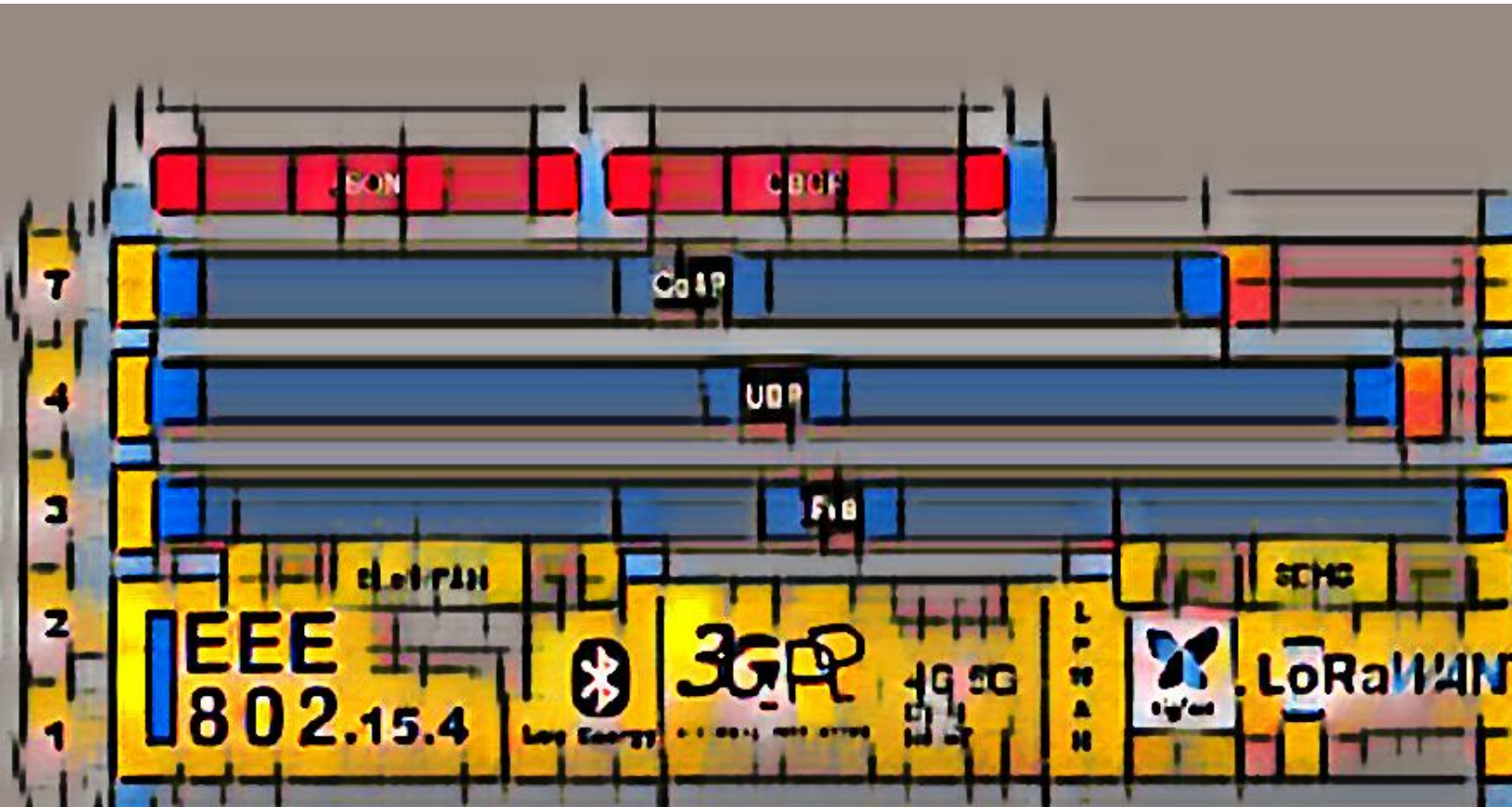
PROGRAMMING THE INTERNET OF THINGS

Laurent TOUTAIN



IMT ATLANTIQUE

Based on the PLIDO MOOC.
Published 20 avril 2022

3GPP 3rd Generation Partnership Project
ABP Authentication By Personalisation
ADSL Asymmetric Digital Subscriber Line
AMQP Advanced Message Queuing Protocol
AS Application Server
ASCII American Standard Code for Information Interchange
BLE Bluetooth Low Energy
CBOR Concise Binaire Object Representation
CoAP Constrained Application Protocol
Cosem Companion Specification for Energy Management
CRC Cyclic Redundancy Check
CSV Comma Separated Values
DLMS Device Language Message Specification
DTT Digital Terrestrial Television
DR Data Rate
GSMA GSM Association
HTML HyperText Markup Language
HTTP HyperText Transport Protocol
HTTPS HyperText Transport Protocol Secure
IANA Internet Assigned Numbers Authority
IBAN International Bank Account Number
IEEE Institute of Electrical and Electronics Engineers
IETF Internet Engineering Task Force
IoT Internet of Things
IP Internet Protocol
IPv4 Internet Protocol version 4
IPv6 Internet Protocol version 6
IPSO IP for Smart Objects
ITU International Telecommunication Union
IRI International Resource Identifier
ISBN International Standard Book Number
ISO International Standardization Organization
JMS Java Messaging Service
JSON JavaScript Object Notation
JSON-LD JavaScript Object Notation for Linked Data
LCIM Levels of Conceptual Interoperability Model
LPWAN Low Power Wide Area Network
LwM2M Lightweight Machine to Machine
LNS LoRaWAN Network Server
MQTT Message Queuing Telemetry Transport
NAT Network Address Translation
NGW Network GateWay
NIDD Non IP Data Delivery
OMA Open Mobile Alliance
OTAA Over The Air Authentication
OVH On Vous Herbèrge
PAC Porting Authorization Code
REST REpresentational State Transfer
RFC Request For Comments

