# Radio chirp data incorporated in an MQTT environment

BY LARS 26 OCTOBER, 2016

Internet-of-things does not require that every device has to be directly connected to the Internet. The complexity and possible security issues with every sensor having its own IP address would in fact be overwhelming. A better approach would be to use more light-weight protocols for the sensor and actuator data and locally aggregate and filter these data at common points before making them available on the Internet. In this post I will describe a theory and implementation of transmitting small radio chirp messages from an Arduino Pro mini and then receive these data on a Raspberry Pi for transformation to MQTT messages for the Internet.

# Background

For some time now, I have experimented with IoT-nodes at home for doing automation and collecting data. With empirical learning from this, I have found a way to scale IoT-nodes in a simple, pragmatic and inexpensive way. Meanwhile I have also read *Rethinking the Internet of things* by Francis daCosta:

It contains a more extensive and formalized description of parts that I found out by experience, so I will borrow some terminology from this book. BTW, the book is worth reading and you can get the ebook edition for free from ApressOpen: http://www.apress.com/gp/book/9781430257400

# An example architecture for collecting sensor data

Data (e.g. temperature) is collected by a sensor connected to an *IoT end device* (e.g. an Arduino board) – they are at the far end of the IoT network – and several sensors can be attached to the same device. The end device makes a minimal *chirp* message of the data and broadcasts it wirelessly. Simple devices can use radio messages (I use 433MHz transmitters) and more complex end devices (like an ESP8266 board) can publish messages over WiFi.

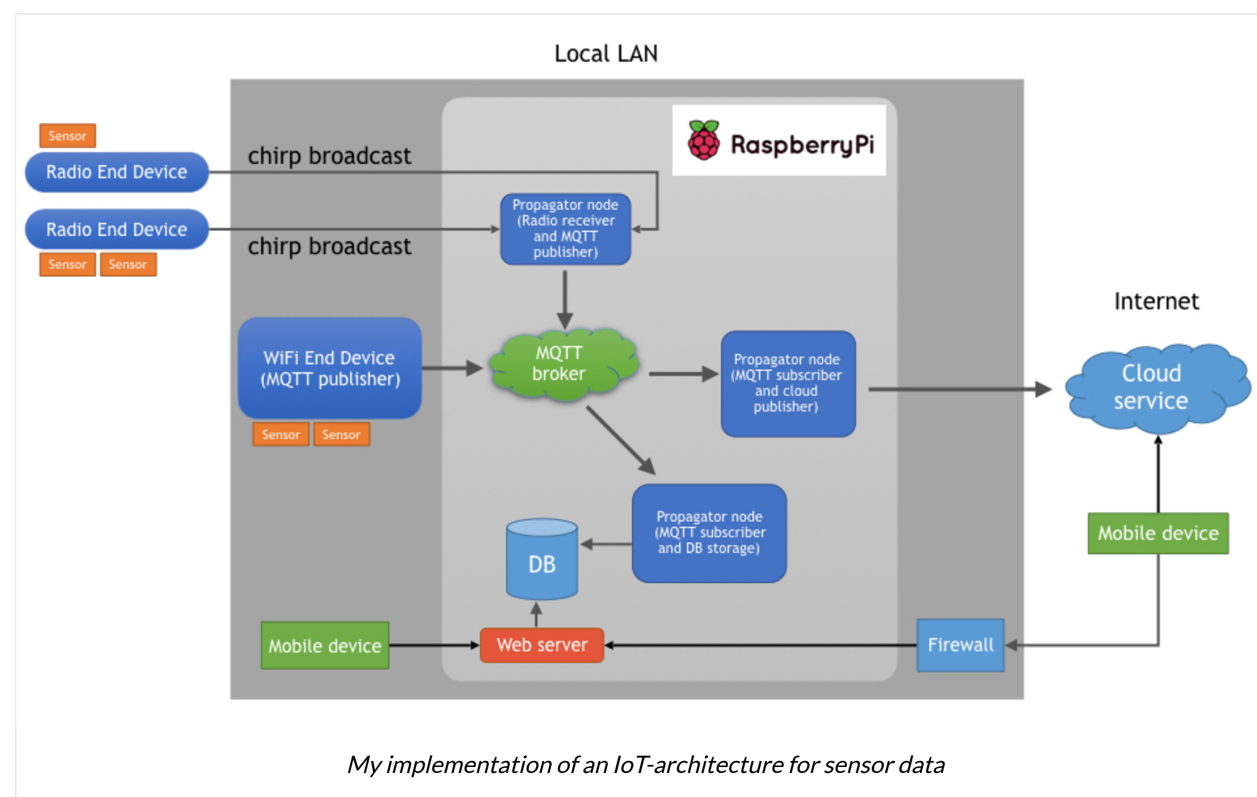
The 433 MHz radio protocol has no acknowledge of received data, so missed or duplicated messages on the receiver side can happen. Thus, this light-weight communication protocol suites scenarios where the data is non-critical. For example, if we have a device that broadcasts the current indoor house temperature every 15 minutes, a missed or duplicated data point is unlikely to cause any trouble or disasters.

