

# LR2: Lora Robot Rescue

Embedded Systems - Intelligent Systems and Robotics Laboratory Project



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA

Professors: Luigi Pomante, Giovanni De Gasperis  
Project Tutor: Marco Santic  
Student: Agnese Salutari - [agnes92@gmail.com](mailto:agnes92@gmail.com)

# LR2: Lora Robot Rescue

- What's the problem?
- LR2 Solution
- What is LoRa?
- Lora and LoraWAN
- LoRa/LoRaWAN Applications
- Frequency Modulation
- Chirp Spread Spectrum
- LoRa Radio Frequencies

# LR2: Lora Robot Rescue

- LoRaWAN Devices
- The Things Network
- Project Scheme
- Components
- Possible HW Solutions
- HW used in the Project
- Arduino MKR 1300
- My Application

# LR2: Lora Robot Rescue

- My Code
- Tests
- References

# What's the problem?

Working with outdoor robots, especially with rovers and drones, we always have some bothering problems:

- Limited energy autonomy
- Unpredictable breakdowns
- Troubles with communication in rural areas

... and finally the worst of all ...

- If one or more of the previous conditions occurs, how can I find my (maybe very expensive) no more traceable robot?

# LR2 Solution

LR2 is a module that can be installed on a Robot to:

- Periodically send the GPS location, the charge value of the battery and the robot temperature via a LoRaWAN connection
- When, for example, power is under and/or temperature is over a certain threshold, send an alert (containing these parameters) and signal the robot to enter in power-safe mode
- Make the robot more evident, by lighting and emitting sounds, when robot owner is looking for it

Another advantage:

- LR2 could be a nice antitheft for your robots

So now you have no more excuses to take your robots always locked in!

# What is LoRa?

LoRa = Long Range is a wireless technology to transmit small data packages (0.3 to 5.5 kb/s) over a long distance with low power consumption → important feature for embedded systems applications.

LoRa is used for LPWAN = Low Power Wide Area Network.

LoRa technology consists of radio transmission, based on spread spectrum modulation:

- CSS = Chirp Spread Spectrum modulation

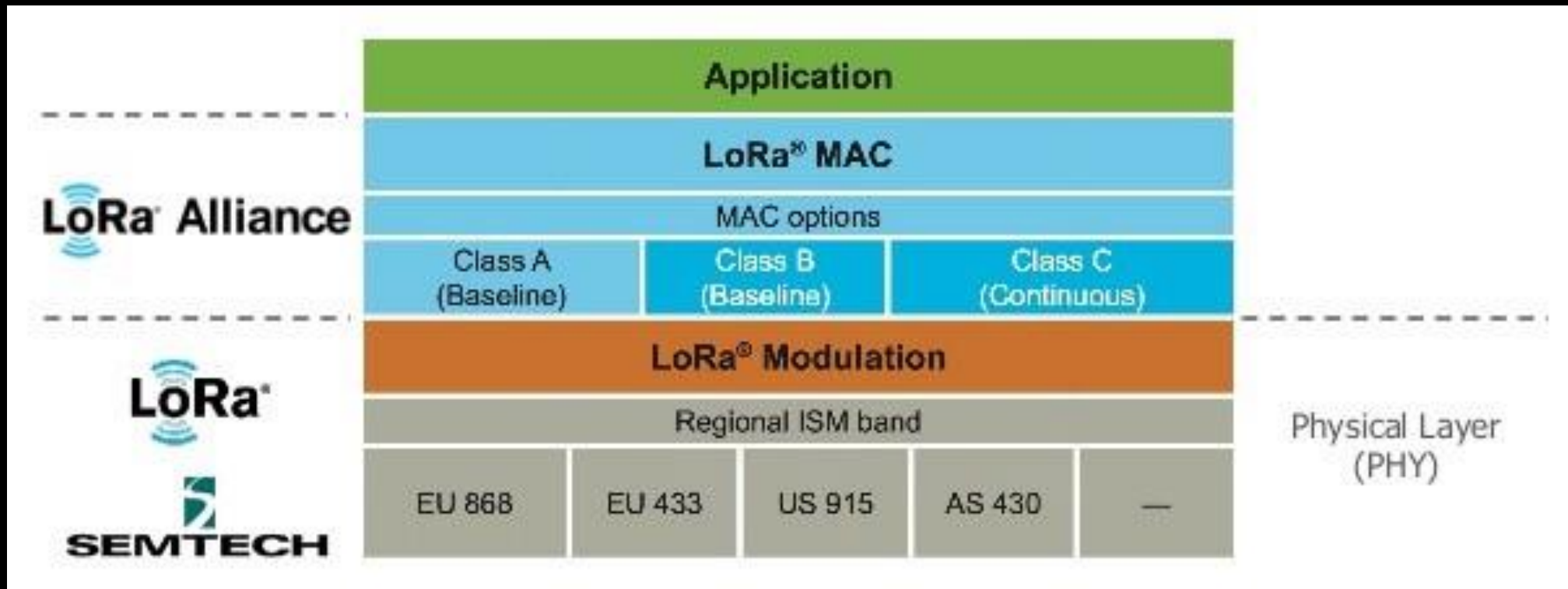
LoRa is very useful in IoT contests because it's efficient, flexible, has low power consumption and is a great choice for indoor and rural areas, that are not covered by cellular and Wi-Fi networks.

LoRa has been developed by Cycleo, that has been acquired by Semtech:

- Semtech (<https://www.semtech.com/>) is one of the founders of LoRa Alliance (<https://lora-alliance.org/>) and is the main LoRa chips producer

# LoRa and LoRaWAN

LoRaWAN is the network technology built upon LoRa physical layer:





# LoRa and LoRaWAN

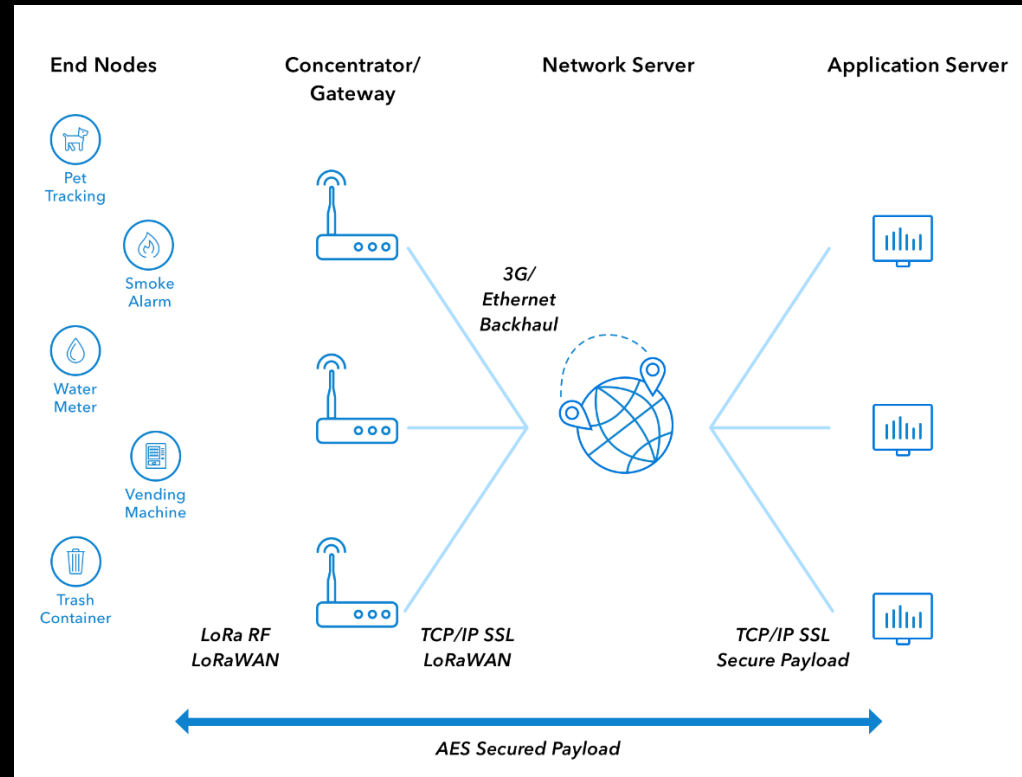
LoRaWAN is the media access control protocol for the management of the communication between the LPWAN gateway and the end-devices:

- It's managed by LoRa Alliance
- It defines the network topology, the communication protocol, communication frequencies, data rate and power management aspects
- The devices are asynchronous
- Data coming from an end-device may be received by many gateways, that are in charge of sending that data to a centralized server:
  - This server has to filter duplicate packets, perform security checks and manage the network
  - Finally, the data are delivered to applications servers

# LoRa and LoRaWAN

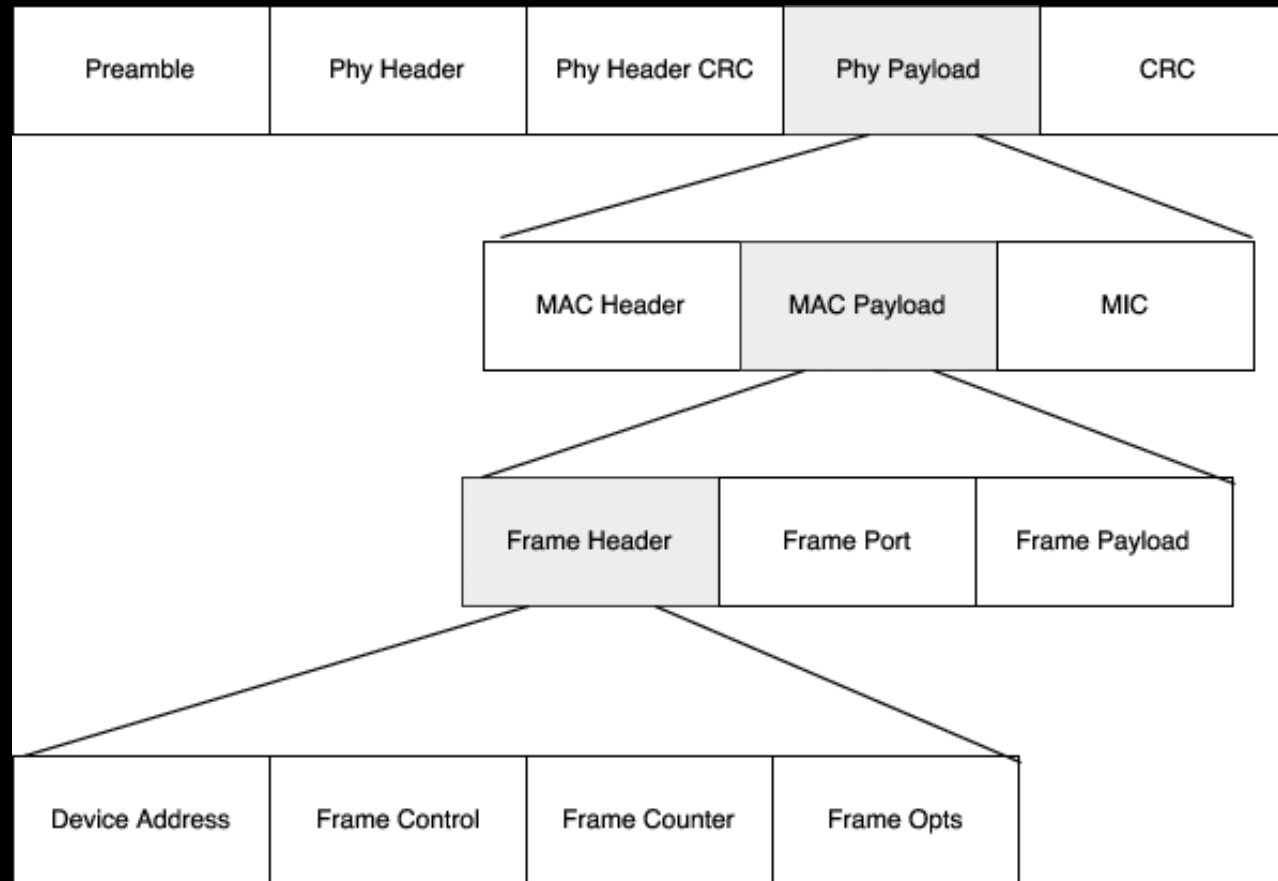
A LoRaWAN network has a star topology:

- End-Nodes are connected to one or more gateways (or concentrators)
- Each gateway (star-center) is connected to the Internet and can interact with servers



# LoRa and LoRaWAN

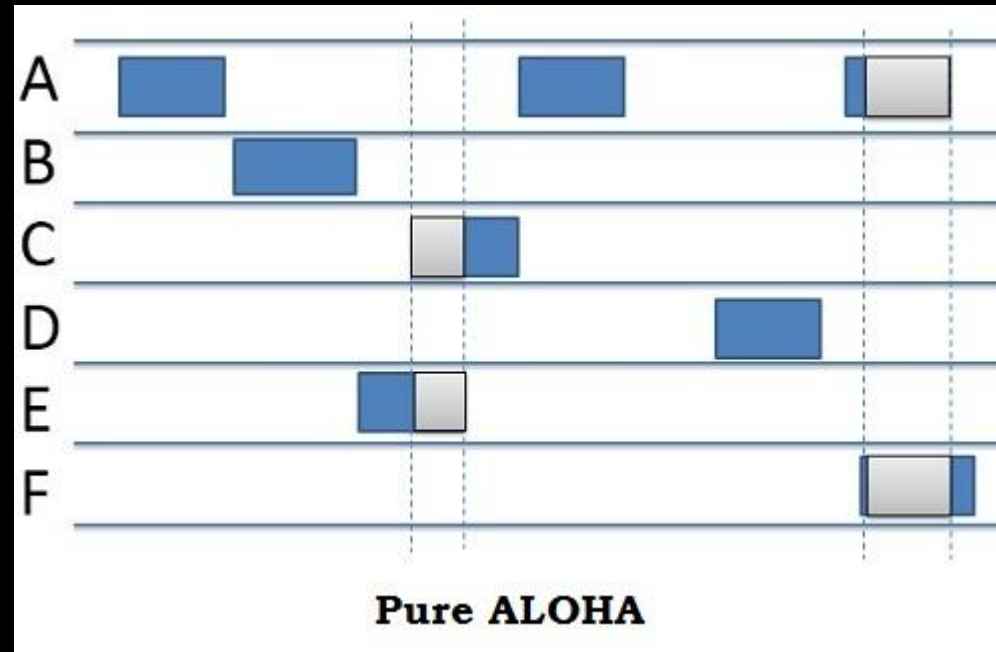
LoRaWAN Frame Structure:



# LoRa and LoRaWAN

Collisions are managed by using ALOHA protocol:

- Nodes are listening during transmissions to detect any collision



# LoRa/LoRaWAN Applications

LoRa/LoRaWAN technology can be used in different applications:

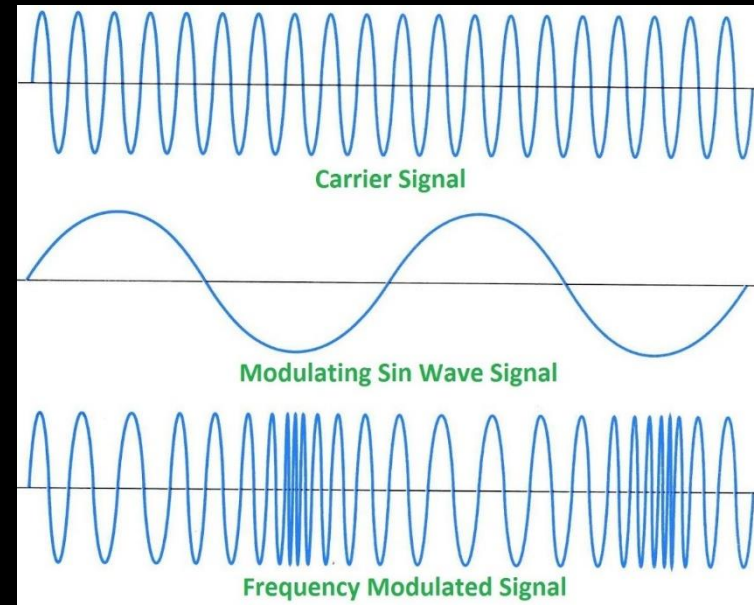
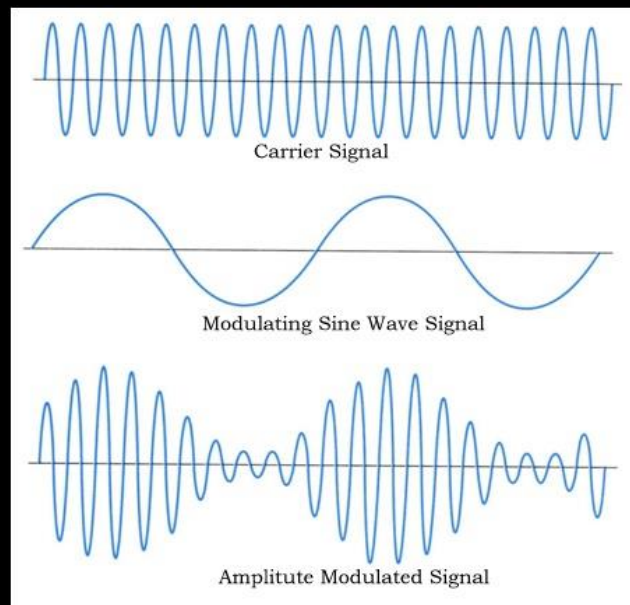
- Sensors Monitoring
- Resources Management
- Security
- Position Tracking
- Smart Home
- Smart City
- Smart Agriculture
- ...

# Frequency Modulation

Modulation is the technique used to represent data by performing appropriate changes on the transmitted signal.

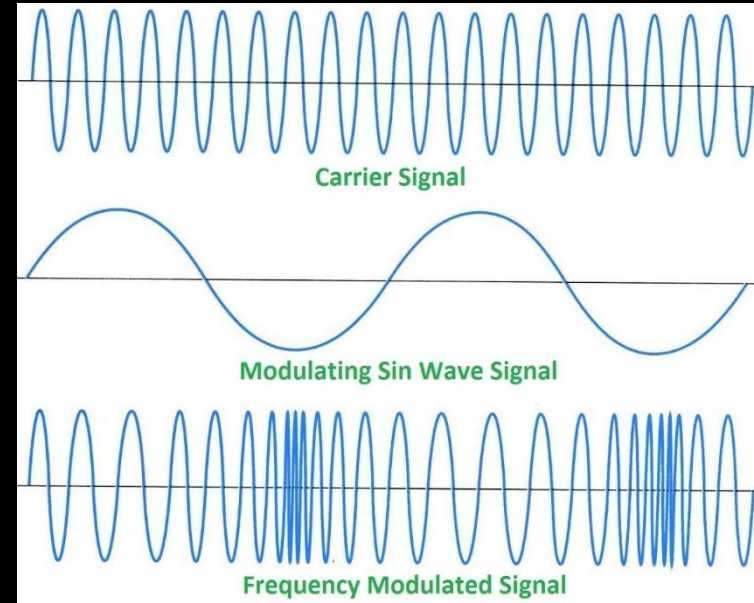
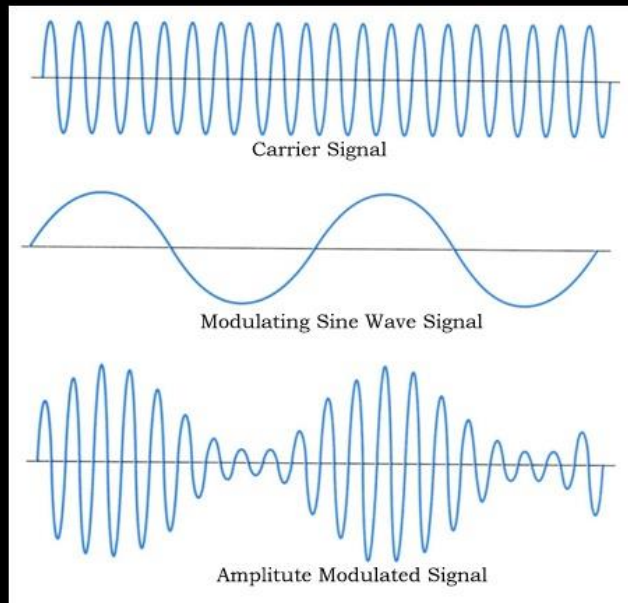
The 2 main modulation methods are:

- AM = Amplitude Modulation
- FM = Frequency Modulation



# Frequency Modulation

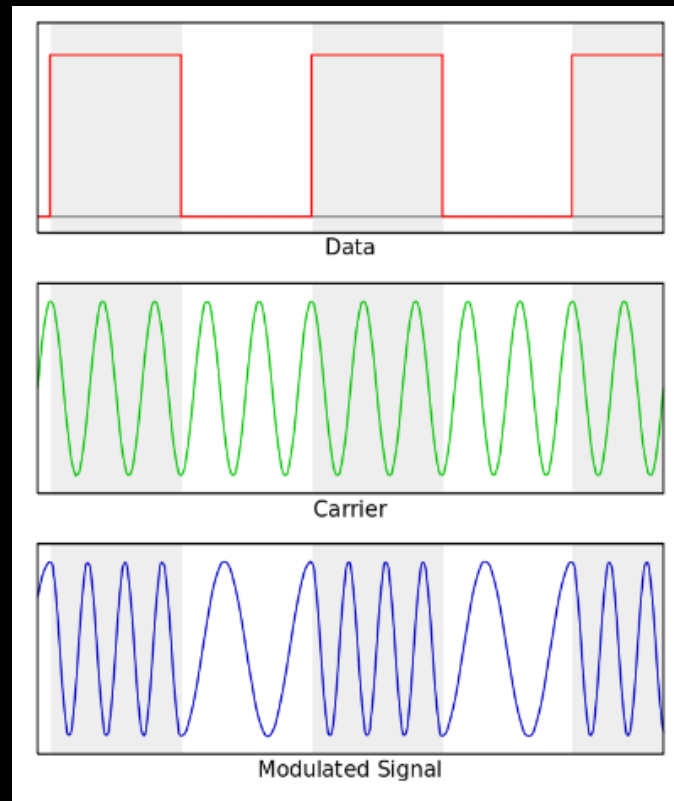
FM modulated signals are more noise resistant and have a higher quality, but AM modulated signals can be sent over a bigger distance.



# Frequency Modulation

FSK = Frequency Shift Keying modulation technique is a kind of frequency modulation.

It is based on a digital modulating signal → it has a digital signal input and an analog signal output.

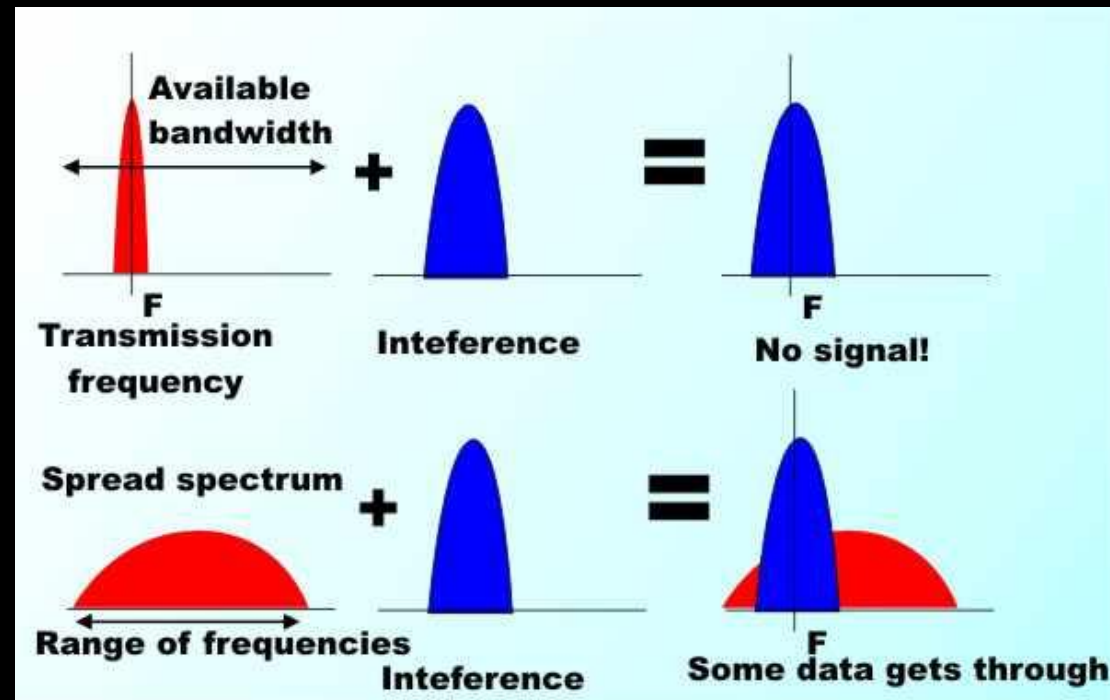




# CSS = Chirp Spread Spectrum

«Spread Spectrum» is referred to the fact that this transmission technique uses a bigger frequency band than strictly needed one:

- The advantage of doing such a thing is to increase the signal/noise relation, eliminating interferences:



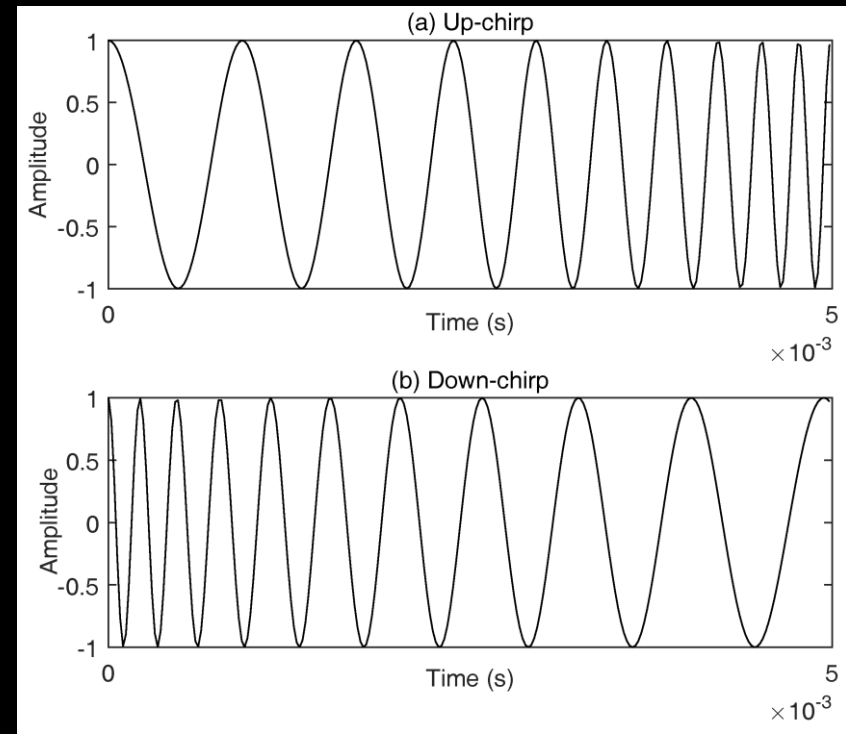
# CSS = Chirp Spread Spectrum

Chirp protocol has been developed during World War II for sonar and radar applications.

It is a particular implementation of Spread Spectrum, that has been optimized for high transmission reliability and low power consumption.