RGW Radio GateWay
RNIPP Répertoire National d'Identification des Personnes Physiques
RSSI Received Signal Strength Indicator
RTT Round Trip Time
SCEF Service Capability Exposure Function
SenML Sensor Measuring List
SCHC Static Context Header Compression
SF Spreading Factor
SNR Signal to Noise Ratio
SSID Service Set IDentifier
STIC Sciences et Technologies de l'Information et de la Communication
TCP Transmission Control Protocol
TLV Type Length Value
TNT Télévision Numérique Terrestre
TTN The Things Network
UDP User Datagram Protocol
UIT Union internationale des télécommunications
UNB (Ultra Narrow-Band
URI Universal Resource Identifier
URL Univeral Resource Locator
URN Univeral Resource Name
VPS Virtual Private Server
W3C World Wide Web Consortium
WWW World Wide Web
XML Extensible Markup Language
XMPP Extensible Messaging Protocol et Presence

The virtual sensors that we have programmed so far emit data each time they have a measurement. This allows the server to follow the behavior of the studied system in real time. But in some cases, real time is not necessary and it is preferable to limit the number of emissions, for example to save the energy of the sensor.

To do this, we can use an array that will accumulate the values and send it when the array reaches a certain size.

Youtube

1.1 Sending an array

The program `minimal_humidity1.py` accumulates the data in an array `humidity` when this one reaches 30 elements (line 17), the data are sent to the server.

Listing 1.1 – minimal_humidity1.py

```
1  from virtual_sensor import virtual_sensor
   import time
3  import socket
   import cbor2 as cbor
5
   humidity    = virtual_sensor(start=30, variation = 3, min=20, max=80)
7
   s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
9  NB_ELEMENT = 30
   h_history = []
11
   while True:
13
       h = int(humidity.read_value()*100)
15
       # No more room to store value, send it.
17     if len(h_history) >= NB_ELEMENT:
           s.sendto (cbor.dumps(h_history), ("127.0.0.1", 33033))
```

```
19          h_history = []

21      h_history.append(h)
        print (len(h_history), len(cbor.dumps(h_history)), h_history)
23
        time.sleep(10)
```

```
1 4 [3241]
2 7 [3241, 2945]
3 10 [3241, 2945, 2762]
4 13 [3241, 2945, 2762, 2625]
5 16 [3241, 2945, 2762, 2625, 2480]
6 19 [3241, 2945, 2762, 2625, 2480, 2769]
```

The first digit of the line indicates the number of elements and the second the size in the CBOR coding. We notice that adding an element increases the size of the array by 3 bytes. The values corresponding to a moisture measurement do not vary greatly. Thus an array of 30 measurements has a size of 92 bytes.

1.2 Differential coding

The amount of data transferred can be optimized by using delta coding (i.e. the variation in humidity). The first value in the table is the measured value while the following values represent the difference between the measured value and the previous one.

Listing 1.2 – minimal_humidity2.py

```
    if len(h_history) == 0:
18      h_history = [h]
    elif len(h_history) >= NB_ELEMENT:
20      print ("send")
        s.sendto (cbor.dumps(h_history), ("127.0.0.1", 33033))
22      h_history = [h]
    else:
24      h_history.append(h-prev)

26  prev = h
```

The program `minimal_humidity2.py` handles the filling of the array differently :

— line 14 and 15, if the table is empty, the table is created with the measured value,
— line 16 to 22, otherwise if the table is full, it is serialized in CBOR and sent to the server, then reset with the measured value,
— line 23 and 24, otherwise the difference between the previous value and the measured value is stored in the table.

