



UNIVERSITÀ DI PISA

Laurea Magistrale (MSc) in Artificial Intelligence and Data Engineering

Project

Internet of Things

Smart Supermarket

*Enrico Nello
Giacomo Pacini*

A.Y. 2020/21

INDEX

Introduction	1
System Architecture	2
Smart Shelves	2
weight sensor	2
price display	3
Control Logic	4
Smart Fridge	4
fridge temperature sensor	4
Control Logic	6
Collector	6
CoAP registration server	8
Deployment	10
Data Storing	10
Implementation Choices	12

1 Introduction

Today, technology is proliferating in nearly every industry, including the retail industry.

Technology has revolutionized both the way people shop, and the way businesses market their products.

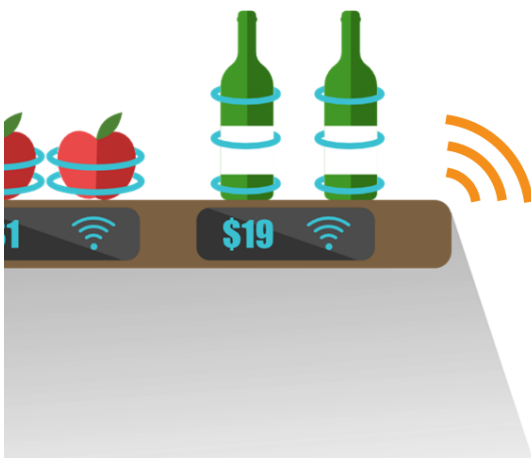
Smart Supermarket is an automation application for smart management of retail shops in the food industry.

In this project we developed two devices for transforming a common Supermarket into a Smart Supermarket:

- Smart Shelves
- Smart Fridges

In brief, **smart shelves** are composed of two different devices: a weight sensor and a price display.

Shelves are designed to automatically keep track of inventory in any retail establishment (in our case, supermarkets).



With smart shelves, supermarket owners can collect real-time data about their products.

Infact, the shelves are able to gather information about which products have or haven't been bought, in which period of time.



Business owners are therefore more informed about products selling ratios / buying trends. They now know what they should offer their customers and how they should supply their stores.



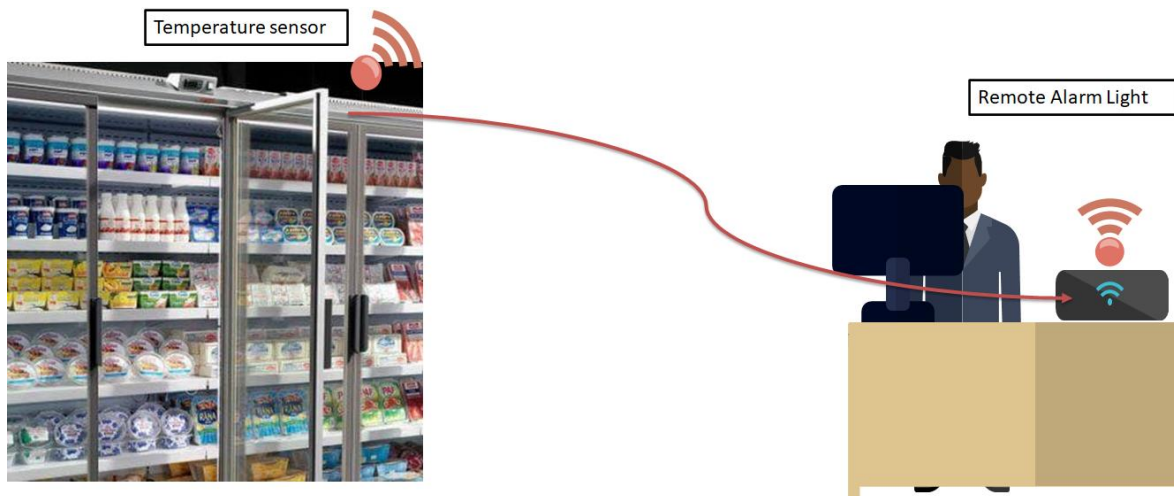
Smart shelves can optimize in-store operations. Sensors can alert employees to things like low inventory, allowing them to restock/refill shelves right away.



Smart shelves can schedule coupons, discounts and more, thanks to the statistics coming from the selling ratios.

Smart shelves can get this done automatically, so personnel can pay better attention to customer needs. This helps make team management easier because they are going to have more freedom and time to complete other tasks.

We also developed refrigerator temperature sensors to increase energy saving and avoid long periods of opened doors caused by peoples' negligence.