*IoT propagator nodes* listens for chirp messages, filters/selects data of interest, aggregates it and make a transform into MQTT IP packages for the local network. A higher-level propagator node subscribes to these messages and when receiving data it executes some actions. The actions could be further propagation of the message from the LAN to Internet and/or store the data locally.

So, the main idea is to use as simple end devices/sensor nodes as possible and

then propagate the information upwards in the "value chain" where more advanced handling of the data can be added at every step.

The collected data can be consumed by mobile devices by accessing a web application on the local LAN or by interacting with a cloud service (e.g. Adafruit IO) that has got a copy of the data. This is the final step where the most advanced node (a human being) analyzes the data.




*My implementation of an IoT-architecture for sensor data*

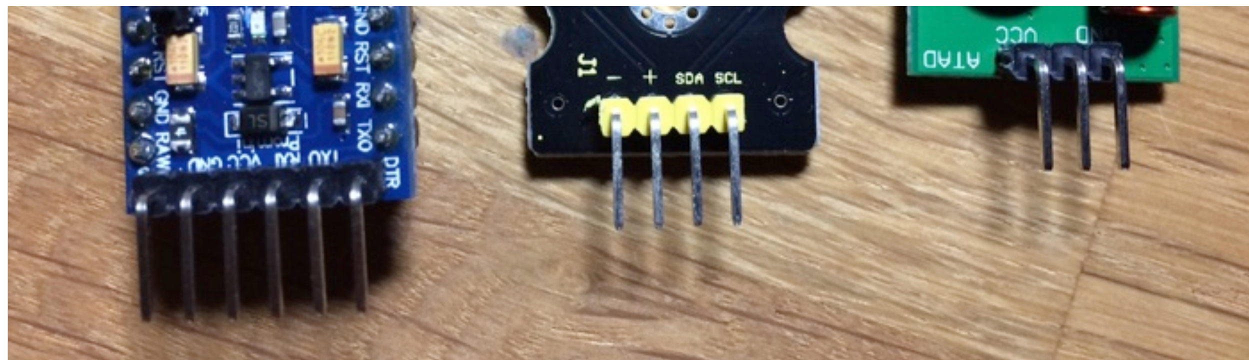
I have covered local MQTT environments in some previous posts, so I will not go

into detail about subscribers and publishers here. Check out these texts instead:

- A self-hosted MQTT environment for IoT – part 1
- A self-hosted MQTT environment for IoT – part 2
- A self-hosted MQTT environment for IoT – part 3

## An example implementation

As an example implementation I will use an Arduino Pro Mini (3.3V edition) as IoT-end device. It uses a BMP180 pressure/temperature sensor and has a 433 MHz radio transmitter. I chose this Arduino model as it can easily be made to consume very little power when idling. By using a sleep library and removing the "on"-led on the board, the required current while idling is only 0.25mA – that's ok for running the device on batteries. The power consumption can be reduced even further, but for my purpose, this is sufficient.



