Pervasive Computing Project 1 - Group 4

Johan Arleth
IT University of Copenhagen
johaa@itu.dk

Peter Øvergård Clausen
IT University of Copenhagen
pvcl@itu.dk

Kenneth Ry Ulrik
IT University of Copenhagen
kulr@itu.dk

ABSTRACT

To shed light upon the daily squabbles that any plant-owner will recognize, namely when one's plants should be watered, we have deployed the I2C Water Moisture Sensor at a plant located at the IT University of Copenhagen, and have had the sensor be part of a node in a LoRaWan setting, sending data back to the `PitLab` gateway, and performing a qualitative analysis on the gathered measurements.

We have found that reading the moisture level of a plant is the primary aspect of a plant's health, and being able to tell an owner when it reaches a certain low level, would enable uninterested caretakers of plants to be prompted to water them, thus seamlessly enabling them to keep their plants alive for longer.

Keywords

Smart agriculture, LoPy, Python, LoRa, Plants, Earth moisture sensor, Pervasive computing, Ubiquituis computing.

1. INTRODUCTION

The daily-life overhead of taking care of a plant can be immensely stressful. When did I last water it? When is the optimal time to water it again? How can I - with a healthy mind - allow myself to water all my plants at once, when I know they have varying needs? When does this particular plant need to be watered next? Has someone else watered the plant for me?

These are all questions we wish to enlighten the world about. In particular, we will attempt to set up the scene for gathering data about a particular plant located right outside the `PitLab` [4] at the IT University of Copenhagen, Denmark, and perform a qualitative analysis attempting to shed light upon these daily nightmarish challenges mentioned above.

During this project we will design and implement a system that is able to monitor the well being of a plant, namely the plant's earth's moisture level, light intensity on the plant, and the temperature around it. These parameters will be sampled by our system over time, and be sent over a wireless connection using LoRa to a server, so that it can later be analysed.

2. SYSTEM OVERVIEW

A LoPy board [2] is mounted onto an expansion board, and an I2C Soil Moisture Sensor [1] is then connected to the expansion board pins as seen as in figure 1.

In particular, for power to the sensor, GND goes to GND and 3.3V goes to VCC. For communication with the sensor, the ports G16 and G17 on the expansion board are used and are connected to the "SDA MOSI" and "SCK SCL" pins on the sensor respectively.

The expansion board with the LoPy mounted has been deployed in a plastic container to contain the electronics safely, and a Wi-Fi antenna has been mounted to enhance the signal to the `PitLab` gateway.

The sensor is mounted halfway (as indicated on the device (not shown in figure 1)) into the plant's soil, and finally, power is supplied from a nearby power outlet through a USB cable.

This entire setup when mounted is then illustrated in figure 2.

2.1 LoPy board

A LoPy is a small micro-controller which supports 3 different communication protocols, namely Wi-Fi, Bluetooth Low Energy (BLE) and LoRaWan [3]. An image of a LoPy board can be seen in figure 3.

We use the LoPy board as our primary controller, where we have used Pymakr IDE [5] to upload our Python script to be executed by the LoPy micro-controller, which - via the LoPy expansion board - has the I2C Water Moisture Chirp Sensor connected for sampling of data, as described previously.

Expansion board

The LoPy board, as mentioned, is mounted on top of the expansion board, enabling various additional pins for accessing. An image of the expansion board can be seen in figure 4.

2.2 Sensors

We have utilized a soil moisture sensor from Catnip Electronics which supports measuring soil moisture, light intensity and temperature [1]. This sensor is a compact solution with all the needed sensors in one unit, and it supports soil insertion. The sensor is cheap, easy to use and has water-

