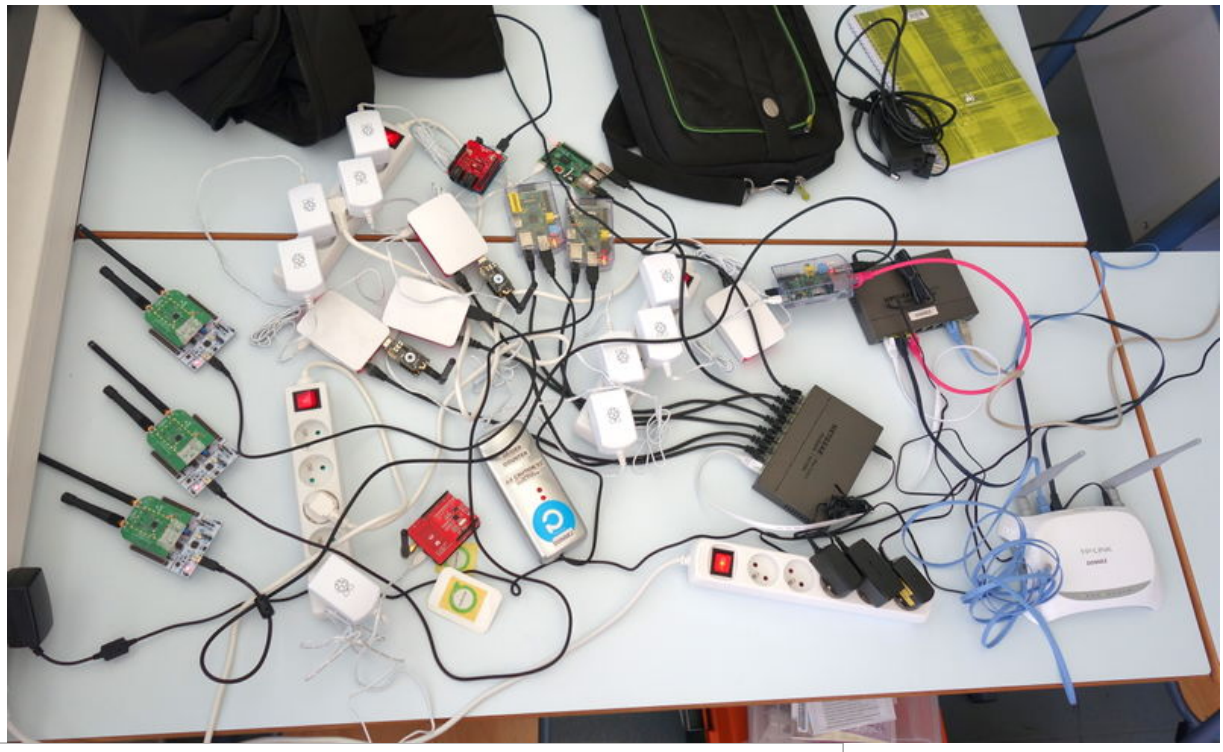
Developing IoT Mashups with Docker, MQTT, Node-RED, InfluxDB, Grafana

From air

Page for the tutorial " Developing IoT Mashups with Docker, MQTT, Node-RED, InfluxDB, Chronograf, Grafana " at Eclipse IoT Days Grenoble 2016 (https://wiki.eclipse.org/Eclipse_IoT_Day_Grenoble_2016#Developping_IoT_Mashups_with_Docker.2C_MQTT.2C_Node-RED.2C_InfluxDB.2C_Grafana)



Contents

Introduction

presentation ...

The goal of this tutorial is building rapidly a minimal IoT stack from sensors to dataviz and realtime analytics.

The hardware compoments used in this stack are:

- various (wireless) sensors,
- USB radio modems plugged into embedded IoT gateways,
- Raspberry Pi 1 and 2 playing the role of embedded IoT gateways,
- Your PC/Mac playing the role of the IoT datacenter server,
- A t2 micro AWS instance playing the role of the IoT datacenter server if a public IP address is required (case for Sigfox callback).

The software compoments used in this IoT stack are :

- Docker,
- Node.js and NPM,
- Node-RED and node contributions,
- Mosquitto
- InfluxDB
- MongoDB
- Chronograf
- Grafana (not on RPI)
- Spark (not on RPI)

Those software compoments are preinstalled on the Raspian Jessie distribution of the Raspberry PI boards in order to speed the installation and prevent WiFi whims.

Tools are:

- vi
- Arduino IDE (https://www.arduino.cc/en/Main/Software)
- Waspmote IDE (http://www.libelium.com/development/waspmote/sdk_applications/)
- MBed IDE (https://developer.mbed.org/)
- Energia for TI SensorTag2 (http://energia.nu/download/)

Security, Avaibility and Performance issues are not in the scope of this tutorial.

Radio communication technologies are : Wifi, BLE, LoRa, SigFox, NFC and Rfxcom 433 MHz.

Info: the username and password on the RPIs are `pi` and `IoTTraining`

Install Docker

MacOS and Debian

See Docker

Raspian

```
wget --no-check-certificate https://downloads.hypriot.com/docker-hypriot_1.10.2-1_armhf.deb
sudo dpkg -i docker-hypriot_1.10.2-1_armhf.deb
sudo service docker restart
sudo docker info
```

Install Node.js and NPM

http://nodered.org/docs/hardware/raspberrypi

On Raspian Jessie : Node.JS is already installed but NPM not

```
nodejs -v
sudo apt-get update
sudo apt-get install npm
npm -v
```

On Raspian Wheezy

```
sudo apt-get update
sudo apt-get install nodejs npm
nodejs -v
npm -v
```

On MacOS X

```
TODO
```

With Docker

```
docker pull node
```

See https://hub.docker.com/_/node/

Install Node-RED

http://nodered.org/docs/hardware/raspberrypi

On Raspian Jessie : Node-RED is already installed (but you can update it with `sudo apt-get update ; sudo apt-get install nodered`)

```
node-red-start
pgrep node-red
```

The default directory is : ~/.node-red

Use `node-red-stop` to stop Node-RED

Use `node-red-start` to start Node-RED again

Use `sudo systemctl enable nodered.service` to autostart Node-RED at every boot

Use `sudo systemctl disable nodered.service` to disable autostart on boot

Browse http://127.0.0.1:1880/

Now, you can write your own Node-RED flows and functions (http://nodered.org/docs/writing-functions) .

Install the admin cmdline tool

```
sudo npm install -g node-red-admin
node-red-admin help

# installed modules
node-red-admin list
```

```
# search modules
node-red-admin search serial
node-red-admin search rfxcom
node-red-admin search zwave
node-red-admin search ble
node-red-admin search bluetooth
node-red-admin search sensortag

node-red-admin search beaglebone
node-red-admin search gpio

node-red-admin search mongodb
node-red-admin search redis
node-red-admin search cassandra
node-red-admin search elastic
node-red-admin search influxdb
node-red-admin search kafka
```

Remark: node-red-admin install <module> does not work ! Use `npm install -g module`

```
cd ~/.node-red
sudo npm install -g node-red-node-redis
sudo npm install -g node-red-contrib-kafka-consumer
sudo npm install -g node-red-node-mongodb
```

Restart Node-Red

```
node-red-admin list
```

With Docker

```
docker pull nodered/node-red-docker
```

Install extra nodes for Node-RED

Extra nodes are provided with the Node-RED community. There are listed here (http://flows.nodered.org/) .

Serial for Geiger Counter

Install the *serial* node (node-red-node-serialport (https://www.npmjs.com/package/node-red-node-serialport))

```
npm install node-red-node-serialport
```

or

```
sudo npm install -g npm@2.x
npm install node-red-node-serialport@0.0.5
```

if node.js prior to 4.x (ie v0.10.x and v0.12.x)

Restart Node-RED with `node-red -v`.

Check the module in the list

```
node-red-admin list | grep serial
```

Sparkfun's geiger counter

Check available serial ports (/dev/tty.usbserial* on MacOS X, /dev/ttyUSB* ...) with `ls /dev/tty.*`.

Connect the Geiger counter (http://www.sparkfun.com/products/9848) to the host.

Check available serial ports (/dev/tty.usbserial* on MacOS X, ...) with `ls /dev/tty.*`.

