A Modular System for Environment Monitoring of Plants

Bram den Ouden - 4472993
Maarten de Jong - 4290933

November 2020

***Abstract*—This paper discusses the approach towards monitoring the environment in which plants grow, addressing various aspects such as soil moisture, humidity, air temperature, and any other sensors that can provide valuable information. The approach is based around the use of a Raspberry Pi as a central server which communicates with ESP32 or ESP8266 modules that function as relay for the sensor data retrieved from attached sensors. Using a progressive web application allows for displaying all relevant information in a manner that is both desktop and mobile friendly. The end result is a solution which is modular, user-friendly, and requires only basic technical knowledge to use.**

I. Introduction

This paper describes a modular system capable of monitoring the growing environment of plants. Possible applications would be greenhouses, garden stores or even large private/public gardens. The system consists of a user interface, a central controller, and various different sensor modules that can be added on demand. This results in a scalable system that allows the user to add and (re)move sensors depending on their needs.

Figure 1 shows the flow of data within the system. To follow the sensor data, this figure will be traversed from bottom to top where two consecutive layers follow a many-to-one relationship. In this order, the layers respectively represent:

1) Multiple sensors connect to a single sensor nodes
2) Multiple sensor nodes connect to a single node controller
3) Multiple node controllers connect to a single user interface: the PWA

The information entered by the user will follow this tree from top to bottom: information enters via the PWA and is filtered and extended up to sensor nodes

The system will be elaborated based on the three layers described above:

1) The sensor nodes which collect and transmit the data from multiple sensors to the node controller will be discussed in section II.
2) The node controller which manages all nodes, calculates results from raw values and relays these values to the database behind the progressive web application will be discussed in section III.
3) The progressive web application which serves as a user interface will be discussed in section IV.

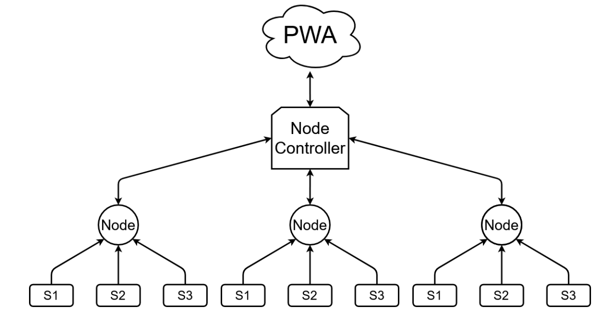

Fig. 1: Connection tree between the progressive web app (PWA), Node controller, Nodes, and Sensors (Sx)

Furthermore, section V will elaborate on getting the system up and running in different scenarios. Finally section VI will discuss points of improvement and limitations.

II. Sensor node

A sensor node is defined as a single device tasked with collecting data from physically attached sensors and transmitting this data to a node controller discussed in section III.

The hardware of a node consists of either an ESP32- or ESP8266-based board and a power source. The aim is to keep nodes physically as small as possible so they can easily be deployed throughout the area they will be monitoring.

A. Firmware

To exchange information, nodes connect to the controller using a websocket protocol. This lightweight protocol allows fast and reliable bidirectional communication, keeping power usage to a minimum. In order to exchange data, the following steps are executed in order.

1) Connect to node controller
2) Read values from the attached sensors
3) Transmit the collected data
4) Verify the current configuration and update if applicable
5) Receive sleep time
6) Close connection
7) Sleep for received amount of time

The firmware includes custom written libraries to allow dynamic addition and removal of sensors. This dynamic configuration allows hardware to change and have each node function differently without having to re-flash the firmware to the nodes.

How this configuration is created will be discussed in section III. From the node's perspective, a JSON is received containing a unique ID, type and the required pins for each sensor as well as a version string, see Listing 1.

Listing 1: JSON config for both an analog and an I2C sensor

```
{
  "config-version": "12345678",
  "config":
  [
    {
      "link-id": 8,
      "type": "analog",
      "pins": [0]
    },
    {
      "link-id": 26,
      "type": "am232x",
      "pins": [22,21,0]
    }
  ]
}
```

Each node can support up to 16 sensors at a time. This limitation is based on available dynamic memory of the nodes. To increase the number of sensors supported tailor-made firmware could be written for specific applications. Currently supported sensors are:

- Any analog sensor such as light intensity, soil moisture, and some temperature sensors. Up to 8 analog sensors may be connected to a single sensor node.
- AM232x type sensors. These are accurate temperature and humidity sensors. Due to their fixed I2C address, only a single type of these sensors may be attached to each node.
- DHTxx type sensors. These sensors measure temperature and humidity but use a one-wire communication protocol allowing up to 16 of them to be attached to a single sensor node.

