Environmental context monitoring using custom built IoT system

Primož Kočevar, Marko Meža

Faculty of Electrical Engineering, University of Ljubljana
E-mail: pk6695@student.uni-lj.si, marko.meza@ldos.fe.uni-lj.si

Abstract

In this paper we present a way to build an affordable ESP8266 based Internet of Things (IoT) solution that collects data from different sensors and sends it wirelessly to the server on a PC. Server is based on Python and uses MongoDB database for storing the data.

Context awareness is a term usually associated with mobile devices, but with our IoT solution we can also provide context to other kinds of applications and machine learning problems. In this case we shall be talking about context from the environment or ambient.

1 Introduction

From the early days of technology, we always wanted to make machines or computers to be able to predict and decide. When a computer is trying to decide, it can always be helpful to have some context from which it can get extra information on the situation. With the advent of technology, we are seeing the rise of internet of things (IoT) solutions collecting a lot of information which we can now use to make context aware solutions.

Our goal is to design a user friendly context aware communicator for the elderly which can help them choose the right path and the kind of public transport they can take to the desired destination (e.g. to the market in the morning). In trying to figure out the right amount of walking distance and different other parameters, we need to know what kind of mood the user is in. If we want to set the mood of our user correctly, we need to have some information on the ambient in which they are currently staying. Therefore, we need to know at least the temperature, humidity, air pressure and light in the room, which we want to get from our microcontroller.

We can certainly say that similar IoT solutions already exist and are probably well known [1], but not one of them had all of the features we were looking for:

- Our solution should measure ambient parameters, influencing user's behavior. We selected a subset of parameters: humidity, temperature, air pressure and light. Required sampling interval for those signals should only be around 20-30 seconds.
- We wanted our solution to be built with the tools we were already familiar with in the laboratory, so it could merge well with the other components of the whole system for context aware communicator.
- Complete control over the whole IoT solution was also our priority, so we could completely understand its workings and constantly adapt it to our needs.
- Privacy is nowadays a big concern in developing internet technologies, and especially IoT privacy issues are being debated often [2]. In the light of these events, we figured that the best way to make our solution private is Privacy by Design (PbD). We can achieve that by making our solution work without the access to the Internet, with restricted communications only inside the LAN network. We implemented that by integrating multiple technologies which are not dependent on the Internet, instead of the ones that are (e.g. making our own mechanism for getting timestamps instead of using Network Time Protocol).

Now it is clear that all of these features could only be achieved by creating the solution internally.

The paper presents a solution through description of used hardware and software resources in the first Section. Implemented structure of the solution is presented in the Architecture Section. Finally, the Results Section describes achieved performance and stability of the solution, and also presents an example of the data gathered by the solution.

2 Materials and Methods

In this section we are going to describe hardware and software resources we used in building this IoT solution. Software and hardware tools were chosen carefully for our purpose, so they would meet the required features described in Section 1.

2.1 Hardware resources

First, we needed a source of information about the ambient that would bring us a context aware solution. For this we needed some sensors. We decided to use DHT22 sensor for measuring temperature and humidity, BMP180 sensor for measuring air pressure and a simple photoresistor for measuring the amount of light in the room.

DHT22 is a low cost digital temperature and humidity sensor. For measuring it uses a capacitive humidity sensor and a thermistor. Measured data is then sent out in a form of a digital signal on the data pin every two seconds, and that is why it needs careful timing. Two seconds time between the readings makes the sensor a slow one, but for our purpose it is fast enough. It can measure humidity from 0% to 100% with

2-5% accuracy and temperature from -40°C to 80°C with ±0.5% accuracy. Sensor works in 3-5V range on the I/O and power. More information about the sensor can be found in the datasheet [5].

BMP180 is a low-cost and low power barometric pressure sensor that also includes a temperature sensor. It can work with 3.3V or 5V since it includes a I2C level shifter circuit. It can measure barometric air pressure in the range from 30-110kPa with 0.006kPa accuracy. The Inter-Integrated Circuit (I2C) interface allows for easy system integration with a microcontroller. More information about the sensor can be found in the datasheet [6].

Photoresistor is a light-controlled variable resistor. If we measure the voltage drop on the photoresistor, ohm's law states that, with the constant current, we also measure the resistance. Therefore, with measuring the voltage drop on the analog input pin of our microcontroller, we also measure the amount of light on the photoresistor.

For the client side we needed a microcontroller which would collect data from mentioned sensors and send it to the PC server. First we tried using one of the Arduino boards which seemed the best supported, but then we found a cheaper and better wireless solution in the form of *ESP8266 NodeMCU 0.9* [3]. This is an open-source development kit based on the ESP-12 chip, which has 32Mbits of flash memory. The I/O pins on the NodeMCU kit work on 3.3V, which makes it compatible with our sensors. It contains 10 GPIO pins using Pulse Width Modulation (PMW) that can be configured to use I2C interface. It can also be
programmed and flashed through micro USB just like an Arduino.