Add a node *serial* "Geiger" with a Settings of *9600/8/N/1* and 'Split input into fixed lenghts of 1 chars'.

Add a node *debug*.

Connect "Geiger" to *debug*.

Deploy the flow.

The Geiger Counter sends a random sequence of 0 and 1.

Add a node *function* "Count Particles" with a flow-scoped variable (geiger/count):

```
var COUNT='geiger/count';

// initialise the counter to 0 if it doesn't exist already
var count = flow.get(COUNT)||0;
count += 1;
// store the value back
flow.set(COUNT,count);
// make it part of the outgoing msg object
msg.count = count;
msg.payload = {};
msg.payload.device = "geiger";
msg.payload.count = count;

node.log("Received particles : " + count);

return msg;
```

Connect node "Geiger" to node "Count Particles".

Deploy the new flow.

Edit the node "Count Particles" and add the 2 following statements in order to display the count into the node's status.

```
...
setTimeout(function() { node.status({}); }, 1000);
// The shape property can be: ring or dot.
// The fill property can be: red, green, yellow, blue or grey
node.status({fill:"green",shape:"dot",text:"#"+count});
return msg;
```

Deploy the new flow.

Add a node *inject* "One minute timer" with a repeat interval of 1 minute.

Add a node *function* "Reset Particles Count" with a flow-scoped variable (geiger/count):

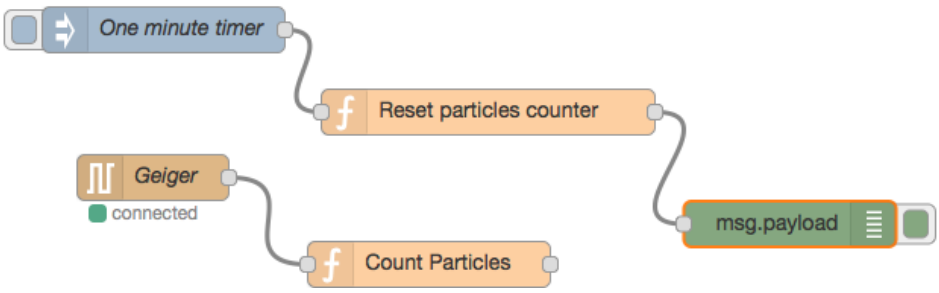
```
var COUNT='geiger/count';

// initialise the counter to 0 if it doesn't exist already
var count = flow.get(COUNT)||0;

msg.count = count;
msg.payload = {};
msg.payload.device = "geiger";
msg.payload.ppm = count;

node.log("Reset counter at " + count);

// make it part of the outgoing msg object
count = 0;

```

```
    // store the value back
16.
    flow.set(COUNT,count);
17.

18.
    return msg;
```

Connect node "One minute timer" to node "Reset Particles Count" and node "Reset Particles Count" to node *debug*.

Deploy the new flow.

The result is:



RFXCom for Oregon Weather Sensors

Install the *rfxcom* node (node-red-contrib-rfxcom (https://www.npmjs.com/package/node-red-contrib-rfxcom))

```
cd ~/.node-red
npm install -g node-red-contrib-rfxcom
```

Restart Node-RED with `node-red -v`.

Check the module in the list

```
node-red-admin list | grep rfx
```

Connect the RFXCom receiver to the host.

Check available serial ports (/dev/tty.usbserial* on MacOS X, ...) with `ls /dev/tty.*`.

Add a node *rfxcom-sensors* "RFXCom" with the correct serial port.

Add a node *debug* display the full message (not only msg.payload).

Connect "RFXCom" to *debug*.

Deploy the flow.

The flow loos like that:

Serial for Arduino + Weather Shield


Sparkfun Weather Shield

Plug the SparkFun Weather Shield into the Arduino Leonardo. Connect the Weather Meters to the shield sockets.

Install the Arduino IDE on your host (link (https://www.arduino.cc/en/Main/Software))

Install 2 extra libraries (Menu Sketch > Include Library > Add ZIP Library)

- https://codeload.github.com/sparkfun/SparkFun_HTU21D_Breakout_Arduino_Library/zip/master
- https://codeload.github.com/sparkfun/SparkFun_MPL3115A2_Breakout_Arduino_Library/zip/master

Select the right board (Arduino Leonardo) with the menu Tool

Load and unzip the Weather_Shield sketch bundle
https://codeload.github.com/sparkfun/Weather_Shield/zip/master

Load the sketch *Weather_Shield/firmware/Weather_Shield/Weather_Shield.ino* into the Arduino Leonardo

Check the output with the menu Tool > Serial Monitor

The output looks like this:

```
$,winddir=-1,windspeedmph=3.1,windgustmph=117.7,windgustdir=-1,windspdmph_avg2m=3.1,winddir_avg2m=0,windgustmph_10m=117.7,windgustdir_10m=-1,humidi
```

Install the *serial* node (node-red-node-serialport (https://www.npmjs.com/package/node-red-node-serialport))

```
npm install node-red-node-serialport
```

or

```
sudo npm install -g npm@2.x
npm install node-red-node-serialport@0.0.5
```

if node.js prior to 4.x (ie v0.10.x and v0.12.x)

Restart Node-RED with `node-red -v`.

Check the module in the list

```
node-red-admin list | grep serial
```

Check available serial ports (/dev/tty.usbserial* on MacOS X, /dev/ttyUSB* ...) with `ls /dev/tty.*`.

Start Node-RED

```
node-red -v
```

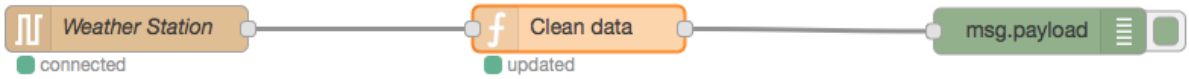
Add a node *debug*

Add a node *function* "Clean Data" with the following statements:

```
var m=msg.payload;
var i=m.indexOf(",")+1;
msg.payload=m.substr(i,m.lastIndexOf(",")-i);
setTimeout(function() { node.status({}); }, 500)
node.status({fill:"green",shape:"dot",text:"updated"});
return msg;
```

Add a node *serial* "Weather Station" with settings 9600/8/N/1 and split input on charracter /n

Connect them together like that



The flow is :

```
[{"id":"40c61a94.428894","type":"serial-port","z":"65e2c142.626cf8","serialport":"/dev/tty.usbmodem1451","serialbaud":"9600","databits":"8","parity
```

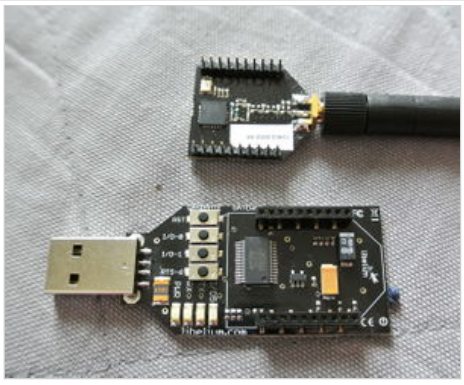
Serial for LoRa Libelium

Install the Waspmote PRO IDE from Libelium on your host (link (http://www.libelium.com/development/waspmote/sdk_applications/))

Load the sketch SX_02a_TX_LoRa on the LoRa Waspmote.

```
/*
 *  ------ [SX_02a] - TX LoRa --------
 *
 *  Explanation: This example shows how to configure the semtech
 *  module in LoRa mode and then send packets with plain-text payloads
 */
// Include this library to transmit with sx1272
#include <WaspSX1272.h>


// define the destination address to send packets
uint8_t rx_address = 8;


// status variable
int8_t e;


void setup()
{
  // Init USB port
  USB.ON();
  USB.println(F("SX_02a example"));
  USB.println(F("Semtech SX1272 module TX in LoRa"));


  USB.println(F("----------------------------------------"));
  USB.println(F("Setting configuration:"));
  USB.println(F("----------------------------------------"));


  // Init sx1272 module
  sx1272.ON();


  // Select frequency channel
  e = sx1272.setChannel(CH_10_868);
  USB.print(F("Setting Channel CH_10_868.\t state "));
  USB.println(e);
```