The following listing gives an example of execution.

```
1 4 [2521]
2 6 [2521, 79]
3 8 [2521, 79, 224]
4 10 [2521, 79, 224, -40]
5 12 [2521, 79, 224, -40, -112]
6 13 [2521, 79, 224, -40, -112, 1]
```

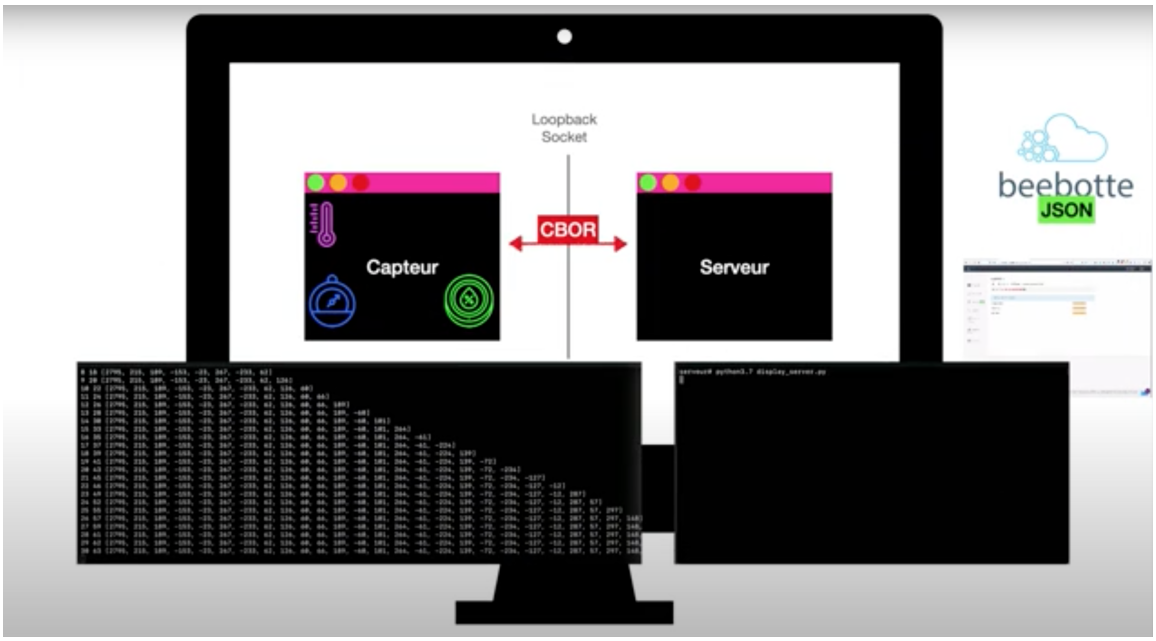


FIGURE 1.1 – Client/Server Architecture

```
7  15  [2521, 79, 224, -40, -112, 1, 130]
8  18  [2521, 79, 224, -40, -112, 1, 130, -288]
9  21  [2521, 79, 224, -40, -112, 1, 130, -288, 299]
```

This highlights two flexibilities of CBOR :

— the size of the table is dynamic. If the number of values to be transmitted is changed, the array indicates this and there is no need to modify the receiver code ;

— the size of the data depends on its value. For variations between -24 and +23, only one byte will be necessary. We can see it on the previous example : the addition of the value '1' in the table increases the size of the CBOR representation from 12 to 135 bytes. The values between 256 and +255 are transmitted on 2 bytes ; it is thus possible in this way to optimize the transmission without adding constraints. If there was a sudden change in humidity, the CBOR representation would adapt to transmit it.

The size is reduced by one third (about 66 bytes) to transmit the same information.

1.3 Architecture

Figure **??** represents the general architecture of the system. The program $\texttt{minimal}_h umidity2.py$ provides the time series. It remains to define the server program which will process them and call another service to display them in the form of a graph.

If we follow the information flow, the sensor will produce data in CBOR format to be compacted and the server program will transform this information into a JSON structure respecting the specifications of the display service.

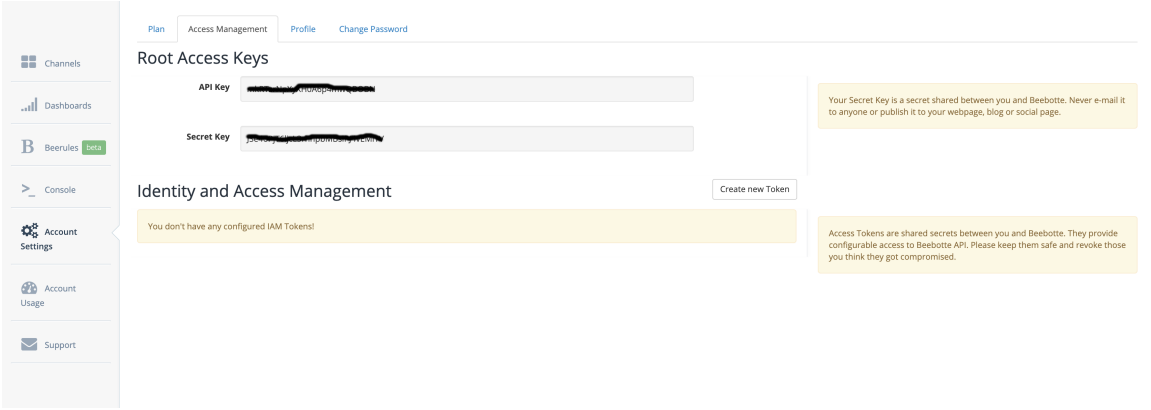


FIGURE 1.2 – Key and secret for authentication

1.4 Beebotte

There are several sites that allow you to do this. We are going to use `https://beebotte.com`, but what we are going to present can very well be applied to other sites.