Modulated signal is made of chirp impulses:

- upchirp (rising in frequency)
- downchirp (falling in frequency)



# CSS = Chirp Spread Spectrum

In CSS, many chirps represent one single bit:

- SF = Spread Factor = #chirps/bit is in the interval [7, 12]
  - If SF is low, the transmission is faster, but it is less reliable
  - If SF is high, the transmission is slower, but it is more reliable

The SF that is used for LoRa transmissions can be defined in the configuration, that is different for each continent:

DataRate	Configuration	Indicative physical bit rate [bit/s]
0	LoRa: SF12 / 125 kHz	250
1	LoRa: SF11 / 125 kHz	440
2	LoRa: SF10 / 125 kHz	980
3	LoRa: SF9 / 125 kHz	1760
4	LoRa: SF8 / 125 kHz	3125
5	LoRa: SF7 / 125 kHz	5470
6	LoRa: SF7 / 250 kHz	11000
7	FSK: 50 kbps	50000

# CSS = Chirp Spread Spectrum

LoRa uses a Chip Sequence, or Spread Code, as a pattern to multiply data signal in order to perform the spreading.

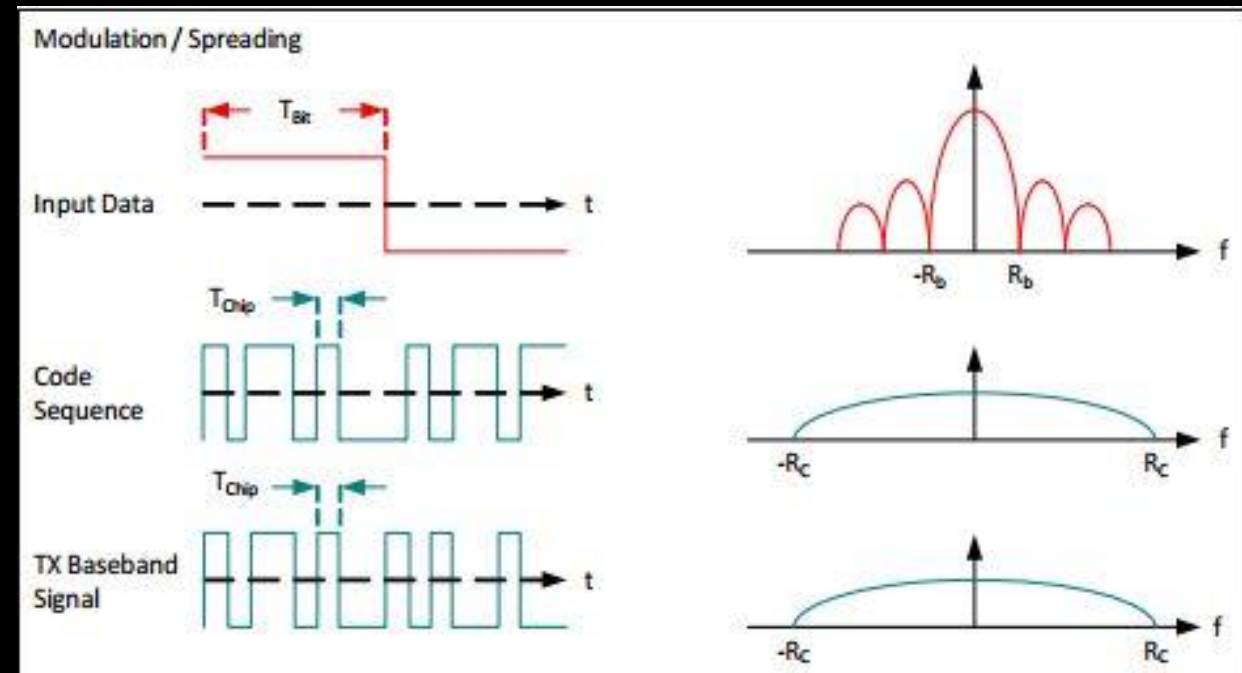
The relation between data bit rate and LoRa modulation chip rate is given by the modulation bit rate:

$$R_b = SF * 1/[2^{(SF)} / BW] \text{ bits/sec}$$

Where:

SF = Spreading Factor (7 - 12)

BW = modulation bandwidth (Hz)



# LoRa Radio Frequencies

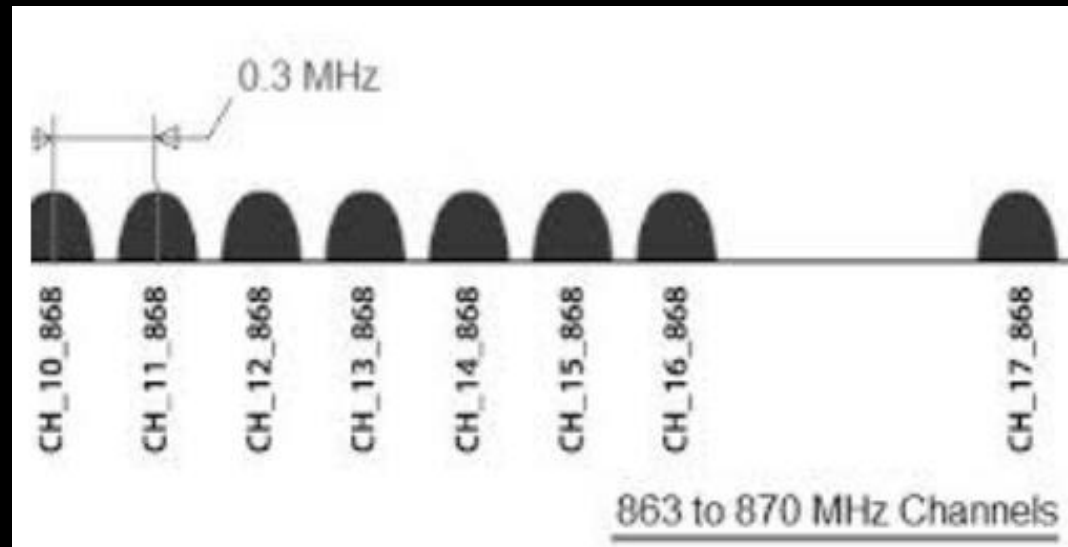
LoRa works on unlicensed band frequencies, that are different for every continent (and are managed by different rules in each country):

- 169 MHz
- 433 MHz
- 868 MHz (Europe)
- 915 MHz (North America)

# LoRaWAN Devices

There are many kinds of LoRaWAN devices:

- Gateways (or concentrators) are devices that receive uplink packets from end-devices and deliver downlink packets to end-devices, thanks to their software, the Packet Forwarder. They can be:
  - Single Channel Packet Forwarder
  - Multi Channel Packet Forwarder



# LoRaWAN Devices

There are many kinds of LoRaWAN devices:

- End-Devices are the nodes communicating with the gateways. They can belong to a different class:
  - Class A (bi-directional end-devices):
    - Allowed for bi-directional communications
    - Each end-device's uplink transmission is followed by two short downlink receive windows. The transmission slot scheduled is left to the end-device with a small variation based on a random time basis (ALOHA-type of protocol)
    - This Class A operation is the lowest power end-device system and is indicated for applications that only require downlink communication from the server shortly after the end-device has sent an uplink transmission. Downlink communications from the server at any other time will have to wait until the next scheduled uplink

# LoRaWAN Devices

There are many kinds of LoRaWAN devices:

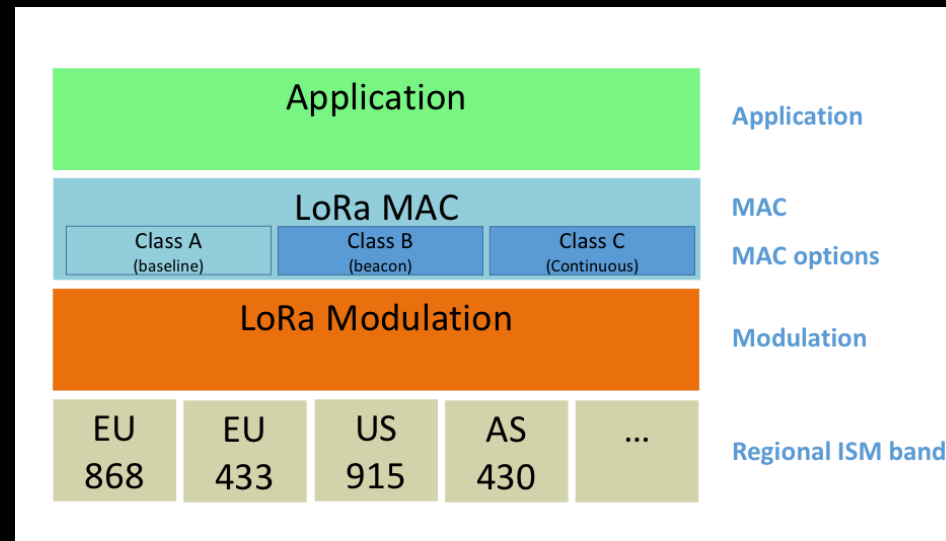
- End-Devices are the nodes communicating with the gateways. They can belong to a different class:
  - Class B (bi-directional end-devices with scheduled receive slots):
    - Allowed for periodic receive slots. In addition to the Class A random receive windows, Class B devices open extra receive windows at scheduled times
    - In order for the End-device to open its receive window at the scheduled time, it receives a time synchronized Beacon from the gateway. This allows the server to know when the end-device is listening



# LoRaWAN Devices

There are many kinds of LoRaWAN devices:

- End-Devices are the nodes communicating with the gateways. They can belong to a different class:
  - Class C (bi-directional end-devices with maximal receive slots):
    - They have almost-continuous open receive windows (closed when transmitting)
    - Class C end-device use more power to operate than Class A or Class B but they offer the lowest latency for server to end-device communication



# The Things Network

TTN = The Things Network (<https://www.thethingsnetwork.org/>) is one of the most famous LoRaWAN platforms:

- It's a free LoRaWAN service provider in Europe, USA and Asia
- In order to use it, you have to make a free account and register your applications and their devices:
  - You can configure device Activation Method as:
    - ABP (static): preferable if the device is always connected to the same network
    - OTAA (dynamic): preferable if the device often connect to different networks
- After doing that, you can use every TTN gateway that is reachable by your devices to send data through TTN → you do not have to buy your own gateway
- Devices with fixed hardcoded data rates of SF12 or SF11 are not allowed to join the network

# The Things Network

- You can send a limited amount of data through TTN by day:
  - About 30 seconds of uplink messages by day, for each device
  - At most 10 downlink messages by day (ACKs for uplink messages with confirmation request are included)
  - For the best performance your payload should be under 12 bytes (the maximum is 51 bytes long) and delay between messages should be at least several minutes:
    - The bigger the payload is, the slower is the transmission (and it's even worse if the gateway is far from the device)
    - The LoRaWAN protocol adds at least 13 bytes to the application payload
- In EU (ISM di 863-870MHz) there is a limit for the duty-cycle too:
  - It has to be equal to 1% for each sub-band (there are 8 sub-bands):
    - $\text{Toff}[\text{sub-band}] = (\text{TimeOnAir} / \text{DutyCycle}[\text{sub-band}]) - \text{TimeOnAir}$

# The Things Network

The screenshot shows a web browser window with two tabs: 'The Things Network' and 'The Things Network Console'. The address bar shows the URL 'console.thethingsnetwork.org/applications/lr2'. The console interface has a header with the 'THE THINGS NETWORK CONSOLE COMMUNITY EDITION' logo and navigation links for 'Applications', 'Gateways', and 'Support'. A user profile 'agnsal' is logged in. The main content area is titled 'Applications > lr2'. It displays the following details for the application 'lr2':

- Application ID**: lr2 (with a [documentation](#) link)
- Description**: Lora Robot Rescue Application
- Created**: 16 seconds ago
- Handler**: ttn-handler-eu (current handler)

Below this, there is a section for 'APPLICATION EUIs' with a [manage euis](#) link. It shows a single EUI: 70 B3 D5 7E D0 02 72 B3, with expand/collapse and copy icons.

The next section is 'DEVICES' with links for [register device](#) and [manage devices](#). It shows 0 registered devices with a server icon.