B. Enclosure

The enclosure of a sensor node contains the circuit board (PCB) and battery as can be seen in Figure 2. It protects these components from damage caused by collision damage and environmental effect such as moisture. The only part exposed to the environment is the pins of the PCB where sensors are attached. To avoid water from getting into the enclosure via these holes a rubber or silicon seal can be used along the pins. The flat pin of the enclosure can be used to stick the node into the soil in an upright position.

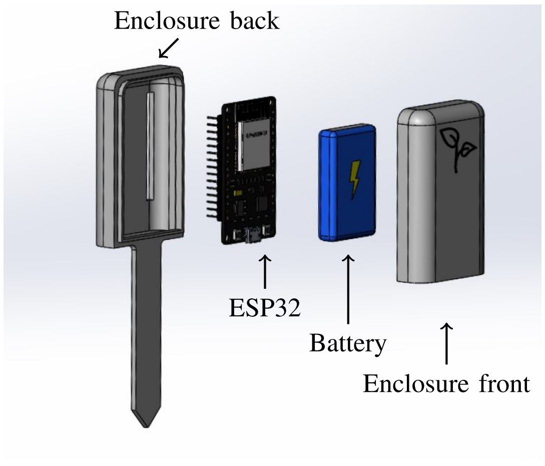

Fig. 2: Exploded view of the enclosure

III. Node controller

Each web needs a spider. In the web of nodes in this system the node controller is the spider. A node controller is tasked with:

- Configuring each node
- Collecting data from each node
- Schedule the reconnect time of each node
- Upload data to the PWA
- Notify the PWA of new nodes
- Receive the user's configuration from the PWA

A. Node connection

Figure 1 shows the tree structure of this project and shows the central role of the node controller: Sensor data is collected, filtered and transmitted to the PWA. The information entered by the user will follow this tree in reverse order but is also filtered at the node controller. Since not all nodes have the same sensors attached to them, the controller keeps track of each node individually. This is done by applying an object-oriented approach where each controller holds multiple node objects and each node object refers to multiple sensor objects. Each sensor object is allocated to a single node object according to the configuration set by the user.

Sensor objects contain the raw value, calculated value, boundaries for both raw and calculated value, the type of the sensor, and the physical pins of the node to which the sensor is connected.

Node objects contain a list of attached sensor objects, the configuration version, firmware version, unique ID of the ESP chip and timestamps of when the object was created and last updated.

When a node connects to the node controller and the connection is established, it will immediately transmit the data it has collected. An example of such a message can be seen in Listing 2.

Due to the asynchronous approach of the websocket server, many nodes can connect to a single node controller at the same time.

Listing 2: JSON data as tranmsitted by the node to the controller. The keys 8 and 26 represent the unique id of the sensor-node combinations as received from the PWA.

```
{
  "chipID": 915947,
  "version": "V1 Oct 7 2020 16:47:51",
  "config_version": "12345678",
  "8": 1130,
  "26": [605, 249]
}
```

Looking at the examples of Listing 1 and Listing 2, their config_version match; there is no need for the controller to transmit a new configuration. The controller will transmit the interval in milliseconds before the node should attempt to reconnect again. When this message is acknowledged by the node, the controller will indicate it has no further information to offer and the client will close the connection.

When the node config_version of the node does not match the configuration at the controller, the node object at the controller will return a new configuration based on the properties of its sensor objects.

B. *PWA connection*

When the node controller starts running, it will attempt to connect to the PWA to retrieve a list of configured nodes. Each known node which connects and does not yet have the right configuration will be provided with a correct configuration following the process described above.

If the connection to the PWA fails or there are no known nodes, the node controller will start to build its own list of known nodes as the nodes connect to it. Each time a new node connects to the node controller, it will try to update the PWA to allow the user to create a configuration.

After a configurable amount of time the controller will send an update to the PWA containing all nodes and their associated sensor data. If, for any reason, the data cannot be transferred to the PWA, a backlog file will be created. This backlog file contains all the data that would otherwise be transmitted to the PWA including a timestamp. When the connection is confirmed to be working by means of a successful update, the entire backlog will be transmitted to the PWA. The local copies of the backlog files will be deleted upon success.