1.4.1 Configuration

The first step is to create an account by clicking on *Sign Up* on the front page and filling out a standard form with your login, email address and password. Once the account is validated, the service is accessible.

The account allows us to authenticate ourselves to manage the data on the site, but we also need to have authorization to be able to deposit data via the **API REST**. To do this, you must go to the *Account Setting* page and then the *Access Management* tab. This page (see figure **??**) gives a key and a secret to manage all the data on the site.

Note these values and store them in a file called `config_bbt.py` that looks like this (your values are bound to be different) :

Listing 1.3 – config_bbt.py

```
1 API_KEY    = "GAJ3SFmUZSXmB2zqdcmzcuXc"
  SECRET_KEY = "4NCsrM1cfmFdMZF4E47aTfmCaU3UfyQo"
```

We are now going to create a channel (*channel*) in which we will define the objects corresponding to the sensors. By clicking on *Channels* and then *Create New*, the page shown in figure reffig-new-channel appears.

You have to give a name to the channel (*sensors* in the example), check the *public* box and create three resources for the three values we are interested in (*temperature*, *humidity*, *presure*) and match the units.

1.4.2 Resource registration

The program `display_server.py` allows to correspond with Beebotte via its REST API. It starts by importing the necessary modules :

Listing 1.4 – display_server.py

```
import socket
import binascii
import cbor2 as cbor
import beebotte
import config_bbt #secret keys
import datetime
import time
import pprint
```

— line 4, the Python module `beebotte` is available to simplify the manipulation of data [1].
— line 5, the module `contains` the key and the secret necessary to the connection obtained previously.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('0.0.0.0', 33033))
```

— line 10 and 11 allow to open the socket to communicate with the sensors.

```
bbt = beebotte.BBT(config_bbt.API_KEY, config_bbt.SECRET_KEY)
```

— line 13 an instance allowing the connection with the Beebotte servers is defined thanks to the function `BBT`. The connection parameters from the module `config_bbt` are taken into account.

```
while True:
    data, addr = s.recvfrom(1500)

    j = cbor.loads(data)
    to_bbt("capteurs", "temperature", j, factor=0.01)
```

In the main program, an endless loop waits for the CBOR coded time series coming from the sensor (line 42), transforms them into a Python array (line 44) and calls the function `to_btt` by specifying :

— the channel and resource that were previously defined on Beebotte ;
— the time series ;
— the precision to transform these integers into floats.

```
def to_bbt(channel, res_name, cbor_msg, factor=1, period=10, epoch=None):
    global bbt

    prev_value = 0
    data_list = []
    if epoch:
        back_time = epoch
    else:
        back_time = time.mktime(datetime.datetime.now().timetuple())

    back_time -= len(cbor_msg)*period

    for e in cbor_msg:
```

1. If it was not present on your computer, you should install it with the command `pip3 install beebotte`.

```
28          prev_value += e

30          back_time += period

32          data_list.append({"resource": res_name,
                              "data" : prev_value*factor,
34                            "ts": back_time*1000} )

36      pprint.pprint (data_list)

38      bbt.writeBulk(channel, data_list)
```

The function `to_bbt` does most of the transformation work. It takes as argument :

— the name of the channel created on Beebotte. In our case, it will be `sensors` ;
— the name of the object in this channel that we have also created on the website. In our case, it will be `humidity` ;
— the Python table of delta-coded measurements ;
— the multiplicative factor, i.e. the precision. Here, you will have to divide by 100 ;
— the period between two measurements ; this will allow us to calculate the time of the measurement. By default, the period is 10 seconds ;
— the time of reception of the message to date the samples. If it is not specified, the current time is taken.

This function transforms the following Python table :

```
[3311, 124, -144, -188, -94, 289, -1, -72, 1 ...
```

into a dictionary table :

