Albert Garcia
Jannis Ihrig
Marian Sigler
Manar Zaboub

Software Project
Internet Communication
Final Report

FU Berlin, Summer Term 2018
Prof. Dr. Matthias Wählisch

Contents

Introduction **2**

Scope of Application **2**

Planning **2**
Planned Solution . . . 2
Milestones . . . 3

Implementation **3**
Sensors . . . 3
Network . . . 4
H2O Protocol . . . 5
Pump Control . . . 6
Gateway . . . 9
Data presentation . . . 10

Conclusion **10**

References **11**

Introduction

Scope of Application

In the following section we describe the scope of application for which our application was build for. We deduced it from the more general task of building a IoT application for planter pots given at the start of the course.

We assume several separate planter pots are given. These can have some distance between each other but are located in one general location.

The application to build should allow to monitor the humidity of each single planter pot separately and remotely, e.g. from a computer indoors. The parts of the system responsible for humidity monitoring, i.e. sensor nodes, should be easy to install, require low maintenance and so be able to remain with the planter pots or inside the soil for long periods of time. As an extra prerequisite specific for the course, we were asked to use the Atmel *SAM R21 Xplained Pro* MCU (see [4], SAMR21 in the following) as hardware for the sensor nodes. Another option was to use the Phytec *PhyNode* MCU (see [2]), but is was clear from the beginning of the course, that they were only available in limited numbers. Moreover, these MCUs should run *RIOT* as their operating system.

Additionally to humidity monitoring, the system should take care of watering the planter pots automatically.

Planning

Planned Solution

Sensor and pump control nodes To measure humidity of the single planter pots, we decided to use the SAMR21 MCUs as sensor nodes which should run the RIOT operating system. These should therefore each be equipped with an external humidity sensor to measure soil moisture. We regarded the integration of additional sensors as temperature, light or air humidity sensors as desirable but not a central part of the project. We also planned for the possibility of multi-hop routing between the sensor nodes to increase the maximum range in which they can be placed.

Additionally to monitoring purposes, one central goal described in the section *Scope of Application* was to water the planter pots. We therefore decided to introduce an additional node which should collect all sensor data and control a water pump. When turned on, the pump should water all plants collectively to simplify the setup.

The decision of using the SAMR21 as sensor nodes and pump controller had a big impact on the planned system architecture, as it has a low power consumption and offers wireless connectivity via 6LoWPAN over IEEE802.15.4. Together, this should increase battery lifetime and reduce the need for maintenance, as was one of our set goals.

Gateway node and web front-end Another node should receive data from the pump control node and publish it to a web server under our control. It therefore had to be able to connect to the local network via Ethernet or WiFi based on IEEE802.11. A web front-end should then allow to monitor all sensor data remotely.

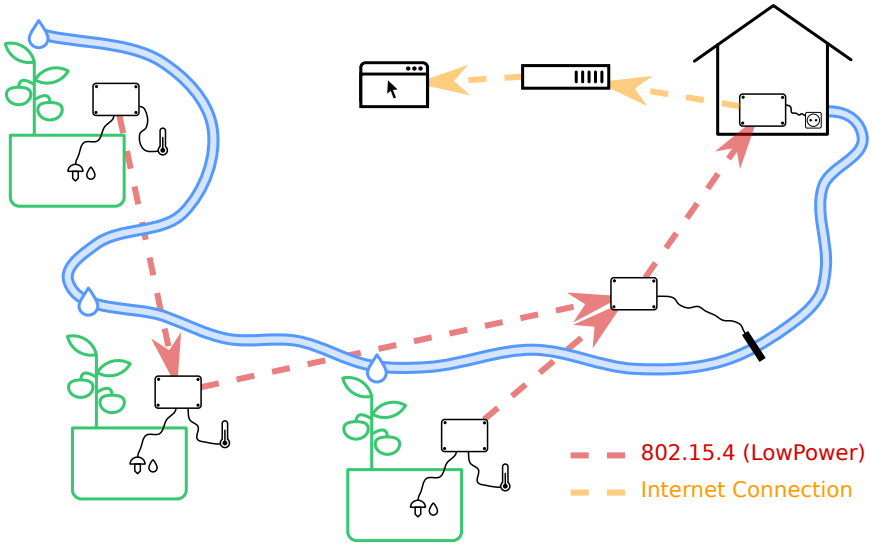

Figure 1: Overview of the general system architecture resulting from the analysis of the given task described in the previous section.

