Faculté de physique et ingénierie
Université de Strasbourg

Faculty of Physics and Engineering
University of Starsbourg

End of studies' project

Bachelor in Electronic Systems

Weather Station: Central

Carried out by:

EL KHAYDER Zakaria

OUFASKI Nouhaila

FRIES Léo

2022-2023

Made with LaTeX

Table of Contents

List of Figures

List of Listings

List of Tables

1 Introduction

1.1 Presentation

The title of our project is: "weather station: central", which means that we will have to centralize several meteorological data acquired thanks to different sensors to a single place in which the user can use them for personal or professional use. We know that it is important to know several information about the weather in everyday life such as the temperature or the humidity level, indeed for personal use, the user may wish to know if he is beautiful in its garden in order to enjoy the outdoors, while for professional use, a company could be led to know the temperature in different rooms of its premises without wanting to go there.

1.2 Motivation

We decided to chose this subject for our project because it seemed to us to be something interesting to set up, requiring an electronic part using for example arduino cards, as well as a mechanical part having to protect the components from the weather.

2 Nodes

Our weather station's primary components are the nodes, which gather weather data through various sensors. We have selected three specific sensors: temperature, humidity, and pressure.
To achieve this, we have employed two Arduino Nano devices situated in different locations. Each Arduino Nano is equipped with temperature, pressure, & humidity sensors.
Through LoRa communication, the collected data is transmitted to the central system, where it undergoes processing and storage for future utilization. Access to the accumulated data is facilitated through the website.
Keep in mind: Our project is mainly the central part of the weather station, so this section won't go into detail as the Central section.

2.1 Choice of parts & material

2.1.1 Arduino

For the Arduino, we don't really need anything fancy, a simple Arduino Nano will be more than sufficient.

MCU	**ATmega328P**
Architecture	AVR
Operating Voltage	5V
Input Voltage	7V - 12V
Clock Speed	16 MHz
Flash Memory	32 KB (2 KB used by bootloader)
SRAM	2 KB
EEPROM	1 KB
Digital IO Pins	22 (6 PWM)
Analog Input Pins	8

Table 1: Arduino Nano Specs

2.1.2 LoRa Module

We chose `Grove - Long Range 868MHz` because of its capabilities and ease of use.

- Using RFM95 module based on SX1276 LoRa®
- Working Voltage: 5V/3.3V
- 28mA(Avg) @+20dBm continuous transmit
- 8.4mA(Avg) @standby mode
- 20mA(Avg) @receive mode, BW-500kHz
- Working Temperature: -20 – 70℃
- Interface: Grove - UART (RX, TX, VCC, GND)
- Simple wire antenna or MHF Connector for external high gain antenna
- Working Frequency: 868MHz/433MHz
- +20dBm 100 mW Power Output Capability
- Size: 20*40mm
- Rate: 0.3kps $\tilde{5}$0kps
- Ready-to-go Arduino libraries
- Reserved MHF antenna connector

Table 2: Grove - Long Range 868MHz Specs & Features

The module already provides an Arduino Library to interface with it through Serial communication.

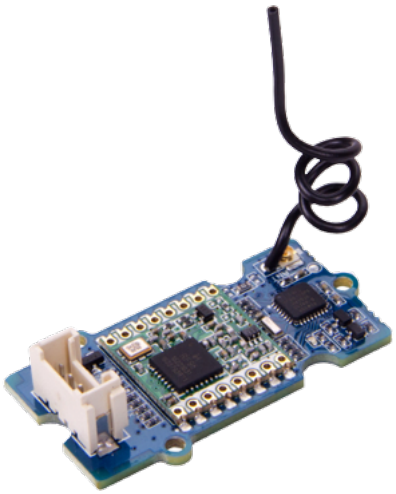


Figure 1: Grove - Long Range 868MHz (LoRa module)

2.1.3 Humidity and Temperature module

We chose `DHT11` as a temperature and humidity sensor. it is cheap and very easy to interface with requiring only one data line.

- Operating Voltage: 3.5V to 5.5V
- Operating current: 0.3mA (measuring), 60uA (standby)
- Output: Serial data
- Temperature Range: 0°C to 50°C
- Humidity Range: 20% to 90%
- Resolution: Temperature and Humidity both are 16-bit
- Accuracy: ±1°C and ±1%

Table 3: DHT11 Specs & Features

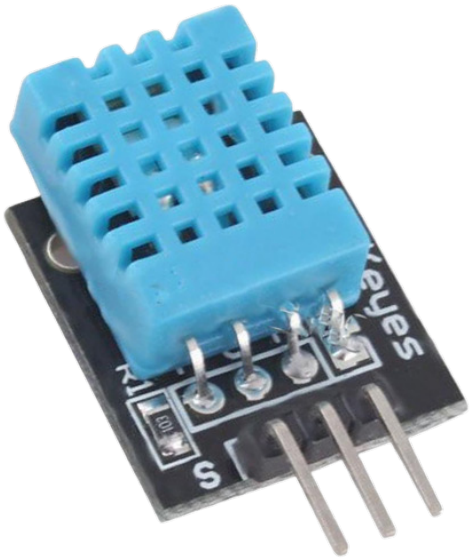


Figure 2: DHT 11

2.1.4 Pressure Sensor

We used `BMP388` as a pressure sensor, it also measures the temperature, but we won't be using it since we already have the `DHT11`.

Parameter	**Value**
Operation range (Pressure)	300...1250 hPa
Supply voltage (VDDIO)	1.2 V...3.6 V
Supply voltage (VDD)	1.65 V...3.6 V
Interface	I^2C and SPI
Average typical current consumption (1 Hz data rate)	3.4 µA @ 1Hz
Absolute accuracy pressure (typ.)	P=900...1100 hPa (T=25...40°C) & ±0.5 hPa
Relative accuracy pressure (typ.)	P=900...1100 hPa (T=25...40°C) & ±0.08 hPa
Noise in pressure (lowest bandwidth, highest resolution)	0.03 Pa
Temperature coefficient offset (-20°...65°C @ 700 hPa to 1100 hPa)	±0.75 Pa/K
Long-term stability (12 months)	±0.33 hPa
Solder drift	¡ ±1.0 hPa
Maximum sampling rate	200 Hz

Table 4: BMP388 Specs

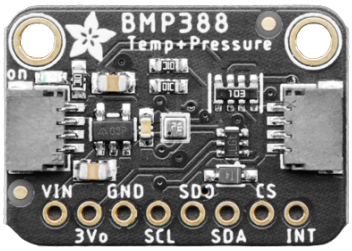


Figure 3: DHT 11

2.2 Electrical Schematic

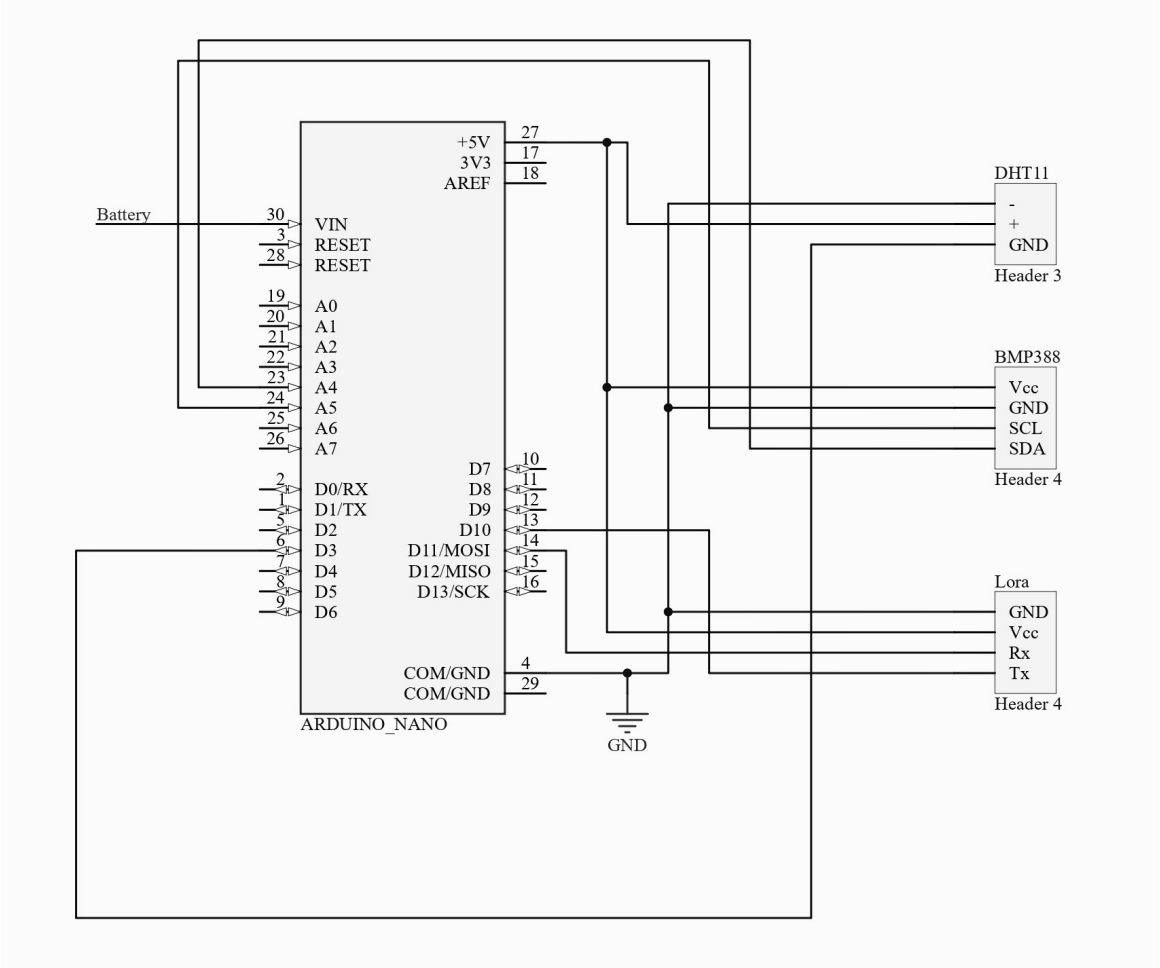


Figure 4: Node Schematic

2.3 Implementation

2.3.1 DHT11

The module is very simple to interface with using Arduino & its dedicated library. It is as simple as these for lines to initialize and fetch data.

```
DHT* _module = new DHT(_pin, DHT11);
_module->begin();
```