```
[{'data': 33.11, 'resource': 'humidity', 'ts': 1596730115000.0},
 {'data': 34.35, 'resource': 'humidity', 'ts': 1596730125000.0},
 {'data': 32.91, 'resource': 'humidity', 'ts': 1596730135000.0},
 {'data': 31.03, 'resource': 'humidity', 'ts': 1596730145000.0},
 ...
```

Each dictionary contains three elements imposed by Beebotte :

— the name of the resource (`resource`) as defined on the interface for the channel ;
— the associated value for this resource (`data`) ;
— the time at which this measurement was made (`ts`). The time is represented according to the format **Epoch** which counts the number of seconds since the first of January 1970 [2].

The calculation of the timestamp (`ts`) is the most complex operation of this function but the modules `time` and `datetime` facilitate the calculation. If the argument `epoch` has been provided at the time of the call, the function takes this value, otherwise it calculates it on line 23. The function `now` returns the current date and time, which is transformed into a tuple by the function `timetuple`. From the latter, the function `maketime` converts it into epoch.

Line 25, the epoch at which the first measurement of the array was made is calculated by taking the current time (this assumes that processing and transmission time are neglected) minus the

2. see `https://www.epochconverter.com/` for the conversions

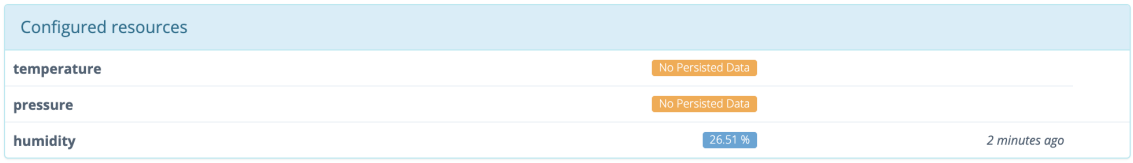


FIGURE 1.3 – Resource status

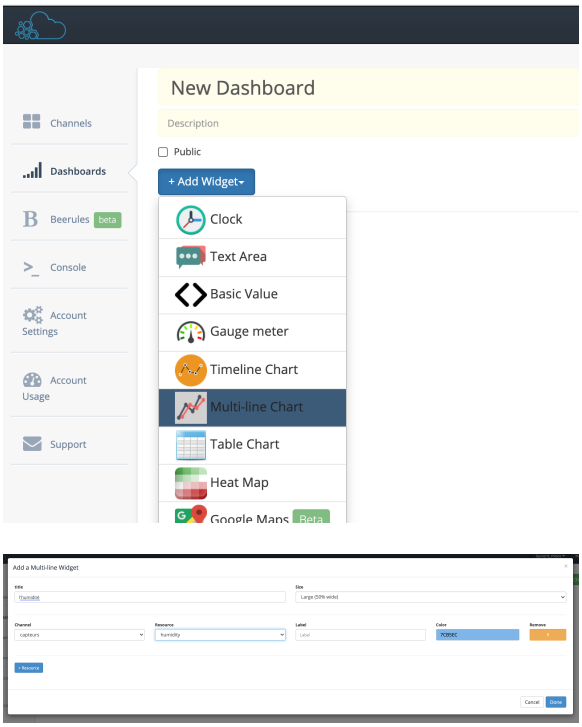


FIGURE 1.4 – Creating a widget

duration of the capture, i.e. as the number of elements in the array multiplied by the interval between each measurement (`period`).

Line 32 to 34 the structure expected by Beebotte is built. The result is sent, line 38, thanks to the function `writeBulk` which allows to send a set of values in an array.

We can verify that Beebotte has received data by viewing the sensor channel on the web interface. We can see on the figure **??** that only the resource `humidity` has received data. The interface displays the last value received and the date of reception.

1.4.3 Resource visualization

Now that the resources are stored in the Beebotte servers, it is possible to visualize them graphically, by going to *Dashboard* and then *create Dashboard* and *Add Widget* to select a widget such as *Multi-line chart*.

Then, configure the widget by setting the channel and the resource for that channel as shown in figure **??**.

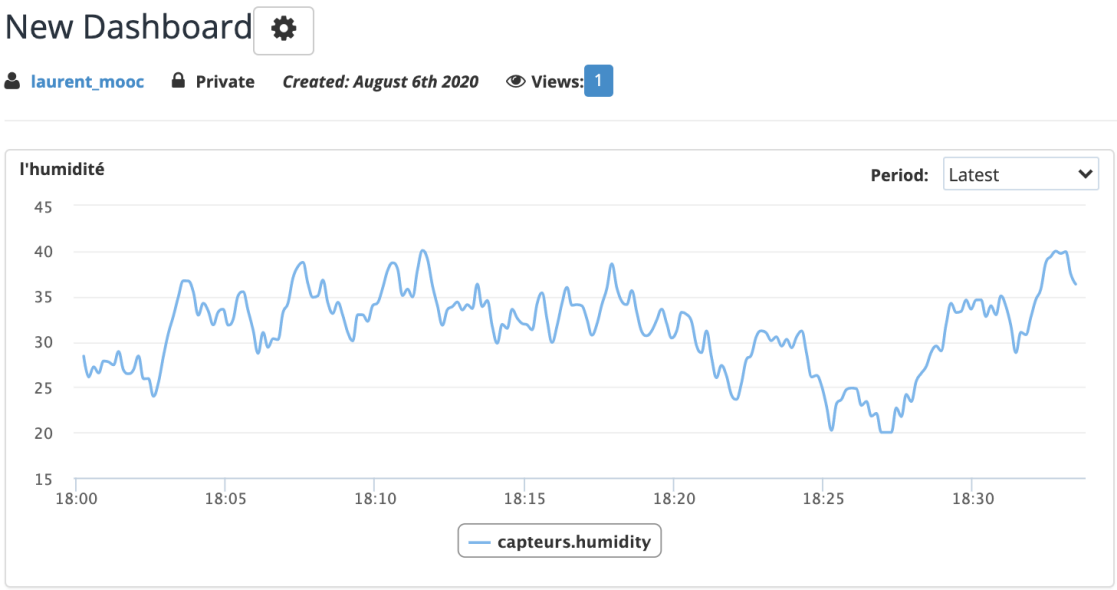


FIGURE 1.5 – Humidity monitoring

By going back to the dashboard, we can see the evolution of the humidity over time (see figure **??**).

1.5 Interoperability

The chain of information collection that we have just built, from the sensor to the display, is not completely interoperable. Certainly the sensor sends data in CBOR format that can be interpreted by the other end, but the receiver does not know :

—
—
—
—

This information was specified in the program `display_server.py`, as well as the transformation of the table structure of the time series into a dictionary with keywords specific to Beebotte was engraved in the program.

We will see later on how to improve this interoperability.

1.6 and SenML ?

In the communication with Beebotte, the site structures the sending of measurements by defining a JSON dictionary with particular keywords. To use another site, the exchange format must be modified even if the information remains identical.

In addition, when configuring the resources on the Beebotte site, the nature of the measurement had to be specified ; for example, whether it is a temperature, a humidity level... It is also sometimes

necessary to indicate the type of measurement (text, integer, float...) and even the units.

SenML! (**SenML!**) defined in the RFC 8428 structures the data provided by the sensor. To reduce the impact of the transmission, the field names have been chosen to be as compact as possible. For example, the letter `v` will indicate a value (to be compared with the key `data` used during the communication with Beebotte). To be even more compact, the representation in CBOR will use short integers instead of characters.

It is also possible to transport the unit of measurement with the keyword `u` .

SenML does not only define units of the international system, but also secondary units to limit the size of the representation. It will be more compact to transmit :