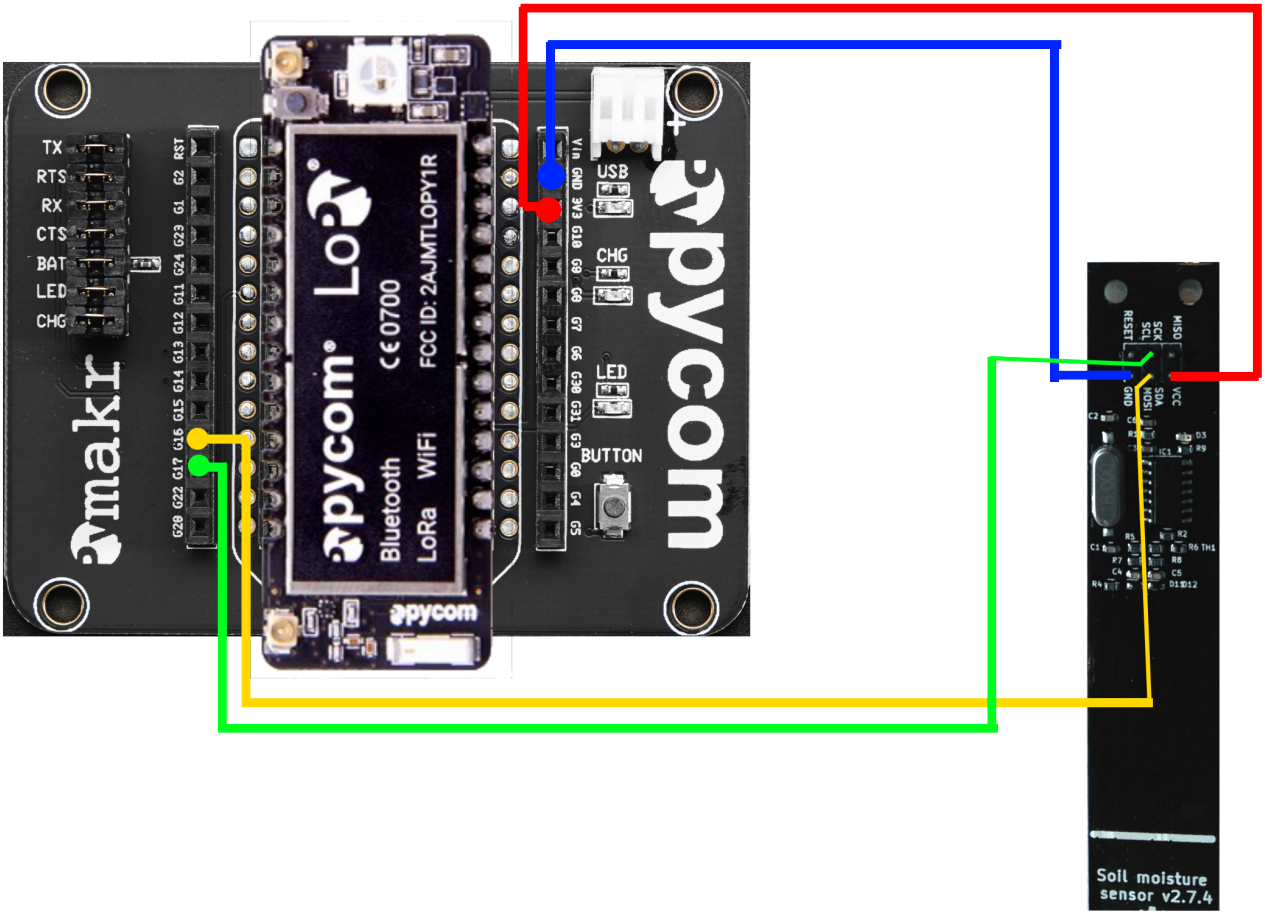

Figure 1: An illustration of the overall electronic setup, involving a LoPy board attached to an expansion board and an I2C Soil Moisture Sensor attached.

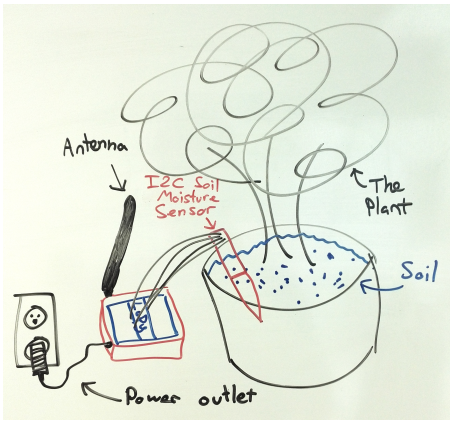

Figure 2: The full setup.

proofing so no special considerations are needed for use near water, as long as water is not poured onto the upper part of the sensor.