Listing 1: DHT11 initialization

```
float humidity = _module->readHumidity();
float temperature = _module->readTemperature();
```

Listing 2: DHT11 interfacing

2.3.2 BMP988

This module is a bit more verbose than its predecessor, but its library does more of the heavy lifting

```
DFRobot_BMP388_I2C* _module;
BMP388::BMP388()
{
    _module = new DFRobot_BMP388_I2C();
    _module->set_iic_addr(BMP3_I2C_ADDR_PRIM);
}

void BMP388::setup()
{
    Serial.println("Initializing BMP388 (Pressure Sensor)...");
    while (_module->begin() != BMP3_OK)
    {
        Serial.println("BMP388 initialization failed, Retrying...");
        delay(1000);
    }
    Serial.println("BMP388 initialized");
}

void BMP388::run()
{
```

```
    float pressure = _module->readPressure();

    float temperature = _module->readTemperature();
}
```

Listing 3: BMP388 initiation interfacing

2.3.3 LoRa

```
Lora::Lora(uint8_t Rx, uint8_t Tx)
{
    _serial = new SoftwareSerial(Rx, Tx);
    _rf95 = new RH_RF95<SoftwareSerial>(*_serial);
}

void Lora::begin(double frequency)
{
    Serial.println("Initializing RF95 (LoRa Module)...");

    while (!_rf95->init())
    {
        Serial.println("RF95 initialization failed, Retrying...");
        delay(1000);
    }

    if (!_rf95->setFrequency(frequency))
    {
        Serial.println("RF95 frequency out of range");
        while (1)
            ;
    }

    Serial.println("RF95 initialized");
}

String Lora::_joinPayloads(String *data, size_t len)
{
    String r;

    for (int i = 0; i < len; i++)
    {
        r += data[i] + '\n';
```

```
34      }
35
36      return r;
37  }
38
39  bool Lora::sendPayloads(String *data, size_t len)
40  {
41      String result = _joinPayloads(data, len);
42
43      Serial.print("Sending ");
44      Serial.print(len);
45      Serial.println(" payload(s)");
46      for (int i = 0; i < len; i++)
47      {
48          Serial.print("    ");
49          Serial.println(data[i]);
50      }
51
52      uint8_t bytesBuffer[result.length()];
53
54      result.toCharArray((char *)bytesBuffer, sizeof(bytesBuffer));
55
56      _rf95->send(bytesBuffer, sizeof(bytesBuffer));
57      _rf95->waitPacketSent();
58  }
```

3 Central

The Central is the brain and muscles of our system, I t lives in the *center* (fair enough), between the nodes and the user. It communicates with other nodes, gathers and processes data, hosts the server, and serves the web interface (HMI).

For this type of project, Raspberry-PI is highly recommended for such a task since it is basically a tiny computer, we can easily set up and host a Database and a server to handle Hypertext Transfer Protocol (HTTP) requests. Unfortunately, the only available Raspi in the faculty got stolen, and due to the global chip shortage, we couldn't order another one because they were often unavailable or, if we were lucky, unbelievably expensive (at least 150 €).

Because of the above, the only option we had left was to use an Arduino for the Central. The task was already hard enough, but building it on an Arduino made it x10 harder. Nevertheless, we decided to go ahead with it, challenge ourselves, and learn something new.

3.1 Choice of parts & material

3.1.1 Arduino

The Arduino ecosystem is a vast ocean. There are around 25 official Arduino models, making the choice of the proper Arduino harder. So before choosing one, we should determine what we need first:

- **Internet Connectivity**: Ethernet is acceptable, but WiFi is preferred. This module can be external.
- **ROM & SRAM**: The Central will have to execute some expensive processes, which makes RAM indispensable. ROM isn't as crucial as SRAM, we can always add external memory space if needed, which leads us to the next point.
- **SD Card Reader**: Sensors data should be retrievable. We can also use an external module.

After research, we decided to go with **Arduino MKR WiFi 1010**. It responds to all of our requirements.

- **Internet Connectivity**: Built-in Nina W102 uBlox module. Compatible with WiFi 802.11b/g/n & Dual-mode Bluetooth v4.2.
- **ROM & SRAM**: With 32KB of SRAM, we believe it is more than enough to handle our internal request and process nodes' communication. And with 256KB Flash memory (ROM), we can comfortably upload our sketch without compromises.
- **SD Card Reader**: Unfortunately, the MKR WiFi 1010 does not have an integrated SD Card Reader, but we can easily hook up an external module as stated before.

Attention: One thing to keep in our mind is that Arduino MKR WiFi 1010 works with 3.3V logic level, anything above will damage the board

Board	Name	Arduino® MKR WiFi 1010
	SKU	ABX00023
	Compatibility	MKR
Microcontroller	SAMD21 Cortex®-M0+ 32bit low power ARM MCU	
USB connector	Micro USB (USB-B)	
Pins	Built-in LED Pin	6
	Digital I/O Pins	8
	Analog Input Pins	7 (ADC 8/10/12 bit)
	Analog Output Pins	1 (DAC 10 bit)
	PMW Pins	13 (0 - 8, 10, 12, A3, A4)
	External interrupts	10 (0, 1, 4, 5, 6, 7, 8 ,9, A1, A2)
Connectivity	Bluetooth®	Nina W102 uBlox module
	Wi-Fi	Nina W102 uBlox module
	Secure element	ATECC508A
Communication	UART	Yes
	I2C	Yes
	SPI	Yes
Power	I/O Voltage	3.3V
	Input Voltage (nominal)	5-7V
	DC Current per I/O pin	7 mA
	Supported battery	Li-Po Single Cell, 3.7V, 1024mAh Minimum
	Battery connector	JST PH
Clock speed	Processor	48 MHz
	RTC	32.768 kHz
Memory	SAMD21G18A	256KB Flash, 32KB SRAM
	Nina W102 uBlox module	448 KB ROM, 520KB SRAM, 2MB Flash
Dimensions	Weight	32 g
	Width	25 mm
	Length	61.5 mm

Table 5: Arduino MKR WiFi 1010 Specs

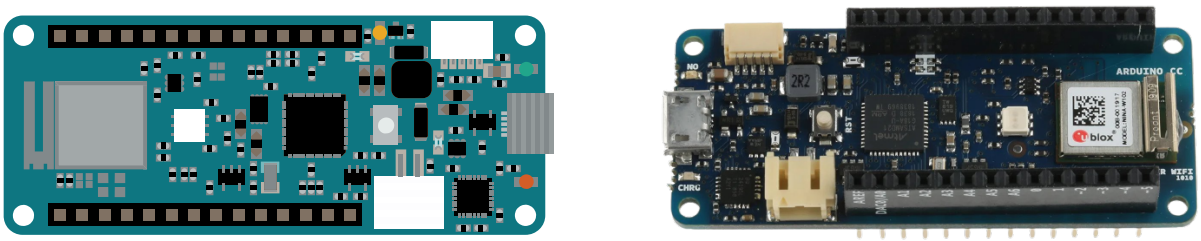


Figure 5: Arduino MKR WiFi 1010

3.1.2 SD Card Reader

This one is more straightforward, we chose Pmod MicroSD for the simple reason of it being already available in the Faculty's inventory. Fortunately, it itself works with 3.3V logic level, which makes it compatible with Arduino MKR WiFi 1010 that we chose. In addition, it has many other impressive features:

- Store and access large amounts of data from the host board
- No limitation on the file system or memory size of microSD card used, which is great for reasons we will discuss later
- 1-bit and 4-bit communication
- 12-pin Pmod port with **SPI interface**

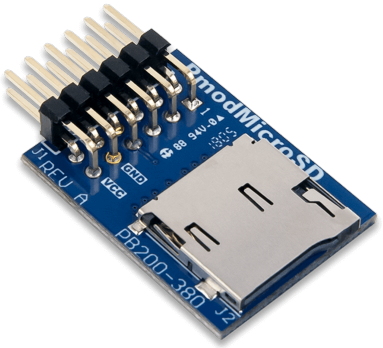


Figure 6: Pmod MicroSD Reader

3.1.3 Display

We will be hosting the web interface on Arduino, and we will need the IP address to access the website. We can always find it out from the Router webpanel, but it would be easier for us and the user to know it without the hassle of searching for it elsewhere, here is where comes the need for a Display, besides, we can use it also to show other information such as WiFi connection Status & LoRa signal for example.
We chose to use a 20x4 I^2C LCD Display.

- Requires minimal cable connections (Only 4: Power, Ground, SDA & SCL)
- Fully compatible with Arduino with numerous up-to-date libraries.
- Available in faculty inventory.