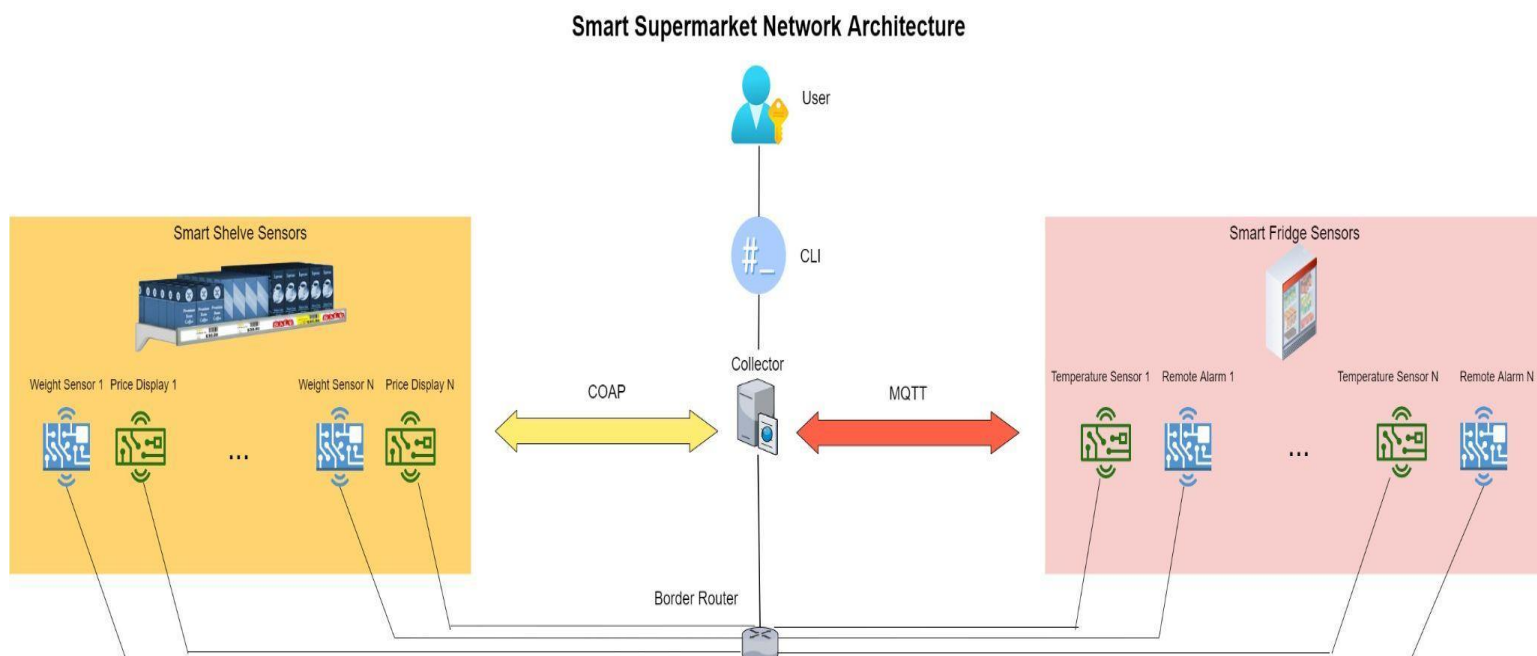
Smart Fridge is composed of two devices: a temperature sensor and a remote light alarm.

2 System Architecture

We implemented the nodes' firmware using Contiki-NG as the operating system. The physical nodes that we used to deploy our system on are NRF52840.

Our project aims to make a Supermarket «Smart»:

1. Smart Shelves [CoAP]:
 - a. Sensors: Weight
 - b. Actuators: Price Displays
2. Smart Fridges [MQTT]:
 - a. Sensors: Temperature
 - b. Actuators: Remote Alarm



3 Smart Shelves

3.1 Implementation

As already mentioned, the smart shelf is made of two different devices, coupled by the collector (which states the binding logic).

For the **weight sensor** and the **price display** nodes kinds we used CoAP as the communication protocol for the application level.

3.2 weight sensor

3.2.1 Weight changes logic

Inside the variable `current_weight` we have the current (simulated) measured weight.

Periodically we simulate a weight misuration from the sensor by calling the function `measure_weight()` which decreases the `current_weight` by a random value.

If the `current_weight` goes under a certain threshold, there is a call to `refill_shelf()`.

`refill_shelf()` is a function that resets `current_weight` to `MAX_VALUE` and that updates the `last_refill_ts` to `clock_seconds()`.

3.2.2 Exposed resources

The weight sensor exposes two resources: “/weight” and “/refill”.

3.2.2.1 The “/weight” resource

It is an observable resource that implements only the GET method. It is observable, so that every observer will be notified every time that the weight changes.