Milestones

We decided to set 3 Milestones for the project as follows:

M1:
- Implementation of the basic network functionality of the sensor nodes.
- Selection and basic integration of the soil moisture sensor for the sensor nodes.
- Basic web server working on mock sensor data.
- Basic pump control algorithm working on mock sensor data.

M2:
- Improved integration of sensors.
- Improved network stack.
- Integration of pump hardware.
- Improved pump control algorithm.
- Gateway node connecting pump node to web server.

M3:
- Full integration of sensor nodes, pump control node, gateway and web server / web interface.
- Watering based on real sensor data.
- Multi-hop routing for sensor nodes.

Implementation

Sensors

The *SEN0114* [5] sensor was selected for the central task of collecting data on soil moisture. This was decided due to the following reasons:

- There was previous experience with integrating the sensor with the SAMR21 and RIOT (see [6]).

- The integration via RIOT's ADC interface [1] seemed simple.
- The sensor uses a gold plating which leads good resistance against corrosion and provides longlivety.

A basic integration of the sensor was achieved during M1. To do so, we connected the analog output of the sensor with the SAMR21's GPIO PA06 which is one of the default inputs for RIOT's ADC interface. We used GPIO PA13 to supply power but only switched it on during measurements to further reduce the risk of corroding the sensor. This allowed us to read sensor data, although it still was uncalibrated.

As only few SAMR21 were available for the project and to extend the selection of sensors, the PhyNode's air humidity and temperature sensors were also integrated with the sensor nodes software.

During M2, a unified interface for all sensors, including mock sensor data for RIOT's 'native' platform was implemented.

Unexpectedly, it took until the end of M3 to collect sensible and calibrated sensor data with the SEN0114. This was due to the limited ADC interface of RIOT and its lacking documentation on how to modify the used reference voltage. By default, RIOT's ADC interface uses a reference voltage of 1 V. Experiments with the sensor showed signals between 0 V and 1.6 V. We therefore hat to set the reference voltage to 2 V, which can currently only be achieved by modifying the RIOT sources. We therefore had to fork the RIOT code.