(a) LCD (b) I_2C Module*

Figure 7: LCD Display 20x4 with I_2C communication module

* Module usually comes pre-soldered to the LCD display

3.2 Architecture

Arduino wasn't really built to be a server really, so implementing one won't be the smoothest thing to do, no doubt we will be having too many moving parts that we need to build ourselves.

- **WiFi**: Handle WiFi connection & verify connection status regularly. It should also handle any errors while connecting or that can cause a disconnection from the Access Point (AP)
- **Storage**: Implement a way to easily fetch and write data to & from the SD card. It should act like a DB.
- **Clock**: We intend to log the data received from the nodes to keep a history of previous measurements, we will need to also keep track of the time when we received those data.
- **HTTP handler**: Keep an eye on new HTTP connections and handle them. We will refer to this part as "server" or "application" from now on.
- **LCD**: Handle LCD updates, which should be triggered when data changes.
- **LoRa Communication**

Keep in mind: Before starting to code a project with this many moving parts, we should think about having only one source of truth, meaning that we should have only one place from where we can get information about something, for example, WiFi status. This concept is also known as having a ***Single App Store*** or ***Central App State***.

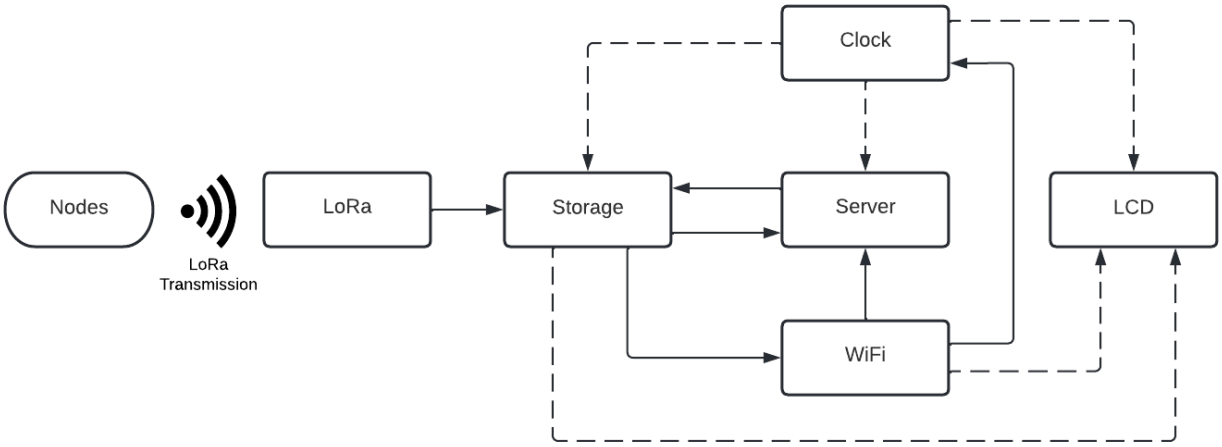


Figure 8: Central Architecture

Legend:

- A $\longrightarrow$ B: A affects B (Change its value).
- A - $\rightarrow$ B: B reads from A.

3.3 Implementation

3.3.1 WiFi

Thanks to using MKR WiFi 1010, initiating a WiFi connection is straightforward, however, we should keep an eye on the connection status every few minutes or so in case of connection loss.

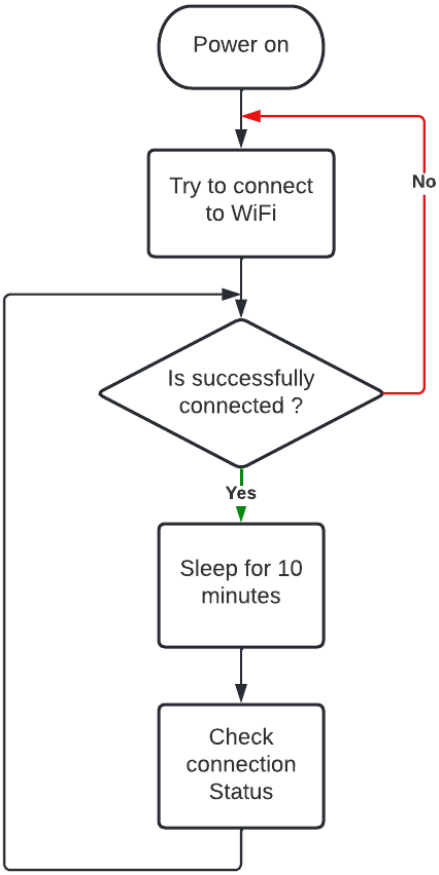


Figure 9: WiFi implementation flowchart

Listing 4: WiFi first connection implementation

```
#include <WiFiNINA.h>

void setup() {
    if (::WiFi.status() == WL_NO_MODULE) {
    //  ^^ main namespace scope
        errHalt(F("WiFi"), F("Module not found"));
    }
    String fv = ::WiFi.firmwareVersion();
    Serial.print(F("Wifi Module detected, firmware version: "));
    Serial.println(fv);
    connect();
}
```

Listing 5: WiFi connect implementation

```
void connect() {
    while (_status != WL_CONNECTED) {
        _status = ::WiFi.begin(_ssid, _password);
        delay(5000); // Wait up to 5s for connection to establish
    }
    local_ip = ::WiFi.localIP();
}
```

Listing 6: WiFi loop implementation

```
void loop() {
    status = ::WiFi.status();
    if (_status != WL_CONNECTED) {
        connect();
    }
    delay(10 * 60 * 1000); // 10min
}
```

This solution will work fine... until it won't! Once we will start adding other components to our systems like the storage or the server, it will brake our system because of the `delay` function.

The `delay` function is a blocking function, meaning, while it is executing, it wouldn't allow anything else to run until it finishes, so for our case, for 10min, the Arduino will be stuck in the delay function, and wasting precious time & processing power.

Before going any further, we should find a way around this limitation.

3.3.2 Poll

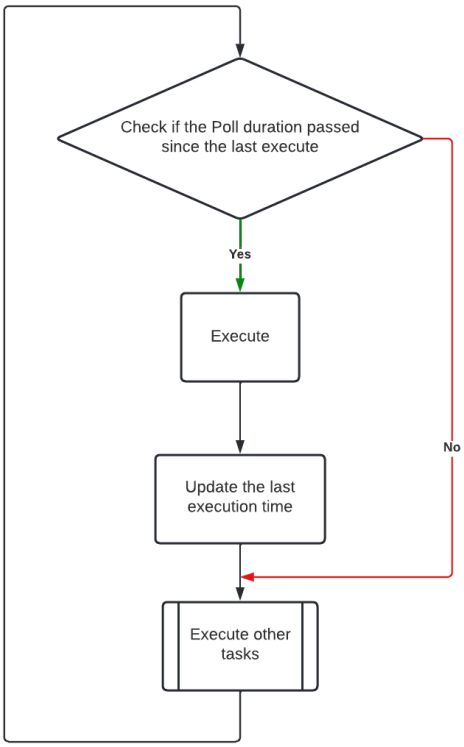


Figure 10: Poll implementation flowchart

We will be using this polling concept many times throughout our central implementation, so it is best to factor it into a C++ class.

We can use Arduino's `millis` function to get the number of milliseconds since the program started (Arduino power on).

Listing 7: Poll implementation

```
class Poll {
    public:
        uint32_t duration;

        Poll(uint32_t duration) : duration(duration) {};

        bool shouldExecute() {
            return millis() > _last_executed_at + duration;
        }

        void setExecuted() {
            _last_executed_at = millis();
        }
```

```
14
15      private:
16          uint32_t _last_executed_at = 0; // 32bit unsigned int
17  };
```

Now we can initiate an instance of our Poll class instead of a delay whenever we need to implement a recurring task.

Listing 8: Poll usage example

```
1   Poll _poll(60 * 1000); // 60_000ms = 1min
2   if(_poll.shouldExecute()) { <---------------------- <-
3       _poll.setExecuted(); // Update execution time     ^
4       // Code to execute                                ^
5   } // ----------------------------------------------> ^
```

3.3.3 Storage

The storage program is very straightforward, we should only initiate the SD module connection on the Arduino's startup, however, there are some limitation that we should be aware of that concerns the FAT32 filesystem, which is the only filesystem supported by the default Arduino SD library.

Filesystem	Max. Path Length	Max. Filename Length
(*) Btrfs	No limit defined	255 bytes
(*) ext2	No limit defined	255 bytes
(*) ext3	No limit defined	255 bytes
(*) ext4	No limit defined	255 bytes
(*) XFS	No limit defined	255 bytes
(*) ZFS	No limit defined	255 bytes
APFS	Unknown (**)	255 UTF-8 characters
FAT32	32,760 Unicode characters with each path component no more than 255 characters	8.3 (255 UCS-2 code units with VFAT LFNs)
exFAT	32,760 Unicode characters with each path component no more than 255 characters	255 UTF-16 characters
NTFS	32,767 Unicode characters with each path component (directory or filename) up to 255 characters long (MAX_PATH)	255 characters

Table 6: Path and Filename Length Limitations for different file systems

* In Unix environments, PATH_MAX with 4096 bytes and NAME_MAX with 255 bytes are very common limitations for applications including the Shell.

** Although not officially documented, when searching on the internet there is a limit with path names exceeding 1024 bytes. Users

report warnings in Finder, the Shell, or apps about this behavior.

As stated by the above table, FAT32 has a very strict limit on the file name, where it can only have 8 characters maximum for the name and 3 others for the file extension.
That already makes it hard for us since the default HTML file extension length is 4 characters (.html), we will also have a problem if we need to ever include an image file such as a JPEG (.jpeg) or something else, let alone the filenames which are limited to 8 characters maximum.