It answers to GET requests with a JSON-encoded object containing the fields:

- `weight`: an integer containing the value of `current_weight`
- `now`: an integer containing the value returned by `clock_seconds()`
- `id`: a string containing the `sensor_id` of the node

3.2.2.2 The “/refill” resource

It is an observable resource that implements the GET, POST and PUT methods.

It answers to GET requests with a JSON-encoded object containing the fields:

- `now`: an integer containing the value returned by `clock_seconds()`
- `last_refill_ts`: an integer containing the value of `last_refill_ts`
- `id`: a string containing the `sensor_id` of the node

The POST request handler of this endpoint implements the possibility to simulate a forced refill of the shelf.

In order to request a refill successfully, it is necessary to put in the body of the request a parameter “refill” set to true.

This endpoint handles the POST and PUT requests in the same way.

3.3 price display

The price display is an *actuator node* that has two global variables, `current_price` and `last_price_change` that are exposed in the only resource of this node: “/price”.

It is implemented also a function to update the `current_price` and another, `check_price_validity()`, which verifies that the price received is over a certain `MINIMUM_PRICE` threshold.

3.3.1 Exposed resources

3.3.1.1 The “/price” resource

It is an observable resource that implements the GET, POST and PUT methods.

It answers to GET requests with a JSON-encoded object containing the fields:

- `price`: an integer containing the value of `current_price`
- `now`: an integer containing the value returned by `clock_seconds()`
- `last_chg`: an integer containing the value of `last_price_change`
- `id`: a string containing the `sensor_id` of the node

The POST request handler of this endpoint implements the possibility to update the `current_price` of the shelf.

In order to update the price successfully, it is necessary to put in the body of the request a parameter “new_price” set to the new desired price value.

This endpoint handles the POST and PUT requests in the same way.

3.4 Control Logic

The control logic executed on the collector for the shelf device handles the price variation for the shelves, and it is implemented by the `weight_changes_handler()` inside the **ScaleDevice** class object. This handler is executed every time that a new message is registered from the weight sensor node.

The Price Changes’ Rules:

- 1a. A product has a **good** selling ratio if `num_of_shelf_refills / hour` is more than 100
- 1b. A product has a **bad** selling ratio if `num_of_shelf_refills / hour` is less than 50
- 1c. A product has a **terrible** selling ratio if `num_of_shelf_refills / hour` is less than 10
- 2a. A product can *increase* its price until **30%** more than the initial price
- 2b. A product can *decrease* its price until **60%** less than the initial price
- 3a. Do not change the price of a product, if it has changed recently (if `price_obj.last_price_change` is in the last 1 minute)

4 Smart Fridge

4.1 Implementation

As already mentioned, the smart fridge is made of two different devices, coupled by the collector (which states the binding logic).

For the **fridge temperature sensor** and the **fridge alarm light** nodes we used MQTT as the communication protocol for the application level.

4.2 fridge temperature sensor

4.2.1 Temperature changes logic

Inside the variable `current_temperature` we have the current (simulated) measured temperature inside the fridge. The variable `desired_temperature` is user modifiable, and contains the desired temperature for the fridge cell. It will change the way in which the fridge compressor changes its states.

Periodically we simulate a temperature misuration from the sensor by calling the function `sense_temperature()` which modifies the `current_temperature` value by a specific logic: in particular it updates the `compressor_state` to ON or OFF according to the previous value of `current_temperature` and it changes the value of `current_temperature` depending on the `compressor_state` value and the returned value by the function `door_is_open()` which simulates the evolution of the fridge door’s state by using a `roll_dice()` function and some different probability values depending on the number of consecutive door state open detected (see the implementation for more details).

4.2.2 The publish topics

4.2.2.1 “/discovery”

Every MQTT node in our system publishes at least in one topic in order to be discoverable by the Collector: the “/discovery” topic.

The expected messages in this topic are JSON-encoded objects containing the fields:

- `id`: a string containing the `sensor_id` of the node
- `kind`: a string containing the kind of the node, in this case “`fridge_temp_sensor`”

4.2.2.2 “fridge/___ID___/temperature”

The fridge temperature sensor periodically publishes to this topic a JSON-encoded object containing the fields:

- `temperature`: an integer containing the `current_temperature` measured by the node
- `timestamp`: an integer containing the value returned by `clock_seconds()`
- `unit`: a string containing the “`celsius`”
- `desired_temp`: an integer containing the `desired_temperature` measured by the node

This topic is specific not only for this kind of node, but also for every instance of fridge temperature sensor. We made this choice because in this way the collector (and more in general, someone that wants to retrieve data from a fridge temperature sensor), can easily know which are the nodes currently connected by subscribing to the “/discovery” topic, and after can subscribe to the publish topic specific for the node in which it is interested. In that way it will not have to filter the traffic from the various nodes, it is not requested to publish every time the id of the node, and it is easy to bind a dedicated thread for each node.