Libelium Waspmote with LoRa modem


Libelium Waspmote with LoRa modem


Libelium Waspmote with LoRa modem

Libelium Waspmote USB Dongle with LoRa modem on RPI2

```
34.
35.   // Select implicit (off) or explicit (on) header mode
36.   e = sx1272.setHeaderON();
37.   USB.print(F("Setting Header ON.\t\t state "));
38.   USB.println(e);
39.
40.   // Select mode: from 1 to 10
41.   e = sx1272.setMode(1);
42.   USB.print(F("Setting Mode '1'.\t\t state "));
43.   USB.println(e);
44.
45.   // Select CRC on or off
46.   e = sx1272.setCRC_ON();
47.   USB.print(F("Setting CRC ON.\t\t\t state "));
48.   USB.println(e);
49.
50.   // Select output power (Max, High or Low)
51.   e = sx1272.setPower('L');
52.   USB.print(F("Setting Power to 'L'.\t\t state "));
53.   USB.println(e);
54.
55.   // Select the node address value: from 2 to 255
56.   e = sx1272.setNodeAddress(2);
57.   USB.print(F("Setting Node Address to '2'.\t state "));
58.   USB.println(e);
59.   USB.println();
60.
61.   delay(1000);
62.
63.   USB.println(F("----------------------------------------"));
64.   USB.println(F("Sending:"));
65.   USB.println(F("----------------------------------------"));
66. }
67.
68. void loop()
69. {
70.   // Sending packet before ending a timeout
71.   e = sx1272.sendPacketTimeout( rx_address, "This_is_a_new_message");
72.
73.   // Check sending status
74.   if( e == 0 )
75.   {
76.     USB.println(F("Packet sent OK"));
77.   }
78.   else
79.   {
```

```
    USB.println(F("Error sending the packet"));
    USB.print(F("state: "));
    USB.println(e, DEC);
  }
  delay(2500);
}
```

Check with the IDE Serial monitor the output of the Waspmote

Load the sketch SX_02b_RX_LoRa on the LoRa Dongle.

```
/*
 *  ------ [SX_02b] - RX LoRa --------
 *  Explanation: This example shows how to configure the semtech
 *  module in LoRa mode and then receive packets with plain-text payloads
 */
// Include this library for transmit with sx1272
#include <WaspSX1272.h>

// status variable
int8_t e;

void setup()
{
  // Init USB port
  USB.ON();
  USB.println(F("SX_02b example"));
  USB.println(F("Semtech SX1272 module RX in LoRa"));

  USB.println(F("----------------------------------------"));
  USB.println(F("Setting configuration:"));
  USB.println(F("----------------------------------------"));

  // Init sx1272 module
  sx1272.ON();

  // Select frequency channel
  e = sx1272.setChannel(CH_10_868);
  USB.print(F("Setting Channel CH_10_868.\t state "));
  USB.println(e);

  // Select implicit (off) or explicit (on) header mode
  e = sx1272.setHeaderON();
  USB.print(F("Setting Header ON.\t\t state "));
  USB.println(e);

  // Select mode: from 1 to 10
  e = sx1272.setMode(1);
```

```
  USB.print(F("Setting Mode '1'.\t\t state "));
  USB.println(e);

  // Select CRC on or off
  e = sx1272.setCRC_ON();
  USB.print(F("Setting CRC ON.\t\t\t state "));
  USB.println(e);

  // Select output power (Max, High or Low)
  e = sx1272.setPower('L');
  USB.print(F("Setting Power to 'L'.\t\t state "));
  USB.println(e);

  // Select the node address value: from 2 to 255
  e = sx1272.setNodeAddress(8);
  USB.print(F("Setting Node Address to '8'.\t state "));
  USB.println(e);

  delay(1000);

  USB.println(F("----------------------------------------"));
  USB.println(F("Receiving:"));
  USB.println(F("----------------------------------------"));
}

void loop()
{
  // receive packet
  e = sx1272.receivePacketTimeout(10000);

  // check rx status
  if( e == 0 )
  {
    USB.println(F("\nShow packet received: "));

    // show packet received
    sx1272.showReceivedPacket();
  }
  else
  {
    USB.print(F("\nReceiving packet TIMEOUT, state "));
    USB.println(e, DEC);
  }
}
```

Check with the IDE Serial monitor the output of the USB Dongle

Install the *serial* node (node-red-node-serialport (https://www.npmjs.com/package/node-red-node-serialport))

```
npm install node-red-node-serialport
```

or

```
sudo npm install -g npm@2.x
npm install node-red-node-serialport@0.0.5
```

if node.js prior to 4.x (ie v0.10.x and v0.12.x)

Restart Node-RED with `node-red -v`.

Check the module in the list

```
node-red-admin list | grep serial
```

Check available serial ports (`/dev/tty.usbserial*` on MacOS X, `/dev/ttyUSB*<code>, <code>/dev/ttyAMA*<code>, <code>/dev/ttyACM*<code> ...) with <code>ls /dev/tty.*`.

The RX boards receive LoRa frames and output the following lines on the console when the TX boards are powered :

```
...
SX_02b example
Semtech SX1272 module RX in LoRa
----------------------------------------
Setting configuration:
----------------------------------------
Setting Channel CH_10_868.       state 0
Setting Header ON.               state 0
Setting Mode '1'.                state 0
Setting CRC ON.                  state 0
Setting Power to 'L'.            state 0
Setting Node Address to '8'.     state 0
----------------------------------------
Receiving:
----------------------------------------

Show packet received:
==============================
dest: 8
src: 2
packnum: 171
length: 26
retry: 0
payload (HEX): 546869735F69735F615F6E65775F6D657373616765
payload (string): This_is_a_new_message
==============================
...
```

Create a new flow or start Node-RED

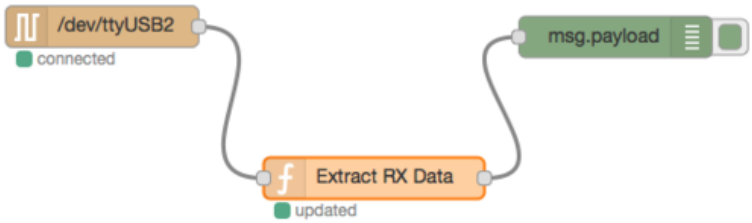
Add a node *debug*

Add a node *function* "Extract RX Data" with the following statements

```
var prefix="payload (HEX): ";
var m=msg.payload;
var i=m.indexOf(prefix);
if(i===0) {
    msg.payload=new Buffer(m.substr(prefix.length),"HEX");
    setTimeout(function() { node.status({}); }, 500)
    node.status({fill:"green",shape:"dot",text:"updated"});
    return msg;
} else {
    return null;
}
```

Add a node *serial* "LoRa RX Modem" with settings 115200/8/N/1 and split input on character /n. On RPI, the port is `/dev/ttyUSB0`.

Connect them together like that



The flow is :

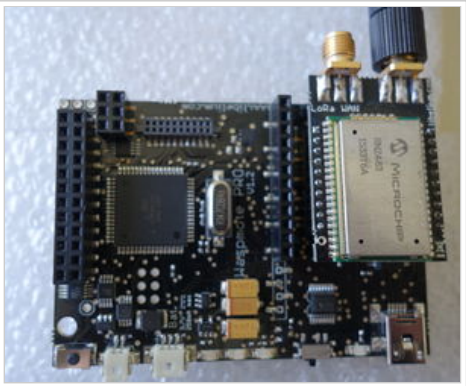
```
[{"id":"3124f278.d5cc86","type":"serial-port","z":"3cf8afc1.fe41a","serialport":"/dev/ttyUSB2","serialbaud":"115200","databits":"8","parity":"none"
```