We could try to find a way around these limitations with some fancy tricks, but as the French proverb says: "Chercher midi à quatorze heures".

We can use a 3rd party Arduino library `SdFat.h` to do the heavy lifting for us. The usage is almost identical for the default Arduino one, except that for this library, we need to define a global variable to access the storage.

```
SdFat sd;

void setup() {
    if (!sd.begin(SdSpiConfig(SD_CS_PIN, DEDICATED_SPI, SD_SCK_MHZ(50)))) {
        // Throw an error and halt the program
        errHalt(F("SD card"), F("Initialisation failed!"));
    }
    Serial.println(F("SD Card initialized successfully!"));
}
```

Listing 9: SD card reader initialization

For now, that is all that we need for the storage. Each time we need to read a file from the SD card, we can use the already initiated `sd` variable.

3.3.4 Clock

We will use worldtimeapi.com's free API to get the current Unix time. This step should be executed only after already initializing the WiFi connection.
The API returns either a JSON or simple text response, we can get the JSON and parse it the same way we used to parse payloads in our nodes using `ArduinoJSON.h` library.
We can also add a timer to refresh the time every 24 hours for example. this step is not really needed since Arduino can already keep a relatively precise time, but it won't hurt to add this step.

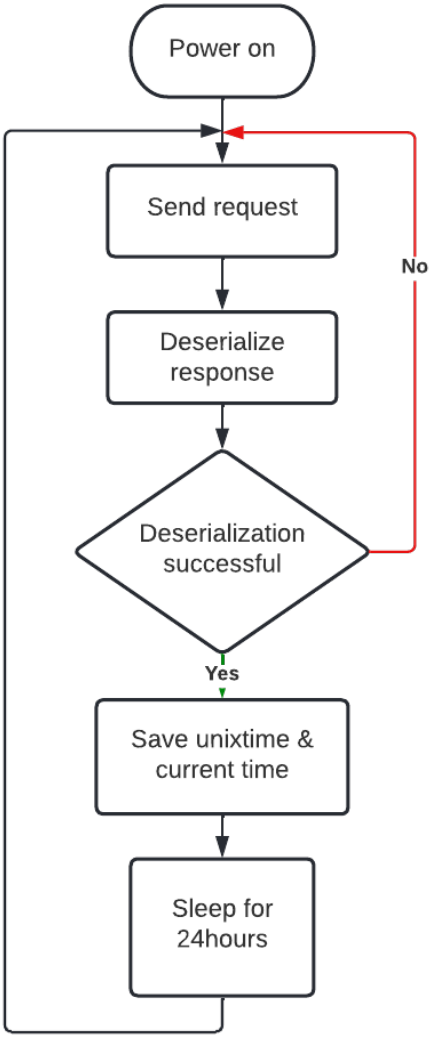


Figure 11: Clock implementation flowchart

Now getting the current Unix time at any moment is as simple as this simple equation

$$t_{now} = t_{UnixTime} + \frac{millis() - t_{FetchedAt}}{1000}$$

We should divide by 1000 to convert from milliseconds to seconds.

We can encapsulate this code in a single function for ease of use.

Listing 10: Clock time fetching implementation

```
uint32_t _unix_timestamp = 0;
const String _HOST_NAME = F("www.worldtimeapi.org");
const String _PATH_NAME = F("/api/timezone/Europe/Paris");
Poll _fetch_poll(24 * 60 * 60 * 1000); // 24 hours
WiFiClient _wifi = WiFiClient();
```

```
HttpClient _client = HttpClient(_wifi, _HOST_NAME);
uint32_t _fetched_at = 0;

void fetch() {
    DynamicJsonDocument json(1024);
    _client.get(_PATH_NAME);
    int statusCode = _client.responseStatusCode();
    if (statusCode < 200 || statusCode >= 300) return;
    String response = _client.responseBody();
    _client.stop();
    DeserializationError error = deserializeJson(json, response.c_str());
    if (error) return;
    _unix_timestamp = json["unixtime"];
     _fetched_at = millis();
    _fetch_poll.setExecuted();
}

uint32_t time()  {
    return _unix_timestamp + (millis() - _fetched_at) / 1000;
}
```

3.3.5 LoRa

We already implemented the LoRa module initialization code in the nodes section. However, we should implement two functions for receiving and parsing the payloads.

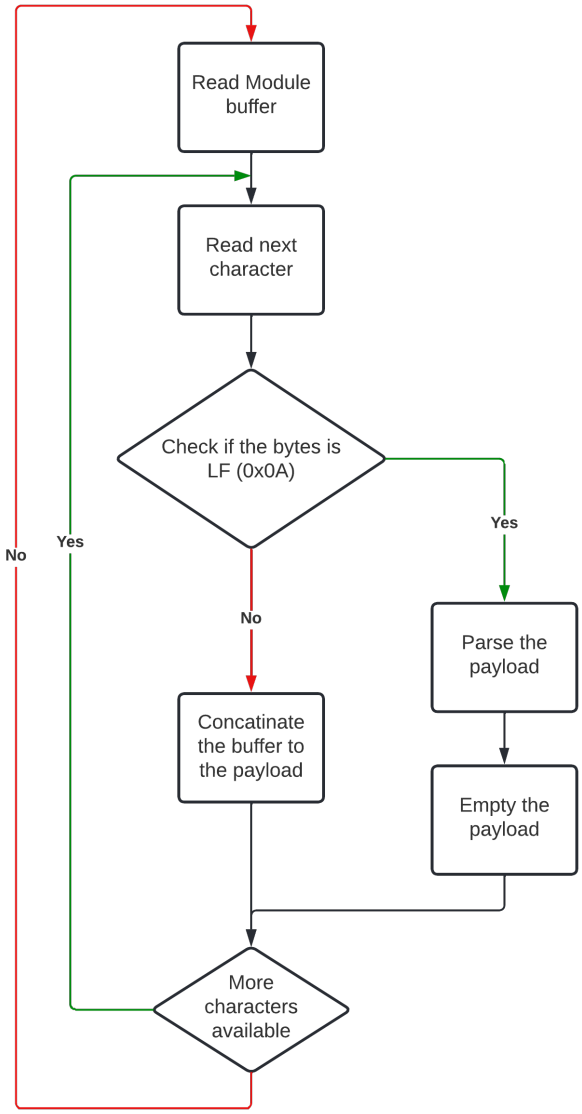


Figure 12: LoRa payload read flowchart

```
void loop() {
    if (rf95.available()) {
        char buf[RH_RF95_MAX_MESSAGE_LEN];
        uint8_t len = sizeof(buf); // Equal to RH_RF95_MAX_MESSAGE_LEN
        if (!rf95.recv((uint8_t *)buf, &len)) {
            Serial.println("LoRa receive failed");
            return;
        }
        for (size_t i = 0; i < len; i++) {
            if (buf[i] == '\n') {
                parsePayload();
                buffer = "";
```

```
            } else {
                buffer += buf[i];
            }
        }
    }
}
```

Payload parsing is straightforward, we should only deserialize the JSON data, and save it to the db file with the current Unix time as a Comma-seperated values (CSV) entry.

Listing 11: LoRa payload parse implementation

```
void parsePayload() {
    DynamicJsonDocument doc(512);
    DeserializationError error = deserializeJson(doc, buffer);
    if (error) return;
    const char *node = doc["node"];
    const char *measure = doc["measure"];
    double value = doc["value"];
    File db = Storage::sd.open("db.csv", FILE_WRITE);
    if (!db)
        errHalt(F("SD Card"), F("DB file open failed"))
    db.print(node);
    db.print(",");
    db.print(measure);
    db.print(",");
    db.print(value);
    db.print(",");
    db.println(Clock::time());
    db.close();
}
```

3.3.6 Server

Now that we implemented everything, we can jump on to implementing the actual HTTP server.
For the sake of not having to parse the verbose HTTP protocol request ourselves, we will be using an external library that will do the heavy lifting for us: `aWOT.h`.
The library is easy to use, we should just initialize it once by registering allowed routes and their handlers on startup and then we can pass HTTPServer requests to it to process.

Listing 12: Server

```
void setup() {
    _server.begin();
    _app.get("/api/db", &RequestHandlers::db);
    _app.get(&RequestHandlers::staticFiles);
}

void loop() {
    WiFiClient client = _server.available();
    if (client.available()) {
        _app.process(&client);
        client.stop(); //  Force disconnect to free the queue
    }
}
```

Listing 13: Server setup and loop implementations

We register three handlers:

- **/api/db**: This route will return the db file content
- **Static files (website)**: Should look through the storage, if the file exists it should return its content

```
namespace RequestHandlers {
    void staticFiles(Request &req, Response &res) {
        if (strlen(req.path()) == 1) {
            return file(req, res, "/index.html");
        }
        return file(req, res, req.path());
    }

    void db(Request &req, Response &res) {
        File file = Storage::sd.open("db.csv");
        if (!file)
            return;
        while (file.available())
            res.write(file.read());
        file.close();
        res.end();
    }

    void file(Request &req, Response &res, const char *filename) {
        String filepath = Server::STATIC_FOLDER + filename;
```

```
        if (!Storage::sd.exists(filepath)) {
            res.sendStatus(404);
            return;
        }
        File file = Storage::sd.open(filepath);
        if (file.isDirectory()) {
             res.sendStatus(403);
            return;
        }
        const String *contentType = Server::fileContentType(&filepath);
        if (contentType != NULL)
            res.set("Content-Type", contentType->c_str());
        while (file.available())
            res.write(file.read());
        res.end();
        file.close();
        return;
    }
}
```

Listing 14: Server requests handlers implementation

3.3.7 Main .ino file

After implementing all the above, we need to call each of these functions in our main Arduino .ino file.

```
#include "src/WiFi.h"
#include "src/Server.h"
#include "src/Clock.h"
#include "src/Storage.h"
#include "src/LCD.h"
#include "src/Lora.h"

const char *WiFi::_ssid = "TestNetwork";
const char *WiFi::_password = "00000000";

void setup()
{
    Serial.begin(115200);
    LCD::setup();
    Storage::setup();
    Lora::setup();
```

```
17      WiFi::setup();
18      Server::setup();
19      Clock::fetch();
20  }
21
22  void loop() {
23      WiFi::loop();
24      Server::loop();
25      Clock::loop();
26      Lora::loop();
27  }
```

Listing 15: Main .ino file implementation

That was the last part we had to implement in our server, we now should just copy the website files into the static folder on our SD card and insert it into the SD card Reader.