4.2.3 The subscribe topics

4.2.3.1 “fridge/___ID___/desired_temp”

The node expects to find in the messages published in this topic the new value desired to set for `desired_temp`.

4.3 Fridge Alarm Light

The fridge alarm light is an *actuator node* that has a global variable, `alarm_state` whose value can be modified by the collector. According to that value, the leds on the node will blink or stay off.

4.3.1 The publish topics

4.3.1.1 “/discovery”

In the same way as the other nodes, the published messages in this topic are JSON-encoded objects containing the fields:

- `id`: a string containing the `sensor_id` of the node
- `kind`: a string containing the kind of the node, in this case “`fridge_temp_sensor`”

4.3.1.2 “alarm/___ID___/state”

The fridge alarm periodically publishes to this topic a JSON-encoded object containing the fields:

- `alarm_state`: a string containing the current `alarm_state` of the node
- `timestamp`: an integer containing the value returned by `clock_seconds()`

4.3.2 The subscribe topics

4.3.2.1 “alarm/___ID___/actuator-cmd

The node expects to find in the messages published in this topic the new desired state to set for `alarm_state`.

4.4 Control Logic

The control logic executed on the collector for the smart fridge handles the state of the alarm depending on the temperature variation measured for the fridges, and it is implemented by the `check_temperature_threshold_routine()` inside the **FridgeTempSensor** class object. This handler is executed every time that a new message is registered from the fridge temperature sensor node.

5 Collector

The Collector, written in Python, is in charge of accepting connection of CoAP devices and receiving updates from the MQTT Broker, and storing all the sensor-generated data in a MySQL database.

Clients can interact with nodes through the Collector by sending commands and receiving various information.

5.1 Command User Interface

We provide the user with a CLI with the following commands available:

list:

lists all connected nodes, their IDs and kinds

```
paciollen@smart_supermarket:collector$ list
list of all the connected nodes
NodeID          Node Kind
070007          fridge_alarm_light
060006          fridge_temp_sensor
080008          price_display
020002          shelf_scale
```

list KIND:

if KIND is a recognised kind, lists all the nodes of that KIND

list-kinds:

lists all existing nodes' kind

list-couples-kinds:

lists all existing couples' kinds

list-couples:

lists all the couples of every couple kind

```
paciollen@smart_supermarket:collector$ list-couples
list of all the couples:

ScaleDevice      PriceDisplay:
020002           080008

FridgeTempSensor FridgeAlarmLight:
060006           070007
```

list-couples --kind KIND:

lists all the couples of the specified KIND, if KIND is a valid COUPLE-KIND

list-couples --spare:

lists all the PriceDisplay / ScaleDevice / FridgeTempSensor / FridgeAlarmLight spare nodes

list-couples --spare KIND:

lists all the spare nodes of specific kind

show-price ID:

if ID is the ID of a PriceDisplay, returns the current_price shown by that node

show-prices:

returns the current_price for each PriceDisplay node

set-price ID value:

if ID is the ID of a PriceDisplay, set a new price for that node

temp-info ID:

if ID is the ID of a FridgeTemperatureSensor, returns current temperature and desired temperature of that sensor

temp-infos:

returns current temperature and desired temperature for each connected FridgeTemperatureSensor

set-desired-temp ID value:

if ID is the ID of a FridgeTemperatureSensor, set a new desired temperature for that node

get-fridge-alarm-states:

returns current alarm state and last state change for each connected FridgeAlarmLight

get-fridge-alarm-state ID:

if ID is the ID of a FridgeAlarmLight, returns current alarm state and last state change for that FridgeAlarmLight

set-fridge-alarm-state ID X:

if ID is the ID of a FridgeAlarmLight and if X is one of ["ON", "OFF"] sets alarm state for that FridgeAlarmLight

shelf-info ID:

if ID is the ID of a ShelfScaleDevice, returns current weight, number of refills and last refill timestamp of that sensor