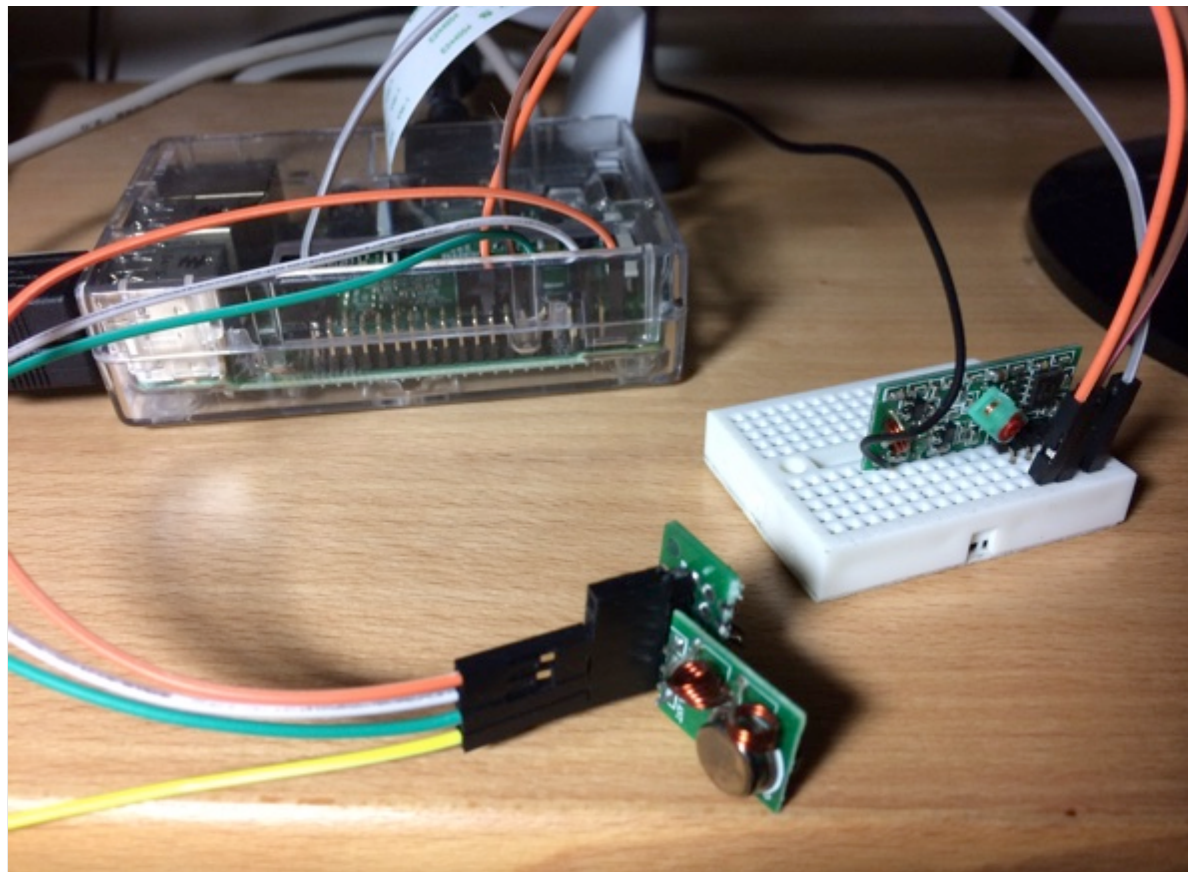

*Arduino Pro Mini 3.3V, BMP180 and 433MHz transmitter*

The cost for this IoT-end device is:

Arduino Pro Mini 3.3V (OpenSmart) = $4
BMP180 sensor (Keyestudio) = $4
433 MHz transmitter (noname) = $3
battery holder + wires = $3

That is, around $14 in total, and the parts can be reused for other projects (I have not included my re-chargeable AA-batteries in the budget above as I expect to be using them for several hundred charge cycles).