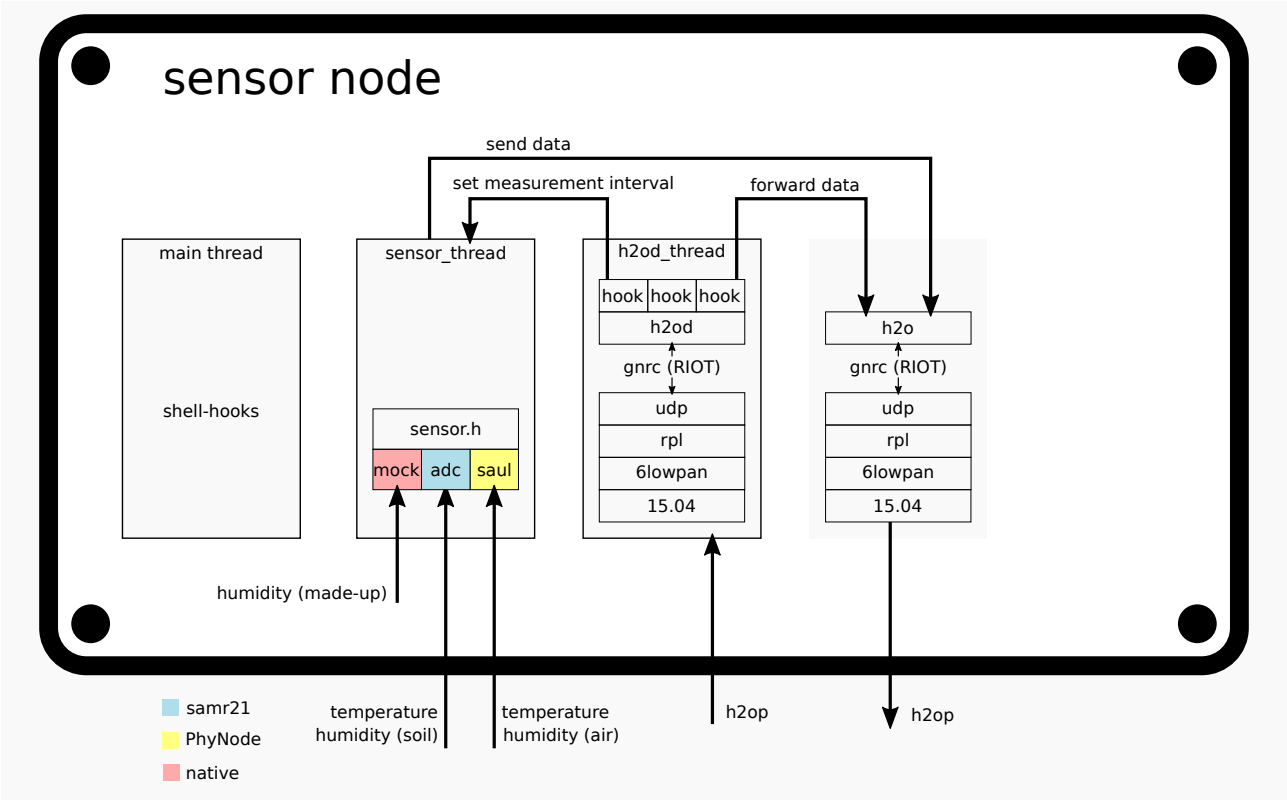

Figure 2: Implementation of the sensor node. It shows the sensor integration and the implemented network stack.

Network

The network layout we planned was based on the following assumptions:

1. The sensor nodes are connected via IEEE 802.15.4, they should be able to run from a battery or solar cell. Some of them may need to be positioned with some distance from the house.
2. As the pump needs a lot of power anyways, the pump controller node cannot run from battery but needs a power adapter connected to a wall socket. Water for the plants typically is also best available near a house, no matter if it is from a tap or from a tank storing rain water from the roofs.
3. To communicate data to a web server, we need a gateway that translates from the IOT 15.4 network to TCP/IP internet.

From (2) it follows that we can require the pump to be near a house. This means we can put the gateway required by (3) inside the house, where it has connection both to a WIFI router and the pump control node. Further, (1) dictates that we use hop-to-hop communication between the sensor and pump control nodes, because some of the nodes might be positioned too far away from the house. As the pump control needs to know the sensor data of all sensors, it makes sense to send all data to it and have it forward everything to the gateway. Because of this central role we are calling it the *controller* node. These considerations lead to the network layout shown in Figure 1.

As the 802.15.4 network is not directly connected to the internet, we decided to use a hard-configured unique local network from the fc00::/8 address range[7]. This also allows automatic hot-plug configuration of just a sensor and a controller, without an internet connection. In contrast to link-local addresses (fe80::/10), this allows us to use hardcoded, fixed addresses for some special nodes, namely the controller and the gateway.

Every node has a 16 bit Node ID. In the case of the controller and the gateway, those are hardcoded, for the sensor nodes one is generated deterministically from hardware properties (in fact, we use the least significant 16 bits of the interface identifier), but can also be assigned at compile time if desired. Each node's unique local address is deduced from this Node ID.

For the hop-to-hop network, we use a RPL tree, rooted at the controller node. However it is also possible to explicitly define a node's upstream node, should this prove more reliable. The upstream node, in turn, forwards the data on the application level to its own upstream node. This forwarding mode in fact is a remainder from the early development phase when we had not set up RPL yet, but we decided to keep it to increase flexibility.