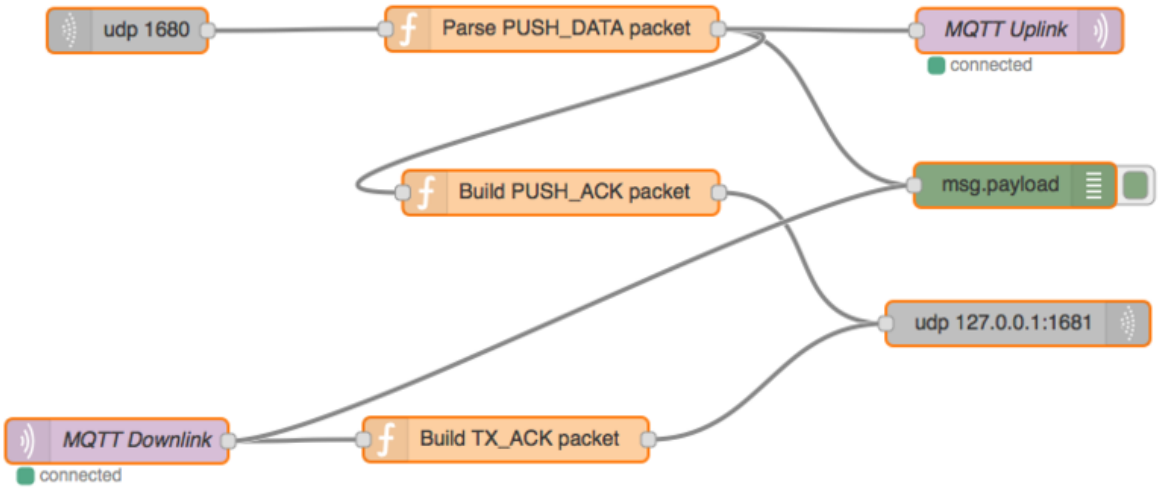
Serial for LoRaWAN Libelium

Install the Waspmote PRO IDE from Libelium on your host (link (http://www.libelium.com/development/waspmote/sdk_applications/))

Load the sketch LoRaWAN_02_join_abp_send_unconfirmed (from Examples > 03. Communications > LoRaWAN) on the LoRaWAN Waspmote. (API documentation (http://www.libelium.com/downloads/documentation/waspmote-lorawan-networking-guide.pdf))

Add the following flow into the Node-RED server embedded in the Multitech gateway. This flow adapts the [Shttps://github.com/Lora-net/packet_forwarder/blob/master/PROTOCOL.TXT emtech packer forwarder protocol] to MQTT. TBC


Libelium Waspmote with LoRaWAN modem



The flow is :

```
[{"id":"5bf2579.78cd328","type":"mqtt-broker","broker":"192.168.0.101","port":"1883","clientid":""},{"id":"b1939dd4.d3375","type":"debug","name":""
```

Test with `mosquitto_sub -t "gateway/up/#" -d`

Serial for LoRa Nucleo

Install the *serial* node (node-red-node-serialport (https://www.npmjs.com/package/node-red-node-serialport))

```
npm install node-red-node-serialport
```

or

```
sudo npm install -g npm@2.x
npm install node-red-node-serialport@0.0.5
```

if node.js prior to 4.x (ie v0.10.x and v0.12.x)

Restart Node-RED with `node-red -v`.

Check the module in the list

```
node-red-admin list | grep serial
```

STM32 Nucleo F411 + Semtech SX1276 Eval Kit

Check available serial ports (`/dev/tty.usbserial*` on MacOS X, `/dev/ttyUSB*`, `/dev/ttyAMA*`, `/dev/ttyACM*` ...) with `ls /dev/tty.*`.

Compile and load the SX1276Receiver program https://developer.mbed.org/users/donsez/code/SX1276Receiver/ on the Nucleo board.

For the receiver, set line 16 to `#define RX_MODE 1`

For the transmitter, set line 16 to `#define RX_MODE 0`

Remark: select the right board (F411 or L152 for instance).

The RX boards receive LoRa frames and output the following lines on the console when the TX boards are powered :

```
...
>INFO RX modem=1 size=17 rssi=-41 snr=31 freq=868100000 bw=0 sf=12 cr=1 buffer=544553540004411d616263646566676869
RX;1;17;-41;31;868100000;0;12;1;544553540004411d616263646566676869
...
```

Create a new flow or start Node-RED

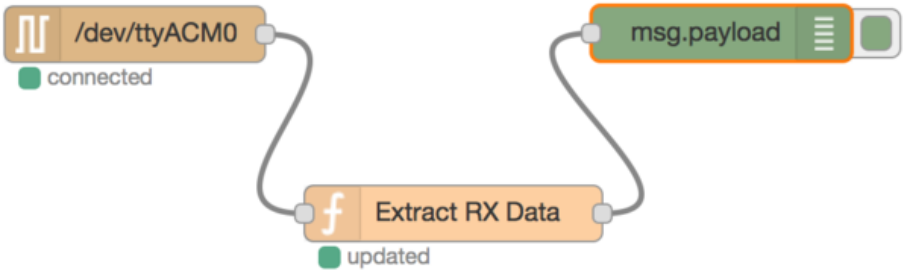
Add a node *debug*

Add a node *function* "Extract RX Data" with the following statements

```
var prefix="RX;";
var m=msg.payload;
var i=m.indexOf(prefix);
if(i===0) {
    msg.payload=m.substr(prefix.length);
    setTimeout(function() { node.status({}); }, 500)
    node.status({fill:"green",shape:"dot",text:"updated"});
    return msg;
} else {
    return null;
}
```

Add a node *serial* "LoRa RX Modem" with settings 115200/8/N/1 and split input on character /n. On RPI, the port is `/dev/ttyACM0`.

Connect them together like that



The flow is :

```
TODO
```

UDP for LoRaWAN

Add a node 'udp in' listening the port 8123.

Add a node 'function' "toString" with the following statement:

```
msg.payload=msg.payload.toString();
return msg;
```

Add a node 'debug'.

Connect them together.

Deploy the new flow.

Test the flow with the shell command

```
echo -n -e '{"device"="123456","temperature"=37.2}' > /dev/udp/127.0.0.1/8123
```

The flow looks like that:

Add a node 'json' into the flow.

Now you can filter and send the output of a process to the flow througth an 'udp in' node.

TO BE CONTINUED

UDP for ESP8266 + PowerSensor


ESP8266 Module 12


ESP8266 Module 12 Arduino

Install the Arduino IDE on your host (link (https://www.arduino.cc/en/Main/Software))

Add the "ESP8266 Modules" support with the Tools menu > Board Manager.

Select "NodeMCU 0.9 (ESP8266-12 Module)" in the Tools menu.

Load the following sketch on the ESP8266.

```
/*
From http://www.iotfrog.com/en/articles/article/227
*/

#include <ESP8266WiFi.h>
#include <WiFiUDP.h>

const char* ssid     = "iotdays";
const char* password = "eclipse2016";

//IPAddress ipBroadCast(192,168,1,255);
IPAddress ipBroadCast;

unsigned int udpRemotePort = 8123;
unsigned int udplocalPort = 8124;
const int UDP_PACKET_SIZE = 48;

char udpBuffer[ UDP_PACKET_SIZE];
```

```
19.
    WiFiUDP udp;
20.
21.
    // Setup the Wifi connection
22.
    void connectWifi() {
23.
      Serial.print("Connecting to ");
24.
      Serial.println(ssid);
25.
26.
      //  Try to connect to wifi access point
27.
      WiFi.begin(ssid, password);
28.
      while (WiFi.status() != WL_CONNECTED) {
29.
        delay(500);
30.
        Serial.print(".");
31.
      }
32.
      Serial.println("");
33.
      Serial.println("WiFi connected");
34.
      Serial.print("IP address: ");
35.
      Serial.println(WiFi.localIP());
36.
37.
      ipBroadCast = ~WiFi.subnetMask() | WiFi.gatewayIP();
38.
    }
39.
40.
    // Send udp message
41.
    void udpSend()
42.
    {
43.
      // TODO read the current value of an analog pin
44.
45.
      strcpy(udpBuffer, "{\"device\"=\"12348266\",\"load\"=10.2}");
46.
      udp.beginPacket(ipBroadCast, udpRemotePort);
47.
      udp.write(udpBuffer, sizeof(udpBuffer));
48.
      udp.endPacket();
49.
      Serial.print("Broadcast: ");
50.
      Serial.println(udpBuffer);
51.
    }
52.
53.
    // Setup hardware, serial port, and connect to wifi.
54.
    void setup() {
55.
      Serial.begin(115200);
56.
      delay(10);
57.
      // We start by connecting to a WiFi network
58.
      connectWifi();
59.
60.
      Serial.println("Starting UDP");
61.
      // set udp port for listen
62.
      udp.begin(udplocalPort);
63.
      Serial.print("Local port: ");
64.
      Serial.println(udp.localPort());
```

```
}

// LOOP MAIN
// Send udp packet each 10 secconds
void loop() {
  udpSend();
  delay (10000);
}
```

Add a node 'udp in' listening the port 8123 with a *Buffer* output.