The propagator nodes are implemented as services with Python on a Raspberry Pi 3. The RPi has a 433 MHz receiver and WiFi connected to the local LAN.



*Raspberry Pi 3 with radio transmitter and receiver*

My RPi 3 is setup with a 433 MHz receiver and a transmitter. The transmitter is used for sending data to IoT-nodes with actuators, but as this post is about a sensor scenario, I will not cover the RPi transmitter in this text (if you are curious, you can read more in this post https://larsbergqvist.wordpress.com/2016/05/15/rcswitch-revisited-control-rc-outlets-with-a-web-app/).

## The 433 MHz communication protocol

The data that is sent over 433 MHz radio is handled by the RCSwitch library on the Arduino and the pi-switch library on the Raspberry Pi. These libraries support sending and receiving data to/from RC outlets and remote controls but can be used for other purposes. I have used the libraries in previous posts for communication between devices, for example in this one:

https://larsbergqvist.wordpress.com/2016/03/20/arduino-to-raspberry-wireless-communication-some-improvements/

RCSwitch uses 32 bits / 4 bytes for a message – this is my *chirp*! I only use two bytes for the actual sensor data, the other two bytes are used for data identification and a checksum.

My chirp can contain an encoded unsigned two byte integer value from a sensor. The whole 32 bit message looks like this in that case:

| Byte 3 | Byte 2 | Byte 1 | Byte 0 |
|---|---|---|---|
| 0000 0000 | 00000000 | 00000000 | 00000000 |
| Sensor ID (0-15) and a rolling sequence number for each new measurement (0-15) | High byte of a two byte unsigned integer | Low byte of a two byte unsigned integer | A check sum (sensor id + seq num + data) |

Thus, sensor values between 0 and 65535 can be sent.

My protocol also supports sending signed float values with two decimals. In that case I multiply the value by 100 and store the data in two bytes. The highest bit is a sign flag that indicates if the value is positive or negative. This way, float values between -327,67 and +327,67 can be sent.

| Byte 3 | Byte 2 | Byte 1 | Byte 0 |
|---|---|---|---|
| 0000 0000 | 00000000 | 00000000 | 00000000 |
| Sensor ID (0-15) and a rolling sequence number for each new measurement (0-15) | High byte of a two byte float value + sign flag on highest bit | Low byte of a two byte float value | A check sum (sensor id + seq num + data) |

By looking at the sensor id, the receiver knows if the data should be treated as an unsigned integer or a signed float value (so the sender and the receiver have to agree on what the different sensor id:s mean). By extending the sender- and

receiver side, additional data types can be implemented – e.g. a signed two byte integer (values between -32767 and +32767).