# The Things Network

For this project implementation I chose TTN because it's free and I could easily use their service and infrastructure in L'Aquila city:

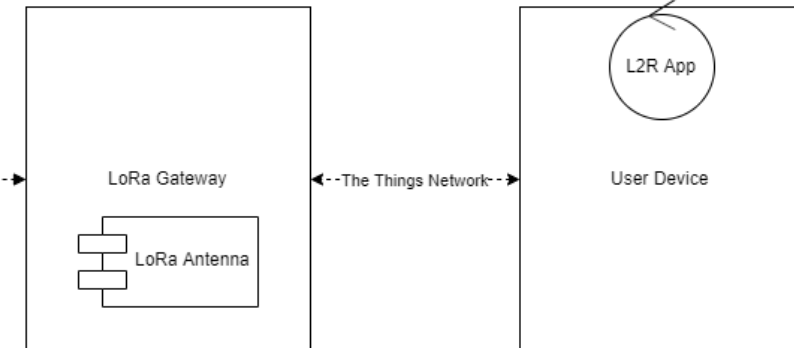
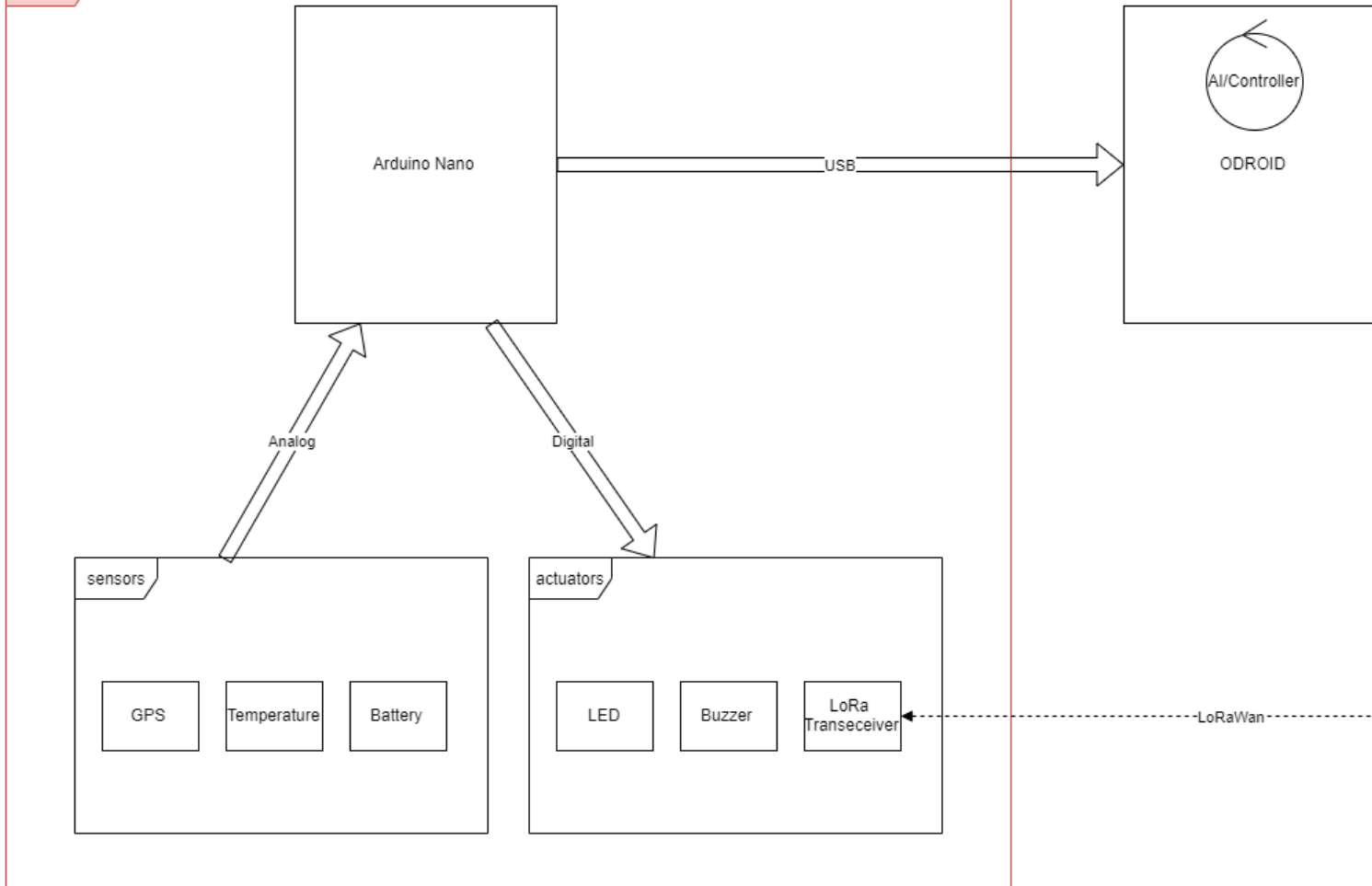
There are other LoRaWAN platforms you can use instead for real application, that are not free but allow you to use a wider band and provide a more efficient service:

- Globalsat
- ThingsConnected
- ResIOT
- Everynet
- LORIoT.io
- ...

# Project Scheme

Robot

L2R



# Components

- An end-node, made of:
  - An Arduino board
  - A LoRa Transceiver Module
  - Sensors:
    - A voltage level sensor
    - A temperature sensor
    - A GPS sensor
  - Actuators:
    - A LED
    - A buzzer

# Components

- [An Odroid board (the robot controller that receives data from Arduino board)]
- [A 3D printed plastic case, containing the Odroid board and LR2]
- [A robot to host LR2 (in its plastic case)]
- A LoRa gateway (with a LoRa antenna)
- Breadboard and circuitry



# Possible HW Solutions

- Arduino Board and SX1278 LoRa Module
- Arduino MKR WAN 1300 (or a similar board): a LoRa Module (integrated) on Arduino board
  - Preferable (very tiny)
  - <https://store.arduino.cc/arduino-mkr-wan-1300-lora-connectivity-1414>



# Possible HW Solutions

- There is a GPS shield for Arduino MKR 1300 (<https://store.arduino.cc/arduino-mkr-gps-shield>), that can be easily installed on the board, but it was sold out when I was buying project stuff, so I finally chose another kind of GPS that seems to work fine with Arduino boards and that is quite precise and not very expensive: the NEO-6M GPS ([https://www.amazon.it/HiLetgo-GY-NEO6MV2-Controller-Ceramic-Antenna/dp/B07B4658XJ/ref=cm\\_cr\\_arp\\_d\\_product\\_top?ie=UTF8](https://www.amazon.it/HiLetgo-GY-NEO6MV2-Controller-Ceramic-Antenna/dp/B07B4658XJ/ref=cm_cr_arp_d_product_top?ie=UTF8)).



# Possible HW Solutions

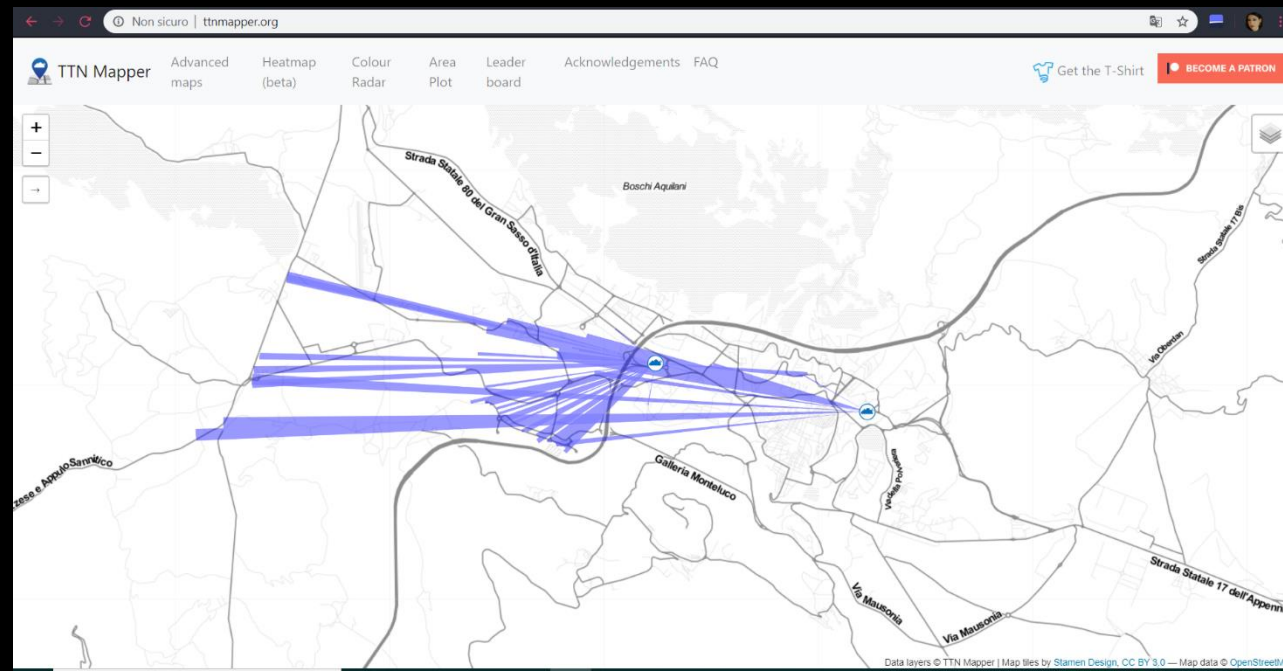
- A good gateway could be the TTN one, The Things Gateway (<https://www.kickstarter.com/projects/419277966/the-things-network>):
  - Easy to set up
  - Multi-channel gateway
  - Cheap
  - Provides up to 10 km / 6 miles radius of network coverage
  - Connects easily to your WiFi or Ethernet connection
  - Security through the https connection and embedded in the LoRaWAN protocol
  - Runs on open hardware
  - Contains GPS to determine the gateway's location and node's location later
  - Can serve up to 10000 nodes

# HW used in the Project

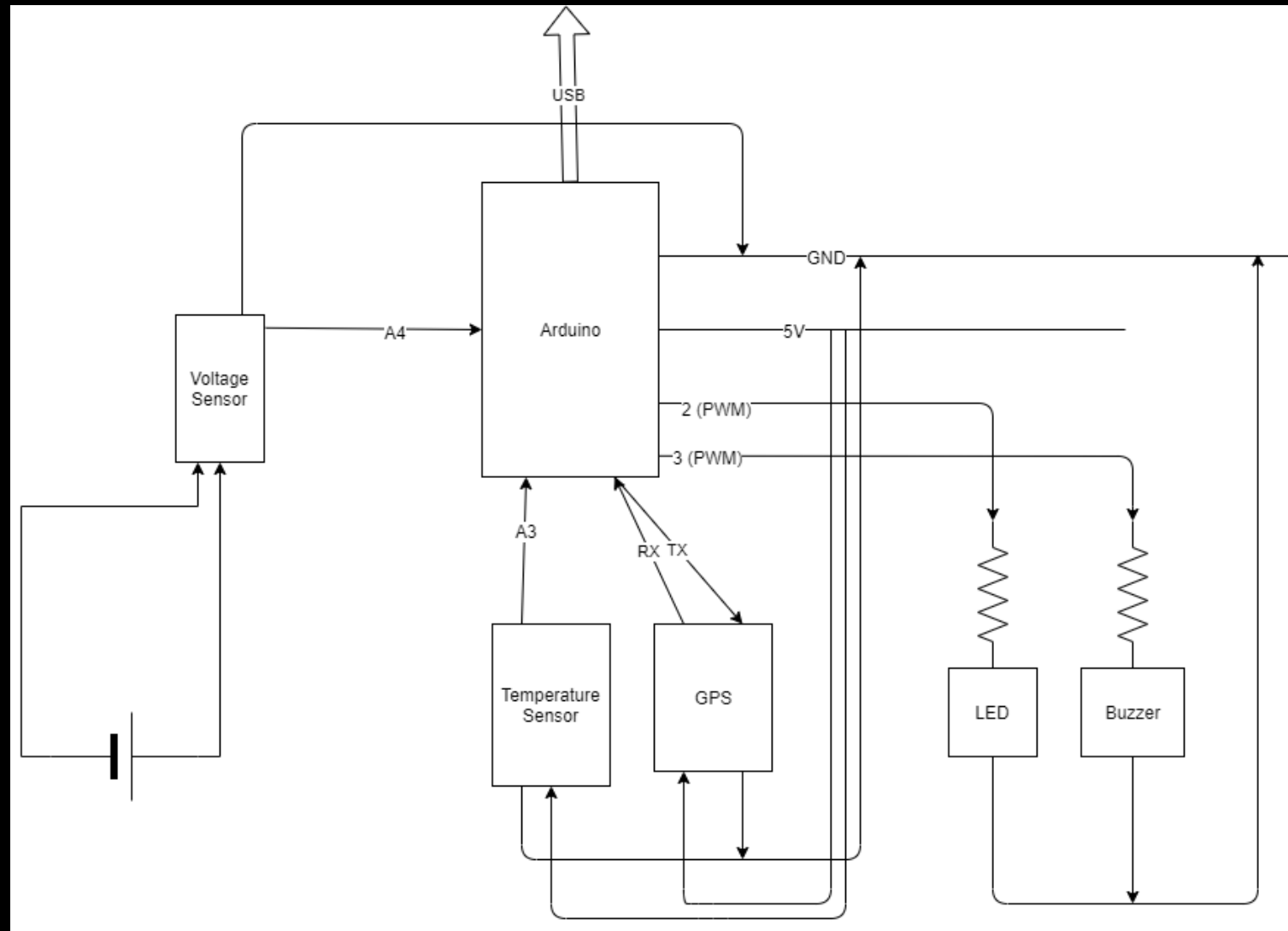
- Arduino MKR WAN 1300 (<https://store.arduino.cc/arduino-mkr-wan-1300-lora-connectivity-1414>)
- DHT22 Temperature Sensor ([https://www.amazon.it/gp/product/B07M6N3VGC/ref=ppx\\_yo\\_dt\\_b\\_asin\\_title\\_001\\_so0?ie=UTF8&psc=1](https://www.amazon.it/gp/product/B07M6N3VGC/ref=ppx_yo_dt_b_asin_title_001_so0?ie=UTF8&psc=1))
  - It works from -40 to 80 Celsius degrees (it's more expensive than DHT11 or other sensors, but it's necessary for cold locations)
  - For bureaucracy problems, I had to buy materials by myself and unfortunately this sensor has not arrived yet → I used a model we already had in the Lab, that is not so good
- NEO-6M GPS ([https://www.amazon.it/HiLetgo-GY-NEO6MV2-Controller-Ceramic-Antenna/dp/B07B4658XJ/ref=cm\\_cr\\_ar\\_p\\_d\\_product\\_top?ie=UTF8](https://www.amazon.it/HiLetgo-GY-NEO6MV2-Controller-Ceramic-Antenna/dp/B07B4658XJ/ref=cm_cr_ar_p_d_product_top?ie=UTF8))
- Arduino Compatible Voltage Sensor ([https://www.amazon.it/gp/product/B07TBFLHBX/ref=ppx\\_yo\\_dt\\_b\\_asin\\_title\\_000\\_so0?ie=UTF8&psc=1](https://www.amazon.it/gp/product/B07TBFLHBX/ref=ppx_yo_dt_b_asin_title_000_so0?ie=UTF8&psc=1)):
  - It's a Voltage Divider: for 3.3V systems, input voltage has to be not greater than  $3.3V \times 5 = 16.5V$

