

Deploying an End-to-End LoRaWAN Platform

Starting from September 2016, Saint-Joseph University of Beirut (USJ) will be deploying the first academic LoRa network in Lebanon. The network will support monitoring of micro-climate conditions in vineyards. Here below you can find a detailed description of the experimental platform implementing an end-to-end LoRaWAN solution. The platform consists of the following elements:

- Devices that communicate to one or more gateways via a wireless interface using single hop LoRa and implementing the LoRaWAN protocol. These devices are physically connected to sensors that generate data.
- Gateways or base stations that forward frames between the devices and the network server. Gateways are connected to the network server via IP interfaces.
- A LoRaWAN backend that implements the network server functions and provides frame control and security.
- Applications that enable to visualize and store the sensor data obtained from the devices.

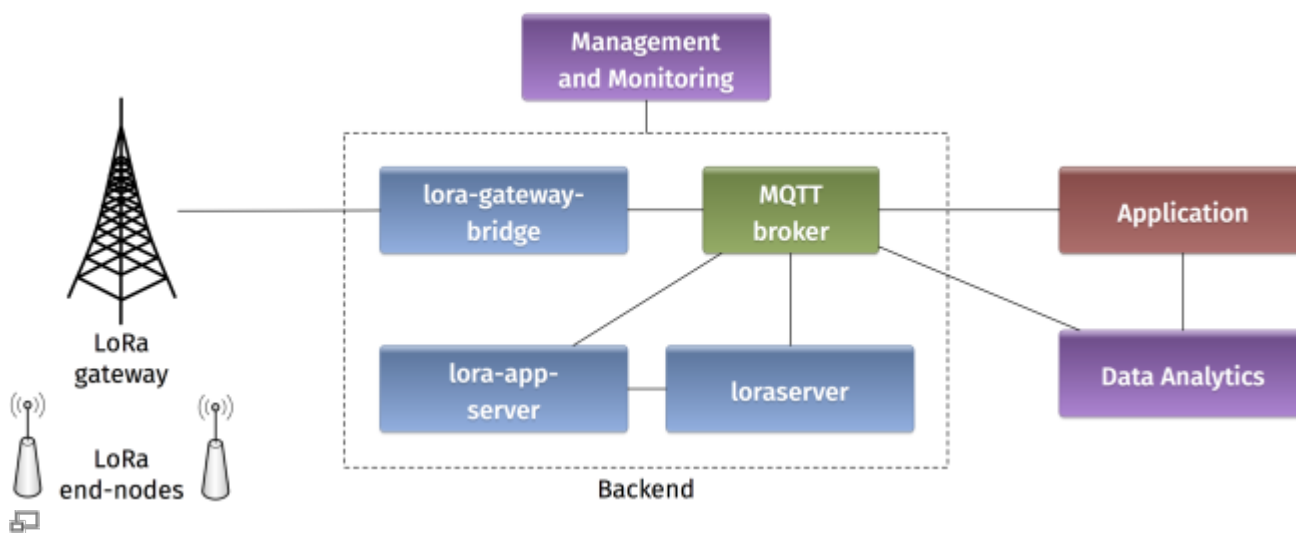


Figure 2. Architecture of the LoRaWAN Platform

- . Devices

- . Autonomo with LoRaBee

Starting with the devices in the LoRaWAN platform, we will use an [Autonomo](#) board with a LoRaBee Microchip RN2483 module. According to <http://shop.sodaq.com>, Autonomo is a matchbox-sized powerhouse which uses the new Atmel Cortex M0+ 32bit micro controller. One advantage of such device is that it can be powered by a smartphone-sized solar panel.

In order to configure the Autonomo with LoRaBee device, you should follow these steps:

1. Verify that you have the latest Arduino IDE from <https://www.arduino.cc/en/Main/Software> on your computer.
2. Install the board files as noted in <http://support.sodaq.com/sodaq-one/autonomo/getting-started-autonomo/>.
3. Add the following library

sodaq_rn2483_2.zip

to your Arduino IDE as explained in <https://www.arduino.cc/en/guide/libraries>.

Now you are ready to write a sketch for the device. Here is one example sketch

test-lorawan-combined-loraserver-example.zip

where the autonomo is connected to three sensors: light, moisture, and temperature. Let us analyse some extracts of the code.

In this part, you should put the keys for Over-The-Air Activation (OTAA) as explained in the

LoRaWAN specification

:

```
// USE YOUR OWN KEYS!  
const uint8_t devEUI[8] =  
{ };  
  
// USE YOUR OWN KEYS!  
const uint8_t appEUI[8] =  
{ };  
  
const uint8_t appKey[16] =  
{ };
```

The pins for connecting the sensors are specified in these declarations (A0 for light sensor, A2 for moisture sensor, and D0 temperature sensor):

```
int light_pin = A0;  
int moisture_pin = A2;  
int temperature_pin = 0;  
int temperature_vcc_pin = 1;  
int moisture_vcc_pin = 8;  
int moisture_gnd_pin = 7;
```

The OTAA method is used for joining the network and Adaptive Data Rate (ADR) is activated:

```
LoRaBee.initOTA(loraSerial, devEUI, appEUI, appKey, true)
```