## Arduino implementation

The sketch for the Arduino Pro Mini in the Arduino IDE looks like this:

```
1   //
2   // An Arduino sketch for an IoT node that broadcasts sensor values via
3   // 433 MHz radio signals
4   // The RCSwitch library is used for the transmissions
5   // The Narcopleptic library is used for power save during delay
6   // Sensor values are fetched from an BPM180/085 sensor via i2C
7   //
8
9   #include <Wire.h>
10  #include <Adafruit_BMP085.h>
11  #include "RCSwitch.h"
12  #include <Narcoleptic.h>
13
14  #define CLIENT_NAME "TopFloorClient"
15  #define TX_PIN 10                     // PWM output pin to use for transmission
16
17  //
18  // Sensor setup
```

```
19 // The BMP085 module measure ait pressure and temperature and operates via i2C
20 //
21 Adafruit_BMP085 bmp; // pin 4, SDA (data), pin 5, SLC (clock)
22
23 //
24 // Data transmission setup
25 //
26 #define TOPFLOOR_TEMP_ID    1
27 #define BMP_PRESSURE_ID     2
28 RCSwitch transmitter = RCSwitch();
29
30 void setup()
31 {
32   Serial.begin(9600);
33
34   bmp.begin();
35
36   transmitter.enableTransmit(TX_PIN);
37   transmitter.setRepeatTransmit(25);
38 }
39
40 unsigned long seqNum=0;
41 void loop()
42 {
43   float temp = bmp.readTemperature();
44   Serial.print("Temperature = ");
45   Serial.print(temp);
46   Serial.println(" *C");
47
48   unsigned int encodedFloat = EncodeFloatToTwoBytes(temp);
49   unsigned long dataToSend = Code32BitsToSend(TOPFLOOR_TEMP_ID,seqNum,encodedFloat);
```

```
50    TransmitWithRepeat(dataToSend);
51
52    float pressure = bmp.readPressure();
53    unsigned int pressureAsInt = pressure/100;
54    Serial.print("Pressure = ");
55    Serial.print(pressureAsInt);
56    Serial.println(" hPa");
57    dataToSend = Code32BitsToSend(BMP_PRESSURE_ID,seqNum,pressureAsInt);
58    TransmitWithRepeat(dataToSend);
59
60    for (int i=0; i< 100; i++)
61    {
62      // Max narcoleptic delay is 8s
63      Narcoleptic.delay(8000);
64    }
65
66    seqNum++;
67    if (seqNum > 15)
68    {
69      seqNum = 0;
70    }
71  }
72
73
74  unsigned long Code32BitsToSend(int measurementTypeID, unsigned long seq, unsigned long d
75  {
76      unsigned long checkSum = measurementTypeID + seq + data;
77      unsigned long byte3 = ((0x0F & measurementTypeID) << 4) + (0x0F & seq);
78      unsigned long byte2_and_byte_1 = 0xFFFF & data;
79      unsigned long byte0 = 0xFF & checkSum;
80      unsigned long dataToSend = (byte3 << 24) + (byte2_and_byte_1 << 8) + byte0;
```

```
81
82      return dataToSend;
83  }
84
85  // Encode a float as two bytes by multiplying with 100
86  // and reserving the highest bit as a sign flag
87  // Values that can be encoded correctly are between -327,67 and +327,67
88  unsigned int EncodeFloatToTwoBytes(float floatValue)
89  {
90    bool sign = false;
91
92    if (floatValue < 0)
93      sign=true;
94
95    int integer = (100*fabs(floatValue));
96    unsigned int word = integer & 0XFFFF;
97
98    if (sign)
99      word |= 1 << 15;
100
101   return word;
102 }
103
104 void TransmitWithRepeat(unsigned long dataToSend)
105 {
106     transmitter.send(dataToSend, 32);
107     Narcoleptic.delay(2000);
108     transmitter.send(dataToSend, 32);
109     Narcoleptic.delay(2000);
110 }
```

view raw

**TopFloorClient_RC_433MHz.ino**

hosted with by **GitHub**

The sketch uses the Narcoleptic library to save power while idling. As the maximum delay time for this library is 8 seconds, I need a loop around repeated delay calls to achieve an idling time of around 15 minutes.

The rolling sequence number for each measurement is used so that the receiver can detect duplicate data. To increase the likelihood of a message reaching the receiver, the message is sent several times. If a receiver gets a sequence of identical data (including the same sequence number), then it knows that the messages are duplicates and not just the same sensor values sent at different points in time.

## Raspberry Pi implementation

For the propagator node implementation, there are some prerequisites:

- Python 2.* installed
- The Python pi-switch library installed
- A running MQTT broker somewhere (I use a mosquitto broker running on

the Raspberry Pi)

I have a PropagatorApplication that acts as an IoT-propagator node. It is started with a *runserver.py* script:

```
#!/usr/bin/env python
from propagatornode.propagatorapplication import PropagatorApplication

if __name__ == '__main__':
    wiringPiPinForReceiver = 2
    brokerIP = "192.168.1.16"
    brokerPort = 1883
    app = PropagatorApplication(wiringPiPinForReceiver,brokerIP,brokerPort)
    app.run()
```

view raw

runserver.py

hosted with by GitHub

The script initializes the application with the pin to use for the 433 MHz receiver and the address for the broker where the transformed messages should be published.