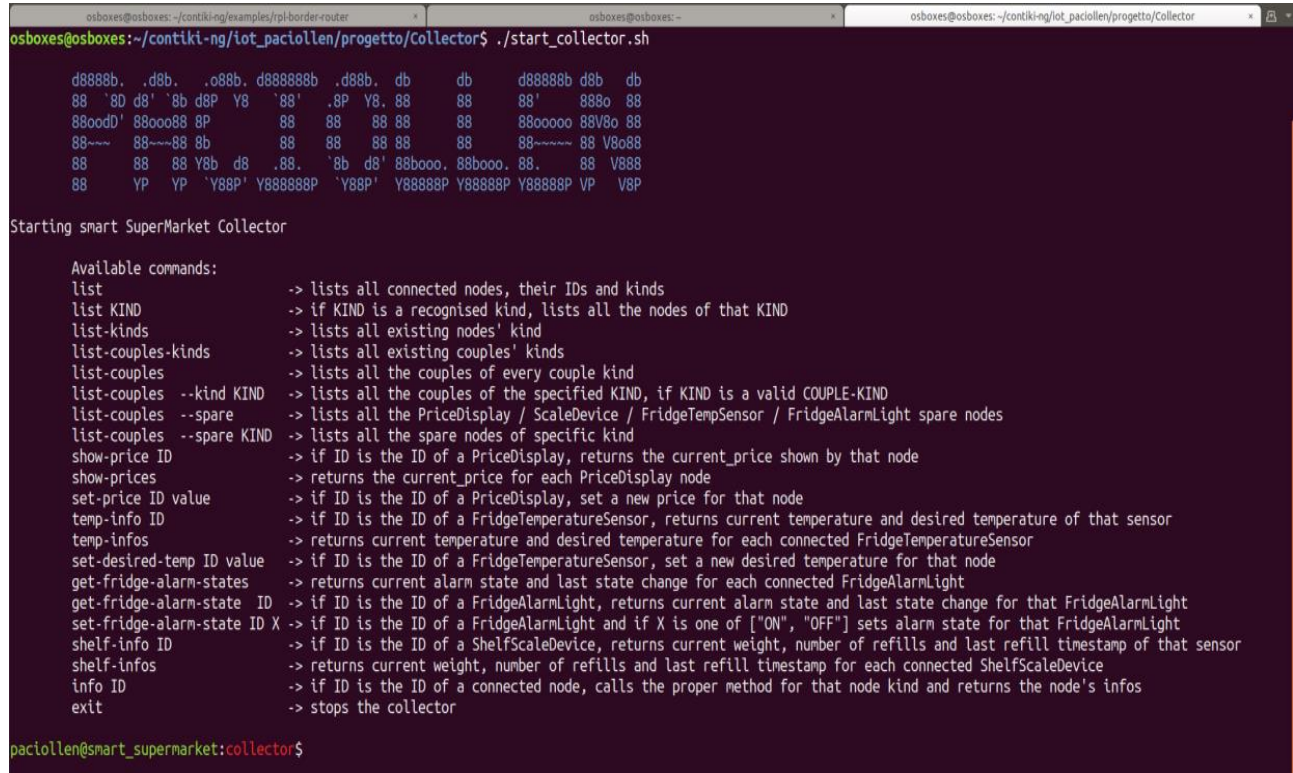
shelf-infos:

returns current weight, number of refills and last refill timestamp for each connected ShelfScaleDevice

info ID:

if ID is the ID of a connected node, calls the proper method for that node kind and returns the node's infos

Here is a picture showing the CLI during a demonstration:



```
osboxes@osboxes:~/contiki-ng/examples/rpl-border-router$ ./start_collector.sh
osboxes@osboxes:~/contiki-ng/iot_paciollen/progetto/Collector$ ./start_collector.sh

d8888b. .d8b. .o88b. d888888b .d88b. db db d88888b d8b db
88 '80 d8' '8b d8P Y8 '88' .8P Y8. 88 88 88' 888o 88
88oodD' 88ooo88 8P 88 88 88 88 88 88oooo 88V8o 88
88~ 88~ 88 8b 88 88 88 88 88 88~ 88 V8o88
88 88 88 Y8b d8 .88. '8b d8' 88booo. 88booo. 88. 88 V888
88 YP YP 'Y88P' Y888888P 'Y88P' Y888888P Y888888P VP V8P

Starting smart SuperMarket Collector

Available commands:
list -> lists all connected nodes, their IDs and kinds
list KIND -> if KIND is a recognised kind, lists all the nodes of that KIND
list-kinds -> lists all existing nodes' kind
list-couples-kinds -> lists all existing couples' kinds
list-couples -> lists all the couples of every couple kind
list-couples --kind KIND -> lists all the couples of the specified KIND, if KIND is a valid COUPLE-KIND
list-couples --spare -> lists all the PriceDisplay / ScaleDevice / FridgeTempSensor / FridgeAlarmLight spare nodes
list-couples --spare KIND -> lists all the spare nodes of specific kind
show-price ID -> if ID is the ID of a PriceDisplay, returns the current_price shown by that node
show-prices -> returns the current_price for each PriceDisplay node
set-price ID value -> if ID is the ID of a PriceDisplay, set a new price for that node
temp-info ID -> if ID is the ID of a FridgeTemperatureSensor, returns current temperature and desired temperature of that sensor
temp-infos -> returns current temperature and desired temperature for each connected FridgeTemperatureSensor
set-desired-temp ID value -> if ID is the ID of a FridgeTemperatureSensor, set a new desired temperature for that node
get-fridge-alarm-states -> returns current alarm state and last state change for each connected FridgeAlarmLight
get-fridge-alarm-state ID -> if ID is the ID of a FridgeAlarmLight, returns current alarm state and last state change for that FridgeAlarmLight
set-fridge-alarm-state ID X -> if ID is the ID of a FridgeAlarmLight and if X is one of ["ON", "OFF"] sets alarm state for that FridgeAlarmLight
shelf-info ID -> if ID is the ID of a ShelfScaleDevice, returns current weight, number of refills and last refill timestamp of that sensor
shelf-infos -> returns current weight, number of refills and last refill timestamp for each connected ShelfScaleDevice
info ID -> if ID is the ID of a connected node, calls the proper method for that node kind and returns the node's infos
exit -> stops the collector

paciollen@smart_supermarket:collector$
```