Eight different sub channels are activated with data rate ranges from 0 to 5:

```
LoRaBee.configChFreq(0, 868100000L, 0, 5, 1);  
LoRaBee.configChFreq(1, 868300000L, 0, 5, 1);  
LoRaBee.configChFreq(2, 868500000L, 0, 5, 1);  
LoRaBee.configChFreq(3, 867100000L, 0, 5, 1);  
LoRaBee.configChFreq(4, 867300000L, 0, 5, 1);  
LoRaBee.configChFreq(5, 867500000L, 0, 5, 1);  
LoRaBee.configChFreq(6, 867700000L, 0, 5, 1);  
LoRaBee.configChFreq(7, 867900000L, 0, 5, 1);
```

Finally, the message containing the sensor values is sent in an unconfirmed uplink message:

```
LoRaBee.send(1, (uint8_t*)message.c_str(), message.length())
```

-. Arduino with Dragino Shield

-. Periodic Message Sending

Devices in the LoRaWAN platform can also be implemented on Arduino boards with Dragino shields. The combined module as well as the basic configuration steps are presented in [Simple Prototype of LoRa Communications](#). Similarly to the Autonomo device, you can download the following sketch

[test-loraserver-comb-loraserver-dragino.zip](#)

and modify it according to your preferences. Below you can find some commented extracts of the sketch.

The pin mapping corresponds to the Dragino electronic schematic:

```
const lmic_pinmap lmic_pins = {
  .nss = 10,
  .rxtx = LMIC_UNUSED_PIN,
  .rst = 9,
  .dio = {2, 6, 7},
};
```

The send function is rescheduled TX_INTERVAL seconds after each transmission complete event:

```
    case EV_TXCOMPLETE:
      Serial.println(F("EV_TXCOMPLETE (includes waiting for RX
windows)"));
      if(LMIC.dataLen) {
        // data received in rx slot after tx
        Serial.print(F("Data Received: "));
        Serial.write(LMIC.frame+LMIC.dataBeg, LMIC.dataLen);
        Serial.println();
      }
      // Schedule next transmission
      os_setTimedCallback(&sendjob,
os_getTime()+sec2osticks(TX_INTERVAL), do_send);
      break;
```

The send function is initially scheduled here:

```
do_send(&sendjob);
```

The message containing the sensor values is transmitted on one of the radio channels (as in the Autonomo case):

```
LMIC_setTxData2(1, (uint8_t*) buffer, message.length() , 0);
```

The adaptive data rate is not supported, and the spreading factor is configured as follows:

```
LMIC_setDrTxpow(DR_SF7, 14);
```

- . Triggered Message Sending

You can also find another example of sketch to download:

[test-loraserver-moisture-on-move.ino.zip](#)

. Here the message sending is not periodic but related to an event. For example, an infrared sensor detects a movement and triggers a signal for the device to send a LoRaWAN message. Note also that the join method used in this second sketch is Activation by Personalisation (ABP): the device address, the network session key, and the application session key are directly configured on the device.

- . Gateways

- . Single Channel Gateway

The single channel gateway includes a LoRa transmission module (Dragino Shield) connected to a Raspberry Pi (2 or 3) as shown in Figure 1. Communication between the two modules is done over an SPI interface.

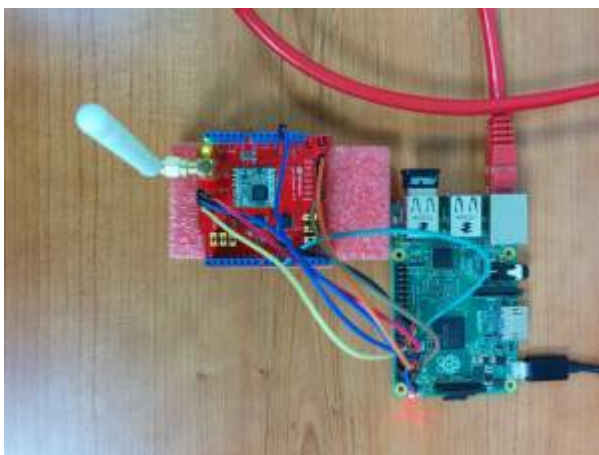


Figure 2. LoRa single channel gateway

In order to assemble the gateway, start by making the wire connections: the connection pins are identified in Figures 2 and 3.