# HW used in the Project

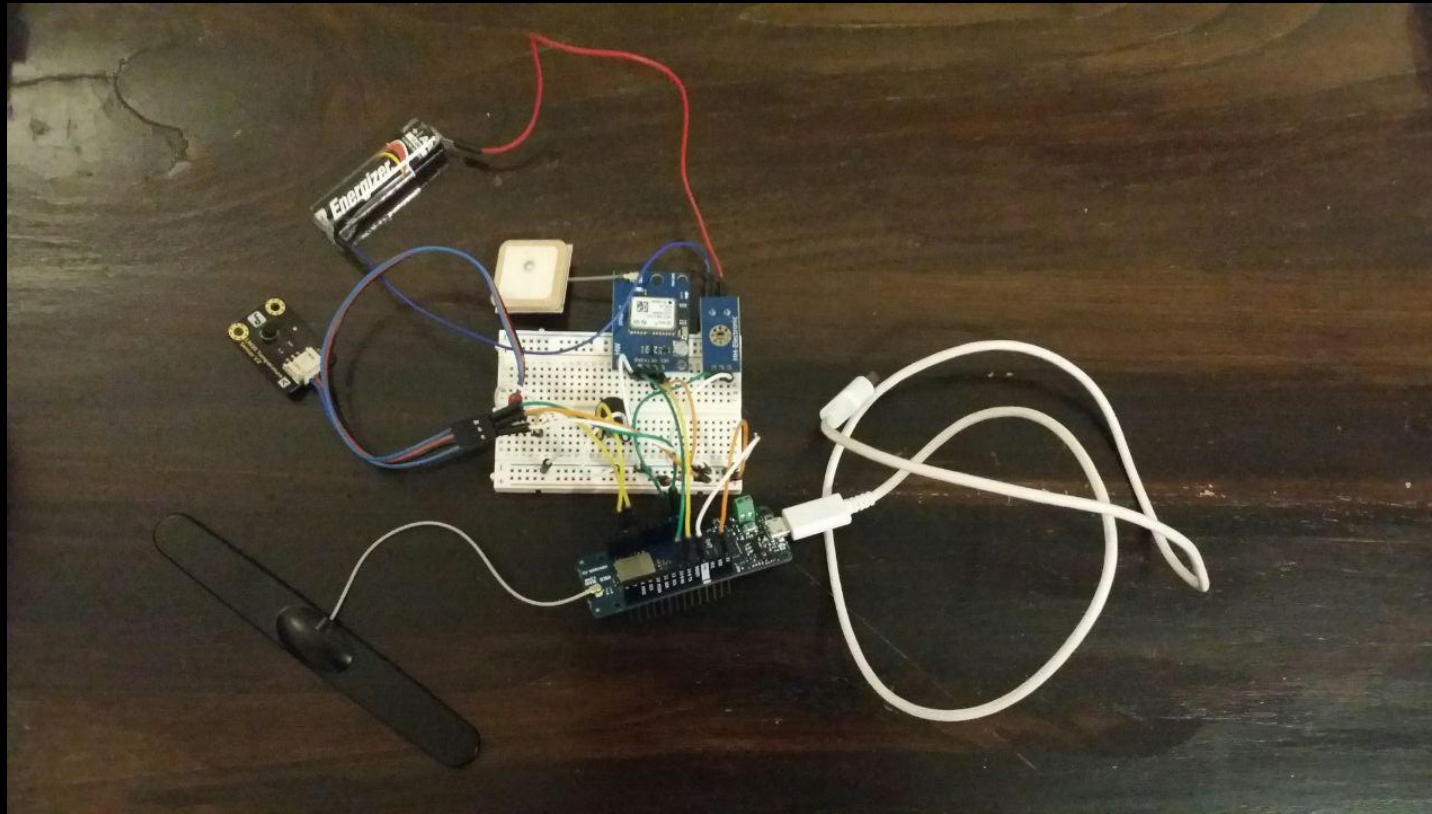
- I used a gateway provided by Fab Lab (<https://www.fablaquila.org/chi-siamo/>):
  - I couldn't use LoRaWAN from the university (probably because of some obstacle), so they kindly allowed me to perform some tests in Fab Lab building and shared their experience with me (a special thank to Luca Anastasio)
  - They helped improving the following map of L'Aquila city LoRaWAN coverage (<http://ttnmapper.org/>)



# HW used in the Project



## HW used in the Project



Obviously, breadboard and cables are acceptable for testing and prototyping purposes only.



# Arduino MKR 1300

- The [Arduino MKR WAN 1300](#) and is designed to offer a practical and cost-effective solution for developers, makers and enterprises, enabling them to quickly add connectivity to their projects and ease the development of battery-powered IoT edge applications
- The highly compact board measures just 67.64 x 25mm, together with low power consumption, making it an ideal choice for emerging battery-powered IoT edge devices in the MKR form factor for applications such as environmental monitoring, tracking, agriculture, energy monitoring and home automation
- Offering 32-bit computational power similar to the [Arduino MKR ZERO board](#), the MKR WAN 1300 is based around the Murata LoRa low-power connectivity module and the Microchip SAM D21 microcontroller, which integrates an ARM Cortex-M0+ processor, 256KB Flash memory and 32KB SRAM. The board's design includes the ability to be powered by either two 1.5V AA or AAA batteries or an external 5V input via the USB interface – with automatic switching between the two power sources



# Arduino MKR 1300

- It has an operating voltage of 3.3V, eight digital I/Os, 12 PWM outputs and UART, SPI and I2C interfaces

BOARD	USB CDC NAME	SERIAL PINS	SERIAL1 PINS	SERIAL2 PINS	SERIAL3 PINS
Uno, Nano, Mini		0(RX), 1(TX)			
Mega		0(RX), 1(TX)	19(RX), 18(TX)	17(RX), 16(TX)	15(RX), 14(TX)
Leonardo, Micro, Yún	Serial		0(RX), 1(TX)		
Uno WiFi Rev.2		Connected to USB	0(RX), 1(TX)	Connected to NINA	
MKR boards	Serial		13(RX), 14(TX)		
Zero	SerialUSB (Native USB Port only)	Connected to Programming Port	0(RX), 1(TX)		
Due	SerialUSB (Native USB Port only)	0(RX), 1(TX)	19(RX), 18(TX)	17(RX), 16(TX)	15(RX), 14(TX)
101	Serial		0(RX), 1(TX)		

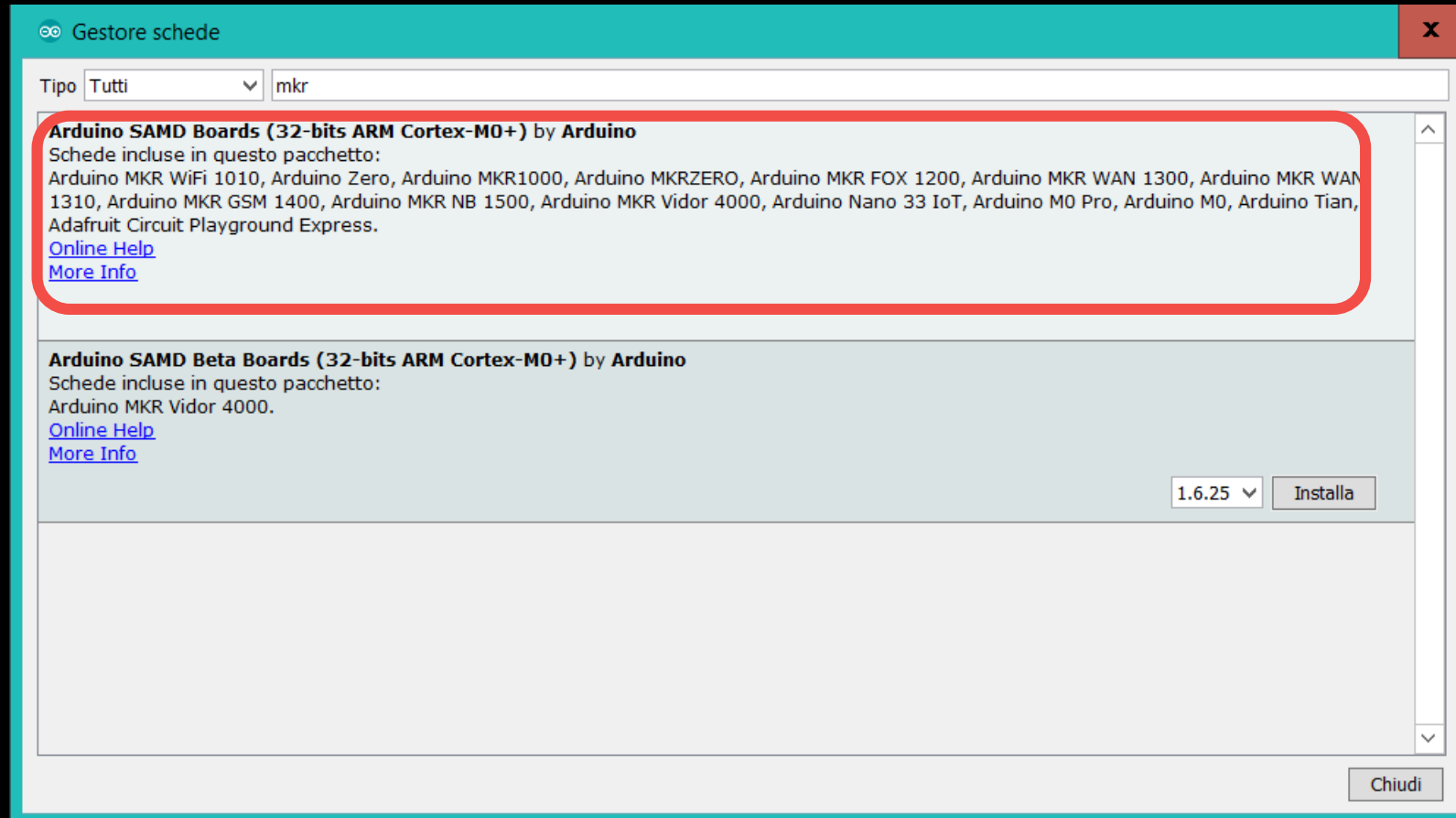
# Arduino MKR 1300

Microcontroller	SAMD21 Cortex-M0+ 32bit low power ARM MCU
Radio module	CMWX1ZZABZ (datasheet)
Board Power Supply (USB/VIN)	5V
Supported Batteries(*)	2x AA or AAA
Circuit Operating Voltage	3.3V
Digital I/O Pins	8
PWM Pins	12 (0, 1, 2, 3, 4, 5, 6, 7, 8, 10, A3 - or 18 -, A4 -or 19)
UART	1
SPI	1
I2C	1
Analog Input Pins	7 (ADC 8/10/12 bit)
Analog Output Pins	1 (DAC 10 bit)
External Interrupts	8 (0, 1, 4, 5, 6, 7, 8, A1 -or 16-, A2 - or 17)
DC Current per I/O Pin	7 mA

Flash Memory	256 KB
SRAM	32 KB
EEPROM	no
Clock Speed	32.768 kHz (RTC), 48 MHz
LED_BUILTIN	6
Full-Speed USB Device and embedded Host	
Antenna power	2dB
Carrier frequency	433/868/915 MHz
Working region	EU/US
Length	67.64 mm
Width	25 mm
Weight	32 gr.

# Arduino MKR 1300

- Setting up Arduino IDE:



# Arduino MKR 1300

- Pro:
  - It is configurable as a Class C device → nice for IoT applications
  - Arduino boards are open-source
  - Arduino has a big community, so you can easily find a solution for any problem
  - There are a lot of libraries you can use for IoT projects
  - The transceiver is onboard and can be easily managed with its library
- Cons:
  - Arduino MKR 1300 is one of the latest Arduino boards, so there is not much online:
    - Documentation too is not very complete → You have to test a lot of features by yourself
    - It is not very cheap

# My Application

- The device has to:
  - Check for incoming commands from USB
  - Check for incoming commands from LoRaWAN
  - React to received commands with the proper behaviour
  - Periodically update sensor reading values
  - Check for alarms conditions
  - Properly react to alarms (sensor reading delay time is reduced in case of alarms)
  - Periodically send its state and sensor reading values via USB
  - Periodically send its state and sensor reading values via LoRaWAN

# My Application

- Via USB and LoRaWAN commands, the user can:
  - Define the configuration parameters:
    - Temperature (middle value and range)
    - Voltage (minimum value)
    - Latitude (middle value and range)
    - Longitude (middle value and range)
    - Standard delay time for sensor update (its reduced during alarms)
    - LoRaWAN messages delay time
  - Send the noisy command:
    - To make the device blinking and emitting sounds
  - Send the stop command
  - Send the stop-usb-sending command
  - Send the stop-lora-sending command

# My Code

- To manage LoRaWAN communication, I used Arduino MKRWAN library.
- To manage GPS sensor, I used TinyGPS library:
  - I used Serial1 for GPS sensor and Serial (Serial0) for USB serial communication

```
// LoRaWan dependencies:  
#include <MKRWAN.h>  
#include "arduino_secrets.h"  
  
// GPS dependencies:  
#include <TinyGPS.h>
```

```
// Other needed Serials:  
# define gpsSerial Serial1 // rx and tx pins (microUsb is Serial, or Serial0)
```

```
// LoRaWan manager:  
LoRaModem modem;  
// Uncomment if using the Murata chip as a module  
// LoRaModem modem(Serial1);  
  
// GPS manager:  
TinyGPS gps; // create gps object
```