2.2 Software resources

When choosing the software for our client ESP8266 NodeMCU we had a few options. Our version of the development kit was designed to work with NodeMCU firmware which provides a Lua development and execution environment, but we were not going to use it since it is still in the early stages of development and lacks support. Instead we have been using it with the *Arduino Integrated Development Environment (IDE)*, which is better supported and has plenty of libraries to choose from. More information on how to use the ESP8266 with the Arduino IDE can be found in the documentation [4]. However, this decision brought us a few challenges, because the ESP8266 is still a somewhat new development platform, and all of the libraries working on Arduino do not necessarily work with ESP8266. A list of some compatible Arduino libraries can also be found in the documentation [4].

Server part of the solution runs on a PC in the same LAN with the help of a framework in Python called *Bottle*. Bottle is a lightweight Web Server Gateway Interface, which is distributed as a single file module and has no dependencies other than the Python Standard Library [7].

For storing the data which is sent to our server, we decided to use *MongoDB* database with the help of the tools provided by *PyMongo*. *MongoDB* is classified as a NoSQL database and brings a different structure than the traditional table-based relational database. A measurement stored in MongoDB is a document, which is similar in structure to a JSON object and is named BSON [8]. PyMongo is a Python distribution that includes recommended tools for working with MongoDB database from Python [9].

To see and analyze the stored data we now needed to pull the data out of the database and draw a graph. The tool we used for that is a Python library *Pandas*. Pandas provides plotting and data-analysis tools through simple data structures such as DataFrame and Series. More information about the library and how to use it can be found in the documentation [10].

3 Architecture and description of the solution

In this section we are going to explain how we connected all the parts, from the sensors up to the data visualization, and how we built a working solution that can give us the results we wanted. The description written shall follow the architecture drawn in Fig. 1.

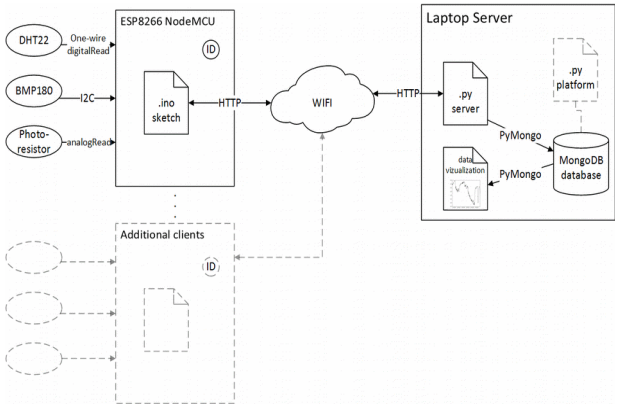

Figure 1: Architecture of the whole IoT solution. We could use it with multiple clients with different ID's. We could also use our database for context awareness in a future context aware communicator.

First we started connecting the sensors to ESP8266 development kit through different interfaces and GPIO's. Data pin from the temperature and humidity sensor DHT22 is connected to a digital input pin on a development kit, and with the combination of DHT22 Arduino library [11], which also controls the timing of the digital readings, we can get temperature and humidity readings every 2 seconds. The simplest one-wire communication is implemented in this case, therefore every additional sensor must be connected to a different pin. Photoresistor is connected to the single analog input on the development kit, and with measuring the voltage drop on the resistor, we can measure the amount of light in the room. The values returned from analog readings are between 0-1024, we only scale them down to 0-100% of light in the room, therefore we do not use any specific units. Sensor BMP108 has the most sophisticated communication with the development kit, as it uses I2C protocol. This protocol enables multiple devices connected to a single

bus, which can be really useful if we have multiple sensors that use I2C and only two I2C-enabled pins. First we needed to connect SDA and SCL pins on BMP180 to two I2C enabled pins on our development platform (specifically D1 and D2). To implement I2C protocol, we used BMP180 library combined with the examples on how to use it [12].

But only measuring the required data was not enough, we also needed to add a timestamp, when the measurement happened. Here we stumbled upon a small problem, either Arduino, or our ESP development kit, do not have an internal clock. Therefore, we should connect a Real Time Clock (RTC) Module to the I2C serial bus, and it would keep track of time even if there is no power. However, instead of adding another module to our board, we decided to get time from our server every time the module is reset, and save it as a *setSyncProvider(unixTime)* that runs the clock with the help of the function *millis()*. We also set *setSyncInterval(syncInterval)* to regularly update the time we have set. Described functions that provide easy manipulation with time are a part of Arduino Time Library [13].