Figure 3: An image of a LoPy board.

These attributes together made us choose this sensor for our use case.

2.3 Software

The piece of Python code deployed to the LoPy can be seen in its entirety in Appendix: 8.1.

The goal of the code is to first attempt to establish a reliable connection (socket-to-socket) with the `PitLab` Lo-

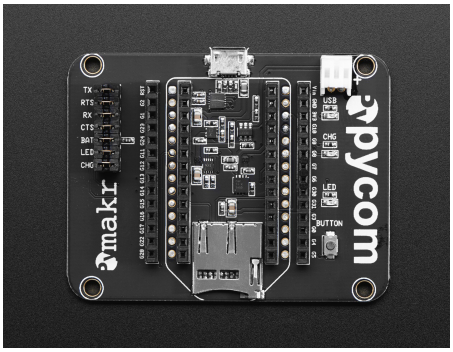

Figure 4: An image of an expansion board for a LoPy board.

RaWan gateway. In particular, the code retries connecting until it is successful in doing so.

Following this, the data-gathering can begin.

For all measurement types, we get a quite obscure value which we must first unpack, using our `evaluate` function, which returns an integer number. The values are interpreted in the following way:

- Temperature: We receive an integer value 10 times larger than the Celcius value for the measured temperature. For instance, if the measured temperature were 23.6 degrees Celcius, we receive the number 236.
- Moisture: We receive an integer in the range of potentially 0 at complete dryness to 672 when the sensor is enveloped in water.
- Light: For light, we receive a value between 0, in the case of maximum light and 65535 in the case of total darkness.

For all data we send a packet containing exactly 3 bytes for each of the units we measure, namely light, temperature and moisture. This is by convention of the `PitLab` LoRaWan gateway, because it requires receiving a designated ID for the type of measurement followed by two bytes representing the data. As a neat note, it just to happens that the maximum value for representing the amount of light measured, 65535, is exactly the maximum value that can be represented with 2 bytes.

Measuring all three units, we send 9 bytes of data in total, contained in 3 separate packets, for each sampling that is performed. It takes around 8 seconds to send all 3 packets synchronously.

Finally, the step of sending these packets is performed endlessly, with a sampling frequency of 10 seconds. It should be noted that the waiting time of the 10 seconds is not begun until after all the sampling packets have been sent.

2.4 Sampling and communication interval

A more frequent sampling and communication interval would have a higher battery consumption.

Therefore reducing the amount of sampling and communication rate per hour would result in a lower battery consumption, which would expand the time the battery lasts, when the system is deployed using battery as a power source.

However, if the LoPy fails to send it's data, there might be larger holes in the data collection.

While building the system, we had a sampling and communication rate of every 10 seconds. This can be used to see if the system works while developing it, however it uses too much power if the system is to run on battery.

We have chosen the sampling and communication rate of two times every hour to be sufficient in monitoring the state of the plant, and lowering the power consumption when the system is deployed with a battery as power source.

2.5 Power consumption model

To figure out the power consumption model, we need to measure how much power the device is using. First we will go through how we would have liked to measure it, and then how we ended up measuring it:

2.5.1 How we would have liked to measure current

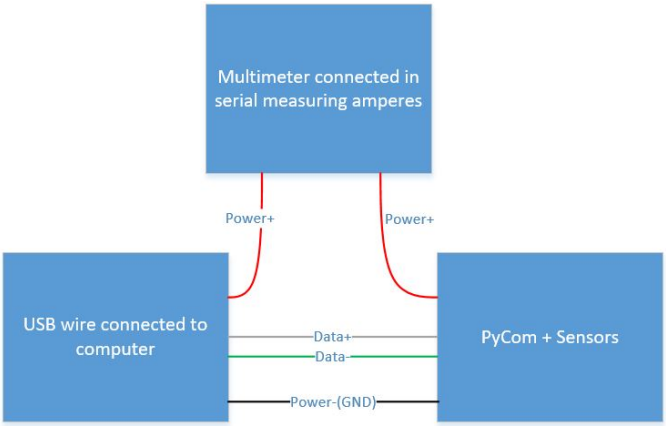

Figure 5: Measuring current (amperes) in serial through a USB power cord with a multimeter

