LINFO2146 - Group Project

Chloé Dekeyser, Luca de Santos
https://github.com/ldesanto/embedded-project

May 21, 2023

Contents

1 Introduction

It has been requested to build a fictive management system that employs numerous wireless sensor nodes connected to motion sensors in order to track activity in specific areas. The counter within each sensor node increments anytime someone passes through its range, then will send their respective counts over an IEEE 802.15.4 multi-hop network utilizing wireless technology to an external server for analysis purposes. Nevertheless, it is important that we maintain efficiency and reliability on this network by employing an appropriate scheduling mechanism for these individual nodes which enable them govern themselves effectively. Therefore, implementing special coordinator devices that are connected directly to a border router in order to preventing signal interference between different nodes is necessary. This paper reports the implementation and discusses the solution proposed.

2 Node types

We defined 3 node types to model the sensor/coordinator nodes:

- **Undecided Node**: This is the neutral state a node starts with before becoming either a sensor node or a coordinator node.
- **Sensor Node**: Sensor nodes gather data and sends it to their parent "coordinator" node.

- **Coordinator Node**: Coordinator nodes receive data from their sensor children and forwards it to the edge router.
- **Border router**: Border router allows external communication between a server and the model, it assigns time slots to the several coordinator nodes, and then forwards the gathered data to the server, and vice versa.

3 Sensor and Coordinator Nodes Setup

When a node starts, it is initially in the "undecided" state and will broadcast a "new" message to all nearby nodes. When a node receives a "new" broadcast, it will reply with its type, either "sensor" or "coordinator" (If the node is still undecided or if the node is a coordinator node with *MAX_CHILDREN* children, it will ignore the message). When the new node receives a "sensor" or "coordinator" message, it will add the corresponding sender's address to either the *sensor_candidate* array or the *coord_candidate* array. It will also add the corresponding sender's RSSI (Received Signal Strength Indicator) to the corresponding array. After a set amount of time (the *GATHER_TIME* constant), it will choose a parent according to these rules:

- The new node checks if any of the potential parent candidates is a coordinator node. If there is only one coordinator node among the candidates, the coordinator node becomes the parent node of the new node.
- If there are multiple coordinator nodes as potential parent nodes, the new node uses the signal strength to select the parent node. In this case, the coordinator node with the strongest signal strength becomes the parent node of the new node.
- If there are only sensor nodes as potential parent nodes, the new node also uses the signal strength to select its parent node. The sensor node with the strongest signal strength becomes the parent node of the new node.
- If no parent candidates have been received, the new node sets itself as coordinator.

Once the new node has selected a parent node, it will send a "child" message to confirm the parent is aware of its new child.

When the parent node receives the "child" message, it will act according to its own type:

- If it is a coordinator node, it will add the child to the child array and reply with "parent".
- If it is a sensor node, it will reply with "no" to signal the potential child that it already has a parent and therefore cannot become a parent itself.
- If it is still undecided, it will set itself as coordinator and add the child to the child array and reply with "parent".

If the new node receives a "no" message back from its potential parent, it will restart its setup process up to a maximum of *MAX_RETRIES* time. If after *MAX_RETRIES* tries it still hasn't found a parent, it will set itself as a coordinator node.

It will then start either the **main_sensor** or the **main_coordinator** process.

4 Coordinator Node Main Process

Once the coordinator node has entered its main process, it will send a "coordinator" message to its parent (border router) to signal its presence. The border router will compute a new window attribution scheme and send it to the coordinator node (c.f. section 7). When it is its turn to forward sensor data, the coordinator node will cycle through the list of all his children. For each children, the coordinator will send a "sensor" message to the border router, followed by the sensor's address. It will then send a "poll" message to the sensor and forward every message from that child to the border router until a "done" message is received by the current child. It will then process to the next child. If a child does not reply during the span of a window, the coordinator will assume the child has disconnected and remove it from its list of children. If the coordinator doesn't have any child, it will still send a "ping" message to the border router to signal it's presence and prevent the edge router from assuming it has been disconnected.

5 Sensor Node Main Process

Once the sensor node has enter its main process, it will wait for a "poll" message from the coordinator node. When it receives a "poll" message, it will send his data in *DATA_LENGTH* packets. When it has sent all its data packets, it will send a "done" message to signal to the coordinator node it has finished sending data.

Each time a "poll" message is received, the *last_poll* variable is updated with the timestamp of the last poll message. If no poll message has been received in *MAX_WAIT* seconds, it will assume the parent has disconnected and will restart its setup process to either find a new coordinator node parent or become a coordinator node itself.

6 Border router node setup

The creation of a border router is required in order to be able to permit an external connection to the internal network, although communication management is considerably more flexible in the absence of RPL.

To communicate with the server via its serial interface when building a simulation, a Serial socket server has to be added to the mote, and the use of tunslip6 in order to make the connection properly. The serial_line protocol makes the exchage of messages possible between the mote and the server.

In order to leave communications unrestricted and tests simpler, the connectivity with the server has been segregated into code-defined functions.