The foundation for getting time from the server and then sending measured data to it is communication based on HTTP protocol, which is described on Fig. 2. Specifically, we use GET requests to send a request for time and also measured data packed in the Uniform Resource Identifier (URI).

Creating URI is the most important thing that contains all of the information relevant to the server. In order to get time from the server, we need to send the URI that contains ”/getDateTime”. But every few seconds (defined as dataInterval in Fig. 2) we want to send the measurement data, the timestamp and the device ID to the server, in which case we create a URI like this: *”/data?temperature="+String(t) +"&humidity="+String(h)+"&light="+String(l) +"&pressure="+String(p) +"&deviceid="+device+"&time="+String(sendingTim e);* [7]

On the server side, running bottle in a Python script, we are expecting two different URI’s. If the server gets a “/getDateTime” URI request, it sends back a response containing time in unix format.

But if it gets a request on URI “/data”, then it takes the containing information about the measurements out of the whole URI and saves them in a JSON-style document e.g.:

{“ Temperature”: request.query.temperature,
“Humidity": request.query.humidity , …} [7] [9].

If MongoDB is already running, we now only need to select a database and a collection in which we want to save our document and then save it with PyMongo (e.g. *collection.insert_one(document)) [9].*

Analysis and plotting of our data is done in a different Python script, and also works with the help of PyMongo and a Python plotting and analysis library Pandas. First we load a MongoDB collection into a list which we need to create a Pandas structure DataFrame.

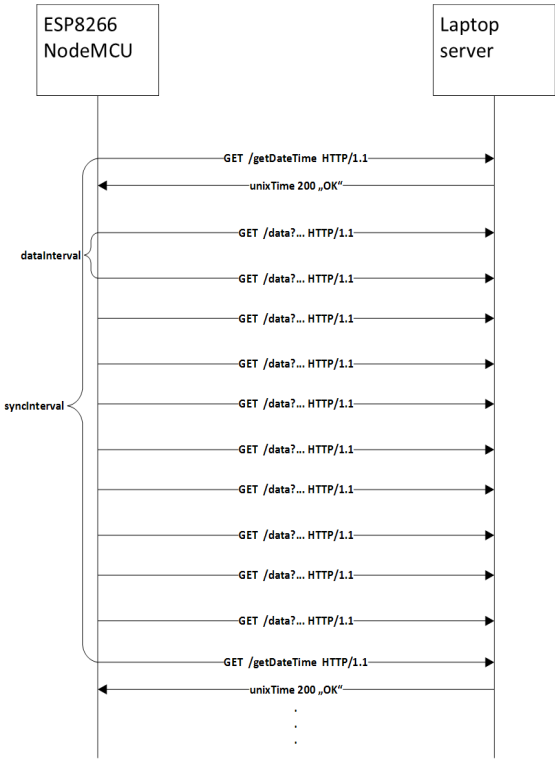

Figure 2: Communication diagram between our client ESP8266 and a laptop server running bottle. We can see how syncInterval and dataInterval (sampling time) affect the communication.

When we have our data stored in a DataFrame as a string, we can easily convert it to numeric or datetime format, so that we can analyze and plot it (e.g. *data.describe()* and *plt.plot(data.Temperature)*) [10].

4 Results

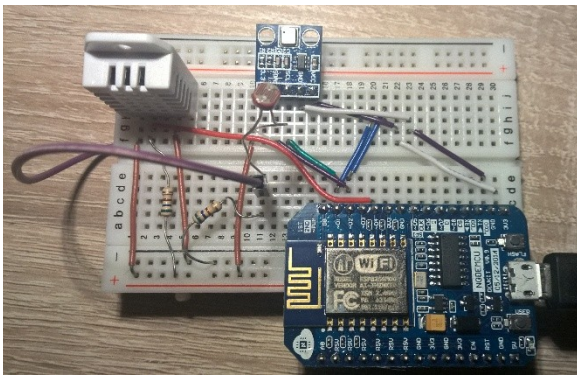

Figure 3: Client part of the solution assembled on a breadboard containing ESP8266 NodeMCU, DHT22, BMP180 and a photoresistor.

IoT platform on the client side was implemented, as shown in Fig. 3, on the breadboard which proved sufficient for our use, so there was no need for building a discrete circuit. The client part needs to be powered through micro USB, or by 3.3V on the VCC pin, therefore it is very simple to plug it in and use it in the same room as a PC that has the server running.

In testing our solution, we left it to gather data for approximately 12 hours and 21 minutes with sampling time set to 30 seconds. The results can be seen on Fig. 4

and Table 1. It gathered 1442 measurements that occupied 223,6 KiB of storage in the database.