```
{"u": "MHz", "v": 868}
```

than

```
{"u": Hz", "v": 868000000}.
```

The standard also defines base times and base values to which the times and values will refer; this also makes it possible to reduce the size of the values. Finally, the object(s) can be identified in the transmitted data by defining a base name (`bn` : *base name*), the name of the sensor (`n` : *name*) completes the base name.

Sending

Listing 1.5 – minimal_senml_client.py

```
from virtual_sensor import virtual_sensor
import time
import socket
import json
import kpn_senml as senml
import pprint
import binascii
import datetime
import time
import pprint


NB_ELEMENT = 5


temperature = virtual_sensor(start=20, variation = 0.1)
pressure    = virtual_sensor(start=1000, variation = 1)
humidity    = virtual_sensor(start=30, variation = 3, min=20, max=80)

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

while True:
    pack = senml.SenmlPack("device1")
    pack.base_time = time.mktime( datetime.datetime.now().timetuple())

    for k in range(NB_ELEMENT):
        t = round(temperature.read_value(), 2)
        h = round(humidity.read_value(), 2)
        p = int(pressure.read_value() *100) # unit is Pa not hPa
```

```
31
        rec = senml.SenmlRecord("temperature",
33          unit=senml.SenmlUnits.SENML_UNIT_DEGREES_CELSIUS,
            value=t)
35      rec.time = time.mktime(datetime.datetime.now().timetuple())
        pack.add(rec)
37
        rec = senml.SenmlRecord("humidity",
39          unit=senml.SenmlUnits.SENML_UNIT_RELATIVE_HUMIDITY,
            value=h)
41      pack.add(rec)

43      rec = senml.SenmlRecord("pressure",
            unit=senml.SenmlUnits.SENML_UNIT_PASCAL,
45          value=p)
        pack.add(rec)
47
        time.sleep(10)
49
        pprint.pprint(json.loads(pack.to_json()))
51      print ("JSON␣length:␣", len(pack.to_json()), "bytes")
        print ("CBOR␣length:␣", len(pack.to_cbor()), "bytes")
53
    s.sendto(pack.to_cbor(), ("127.0.0.1", 33033))
```

The program `minimal_senml_client.py` illustrates how SenML works. It is based on two objects :

— the object `SenmlPack` includes information common to the object, such as the base name (here `device1` line 23) or the time base, line 24.

— the object `SenmlRecord` contains a measure where we can specify its name, its unit and its value (lines 32, 38 and 43). The time is also specified on line 35. These records are added to the object `pack`.

The program retrieves the three values of temperature, humidity and pressure (lines 27 to 29) by rounding them to 2 digits after the decimal point for temperature and humidity and converts the pressure from hecto Pascal to Pascal since this is the unit defined by SenML.

Measurements are made every 10 seconds (delays line 48) and when the number of measurements defined on line 13 is reached, the SenML coding in CBOR is sent to the server.

```
[{'bn': 'device1', 'bt': 1640110457.0,
  'n': 'temperature', 't': 0.0, 'u': 'Cel',    'v': 19.98},
 {'n': 'humidity', 'u': '%RH', 'v': 28.46},
 {'n': 'pressure', 'u': 'Pa', 'v': 100093}]
JSON length:  177 bytes
CBOR length:  104 bytes
```

This first listing shows the first record for the three measured quantities. It is a table of 3 elements. The first contains the basic values (here the name and the reference time) followed by the quantity to be measured, its unit and its value. The second and third elements update the name of the quantity, its unit and its value, the other information previously defined remains valid.