# My Code

- I defined the default configuration parameters values:

```
// Configurations:
float middleTemp = 20, radiusTemp = 20; // Celsius
float middleLat = 42.094079, radiusLat = 0.100000;
float middleLon = 13.3714029, radiusLon = 0.100000;
uint8_t minVolt = 0; // Volts
uint8_t lookAtMeLevel = 5;
uint16_t cycleDelayTime = 600; // Seconds/10
uint8_t loraDelayTime = 1; // Minutes
int actualDelayTime = 0; // Milliseconds
uint8_t loraSendAttempts = 2; // Times
float R1 = 7501, R2 = 3002; // Voltage Sensor R values
// Please enter your sensitive data in the Secret tab or arduino_secrets.h
String appEui = SECRET_APP_EUI;
String appKey = SECRET_APP_KEY;
```



# My Code

- I defined a struct to have standard LoRaWAN messages (13 bytes):
  - alarms =
    - 00000000 for no alarms
    - 10000000 for noisy mode (blinking and emitting sounds)
    - 00100000 for low temperature alarm
    - 01000000 for high temperature alarm
    - 00010000 for low voltage alarm
    - 00001000 for position alarm
    - Last 3 bits are not currently used:  
They can be used in future to have additional alarms
  - Sensor reading values (changed into unsigned integers):
    - temp, volt, lat and lon

```
typedef struct {  
    uint8_t alarms;  
    uint16_t temp;  
    uint16_t volt;  
    uint32_t lat;  
    uint32_t lon;  
} msgLoraT;
```

## My Code

- Here is the loop:

```
checkUsbInMsg();
checkLoraInMsg(); // The input from the user (via Lora)
char msg[] = "0000"; // The state list (string): "noisy,
if(!lastInMsg.startsWith(stopCmd)) {
    updateCurrentTemp();
    updateCurrentVolt();
    updateCurrentLatAndLon();
    actualDelayTime = cycleDelayTime * 100; // Ok msg.
    manageAlarms(msg);
    manageCommands(msg);
    if(!stopUsbSending) {
        sendUsbMsg(msg);
    }
    if (!stopLoraSending && millis() >= loraNextTime) {
        uint8_t attempt = 1;
        boolean loraSent = sendLoraMsg(msg);
        while(loraSent && attempt < loraSendAttempts){
            loraSent = sendLoraMsg(msg);
            attempt ++;
        }
        if(loraSent) {
            updateLoraNextTime();
        }
    }
}
smartDelay(actualDelayTime);
```

# My Code

```
void manageCommands(char msg[]) {
    if(lastInMsg.startsWith(noisyCmd)) {
        lookAtMe();
        msg[0] = '1';
        actualDelayTime = 0;
    }
    else if(lastInMsg.startsWith(stopUsbSendingCmd)) {
        stopUsbSending = true;
        actualDelayTime = 0;
    }
    else if(lastInMsg.startsWith(startUsbSendingCmd)) {
        stopUsbSending = false;
        actualDelayTime = 0;
    }
    else if(lastInMsg.startsWith(stopLoraSendingCmd)) {
        stopLoraSending = true;
        actualDelayTime = 0;
    }
    else if(lastInMsg.startsWith(startLoraSendingCmd)) {
        stopLoraSending = false;
        actualDelayTime = 0;
    }
    else if(lastInMsg.startsWith(newDelayCmd)) {
        setCycleDelayTime(lastInMsg.substring(2).toFloat());
        actualDelayTime = 0;
    }
}
```

```
    else if(lastInMsg.startsWith(newLoraDelayCmd)) {
        setLoraDelayTime(lastInMsg.substring(3).toFloat());
        actualDelayTime = 0;
        lastInMsg = "";
    }
    else if(lastInMsg.startsWith(newMiddleTempCmd)) {
        setMiddleTemp(lastInMsg.substring(3).toFloat());
        actualDelayTime = 0;
        lastInMsg = "";
    }
    else if(lastInMsg.startsWith(newRadiusTempCmd)) {
        setRadiusTemp(lastInMsg.substring(3).toFloat());
        actualDelayTime = 0;
        lastInMsg = "";
    }
    else if(lastInMsg.startsWith(newMiddleLatCmd)) {
        setMiddleLat(lastInMsg.substring(3).toFloat());
        actualDelayTime = 0;
        lastInMsg = "";
    }
    else if(lastInMsg.startsWith(newRadiusLatCmd)) {
        setRadiusLat(lastInMsg.substring(3).toFloat());
        actualDelayTime = 0;
        lastInMsg = "";
    }
}
```

# My Code

```
void manageAlarms(char msg[]) {  
    if(currentTemp < middleTemp - radiusTemp) {  
        msg[1] = '1';  
        actualDelayTime = cycleDelayTime * 10;  
    }  
    if(currentTemp > middleTemp + radiusTemp) {  
        msg[1] = '2';  
        actualDelayTime = cycleDelayTime * 10;  
    }  
    if(currentVolt < minVolt) {  
        msg[2] = '1';  
        actualDelayTime = cycleDelayTime * 10;  
    }  
    if(currentLat < middleLat - radiusLat || middleLat > middleLat + radiusLat |  
        msg[3] = '1';  
        actualDelayTime = cycleDelayTime * 10;  
    }  
}
```

## My Code

```
void smartDelay(int timeToWait) {  
    unsigned long theEnd = timeToWait + millis();  
    while(millis() < theEnd && !Serial.available()){  
        delay(100);  
    }  
}
```

# My Code

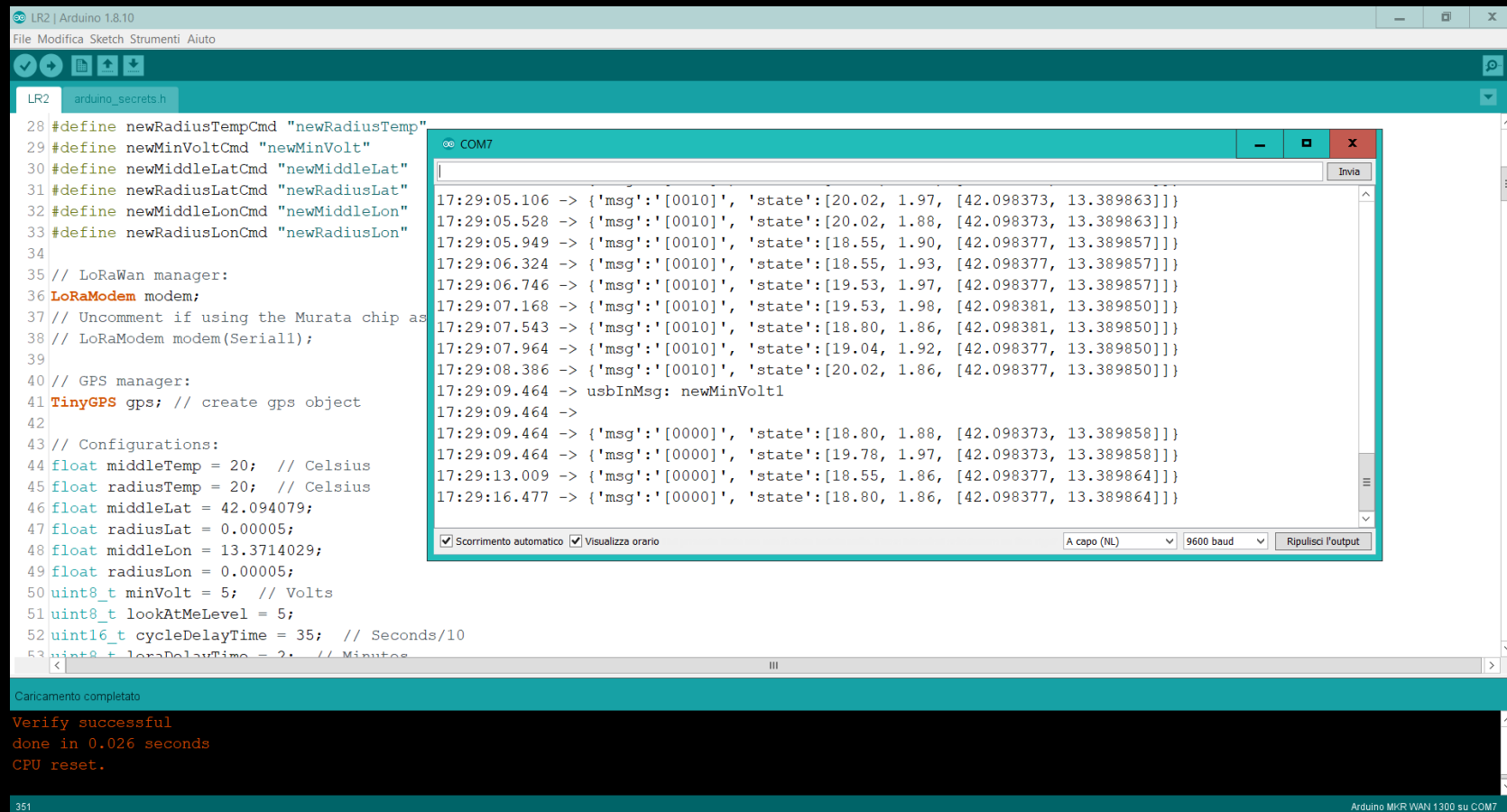
- For USB messages, I chose JSON format; LoRaWAN messages are bytes arrays:

```
void sendUsbMsg(char msg[]) {  
    Serial.print("{ 'msg': '[');  
    Serial.print(msg);  
    Serial.print("] ', 'state': [");  
    Serial.print(currentTemp);  
    Serial.print(", ");  
    Serial.print(currentVolt);  
    Serial.print(", ");  
    Serial.print("[");  
    Serial.print(currentLat, 6);  
    Serial.print(", ");  
    Serial.print(currentLon, 6);  
    Serial.print("]");  
    Serial.println("]}");  
}
```

```
boolean sendLoraMsg(char msg[]) {  
    msgLoraT loraMsg;  
    loraMsg.alarms = (msg[0] << 7) | (msg[1] << 5) | (msg[2] << 4) | msg[3];  
    loraMsg.temp = (int)((currentTemp + 20) * 100);  
    loraMsg.volt = (int)(currentVolt * 100);  
    loraMsg.lat = (int)((currentLat + 90) * 1000000);  
    loraMsg.lon = (int)((currentLon + 180) * 1000000);  
    int err;  
    modem.beginPacket();  
    modem.write(byte(loraMsg.alarms));  
    modem.write((byte*)&loraMsg.temp, 2);  
    modem.write((byte*)&loraMsg.volt, 2);  
    modem.write((byte*)&loraMsg.lat, 4);  
    modem.write((byte*)&loraMsg.lon, 4);  
    err = modem.endPacket(true);  
    if (err > 0) {  
        Serial.println("Message sent correctly via Lora!"); // Test  
        return true;  
    } else {  
        // Test  
        Serial.println("Error sending message via Lora:");  
        Serial.println("(you may send a limited amount of messages per minute,  
        Serial.println("it may vary from 1 message every couple of seconds to 1  
        return false;  
    }  
}
```

# My Code

- Let it run...



```
28 #define newRadiusTempCmd "newRadiusTemp"
29 #define newMinVoltCmd "newMinVolt"
30 #define newMiddleLatCmd "newMiddleLat"
31 #define newRadiusLatCmd "newRadiusLat"
32 #define newMiddleLonCmd "newMiddleLon"
33 #define newRadiusLonCmd "newRadiusLon"
34
35 // LoRaWAN manager:
36 LoRaModem modem;
37 // Uncomment if using the Murata chip as
38 // LoRaModem modem(Serial1);
39
40 // GPS manager:
41 TinyGPS gps; // create gps object
42
43 // Configurations:
44 float middleTemp = 20; // Celsius
45 float radiusTemp = 20; // Celsius
46 float middleLat = 42.094079;
47 float radiusLat = 0.00005;
48 float middleLon = 13.3714029;
49 float radiusLon = 0.00005;
50 uint8_t minVolt = 5; // Volts
51 uint8_t lookAtMeLevel = 5;
52 uint16_t cycleDelayTime = 35; // Seconds/10
53 uint8_t loraDelayTime = 2; // Minutes
```

COM7

```
17:29:05.106 -> {'msg':'[0010]', 'state':[20.02, 1.97, [42.098373, 13.389863]]}
17:29:05.528 -> {'msg':'[0010]', 'state':[20.02, 1.88, [42.098373, 13.389863]]}
17:29:05.949 -> {'msg':'[0010]', 'state':[18.55, 1.90, [42.098377, 13.389857]]}
17:29:06.324 -> {'msg':'[0010]', 'state':[18.55, 1.93, [42.098377, 13.389857]]}
17:29:06.746 -> {'msg':'[0010]', 'state':[19.53, 1.97, [42.098377, 13.389857]]}
17:29:07.168 -> {'msg':'[0010]', 'state':[19.53, 1.98, [42.098381, 13.389850]]}
17:29:07.543 -> {'msg':'[0010]', 'state':[18.80, 1.86, [42.098381, 13.389850]]}
17:29:07.964 -> {'msg':'[0010]', 'state':[19.04, 1.92, [42.098377, 13.389850]]}
17:29:08.386 -> {'msg':'[0010]', 'state':[20.02, 1.86, [42.098377, 13.389850]]}
17:29:09.464 -> usbInMsg: newMinVolt1
17:29:09.464 ->
17:29:09.464 -> {'msg':'[0000]', 'state':[18.80, 1.88, [42.098373, 13.389858]]}
17:29:09.464 -> {'msg':'[0000]', 'state':[19.78, 1.97, [42.098373, 13.389858]]}
17:29:13.009 -> {'msg':'[0000]', 'state':[18.55, 1.86, [42.098377, 13.389864]]}
17:29:16.477 -> {'msg':'[0000]', 'state':[18.80, 1.86, [42.098377, 13.389864]]}
```