H2O Protocol

When starting the project, we decided to implement our own simple UDP-based protocol, dubbed h2o protocol, in order to be able to quickly start the development of the other components without first evaluating different existing protocols. We had planned to possibly switch to an existing protocol (for example, CoAP) later, but in the end, our little protocol turned out flexible enough so we kept it. For example, adding the `set measurement interval` message type (see below) and the associated hooks could be done easily in a few minutes and lines of code. Still, if this was a longer ongoing project, the extra maintenance overhead caused by maintaining our own protocol would probably justify a migration.

Header Format

The h2o protocol is designed to run on UDP. Packages are very small to allow it to run on 802.15.4 which has a 127 byte payload limit. The package header is as follows (all offsets and lengths measured in bytes). See the file `extra/protokoll` for details (such as byte order and failure behavior).

Offset	Length	Type	Description
0	1	`uint8_t`	magic number (`0xac`)
1	1	`uint8_t`	version number (`0x01`)
2	1	`uint8_t`	packet length (including header)
3	1	`uint8_t`	message type (see below)
4	2	`nodeid_t` (`typedef uint16_t`)	node ID of the node the package originates from
6	2	`uint16_t`	Packet checksum
8	...	`uint8_t[]`	Data (depends on the message type, see below)

The node ID field is required to conserve the data's original author after the packet has been forwarded.

The message type consists of a supertype (most significant four bits) that defines the type of message and associated behavior (for example, the controller only forwards `DATA MESSAGE` packets to the gateway); and a subtype defining the exact contents.

Type		Description	Data Field Contents
`0x1?`		DATA MESSAGE	
	`0x11`	temperature	`int16_t`, degrees Celsius
	`0x12`	humidity	`int16_t`, 0 to 100
`0x2?`		DATA REQUEST (reserved)	
`0x3?`		CONFIGURATION	
	`0x31`	set measurement interval	`uint32_t`, microseconds
`0x9?`		WARN (currently unused)	
`0xA?`		INFO (currently unused)	

Server and Client Implementation

→ *See `lib/network.c` for details.*

Receiving is implemented in a very flexible way. The server runs in the background. It deals with basic packet parsing (for example, version and checksum verification), but does nothing more.

Functionality is implemented via hooks that can be registered dynamically (via the `h2op_add_receive_hook` and `h2op_del_receive_hook`) functions). This is how we configure different behavior for different node types, for example, only the controller node needs to pass sensor data to the pump control module. Each registered hook is run for every received packet, the hooks then check whether the packet is of the type they are interested in and acts accordingly. Internal behavior, such as the debugging of incoming packets, is also implemented as a hook.

To make the implementation more coherent, we also use this interface for internal communication within a node, for example for communicating the pump bucket's fill level to the pump controller.

Sending is trivial, it is implemented via the `h2op_send` function that gets passed all the variable header fields and a pointer to the data. `h2op_send` then constructs the packet, calculates the checksum and sends the packet. Because this is quite quick, it happens within the caller thread.

Pump Control

The Pump Control is the set of algorithms that manage the water pump. In this subsection we present the functions that are being used and the reason of the developing of these functions.

First of all we have to present the design of the pump controller, in this case we have implemented a threshold system which allows the controller to decide the next action of the pump. As we have two

types of sensors (the one that controls the water level of the pump and the rest) there are two types of thresholds with different values that are defined after a couple of practical tests.

The sensor's data is stored in an array (dubbed *table* in the following), stored globally.

Once the design is presented we can start to describe the different functions that are used in the pump control.

Function `reset_table( int table[ ][ ])`

This function fills the bidimensional array of 0 and also set to 0 different variables.

The function is used when the pump changes the state(open or close) or when all the normal sensors (the ones not in water) had sent data.