4 Website

In this section, we will not be providing any piece of code because of the verbose nature of web coding (Especially CSS).

All of the source code can be found in the Seafile folder dedicated to this project.

4.1 Used Technologies

- HTML: to build the website skeleton
- Tailwind CSS: used instead of css to avoid `CSS hell`
- Typescript: used instead of Javascript to write a safer code that won't fail on runtime
- Vue.js: used as frontend library to make reactivity easier

4.2 Design

We took inspiration from a Dribbble.com design of Syahrul Falah.

Figure 13: Website reference design

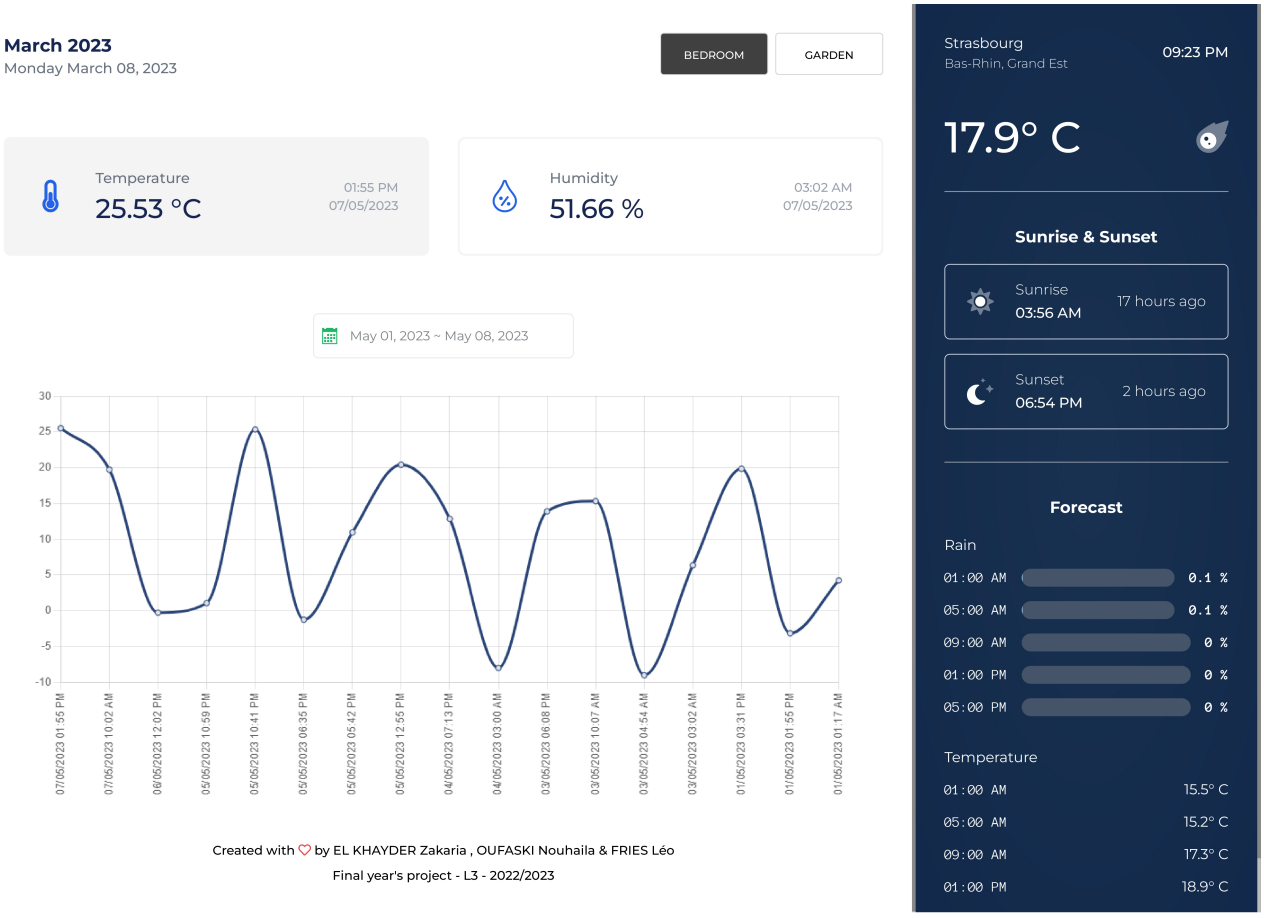


Figure 14: Website final result

4.3 Functionalities

4.3.1 Geolocation

We can get the user's current GPS location using the navigator's geolocation API, we can then reverse it back into an address using geocode.maps.co free API.

4.3.2 Weather

Using open-meteo.com free API, we can get the current temperature for specific GPS coordinates.

4.3.3 Sunrise & sunset

Using sunrise-sunset.org free API, we can get the sunrise & sunset for specific GPS coordinates.

4.3.4 Weather Forecast

We can use open-meteo.com free API to also get a weather forecast for specific GPS coordinates, such as rain probability and temperature forecast.

4.3.5 Nodes data visualization

We can parse the DB file that we can fetch from `/api/db` and transform it into an object that we can use to analyze and visualize the data.
We used a graph to show different measurements of different nodes for a specific period that can be chosen with the date range selector.
We are also showing the latest measurement in hindsight for ease of use.

5 Casing & Protection

We decided to think about the protection of the components of the nodes and of the central so we started to design different types of boxes on Autodesk Inventor with the intention of 3D printing it.

5.1 Central Box

First, we created the frame for the central, it should protect the borders of the screen and the components in a way to improve their lifespan and to give a better design to the central.

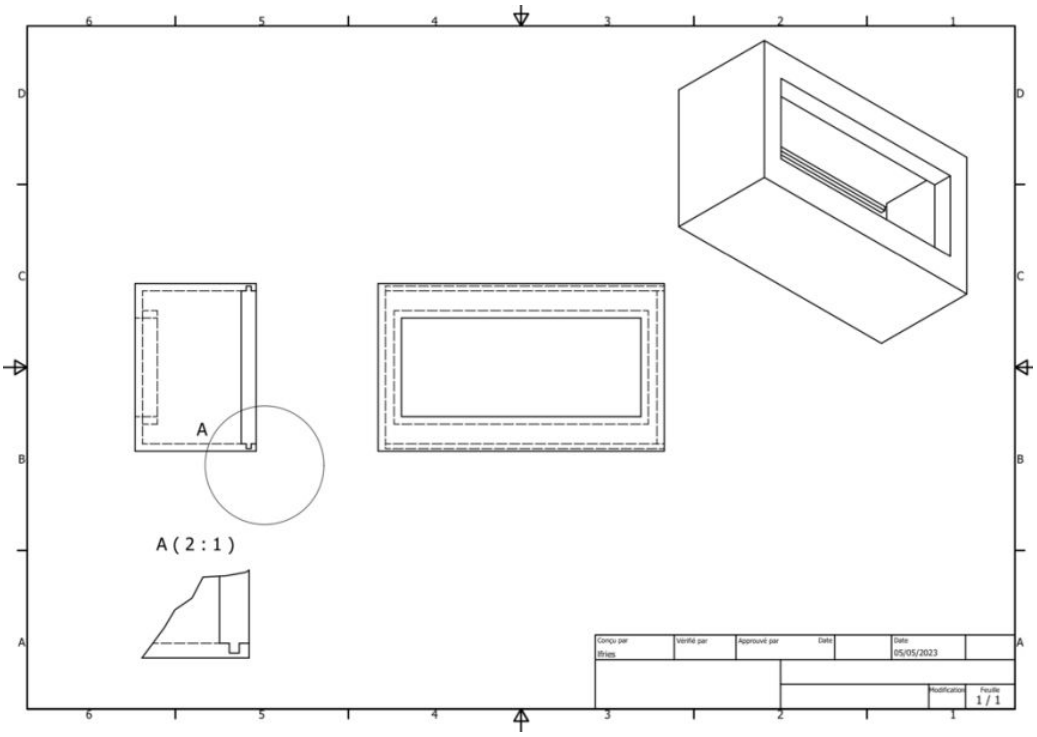
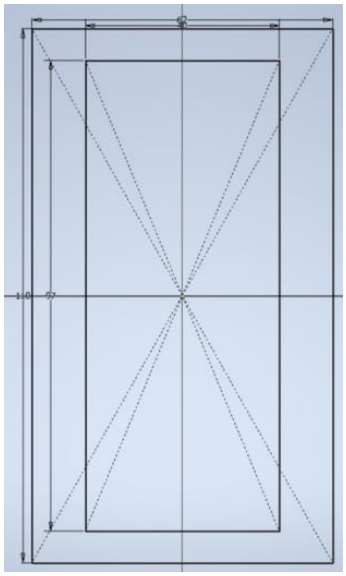
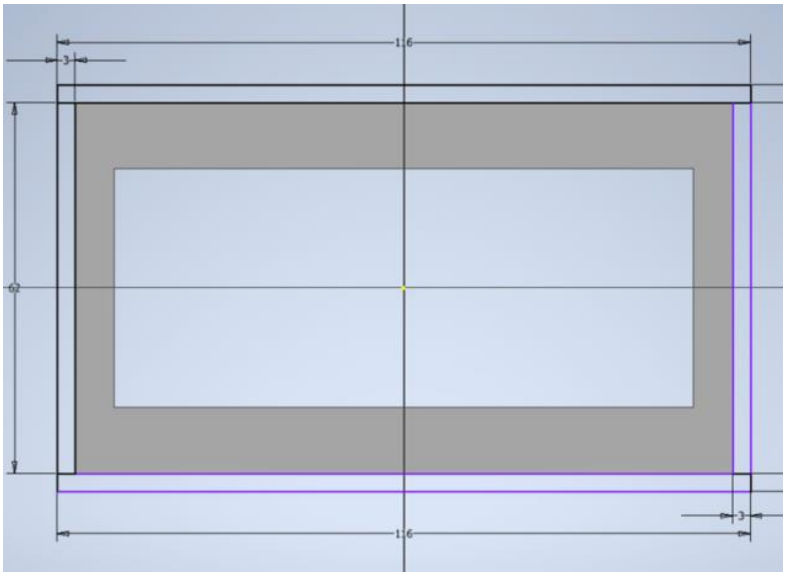