Add a node 'function' "toStr" with the following statement:

```
msg.payload=msg.payload.toString();
return msg;
```

Add a node 'debug'.

Connect them together.

Deploy the new flow.

The flow looks like that:

Remark: you can test the flow with this shell command

```
echo -n -e '{"device"="12348266","load"=10}' > /dev/udp/127.0.0.1/8123
```

The flow is:

```
[{"id":"b32d523b.fd02d8","type":"udp in","z":"a994a424.14415","name":"","iface":"","port":"8123","ipv":"udp4","multicast":"false","group":"","datat
```

Second part:

Plug the SCT-013 Current Clamp power shield into the ESP8266 Arduino board.

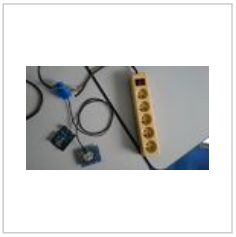
Compile and load the following sketch on the ESP8266 Arduino board.

```
#include <ESP8266WiFi.h>          //https://github.com/esp8266/Arduino

//needed for library
#include <DNSServer.h>
#include <ESP8266WebServer.h>
#include <stdlib.h>
#include "EmonLib.h"             // Include Emon Library

EnergyMonitor emon1;             // Create an instance
const int ANALOG_PIN = A0; // The only analog pin on the Thing

const char host[] = "52.50.55.74";
const int port = 80;
const char nodename[] = "sct013";
```

```
15.
    const char apiKey[] = "&apikey=7af3b1772fa1b624ef73925ba117d562";
16.
17.
18.
    const char* ssid     = "iotdays";
19.
    const char* password = "eclipse2016";
20.
21.
    // Use WiFiClient class to create TCP connections
22.
    WiFiClient client;
23.
24.
25.
    void setup()
26.
    {
27.
      Serial.begin(115200);
28.
      delay(10);
29.
30.
31.
      // We start by connecting to a WiFi network
32.
33.
      Serial.println();
34.
      Serial.println();
35.
      Serial.print("Connecting to ");
36.
      Serial.println(ssid);
37.
      WiFi.begin(ssid, password);
38.
39.
      while (WiFi.status() != WL_CONNECTED) {
40.
        delay(500);
41.
        Serial.print(".");
42.
      }
43.
44.
      Serial.println("");
45.
      Serial.println("WiFi connected");
46.
      Serial.println("IP address: ");
47.
      Serial.println(WiFi.localIP());
48.
      emon1.current(A0, 29.1);       // Current: input pin, calibration.
49.
    }
50.
51.
    void loop()
52.
    {
53.
      Serial.print("waiting a delay\n");
54.
      delay(5000);
55.
56.
      Serial.print("connecting to ");
57.
      Serial.println(host);
58.
59.
      double Irms = emon1.calcIrms(1480);  // Calculate Irms only
60.
      Serial.println("begin loop");
```

```
61.
62.
      const int httpPort = 80;
63.
      if (!client.connect(host, httpPort)) {
64.
        Serial.println("connection failed");
65.
        return;
66.
      }
67.
68.
      // This will send the request to the server
69.
      const String url =  "/input/post.json?node=" + String(nodename) + "&json={power:" + String(Irms * 1.414 * 235 / 10
70.
      client.print("GET " + url + " HTTP/1.1\r\n" +
71.
                   "Host:" + host + "\r\n" +
72.
                   "Connection: close\r\n\r\n");
73.
      Serial.print("Requesting URL: ");
74.
      Serial.println(url);
75.
76.
      delay(10);
77.
78.
      // Read all the lines of the reply from server and print them to Serial
79.
      while (client.available()) {
80.
        String line = client.readStringUntil('\r');
81.
        Serial.print(line);
82.
      }
83.
84.
      Serial.println();
85.
      Serial.println("closing connection");
86.
    }
```

Change the Node-RED to display the received data.

Third part: Change the shetch in order to use MQTT for the message transport : see https://github.com/tuanpmt/esp_mqtt

ACR122U NFC Reader

Plug the ACR122U NFC reader in the host and check it with `lsusb`

Install LibNFC (http://nfc-tools.org/index.php?title=Libnfc) (on Debian)

```
sudo apt-get install libnfc-dev libnfc-bin libnfc-examples
```

```
sudo nfc-list
```

Launch nfc-poll and put a NFC card/tag on the reader

```
sudo nfc-poll -v
```

The ID of the NFC card/tag is prefixed by the string `UID (NFCID1):`.

Execute the following script `sudo ./nfcevent.sh`:

```
#!/bin/sh
while true
do
    nfc-poll | grep "UID (NFCID1):" > /tmp/nfc.log
done
```

Into the Node-RED dashboard, create a new flow.

Add a node *tail* for reading the tail of the log file containing the card/tags identifiers.

Add a node *debug*.

Connect the node *tail* to the node *debug*.

Deploy the new flow.

Add a node *function* "Build message" with the following statements:

```
var m=msg.payload;
var i=m.indexOf(":");
if(i>=0) {
    var tid = m.substr(i+1).replace(/\s+/g, '');
    msg.payload={}
    msg.payload.device="acr122u";
    msg.payload.nfc=tid;
    return msg;
}
return null;
```

Connect the node *tail* to the node "Build message" and the node "Build message" to the node *debug*.

Deploy the new flow.

Edit the node "Build message" and add the 2 following statements in order to display the last tid into the node's status during one second.

```
    ...
    setTimeout(function() { node.status({}); }, 1000);
    // The shape property can be: ring or dot.
    // The fill property can be: red, green, yellow, blue or grey
    node.status({fill:"green",shape:"dot",text:"#"+tid});
    return msg;
}
return null;
```

Deploy the new flow.

Sigfox Weather Station : Sigfox Areku Board + DHT11

Connect the DHT11 pins into the Areku board (http://snootlab.com/lang-en/snootlab-shields/829-akeru-beta-33-en.html) pins (or the Akene shield with an Arduino or STM32 Nucleo boards).

- + --> 5V
- - --> GND
- OUT --> D2

Install the libraries Areku and DHT11 libraries into the Arduino IDE.

Compile and load the following sketch into the Areku board (http://snootlab.com/lang-en/snootlab-shields/829-akeru-beta-33-en.html) . The board type of the Areku board is **"Uno"**.

```
#include <SoftwareSerial.h>
#define RX_PIN 5
#define TX_PIN 4

#include "DHT.h"

#define DHTPIN 2     // what pin we're connected to

// Uncomment whatever type you're using!
#define DHTTYPE DHT11   // DHT 11
//#define DHTTYPE DHT22   // DHT 22  (AM2302)
//#define DHTTYPE DHT21   // DHT 21 (AM2301)

// Connect pin 1 (on the left) of the sensor to +5V
// Connect pin 2 of the sensor to whatever your DHTPIN is
// Connect pin 4 (on the right) of the sensor to GROUND
// Connect a 10K resistor from pin 2 (data) to pin 1 (power) of the sensor

DHT dht(DHTPIN, DHTTYPE);

SoftwareSerial sigfox(RX_PIN,TX_PIN);

// Sigfox message size is 12 bytes max.
#define BUFFERSIZE 256
char downlinkbuffer[BUFFERSIZE];

// Send message every 11 minutes (max 140 messages per day)
#define PERIOD 660000

void getDownlinkLine(char * buffer)
{
  uint8_t idx = 0;
  char c;
  do
  {
    while (sigfox.available() == 0) ; // wait for a char this causes the blocking
    c = sigfox.read();
    buffer[idx++] = c;
  }
  while (c != '\n' && c != '\r' && idx >= BUFFERSIZE);
  buffer[idx] = 0;
}

void setup(){
```

```
45.
    Serial.begin(9600);
46.
    sigfox.begin(9600);
47.
    dht.begin();
48.
}
49.

50.
void loop(){
51.
  // Reading temperature or humidity takes about 250 milliseconds!
52.
  // Sensor readings may also be up to 2 seconds 'old' (its a very slow sensor)
53.
  float fh = dht.readHumidity();
54.
  float ft = dht.readTemperature();
55.

56.
  // check if returns are valid, if they are NaN (not a number) then something went wrong!
57.
  if (isnan(ft) || isnan(fh)) {
58.
    Serial.println("Failed to read from DHT");
59.
  } else {
60.
    Serial.print("Temperature: ");
61.
    Serial.print(ft);
62.
    Serial.println(" *C");
63.
    Serial.print("Humidity: ");
64.
    Serial.print(fh);
65.
    Serial.println(" %\t");
66.

67.
    Serial.print("Send message\n");
68.
    sigfox.write("AT$SF=");
69.

70.
    sigfox.print((byte)ft, HEX);
71.
    sigfox.write(" ");
72.
    sigfox.print((byte)fh, HEX);
73.

74.
    sigfox.write("\n");
75.

76.
    getDownlinkLine(downlinkbuffer);
77.
    Serial.print(downlinkbuffer);
78.
    Serial.print("\n");
79.

80.
    Serial.print("Wait ");
81.
    Serial.print(PERIOD/1000);
82.
    Serial.print(" second before next uplink\n");
83.

84.
  }
85.
  delay(PERIOD);
86.
 }
```