2.5.2 How we would like to measure voltage

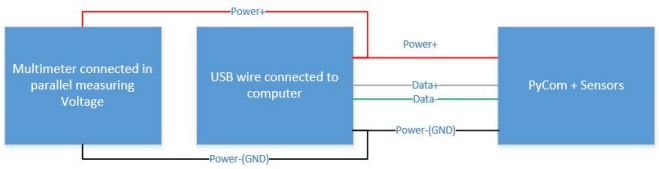

Figure 6: Measuring voltage in parallel with a multimeter

To measure the power consumption we could use a multimeter to measure amperes as seen in figure 5 and volts in figure 6. We could then get the amount of Watts by multiplying volts with amperes:

Amperes * Volts = Watts

2.5.3 How we ended up measuring it

Because we did not have a USB power cord to cut open in time, we ended up using a MUKER [1], which is a USB device that is able to measure Voltage, Current and MilliAmpHours (mAh) of a connected USB device over time. In figure 7, you

[1] Amazon.co.uk store page for a MUKER

Figure 7: Measuring power consumption with a MUKER USB.

will see our measurements over time. The measurements are taken after 5 minutes, 10 minutes and 15 minutes shown as T, and the mAh consumed by the device in that time is shown below that.

From this we are able to calculate the consumption (mAh) per minute:

5 minutes / 3mAh = 1.667 mAh per minute.

10 minutes / 7mAh = 1.43 mAh per minute.

15 minutes / 10mAh = 1.5 mAh per minute.

From this we can see that the device uses around 1.5 mAh per minute.

We can now figure out how much power would be consumed running the device in a month:

30 days * 24 hours * 60 minutes = 43200 minutes.

43200 minutes * 1.5 mAh = 64800 mAh.

Our power consumption of our current setup is 64800 mAh per month.

Or in watts that would be:

5 Volts * 0.04 Amperes = 0.2 Watts.

2.5.4 What we learned from this

We were expecting the device to use varying currents (amperes) based on when the device is sampling data, sending data using LoRa and sleeping, however from our measurements it looks like the power consumption is constant over time.

In part 2 of this assignment, we could experiment with using machine.sleep()[2] instead of time.sleep().

We tried using machine.sleep() instead of time.sleep() the last day of working on this assignment, however the pycom never continued operation after being put to sleep. We might be missing a method call in order to wake it up again as discussed in this forum post. However we did not have time to find out what the cause of this were.

We ended up using time.sleep() instead, but we might experiment with machine.sleep() in part 2 of the project.

3. SAMPLE FINDINGS AND ANALYSIS

By reading the recorded data from the system, we can see that the data is consistently sent and recorded. In figure 8 we can see the data samples over the course of a week. From the graph we see that the recorded values jump slightly up and down, suggesting that the sensor is not pinpoint accurate, but this does not matter since it is not important to know the exact values, just an approximate.

If we look at the red line representing moisture, we see that it has a downward trend, with occasional upwards spikes. This makes sense, since water is being consumed by the plant as well as evaporating from the soil, and the steep upwards spikes represents when the plant is being watered.

The purple line representing temperature remains pretty stable, only moving in a range of around 2-3 degrees Celsius, excluding the temperature drop on April the 9th.

The green line representing light intensity, which is going up and down in blocks, with a very stable line at night and a somewhat spiky line during the day. This matches with the lamp located over the plants being turned on and off for day and night time. At night we would expects nothing much is around to interfere with the readings, so it makes sense that we get a flat line. During the day, in addition to the lamp being turned on, we would have shifting light conditions depending on the sun, as well as people walking by and casting shadows, which accounts for the spikiness of the daytime readings.

Overall from the readings, we can conclude that the system is working and the readings are in line with expected trends, confirming that it is reliable.

By looking at the graph, we can see that after the plant has been watered, it takes between 3 and 4 days before the water level drops down to pre watering levels, at around 400. Looking at the graph, it would seem that watering the plant once the sensor reads around 400 in moisture level makes sense, and presuming it is watered until the sensor reads around 650-700, it is safe to water the plant every 4 days. This is assuming the environment at the IT-university, where a constant temperature of around 24-26 degrees is maintained.