Figure 15: Plan of the frame for the screen

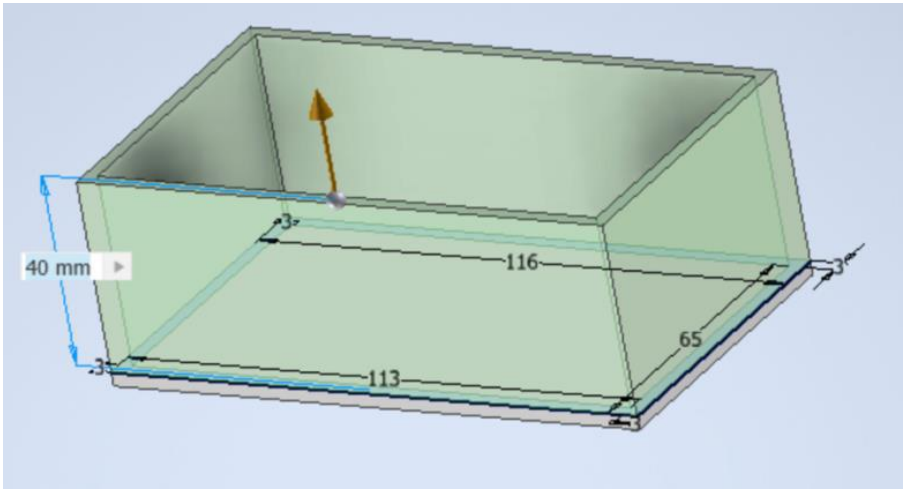
— Step-by-Step Construction

First, we draw a rectangle with another rectangle hole in the middle of the first one. This hole will receive the screen of the central. We give the external rectangle a thickness of 3mm and then we create the walls of the box, they will have the same thickness as the first rectangle.

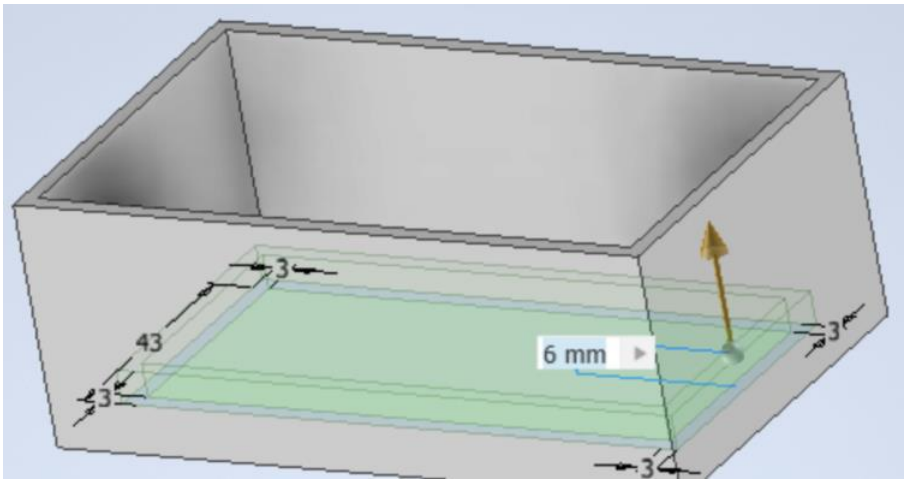




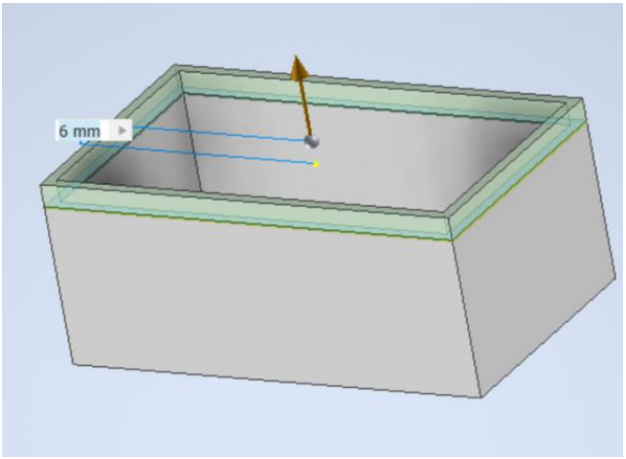
This box has to be high enough to contain all the components of the central so we decided to make it approximately 40mm high.

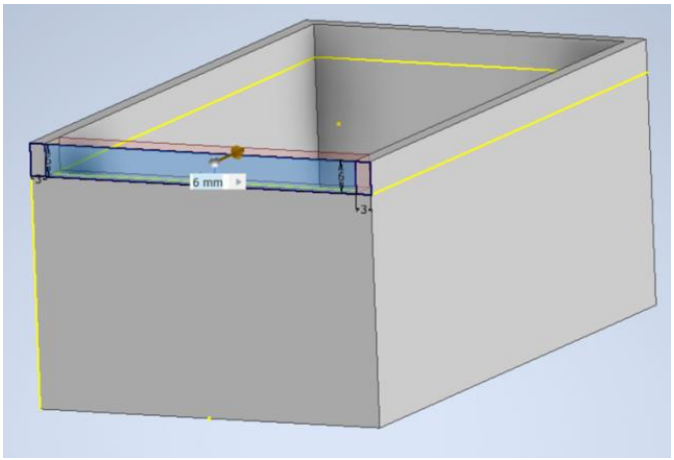


We also decided to integrate the screen as cleanly as possible, so we had to create elevations around the screen hole to put the screen at the same level as the front wall of the box. This way, the screen will be better protected and it will have a nicer design.



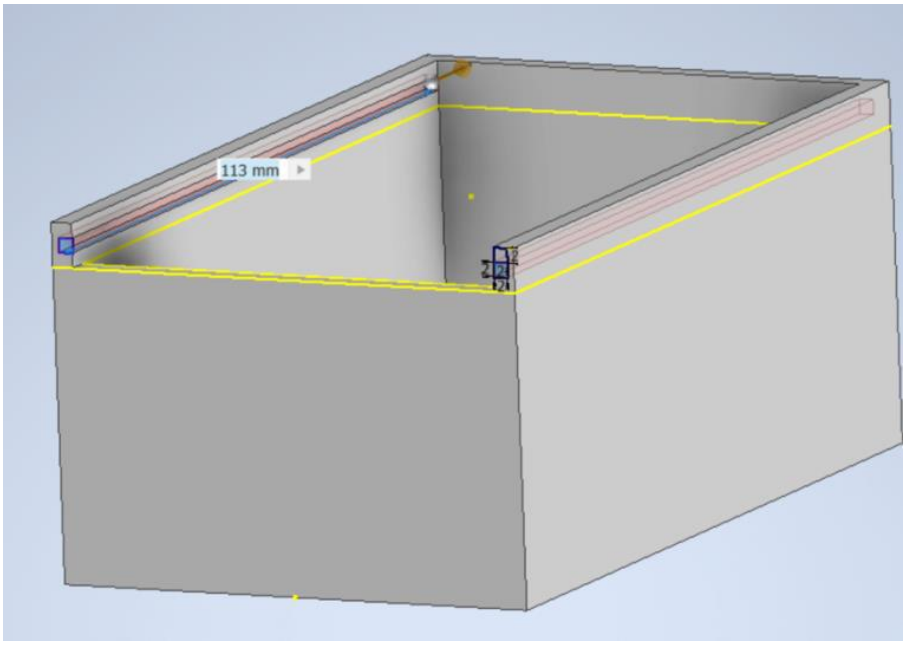
The last question about this box was "How to close it ?", to answer this question we decided to create a cover that would slide just behind the box and will remain tightly closed as we have left a fairly tight space.





To let enough space for the cover and for the components, we had to enlarge a bit the box and to create an

opening on one side. Finally, we had to create slits on both sides of the box to let the cover slide in its space.



When we finished the design of the box we had to think about the design of the cover. We decided to create a simple board that could slip in and out of its space.