Check with the IDE Serial monitor the USB output of the board.

Install and run the following Node.js script:

```
1.
var PORT = 3000;
```

```
var PATH = '/sigfox';

//var BROKER_URL = "mqtt://localhost";
var BROKER_URL = "mqtt://test.mosquitto.org";
var TOPIC = "iotdays/sensors/sigfox";

var express = require('express');
var app = express();

var mqtt    = require('mqtt');
var client  = mqtt.connect(BROKER_URL);
var mqttConnected = false;

client.on('connect', function () {
  mqttConnected = true;
  console.log('Sigfox callback server : MQTT connected to ',BROKER_URL);
});

client.on('reconnect', function () {
  mqttConnected = true;
  console.log('Sigfox callback server : MQTT reconnected to ',BROKER_URL);
});

client.on('offline', function () {
  mqttConnected = false;
  console.log('Sigfox callback server : MQTT offline ');
});

client.on('close', function () {
  mqttConnected = false;
  console.log('Sigfox callback server : MQTT closed');
});

client.on('error', function (error) {
  mqttConnected = false;
  console.log('Sigfox callback server : MQTT Error ',error);
});


app.get(PATH, function (req, res) {
  console.log("Received ", JSON.stringify(req.query));
  if(mqttConnected) {
        client.publish(TOPIC, JSON.stringify(req.query));
        res.send('Callback received');
  } else {
        res.status(500).send('Could not delivered callback');
```

```
  }
});

app.listen(PORT, function () {
  console.log('Sigfox callback server listening on port ', PORT);
});
```

```
npm init
npm install express --save
sudo npm install mqtt --save
cat package.json
node server.js
```

Configure the callback http://52.50.55.74:3000/sigfox?data={data}&id={device}&time={time}&snr={snr}&station={station}&avgSnr={avgSnr}&rssi={rssi}&lat={lat}&lng={lng}&seqNumber={seqNumber}&duplicate={duplicate} for the device into the Sigfox backend (https://backend.sigfox.com/devicetype/562fa7229336f078ba2404c2/callbacks) . (account is required)

Create a flow with a *MQTT in* node and a *debug* node.

Parse and display the values sent by the Sigfox backend : device, time, duplicate, snr, station, data, avgSnr, lat, lng, rssi, seqNumber

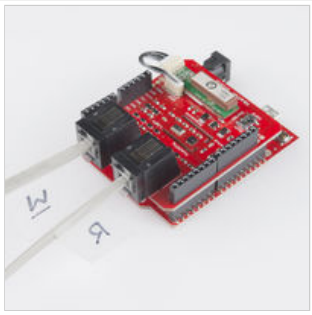
TBC

Remarks:

- The Sigfox modem can not send more than 144 messages due to the ETSI regulation.
- A Sigfox message contains up to 12 bytes (authenticated but not encrypted)



Sigfox Patrol Man : Sigfox Areku Board + Sparkfun's Weather Station Shield


Sparkfun Weather Shield

Plug the SparkFun Weather Shield into the Areku board. Plug the Weather Meters into the shield sockets.

Install the Arduino IDE on your host (link (https://www.arduino.cc/en/Main/Software))

Install 2 extra libraries (Menu Sketch > Include Library > Add ZIP Library)

- https://codeload.github.com/sparkfun/SparkFun_HTU21D_Breakout_Arduino_Library/zip/master
- https://codeload.github.com/sparkfun/SparkFun_MPL3115A2_Breakout_Arduino_Library/zip/master

Select the right board (Arduino Uno) for the Akeru board with the menu Tool

Load and unzip the Weather_Shield sketch bundle
https://codeload.github.com/sparkfun/Weather_Shield/zip/master

Load the sketch *Weather_Shield/firmware/Weather_Shield/Weather_Shield.ino* into the Arduino Leonardo

Check the output with the menu Tool > Serial Monitor

TODO

Sigfox Patrol Man : Sigfox Areku Board + Adafruit NFC Shield

Install the libraries Areku and Adafruit_PN532 (https://github.com/PM2M-2016-NFCSigfox/pm2m/tree/master/arduino/libraries) into the Arduino IDE.

Compile and load the sketch into the Areku board (http://snootlab.com/lang-en/snootlab-shields/829-akeru-beta-33-en.html) . (https://github.com/PM2M-2016-NFCSigfox/pm2m/blob/master/arduino/rondesNfc/rondesNfc.ino). The board type of the Areku board is **"Uno"**.

```
#define DELAY_BETWEEN_MESSAGES 10000
#include <Akeru.h>
```

```
3.
4.  #include <Adafruit_PN532.h>
5.  #include <SPI.h>
6.  #include <SoftwareSerial.h>
7.
8.  #define PN532_CS 10 // La pin CS peut être connectée à la sortie D9 ou D10.
9.  Adafruit_PN532 nfc(PN532_CS);
10. #define NFC_DEMO_DEBUG 1
11.
12. void setup()
13. {
14. #ifdef NFC_DEMO_DEBUG
15.     Serial.begin(9600); // Vérification liaison série.
16.     // Init modem
17.     Akeru.begin();
18.     Serial.println("Bonjour!");
19. #endif
20.     nfc.begin(); // Démarrage puce PN532.
21.
22.     uint32_t versiondata = nfc.getFirmwareVersion();
23.
24.     if (!versiondata) {
25. #ifdef NFC_DEMO_DEBUG // Vérification PN532.
26.         Serial.print("Puce PN532 absente");
27. #endif
28.         while (1)
29.             ;
30.     }
31. #ifdef NFC_DEMO_DEBUG // Vérification paramétrage de la puce.
32.     Serial.print("Puce detectee, PN5");
33.     Serial.println((versiondata >> 24) & 0xFF, HEX);
34.     Serial.print("Firmware version: ");
35.     Serial.print((versiondata >> 16) & 0xFF, DEC);
36.     Serial.print('.');
37.     Serial.println((versiondata >> 8) & 0xFF, DEC);
38.     Serial.print("Supports ");
39.     Serial.println(versiondata & 0xFF, HEX);
40.
41. #endif
42.     nfc.SAMConfig(); // Configuration de la carte pour lire des tags et cartes RFID.
43. }
44.
45. void loop()
46. {
47.
48.     // Attente de disponiblité (envoi d'un messages toutes les 11 minutes : 140 messages par jour max)
```

```
    while (!Akeru.isReady()) {
        Serial.println("Modem en mode attente.");
        delay(2500);
    }

    Serial.println("Modem OK");

    uint8_t uid[] = { 0, 0, 0, 0, 0, 0, 0 }; // Buffer pour stocker l'UID NFC
    uint8_t uidLength; // Taille de l'UID (4 or 7 octets)

    bool success = nfc.readPassiveTargetID(PN532_MIFARE_ISO14443A, uid, &uidLength);

    if (success) {
        Serial.print("UID Length: ");
        Serial.print(uidLength, DEC);
        Serial.println(" bytes");

        Serial.print("UID Value: ");

        for (uint8_t i = 0; i < uidLength; i++) {
            Serial.print(" 0x");
            Serial.print((byte)uid[i], HEX);
        }
        Serial.println("");

        // Envoyer les données
        if (Akeru.send(uid, uidLength)) {
            Serial.println("Send OK");
        }
        else {
            Serial.println("Send NOK");
        }
        Serial.print("Mode attente pendant ");
        Serial.print(DELAY_BETWEEN_MESSAGES);
        Serial.println("ms.");
        delay(DELAY_BETWEEN_MESSAGES);
    }
    else {
        Serial.println("Timed out waiting for a card");
    }
}
```

Check with the IDE Serial monitor the USB output of the board and put a NFC tag on the PCB antenna of the Adafruit NFC Shield.