```
[{'bn': 'device1', 'bt': 1640110457.0,
 'n': 'temperature', 't': 0.0, 'u': 'Cel', 'v': 19.98},
 {'n': 'humidity', 'u': '%RH', 'v': 28.46},
```

```
 {'n': 'pressure', 'u': 'Pa', 'v': 100093},
 {'n': 'temperature', 't': 10.0, 'u': 'Cel', 'v': 20.03},
 {'n': 'humidity', 'u': '%RH', 'v': 26.86},
 {'n': 'pressure', 'u': 'Pa', 'v': 100065}]
JSON length:  318 bytes
CBOR length:  188 bytes
```

When new measurements are added 10 seconds later, a relative time of 10 seconds is indicated for the temperature recording and it remains valid for the following recordings.

Question 1.6.1: codage

A quoi correspond la clé `'u' : 'Cel'` que l'on retrouve dans la structure précédente ?

Question 1.6.2: Accroissement

Dans les deux représentations JSON et CBOR, de combien la taille est-elle accrue par l'ajout des mesures effectuées ? d'où viennent ces différences ?

Question 1.6.3: Une seule grandeur

Si on ne s'intéressait qu'à une seule grandeur, par exemple l'humidité. A quoi ressemblerait la structure SenML en JSON ?

Réception

Le traitement par le module SenML tel qu'il est mis en œuvre n'est pas complet, il ne gère pas correctement les timestamps. Mais, il n'est pas vraiment nécessaire pour traiter ces messages. En effet, comme on l'a vu précédemment, les objets SenML sont cumulatifs, une clé reste présente dans les enregistrements suivants sauf si elle est redéfinie.

Listing 1.6 – minimal_senml_server.py

```
import socket
import pprint
import binascii
import pprint
import cbor2 as cbor

import beebotte
import config_bbt #secret keys

naming_map = {'bn': -2, 'bt': -3, 'bu': -4, 'bv': -5, 'bs': -16,
                      'n': 0, 'u': 1, 'v': 2, 'vs': 3, 'vb': 4,
                      'vd': 8, 's': 5, 't': 6, 'ut': 7}

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('0.0.0.0', 33033))

bbt = beebotte.BBT(config_bbt.API_KEY, config_bbt.SECRET_KEY)


while True:
    data, addr = s.recvfrom(1500)
```

```
23      sml_data = cbor.loads(data)

25      sml_record = {}
        bbt_record = []
27
        for e in sml_data:
29          sml_record = {**sml_record, **e} # merge dict
            print (sml_record)
31
            ts =  sml_record[naming_map["t"]]
33          if naming_map["bt"] in sml_record:
                ts += sml_record[naming_map["bt"]]
35
            res = sml_record[naming_map["n"]]
37
            data = sml_record[naming_map["v"]]
39          if naming_map["bv"] in sml_record:
                data += sml_record[naming_map["bv"]]
41


43          bbt_record.append({"resource" : res, "data": data, "ts": ts*1000})

45      pprint.pprint (bbt_record)
        channel = sml_record[naming_map["bn"]]
47      bbt.writeBulk(channel, bbt_record)
```

Le programme `minimal_senml_server.py` va convertir le format SenML codé en CBOR dans le format attendu par Beebotte. La version CBOR utilise ds nombres plutôt que des tags. Le dictionnaire `naming_map` defini lignes 10 à 12 permet la correspondance utilisé par la suite pour rendre le code plus lisible.

Les lignes 14 à 17 initialisent les communications venant du capteur et celles allant à Beebotte.

Les données reçues ligne 21 sont transformée en structure Python ligne 23. Cette correspondance est possible car Python autorise des clés numériques et celles-ci ne sont pas répétées plusieurs fois dans une map CBOR.

La boucle commençant ligne 28 permet d'explorer tous les éléments du tableau SenML, les nouvelles entrées sont fusionnées avec les anciennes (ligne 29) [3].

Les informations concernant le temps sont ensuite recherchées. D'abord le temps (ligne 32) et s'il un temps de base existe (ligne 33) il est ajouté. On procède de même pour la valeur (lignes 38 à 40). Pour le nom, il n'y a pas de concaténation car le nom de base sera utilisé comme canal Beebotte, il est récupéré à la fin ligne 46.

A partir de ces informations, la structure attendue par Beebotte est construite ligne 43 en ajoutant le dictionnaire dans le tableau `bbt_record`.

Ligne 47, l'information est envoyée à Beebotte. Si les clés d'authentification, le nom du canal et des ressources sont correct, les information s'affiche sur le site, comme précédemment.

```
{0: 'temperature', 1: 'Cel', 2: 19.44, 6: 0.0, -3: 1640168049.0, -2: 'device1'}
{0: 'humidity', 1: '%RH', 2: 24.13, 6: 0.0, -3: 1640168049.0, -2: 'device1'}
{0: 'pressure', 1: 'Pa', 2: 101080, 6: 0.0, -3: 1640168049.0, -2: 'device1'}
{0: 'temperature', 1: 'Cel', 2: 19.48, 6: 10.0, -3: 1640168049.0, -2: 'device1'}
{0: 'humidity', 1: '%RH', 2: 21.85, 6: 10.0, -3: 1640168049.0, -2: 'device1'}
{0: 'pressure', 1: 'Pa', 2: 101090, 6: 10.0, -3: 1640168049.0, -2: 'device1'}
{0: 'temperature', 1: 'Cel', 2: 19.57, 6: 20.0, -3: 1640168049.0, -2: 'device1'}
{0: 'humidity', 1: '%RH', 2: 21.09, 6: 20.0, -3: 1640168049.0, -2: 'device1'}
```

3. Dans les version plus récentes de Python, il est possible d'utiliser l'opérateur |.