5.2 Nodes' registration

When a new node connects to the network, it tries to connect to the collector, so that the list of the connected nodes is always up to date.

Depending on the nodes' kind of connection (CoAP or MQTT) it is implemented a different mechanism both by the side of the node and by the side of the Collector.

5.2.1 CoAP registration server

The collector implements a CoAP server (using the CoAPthon library) that handles new registration requests and keep-alive messages from the nodes.

In particular, it implements a resource whose path is **“/registration”** that handles GET and POST requests.

A POST request is expected from a still not registered node that wants to connect to the Collector.

In the body of the request the collector expects a string corresponding to the `node_kind`; the collector will check that the given `node_kind` exists, then will check if the source IP address is already connected, and will eventually register the new node.

The GET request handler will only check if the source IP address is already registered and, in that case, will update the `last_seen` attribute for that node.

5.2.2 MQTT Discoverer

During the start-up phase the Collector subscribes only to one particular topic: **“/discovery”**.

The messages that the collector expects to find in this topic are JSON-encoded objects containing the fields:

- `id`: a string containing the `sensor_id` of the node
- `kind`: a string containing the kind of the node

Every MQTT node connected to our system is expected to periodically publish a message on this topic, in order to let the collector know that the node is still connected or, if it is a newly connected node, in order to let it know that a new node joined the network.

5.3 Disconnection handling

As already mentioned above, the collector keeps information about the `last_seen` time for each node, in order to keep an updated list of the effectively connected nodes.

In particular, each object, representing a node connection in our collector, extends a particular class, the **Node** class, which implements some methods to keep track of the `last_seen` timestamp for each node.

The constructor of the Node class creates a thread scheduled with a timer whose initial value is `CHECK_PRESENCE_INTERVAL` and whose target function to execute is the `check_presence()` function, which, in the case of a CoAP Node, will try to make a GET request to the IP of the node, and which, in the case of a MQTT Node, will directly mark the node as disconnected.

Every time that `update_last_seen()` is called, the Timer of the Node connection checker thread is restarted.

```
2021-09-09 17:19:13,058 - Thread-11 - Node_Class - DEBUG - Checking if the node 050005 | kind: price_display is still connected
2021-09-09 17:19:23,141 - Thread-11 - Node_Class - DEBUG - cannot connect to the node! going to delete this instance
2021-09-09 17:19:23,141 - Thread-11 - COAPModule - DEBUG - going to delete node 050005 | kind = price_display
2021-09-09 17:19:23,141 - Thread-11 - COAPModule - DEBUG - PriceDisplay id 050005 beeing deallocated!
2021-09-09 17:19:23,185 - Thread-11 - Node_Class - DEBUG - deleted connection checker thread for node 050005
2021-09-09 17:19:23,186 - Thread-11 - COAPModule - DEBUG - just deleted node 050005 | kind = price_display
2021-09-09 17:19:23,186 - Dummy-54 - COAPModule - DEBUG - PriceDisplay id 050005 beeing deallocated!
2021-09-09 17:19:25,017 - Thread-39 - Node_Class - DEBUG - Checking if the node 020002 | kind: shelf_scale is still connected
2021-09-09 17:19:30,819 - Thread-51 - Node_Class - DEBUG - Checking if the node 040004 | kind: price_display is still connected
2021-09-09 17:19:31,029 - Thread-52 - Node_Class - DEBUG - Checking if the node 030003 | kind: shelf_scale is still connected
2021-09-09 17:19:40,850 - Thread-51 - Node_Class - DEBUG - cannot connect to the node! going to delete this instance
2021-09-09 17:19:40,852 - Thread-51 - COAPModule - DEBUG - going to delete node 040004 | kind = price_display
2021-09-09 17:19:40,852 - Thread-51 - COAPModule - DEBUG - PriceDisplay id 040004 beeing deallocated!
2021-09-09 17:19:40,935 - Thread-51 - Node_Class - DEBUG - deleted connection checker thread for node 040004
2021-09-09 17:19:40,936 - Thread-51 - COAPModule - DEBUG - just deleted node 040004 | kind = price_display
2021-09-09 17:19:40,937 - Dummy-59 - COAPModule - DEBUG - PriceDisplay id 040004 beeing deallocated!
2021-09-09 17:19:45,207 - Thread-39 - COAPModule - DEBUG - going to delete node 020002 | kind = shelf_scale
2021-09-09 17:19:45,208 - Thread-39 - COAPModule - DEBUG - ScaledDevice id 020002 beeing deallocated!
2021-09-09 17:19:45,231 - Thread-39 - Node_Class - DEBUG - deleted connection checker thread for node 020002
2021-09-09 17:19:45,232 - Thread-39 - COAPModule - DEBUG - just deleted node 020002 | kind = shelf_scale
2021-09-09 17:19:45,232 - Dummy-61 - COAPModule - DEBUG - ScaledDevice id 020002 beeing deallocated!
2021-09-09 17:19:51,081 - Thread-52 - COAPModule - DEBUG - going to delete node 030003 | kind = shelf_scale
2021-09-09 17:19:51,082 - Thread-52 - COAPModule - DEBUG - ScaledDevice id 030003 beeing deallocated!
2021-09-09 17:19:51,155 - Thread-52 - Node_Class - DEBUG - deleted connection checker thread for node 030003
2021-09-09 17:19:51,157 - Thread-52 - COAPModule - DEBUG - just deleted node 030003 | kind = shelf_scale
2021-09-09 17:19:51,157 - Dummy-62 - COAPModule - DEBUG - ScaledDevice id 030003 beeing deallocated!
```