IV. Progressive web app

The choice for creating a progressive web app as opposed to a native (mobile) application is made because the flexibility of the progressive web app as well as the multi-platform compatibility is better in the case of the PWA. This means that fast design iterations can be made and deployed across web and mobile platform. Using this method it is possible to create both the API layer as well as the application layer within one application, and on the same webserver. This choice for using the same webserver also fits within the decision making process as it allows for running the API and website on the Raspberry Pi which also functions as the node controller. This allows all data to be stored in a central place. However, this also allows the PWA to be on a remote server, as the node controller can communicate with the webserver wherever it is, as it doesn't rely on bidirectional communication.

A. *Framework*

The framework used to make this project is a custom PHP-based framework that defines interactions with the MariaDB database as well as creating url's for specific pages of different types, such as a regular webpage, an API page, or a cronjob. It is based on the Model-View-Controller (MVC) principle, which splits database code (model), from front-end code (view), and the logic code (controller). The latter is only used to trigger specific events that need to be updated on a regular basis, such as updating whether one of the sensors has reached a critical threshold set by the user. The difference in the various page type is that they require different permissions; the cronjob page requires no permissions as it only triggers background tasks, the API page requires the login information of the API user which is created in the setup process, and the regular webpage is available for everyone, but has different levels of permissions built into the webpage.

B. *API Implementation*

To implement the communication between the controller and the database multiple API endpoints have been configured to accept the different requests that the controller needs. These endpoints are:

- /api/get_devices (Retrieve all known nodes)
- /api/new_device (Register a new node)
- /api/update (Update the sensor data for any number of nodes provided as JSON)
- /api/update_triggers (Check whether any of the threshold values have been exceeded)

These four main endpoints allow for all necessary interaction to happen between the controller and the database. By ensuring that communication happens in one direction it means that it is not necessary to be able to communicate with the controller. This is difficult because the controller will almost always be in a closed network in someone's home, or an office network. This also means that the API layer needs to be able to handle some issues by itself, such as what to do when an update is received for a node that does not exist in the database. In this case the API adds that node ID to the database, and shows this in the interface so the user can configure it. From then on the node will correctly register all its data. Using this methodology it is possible to create a feeling of easy communication between the different layers, ensuring a more pleasant experience for the user.

C. *Front-end*

The front-end makes use of the Bootstrap 4 framework to easily create appealing graphical interfaces for the user to interact with. Each page consists of the header and footer, and in between different content can be loaded. For the API pages only the content layer is loaded, as the header and footer are not necessary in an API response. This setup allows for easy adaptation of pages while keeping the general theme the same. Generally, the theme is quite simple with a dark header bar containing all tabs the user has access to (only the login screen when not logged in, but more upon logging in with an admin account).

Upon logging in the user is greeted by the weather forecast as well as the latest sensor data, giving a quick overview of the most critical information. From there the user can move to other features such as linking new modules to nodes, setting specific triggers, or viewing a graphical interpretation of sensor location and sensor data values. These pages are designed for use by the administrator of the product, where it is easily possibly to implement more user roles and user management in the future to hide specific pages from lower user-levels. In the current version this is not the focus, and as such has not been implemented in use yet, though the database and page controllers are designed with such a modification in mind.

D. *Usage*

Usage of the PWA is based around providing the right amount of necessary interactions for a user to configure their sensors and nodes, and providing the right amount of information following that. Currently, choices are made such as that the sensor graph shows 10-minute moving averages for the past two hours, as this will provide the initial overview for a user. For further use-cases it is desirable to have a graph with an adjustable time-span to view how specific sensor values have developed over time allowing for long-term adjustments to be made.

When linking modules to the nodes, a user can currently connect as many modules as they would like, which provides a measure of flexibility. This also relies on users have a slight understanding of the technical implementation of the product they are working with, as it asks for the pins that the sensors are connected to, and provides some (less relevant) information such as raw minimum and maximum. The understanding of the real minimum and maximum values is interesting to have, as this provides information on whether the sensor is suitable for the use-case the user has in mind.

The page used to create triggers is also kept simple to provide only the necessary information, and not over complicate the task for the user. Here, a user chooses a sensor, whether it should be triggered when the value is greater- or less-than a specified value, and an email address to be notified on. In the future this can be expanded with personalized push notifications, tying triggers to user-accounts and their respective devices. This allows for further user access control and determining responsibilities in, for example, a greenhouse with multiple employees. Here, different employees could receive notifications for sensors in different sections, or depending on time of day and when which employee is working. Currently, the triggers are globally shared, and send an email notification using a specified SMTP server and email address. As mentioned, these triggers are updated by a cronjob task which checks if the sensor value of a specific trigger has passed its threshold and subsequently sends an email using the PHPMailer library.

V. Deployment

Each of the three discussed parts of the system has to be deployed separately. Two possible use cases that will be discussed are local and distributed deployment. In both cases the steps provided in this section are required, albeit on physically different machines.