Function `print_table (int table[ ][ ])`

This function prints in the console the ID, the last value received and the time when the value was received of each sensor stored in the table. The function is used to control the correct functioning of the table and of the algorithm and to control that the pump is receiving the correct values of the different sensors. It is a purely informative function.

Function `initialize_pump()`

This function is used to initialize the GPIO that allows us to control the pump through the samr21 board.

The function calls some GPIO functions needed to initialize the GPIO and sets it low because we assume that when we start the system, the plants do not need water.

Function `make_pump_close()`

This function closes the pump, when the algorithm calls this function had already checked that the plants have enough water and the pump is still on or the opening time has been exceeded.

To close the pump the function sets the GPIO to low using the GPIO function gpio_clear.

Function `make_pump_open(int aux)`

This function opens the pump, when the algorithm calls this functions had already checked that the plants need water and the pump is closed. The pump is opened for a concrete amount of time defined by the variable `aux` and then calls the function `make_pump_close()` to close the pump.

To open the pump the function sets the GPIO to high using the GPIO function `gpio_set`.

Function `water_level_sensor_control (int data)`

This function is used to avoid the malfunctioning of the pump caused by the lack of water. If the data sent by the water sensor indicates that the water level is under a concrete threshold the function calls the function `make_pump_close`.

Function `add_data_table(int id, int data)`

This function add to the table the data received from the sensors. First of all the function checks if there is existing data of this sensor already in the table, if they exist the function updates the value, if not it add a new line to the table with the new data and the actual time; finally the function calls `print_table` to show the new state of the table.

Function `add_pid_controller(int data)`

This function is used to adjust the time that the pump is open, to determine this time a set of algorithms are used that take into account the values received from the other sensors and depending on the distance between them and with the thresholds, the opening time of the pump is defined.

Function `add_set_data(int id,int data)`

This function decides if the pump should open or not, in order to do that it compares the values sent from the sensors with a set of thresholds and decides to call `make_pump_open`, `make_pump_close` or wait for more values of more sensors, it also calls the function that controls the water level sensor if it is necessary.

Function `shell_pump_set_data( int argc, char * argv[])`

This function (registered as a shell hook) is used to control the correct functioning of the pump controller algorithm and allows us to call the functions without sending real data from the sensors but sending it from a console.