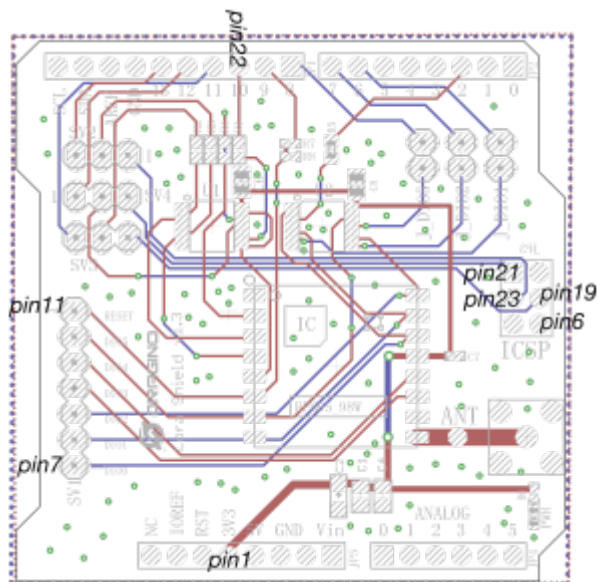


Figure 2. Dragino pin mapping

Pin#	NAME	NAME	Pin#
01	3.3v DC Power	DC Power 5v	02
03	GPIO02 (SDA1 , I ² C)	DC Power 5v	04
05	GPIO03 (SCL1 , I ² C)	Ground	06
07	GPIO04 (GPIO_GCLK)	(TXD0) GPIO14	08
09	Ground	(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)	(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)	Ground	14
15	GPIO22 (GPIO_GEN3)	(GPIO_GEN4) GPIO23	16
17	3.3v DC Power	(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)	Ground	20
21	GPIO09 (SPI_MISO)	(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)	(SPI_CE0_N) GPIO08	24
25	Ground	(SPI_CE1_N) GPIO07	26
27	ID_SD (I ² C ID EEPROM)	(I ² C ID EEPROM) ID_SC	28
29	GPIO05	Ground	30
31	GPIO06	GPIO12	32
33	GPIO13	Ground	34
35	GPIO19	GPIO16	36
37	GPIO26	GPIO20	38
39	Ground	GPIO21	40

Figure 3. Raspberry pi 3 pins

Connect the Raspberry Pi to the Internet and install the packet forwarding software. The source code of the single channel packet forwarder is available on:

https://github.com/samerlahoud/single_chan_pkt_fwd. In order to install it, you need to:

- Enable SPI on the Raspberry Pi using raspi-config
- Download and unzip the source code:

```
wget https://github.com/hallard/single_chan_pkt_fwd/archive/master.zip
unzip master.zip
```

- Install the wiring library:

```
apt-get update
apt-get install wiring
```

Compile the packet forwarder:

```
make all
```

For gcc version 4.6.3, a compilation error results in the following warning unrecognized command line option '-std=c++11'. Replace -std=c++11 by -std=c++0x in the Makefile and recompile:

```
CFLAGS = -std=c++0x -c -Wall -I include/
```

Now, you need to configure the single channel packet forwarder. This is done in the

global_config.json.zip configuration file. Particularly, you need to choose the channel, the spreading factor, the pins for SPI communication, and the address of the backend server. Note that you can specify multiple backends for testing purposes.

Finally, you can run the packet forwarder as root!

```
nohup ./single_chan_pkt_fwd &
```

-. Kerlink IoT Station

```
# activates eth0 at startup
ETHERNET=yes
# claims dhcp request on eth0
ETHDHCP=yes

# Selector operator APN
GPRSAPN=gprs.touch.com.lb
# Enter pin code if activated
GPRSPIN=0000
# Update /etc/resolv.conf to get dns facilities
GPRSDNS=yes
# PAP authentication
GPRSUSER=
GPRSPASSWORD=

# Bearers priority order
#BEARERS_PRIORITY="eth0,ppp0,eth1"
BEARERS_PRIORITY="ppp0,eth0,eth1"
```

```
./gps-pkt-fwd.sh > /dev/null &
```

```
3270 root      2548 S    /bin/sh ./gps-pkt-fwd.sh
3288 root      34908 S   ./gps_pkt_fwd
```

```
/etc/init.d/gprs start
```

```
[root@Wirgrid_0b03008c demo_gps_loramote]# /etc/init.d/gprs status
pppd (pid 5273) is running...
Session: Rx=58, Tx=163
```

```
Globals: Rx=1130457, Tx=1195592
Sum:      Rx=1130515, Tx=1195755
[root@Wirgrid_0b03008c demo_gps_loramote]#
```

- . Backend

- . Loraserver

- . The Things Network

- . Applications

- . mqtt-spy

mqtt-spy is an open source utility intended to help you with monitoring activity on MQTT topics. It has been designed to deal with high volumes of messages, as well as occasional publications. mqtt-spy is a JavaFX application, so it should work on any operating system with an appropriate version of Java 8 installed. A very useful tutorial is available on <https://github.com/eclipse/paho.mqtt-spy/wiki>. You can use mqtt-spy to debug the messages received from the LoRaWAN devices. For this, you should download the software tool from <https://github.com/eclipse/paho.mqtt-spy/wiki>. After starting the application, configure a new connection to the MQTT broker by simply adding the IP address of the broker in the Server URI field. Now you can subscribe to any MQTT topic. If you want to receive all messages arriving at the backend, you can use the generic topic #. You can also limit to the topic including the messages of any specific device:
application/APPLICATION_ID/node/DEVICE_EUI/rx.

- . Emoncms

From:

<http://wiki.lahoud.fr/> - **wikiroute**

Permanent link:

http://wiki.lahoud.fr/doku.php?id=deploying_lorawan&rev=1493735719

Last update: **2017/05/02 16:35**