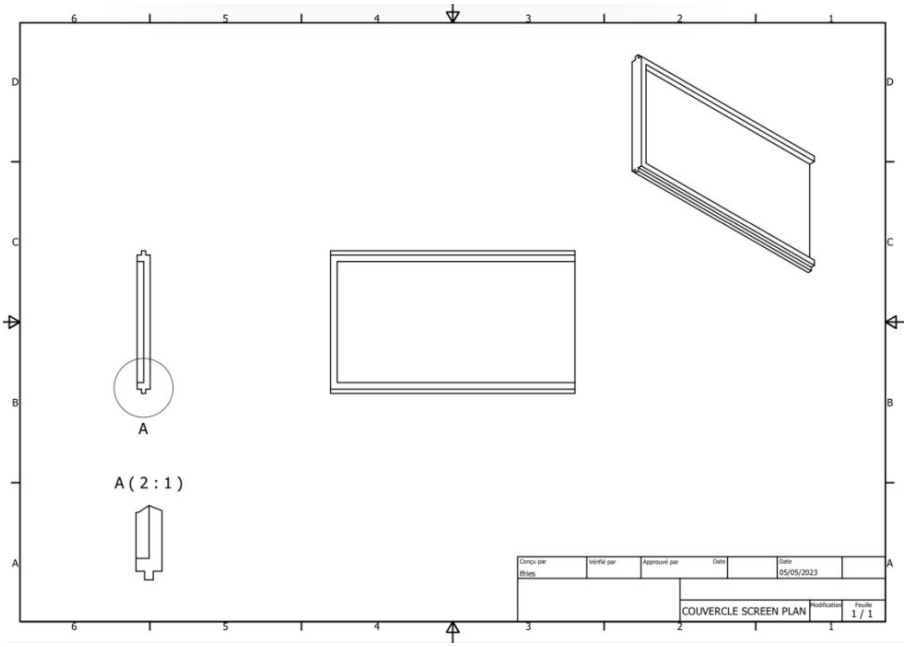
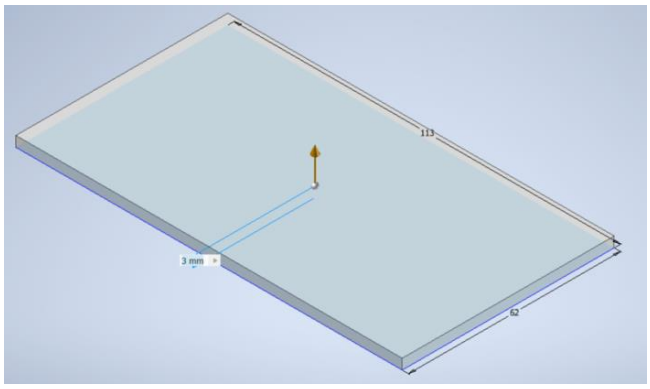
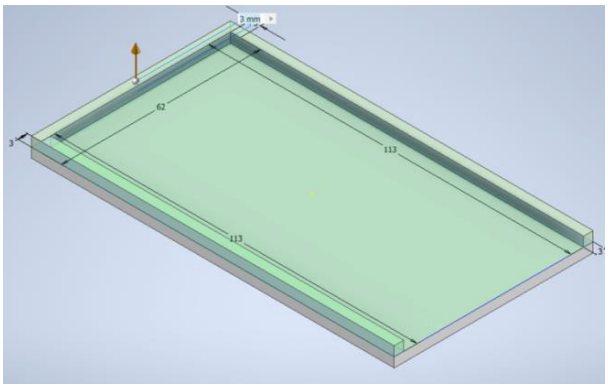


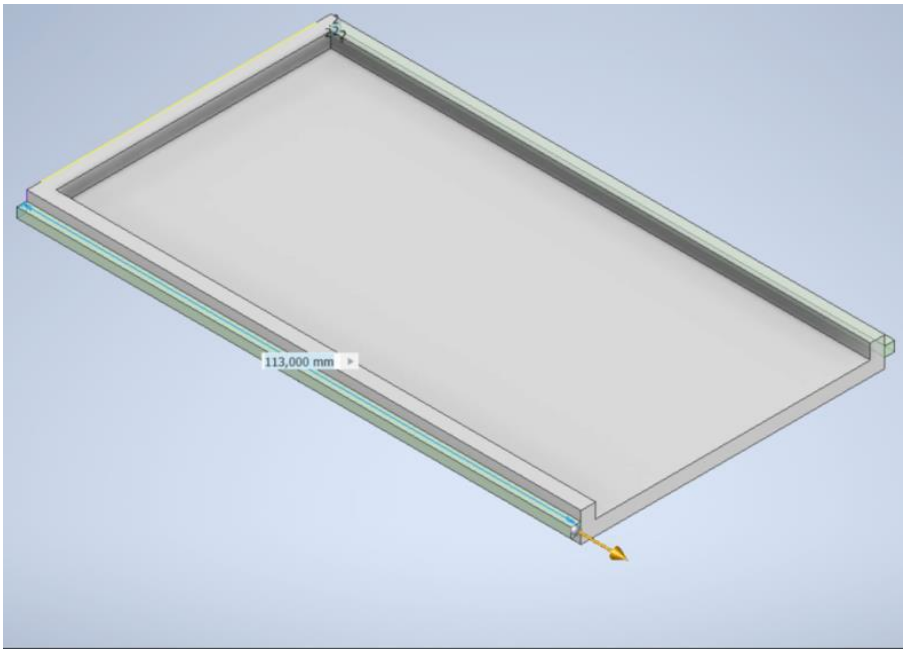
Figure 16: Plan of the cover of the central

To make this cover, we first started to create a rectangle of the right dimensions and we gave this rectangle a thickness of 3 mm. Then, we also added 3 mm to the sides of this rectangle so it can correctly close the box.





To finish, we created the male version of the slits in the box to allow the cover only to translate in one direction



5.2 Nodes Box

To start with this box, we first created the frame that will receive the components as the Arduino Nano and the sensors. We have to let an opening on the front face of this box to let air pass through it and allow the sensors to get the right values. We also have to let an available place outside the box to place the antenna.

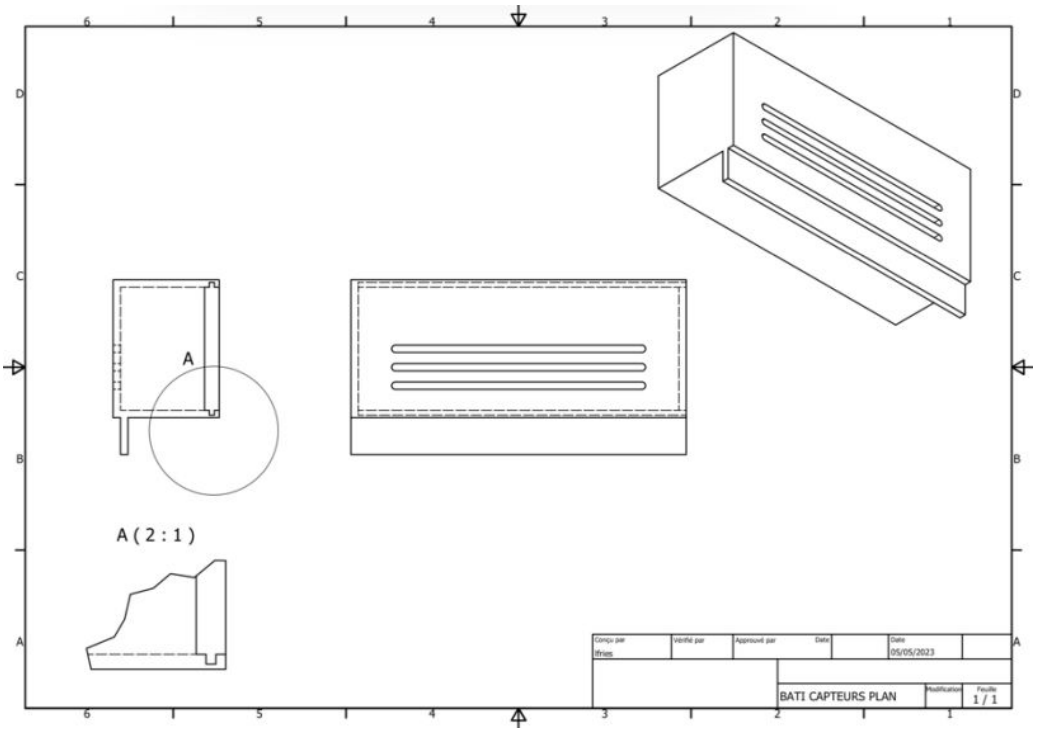
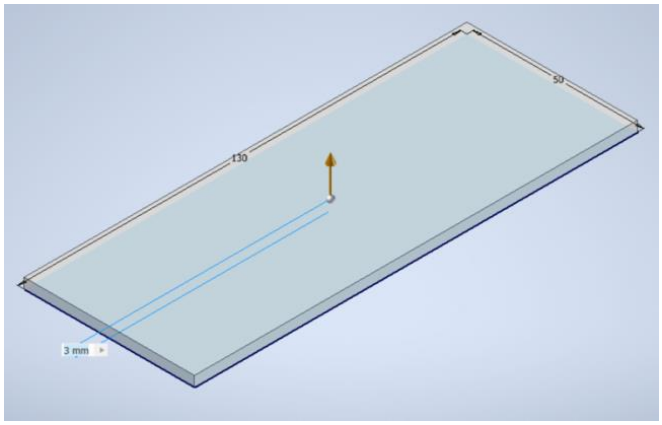
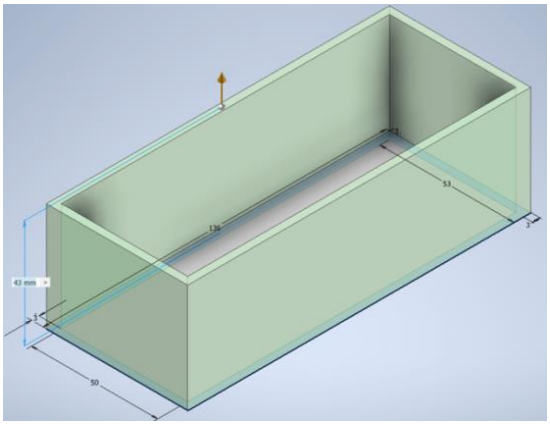