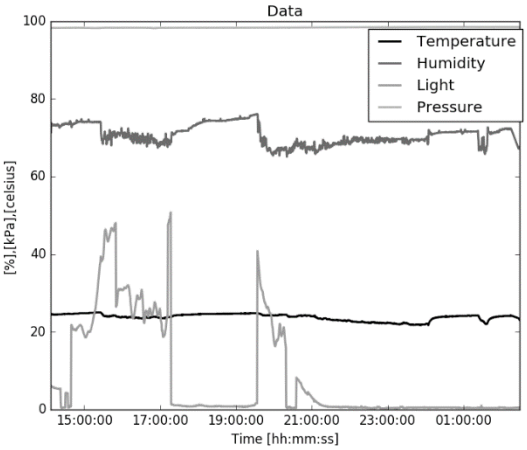

Figure 4: Data gathered in approximately 12 hours, with a sampling frequency of 30 seconds, and plotted in Python.

Table 1: Result of data analysis by pandas

	Humidity	*Light*	*Pressure*	*Temperature*
Count	1442.00	1442.00	1442.00	1442.00
Mean	70.76	8.52	98402.82	23.77
Standard Deviation	2.51	12.89	94.40	0.88
Minimum	65.10	0.29	98214.00	21.70
Maximum	76.10	50.78	98565.00	25.00

The best performance we could achieve was a sampling interval of 5 seconds, a faster sampling frequency caused a loss of data and unreliability of the communication. Reliability at faster sampling speeds would be difficult to achieve with this platform, because of the delays in HTTP communication. Even if we could send data faster, we would be limited by the slow sampling speeds (2 seconds) of our DHT22 sensor.

During our testing we have also noticed some stability issues. These issues were mainly caused by unpredictable crashes of the database after running for a few days. We did not notice any issues on the client side during our few-day test. Otherwise stability of the whole solution was satisfying as it ran 24 hours without a problem multiple times.

5 Conclusion and discussion

The solution described here is collecting useful data, with the help of the microcontroller ESP-8266 NodeMCU, from different sensors that measure temperature, humidity, air pressure and the amount of light in the room. We want to use this data in machine learning, with the combination of other parameters, trying to predict different states of the user in a room. The state of the user then helps to recommend the best possible option for him in our context aware communicator described in Section 1.

When testing we also encountered some problems, like a relatively slow sampling interval of 5 seconds and some potential reliability issues if the platform is left to work for a whole week.

However, for our purpose the results are more than satisfying:

- The required sampling frequency is lower than the achieved one, since environmental parameters are relatively slow changing (e.g. Temperature).
- Our context aware communicator will only be interested in current values of the measurements, therefore a long-term (few weeks) reliability would not be necessary.
- Storing data in MongoDB database enables an easy integration of our solution with a future context aware communicator.

Future work will start with writing a Python script which will provide access for the platform to our MongoDB database, as can be seen in Fig.1. Further work will also include implementation of several nodes in order to monitor more complex setups.

References

[1] Web portal of ThingSpeak, available: https://thingspeak.com/, accessed: 27.6.2016.

[2] Rolf H. Weber: “Internet of Things – New security and privacy challenges”, Computer Law & Security Review, vol. 26, Issue 1, January 2010, Pages 23–30.

[3] NodeMCU wiki and documentation web portal, available: http://www.esp8266.com/wiki/doku.php?id=nodemcu, accessed: 27.6.2016.

[4] ESP8266 and Arduino online documentation: available: https://github.com/esp8266/Arduino, accessed: 27.6.2016

[5] Aosong(Guangzhou) Electronics Co., “Digital-output relative humidity & temperature sensor/module AM2303”, DHT22 datasheet.

[6] Bosch, “BMP180 Digital pressure sensor”, BMP180 datasheet, April 2013.

[7] Marcel Hellkamp, “Bottle Documentation”, Release 0.13-dev, June 26 2016.

[8] Online documentation MongoDB, available: https://docs.mongodb.com/manual/ , accessed: 27.6.2016.

[9] Online documentation PyMongo, available: https://api.mongodb.com/python/current/# , accessed: 27.6.2016.

[10] Wes McKinney & PyData Development Team, “pandas: powerful Python data analysis toolkit”, Release 0.18.1, May 03, 2016.

[11] Adafruit, “Arduino library for DHT22”, available: https://github.com/adafruit/DHT-sensor-library , accessed: 28.6.2016.

[12] Adafruit, “BMP180 Library”, available: https://github.com/adafruit/Adafruit-BMP085-Library , accessed: 28.6.2016.

[13] Paul Stofferegen, “Arduino Time Library”, available: https://github.com/PaulStoffregen/Time , accessed: 28.6.2016.