Deploy locally: the node controller and web app are run on the same network. This is the case when the user does not necessarily want to access the web interface from outside their local network. This situation is most likely for a corporate configuration, where security is more important than the ability to access the data from everywhere. Using corporate VPN networks it can then become possible to still access the web interface.

Deploy distributed: the node controller and web app are run on separate machines who are not necessarily connected to the same network. In this case the web interface and database can be hosted on a remote server, for example a webhost or VPS. This is a likely configuration for slightly more advanced users who want to be able to view their sensor data from anywhere in the world.

A. *Deploy PWA*

1) Clone the repository
2) Install docker-compose and composer
3) From the directory run `docker-compose up`. This might take a few minutes the first time.
4) Direct to the pwa directory and run `./migrate` to initialise the database (N.B. On Windows machines this command will be `python migrate`)
5) Done! You should now be able to browse to you localhost or ip address of the machine you performed these steps on to be greeted by the welcome screen!

B. *Deploy node controller*

1) Clone the repository
2) Enter the credentials obtained from the PWA setup into `Server\objects\api.py`
3) From the *server* directory, run websocketserver.py
4) Done! The terminal will show information about the current state of the node controller.

C. *Deploy nodes*

Due to the current version of the core ESP32 firmware, hostnames such as *raspberrypi.local* are not yet supported. The ESP8266 has a newer core firmware which does support this functionality. When using an ESP32 it is therefore required to give the node controller a fixed IP address and update the *SERVER_ADDR* variable in the websocketclient sketch accordingly.

1) Power-on the node
2) Connect to a WiFi network called *NODE_xxxxxx* where *xxxxxx* represents a unique id for the node
3) Visit 192.168.4.1 in your browser and follow the steps to connect your node to the same WiFi network as your node controller
4) Done!

VI. Future work

Much of the future work of this application is based around expanding upon the existing functionalities and creating more 'comfort'-features. Such features include improving user management, so that an administrator is able to create new accounts and assign what features which account can view. This would mostly benefit the situations in which companies are using the solution, as multi-user setups and access control is a less necessary feature for backyard use. Therefore, this is not yet prioritized, though all initial steps are in place to implement this change.

Another aspect upon which can be improved in future work is that the setup process is not fully automated at the current stage. Preferably someone is able to pull the latest release from the GitHub repository and run a single command to be up and running. This is mainly an improvement to be made when the solution is to ship to end-users for the first time, as the user-friendliness is currently lacking in that aspect.

Example of such an improvement lies with the ESP32. Due to the current version of the linklayer library used by its core firmware, it cannot connect to local domains. The ES8266 features a later version of this library and can therefore do this natively. It is expected that this update will soon come to the ESP32 as well. This will allow for the firmware to connect to domain names like modfarm.local instead of the local IP address, which makes deployment easier.

The firmware currently reconfigures itself by means of adding supported sensor objects to a predefined array. Although this is very reliable and makes debugging easy, it's not very memory efficient. The websocket protocol supports transmitting binary files which could be used to transmit completely new firmware versions. This feature can be used for over the air (OTA) firmware updates. Besides OTA updates, this feature could also be used to provide each node with firmware specific to their attached sensors. This firmware could be automatically generated by the node controller which would keep the benefit of real-time reconfiguration of nodes whilst limiting the amount of memory required.

A final part of the future work is a 2D layout of multiple sensors showing the same sensor type. The example of Figure 3 shows the current version. Improving this feature would allow large-scale gardeners to easily detect where certain measures should be taken.

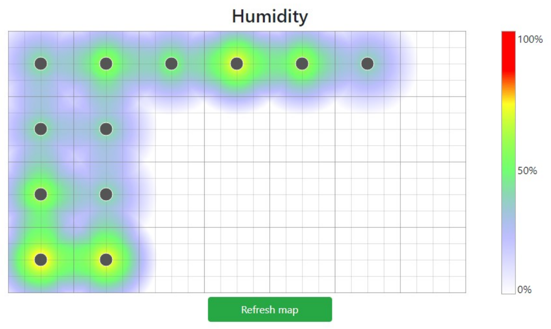

Fig. 3: Map showing different measurements in a 2d grid

A. *Limitations*

The current version of the system does have a few limitations. The list below will show them in a [topic]:[issue] style

- ESP32: only the ADC channels connected to ADC 1 can be used while WiFi is enabled.
- ESP32: linklayer library version does not yet support *.local domain names so local IP address must be entered for now
- PWA: triggers cannot be set for multi-sensors yet