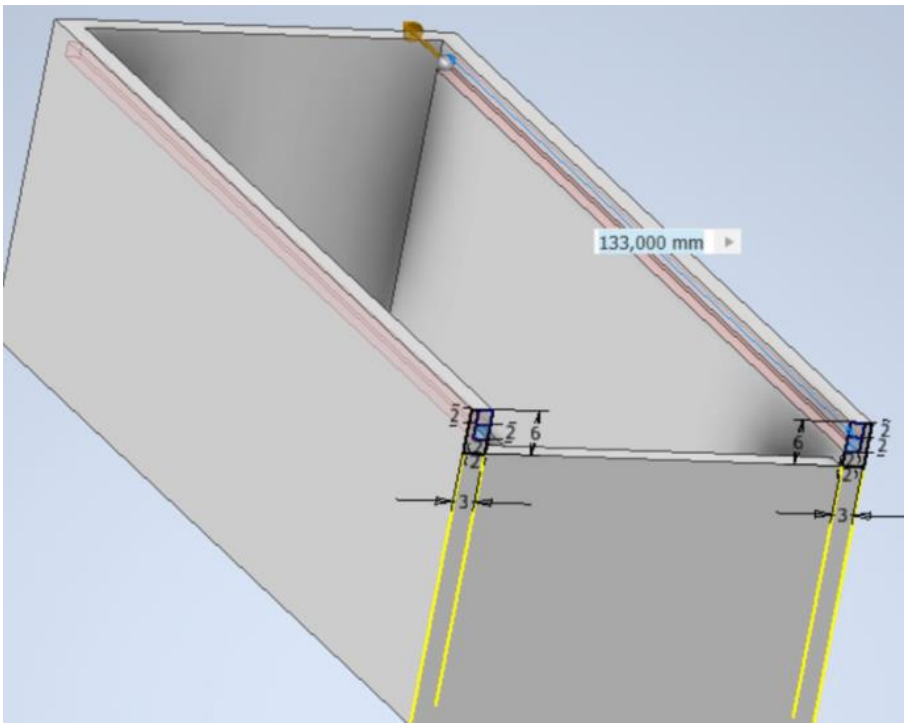
Figure 17: Plan of the box for the node

As for the central box, we created a rectangle of the right dimensions to receive the different parts to protect. We gave this rectangle a thickness of 3mm and also created walls to close the sides of the box.

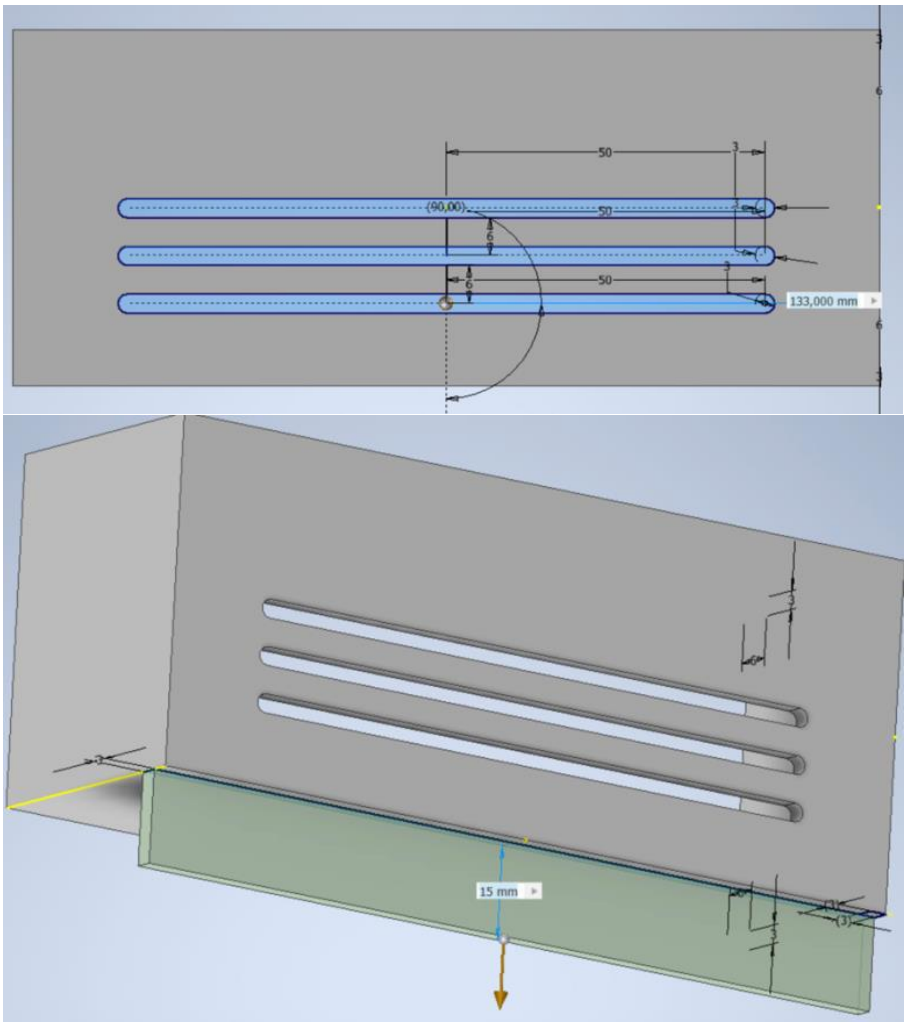




We also created the same slits as the first on the back of this box, in the way to close it with the same type of cover.



Finally, we had to create the openings on the front face and prepare space for the antenna.



To close this box we created the same type of cover as for the first box.

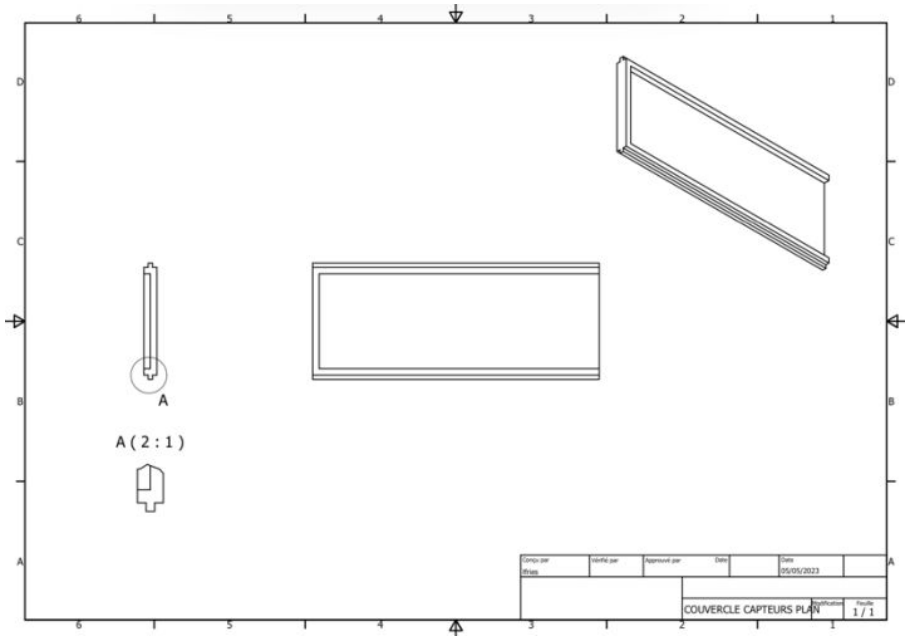


Figure 18: Plan of the cover for the node box

5.3 Remarks

After the impression and thinking of other problems, like putting the node box outside, we thought about different things that we could have done in another way.

— First, we could have created a little roof on the front face of the node box, so in rainy situations, water wouldn't really go inside it.

— We also thought that it would have been better if we printed this box in white because, in the case of high temperatures outside, our black box will be hot really fast.

— For both of the boxes, we also tough that we could have created a wall bracket to place our box on the wall with a screw or double-face adhesive.

6 Conclusion

In conclusion, the development and implementation of our weather station project have proven to be a resounding success. Through the careful selection and integration of sensors, namely temperature, pressure, and wind, we have been able to gather accurate and comprehensive weather data. The use of Arduino Nano devices, strategically positioned in different locations, ensures reliable and consistent data collection from different spots. The adoption of LoRa communication has allowed for the seamless transmission of the collected information to the central system, where it is processed and stored for future analysis. This central enables easy access to weather data through our website interface. Overall, this project serves as a testament to our team's dedication, technical skills, and problem-solving abilities.

It has provided us with valuable hands-on experience in designing and implementing an effective project resulting a well-functioning weather station.

Glossary

AP Access Point. 12

CSV Comma-seperated values. 22

DB Database. 12

HTTP Hypertext Transfer Protocol. 8, 12

LoRa Long Range - Wireless transmission technique. 12

WiFi Wireless Fidelity. 12