```
{0: 'pressure', 1: 'Pa', 2: 101058, 6: 20.0, -3: 1640168049.0, -2: 'device1'}
{0: 'temperature', 1: 'Cel', 2: 19.52, 6: 30.0, -3: 1640168049.0, -2: 'device1'}
{0: 'humidity', 1: '%RH', 2: 20.39, 6: 30.0, -3: 1640168049.0, -2: 'device1'}
{0: 'pressure', 1: 'Pa', 2: 101120, 6: 30.0, -3: 1640168049.0, -2: 'device1'}
{0: 'temperature', 1: 'Cel', 2: 19.44, 6: 40.0, -3: 1640168049.0, -2: 'device1'}
{0: 'humidity', 1: '%RH', 2: 21.94, 6: 40.0, -3: 1640168049.0, -2: 'device1'}
{0: 'pressure', 1: 'Pa', 2: 101128, 6: 40.0, -3: 1640168049.0, -2: 'device1'}
[{'data': 19.44, 'resource': 'temperature', 'ts': 1640168049000.0},
 {'data': 24.13, 'resource': 'humidity', 'ts': 1640168049000.0},
 {'data': 101080, 'resource': 'pressure', 'ts': 1640168049000.0},
 {'data': 19.48, 'resource': 'temperature', 'ts': 1640168059000.0},
 {'data': 21.85, 'resource': 'humidity', 'ts': 1640168059000.0},
 {'data': 101090, 'resource': 'pressure', 'ts': 1640168059000.0},
 {'data': 19.57, 'resource': 'temperature', 'ts': 1640168069000.0},
 {'data': 21.09, 'resource': 'humidity', 'ts': 1640168069000.0},
 {'data': 101058, 'resource': 'pressure', 'ts': 1640168069000.0},
 {'data': 19.52, 'resource': 'temperature', 'ts': 1640168079000.0},
 {'data': 20.39, 'resource': 'humidity', 'ts': 1640168079000.0},
 {'data': 101120, 'resource': 'pressure', 'ts': 1640168079000.0},
 {'data': 19.44, 'resource': 'temperature', 'ts': 1640168089000.0},
 {'data': 21.94, 'resource': 'humidity', 'ts': 1640168089000.0},
 {'data': 101128, 'resource': 'pressure', 'ts': 1640168089000.0}]
```

Le listing précédent montre cette transformation. Les premières lignes correspondent aux enregistrements fusionnées et le tableau final, ce qui a été envoyé à Beebotte.

Question 1.6.4: base value

Pourrait-on utiliser le champ SenML *base value* pour diminuer la taille des données de pression atmosphérique ?

Question 1.6.1 page 16 A quoi correspond la clé `'u' : 'Cel'` que l'on retrouve dans la structure précédente ?

Unité = degrés Celcius

Question 1.6.2 page 16 Dans les deux représentations JSON et CBOR, de combien la taille est-elle accrue par l'ajout des mesures effectuées ? d'où viennent ces différences ?

Si l'on regarde le listing précédent, l'ajout des trois mesures fait augmenter la taille de 141 octets pour JSON et 84 pour CBOR. La différence vient de l'utilisation de nombre plus que de chaînes de caractères pour les clés. Ainsi 't' demande 3 caractères en JSON avec les guillemets, codé en CBOR, il faudrait 2 octets, un nombre inférieur à 23 se code sur un seul octet. Il y a egalement les virgules, espaces et fermeture de crochets qui ne sont pas présent en CBOR. Les nombres flottant comme `19.98` ont une représentation plus compacte en JSON (5 octets) qu'en CBOR où ils consomment 9 octets. Dans tous les cas l'accroissement est fortement dépendant du nom des éléments. Ici, il faut répéter à chaque fois `temperature`, `humidity` et `pressure`, soit 26 caractères.

Question 1.6.3 page 16 Si on ne s'intéressait qu'à une seule grandeur, par exemple l'humidité. A quoi ressemblerait la structure SenML en JSON ?

Chaque nouvelle entrée ajoute 35 octets à la structure : [{'bn' : 'device1', 'bt' : 1640110457.0, 'n' : 'humidty', 'u' : '%RH', 'v' : 28.46},
{'t' : 10.0, 'v' : 26.86},
{'t' : 20.0, 'v' : 26.96},
{'t' : 30.0, 'v' : 27.01}]

Question 1.6.4 page 18 Pourrait-on utiliser le champ SenML *base value* pour diminuer la taille des

données de pression atmosphérique ?

Cela serait possible, si cette ressource était envoyée seule.