Scorrimto automatico Visualizza orario A capo (NL) 9600 baud Ripulisci l'output

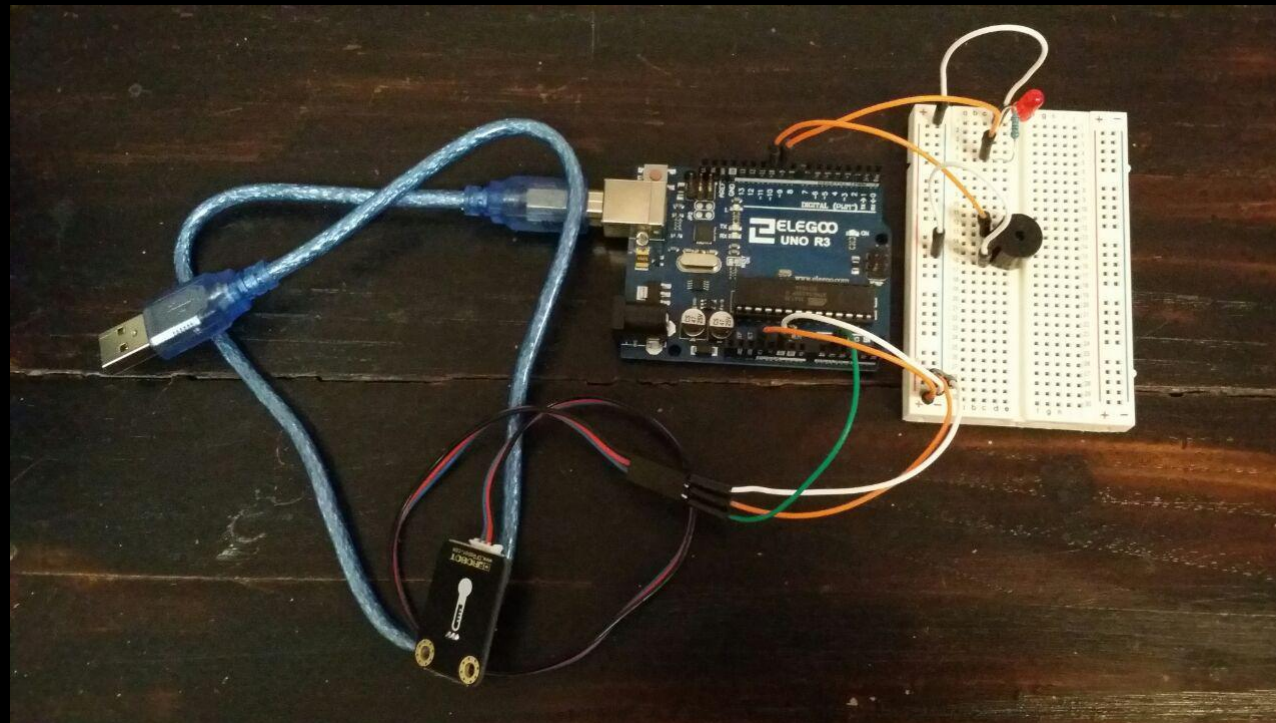
Caricamento completato

Verify successful  
done in 0.026 seconds  
CPU reset.

351 Arduino MKR WAN 1300 su COM7

# Tests

In a first time, I had to face with delivery delay... so I started testing the very first features on a board I already had:





# Tests

Then stuff arrived, piece by piece, and I learned new interesting things (GPS sensor arrived with no pins)...



# Tests

Finally, LoRaWAN connection tests... In my first approach, I decided to send a string (the same content of USB messages), because it's more readable:

The screenshot displays two web interfaces side-by-side. The left interface is 'The Things Network Console', showing the 'APPLICATION DATA' section for device 'mkr1300\_a803'. It lists several data points with timestamps, counters, and payloads. The right interface is 'Code Beautify', specifically the 'Hex to String (Hex to Text)' tool. The hex input field contains the payload from the console: '7B276D7367273A275B3030305D272C20277374617465273A5B392E37372C20302E33322C205B34322E3335343636382C2031332E3431333637395D57D0D0A'. The 'Convert' button is highlighted, and the decoded string is shown in the output field: `{'msg': '[0000]', 'state': [9.77, 0.32, [42.354668, 13.413679]]}`.

**The Things Network Console - APPLICATION DATA**

time	counter	port	dev addr	app eui	dev eui	payload
15:46:16			26 01 24 21	70 B3D57E D002 72 B3	A8 61 0A 32 33 2B 93 03	
15:44:53		0				
15:44:52	2	2	retry confirmed			7B 27 6D 73 67 27 3A 27 5B 30 30 30 5D 27 2C 20 27 73 74 61 74 65 27 3A 5B 39 2E 37 37 2C
15:44:27		0				
15:44:26	2	2	retry confirmed			7B 27 6D 73 67 27 3A 27 5B 30 30 30 5D 27 2C 20 27 73 74 61 74 65 27 3A 5B 39 2E 37 37 2C
15:44:13		0				
15:44:12	2	2	retry confirmed			7B 27 6D 73 67 27 3A 27 5B 30 30 30 5D 27 2C 20 27 73 74 61 74 65 27 3A 5B 39 2E 37 37 2C
15:43:59		0				
15:43:57	2	2	confirmed			7B 27 6D 73 67 27 3A 27 5B 30 30 30 5D 27 2C 20 27 73 74 61 74 65 27 3A 5B 39 2E 37 37 2C
15:43:33		0				

**Code Beautify - Hex to String (Hex to Text)**

Enter the hexadecimal text to decode

7B276D7367273A275B3030305D272C20277374617465273A5B392E37372C20302E33322C205B34322E3335343636382C2031332E3431333637395D57D0D0A

Convert Load Browse

The decoded string:

```
{'msg': '[0000]', 'state': [9.77, 0.32, [42.354668, 13.413679]]}
```

# Tests

Then, I defined a different structure for LoRaWAN messages, to make them lighter (13 bytes):

The screenshot displays the The Things Network Console interface. The main panel shows the 'Payload Formats' configuration for an application named 'lr2'. A custom JavaScript function is defined to decode a 13-byte payload. The function extracts fields like noiseAlarm, tempAlarm, voltAlarm, posAlarm, temp, volt, lat, and lon. Below the code, the 'Payload' field shows the hexadecimal value '01 13 24 00 20 05 5D 4A 80 00 00 00 00'. The 'Decoded' field shows the resulting JSON object: 

```
{ "lat": 0, "lon": 0, "noiseAlarm": 0, "posAlarm": 1, "temp": 29, "tempAlarm": 0, "volt": 0.32, "voltAlarm": 0 }
```

. On the right, a terminal window shows the serial output of the device, confirming the message was sent correctly via LoRa.

```
5 decoded.noiseAlarm = (bytes[0] & 0x00) // 1;  
6 decoded.tempAlarm = (bytes[0] & 0x60) >> 5;  
7 decoded.voltAlarm = (bytes[0] & 0x10) >> 4;  
8 decoded.posAlarm = bytes[0] & 0x0F;  
9 decoded.temp = (((bytes[1] << 8) | bytes[2]) / 100.0;  
10 decoded.volt = (((bytes[3] << 8) | bytes[4]) / 100.0;  
11 decoded.lat = (((bytes[5] << 24) | (bytes[6] << 16)  
12 decoded.lon = (((bytes[9] << 24) | (bytes[10] << 16)  
13  
14 // if (port === 1) decoded.led = bytes[0];  
15  
16 return decoded;  
17 }
```

**Payload**

```
01 13 24 00 20 05 5D 4A 80 00 00 00 00
```

```
{  
  "lat": 0,  
  "lon": 0,  
  "noiseAlarm": 0,  
  "posAlarm": 1,  
  "temp": 29,  
  "tempAlarm": 0,  
  "volt": 0.32,  
  "voltAlarm": 0  
}
```

**COM7**

```
20:07:42.742 -> usbInMsg: startLoraSending  
20:07:42.742 ->  
20:07:42.742 -> {'msg':'[0001]', 'state':[29.00, 0.33, [0.000000, 0.000000]]}  
20:07:42.742 -> 1  
20:07:42.742 -> 1324  
20:07:42.742 -> 20  
20:07:42.742 -> 55D4A80  
20:07:42.742 -> 0  
20:07:42.776 -> Message sent correctly via Lora!
```

**example2.txt - Blocco note**

```
20:07:42.742 -> {'msg':'[0001]', 'state':[29.00, 0.33, [0.000000, 0.000000]]}  
20:07:42.742 -> 1  
20:07:42.742 -> 1324  
20:07:42.742 -> 20  
20:07:42.742 -> 55D4A80  
20:07:42.742 -> 0  
  
01 1324 0020 055D4A80 00000000
```

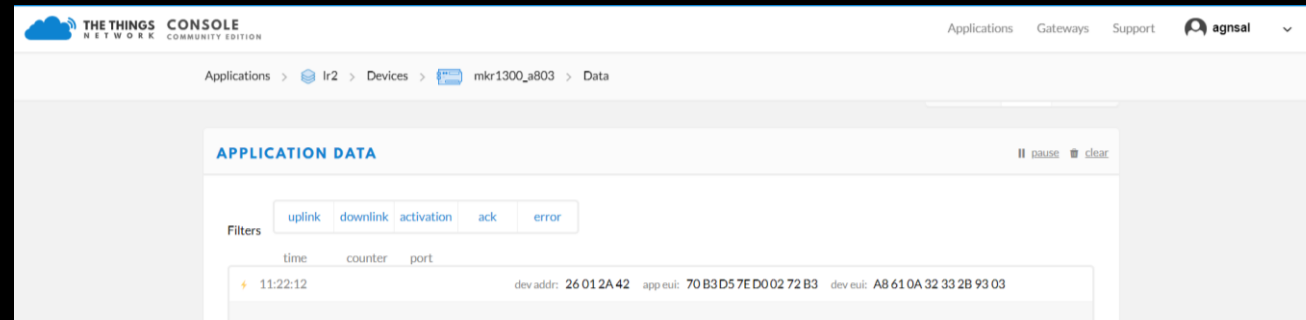
Windows (CRLF) Linea 39, colonna 1

Payload was valid

# Tests

Finally, I tested again LoRaWAN communication with new message structure and smaller (pre-defined) command strings in a different city area (Economics Faculty):

- I couldn't send any message after connecting to LoRaWAN gateway, even if the device had its handshake with the gateway



... finally, I asked the gateway owner if it was currently working and I discovered Fab Lab gateway was not able to connected to the Internet that day:

- My device was connected to the gateway, but it had no access to TTN

# Tests

So I tried in a different area, served by the other gateway (Amitemnum Hotel):

- Then, I could connect to TTN, but I found out data were strange:

The screenshot displays the The Things Network Console interface. On the left, the 'APPLICATION DATA' section shows a list of data points with columns for time, counter, and port. The main area is divided into two panes. The left pane shows a terminal window for device 'mkr1300\_a803' with the following log entries:

```
11:58:51.707 -> Your module version is: ARD-078 1.1.5
11:58:51.755 -> Your device EUI is: a8610a32332b9303
11:59:52.828 -> Something went wrong; are you indoor? Move near a window and retry
12:00:48.541 -> {'msg': '[0000]', 'state': [32.87, 0.21, [42.360493, 13.376807]]}
```

The right pane shows the 'Payload Formats' section for the 'Ir2' application. It displays a decoded payload with the following JSON structure:

```
{
  "lat": 42.360471999999999,
  "lon": 13.376815999999999,
  "noiseAlarm": 1,
  "posAlarm": 0,
  "temp": 27.39,
  "tempAlarm": 1,
  "volt": 0.32,
  "voltAlarm": 1
}
```

Below the JSON, the 'Payload' section shows the raw hex data: 'B0 83 12 20 00 18 A9 E3 07 30 B2 86 0B'. A 'Test' button is visible next to the hex data. The bottom of the screen shows a 'no changes to save' message.

# Tests

So I tried in a different area, served by the other gateway (Amiternum Hotel):

- The problem is that I was receiving bytes in inverse order, so I solved it by changing a little the decoder function:

The screenshot displays two side-by-side views of the The Things Network Console. The left view shows the 'Data' page for device 'mkr1300\_a803'. It lists three data points. The third point, at 12:07:21, is expanded to show its 'Uplink' details. The 'Payload' field contains the hex string 'B0 83 12 20 00 18 A9 E3 07 30 B2 86 0B'. The 'Fields' section shows a JSON object with various sensor readings, including latitude (323.786887), longitude (637.006091), and several alarm status flags. The right view shows the 'Payload Formats' page for the same device. It features a JavaScript decoder function that takes a byte array and returns a decoded object. The function includes logic for parsing noise, temperature, voltage, position, and temperature alarms. Below the decoder, the 'Payload' field is populated with the same hex string 'B0 83 12 20 00 18 A9 E3 07 30 B2 86 0B', and a 'Test' button is visible. A status message at the bottom right indicates 'Payload was valid'.