5.4 Visual Logger

We also developed a Visual Log that shows the output of all the messages from nodes:

```
2021-09-09 17:24:30,506 - Thread-180 - MQTTModule - INFO - [070007][FridgeAlarmLight.parse_state_response]: new node state set: b'{"alarm_state": "OFF", "timestamp": 315}'
2021-09-09 17:24:31,005 - Thread-182 - MQTTModule - WARNING - [change_state]: no change command 'OFF' for the node 070007
2021-09-09 17:24:31,012 - Thread-182 - MQTTModule - INFO - [060006][FridgeTempSensor.parse_state_response]: new node state set: b'{"temperature": 2.30, "timestamp": 315, "unit": "celsius", "desired_temp": 0.0}'
2021-09-09 17:24:31,529 - Thread-92 - COAPModule - INFO - [fd00::202:2:2:2][RefillSensor.parse_state_response]: new node state set: {"now": 360, "last_refill_ts": 360, "id": "020002"}
2021-09-09 17:24:45,497 - Thread-180 - MQTTModule - INFO - [070007][FridgeAlarmLight.parse_state_response]: new node state set: b'{"alarm_state": "OFF", "timestamp": 345}'
2021-09-09 17:24:46,073 - Thread-182 - MQTTModule - WARNING - [change_state]: no change command 'OFF' for the node 070007
2021-09-09 17:24:46,082 - Thread-182 - MQTTModule - INFO - [060006][FridgeTempSensor.parse_state_response]: new node state set: b'{"temperature": 2.0, "timestamp": 345, "unit": "celsius", "desired_temp": 0.0}'
2021-09-09 17:24:49,947 - Thread-175 - Node Class - DEBUG - Checking if the node 020002 | kind: shelf_scale is still connected
2021-09-09 17:24:50,136 - Thread-175 - COAPModule - DEBUG - [price_updater]: price_display nodeID = 080008 | selling_score = 45.45454545454545 | new_price = 18.45 | current_price = 20.5 | seconds_since_begin = 396
2021-09-09 17:24:50,231 - Thread-193 - COAPModule - INFO - [fd00::208:8:8:8][PriceDisplay.parse_state_response]: new node state set: {"price": 18.45, "now": 234, "last_chg": 234, "id": "080008"}
2021-09-09 17:24:50,241 - Thread-214 - COAPModule - DEBUG - set_new_values: received node response = {"price_updated": true, "last_change_ts": 234}
2021-09-09 17:24:50,409 - Thread-175 - COAPModule - INFO - [fd00::208:8:8:8][PriceDisplay.parse_state_response]: no change
2021-09-09 17:24:50,418 - Thread-175 - COAPModule - INFO - [fd00::202:2:2:2][WeightSensor.parse_state_response]: new node state set: {"weight": 1103, "now": 396, "id": "020002"}
2021-09-09 17:24:50,568 - Thread-175 - COAPModule - INFO - [fd00::202:2:2:2][RefillSensor.parse_state_response]: new node state set: {"now": 397, "last_refill_ts": 360, "id": "020002"}
2021-09-09 17:25:00,595 - Thread-180 - MQTTModule - INFO - [070007][FridgeAlarmLight.parse_state_response]: new node state set: b'{"alarm_state": "OFF", "timestamp": 375}'
2021-09-09 17:25:01,121 - Thread-182 - MQTTModule - WARNING - [change_state]: no change command 'OFF' for the node 070007
2021-09-09 17:25:01,128 - Thread-182 - MQTTModule - INFO - [060006][FridgeTempSensor.parse_state_response]: new node state set: b'{"temperature": 2.50, "timestamp": 375, "unit": "celsius", "desired_temp": 0.0}'
```

6 Deployment

For the demonstration of our project, we have implemented the two networks as follows:

6.1 MQTT Network

The MQTT network is deployed using the 4 real sensors from the testbed.

