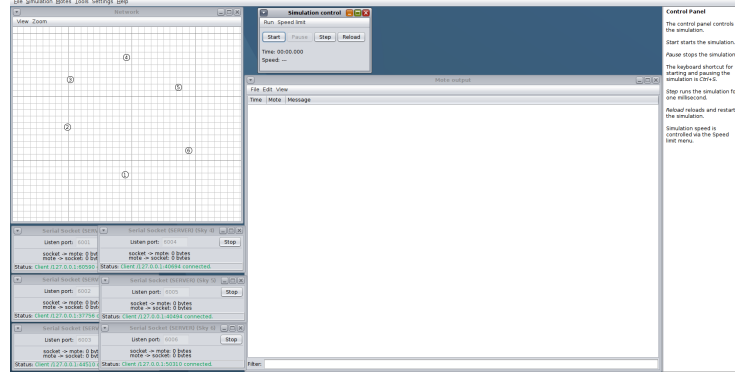

Figure 3: Start of Cooja simulation

The pattern we choose is to move Mote 1 in range of only Mote 2, Mote 3 and Mote 4 respectively, then in range of both Mote 5 and Mote 6 and end the simulation re-entering in range of Mote 2 (Figure 4).
With this order of operations we test interactions between 2 and 3 motes while also testing for re-entry in range of a mote we've been already in contact with.

Motes 1-2 connection   Motes 1-3 connection   Motes 1-4 connection

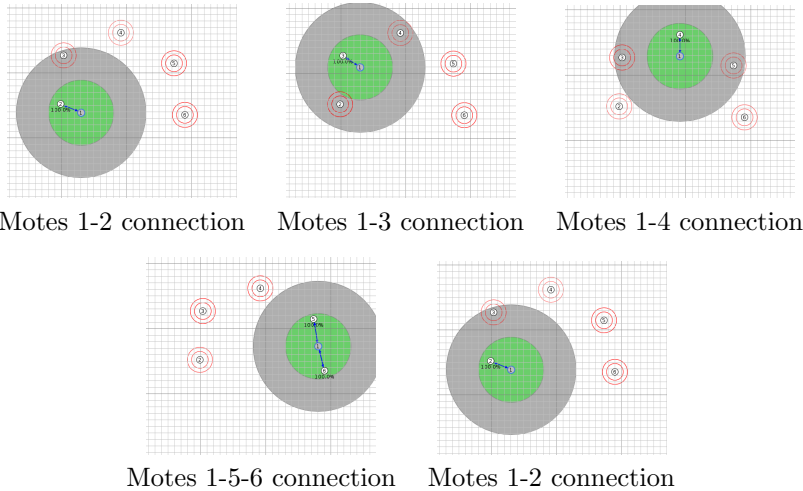Motes 1-5-6 connection   Motes 1-2 connection

Figure 4: Simulation pattern

For completeness, we also add a screenshot of the Cooja at the end of the simulation, with Mote 1 back at its starting point(Figure 5).
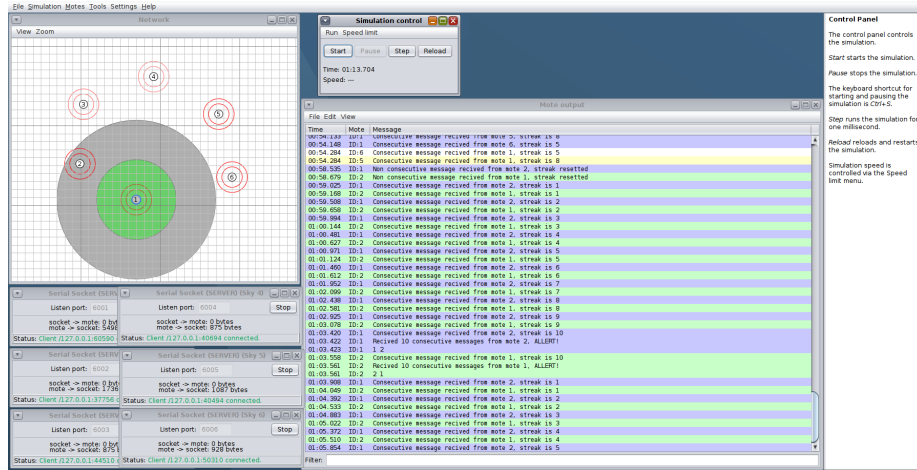The recap of this simulation was created by saving the mote output from Cooja and can be found in the file **log.txt**.

Figure 5: End of the simulation