4. OPERATING AUTONOMOUSLY

4.1 Battery

We figured out in section 2.5, that our system uses around 64800 mAh per month. A battery that should keep this system running for a month should be above 64800 mAh (to run the system in it's current state without machine.sleep()). This battery capacity could be lowered if the battery could be regularly charged by a sustainable energy source such as a solar panel.

[2] machine.sleep() documentation homepage

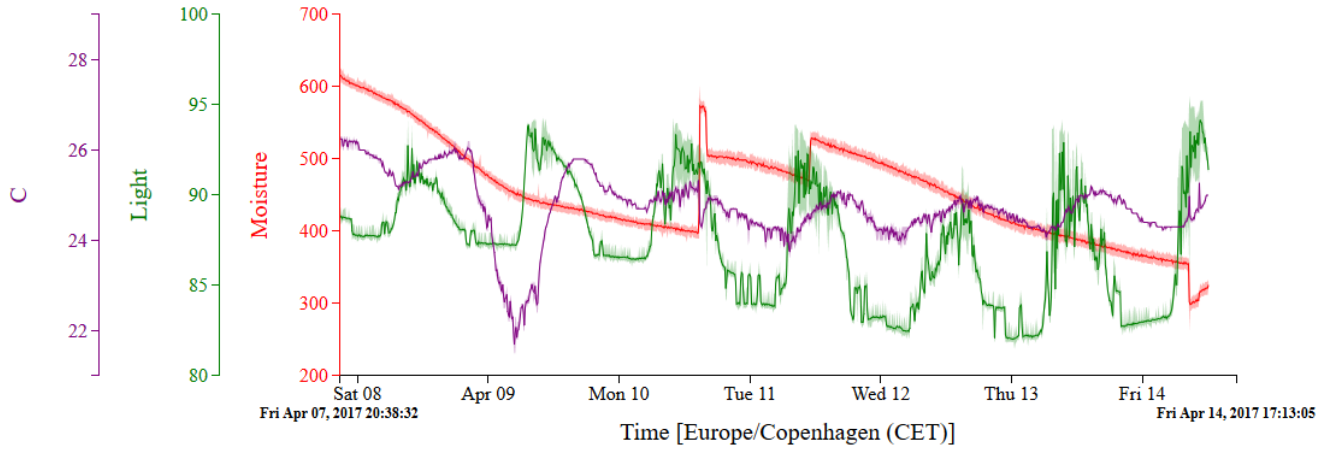

Figure 8: A graph of the data recorded over a week

4.2 Sustainable energy

It is possible to change the power supply to a sustainable source, by using a rechargeable battery.

The system is designed for indoor use, thus limiting the possible options for sustainable energy. Since we are working indoors, most likely at a window, the most reasonable solution is to use solar energy. The plant needs sunlight, so we already know we must be in a position with good light conditions. A solar panel mounted near the window and connected to a battery which the system draws power from should be able to provide power for the system.

Wind energy through a windmill is also an option.

5. DISCUSSION

Since we are using an unlimited power supply, we did not consider power saving, and as such we are sending data pretty much as often as possible, meaning around every 10 seconds. If we were using a battery, this would not make sense. The moisture level is unlikely to change very fast, so changing the sensing interval to something like every few hours would be an obvious power saver. Maybe as little as 2-3 times a day would be sufficient to keep tabs on the moisture. However some of these samples might be lost, hence we ended up thinking every half hour could be a good balance.

We are also maintaining a constant connection over lora, which could be optimized to only connect when we need to send data.

In addition, we could turn off LEDs and other superfluous functions on the lopy.

6. CONCLUSIONS

We can get consistent, realistic readings on soil moisture, temperature and light levels around the plant, and we may be able to plan watering from this recorded data, in particular from the measured moisture level.

From our analysis, we've seen that moisture is of course the main factor for the health of a plant, as is well known, and we've had some insight into university-employed people's plant care-taking abilities, namely having seen quite varying intervals between watering of the plant.