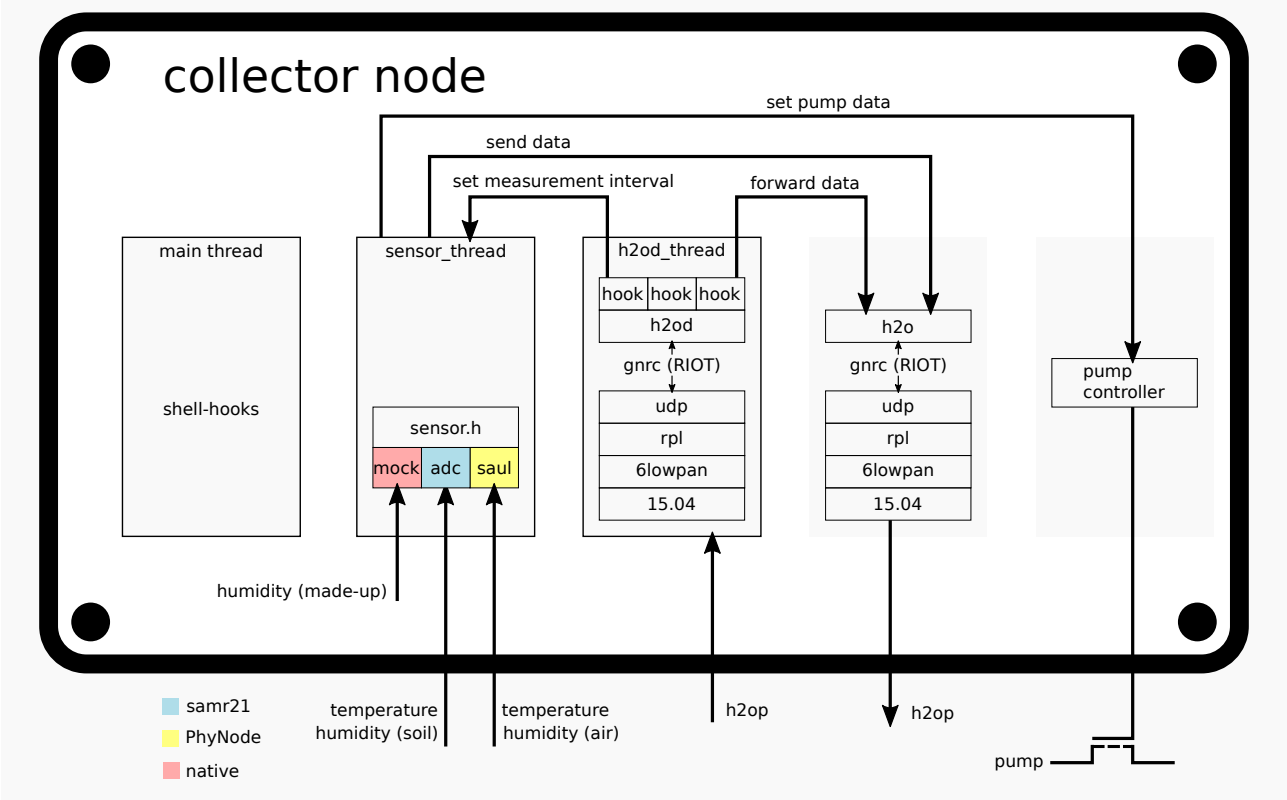

Figure 3: Implementation of the pump control node node. It shares the sensor integration and implemented network stack with the sensor nodes but extends them to control the pump and forward all incoming data to the gateway.

Gateway

The watering system prototype functions autonomously without the intrude of human or any 3rd Party service, but it is a good feature to monitor the system, because with processing the data an error can be predicted and fatal consequences can be avoided with taking the right measures. Among other control functions in the system also monitoring requires remote communications. Connecting with the sensors through Internet can be achieved using a gateway that forwards and transfer all packets from the sensors to some ip address outside the internal network of the sensors.

Serving as a gateway the node needs not to be placed outdoor nor besides the sensors, so it doesn't have to operate on battery, this means that using a high-end programmer as the gateway for our use-case is possible, because unlike the sensors, we need only one host for the gateway in the real scenario. For this reason the Raspberry Pi would perform a good job in our project, it has a powerful 32-bit arm processor with RISC architecture, it will communicate with others sensors through the 6LoWPAN interface, well written documentations of using 6LoWPAN module with Raspberry Pi are being provided by the RIOT community, the only downside in using any arm programmer like the pi is that RIOT can't be used for it, because kernel in RIOT was not ported for the ARM processors. This would break the rule of unifying the system on all nodes of the IoT project, which would be helpful for avoiding redundant costs in the future.

Following the instructions included in RIOT git [3] enables having a functioning 6LoWPAN transceiver on the Raspberry pi without any problems. We began programming the pi from scratch using an erased SD card.

In briefly our setup has included the following:

1. Setup Raspian OS on the SD card, this is a UNIX-like operating system optimised for the Raspberry pi.
2. SSH (Secured Shell Host openBSD) is a program for the communications between computers on different networks using the shell layer, it is installed by default in Raspian but using it requires enabling flags in the configurations of OS.
3. Setup the 6LoWPAN module and connect it to the raspberry pi, it worked without the need to reconfigure the kernel of Raspian.
4. Setup IPv6 for the 6LoWPAN interface and make sure the communications are working by pinging the rest of network.

After debugging and making sure that the communications on the physical layer are working properly and that the rest of network is communicating with the pi per 6LoWPAN, we setup a static IPv6 for the interfaces, so that whenever Raspian boots it uses that address.