The application defines what chirp messages to listen to and what MQTT topic they should be mapped to. A RadioListener class is used for fetching and decoding the radio chirp messages and an MQTTpublisher class is used for publishing the transformed message:

```
1   #
2   # A propagator node in an MQTT System
3   # It listens on messages/chirps via 433MHz radio and translates them to
4   # MQTT packages that are published over TCP/IP to a broker
5   #
6
7   from measurementtype import MeasurementType
8   from MQTTpublisher import MQTTpublisher
9   from radiolistener import RadioListener
10  import time
11
12  class PropagatorApplication:
13
14      wiringPiPinForReceiver = 2
15      brokerIP = ""
16      brokerPort = 1883
17
18      def __init__(self,wiringPiPinForReceiver,brokerIP,brokerPort):
19          self.wiringPiPinForReceiver = wiringPiPinForReceiver
20          self.brokerIP = brokerIP
21          self.brokerPort = brokerPort
```

```
22
23      def run(self):
24          # Defines the radio listener that uses pi-switch to listen to messages
25          # over 433 MHz radio
26          validMeasurementTypes = [
27              MeasurementType(1,"Temp","float","Home/TopFloor/Temperature"),
28              MeasurementType(2,"Pressure(hPa)","int","Home/TopFloor/Pressure"),
29              MeasurementType(3,"DoorOpened","int","Home/FrontDoor/Status")
30              ]
31
32          radioListener = RadioListener(self.wiringPiPinForReceiver,validMeasurementTypes)
33
34          # Defines the publisher that publishes MQTT messages to a broker
35          publisher = MQTTpublisher(self.brokerIP,self.brokerPort)
36
37          while True:
38              if radioListener.newMessageAvailable():
39                  message = radioListener.getLatestMessage()
40                  if message is not None:
41                      # Take the radio message and publish the data as an MQTT message
42                      publisher.postMessage(message.getTopic(),str(message.getValue()))
43
44              time.sleep(1)
45
```

view raw

**propagatorapplication.py**

hosted with ❤ by **GitHub**

The RadioListener uses pi-switch for listening to radio messages and doing the

bit operations needed for decoding a message:

```
1  from radiomessage import RadioMessage
2  from measurementtype import MeasurementType
3  from pi_switch import RCSwitchReceiver
4
5  class RadioListener:
6      validMeasurementTypes = []
7      previousValue = 0
8      numIdenticalValuesInARow = 0
9      latestMessage = None
10     receiver = RCSwitchReceiver()
11
12     def __init__(self,wiringPiPinForReceiver,validMeasurementTypes):
13         wiringPiPinForReceiver
14         self.validMeasurementTypes = validMeasurementTypes
15         self.receiver.enableReceive(wiringPiPinForReceiver)
16
17     def newMessageAvailable(self):
18
19         if self.receiver.available():
20             value = self.receiver.getReceivedValue()
21             self.latestMessage = self.getMessageFromDecodedValue(value)
22             self.receiver.resetAvailable()
23
24             if (self.latestMessage is None):
25                 return False
26             else:
```

```
27                  return True
28
29      def getMessageFromDecodedValue(self,value):
30          if value == self.previousValue:
31              self.numIdenticalValuesInARow += 1
32          else:
33              self.numIdenticalValuesInARow = 1
34
35          # decode byte3
36          byte3 = (0xFF000000 & value) >> 24
37          typeID = int((0xF0 & byte3) >> 4)
38          seqNum = int((0x0F & byte3))
39
40          # decode byte2 and byte1
41          data = int((0x00FFFF00 & value) >> 8)
42
43          # decode byte0
44          checkSum = int((0x000000FF & value))
45
46          # calculate simple check sum
47          calculatedCheckSum = 0xFF & (typeID + seqNum + data)
48
49          # Sanity checks on received data
50          correctData = True
51          if calculatedCheckSum != checkSum:
52              correctData = False
53          elif seqNum > 15:
54              correctData = False
55
56          message = None
57          if correctData:
```

```
58                 self.previousValue = value
59                 if self.numIdenticalValuesInARow == 2:
60                     # only store a value if an identical value was detected twice
61                     # if detected more than two times, ignore the value
62                     measurementType = self.getMeasurementTypeFromId(typeID)
63                     if measurementType is None:
64                         # invalid typeID
65                         print("Invalid type id")
66                         self.latestMessage = None
67                     else:
68                         message = RadioMessage(measurementType, data)
69 
70             return message
71 
72     def getMeasurementTypeFromId(self,typeID):
73         measurementType = next(i for i in self.validMeasurementTypes if i.id == typeID)
74         return measurementType
75 
76     def getLatestMessage(self):
77         return self.latestMessage
78 
```