Applications > lr2 > Devices > mkr1300\_a803 > Data

Filters: [uplink](#) [downlink](#) [activation](#) [ack](#) [error](#)

time	counter	port	payload
12:07:42	1	scheduled	payload: 6E 6F 69 73 79
12:07:20	0		
12:07:21	1	2	retry confirmed payload: B0 83 12 20 00 18 A9 E3 07 30 B2 86 0B lat: 323.786887 lon: 637.006091 noiseAlarm: 1

**Uplink**

**Payload**

B0 83 12 20 00 18 A9 E3 07 30 B2 86 0B

**Fields**

```
{
  "lat": 323.786887,
  "lon": 637.006091,
  "noiseAlarm": 1,
  "posAlarm": 0,
  "temp": 315.54,
  "tempAlarm": 1,
  "volt": 81.92,
  "voltAlarm": 1
}
```

**Metadata**

```
{
  "lat": 42.36847199999999,
  "lon": 13.376815999999999,
  "noiseAlarm": 1,
  "posAlarm": 0,
  "temp": 27.39,
  "tempAlarm": 1,
  "volt": 0.32,
  "voltAlarm": 1
}
```

Applications > lr2 > Payload Formats

```
> decoded.noiseAlarm = (bytes[0] & 0x0F) >> 7;
6 decoded.tempAlarm = (bytes[0] & 0x60) >> 5;
7 decoded.voltAlarm = (bytes[0] & 0x10) >> 4;
8 decoded.posAlarm = bytes[0] & 0x0F;
9 decoded.temp = (((bytes[2] << 8) | bytes[1]) / 100.0) - 20.0;
10 decoded.volt = (((bytes[4] << 8) | bytes[3]) / 100.0);
11 decoded.lat = (((bytes[8] << 24) | (bytes[7] << 16) | (bytes[6] << 8) | bytes[5]) / 1000000.0) - 90.0;
12 decoded.lon = (((bytes[12] << 24) | (bytes[11] << 16) | (bytes[10] << 8) | bytes[9]) / 1000000.0) - 180.0;
13
14 // if (port === 1) decoded.led = bytes[0];
15
16 return decoded;
17 }
```

decoder has no changes

**Payload**

B0 83 12 20 00 18 A9 E3 07 30 B2 86 0B 13 bytes 1 Test

Payload was valid

Cancel no changes to save

# Tests

So I tried in a different area, served by the other gateway (Amiternum Hotel):

- Downlink messages work fine with Arduino MKR 1300 (it supports class C mode):

The image shows two side-by-side browser windows. The left window displays the 'Code Beautify' website, specifically the 'String to Hex (Text to Hex)' converter. The input field contains the text 'noisy', and the output field shows the encoded string '6e6f697379'. The right window shows the 'The Things Network Console' for a device named 'mkr1300\_a803'. It displays the 'DOWNLINK' section with a 'Scheduling' dropdown set to 'replace', a 'FPort' of 1, and a 'Payload' field containing the hex string '6E 6F 69 73 79'. A 'Send' button is visible at the bottom right of the console interface.

**Code Beautify**

String to Hex (Text to Hex)☆

Enter the text to encode to hex

noisy

Convert Load Browse

The encoded string:

6e6f697379

**THE THINGS NETWORK CONSOLE**

Applications > Ir2 > Devices > mkr1300\_a803

Frames up 1 [reset frame counters](#)

Frames down 2

**DOWNLINK**

Scheduling: [replace](#) first last

FPort: 1 ☐ Confirmed

Payload: [bytes](#) [fields](#) 6E 6F 69 73 79 5 bytes

Send

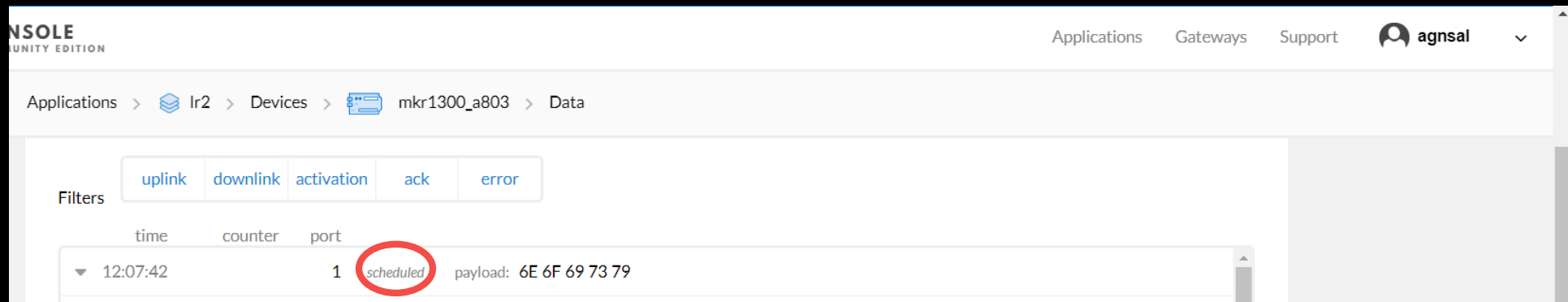
**SIMULATE UPLINK**

FPort Payload

# Tests

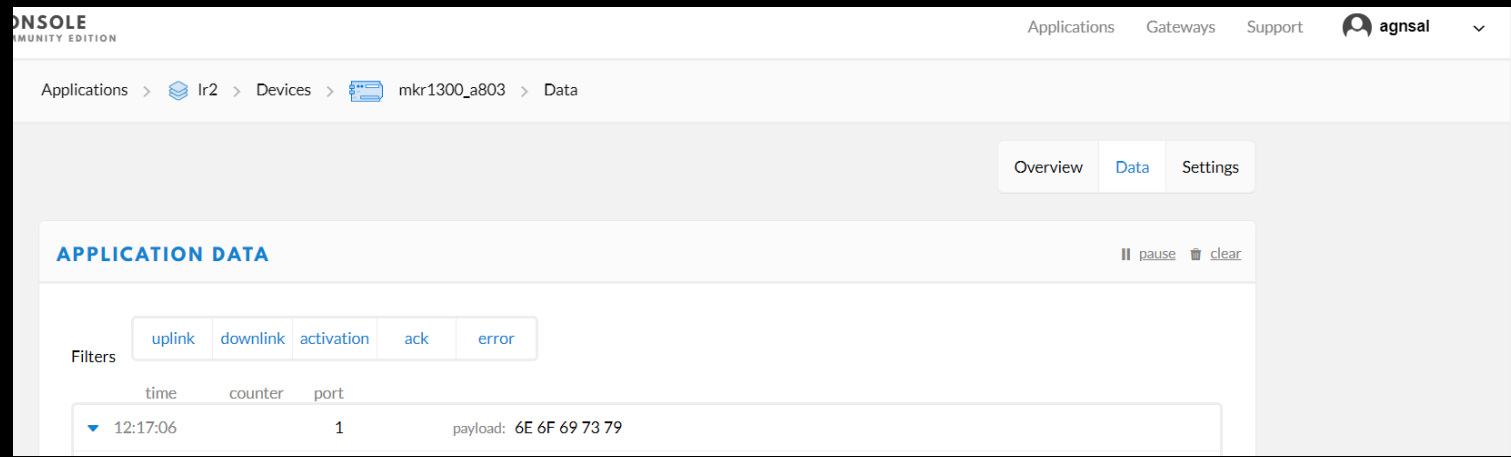
So I tried in a different area, served by the other gateway (Amiternum Hotel):

- Downlink messages work fine with Arduino MKR 1300 (that support class C mode):



The screenshot shows the LoRaWAN console interface for the device 'mkr1300\_a803'. The breadcrumb navigation is 'Applications > Ir2 > Devices > mkr1300\_a803 > Data'. The 'Filters' section has buttons for 'uplink', 'downlink', 'activation', 'ack', and 'error'. The 'downlink' button is selected. Below the filters, a table displays message data with columns 'time', 'counter', 'port', and 'payload'. The first row shows a message at '12:07:42' with counter '1', port '1', and payload '6E 6F 69 73 79'. The status 'scheduled' is circled in red.

time	counter	port	payload
12:07:42	1	1	6E 6F 69 73 79



The screenshot shows the 'Data' tab of the LoRaWAN console for the device 'mkr1300\_a803'. The breadcrumb navigation is 'Applications > Ir2 > Devices > mkr1300\_a803 > Data'. The 'Overview', 'Data', and 'Settings' tabs are visible, with 'Data' selected. The 'APPLICATION DATA' section has a 'pause' button and a 'clear' button. The 'Filters' section has buttons for 'uplink', 'downlink', 'activation', 'ack', and 'error'. The 'downlink' button is selected. Below the filters, a table displays message data with columns 'time', 'counter', 'port', and 'payload'. The first row shows a message at '12:17:06' with counter '1', port '1', and payload '6E 6F 69 73 79'.

time	counter	port	payload
12:17:06	1	1	6E 6F 69 73 79

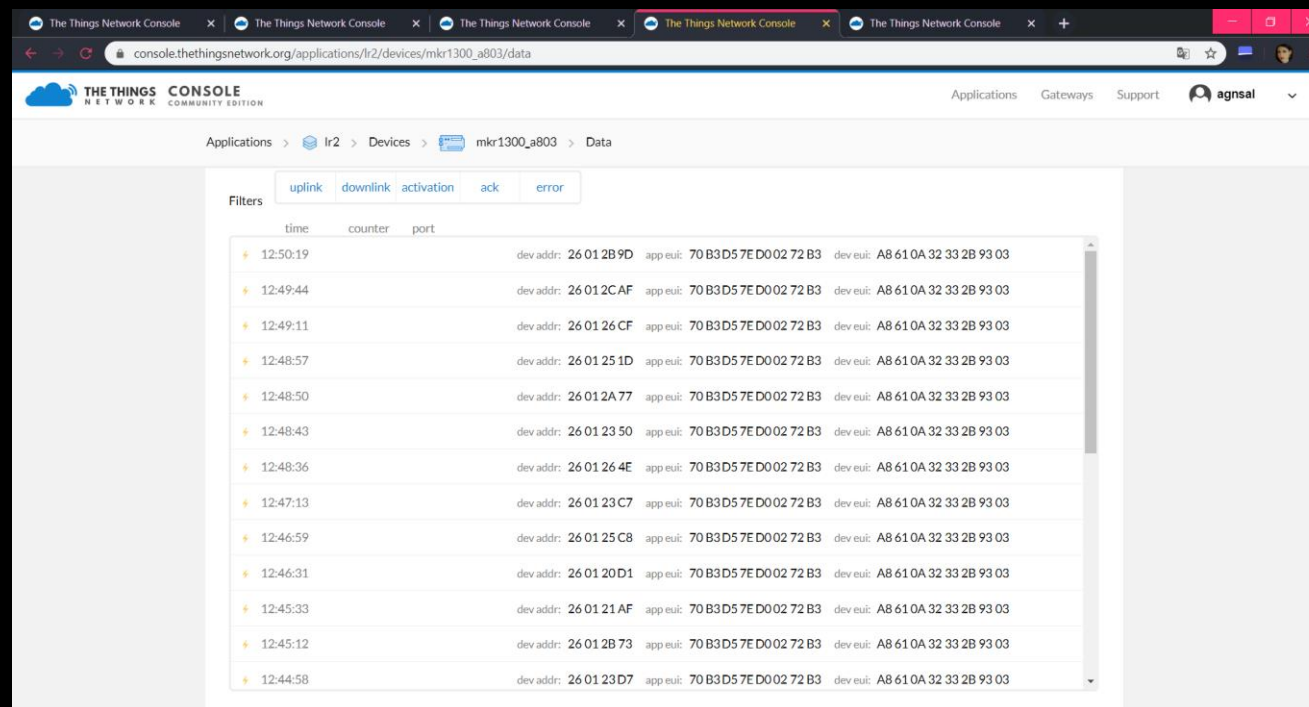


# Tests

- Downlink messages, and TTN service in general, is not very efficient and it depends on the number of connected devices too
- The range of the gateway depends on the gateway type
- LoRaWAN protocol works fine with little traffic, but may have problems related to ACKs when a lot of devices are connected

# Tests

- When a device finishes its daily amount of messages, it can send connection requests to the gateway, but it won't be able to send messages



The screenshot shows the 'Data' tab for device 'mkr1300\_a803' in the The Things Network Console. The interface includes a breadcrumb trail: Applications > Ir2 > Devices > mkr1300\_a803 > Data. Below the breadcrumb, there are tabs for 'uplink', 'downlink', 'activation', 'ack', and 'error', with 'uplink' currently selected. A 'Filters' section is visible above the data table. The data table has columns for 'time', 'counter', and 'port'. The 'time' column contains timestamps from 12:44:58 to 12:50:19. The 'counter' column contains hexadecimal values ranging from 26 01 23 D7 to 26 01 2B 9D. The 'port' column contains three sets of hexadecimal addresses: dev addr, app eui, and dev eui, all of which are consistent across all rows.

time	counter	port
12:50:19	26 01 2B 9D	dev addr: 26 01 2B 9D app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:49:44	26 01 2C AF	dev addr: 26 01 2C AF app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:49:11	26 01 26 CF	dev addr: 26 01 26 CF app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:48:57	26 01 25 1D	dev addr: 26 01 25 1D app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:48:50	26 01 2A 77	dev addr: 26 01 2A 77 app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:48:43	26 01 23 50	dev addr: 26 01 23 50 app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:48:36	26 01 26 4E	dev addr: 26 01 26 4E app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:47:13	26 01 23 C7	dev addr: 26 01 23 C7 app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:46:59	26 01 25 C8	dev addr: 26 01 25 C8 app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:46:31	26 01 20 D1	dev addr: 26 01 20 D1 app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:45:33	26 01 21 AF	dev addr: 26 01 21 AF app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:45:12	26 01 2B 73	dev addr: 26 01 2B 73 app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03
12:44:58	26 01 23 D7	dev addr: 26 01 23 D7 app eui: 70 B3 D5 7E D0 02 72 B3 dev eui: A8 61 0A 32 33 2B 93 03

# Tests

For my tests, I used TTN console, because I was not interested on the real client application with its GUI for this project.

If you want to develop a complete client application, you can connect your TTN application to an external client application with your TTN console.

For example, you could use Node-RED for fast and easy developing (<https://nodered.org/>).

# References

- Capire e Usare LoRa e LoRaWAN: creare una rete LoRaWAN per IoT, di Pier Calderan (<https://www.amazon.it/LoRaWAN-Progetto-dispositivo-ATmega328-Raspberry/dp/8869283232>)
- <https://www.thethingsnetwork.org/forum/t/limitations-data-rate-packet-size-30-seconds-uplink-and-10-messages-downlink-per-day-fair-access-policy-guidelines/1300>
- <https://blog.arduino.cc/2017/09/25/introducing-the-arduino-mkr-wan-1300-and-mkr-gsm-1400/>
- <https://www.arduino.cc/reference/en/language/functions/communication/serial/>
- <https://www.danielealberti.it/2016/04/misurare-una-batteria-con-arduino.html>

Thank you for your attention