There are additional configurations that should be done to the network layer in order to encapsulate the IPv6 packets and forward them to an IP address outside the watering system's network. Ip route provides management tools for manipulating any of the routing tables, it can be used to forward the packets directly with working only on the network layer, it is mentioned in the documentations of RIOT for setting up the 6LoWPAN and raspberry pi, but we could not use this because we also wanted to translate them from h2o protocol on UDP to HTTP on TCP, therefore we have written a python program running in an endless loop to capture all the incoming packets to the raspberry pi, interpret and convert them to the required protocol (HTTP), and finally forwards the packets to a predefined (Webserver) address. The only function for the gateway at this time is monitoring the incoming humidity and temperature data and finally presenting them on a web server, the running python program performs this one specific translation of protocol.

Data presentation

The sensors send periodically new data about humidity and temperature, the values need to be saved in a central database and presented to the user in proper way for reading. One way for reading the data from the sensors remotely is to have a running service providing access to the readers and the writers which are in our case the user (web browser) and the watering system (Sensor). An available option is using a web server running with a static ip address, and an apache service to handle the incoming requests/responses of the clients.

For testing the project we configured a private apache2 web server using ssh communication (as with raspberry pi). We decided to use python add-on called django to handle the HTTP requests, the apache module named mod_wsgi embeds Python applications within the server and allow them to communicate through the Python WSGI interface. Django is a free and open source web application framework, written in Python. A web framework is a set of components that helps you manage a website faster and easier. Django provides easy mechanisms to handle html templates. Moreover processing and reading predefined html files using django is easier in compare to the traditional web languages e.g "PHP", because as a webframework django provides the complete structure for building a website.

We implemented an HTTP API for reading and writing the humidity and temperature values of the sensors.

Functions in our HTTP API include get and set sensor data.

Write data to database:
`http://IP.ADDRESS:PORT:/set_sensor_data` .

Read data from database (Json):
`http://IP.ADDRESS:PORT:/get_sensor_data` .

Finally the data of the sensors is presented in an HTML table designed with bootstrap css, the table rows include data presenting in sqllite database, we used django template language for loading html page dynamically.

In our web server we didn't have to reach the complete RESTful implementation because our interface consist only of two functions, however both of the functions for setting and getting the data use HTTP GET requests.

Conclusion

In all cases accomplishing an IoT project for a specific purpose requires certain rules to follow and a set of communication protocols to implement, so that in the end to we can get a working network of devices working together autonomously on one assignment. Most of the components and protocols we used in our project which aren't specific to watering plants case can be directly used in other different projects, and in the future they will help making our final products reaching the IoT standards. Water is a highly conductive liquid, and in our case it was a challenges to prevent it from ruining the electronics, this concludes that in IoT a developer might also face environmental problems, but with using the right instruments to harvest enough energy from the surrounding to provide power to the system, we could have used the environment to our side.

References

[1] 6LoWPAN Linux Kernel on Raspberry Pi and gateway examples. https://github.com/RIOT-OS/RIOT/wiki/How-to-install-6LoWPAN-Linux-Kernel-on-Raspberry-Pi. Accessed: 2018-08.

[2] Phynode description in the riot documentation. https://github.com/RIOT-OS/RIOT/wiki/Board%3A-Phytec-phyWAVE-KW22. Accessed: 2018-08.

[3] RIOT ADC interface documentation. http://riot-os.org/api/group__drivers__periph__adc.html. Accessed: 2018-08.

[4] Sam r21 xplained pro description in the riot documentation. https://github.com/RIOT-OS/RIOT/wiki/Board%3A-SAMR21-xpro. Accessed: 2018-08.

[5] Sen0114 description. https://www.dfrobot.com/product-599.html. Accessed: 2018-08.

[6] watr.li blog, sensing moisture. https://github.com/watr-li/blog/blob/master/_posts/2015-03-06-Sensing-moisture.md. Accessed: 2018-08.

[7] R. Hinden and B. Haberman. Unique local ipv6 unicast addresses. RFC 4193, October 2005. https://tools.ietf.org/html/rfc4193.