- 1 device deployed as Border Router
- 1 device deployed as Fridge Temperature Sensor
- 1 device deployed as Fridge Alarm Light

6.2 CoAP Network

On the CoAP network simulated with Cooja we decided to deploy the sensor and the actuator related to the Smart Shelf, which are respectively a weight sensor and a price display.

7 Data Storing

The Collector is in charge of collecting data from both MQTT and CoAP sensors and storing them in a MySQL database.

7.1 Data Structures

The *smart_supermarket* database is composed by 5 tables:

1) fridge_temperatures

ID	node_id	timestamp	node_ts_in_seconds	temperature	desired_temp
305	8b73f2	2021-09-04 13:15:49	525	0.5	0
306	8b73f2	2021-09-04 13:16:19	555	1	0
307	8b73f2	2021-09-04 13:16:49	585	1.5	0

2) fridge_alarm_light

ID	node_id	timestamp	node_ts_in_seconds	state	last_state_change
277	7c4622	2021-09-04 13:15:57	570	OFF	2021-09-04 13:15:57
278	7c4622	2021-09-04 13:16:27	600	OFF	2021-09-04 13:15:57
279	7c4622	2021-09-04 13:16:57	630	ON	2021-09-04 13:15:57

3) weight_sensor_state

ID	node_id	timestamp	current_weight
1	030003	2021-09-08 15:10:39	2000
2	030003	2021-09-08 15:10:39	248
3	030003	2021-09-08 15:10:57	2000
4	020002	2021-09-08 15:11:06	2000
5	020002	2021-09-08 15:11:06	1448

4) last_refill

ID	node_id	timestamp	last_refill_ts
1	030003	2021-09-08 15:10:39	2021-09-08 15:09:58
2	030003	2021-09-08 15:10:39	2021-09-08 15:09:57
3	030003	2021-09-08 15:10:57	2021-09-08 15:10:57

5) price_display_state

ID	node_id	timestamp	current_price	last_price_change
1	040004	2021-09-08 15:10:41	20.5	2021-09-08 15:09:57
2	040004	2021-09-08 15:10:41	20.5	2021-09-08 15:09:56
3	050005	2021-09-08 15:10:53	20.5	2021-09-08 15:09:57
4	040004	2021-09-08 15:10:57	22.55	2021-09-08 15:10:57
5	050005	2021-09-08 15:11:26	18.45	2021-09-08 15:11:26

7.2 Data Visualization

To visualize the data stored in the database we used the Grafana, which is a multi-platform web application for interactive visualization and analytics. It provides us charts, graphs, and alerts when connected to our MySQL database.

The picture below shows the data associated with a couple of Smart Shelves sensors (weight changes and the price follows):



The following graph instead shows the data associated with a Smart Fridge sensor (temperatures fluctuates above and beyond the *desired temperature* value and the state of the alarm switch consistently):



8 Implementation Choices

The Collector is written in Python. Regarding the data encoding, we decided to use JSON, because it's lightweight compared to other open data interchange options.

In addition to that, our application can't be considered critical, so we thought a more structured language like XML was not necessary. For single attribute messages we decided to use a simple text encoding, since we didn't need to give a specific structure.

Here are some other reason that brought us to choose JSON:

- It is less verbose, often more readable and it has a more compact style than XML. The lightweight approach of JSON has helped us a lot in simplifying the various parsing data phases.
- JSON is faster. In fact, the XML parsing process can take a long time. One reason for this problem is the DOM manipulation libraries that require more memory to handle large XML files. JSON uses less data overall, so we decided to reduce the overhead and increase the parsing speed.
- Readable: The JSON structure is straightforward and readable. You have an easier time mapping to domain objects, no matter what programming language you're working with.
- Using JSON, our structures match the data to parse: JSON uses a map data structure rather than XML's tree. In some situations, key/value pairs can be limited, but in this way we got a predictable and easy-to-understand data model.

JSON Limitations: The limitations in JSON actually end up being one of its biggest benefits. Even if XML is still used because it supports modeling more objects, JSON's limitations simplify the code, add predictability and increase readability.