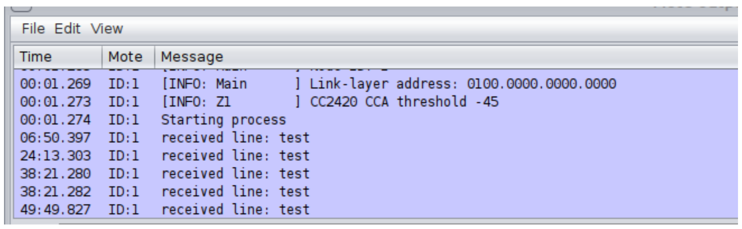
The `assign last counts` function enables the counters of each sensor to be assigned to their respective sensors. This information is sent to the server after each window, or the server can send a request to the border router to initiate the sending of information.

After each window interval, the border router only permits itself to relay the following data:

- The sensor's IDS
- The sensor's associated counters

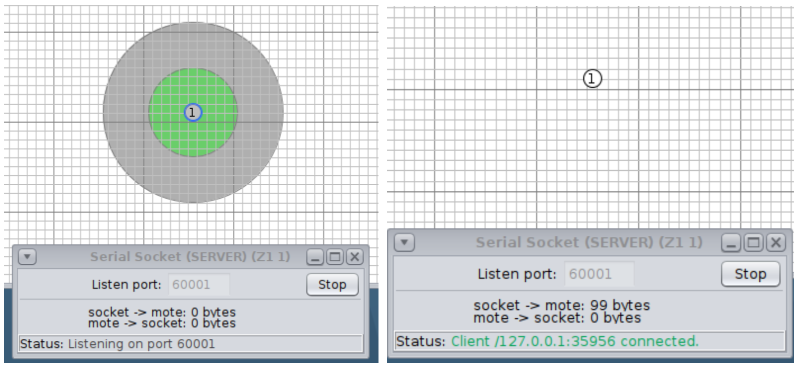
In order to verify the connection, we used the codes provided by the instructor and established the following procedures:

1. Define a mote z1 by incorporating a serial socket server, and then initiate the simulation: Define a monitoring port, which will be used later for the tunslip.
2. On a second terminal, navigate to the `contiki-ng/tools/serial-io` folder and execute the following command: `sudo./tunslip6 -a 127.0.0.0 60001 bbb::1/64` to launch the Serial Line Internet Protocol: There is now a connected client present.
3. Let's pause the simulation, activate the test server with the simulation port and the local address, and then restart the simulation: the mote has received messages from the server.



```
00:01.269  ID:1  [INFO: Main      ] Link-layer address: 0100.0000.0000.0000
00:01.273  ID:1  [INFO: Z1        ] CC2420 CCA threshold -45
00:01.274  ID:1  Starting process
06:50.397  ID:1  received line: test
24:13.303  ID:1  received line: test
38:21.280  ID:1  received line: test
38:21.282  ID:1  received line: test
49:49.827  ID:1  received line: test
```

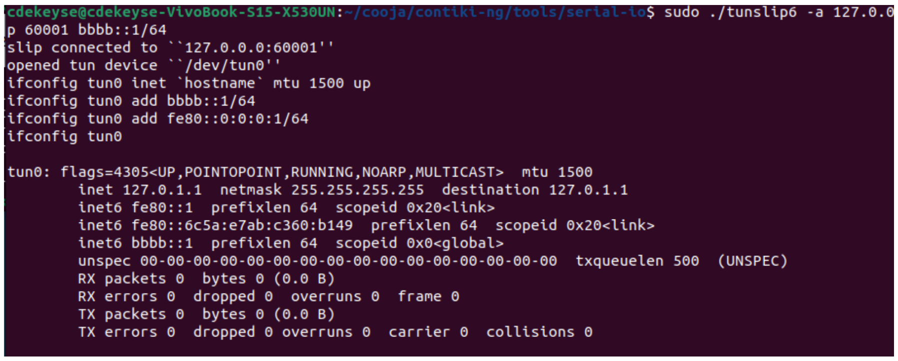
Figure 1: Serial_test.c messages



Serial Socket (SERVER) (Z1 1)
Listen port: 60001 Stop
socket -> mote: 0 bytes
mote -> socket: 0 bytes
Status: Listening on port 60001

Serial Socket (SERVER) (Z1 1)
Listen port: 60001 Stop
socket -> mote: 99 bytes
mote -> socket: 0 bytes
Status: Client /127.0.0.1:35956 connected.