Install and run the following Node.js script:

```
var PORT = 3000;
```

```
var PATH = '/sigfox';

//var BROKER_URL = "mqtt://localhost";
var BROKER_URL = "mqtt://test.mosquitto.org";
var TOPIC = "iotdays/sensors/sigfox";

var express = require('express');
var app = express();

var mqtt    = require('mqtt');
var client  = mqtt.connect(BROKER_URL);
var mqttConnected = false;

client.on('connect', function () {
  mqttConnected = true;
  console.log('Sigfox callback server : MQTT connected to ',BROKER_URL);
});

client.on('reconnect', function () {
  mqttConnected = true;
  console.log('Sigfox callback server : MQTT reconnected to ',BROKER_URL);
});

client.on('offline', function () {
  mqttConnected = false;
  console.log('Sigfox callback server : MQTT offline ');
});

client.on('close', function () {
  mqttConnected = false;
  console.log('Sigfox callback server : MQTT closed');
});

client.on('error', function (error) {
  mqttConnected = false;
  console.log('Sigfox callback server : MQTT Error ',error);
});


app.get(PATH, function (req, res) {
  console.log("Received ", JSON.stringify(req.query));
  if(mqttConnected) {
        client.publish(TOPIC, JSON.stringify(req.query));
        res.send('Callback received');
  } else {
        res.status(500).send('Could not delivered callback');
```

```
    }
});

app.listen(PORT, function () {
    console.log('Sigfox callback server listening on port ', PORT);
});
```

```
npm init
npm install express --save
sudo npm install mqtt --save
cat package.json
node server.js
```

Configure the callback http://52.50.55.74:3000/sigfox?data={data}&id={device}&time={time}&snr={snr}&station={station}&avgSnr={avgSnr}&rssi={rssi}&lat={lat}&lng={lng}&seqNumber={seqNumber}&duplicate={duplicate} for the device into the Sigfox backend (https://backend.sigfox.com/devicetype/562fa7229336f078ba2404c2/callbacks) . (account is required)

Create a flow with a *MQTT in* node and a *debug* node.

Parse and display the values sent by the Sigfox backend : device, time, duplicate, snr, station, data, avgSnr, lat, lng, rssi, seqNumber

TBC

Remarks:

- The Sigfox modem can not send more than 144 messages due to the ETSI regulation.
- A Sigfox message contains up to 12 bytes (authenticated but not encrypted)






LoRa Patrol Man : STM32 Nucleo + NFC Shield

See https://github.com/PM2M2016-STM32NUCLEO/M2M

STM32 Nucleo

Arduino 101 (https://www.arduino.cc/en/Guide/Arduino101) BLE

TODO

Genuino 101

Genuino 101 + DHT11

Intel Quark D2000

TODO

Intel Quark D2000

SensorTag2 BLE

TODO

Follow the instructions here : https://github.com/uwefassnacht/ti-sensor-tag-demo

TI SensorTag2 Dev Kit

TI beacon board plugged with debug module

SensorTag in its protection

TI SensorTag

STEVAL-WESU1

TODO

K8055 Experiment IO Board

TODO

Johnny-Five

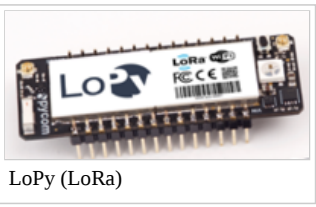
TODO

https://github.com/monteslu/node-red-contrib-gpio

Pycom IoT Boards

1. Pycom LoPy
2. Pycom SiPy
3. Pycom FiPy

TODO



LoPy (LoRa)

SiPy (Sigfox)

File:Fipy.png
FiPy (the full monty)

Install Mosquitto

On MacOS X

```
brew install mosquitto
ln -sfv /usr/local/opt/mosquitto/*.plist ~/Library/LaunchAgents
launchctl load ~/Library/LaunchAgents/homebrew.mxcl.mosquitto.plist

echo Test connectivity
BROKER=localhost
mosquitto_sub -h $BROKER -t '$SYS/#'
```

On Debian

```
sudo apt-get install mosquitto
sudo service mosquitto status
sudo apt-get install mosquitto-clients
echo Test connectivity
BROKER=localhost
mosquitto_sub -h $BROKER -t '$SYS/#'
```

With Docker

```
docker pull eclipse-mosquitto
docker run -it -p 1883:1883 -p 9001:9001 -v mosquitto.conf:/mosquitto/config/mosquitto.conf -v /mosquitto/data -v /mosquitto/log eclipse-mosquitto
```

https://hub.docker.com/_/eclipse-mosquitto/

Test local Mosquitto broker

```
BROKER=localhost
mosquitto_sub -d -h $BROKER -t 'iotdays/sensors/#' &
mosquitto_pub -d -h $BROKER -t 'iotdays/sensors/gw0001' -m '{"device"="123456","temperature"=37.2}'
mosquitto_pub -d -h $BROKER -t 'iotdays/sensors/gw0001' -m '{"device"="345678","temperature"=23.2,"humidity"=50}'
mosquitto_pub -d -h $BROKER -t 'iotdays/sensors/gw0002' -m '{"device"="acr122u","nfc"="04ddf0f9232580"}'
sleep 1
pkill mosquitto_sub
```

Test public Mosquitto broker

```
BROKER=test.mosquitto.org
mosquitto_sub -d -h $BROKER -t 'iotdays/sensors/#' &
mosquitto_pub -d -h $BROKER -t 'iotdays/sensors/gw0001' -m '{"device"="123456","temperature"=37.2}'
mosquitto_pub -d -h $BROKER -t 'iotdays/sensors/gw0001' -m '{"device"="345678","temperature"=23.2,"humidity"=50}'
mosquitto_pub -d -h $BROKER -t 'iotdays/sensors/gw0002' -m '{"device"="acr122u","nfc"="04ddf0f9232580"}'
pkill mosquitto_sub
```

More:

- moquitto_pub manpage (http://mosquitto.org/man/mosquitto_sub-1.html)
- moquitto_sub manpage (http://mosquitto.org/man/mosquitto_pub-1.html)
- MQTT Spy (https://github.com/kamilfb/mqtt-spy/wiki) : a graphical MQTT client in Java

- MQTT Dash for Android (https://play.google.com/store/apps/details?id=net.routix.mqttdash) : a convenient MQTT dashboard for Android.

Using MQTT for collecting sensors data

In the flows defined above, add a node *mqtt out* with a new MQTT broker **localhost:1883** and a topic **iotdays/sensors/gwXXXX** where gwXXXX is the name of the host.

Check the publishing with `mosquitto_sub`

```
BROKER=localhost
mosquitto_sub -d -h $BROKER -t 'iotdays/sensors/#' &
```

Create a new flow for subscribing messages from the Mosquitto MQTT broker.

Add a node *debug* for displaying the full message.

Add a node *mqtt in* with with a new MQTT broker **localhost:1883** and a topic **iotdays/sensors#**

Connect the node *mqtt in* to the node *debug*.

Deploy the new flow.