view raw

radiolistener.py

hosted with by GitHub

The MQTTpublisher class wraps the code needed for publishing a message to an MQTT broker:

```
1   import paho.mqtt.client as mqtt
2   import time
3
4   class MQTTpublisher:
5       brokerIP = ""
6       brokerPort = 0
7       def __init__(self,brokerIP,brokerPort):
8           self.brokerIP = brokerIP
9           self.brokerPort = brokerPort
10
11      def postMessage(self,topic,message):
12          print("Publishing message " + message + " on topic " + topic)
13          # Initialize the client that should connect to the Mosquitto broker
14          client = mqtt.Client()
15          connOK=False
16          print("Connecting to " + self.brokerIP + " on port " + str(self.brokerPort))
17          while(connOK == False):
18              try:
19                  print("try connect")
20                  client.connect(self.brokerIP, self.brokerPort, 60)
21                  connOK = True
22              except:
23                  connOK = False
24              time.sleep(2)
25
26          client.publish(topic,message)
27          print("Publish done")
28          client.disconnect()
```

```
29
```

view raw

**MQTTpublisher.py**

hosted with ❤ by **GitHub**

The complete code for this experiment can be fetched from GitHub: https://github.com/LarsBergqvist/IoT_chirps_to_MQTT

# Conclusions

Using an IoT-end device without IP connection has several benefits. It is less expensive and consumes less power than a device with a TCP/IP-stack. The device is also resilient to hacking (the radio messages can be compromised, but depending on the application and the smartness of the propagator nodes, this might not be an issue).

By aggregating and filtering data from many devices at one or a few propagator nodes, it is easier to adapt the system (what messages to actually store etc) and let's you have one single point where data are pushed out to the Internet (this approach is beneficial regardless if TCP/IP or radio messages are used for the IoT-end devices).

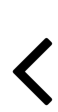
Due to the simplicity of the proposed radio protocol, it will not suite all applications. To send larger chunks of data, a more complex end device is needed. But, when a large "swarm" of devices for simple measurements is needed, I prefer using radio-based IoT-end devices.

POSTED IN ARDUINO, C++, ELECTRONICS, IOT, MQTT, PYTHON, RADIO, RASPBERRY PI, TINKERING •
TAGGED 433 MHZ, ARDUINO PRO MINI, BMP180, I2C, IOT, IOT ARCHITECTURE, IOT CHIRP MESSAGE, IOT-NODE, LOW-COST IOT-NODE, LOW-POWER IOT-NODE, MQTT, PI-SWITCH, PROPAGATOR NODE, PYTHON, RASPBERRY PI, RCSWITCH, RETHINKING THE INTERNET OF THINGS, SELF-HOSTED IOT, TELEMETRY

---

## Published by Lars

View all posts by Lars



**PREV**

Processing and Arduino

**NEXT**

Preparing the remote control app for Christmas

## One thought

Pingback: A sensor monitor with OLED in MicroPython – Thinkering & Tinkering

## Leave a Reply

HOME ABOUT LARS MUSIC PROJECTS LARS ON GITHUB LARS ON LINKEDIN

Blog at WordPress.com.