Figure 2: Serial Socket Server in Cooja while connection



```
cdekeyse@cdekeyse-VivoBook-S15-X530UN:~/cooja/contiki-ng/tools/serial-io$ sudo ./tunslip6 -a 127.0.0
p 60001 bbbb::1/64
slip connected to ``127.0.0.0:60001''
opened tun device ``/dev/tun0''
ifconfig tun0 inet `hostname` mtu 1500 up
ifconfig tun0 add bbbb::1/64
ifconfig tun0 add fe80::0:0:0:1/64
ifconfig tun0

tun0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1500
        inet 127.0.1.1  netmask 255.255.255.255  destination 127.0.1.1
        inet6 fe80::1  prefixlen 64  scopeid 0x20<link>
        inet6 fe80::6c5a:e7ab:c360:b149  prefixlen 64  scopeid 0x20<link>
        inet6 bbbb::1  prefixlen 64  scopeid 0x0<global>
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 500  (UNSPEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 3: SLIP Protocol initiation

contiki-ng offers several functions to communicate across serial lines:

- https://github.com/contiki-ng/contiki-ng/blob/develop/os/dev/serial-line.h

7 Coordinator node and Border router communication

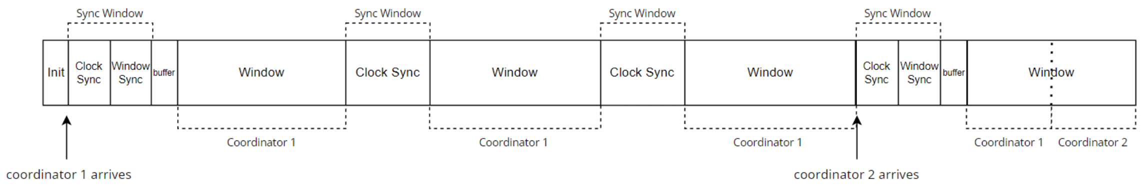


Figure 4: Border node time attribution

7.1 Time synchronization

A coordination protocol has been implemented to be robust against the arrival and departure of certain coordinators in order to maintain synchronization. the synchronization is held by the function `synchronization`, and works as follows :

- The border router broadcast ”clock_request” and then awaits for the accurate number of response, stored in the variable *number_of_coordinators*.
- Then the border router computes the the average clock time, including his, then broadcast a message that contains ”clock_set”, and then the new synchronised clock time that is taken into account, stored in the variable *average_clock*.

- Then the process of time slotting and window attribution begins.

This protocol is invoked when a new window starts, or if there is a protracted absence of a message during a time slot (absence of "ping" messages).

To establish a waiting list and prevent each new coordinator from disrupting the window's operation, lists have been created and read during the synchronization procedure, `pending_list` : They will also receive the clock requests, and then be put into the coordinator taken into acccount during the window, and therefore also listen to them afterwards.

In order to determine which time slot is currently active and which coordinator is responsible for each time slot, an index system has been implemented:

- A list comprising all the coordinators' addresses and a list containing all the active coordinators are contained in the variables `coordinator_list`.
- The indices of these lists correspond to the IDs assigned to coordinators by the border router, as well as the order in which the border router will listen to the time segments.
- The listened-to coordinator is contained in the variable `receiving_from`, with the index of the current listenned coordinator.

This makes it possible to find the departure of a coordinator, to delete it, re-evaluate the separation of the window as well as sending data to the server periodically.

7.2 Window attribution

This ability is held by the functions `timeslotting` and `sendTimeslots`, by the border router. Currently, the window is set to 2000 ticks, resulting in a refresh of the server interface every 2000 ticks, and is distributed evenly among the number of coordinators previously identified by their announcements. There may be performance anomalies, especially with unbalanced topologies, despite the calculation's simplicity and speed.

Then, 3 messages are sent to the coordinators:

- "window" is sent to the coordinators, announcing that the following messages will set up the window attribution process.
- The second message specifies the delay to be considered prior to beginning to send messages to the border router, as well as the delay itself, a "starting point". This timeout is arbitrarily set after several failed variable attempts.
- The third message specifies the length of the timeslot during which the coordinator will transmit its packets from the sensors before yielding to the coordinator of the next slot. Which is simply the $\frac{window\_size}{number\_of\_coordinators}$ operation.

7.3 Collecting counters

The coordinators communicate the counters of the sensors in a very specific order, which is as follows: In order to associate the most recent counter with the correct sensor address, a sentinel variable called *address_received* indicates where we are in the sequence.

- If the "sensor" message is received, an upcoming address communication is imminent, and the *last_address* variable is no longer applicable. The value of *address_received* is set to false.
- If a message is received and the *address_received* variable is set to true, then the counters of the sensor at the address "*last_address*" are being communicated and *last_count* is updated.
- If the variable *address_received* is false and a message is received, then this is likely the address received. Updates are made to *last_address* and *address_received.*

8 Observations and notes

- In order for the null_net communication to be done correctly between the border and the other motes, an initial broadcast had to be established in order for the future unicast to work.
- In the given makefile, the path to the "contiki-ng" folder has to be given in the "CONTIKI" variable.
- The border node must have the address 1,0

9 Annex

9.1 Messages

command	source	description
clock_request	border router	Asking for clock for synchronization
window	border router	Sending start time and window size of coordinator's timeslot
stop	any	flag to stop the border router listenning process
new	unassigned node	the sensor has come online
sensor	sensor	the node announces it is a sensor
child	sensor	the sensor announces to its parent it is its child
done	sensor	the sensor has finished sending its data to its parent
coordinator	coordinator	the node announces it is a coordinator
parent	coordinator	the coordinator accepts the sensor as its child
no	coordinator	the coordinator rejects the sensor as its child
poll	coordinator	the coordinator is polling the sensor