Install InfluxDB

OS X (via Homebrew)

```
brew update
brew install influxdb
```

Ubuntu & Debian (64-bit)

```
wget http://dl.influxdata.com/influxdb/releases/influxdb_0.12.2-1_amd64.deb
sudo dpkg -i influxdb_0.12.2-1_amd64.deb
```

Ubuntu & Debian (ARM)

```
wget  --no-check-certificate http://dl.influxdata.com/influxdb/releases/influxdb_0.12.2-1_armhf.deb
sudo dpkg -i influxdb_0.12.2-1_armhf.deb
```

Docker

```
docker pull influxdb
docker run -p 8083:8083 -p 8086:8086 \
      -v influxdb:/var/lib/influxdb \
      influxdb
```

https://hub.docker.com/_/influxdb/

Populate InfluxDB from Node-RED

Launch the InfluxDB shell

```
influx
```

Enter the following InfluxDB statements

1.
   ```
   CREATE DATABASE iotdb
   ```
2.
   ```
   SHOW retention policies ON iotdb
   ```
3.
   ```
   CREATE retention policy three_hours_iot_training ON iotdb duration 3h replication 1 DEFAULT
   ```
4.
   ```
   SHOW retention policies ON iotdb
   ```
5.
   ```
   exit
   ```

In the Node-RED dashboard, create a new flow to collect sensors data from the MQTT server.

Install the Node-RED InfluxDB node (http://flows.nodered.org/node/node-red-contrib-influxdb) .

Restart Node-RED.

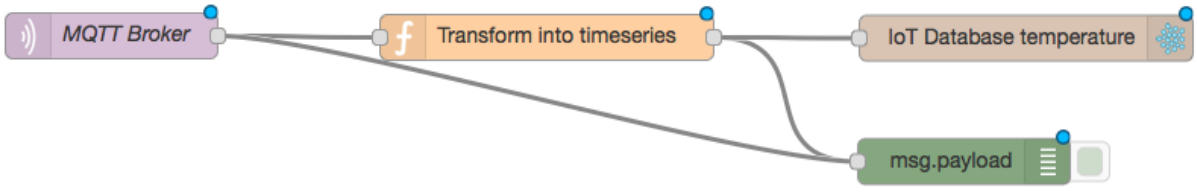
Add a node *mqtt in* "MQTT Broker" for subscribing to the MQTT server (broker is **test.mosquitto.org** or **localhost** topic is **iotdays/sensors/#**)

Add a node *function* "Tranform into time series" to format the JSON messages into **time series** messages.

Add a node *influxdb out* "IoT Database" with the InfluxDB server.

Connect the node "MQTT Broker" to node "Tranform into time series" and to the node "Tranform into time series" to node "IoT Database".

The flow looks like than:



Launch the InfluxDB shell

```
influx
```

Enter the following InfluxDB statements

```
USE iotdb
SHOW measurements
SHOW series

SELECT * FROM temperature
SELECT * FROM humidity
SELECT * FROM nfc
SELECT * FROM ppm
SELECT * FROM CURRENT
SELECT * FROM sigfox
SELECT * FROM lora
SELECT * FROM weather
```

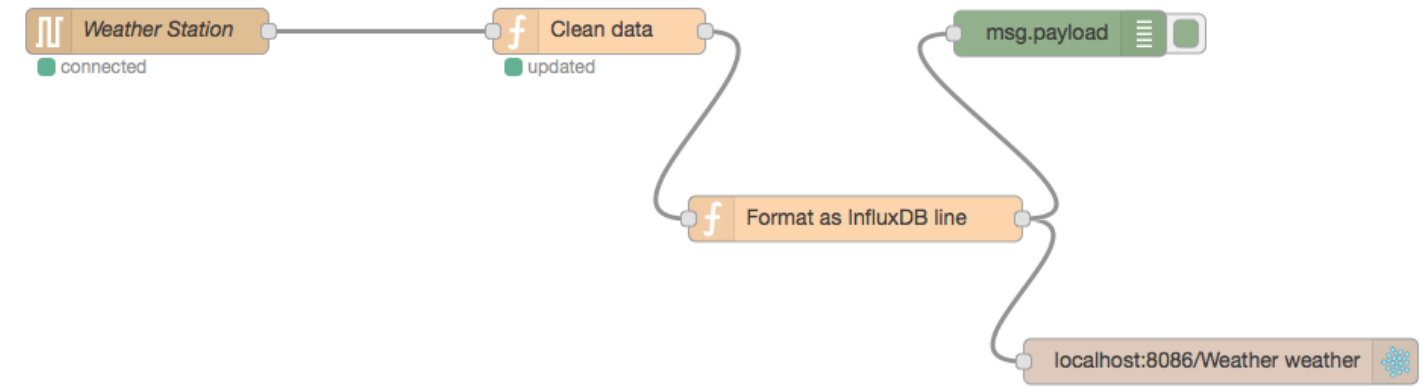
Extra for Arduino and Weather Shield

Add a node *influxdb out*

Add a node *function* with the following statements:

```
var result = {};
msg.payload.split(',').forEach(function(x){
    var arr = x.split('=');
    arr[1] && (result[arr[0]] = parseFloat(arr[1]));
});
msg.payload=result;
return msg;
```

Connect the 2 additionnal nodes like that:



The flow is:

```
[{"id":"57400c2a.74f84c","type":"influxdb","z":"65e2c142.626cf8","hostname":"localhost","port":"8086","database":"iotdb","name":""},{"id":"40c61a94
```

Start the InfluxDB shell:

```
influx
```

Create the database with the foolowing statements:

```
CREATE DATABASE iotdb
SHOW retention policies ON iotdb
CREATE retention policy three_hours_iot_training ON iotdb duration 3h replication 1 DEFAULT
SHOW retention policies ON iotdb

USE iotdb
SHOW measurements
SELECT * FROM weather
```

Install Chronograf

https://influxdata.com/downloads/#chronograf

OS X (via Homebrew)

```
brew update
brew install homebrew/binary/chronograf
```

Ubuntu & Debian

```
wget https://s3.amazonaws.com/get.influxdb.org/chronograf/chronograf_0.12.0_amd64.deb
sudo dpkg -i chronograf_0.12.0_amd64.deb
```

Remark: Chronograf configuration directory is ~/chronograf.db

Docker

https://hub.docker.com/_/chronograf/

Visualize sensors data into the Chronograf dashboard

TODO

Browse https://localhost:10000/

Configure the InfluxDB datasource

Install Grafana (2.6)

On Debian

```
wget https://grafanarel.s3.amazonaws.com/builds/grafana_2.6.0_amd64.deb
sudo dpkg -i xzvf grafana_2.6.0_amd64.deb
sudo service grafana-server start
```

Docker

```
docker pull grafana/grafana
```

https://hub.docker.com/r/grafana/grafana/

On Raspian Jessie

See https://github.com/heziegl/rpi-grafana

Visualize sensors data into the Grafana dashboard

Browse http://localhost:3000

Login with username:password admin:admin

Change your admin password (optional)

Create a new dashboard

Add a data source (InfluxDB 0.9) iotdb (http://localhost:8086 direct root:root)

Save the data source

Enable dashboard edition.

Add a row.

Add a graph in the row.

Edit the graph.

Edit the query for the graph for the data source.

Save the dashboard.

Duplicate the graph and change the measurement in the query

Duplicate the graph and aggregate (mean) all measurements of the same type (temperature, humidity).

Set the timeline and the refresh time.

Save the dashboard.

Move and resize the graphs into the dashboard.

Save the dashboard.

See Grafana

The result should look like that

Install Apache Spark

See Spark

Process a realtime stream of sensors data with Apache Spark

TODO

```
import org.eclipse.paho.client.mqttv3._
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence

import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.mqtt._
import org.apache.spark.SparkConf

val brokerUrl = "tcp://localhost:1883"
val topic = "iotdays/sensors/#"
val ssc = new StreamingContext(sc, Seconds(60))
val sensordatas = MQTTUtils.createStream(ssc, brokerUrl, topic, StorageLevel.MEMORY_ONLY_SER_2)

TODO

ssc.start()
ssc.awaitTermination()
```

See

- Aggregate RDD values per key (http://stackoverflow.com/questions/29741452/aggregate-rdd-values-per-key)
- IBM Bluemix IoT device events to Apache Spark. (https://github.com/sathipal/spark-streaming-mqtt-with-security_2.10-1.3.0)
- https://github.com/pranab/ruscello/blob/master/spark/resource/level_shift_detector_tutorial.txt

Annexes

Raspian Jessie commands

TODO

IBM IoT Watson

TODO

OVH IoT PaaS

With OpenTSDB and Grafana.