In regards to power consumption, we've seen that it is apparently difficult to make a Pycom device enter proper sleep-mode, in order to save power whenever the waiting period between the various samples (which involves sending packets of data) is going on. For now, our implementation uses `time.sleep` to simply stall the program. This does indeed not lower the power consumption, but if replaced with proper sleeping, it would play an essential role of reducing the power consumption of the device, especially with a much lower sampling frequency.

7. REFERENCES

References

[1] *I2C Soil Moisture Sensor*. URL: https://github.com/Miceuz/i2c-moisture-sensor/blob/master/README.md (visited on 04/17/2017).

[2] *Lopy datasheet*. URL: https://www.pycom.io/wp-content/uploads/2017/01/lopySpecsheetGraffiti2017newR.pdf (visited on 04/17/2017).

[3] *LoRa*. URL: https://www.lora-alliance.org/ (visited on 04/17/2017).

[4] *PitLab, IT University of Copenhagen*. URL: https://pitlab.itu.dk/ (visited on 04/17/2017).

[5] *Pycom's Pymakr IDE*. URL: https://www.pycom.io/pymakr/ (visited on 04/17/2017).

8. APPENDIX

8.1 Code

```
from machine import UART
from network import WLAN
import os

uart = UART(0, 115200)
os.dupterm(uart)
wlan = WLAN()
wlan.deinit()
```

Listing 1: boot.py

```
import binascii
import pycom
import socket
import time
from network import LoRa
from machine import I2C
from struct import unpack

# Colors
off = 0x000000
red = 0xff0000
green = 0x00ff00
blue = 0x0000ff

# Turn off heartbeat LED
pycom.heartbeat(False)

# Initialize LoRaWAN radio
lora = LoRa(mode=LoRa.LORAWAN)

# Set network keys
app_eui = binascii.unhexlify('70
    B3D57EF0003F19')
app_key = binascii.unhexlify('17162
    A7852BEFCB8C742C52671E45359')

# Join the network
lora.join(activation=LoRa.OTAA, auth=(
    app_eui, app_key), timeout=0)
pycom.rgbled(red)

# Loop until joined
while not lora.has_joined():
    print('Not joined yet...')
    pycom.rgbled(off)
    time.sleep(0.1)
    pycom.rgbled(red)
    time.sleep(2)

print('Joined')
pycom.rgbled(blue)

# Set up socket
s = socket.socket(socket.AF_LORA, socket.
    SOCK_RAW)
s.setsockopt(socket.SOL_LORA, socket.SO_DR,
     5)
s.setblocking(True)

def evaluate(a):
    v = unpack('<H', a)[0]
    return (v >> 8) + ((v & 0xFF) << 8)  #
    Divide by 10 to get Celsius value (e.g.
     from int 269 to 26,9 C)

def preparePacket(dataId, value):
    res = bytes ( [ dataId, (value & 0xFF00
    ) >> 8, (value & 0x00FF) ] )
    print (dataId, value)
    return res

sensorId = 0x20 # == 32

# Registers
moistureRegister = 0
temperatureRegister = 5
lightRegister = 4

#Receiver IDs
tempId = 0xAA
moistId = 0xBB
lightId = 0xCC

# Sensor readings
i2c = I2C(0, I2C.MASTER, baudrate=10000)

while True:
    # Continuously read data
    i2c.writeto(sensorId, '\x03') # Light
    is special
    time.sleep(1.5)
    aLight = i2c.readfrom_mem(sensorId,
    lightRegister, 2)
    aTemp = i2c.readfrom_mem(sensorId,
    temperatureRegister, 2)
    aMoist = i2c.readfrom_mem(sensorId,
    moistureRegister, 2)

    # Prepare byte arrays for all the
    sensor data
    temp = preparePacket(tempId, evaluate(
    aTemp))
    moist = preparePacket(moistId,
    evaluate(aMoist))
    light = preparePacket(lightId,
    evaluate(aLight))

    # Send the three values to the beacon
    countTemp = s.send(temp)
    countMoist = s.send(moist)
    countLight = s.send(light)

    # Wait an amount of time (sampling
    frequency)
    print('Sleeping')
    pycom.rgbled(green)
    time.sleep(10)
    print('Woke up')
    pycom.rgbled(blue)
```

Listing 2: main.py